ES6+ Features

Classes
Classes are a new feature in ES6, used to describe the blueprint of an object and make EcmaScript's prototypical inheritance model function more like a traditional class-based language.

Object : An object is an instance of a class which is created using the new operator.

When using a dot notation to access a method on the object, this will refer to the object to the left of the dot.

Arrow Functions : ES6 offers some new syntax for dealing with this : "arrow functions".

Template Strings: ES6 introduces a new type of string literal that is marked with back ticks (`). These string literals *can* include newlines, and there is a string interpolation for inserting variables into strings:

```
var name = 'Sam';
var age = 42;
console.log(`hello my name is ${name}, and I am ${age} years old`);
```

Constants and Block Scoped Variables:

ES6 introduces the concept of block scoping. Block scoping will be familiar to programmers from other languages like C, Java, or even PHP. In ES5 JavaScript and earlier, var s are scoped to function s, and they can "see" outside their functions to the outer context.
const and let use { and } blocks as containers, hence "block scope". Block scoping is most useful during loops.

Spread Syntax (Spread Element) and Rest parameters :

A Spread syntax allows in-place expansion of an expression for the following cases:

1. Array
2. Function call
3. Multiple variable destructuring

Rest parameters works in the opposite direction of the spread syntax, it collects an indefinite number of comma separated expressions into an array.

Destructuring:

Destructuring is a way to quickly extract data out of an {} or [] without having to write much code.

```
let foo = ['one', 'two', 'three'];
let [one, two, three] = foo;
console.log(one); // 'one'
```

```
let myModule = {
  drawSquare: function drawSquare(length) { /* implementation */ },
  drawCircle: function drawCircle(radius) { /* implementation */ },
  drawText: function drawText(text) { /* implementation */ },
};
let { drawSquare, drawText } = myModule;
drawSquare(5);
drawText('hello');
```

ES6 Modules

ES6 introduced *module* support. A module in ES6 is single file that allows code and data to be isolated, it helps in organizing and grouping code logically. In other languages it's called a package or library.

All code and data inside the module has file scope, what this means is they are not accessible from code outside the module. To share code or data outside a module, it needs to be exported using the **export** keyword.

```
export const pi = 3.141592;
export const circumference = diameter => diameter * pi;
```

TypeScript

ES6 is the current version of JavaScript. TypeScript is a superset of ES6, which means all ES6 features are part of TypeScript, but not all TypeScript features are part of ES6.

Consequently, TypeScript must be transpiled into ES5 to run in most browsers. One of TypeScript's primary features is the addition of type information, hence the name. This type information can help make JavaScript programs more predictable and easier to reason about.

Types let developers write more explicit "contracts". In other words, things like function signatures are more explicit.

Install the TypeScript transpiler using npm:

```
npm install -g typescript
```

Then use `tsc` to manually compile a TypeScript source file into ES5:

```
tsc test.ts
node test.js
```

TypeScript Features:

Now that producing JavaScript from TypeScript code has been de-mystified, some of its features can be described and experimented with.

- Types
- Interfaces
- Shapes
- Decorators

Types

Many people do not realize it, but JavaScript *does* in fact have types, they're just "duck typed", which roughly means that the programmer does not have to think about them.

JavaScript's types also exist in TypeScript:

- boolean (true/false)
- number integers, floats, Infinity and NaN
- string characters and strings of characters
- [] Arrays of other types, like number[] or boolean[]
- {} Object literal
- undefined not set

TypeScript also adds

- enum enumerations like { Red, Blue, Green }
- any use any type
- void nothing

TypeScript Classes

TypeScript also treats class es as their own type

Interfaces

An *interface* is a TypeScript artifact, it is not part of ECMAScript. An *interface* is a way to define a *contract* on a function with respect to the arguments and their type. Along with functions, an *interface* can also be used with a Class as well to define custom types.

An interface is an abstract type, it does not contain any code as a *class* does. It only defines the 'signature' or shape of an API. During transpilation, an interface will not generate any code, it is only used by Typescript for type checking during development.

Shapes

Underneath TypeScript is JavaScript, and underneath JavaScript is typically a JIT (Just-In-Time compiler). Given JavaScript's underlying semantics, types are typically reasoned about by "shapes". These underlying shapes work like TypeScript's interfaces, and are in fact how TypeScript compares custom types like `class` es and `interface` s.

Decorators

Decorators are functions that are invoked with a prefixed @ symbol, and *immediately* followed by a `class` , parameter, method or property. The decorator function is supplied information about the `class` , parameter or method, and the decorator function returns something in its place, or manipulates its target in some way. Typically the "something" a decorator returns is the same thing that was passed in, but it has been augmented in some way.

Decorators are functions, and there are four things (`class` , parameter, method and property) that can be decorated;

Command Line JavaScript: NodeJS

Node.js is a JavaScript runtime environment that allows JavaScript code to run outside of a browser using Google V8 JavaScript engine. Node.js is used for writting fast executing code on the server to handle events and non-blocking I/O efficently.

- REPL (Read-Eval-Print-Loop) to quickly write and test JavaScript code.
- The V8 JavaScript interpreter.
- Modules for doing OS tasks like file I/O, HTTP, etc.

While Node.js was initially intended for writing server code in JavaScript, today it is widely used by JavaScript tools, which makes it relevant to front-end programmers too.

Back-End Code Sharing and Distribution: npm

`npm` is the "node package manager". It installs with NodeJS, and gives you access to a wide variety of 3rd-party JavaScript modules.

It also performs dependency management for your back-end application. You specify module dependencies in a file called `package.json` ; running `npm install` will resolve, download and install your back-end application's dependencies.

Module Loading, Bundling and Build Tasks: Webpack

Webpack is a JavaScript module bundler. It takes modules with their dependencies and generates static assets representing those modules. Webpack known only how to bundle JavaScript. To bundle other assets likes CSS, HTML, images or just about anything it uses additional loaders. Webpack can also be extended via plugins, for example minification and mangling can be done using the UglifyJS plugin for webpack.

Bootstrapping an Angular Application

Bootstrapping is an essential process in Angular - it is where the application is loaded when Angular comes to life.

Understanding the File Structure

To get started let's create a bare-bones Angular application with a single component. To do this we need the following files:

- *app/app.component.ts* - this is where we define our root component
- *app/app.module.ts* - the entry Angular Module to be bootstrapped
- *index.html* - this is the page the component will be rendered in
- *app/main.ts* - is the glue that combines the component and page together

app/app.component.ts

```
import { Component } from '@angular/core'
@Component({
  selector: 'app-root',
  template: '<b>Bootstrapping an Angular Application</b>'
})
export class AppComponent { }
```

index.html

```
<body>
      <app-root>Loading...</app-root>
</body>
```

app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } '@angular/core';
import { AppComponent } from './app.component'
@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

app/main.ts

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

The bootstrap process loads *main.ts* which is the main entry point of the application. The `AppModule` operates as the root module of our application. The module is configured to use `AppComponent` as the component to bootstrap, and will be rendered on any `app-root` HTML element encountered.

There is an `app` HTML element in the *index.html* file, and we use *app/main.ts* to import the `AppModule` component and the `platformBrowserDynamic().bootstrapModule` function and kickstart the process.

Components in Angular

The core concept of any Angular application is the *component*. In effect, the whole application can be modeled as a tree of these components.

This is how the Angular team defines a component:

A component controls a patch of screen real estate that we could call a view, and declares reusable UI building blocks for an application.

Basically, a component is anything that is visible to the end user and which can be reused many times within an application.

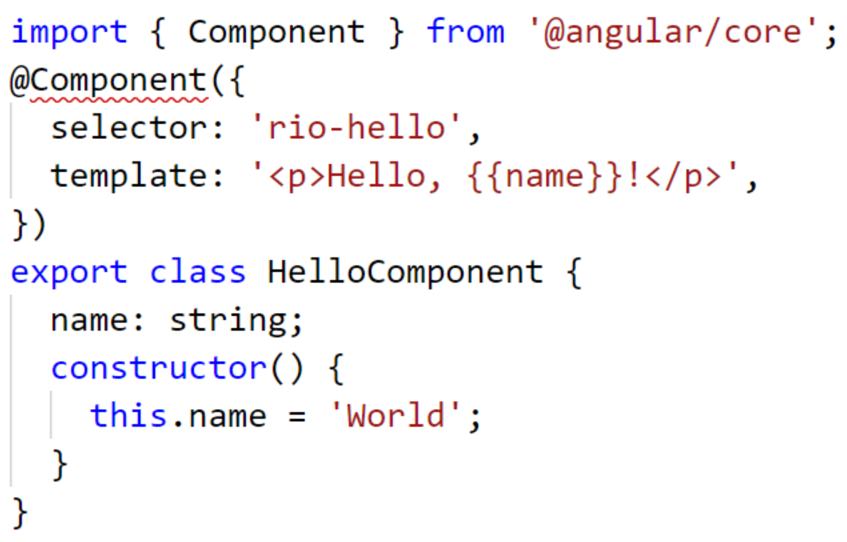
Creating Components

Components in Angular 2 build upon the lessons learned from Angular 1.5. We define a component's application logic inside a class. To this we attach `Component` , a TypeScript `decorator` , which allows you to modify a class or function definition and adds metadata to properties and function arguments.

- *selector* is the element property that we use to tell Angular to create and insert an instance of this component.

- *template* is a form of HTML that tells Angular what needs to be to rendered in the DOM.

The Component below will interpolate the value of name variable into the template between the double braces {{name}} , what get rendered in the view is <p>Hello World</p>



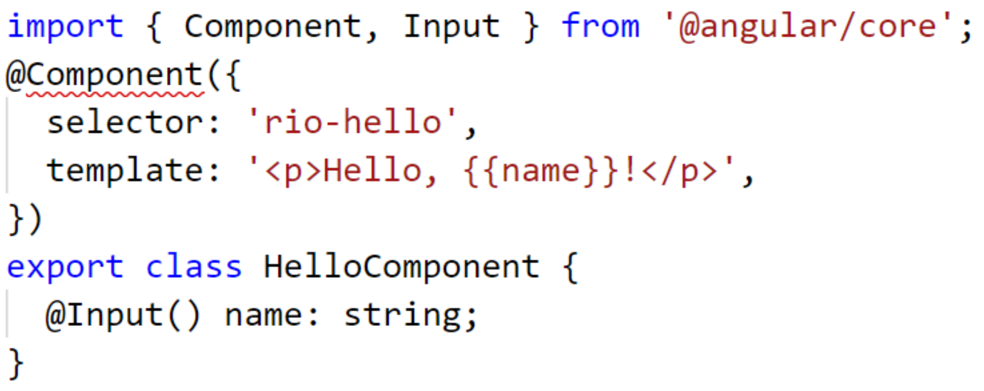
```
import { Component } from '@angular/core';
@Component({
  selector: 'rio-hello',
  template: '<p>Hello, {{name}}!</p>',
})
export class HelloComponent {
  name: string;
  constructor() {
    this.name = 'World';
  }
}
```

We need to import the Component decarator from @angular/core before we can make use of it. To use this component we simply add <rio-hello></rio-hello> to the HTML file or another template, and Angular will insert an instance of the HelloComponent view between those tags.

Passing Data into a Component

There are two ways to pass data into a component, with 'property binding' and 'event binding'. In Angular, data and event change detection happens top-down from parent to children. However for Angular events we can use the DOM event mental model where events flow bottom-up from child to parent. So, Angular events can be treated like regular HTML DOM based events when it comes to cancellable event propagation.

The @Input() decorator defines a set of parameters that can be passed down from the component's parent. For example, we can modify the HelloComponent component so that name can be provided by the parent.

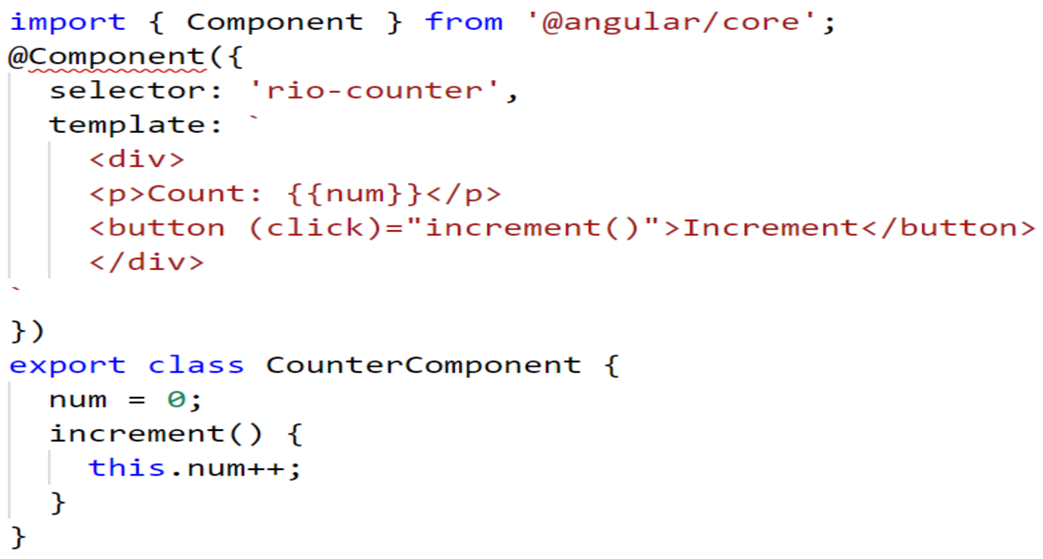


```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'rio-hello',
  template: '<p>Hello, {{name}}!</p>',
})
export class HelloComponent {
  @Input() name: string;
}
```

The point of making components is not only encapsulation, but also reusability. Inputs allow us to configure a particular instance of a component.

Responding to Component Events

An event handler is specified inside the template using round brackets to denote event binding. This event handler is then coded in the class to process the event.



```
import { Component } from '@angular/core';
@Component({
  selector: 'rio-counter',
  template: `
    <div>
    <p>Count: {{num}}</p>
    <button (click)="increment()">Increment</button>
    </div>
`
})
export class CounterComponent {
  num = 0;
  increment() {
    this.num++;
  }
}
```

Using Two-Way Data Binding

Two-way data binding combines the input and output binding into a single notation using the ngModel directive.

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

What this is doing behind the scenes is equivalent to:

```
<input [ngModel]="name" (ngModelChange)="name=$event">
```

To create your own component that supports two-way binding, you must define an @Output property to match an @Input , but suffix it with the Change

Projection

Projection is a very important concept in Angular. It enables developers to build reusable components and make applications more scalable and flexible

Structuring Applications with Components

As the complexity and size of our application grows, we want to divide responsibilities among our components further.

- *Smart / Container components* are application-specific, higher-level, container components, with access to the application's domain model.
- *Dumb / Presentational components* are components responsible for UI rendering and/or behavior of specific entities passed in via components API (i.e component properties and events). Those components are more in-line with the upcoming Web Component standards.

Directives

A Directive modifies the DOM to change apperance, behavior or layout of DOM elements.

Directives are one of the core building blocks Angular uses to build applications. In fact, Angular components are in large part directives with templates.

From an Angular 1.x perspective, Angular 2 components have assumed a lot of the roles directives used to. The majority of issues that involve templates and dependency injection rules will be done through components, and issues that involve modifying generic behaviour is done through directives.

There are three main types of directives in Angular:

- **Component** - directive with a template.
- ***Attribute*** *directives* - directives that change the behavior of a component or element but don't affect the template
- ***Structural*** *directives* - directives that change the behavior of a component or element by affecting how the template is rendered

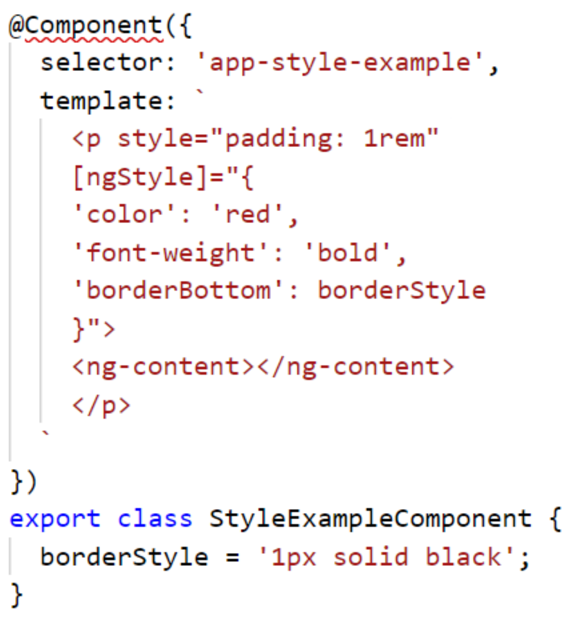
Attribute Directives

Attribute directives are a way of changing the appearance or behavior of a component or a native DOM element. Ideally, a directive should work in a way that is component agnostic and not bound to implementation details.

For example, Angular has built-in attribute directives such as `ngClass` and `ngStyle` that work on any component or element.

NgStyle Directive

Angular provides a built-in directive, `ngStyle` , to modify a component or element's `style` attribute. Here's an example:



```
@Component({
  selector: 'app-style-example',
  template: `
    <p style="padding: 1rem"
    [ngStyle]="{
    'color': 'red',
    'font-weight': 'bold',
    'borderBottom': borderStyle
    }">
    <ng-content></ng-content>
    </p>
  `
})
export class StyleExampleComponent {
  borderStyle = '1px solid black';
}
```

NgClass Directive

The ngClass directive changes the class attribute that is bound to the component or element it's attached to. There are a few different ways of using the directive.

```
@Component({
selector: 'app-class-as-object',
template: `
<p [ngClass]="{ card: true, dark: false, flat: flat }">
<ng-content></ng-content>
<br>
<button type="button" (click)="flat=!flat">Toggle Flat</button>
</p>
`,
styles: [`
.card {
border: 1px solid #eee;
padding: 1rem;
margin: 0.4rem;
font-family: sans-serif;
box-shadow: 2px 2px 2px #888888;
}
.dark {
background-color: #444;
border-color: #000;
color: #fff;
}
.flat {
box-shadow: none;
}
`]
})
export class ClassAsObjectComponent {
flat: boolean = true;
}
```

Here we can see that since the object's card and flat properties are true, those classes are applied but since dark is false, it's not applied.

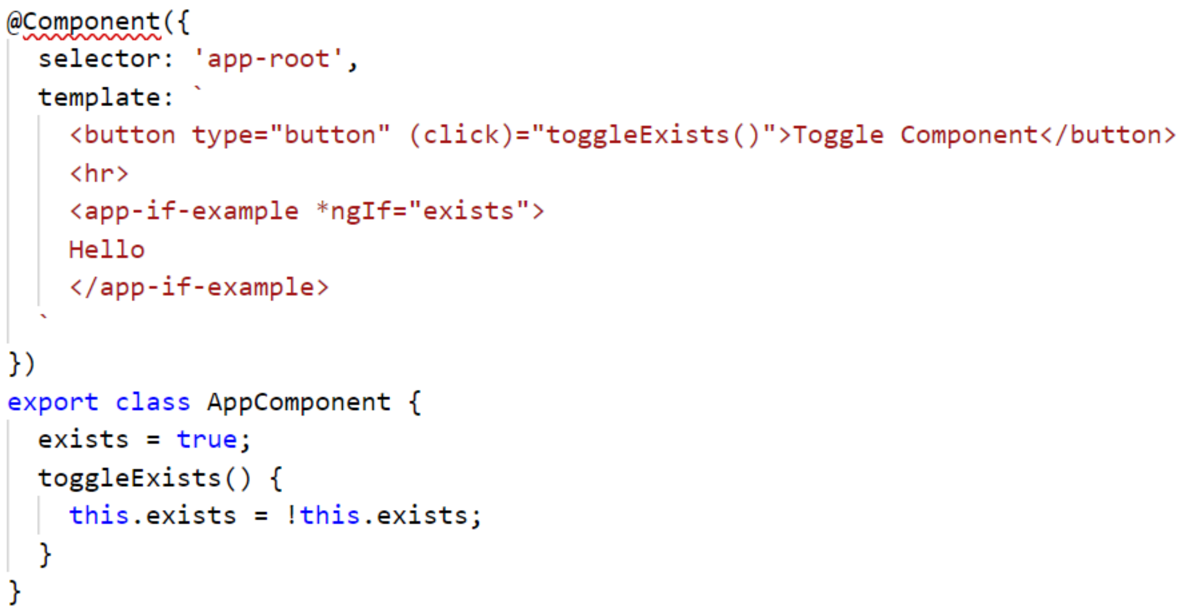
Structural Directives

Structural Directives are a way of handling how a component or element renders through the use of the template tag. This allows us to run some code that decides what the final rendered output will be. Angular has a few built-in structural directives such as ngIf , ngFor , and ngSwitch .

NgIf Directive

The ngIf directive conditionally adds or removes content from the DOM based on whether or not an expression is true or false.

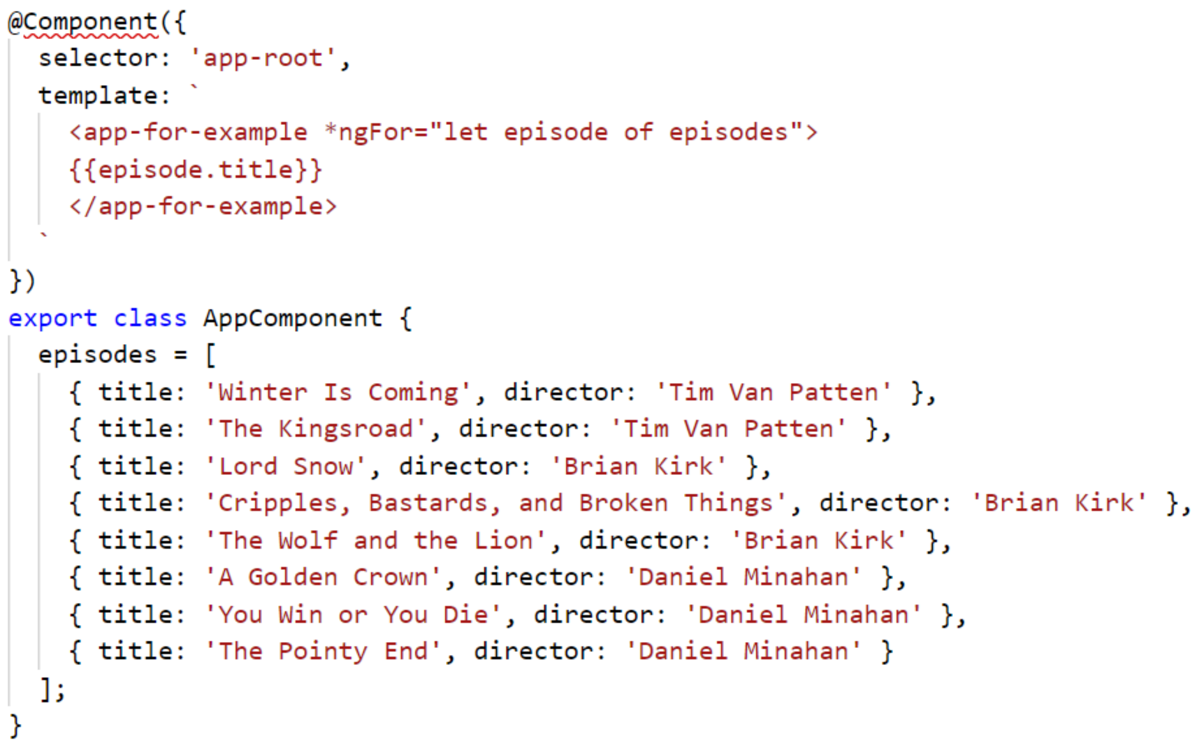
Here's our app component, where we bind the ngIf directive to an example component.



```
@Component({
  selector: 'app-root',
  template: `
    <button type="button" (click)="toggleExists()">Toggle Component</button>
    <hr>
    <app-if-example *ngIf="exists">
    Hello
    </app-if-example>
  `
})
export class AppComponent {
  exists = true;
  toggleExists() {
    this.exists = !this.exists;
  }
}
```

NgFor Directive

The NgFor directive is a way of repeating a template by using each item of an iterable as that template's context.



```
@Component({
  selector: 'app-root',
  template: `
    <app-for-example *ngFor="let episode of episodes">
    {{episode.title}}
    </app-for-example>
  `
})
export class AppComponent {
  episodes = [
    { title: 'Winter Is Coming', director: 'Tim Van Patten' },
    { title: 'The Kingsroad', director: 'Tim Van Patten' },
    { title: 'Lord Snow', director: 'Brian Kirk' },
    { title: 'Cripples, Bastards, and Broken Things', director: 'Brian Kirk' },
    { title: 'The Wolf and the Lion', director: 'Brian Kirk' },
    { title: 'A Golden Crown', director: 'Daniel Minahan' },
    { title: 'You Win or You Die', director: 'Daniel Minahan' },
    { title: 'The Pointy End', director: 'Daniel Minahan' }
  ];
}
```

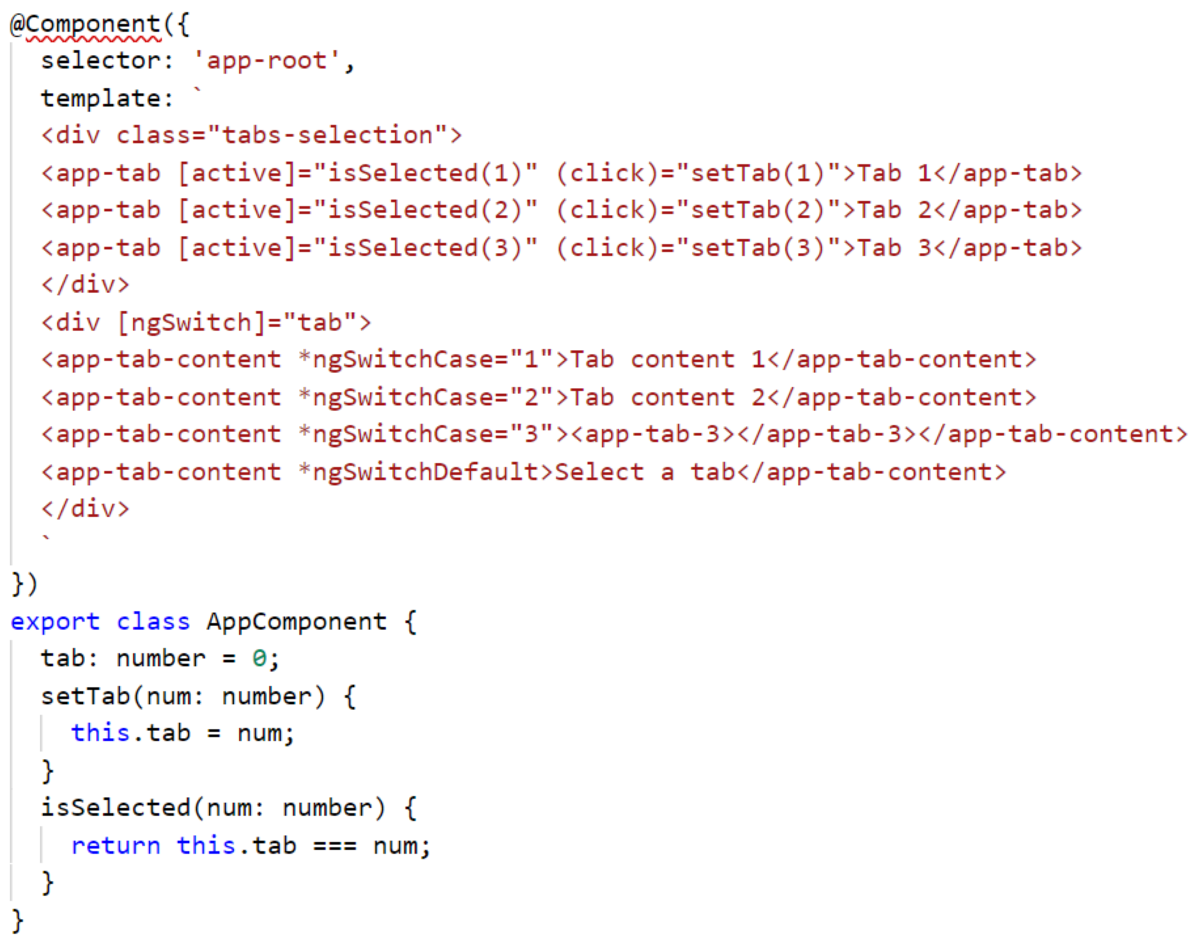
Local Variables

NgFor also provides other values that can be aliased to local variables:

- *index* - position of the current item in the iterable starting at 0
- *first* - true if the current item is the first item in the iterable
- *last* - true if the current item is the last item in the iterable
- *even* - true if the current index is an even number
- *odd* - true if the current index is an odd number

NgSwitch Directives

ngSwitch is actually comprised of two directives, an attribute directive and a structural directive. It's very similar to a switch statement in JavaScript and other programming languages, but in the template.



```
@Component({
  selector: 'app-root',
  template: `
  <div class="tabs-selection">
  <app-tab [active]="isSelected(1)" (click)="setTab(1)">Tab 1</app-tab>
  <app-tab [active]="isSelected(2)" (click)="setTab(2)">Tab 2</app-tab>
  <app-tab [active]="isSelected(3)" (click)="setTab(3)">Tab 3</app-tab>
  </div>
  <div [ngSwitch]="tab">
  <app-tab-content *ngSwitchCase="1">Tab content 1</app-tab-content>
  <app-tab-content *ngSwitchCase="2">Tab content 2</app-tab-content>
  <app-tab-content *ngSwitchCase="3"><app-tab-3></app-tab-3></app-tab-content>
  <app-tab-content *ngSwitchDefault>Select a tab</app-tab-content>
  </div>
  `
})
export class AppComponent {
  tab: number = 0;
  setTab(num: number) {
    this.tab = num;
  }
  isSelected(num: number) {
    return this.tab === num;
  }
}
```

Component Lifecycle

A component has a lifecycle managed by Angular itself. Angular manages creation, rendering, data-bound properties etc. It also offers hooks that allow us to respond to key lifecycle events.

Here is the complete lifecycle hook interface inventory:

- `ngOnChanges` - called when an input binding value changes
- `ngOnInit` - after the first `ngOnChanges`
- `ngDoCheck` - after every run of change detection
- `ngAfterContentInit` - after component content initialized
- `ngAfterContentChecked` - after every check of component content
- `ngAfterViewInit` - after component's view(s) are initialized
- `ngAfterViewChecked` - after every check of a component's view(s)
- `ngOnDestroy` - just before the component is destroyed

View Encapsulation

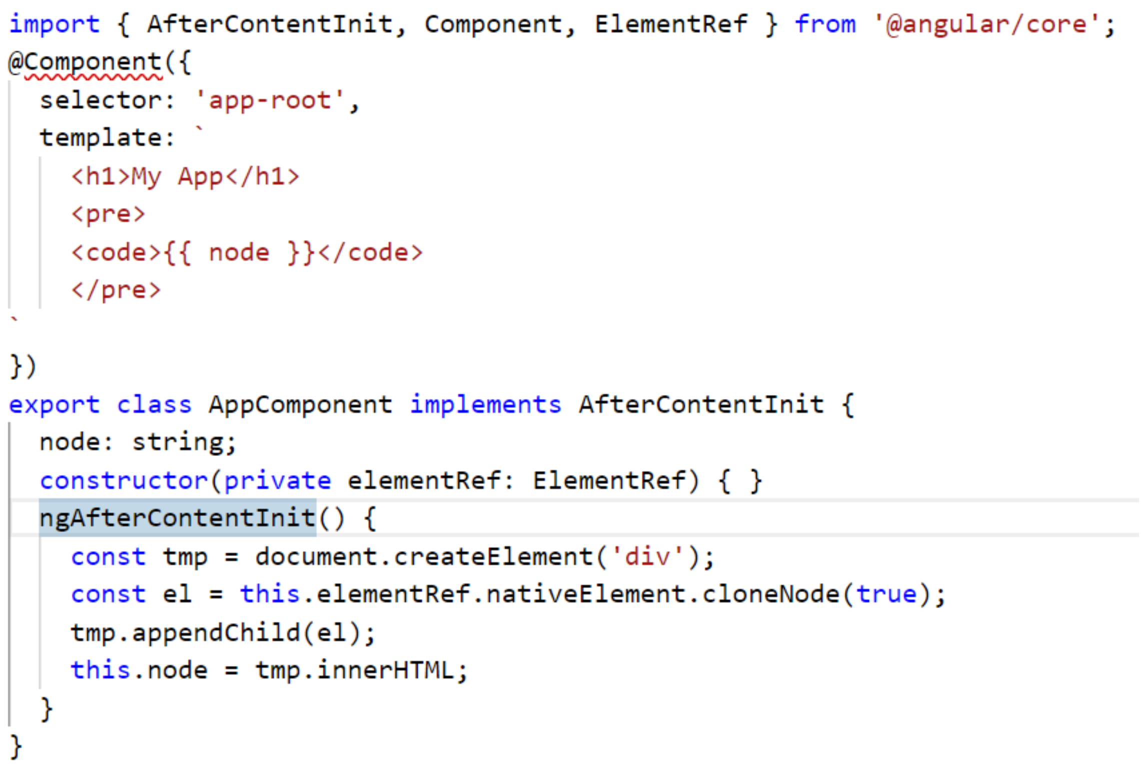
View encapsulation defines whether the template and styles defined within the component can affect the whole application or vice versa. Angular provides three encapsulation strategies:

- Emulated (default) - styles from main HTML propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.
- Native - styles from main HTML do not propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.
- None - styles from the component propagate back to the main HTML and therefore are visible to all components on the page. Be careful with apps that have None and

Native components in the application. All components with None encapsulation will have their styles duplicated in all components with Native encapsulation.

ElementRef

Provides access to the underlying native element (DOM element).



```
import { AfterContentInit, Component, ElementRef } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h1>My App</h1>
    <pre>
    <code>{{ node }}</code>
    </pre>
`

})
export class AppComponent implements AfterContentInit {
  node: string;
  constructor(private elementRef: ElementRef) { }
  ngAfterContentInit() {
    const tmp = document.createElement('div');
    const el = this.elementRef.nativeElement.cloneNode(true);
    tmp.appendChild(el);
    this.node = tmp.innerHTML;
  }
}
```

Observables

An exciting new feature used with Angular is the `Observable` . This isn't an Angular specific feature, but rather a proposed standard for managing async data that will be included in the release of ES7. Observables open up a continuous channel of communication in which multiple values of data can be emitted over time. From this we get a pattern of dealing with data by using array-like operations to parse, modify and maintain data. Angular uses observables extensively

Angular's DI

Angular's DI system is (mostly) controlled through `@NgModule` . Specifically the `providers` and `declarations` array. (`declarations` is where we put components, pipes and directives; `providers` is where we put services)

The Injector Tree

Angular injectors (generally) return singletons. That is, in the previous example, all components in the application will receive the same random number. In Angular 1.x there was only one injector, and all services were singletons. Angular overcomes this limitation by using a tree of injectors.

In Angular there is not just one injector per application, there is *at least* one injector per application. Injectors are organized in a tree that parallels Angular's component tree.

HTTP

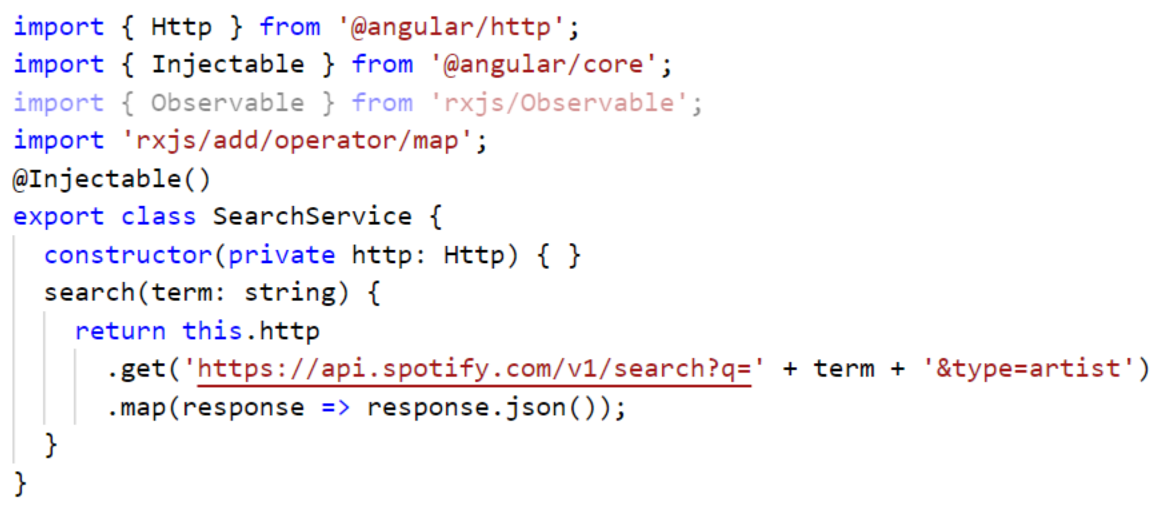
In order to start making HTTP calls from our Angular app we need to import the angular/http module and register for HTTP services. It supports both XHR and JSONP requests exposed through the HttpModule and JsonpModule respectively. In this section we will be focusing only on the HttpModule .

Setting up angular/http

In order to use the various HTTP services we need to include HttpModule in the imports for the root NgModule . This will allow us to access HTTP services from anywhere in the application.

Making HTTP Requests

To make HTTP requests we will use the Http service. In this example we are creating a SearchService to interact with the Spotify API.



```
import { Http } from '@angular/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/map';
@Injectable()
export class SearchService {
  constructor(private http: Http) { }
  search(term: string) {
    return this.http
      .get('https://api.spotify.com/v1/search?q=' + term + '&type=artist')
      .map(response => response.json());
  }
}
```

Here we are making an HTTP GET request which is exposed to us as an observable. You will notice the .map operator chained to .get . The Http service provides us with the raw response as a string. In order to consume the fetched data we have to convert it to JSON.

In addition to `Http.get()` , there are also `Http.post()` , `Http.put()` , `Http.delete()` , etc. They all return observables.

Change Detection

Change detection is the process that allows Angular to keep our views in sync with our models.

Change detection has changed in a big way between the old version of Angular and the new one. In Angular 1, the framework kept a long list of watchers (one for every property bound to our templates) that needed to be checked every-time a digest cycle was started. This was called *dirty checking* and it was the only change detection mechanism available.

In Angular, **the flow of information is unidirectional**, even when using `ngModel` to implement two way data binding, which is only syntactic sugar on top of the unidirectional flow. In this new version of the framework, our code is responsible for updating the models.

Angular is only responsible for reflecting those changes in the components and the DOM by means of the selected change detection strategy.

Change Detector Classes

At runtime, Angular will create special classes that are called *change detectors*, one for every component that we have defined. In this case, Angular will create two classes: `AppComponent` and `AppComponent_ChangeDetector` .

The goal of the change detectors is to know which model properties used in the template of a component have changed since the last time the change detection process ran.

In order to know that, Angular creates an instance of the appropriate change detector class and a link to the component that it's supposed to check.

Change Detection Strategy: Default

By default, Angular defines a certain change detection strategy for every component in our application. To make this definition explicit, we can use the property `changeDetection` of the `@Component` decorator.

Change Detection Strategy: OnPush

To inform Angular that we are going to comply with the conditions mentioned before to improve performance,

Zones

Zone.js provides a mechanism, called zones, for encapsulating and intercepting asynchronous activities in the browser (e.g. `setTimeout` , , promises).

These zones are *execution contexts* that allow Angular to track the start and completion of asynchronous activities and perform tasks as required (e.g. change detection). Zone.js provides a global zone that can be forked and extended to further encapsulate/isolate asynchronous behaviour, which Angular does so in its **NgZone** service, by creating a fork and extending it with its own behaviours.

The **NgZone** service provides us with a number of Observables and methods for determining the state of Angular's zone and to execute code in different ways inside and outside Angular's zone.

In The Zone

NgZone exposes a set of Observables that allow us to determine the current status, or *stability*, of Angular's zone.

- *onUnstable* – Notifies when code has entered and is executing within the Angular zone.
- *onMicrotaskEmpty* - Notifies when no more microtasks are queued for execution. *Angular subscribes to this internally to signal that it should run change detection.*
- *onStable* – Notifies when the last `onMicroTaskEmpty` has run, implying that all tasks have completed and change detection has occurred.
- *onError* – Notifies when an error has occurred. *Angular subscribes to this internally to send uncaught errors to its own error handler, i.e. the errors you see in your console prefixed with 'EXCEPTION:'.*

Change Detection

Since all asynchronous code executed from within Angular's zone can trigger change detection you may prefer to execute some code outside of Angular's zone when change detection is not required.

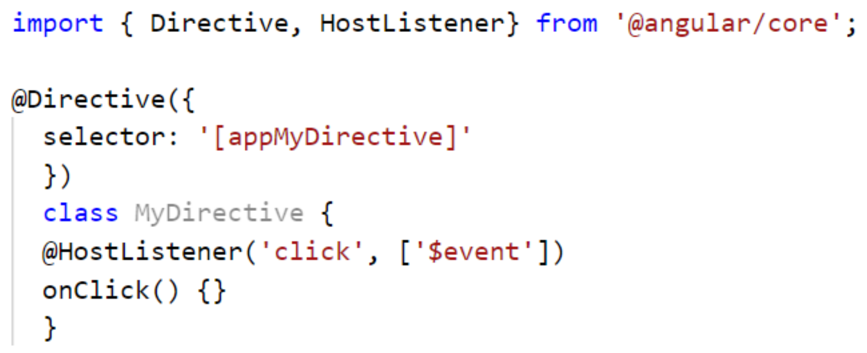
To run code outside of Angular's context, **NgZone** provides a method aptly named **runOutsideAngular**. Using this method, Angular's zone will not interact with your code and will not receive events when the global zone becomes stable.

Angular Directives

Angular built-in directives cover a broad range of functionality, but sometimes creating our own directives will result in more elegant solutions.

Listening to an Element Host

Listening to the *host* - that is, the DOM element the directive is attached to - is among the primary ways directives extend the component or element's behavior. Previously, we saw its common use case.



```
import { Directive, HostListener} from '@angular/core';

@Directive({
  selector: '[appMyDirective]'
  })
  class MyDirective {
  @HostListener('click', ['$event'])
  onClick() {}
  }
```

Setting Properties with a Directive

We can use attribute directives to affect the value of properties on the host node by using the @HostBinding decorator.

The @HostBinding decorator allows us to programatically set a property value on the directive's host element. It works similarly to a property binding defined in a template, except it specifically targets the host element. The binding is checked for every change detection cycle, so it can change dynamically if desired.

AoT in Angular

Every Angular application requires a compilation process before they can run in the browser: the enriched components and templates provided by Angular cannot be understood by the browser directly. During the compilation, Angular's compiler also improves the app run-time performance by taking JavaScript VM's feature (like inline caching) into consideration.

The initial compiler in Angular 1.x and Angular 2 is called JiT (Just-in-Time) compiler. As for AoT, it stands for the Ahead-of-Time compiler that was recently introduced in Angular. Compared to the JiT compilation performed by Angular at run-time, AoT provides a smaller bundle with faster rendering in the browser.

The gist of AoT is moving the compilation from run-time to the building process. That means, first we can remove the JiT compiler (which is around 523kb) from the bundle to have a smaller build, and second, the browser can execute the code without waiting for JiT in the run-time which leads to a faster rendering speed.

Early compilation also means that developers can find template bugs without actually running the code and before it reaches to client. This provides a more robust application with higher security because less client-side HTML and JavaScript are `eval` ed. Also, by introducing compiled code in the building process, AoT makes the application more treeshakable and open to various other optimizations. Bundlers like Rollup and Google Closure can take that advantage and effectively decrease the bundle size.

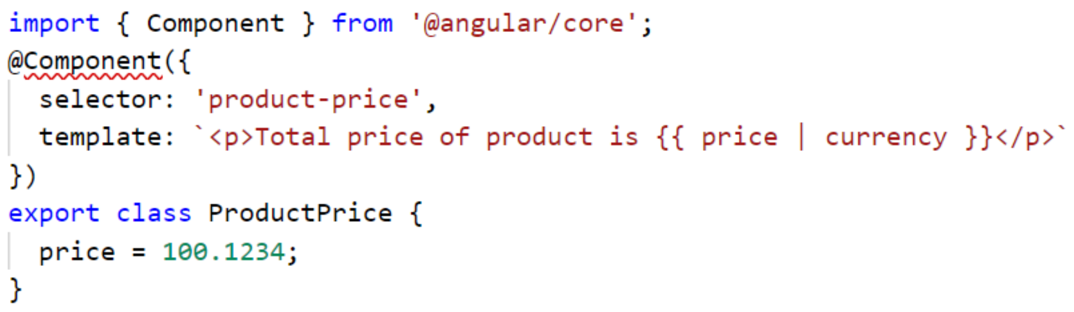
Besides, AoT compiler also inlines HTML templates and CSS files and help reduce the amount of asynchronous requests sent by the application.

Pipes

Angular 2 provides a new way of filtering data: `pipes` . Pipes are a replacement for Angular 1.x's `filters` . Most of the built-in filters from Angular 1.x have been converted to Angular 2 pipes; a few other handy ones have been included as well.

Using Pipes

Like a filter, a pipe also takes data as input and transforms it to the desired output. A basic example of using pipes is shown below:



```
import { Component } from '@angular/core';
@Component({
  selector: 'product-price',
  template: `<p>Total price of product is {{ price | currency }}</p>`
})
export class ProductPrice {
  price = 100.1234;
}
```

Passing Parameters

A pipe can accept optional parameters to modify the output. To pass parameters to a pipe, simply add a colon and the parameter value to the end of the pipe expression

Chaining Pipes

We can chain pipes together to make use of multiple pipes in one expression.

Custom Pipes

Angular allows you to create your own custom pipes:

Each custom pipe implementation must:

- have the `@Pipe` decorator with pipe metadata that has a `name` property. This value will be used to call this pipe in template expressions. It must be a valid JavaScript identifier.

- implement the PipeTransform interface's transform method. This method takes the value being piped and a variable number of arguments of any type and return a transformed ("piped") value.

Each colon-delimited parameter in the template maps to one method argument in the same order.

Stateful Pipes

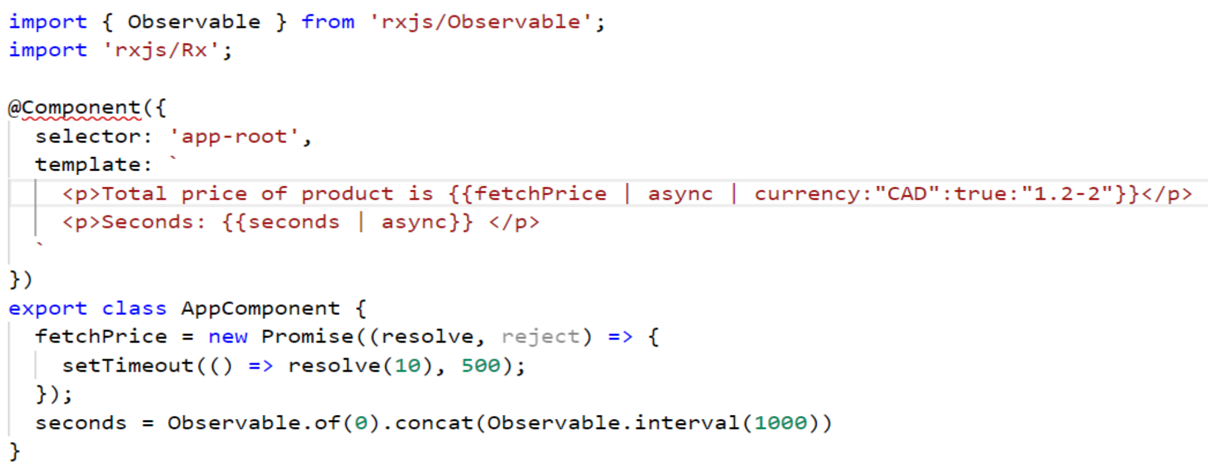
There are two categories of pipes:

- *Stateless* pipes are pure functions that flow input data through without remembering anything or causing detectable side-effects. Most pipes are stateless. The CurrencyPipe we used and the length pipe we created are examples of a stateless pipe.
- *Stateful* pipes are those which can manage the state of the data they transform. A pipe that creates an HTTP request, stores the response and displays the output, is a stateful pipe. Stateful Pipes should be used cautiously.

Angular provides AsyncPipe , which is stateful.

AsyncPipe

AsyncPipe can receive a Promise or Observable as input and subscribe to the input automatically, eventually returning the emitted value(s). It is stateful because the pipe maintains a subscription to the input and its returned values depend on that subscription.



```
import { Observable } from 'rxjs/Observable';
import 'rxjs/Rx';

@Component({
  selector: 'app-root',
  template: `
    <p>Total price of product is {{fetchPrice | async | currency:"CAD":true:"1.2-2"}}</p>
    <p>Seconds: {{seconds | async}} </p>
  `
})
export class AppComponent {
  fetchPrice = new Promise((resolve, reject) => {
    setTimeout(() => resolve(10), 500);
  });
  seconds = Observable.of(0).concat(Observable.interval(1000))
}
```

Forms

An application without user input is just a page. Capturing input from the user is the cornerstone of any application. In many cases, this means dealing with forms and all of their complexities.
Angular is much more flexible than Angular 1.x for handling forms — we are no longer restricted to relying solely on `ngModel` . Instead, we are given degrees of simplicity and power, depending on the form's purpose.

- Template-Driven Forms places most of the form handling logic within that form's template
- Reactive Forms places form handling logic within a component's class properties and provides interaction through observables

Template-Driven Forms
The most straightforward approach to building forms in Angular is to take advantage of the directives provided for you.

Reactive/Model-Driven Forms
While using directives in our templates gives us the power of rapid prototyping without too much boilerplate, we are restricted in what we can do. Reactive forms on the other hand, lets us define our form through code and gives us much more flexibility and control over data validation. There is a little bit of magic in its simplicity at first, but after you're comfortable with the basics, learning its building blocks will allow you to handle more complex use cases.

Reactive Forms Basics
To begin, we must first ensure we are working with the right directives and the right classes in order to take advantage of procedural forms. For this, we need to ensure that the `ReactiveFormsModule` was imported in the bootstrap phase of the application module. This will give us access to components, directives and providers like `FormBuilder` , `FormGroup` , and `FormControl`

Modules

Angular Modules provides a mechanism for creating blocks of functionality that can be combined to build an application.

What is an Angular Module?

In Angular, a module is a mechanism to group components, directives, pipes and services that are related, in such a way that can be combined with other modules to create an application. An Angular application can be thought of as a puzzle where each piece (or each module) is needed to be able to see the full picture.

Another analogy to understand Angular modules is classes. In a class, we can define public or private methods. The public methods are the API that other parts of our code can use to interact with it while the private methods are implementation details that are hidden. In the same way, a module can export or hide components, directives, pipes and services. The exported elements are meant to be used by other modules, while the ones that are not exported (hidden) are just used inside the module itself and cannot be directly accessed by other modules of our application.

A Basic Use of Modules

To be able to define modules we have to use the decorator NgModule .

```
import { NgModule } from '@angular/core';
@NgModule({
imports: [ ... ],
declarations: [ ... ],
bootstrap: [ ... ]
})
export class AppModule { }
```

In the example above, we have turned the class AppModule into an Angular module just by using the NgModule decorator. The NgModule decorator requires at least three properties: imports , declarations and bootstrap .

Creating a Feature Module

When our root module start growing, it starts to be evident that some elements (components, directives, etc.) are related in a way that almost feel like they belong to a library that can be "plugged in".

```
.
├── app
│   ├── app.component.ts
│   └── app.module.ts
├── credit-card
│   ├── credit-card-mask.pipe.ts
│   ├── credit-card.component.ts
│   ├── credit-card.module.ts
│   └── credit-card.service.ts
├── index.html
└── main.ts
```

Notice how each folder has its own module file: *app.module.ts* and *credit-card.module.ts*.

Lazy Loading a Module

Another advantage of using modules to group related pieces of functionality of our application is the ability to load those pieces on demand. Lazy loading modules helps us decrease the startup time. With lazy loading our application does not need to load everything at once, it only needs to load what the user expects to see when the app first loads. Modules that are lazily loaded will only be loaded when the user navigates to their routes.

But more importantly, we can see that whenever we try to go to the path `lazy` , we are going to lazy load a module conveniently called `LazyModule` . Look closely at the definition of that route:

```
{ path: 'lazy', loadChildren: 'lazy/lazy.module#LazyModule' }
```

There's a few important things to notice here:

1. We use the property `loadChildren` instead of `component` .
2. We pass a string instead of a symbol to avoid loading the module eagerly.
3. We define not only the path to the module but the name of the class as well.

Routing

Why Routing?

Routing allows us to express some aspects of the application's state in the URL. Unlike with server-side front-end solutions, this is optional - we can build the full application without ever changing the URL. Adding routing, however, allows the user to go straight into certain aspects of the application. This is very convenient as it can keep your application linkable and bookmarkable and allow users to share links with others.

Routing allows you to:

- Maintain the state of the application
- Implement modular applications
- Implement the application based on the roles (certain roles have access to certain URLs)

Configuring Routes

Base URL Tag

The Base URL tag must be set within the `<head>` tag of index.html:

```
<base href="/">
```

In the demos we use a script tag to set the base tag. In a real application it must be set as above.

Route Definition Object

The `Routes` type is an array of routes that defines the routing for the application. This is where we can set up the expected paths, the components we want to use and what we want our application to understand them as.

Each route can have different attributes; some of the common attributes are:

- *path* - URL to be shown in the browser when application is on the specific route *component* - component to be rendered when the application is on the specific route

- *redirectTo* - redirect route if needed; each route can have either component or redirect attribute defined in the route (covered later in this chapter)
- *pathMatch* - optional property that defaults to 'prefix'; determines whether to match full URLs or just the beginning. When defining a route with empty path string set pathMatch to 'full', otherwise it will match all paths.
- *children* - array of route definitions objects representing the child routes of this route.

To use Routes , create an array of route configurations.

Below is the sample Routes array definition:

```
const routes: Routes = [
      { path: 'component-one', component: ComponentOne },
      { path: 'component-two', component: ComponentTwo }
];
```

RouterModule

RouterModule.forRoot takes the Routes array as an argument and returns a *configured* router module.

Redirecting the Router to Another Route

When your application starts, it navigates to the empty route by default. We can configure the router to redirect to a named route by default:

```
export const routes: Routes = [
      { path: '', redirectTo: 'component-one', pathMatch: 'full' },
      { path: 'component-one', component: ComponentOne },
      { path: 'component-two', component: ComponentTwo }
];
```

The pathMatch property, which is required for redirects, tells the router how it should match the URL provided in order to redirect to the specified route. Since pathMatch: full is provided, the router will redirect to component-one if the entire URL matches the empty path ('').

When starting the application, it will now automatically navigate to the route for componentone

Defining Links Between Routes : RouterLink

Add links to routes using the RouterLink directive.

For example the following code defines a link to the route at path component-one .

```
    <a routerLink="/component-one">Component One</a>
```

Navigating Programmatically

Alternatively, you can navigate to a route by calling the navigate function on the router:

```
    this.router.navigate(['/component-one']);
```

Using Route Parameters

Say we are creating an application that displays a product list. When the user clicks on a product in the list, we want to display a page showing the detailed information about that product. To do this you must: add a route parameter ID link the route to the parameter add the service that reads the parameter.

Declaring Route Parameters

The route for the component that displays the details for a specific product would need a route parameter for the ID of that product. We could implement this using the following Routes :

```
export const routes: Routes = [
      { path: '', redirectTo: 'product-list', pathMatch: 'full' },
      { path: 'product-list', component: ProductList },
      { path: 'product-details/:id', component: ProductDetails }
];
```

Note :id in the path of the product-details route, which places the parameter in the path.

For example, to see the product details page for product with ID 5, you must use the following URL: localhost:4200/product-details/5

Defining Child Routes

When some routes may only be accessible and viewed within other routes it may be appropriate to create them as child routes.
For example: The product details page may have a tabbed navigation section that shows the product overview by default. When the user clicks the "Technical Specs" tab the section shows the specs instead.

If the user clicks on the product with ID 3, we want to show the product details page with the overview:

localhost:4200/product-details/3/overview

When the user clicks "Technical Specs":

localhost:4200/product-details/3/specs

overview and specs are child routes of product-details/:id . They are only reachable within product details.

```
export const routes: Routes = [
      { path: '', redirectTo: 'product-list', pathMatch: 'full' },
      { path: 'product-list', component: ProductList },
      { path: 'product-details/:id', component: ProductDetails,
            children: [
            { path: '', redirectTo: 'overview', pathMatch: 'full' },
            { path: 'overview', component: Overview },
            { path: 'specs', component: Specs }
            ]
      }
];
```

Where would the components for these child routes be displayed? Just like we had a <router-outlet></router-outlet> for the root application component, we would have a router outlet inside the ProductDetails component. The components corresponding to the child routes of product-details would be placed in the router outlet in ProductDetails .

Registering the Route Guards with Routes

In order to use route guards, we must register them with the specific routes we want them to run for.

```
import { Routes, RouterModule } from '@angular/router';
import { AccountPage } from './account-page';
import { LoginRouteGuard } from './login-route-guard';
import { SaveFormsGuard } from './save-forms-guard';
const routes: Routes = [
      { path: 'home', component: HomePage },
      {
            path: 'accounts',
            component: AccountPage,
            canActivate: [LoginRouteGuard],
            canDeactivate: [SaveFormsGuard]
      }
];
export const appRoutingProviders: any[] = [];
export const routing = RouterModule.forRoot(routes);
```

Implementing CanActivate

Let's look at an example activate guard that checks whether the user is logged in:

```
import { CanActivate } from '@angular/router';
import { Injectable } from '@angular/core';
import { LoginService } from './login-service';
@Injectable()
export class LoginRouteGuard implements CanActivate {
      constructor(private loginService: LoginService) {}
      canActivate() {
            return this.loginService.isLoggedIn();
      }
}
```

This class implements the `CanActivate` interface by implementing the `canActivate` function.

When canActivate returns true, the user can activate the route. When canActivate returns false, the user cannot access the route. In the above example, we allow access when the user is logged in.

canActivate can also be used to notify the user that they can't access that part of the application, or redirect them to the login page.

Passing Query Parameters

Use the [queryParams] directive along with [routerLink] to pass query parameters. For example:

```
<a [routerLink]="['product-list']" [queryParams]="{ page: 99 }">
     Go to Page 99
</a>
```

Alternatively, we can navigate programmatically using the Router service:

```
goToPage(pageNum) {
     this.router.navigate(['/product-list'], { queryParams: { page:
     pageNum } });
}
```

Reading Query Parameters

Similar to reading route parameters, the Router service returns an Observable we can subscribe to read the query parameters:

Angular CLI

With all of the new features Angular takes advantage of, like static typing, decorators and ES6 module resolution, comes the added cost of setup and maintenance. Spending a lot of time with different build setups and configuring all of the different tools used to serve a modern JavaScript application can really take a lot of time and drain productivity by not being able to actually work on the app itself.

Seeing the popularity of ember-cli, Angular decided they would provide their own CLI to solve this problem. Angular CLI is geared to be the tool used to create and manage your Angular app. It provides the ability to:

- create a project from scratch
- scaffold components, directives, services, etc.
- lint your code
- serve the application
- run your unit tests and end to end tests.

Angular CLI Setup

Prerequisites

Angular CLI is currently only distributed through npm and requires Node version 4 or greater.

Installation

The Angular CLI can be installed with the following command:

```
npm install -g angular-cli
```

Creating a New App

Use the `ng new [app-name]` command to create a new app. This will generate a basic app in the folder of the app name provided. The app has all of the features available to work with the CLI commands. Creating an app may take a few minutes to complete since npm will need to install all of the dependencies. The directory is automatically set up as a new **git** repository as well. If git is not your version control of choice, simply remove the *.git* folder and *.gitignore* file.

Serving the App

The CLI provides the ability to serve the app with live reload. To serve an application, simply run the command `ng serve` . This will compile the app and copy all of the application specific files to the *dist* folder before serving.

By default, `ng serve` serves the application locally on port 4200(http://localhost:4200
) but this can be changed by using a command line argument: `ng serve --port=8080`

Creating Components

The CLI can scaffold Angular components through the `generate` command. To create a new component run:

```
ng generate component [component-name]
```

Executing the command creates a folder, [component-name], in the project's `src/app` path or the current path the command is executed in if it's a child folder of the project. The folder has the following:

`[component-name].component.ts` the component class file

`[component-name].component.css` for styling the component

`[component-name].component.html` component html

`[component-name].component.spec.ts` tests for the component

Creating Routes

The ng g route [route-name] command will spin up a new folder and route files for you. At the time of writing this feature was temporarily disabled due to ongoing changes happening with Angular routing.

Creating Other Things

The CLI can scaffold other Angular entities such as services, pipes and directives using the generate command.

```
    ng generate [entity] [entity-name]
```

This creates the entity at src/app/[entity-name].[entity].ts along with a spec file, or at the current path if the command is executed in a child folder of the project. The CLI provides blueprints for the following entities out of the box:

Item Command Files generated

- Component: ng g component [name] component, HTML, CSS, test spec files
- Directive: ng g directive [name] component, test spec files
- Pipe: ng g pipe [name] component, test spec files
- Service: ng g service [name] component, test spec files
- Class: ng g class [name] component, test spec files
- Route: ng g route [name] component, HTML, CSS, test spec files

Testing

Apps generated by the CLI integrate automated tests. The CLI does this by using the **Karma test runner.**

Unit Tests

To execute unit tests, run ng test . This will run all the tests that are matched by the Karma configuration file at config/karma.conf.js . It's set to match all TypeScript files that end in .spec.ts by default.

End-to-End Tests

End-to-end tests can be executed by running ng e2e . Before end-to-end tests can be performed, the application must be served at some address. Angular CLI uses protractor. It will attempt to access localhost:4200 by default; if another port is being used, you will have to update the configuration settings located at config/protractor.conf.js .

Linting

To encourage coding best practices Angular CLI provides built-in linting. By default the app will look at the project's tslint.json for configuration. Linting can be executed by running the command ng lint .

CLI Command Overview

One of the advantages of using the Angular CLI is that it automatically configures a number of useful tools that you can use right away. To get more details on the options for each task, use ng --help .

Testing

ng test triggers a build and then runs the unit tests set up for your app using Karma. Use the --watch option to rebuild and retest the app automatically whenever source files change.

Build

ng build will build your app (and minify your code) and place it into the default output path, dist/ .

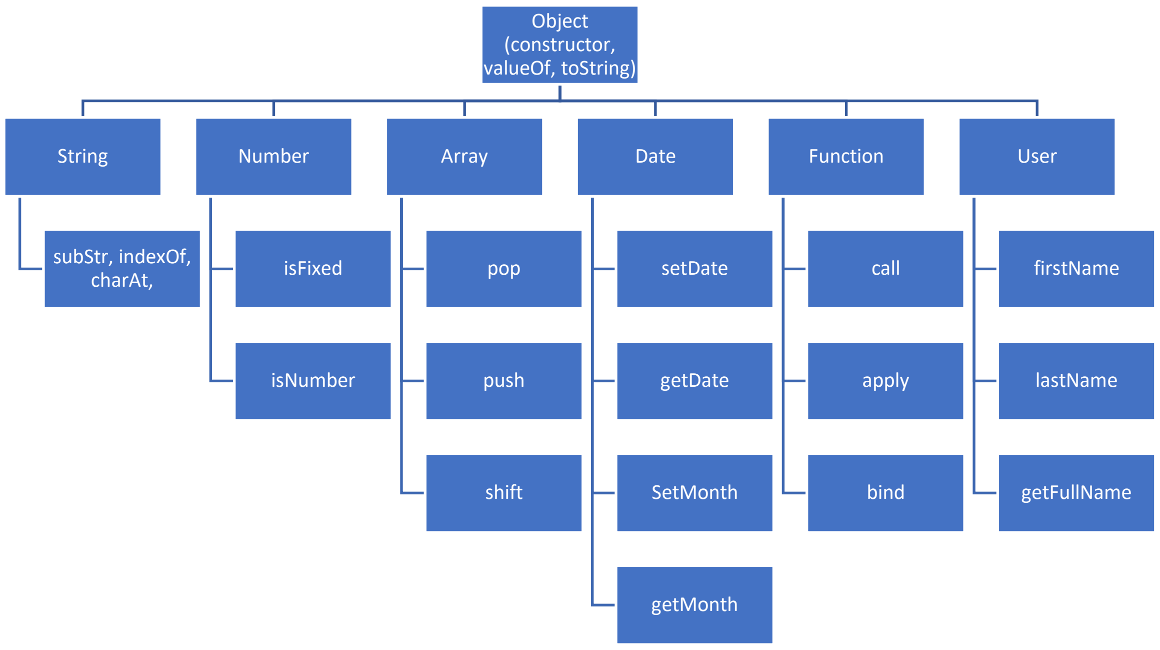
Serve

ng serve builds and serves your app on a local server and will automatically rebuild on file changes. By default, your app will be served on http://localhost:4200/.

JavaScript Specification Development

- ES5 : Vanilla JS; Object based scripting language
- ES6 / ES2015 : 2015 : Class, Inheritance, Arrow functions, destructuring etc
- ES7 / ES2016 : Decorators , Object.assign() etc
- ES8 / ES2017 : Object.freeze, Object.values etc

Object Hierarchy in JavaScript



Transpiling Tool (TS => JS):

- Babel
- Traceur
- Typescript Compiler (npm install typescript -g)

Few Decorator in Angular:

- ➢ Component : @Component({})
- ➢ Pipes : @Pipe()

- Directives : @Directive()
- Modules : @NgModule()
- Services : @Injectable()

<u>Popular JS Libraries/Framework in Market</u>

- Angular
- AngularJS
- ReactJS
- BackboneJS
- PolymerJS
- KnockoutJS
- *EmberJS
- JQuery
- NodeJS
- ExpressJS

References

Angular Tutorial : https://angular.io

JavaScript Tutorial : https://javascript.info

TypeScript Tutorial : https://typescriptlang.org

Bootstrap Tutorial : http://getbootstrap.com

Angular CLI Wiki : https://github.com/angular/angular-cli/wiki/angular-cli

JavaScript/ TypeScript Demo : https://github.com/synergy2411/js-demo

Angular Demo : https://github.com/synergy2411/ng-cap-demo