ZOBASOFT

Developing Angular 7 Applications

Content

Why Angular?

There are many front-end JavaScript frameworks to choose from today, each with its own set of trade-offs. Many people were happy with the functionality that Angular 1.x afforded them. Angular 7 improved on that functionality and made it faster, more scalable and more modern. Organizations that found value in Angular 1.x will find more value in Angular 7.

Angular's Advantages

The first release of Angular provided programmers with the tools to develop and architect large scale JavaScript applications, but its age has revealed a number of flaws and sharp edges. Angular 7 was built on five years of community feedback.

Angular 7 Is Easier

The new Angular codebase is more modern, more capable and easier for new programmers to learn than Angular 1.x, while also being easier for project veterans to work with.

With Angular 1, programmers had to understand the differences between Controllers, Services, Factories, Providers and other concepts that could be confusing, especially for new programmers.

Angular 7 is a more streamlined framework that allows programmers to focus on simply building JavaScript classes. Views and controllers are replaced with components, which can be described as a refined version of directives. Even experienced Angular programmers are not always aware of all the capabilities of Angular 1.x directives. Angular 7 components are considerably easier to read, and their API features less jargon than Angular 1.x's directives. Additionally, to help ease the transition to Angular 7.

TypeScript

Angular 7 was written in TypeScript, a superset of JavaScript that implements many new ES2016+ features.

By focusing on making the framework easier for computers to process, Angular 7 allows for a much richer development ecosystem. Programmers using sophisticated text editors (or IDEs) will notice dramatic improvements with auto-completion and type suggestions. These

improvements help to reduce the cognitive burden of learning Angular 7. Fortunately for traditional ES5 JavaScript programmers this does *not* mean that development must be done in TypeScript or ES2015: programmers can still write vanilla JavaScript that runs without transpilation.

Familiarity

Despite being a complete rewrite, Angular 7 has retained many of its core concepts and conventions with Angular 1.x, e.g. a streamlined, "native JS" implementation of dependency injection. This means that programmers who are already proficient with Angular will have an easier time migrating to Angular 6 than another library like React or framework like Ember.

Performance and Mobile

Angular 7 was designed for mobile from the ground up. Aside from limited processing power, mobile devices have other features and limitations that separate them from traditional computers. Touch interfaces, limited screen real estate and mobile hardware have all been considered in Angular 7.

Desktop computers will also see dramatic improvements in performance and responsiveness.

Angular 7, like React and other modern frameworks, can leverage performance gains by rendering HTML on the server or even in a web worker. Depending on application/site design this isomorphic rendering can make a user's experience *feel* even more instantaneous.

The quest for performance does not end with pre-rendering. Angular 7 makes itself portable to native mobile by integrating with NativeScript, an open source library that bridges JavaScript and mobile. Additionally, the Ionic team is working on an Angular 7 version of their product, providing *another* way to leverage native device features with Angular.

Project Architecture and Maintenance

The first iteration of Angular provided web programmers with a highly flexible framework for developing applications. This was a dramatic shift for many web programmers, and while that framework was helpful, it became evident that it was often too flexible. Over time, best practices evolved, and a community-driven structure was endorsed.

Angular 1.x tried to work around various browser limitations related to JavaScript. This was done by introducing a module system that made use of dependency injection. This system was novel, but unfortunately had issues with tooling, notably minification and static analysis.

Angular 2.x makes use of the ES2015 module system, and modern packaging tools like webpack or SystemJS. Modules are far less coupled to the "Angular way", and it's easier to

write more generic JavaScript and plug it into Angular. The removal of minification workarounds and the addition of rigid prescriptions make maintaining existing applications simpler. The new module system also makes it easier to develop effective tooling that can reason better about larger projects.

New Features

Some of the other interesting features in Angular 6 are:

- Form Builder
- Change Detection
- Templating
- Routing
- Annotations
- Observables
- Shadow DOM

Differences Between Angular 1 & 7

> Note that "Transitional Architecture" refers to a style of Angular 1 application written in a way that mimics Angular 2's component style, but with controllers and directives instead of TypeScript classes.

	Old School Angular 1.x	**Angular 1.x Best Practices**	**Transitional Architecture**	**Angular 7**
Nested scopes ("$scope", watches)	Used heavily	Avoided	**Avoided**	Gone
Directives vs controllers	Use as alternatives	Used together	**Directives as components**	Component directives
Controller and service implementation	Functions	Functions	**ES6 classes**	ES6 classes
Module system	Angular's modules	Angular's modules	**ES6 modules**	ES6 modules
Transpiler required	No	No	**TypeScript**	TypeScript

Bootstrapping an Angular Application

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

Bootstrapping Angular applications is certainly different from Angular 1.x, but is still a straightforward procedure. Let's take a look at how this is done.

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.

Why does Angular bootstrap itself in this way? Well there is actually a very good reason. Since Angular is not a web-only based framework, we can write components that will run in NativeScript, or Cordova, or any other environment that can host Angular applications.

The magic is then in our bootstrapping process - we can import which platform we would like to use, depending on the environment we're operating under. In our example, since we were running our Angular application in the browser, we used the bootstrapping process found in `@angular/platform-browser-dynamic`.

It's also a good idea to leave the bootstrapping process in its own separate *main.ts* file. This makes it easier to test (since the components are isolated from the `bootstrap` call), easier to reuse and gives better organization and structure to our application.

There is more to understanding Angular Modules and `@NgModule` which will be covered later, but for now this is enough to get started.

Bootstrapping Providers

The bootstrap process also starts the dependency injection system in Angular. We won't go over Angular's dependency injection system here - that is covered later. Instead let's take a look at an example of how to bootstrap your application with application-wide providers.

For this, we will register a service called `GreeterService` with the `providers` property of the module we are using to bootstrap the application.

app/app.module.ts

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

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

Components

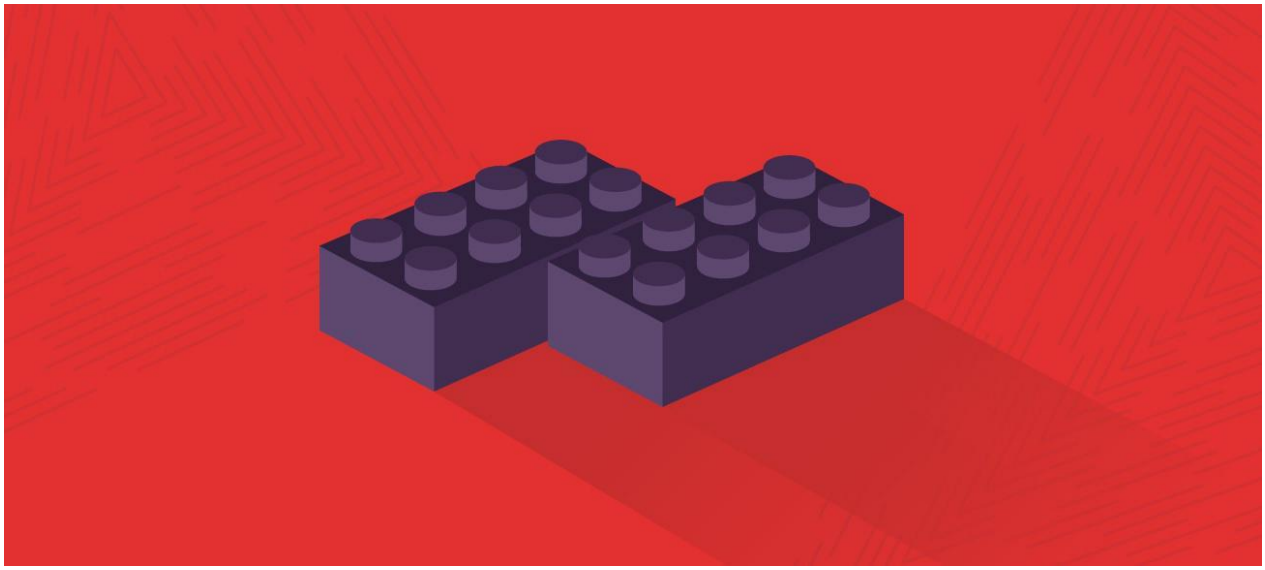


Figure: components

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 6 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.

Application Structure with Components

A useful way of conceptualizing Angular application design is to look at it as a tree of nested components, each having an isolated scope.

For example consider the following:

```
<rio-todo-app>
    <rio-todo-list>
        <rio-todo-item></rio-todo-item>
        <rio-todo-item></rio-todo-item>
        <rio-todo-item></rio-todo-item>
    </rio-todo-list>
    <rio-todo-form></rio-todo-form>
</rio-todo-app>
```

At the root we have `rio-todo-app` which consists of a `rio-todo-list` and a `rio-todo-form` . Within the list we have several `rio-todo-item` s. Each of these components is visible to the user, who can interact with these components and perform actions.

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.

We can now use our component like so:

```
<!-- To bind to a raw string -->
<rio-hello name="World"></rio-hello>
<!-- To bind to a variable in the parent scope --> <rio-hello
[name]="helloName"></rio-hello>
```

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++;
    }
}
```

To send data out of components via outputs, start by defining the outputs attribute. It accepts a list of output parameters that a component exposes to its parent.

app/counter.component.ts

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

@Component({
    selector: 'rio-counter',
    templateUrl: 'app/counter.component.html'
})
export class CounterComponent {
    @Input()        count = 0;
    @Output() result = new EventEmitter<number>();

    increment() {
        this.count++;
        this.result.emit(this.count);
    }
}
```

app/counter.component.html

```
<div>
    <p>Count: {{ count }}</p>
    <button (click)="increment()">Increment</button> </div>
```

app/app.component.ts

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

@Component({
    selector: 'rio-app',
    templateUrl: 'app/app.component.html'
})
export class AppComponent implements OnChange { num = 0;

    parentCount = 0;

            ngOnChange(val: number) {
                this.parentCount = val;
    }
}
```

app/app.component.html

```
<div>
    Parent Num: {{ num }}<br>
    Parent Count: {{ parentCount }}
    <rio-counter [count]="num" (result)="ngOnChange($event)"> </rio-counter>

</div>
```

Together a set of input + output bindings define the public API of your component. In our templates we use the [squareBrackets] to pass inputs and the (parenthesis) to handle outputs.

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` . The code example below, inside class CounterComponent shows how to make property count support two-way binding.

app/counter.component.ts

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

@Component({
    selector: 'rio-counter',
    templateUrl: 'app/counter.component.html'
})
export class CounterComponent {
    @Input() count = 0;
    @Output() countChange = EventEmitter<number>();

    increment() {
        this.count++;
        this.countChange.emit(this.count);
    }
}
```

app/counter.component.html

```
<div>
    <p>
        <ng-content></ng-content>
        Count: {{ count }} -
        <button (click)="increment()">Increment</button> </p>

</div>
```

Access Child Components From the Template

In our templates, we may find ourselves needing to access values provided by the child components which we use to build our own component.

The most straightforward examples of this may be seen dealing with forms or inputs:

app/app.component.html

```
<section >
    <form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)"> <label
        for="name">Name</label>
        <input type="text" name="name" id="name" ngModel> <button
        type="submit">Submit</button>
    </form>
    Form Value: {{formValue}}
</section>
```

app/app.component.ts

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

@Component({
    selector: 'rio-app',
    templateUrl: 'app/app.component.html'
})
export class AppComponent {
    formValue = JSON.stringify({});

    onSubmit (form: NgForm) {
        this.formValue = JSON.stringify(form.value);
    }
}
```

This isn't a magic feature which only forms or inputs have, but rather a way of referencing the instance of a child component in your template. With that reference, you can then access public properties and methods on that component.

app/app.component.html

```
<rio-profile #profile></rio-profile>
My name is {{ profile.name }}
```

app/profile.component.ts

```
@Component({
    selector: 'rio-profile',
    templateUrl: 'app/profile.component.html'
})
export class ProfileComponent {
    name = 'John Doe';
}
```

There are other means of accessing and interfacing with child components, but if you simply need to reference properties or methods of a child, this can be a simple and straightforward method of doing so.

Projection

Projection is a very important concept in Angular. It enables developers to build reusable components and make applications more scalable and flexible. To illustrate that, suppose we have a `ChildComponent` like:

```
@Component({
      selector: 'rio-child',
      template: `
          <div>
              <h4>Child Component</h4>
              {{ childContent }} </div>

      `
})
export class ChildComponent {
    childContent = "Default content";
}
```

What should we do if we want to replace `{{ childContent }}` to any HTML that provided to `ChildComponent` ? One tempting idea is to define an `@Input` containing the text, but what if you wanted to provide styled HTML, or other components? Trying to handle this with an `@Input` can get messy quickly, and this is where content projection comes in. Components by default support projection, and you can use the `ngContent` directive to place the projected content in your template.

So, change `ChildComponent` to use projection:

app/child/child.component.ts

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

@Component({
    selector: 'rio-child',
    template: `
        <div style="border: 1px solid blue; padding: 1rem;">
            <h4>Child Component</h4>
            <ng-content></ng-content>
        </div>
    `
})
export class ChildComponent {
}
```

Then, when we use `ChildComponent` in the template:

app/app.component.html

```
...
    <rio-child>
        <p>My <i>projected</i> content.</p>
    </rio-child>
...
```

This is telling Angular, that for any markup that appears between the opening and closing tag of `<rio-child>`, to place inside of `<ng-content></ng-content>`.

When doing this, we can have other components, markup, etc projected here and the `ChildComponent` does not need to know about or care what is being provided.

But what if we have multiple `<ng-content></ng-content>` and want to specify the position of the projected content to certain `ng-content`? For example, for the previous `ChildComponent`, if we want to format the projected content into an extra `header` and `footer` section:

app/child-select.component.html

```
<div style="...">
    <h4>Child Component with Select</h4>
    <div style="...">
        <ng-content select="header"></ng-content>
    </div>
    <div style="...">
        <ng-content select="section"></ng-content>
    </div>
    <div style="...">
        <ng-content select=".class-select"></ng-content> </div>

    <div style="...">
        <ng-content select="footer"></ng-content>
    </div>
</div>
```

Then in the template, we can use directives, say, `<header>` to specify the position of projected content to the `ng-content` with `select="header"`

app/app.component.html

```
<rio-child-select>
    <section>Section Content</section>
    <div class="class-select">
        div with .class-select
    </div>
    <footer>Footer Content</footer>
    <header>Header Content</header>
</rio-child-select>
...
```

Besides using directives, developers can also select a `ng-content` through css class:

```
<ng-content select=".class-select"></ng-content>
```

`app/app.component.html`

```
        <div class="class-select">
    div with .class-select
</div>
```

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.

Using Other Components

Components depend on other components, directives and pipes. For example, `TodoListComponent` relies on `TodoItemComponent` . To let a component know about these dependencies we group them into a module.

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

import {TodoListComponent} from './components/todo-list.component';
import {TodoItemComponent} from './components/todo-item.component';
 import {TodoInputComponent} from './components/todo-input.component';

@NgModule({
    imports: [ ... ],
    declarations: [
        TodoListComponent,
        TodoItemComponent,
        TodoInputComponent
    ],
    bootstrap: [ ... ]
})
export class ToDoAppModule {
}
```

The property `declarations` expects an array of components, directives and pipes that are part of the module.

Please see the Modules section for more info about `NgModule` .

Directives

A Directive modifies the DOM to change appearance, behaviour 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 6 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 behaviour of a component or element but don't affect the template
  Structural directives - directives that change the behaviour 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';
}
```

Notice that binding a directive works the exact same way as component attribute bindings. Here, we're binding an expression, an object literal, to the ngStyle directive so the directive name must be enclosed in square brackets. ngStyle accepts an object whose properties and values define that element's style. In this case, we can see that both kebab case and lower camel case can be used when specifying a style property. Also notice that both the html style attribute and Angular ngStyle directive are combined when styling the element.

We can remove the style properties out of the template into the component as a property object, which then gets assigned to NgStyle using property binding. This allows dynamic changes to the values as well as provides the flexibility to add and remove style properties.

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

    alertStyles = {
        'color': 'red',
        'font-weight': 'bold',
        'borderBottom': this.borderStyle
    };
}
```

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.

Binding a string

We can bind a string directly to the attribute. This works just like adding an html class attribute.

```
@Component({
    selector: 'app-class-as-string',
    template: `
        <p ngClass="centered-text underlined" class="orange"> <ng-
            content></ng-content>
        </p>
    `,
    styles: [`
        .centered-text {
            text-align: center;
        }

        .underlined {
            border-bottom: 1px solid #ccc;
        }

        .orange {
            color: orange;
        }
    `]
})
export class ClassAsStringComponent {
}
```

In this case, we're binding a string directly so we avoid wrapping the directive in square brackets. Also notice that the ngClass works with the class attribute to combine the final classes.

Binding an array

```
@Component({
    selector: 'app-class-as-array',
    template: `
        <p [ngClass]="['warning', 'big']">
            <ng-content></ng-content>
        </p>
    `,
    styles: [`
        .warning {
            color: red;
            font-weight: bold;
        }

        .big {
            font-size: 1.2rem;
        }
    `]
})
export class ClassAsArrayComponent {
}
```

Here, since we are binding to the `ngClass` directive by using an expression, we need to wrap the directive name in square brackets. Passing in an array is useful when you want to have a function put together the list of applicable class names.

Binding an object

Lastly, an object can be bound to the directive. Angular applies each property name of that object to the component if that property is true.

```
@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`.

Note: For those who are unfamiliar with the `template` tag, it is an HTML element with a few special properties. Content nested in a template tag is not rendered on page load and is something that is meant to be loaded through code at runtime. For more information on the `template` tag, visit the MDN documentation.

Structural directives have their own special syntax in the template that works as syntactic sugar.

```
@Component({
    selector: 'app-directive-example',
    template: `
        <p *structuralDirective="expression">
            Under a structural directive.
        </p>
    `
})
```

Instead of being enclosed by square brackets, our dummy structural directive is prefixed with an asterisk. Notice that the binding is still an expression binding even though there are no square brackets. That's due to the fact that it's syntactic sugar that allows using the directive in a more intuitive way and similar to how directives were used in Angular 1. The component template above is equivalent to the following:

```
@Component({
    selector: 'app-directive-example',
    template: `
        <template [structuralDirective]="expression">
            <p>
                Under a structural directive.
            </p>
        </template>
    `
})
```

Here, we see what was mentioned earlier when we said that structural directives use the `template` tag. Angular also has a built-in `template` directive that does the same thing:

```
@Component({
    selector: 'app-directive-example',
    template: `
                        <p template="structuralDirective expression">
            Under a structural directive.
        </p>
    `
})
```

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;
    }
}
```

Clicking the button will toggle whether or not `IfExampleComponent` is a part of the DOM and not just whether it is visible or not. This means that every time the button is clicked, `IfExampleComponent` will be created or destroyed. This can be an issue with components that have expensive create/destroy actions. For example, a component could have a large child subtree or make several HTTP calls when constructed. In these cases it may be better to avoid using `ngIf` if possible.

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]="episode"> {{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' }
    ];
}
```

The `NgFor` directive has a different syntax from other directives we've seen. If you're familiar with the for...of statement, you'll notice that they're almost identical. `NgFor` lets you specify an iterable object to iterate over and the name to refer to each item by inside the scope. In our example, you can see that `episode` is available for interpolation as well as property binding. The directive does some extra parsing so that when this is expanded to template form, it looks a bit different:

```
@Component({
    selector: 'app',
    template: `
        <template ngFor [ngForOf]="episodes" let-episode> <app-for-
            example [episode]="episode">
                {{episode.title}}
            </app-for-example>
        </template>
    `
})
```

Notice that there is an odd `let-episode` property on the template element. The `NgFor` directive provides some variables as context within its scope. `let-episode` is a context binding and here it takes on the value of each item of the iterable.

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

```
@Component({
  selector: 'app-root',
  template: `
    <app-for-example
      *ngFor="let episode of episodes; let i = index; let isOdd = odd"
      [episode]="episode"
      [ngClass]="{ odd: isOdd }">
      {{i+1}}. {{episode.title}}
    </app-for-example>

    <hr>

    <h2>Desugared</h2>

    <template ngFor [ngForOf]="episodes" let-episode let-i="index" let-isOdd="odd"> <for-example
      [episode]="episode" [ngClass]="{ odd: isOdd }">
        {{i+1}}. {{episode.title}}
      </for-example>
    </template>
  `
})
```

trackBy

Often `NgFor` is used to iterate through a list of objects with a unique ID field. In this case, we can provide a `trackBy` function which helps Angular keep track of items in the list so that it can detect which items have been added or removed and improve performance.

Angular will try and track objects by reference to determine which items should be created and destroyed. However, if you replace the list with a new source of objects, perhaps as a result of an API request - we can get some extra performance by telling Angular how we want to keep track of things.

For example, if the `Add Episode` button was to make a request and return a new list of episodes, we might not want to destroy and re-create every item in the list. If the episodes have a unique ID, we could add a `trackBy` function:

```
@Component({
  selector: 'app-root',
  template: `
  <button
    (click)="addOtherEpisode()"
    [disabled]="otherEpisodes.length === 0">
    Add Episode
  </button>
  <app-for-example
    *ngFor="let episode of episodes;
    let i = index; let isOdd = odd;
    trackBy: trackById" [episode]="episode"
    [ngClass]="{ odd: isOdd }">
    {{episode.title}}
  </app-for-example>
  `
})
export class AppComponent {

  otherEpisodes = [
    { title: 'Two Swords', director: 'D. B. Weiss', id: 8 },
    { title: 'The Lion and the Rose', director: 'Alex Graves', id: 9 },
    { title: 'Breaker of Chains', director: 'Michelle MacLaren', id: 10 },
    { title: 'Oathkeeper', director: 'Michelle MacLaren', id: 11 }]

  episodes = [
    { title: 'Winter Is Coming', director: 'Tim Van Patten', id: 0 },
    { title: 'The Kingsroad', director: 'Tim Van Patten', id: 1 },
    { title: 'Lord Snow', director: 'Brian Kirk', id: 2 },
    { title: 'Cripples, Bastards, and Broken Things', director: 'Brian Kirk', id: 3 },
    { title: 'The Wolf and the Lion', director: 'Brian Kirk', id: 4 },
    { title: 'A Golden Crown', director: 'Daniel Minahan', id: 5 },
    { title: 'You Win or You Die', director: 'Daniel Minahan', id: 6 }
    { title: 'The Pointy End', director: 'Daniel Minahan', id: 7 }
  ];

  addOtherEpisode() {
    //  We want to create a new object reference for sake of example let episodesCopy =
    JSON.parse(JSON.stringify(this.episodes))
    this.episodes=[...episodesCopy,this.otherEpisodes.pop()];
  }
  trackById(index: number, episode: any): number { return
    episode.id;
  }
}
```

To see how this can affect the ForExample component, let's add some logging to it.

```
export class ForExampleComponent {
    @Input() episode;

    ngOnInit() {
        console.log('component created', this.episode)
    }
    ngOnDestroy() {
        console.log('destroying component', this.episode)
    }
}
```

When we view the example, as we click on `Add Episode` , we can see console output indicating that only one component was created - for the newly added item to the list.

However, if we were to remove the `trackBy` from the `*ngFor` - every time we click the button, we would see the items in the component getting destroyed and recreated.

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;
    }
}
```

Here we see the `ngSwitch` attribute directive being attached to an element. This expression bound to the directive defines what will compared against in the switch structural directives. If an expression bound to `ngSwitchCase` matches the one given to `ngSwitch` , those components are created and the others destroyed. If none of the cases match, then components that have `ngSwitchDefault` bound to them will be created and the others destroyed. Note that multiple components can be matched using `ngSwitchCase` and in those cases all matching components will be created. Since components are created or destroyed be aware of the costs in doing so.

Using Multiple Structural Directives

Sometimes we'll want to combine multiple structural directives together, like iterating using `ngFor` but wanting to do an `ngIf` to make sure that the value matches some or multiple conditions. Combining structural directives can lead to unexpected results however, so Angular requires that a template can only be bound to one directive at a time. To apply multiple directives we'll have to expand the sugared syntax or nest template tags.

```
@Component({
    selector: 'app-root',
    template: `
        <template ngFor [ngForOf]="[1,2,3,4,5,6]" let-item> <div *ngIf="item >
            3">
                {{item}}
            </div>
        </template>
    `
})
```

The previous tabs example can use `ngFor` and `ngSwitch` if the tab title and content is abstracted away into the component class.

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

@Component({
    selector: 'app-root',
    template: `
        <div class="tabs-selection">
            <tab
                *ngFor="let tab of tabs; let i = index"
                [active]="isSelected(i)"
                (click)="setTab(i)">

                {{ tab.title }} </tab>

        </div>

        <div [ngSwitch]="tabNumber">
            <template ngFor [ngForOf]="tabs" let-tab let-i="index"> <tab-content
                *ngSwitchCase="i">
                    {{tab.content}}
                </tab-content>
            </template>
            <tab-content *ngSwitchDefault>Select a tab</tab-content> </div>

    `
})
     export class AppComponent {
    tabNumber: number = -1;

    tabs = [
        { title: 'Tab 1', content: 'Tab content 1' },
        { title: 'Tab 2', content: 'Tab content 2' },
        { title: 'Tab 3', content: 'Tab content 3' }, ];


    setTab(num: number) {
        this.tabNumber = num;
    }

    isSelected(num: number) {
        return this.tabNumber === i;
    }
}
```

Advanced Components

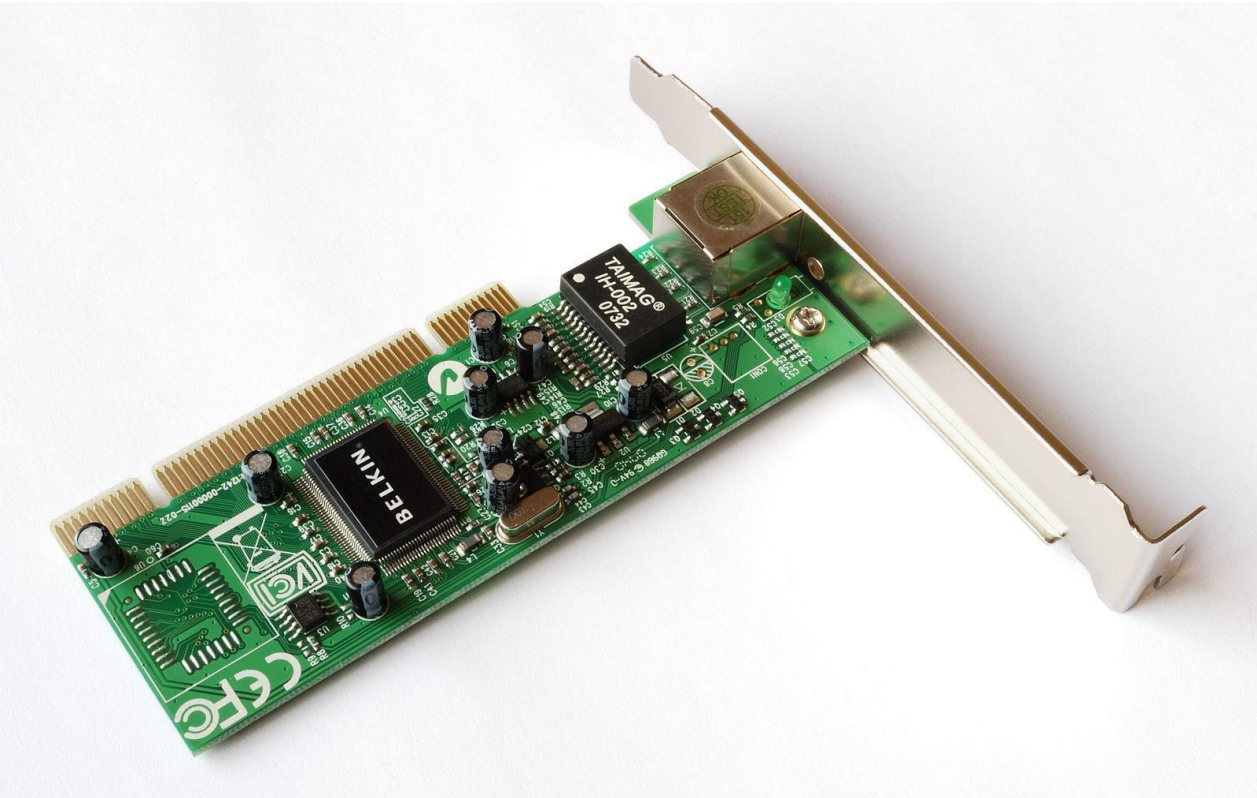


Figure: GB Network PCI Card by Harke is licensed under Public Domain (https://commons.wikimedia.org/wiki/File:GB_Network_PCI_Card.jpg)

Now that we are familiar with component basics, we can look at some of the more interesting things we can do with them.

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
-

Accessing Child Component Classes

@ViewChild and @ViewChildren

The @ViewChild and @ViewChildren decorators provide access to the class of child component from the containing component.

The @ViewChild is a decorator function that takes the name of a component class as its input and finds its selector in the template of the containing component to bind to. @ViewChild can also be passed a template reference variable.

For example, we bind the class AlertComponent to its selector <app-alert> and assign it to the property alert . This allows us to gain access to class methods, like show() .

```
import { Component, ViewChild } from '@angular/core'; import {
AlertComponent } from './alert.component';

@Component({
      selector: 'app-root',
      template: `
      <app-alert>My alert</app-alert>
         <button (click)="showAlert()">Show Alert</button>`
})
export class AppComponent { @ViewChild(AlertComponent) alert:
   AlertComponent;

   showAlert() {
```

```
        this.alert.show();
    }
}
```

In the interest of separation of concerns, we'd normally want to have child elements take care of their own behaviors and pass in an `@Input()` . However, it might be a useful construct in keeping things generic.

When there are multiple embedded components in the template, we can also use `@ViewChildren` . It collects a list of instances of the Alert component, stored in a QueryList object that behaves similar to an array.

```
import { Component, QueryList, ViewChildren } from '@angular/core'; import {
AlertComponent } from './alert.component';

@Component({
        selector: 'app-root',
        template: `
        <app-alert ok="Next" (close)="showAlert(2)">
            Step 1: Learn angular
        </app-alert>
        <app-alert ok="Next" (close)="showAlert(3)">
            Step 2: Love angular
        </app-alert>
        <app-alert ok="Close">
            Step 3: Build app
        </app-alert>
            <button (click)="showAlert(1)">Show steps</button>`
})
export class AppComponent {
    @ViewChildren(AlertComponent) alerts: QueryList<AlertComponent>; alertsArr = [];


    ngAfterViewInit() {
        this.alertsArr = this.alerts.toArray();
    }

    showAlert(step) {
        this.alertsArr[step - 1].show(); // step 1 is alert index 0
    }
}
```

As shown above, given a class type to `@ViewChild` and `@ViewChildren` a child component or a list of children component are selected respectively using their selector from the template. In addition both `@ViewChild` and `@ViewChildren` can be passed a selector string:

```
@Component({
        selector: 'app-root',
        template: `
        <app-alert #first ok="Next" (close)="showAlert(2)"> Step 1: Learn
            angular
        </app-alert>
        <app-alert ok="Next" (close)="showAlert(3)">
            Step 2: Love angular
        </app-alert>
        <app-alert ok="Close">
            Step 3: Build app
        </app-alert>
            <button (click)="showAlert(1)">Show steps</button>`
})
export class AppComponent {
    @ViewChild('first') alert: AlertComponent;
    @ViewChildren(AlertComponent) alerts: QueryList<AlertComponent>;
    // ...
}
```

Note that view children will not be set until the `ngAfterViewInit` lifecycle hook is called.

@ContentChild and @ContentChildren

`@ContentChild` and `@ContentChildren` work the same way as the equivalent `@ViewChild` and `@ViewChildren`, however, the key difference is that `@ContentChild` and `@ContentChildren` select from the projected content within the component.

Again, note that content children will not be set until the `ngAfterContentInit` component lifecycle hook.

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.

```
@Component({
// ...
encapsulation: ViewEncapsulation.None,
styles: [
    // ...
]
})
export class HelloComponent {
//    ...
}
```

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 - you'll see them in the HTTP service and the event system.

Using Observables

Let's take a look at a basic example of how to create and use an Observable in an Angular component:

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

@Component({
        selector: 'app',
        template: `
            <b>Angular Component Using Observables!</b>

            <h6 style="margin-bottom: 0">VALUES:</h6>
            <div *ngFor="let value of values">- {{ value }}</div>

            <h6 style="margin-bottom: 0">ERRORs:</h6>
            <div>Errors: {{anyErrors}}</div>

            <h6 style="margin-bottom: 0">FINISHED:</h6>
            <div>Finished: {{ finished }}</div>

            <button style="margin-top: 2rem;" (click)="init()">Init</button>
        `
})
export class MyApp {

    private data: Observable<Array<number>>;
    private values: Array<number> = [];
    private anyErrors: boolean;
    private finished: boolean;

    constructor() {
    }

    init() {
            this.data = new Observable(observer => {
                    setTimeout(() => {
                            observer.next(42);
                    }, 1000);

                    setTimeout(() => {
                            observer.next(43);
                    }, 2000);

                    setTimeout(() => {
                            observer.complete();
                    }, 3000);
            });
```

```
        let subscription = this.data.subscribe(
            value => this.values.push(value),
            error => this.anyErrors = true,
            () => this.finished = true
        );
    }

}
```

First we import Observable into our component from rxjs/Observable . Next, in our constructor we create a new Observable . Note that this creates an Observable data type that contains data of number type. This illustrates the stream of data that Observables offer as well as giving us the ability to maintain integrity of the type of data we are expecting to receive.

Next we call subscribe on this Observable which allows us to listen in on any data that is coming through. In subscribing we use three distinctive callbacks: the first one is invoked when receiving new values, the second for any errors that arise and the last represents the function to be invoked when the sequence of incoming data is complete and successful.

We can also use forEach to listen for incoming data. The key difference between forEach and subscribe is in how the error and completion callbacks are handled. The forEach call only accepts the 'next value' callback as an argument; it then returns a promise instead of a subscription.

When the Observable completes, the promise resolves. When the Observable encounters an error, the promise is rejected.

You can think of Observable.of(1, 2, 3).forEach(doSomething) as being semantically equivalent to:

```
new Promise((resolve, reject) => {
    Observable.of(1, 2, 3).subscribe(
        doSomething,
        reject,
        resolve);
});
```

The forEach pattern is useful for a sequence of events you only expect to happen once.

```
export class MyApp {

    private data: Observable<Array<number>>;
    private values: Array<number> = [];
    private anyErrors: boolean;
    private finished: boolean;

    constructor() {
    }

    init() {
            this.data = new Observable(observer => {
                    setTimeout(() => {
                            observer.next(42);
                    }, 1000);

                    setTimeout(() => {
                            observer.next(43);
                    }, 2000);

                    setTimeout(() => {
                            observer.complete();
                    }, 3000);

                    this.status = "Started";
            });

                                            let subscription = this.data.forEach(v => this.values.push(v))
                        .then(() => this.status = "Ended");
    }

}
```

Error Handling

If something unexpected arises we can raise an error on the `Observable` stream and use the function reserved for handling errors in our `subscribe` routine to see what happened.

```
export class App {

    values: number[] = [];
    anyErrors: Error;
    private data: Observable<number[]>;

    constructor() {

        this.data = new Observable(observer => {
            setTimeout(() => {
                observer.next(10);
            }, 1500);
            setTimeout(() => {
                observer.error(new Error('Something bad happened!'));
            }, 2000);
            setTimeout(() => {
                observer.next(50);
            }, 2500);
        });

        let subscription = this.data.subscribe(
            value => this.values.push(value),
            error => this.anyErrors = error
        );
    }
}
```

Here an error is raised and caught. One thing to note is that if we included a `.complete()` after we raised the error, this event will not actually fire. Therefore you should remember to include some call in your error handler that will turn off any visual loading states in your application.

Disposing Subscriptions and Releasing Resources

In some scenarios we may want to unsubscribe from an Observable stream. Doing this is pretty straightforward as the .subscribe() call returns a data type that we can call .unsubscribe() on.

```
export class MyApp {

    private data: Observable<Array<string>>;
    private value: string;
    private subscribed: boolean;
    private status: string;

        init() {

                this.data = new Observable(observer => {
                        let timeoutId = setTimeout(() => {
                                observer.next('You will never see this message'); }, 2000);


                        this.status = 'Started';

                        return onUnsubscribe = () => {
                                this.subscribed = false;
                                this.status = 'Finished';
                                clearTimeout(timeoutId);
                        }
                });

                let subscription = this.data.subscribe(
                        value => this.value = value,
                        error => console.log(error),
                        () => this.status = 'Finished';
                );
                this.subscribed = true;

                setTimeout(() => {
                    subscription.unsubscribe();
                }, 1000);
        }

}
```

Calling .unsubscribe() will unhook a member's callbacks listening in on the Observable stream. When creating an Observable you can also return a custom callback, onUnsubscribe , that will be invoked when a member listening to the stream has unsubscribed. This is useful for any kind of cleanup that must be implemented. If we did not clear the setTimeout then values

would still be emitting, but there would be no one listening. To save resources we should stop values from being emitted. An important thing to note is that when you call .unsubscribe() you are destroying the subscription object that is listening, therefore the on-complete event attached to that subscription object will not get called.

In most cases we will not need to explicitly call the unsubscribe method unless we want to cancel early or our Observable has a longer lifespan than our subscription. The default behavior of Observable operators is to dispose of the subscription as soon as .complete() or .error() messages are published. Keep in mind that RxJS was designed to be used in a "fire and forget" fashion most of the time.

Observables vs Promises

Both Promises and Observables provide us with abstractions that help us deal with the asynchronous nature of our applications. However, there are important differences between the two:

- As seen in the example above, Observables can define both the setup and teardown aspects of asynchronous behavior.
- Observables are cancellable.

  Moreover, Observables can be retried using one of the retry operators provided by the API, such as retry and retryWhen . On the other hand, Promises require the caller to have access to the original function that returned the promise in order to have a retry capability.

Using Observables From Other Sources

In the example above we created Observables from scratch which is especially useful in understanding the anatomy of an Observable .

However, we will often create Observables from callbacks, promises, events, collections or using many of the operators available on the API.

Observable HTTP Events

A common operation in any web application is getting or posting data to a server. Angular applications do this with the Http library, which previously used Promises to operate in an

asynchronous manner. The updated `Http` library now incorporates `Observables` for triggering events and getting new data. Let's take a quick look at this:

```
import {Component} from '@angular/core';
import {HttpClient} from '@angular/common/http';
import {flatMap} from 'rxjs/operators';

@Component({
        selector: 'app',
        template: `
            <b>Angular HTTP requests using RxJs Observables!</b> <ul>

                <li *ngFor="let doctor of doctors">{{doctor.name}}</li> </ul>


            `
})

export class MyApp {
    private doctors = [];

    constructor(http: Http) {
        http.get('http://jsonplaceholder.typicode.com/users/')
                .pipe(flatMap((data) => data.json()))
                .subscribe((data) => {
                    this.doctors.push(data);

                });
    }
}
```

This basic example outlines how the `Http` library's common routines like `get` , `post` , `put` and `delete` all return `Observables` that allow us to asynchronously process any resulting data.

Observable Form Events

Let's take a look at how treated as an `Observable` value of the input field.

`Observables` are used in Angular forms. Each field in a form is that we can subscribe to and listen for any changes made to the

```
import {Component} from '@angular/core';
import {FormControl, FormGroup, FormBuilder} from '@angular/forms';
import {map} from 'rxjs/operators';

@Component({
        selector: 'app',
        template: `
```

```
            <form [formGroup]="coolForm">
                    <input formControlName="email">
                </form>
            <div>
                    <b>You Typed Reversed:</b> {{data}}
                </div>
        `
})

export class MyApp {

        email: FormControl;
        coolForm: FormGroup;
        data: string;

        constructor(private fb: FormBuilder) {
                this.email = new FormControl();

                this.coolForm = fb.group({
                        email: this.email
                });

                this.email.valueChanges
                .pipe(map(n=>n.split('').reverse().join('')))
                .subscribe(value => this.data = value);
        }
}
```

Here we have created a new form by initializing a new FormControl field and grouped it into a FormGroup tied to the coolForm HTML form. The Control field has a property .valueChanges that returns an Observable that we can subscribe to. Now whenever a user types something into the field we'll get it immediately.

Observables Array Operations

In addition to simply iterating over an asynchronous collection, we can perform other operations such as filter or map and many more as defined in the RxJS API. This is what bridges an Observable with the iterable pattern, and lets us conceptualize them as collections.

Let's expand our example and do something a little more with our stream:

```
export class MyApp {
    private doctors = [];

    constructor(http: Http) {
        http.get('http://jsonplaceholder.typicode.com/users/')
            .pipe(
                flatMap((response) => response.json()),
```

```
            filter((person) => person.id > 5),
            map((person) => "Dr. " + person.name),
          )
          .subscribe((data) => {
            this.doctors.push(data);
          });
    }
}
```

Here are two really useful array operations - `map` and `filter`. What exactly do these do?

- `map` will create a new array with the results of calling a provided function on every element in this array. In this example we used it to create a new result set by iterating through each item and appending the "Dr." abbreviation in front of every user's name. Now every object in our array has "Dr." prepended to the value of its name property.
- `filter` will create a new array with all elements that pass the test implemented by a provided function. Here we have used it to create a new result set by excluding any user whose `id` property is less than six.

Now when our `subscribe` callback gets invoked, the data it receives will be a list of JSON objects whose `id` properties are greater than or equal to six and whose `name` properties have been prepended with `Dr.`.

Note the chaining function style, and the optional static typing that comes with TypeScript, that we used in this example. Most importantly functions like `filter` return an `Observable`, as in `Observables` beget other `Observables`, similarly to promises. In order to use `map` and `filter` in a chaining sequence we have flattened the results of our `Observable` using `flatMap`. Since `filter` accepts an `Observable`, and not an array, we have to convert our array of JSON objects from `data.json()` to an `Observable` stream. This is done with `flatMap`.

There are many other array operations you can employ in your `Observables`; look for them in the RxJS API.

Cold vs Hot Observables

Observables can be classified into two main groups: hot and cold Observables . Let's start with a cold Observable .

```
const obsv = new Observable(observer => {

    setTimeout(() => {
        observer.next(1);
    }, 1000);

    setTimeout(() => {
        observer.next(2);
    }, 2000);

    setTimeout(() => {
        observer.next(3);
    }, 3000);

    setTimeout(() => {
        observer.next(4);
    }, 4000);

});

//     Subscription A
setTimeout(() => {
    obsv.subscribe(value => console.log(value)); }, 0);


//     Subscription B
setTimeout(() => {
    obsv.subscribe(value => console.log(`>>>> ${value}`)); }, 2500);
```

In the above case subscriber B subscribes 2000ms after subscriber A. Yet subscriber B is starting to get values like subscriber A only time shifted. This behavior is referred to as a *cold Observable* . A useful analogy is watching a pre-recorded video, such as on Netflix. You press Play and the movie starts playing from the beginning. Someone else can start playing the same movie in their own home 25 minutes later.

On the other hand there is also a *hot Observable* , which is more like a live performance. You attend a live band performance from the beginning, but someone else might be 25 minutes late to the show. The band will not start playing from the beginning and the latecomer must start watching the performance from where it is.

We have already encountered both kind of Observables Observable , while the example that uses valueChangesObservable .

. The example above is a cold on our text field input is a hot

Converting from Cold Observables to Hot Observables

A useful method within RxJS API is the `publish` method. This method takes in a cold `Observable` as its source and returns an instance of a `ConnectableObservable` . In this case we will have to explicitly call `connect` on our hot `Observable` to start broadcasting values to its subscribers.

```
const obsv = new Observable(observer => {

    setTimeout(() => {
        observer.next(1);
    }, 1000);

    setTimeout(() => {
        observer.next(2);
    }, 2000);

    setTimeout(() => {
        observer.next(3);
    }, 3000);

    setTimeout(() => {
        observer.next(4);
    }, 4000);

}).publish();

obsv.connect();

//    Subscription A
setTimeout(() => {
    obsv.subscribe(value => console.log(value)); }, 0);


//    Subscription B
setTimeout(() => {
    obsv.subscribe(value => console.log(` ${value}`)); }, 2500);
```

In the case above, the live performance starts at `1000ms` , subscriber A arrived to the concert hall at `0s` to get a good seat and our subscriber B arrived at the performance at `2500ms` and missed a bunch of songs.

Another useful method to work with hot `Observables` instead of `connect` is `refCount` . This is an auto connect method, that will start broadcasting as soon as there is more than one

subscriber. Analogously, it will stop if the number of subscribers goes to 0; in other words, if everyone in the audience walks out, the performance will stop.

Dependency Injection

Dependency Injection (DI) was a core feature in Angular 1.x, and that has not changed in Angular 6. DI is a programming concept that predates Angular. The purpose of DI is to simplify dependency management in software components. By reducing the amount of information a component needs to know about its dependencies, unit testing can be made easier and code is more likely to be flexible.

Angular 6 improves on Angular 1.x's DI model by unifying Angular 1.x's two injection systems. Tooling issues with respect to static analysis, minification and namespace collisions have also been fixed in Angular 6.

What is DI?

So dependency injection makes programmers' lives easier, but what does it *really* do?

Consider the following code:

```
class Hamburger {
    private bun: Bun;
    private patty: Patty;
    private toppings: Toppings;
    constructor() {
        this.bun = new Bun('withSesameSeeds');
        this.patty = new Patty('beef');
        this.toppings = new Toppings(['lettuce', 'pickle', 'tomato']);
    }
}
```

The above code is a contrived class that represents a hamburger. The class assumes a `Hamburger` consists of a `Bun`, `Patty` and `Toppings`. The class is also responsible for *making* the `Bun`, `Patty` and `Toppings`. This is a bad thing. What if a vegetarian burger were needed? One naive approach might be:

```
class VeggieHamburger {
    private bun: Bun;
    private patty: Patty;
    private toppings: Toppings;

    constructor() {
        this.bun = new Bun('withSesameSeeds');
        this.patty = new Patty('tofu');
        this.toppings = new Toppings(['lettuce', 'pickle', 'tomato']);
    }
}
```

There, problem solved right? But what if we need a gluten free hamburger? What if we want different toppings... maybe something more generic like:

```
class Hamburger {
    private bun: Bun;
    private patty: Patty;
    private toppings: Toppings;

    constructor(bunType: string, pattyType: string, toppings: string[]) { this.bun = new
        Bun(bunType);
        this.patty = new Patty(pattyType);
        this.toppings = new Toppings(toppings);
    }
}
```

Okay this is a little different, and it's more flexible in some ways, but it is still quite brittle. What would happen if the `Patty` constructor changed to allow for new features? The whole `Hamburger` class would have to be updated. In fact, any time any of these constructors used in `Hamburger` 's constructor are changed, `Hamburger` would also have to be changed.

Also, what happens during testing? How can `Bun` , `Patty` and `Toppings` be effectively mocked?

Taking those concerns into consideration, the class could be rewritten as:

```
class Hamburger {
    private bun: Bun;
    private patty: Patty;
    private toppings: Toppings;

    constructor(bun: Bun, patty: Patty, toppings: Toppings) { this.bun = bun;

        this.patty = patty;
        this.toppings = toppings;
    }
}
```

Now when `Hamburger` is instantiated it does not need to know anything about its `Bun` , `Patty` , or `Toppings` . The construction of these elements has been moved out of the class. This pattern is so common that TypeScript allows it to be written in shorthand like so:

```
class Hamburger {
    constructor(private bun: Bun, private patty: Patty,
        private toppings: Toppings) {}
}
```

The `Hamburger` class is now simpler and easier to test. This model of having the dependencies provided to `Hamburger` is basic dependency injection.

However there is still a problem. How can the instantiation of `Bun` , `Patty` and `Toppings` best be managed? This is where dependency injection as a *framework* can benefit programmers, and it is what Angular provides with its dependency injection system.

DI Framework

So there's a fancy new `Hamburger` class that is easy to test, but it's currently awkward to work with. Instantiating a `Hamburger` requires:

```
const hamburger = new Hamburger(new Bun(), new Patty('beef'), new Toppings([]));
```

That's a lot of work to create a `Hamburger` , and now all the different pieces of code that make `Hamburger` s have to understand how `Bun` , `Patty` and `Toppings` get instantiated.

One approach to dealing with this new problem might be to make a factory function like so:

```
    function makeHamburger() {
        const bun = new Bun();
  const patty = new Patty('beef');
  const toppings = new Toppings(['lettuce', 'tomato', 'pickles']); return new
  Hamburger(bun, patty, toppings);
}
```

This is an improvement, but when more complex `Hamburger` s need to be created this factory will become confusing. The factory is also responsible for knowing how to create four different components. This is a lot for one function.

This is where a dependency injection framework can help. DI Frameworks have the concept of an `Injector` object. An Injector is a lot like the factory function above, but more general, and powerful. Instead of one giant factory function, an Injector has a factory, or *recipe* (pun intended) for a collection of objects. With an `Injector` , creating a `Hamburger` could be as easy as:

```
const injector = new Injector([Hamburger, Bun, Patty, Toppings]); const burger =
injector.get(Hamburger);
```

Angular's DI

The last example introduced a hypothetical `Injector` object. Angular simplifies DI even further. With Angular, programmers almost never have to get bogged down with injection details.

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)

For example:

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

@Component({
  //  ...
})
class ChatWidget {
  constructor(private authService: AuthService, private authWidget: AuthWidget, private chatSocket:
    ChatSocket) {}
}

@NgModule({
  declarations: [ ChatWidget ]
})
export class AppModule {
};
```

In the above example the `AppModule` is told about the `ChatWidget` class. Another way of saying this is that Angular has been *provided* a `ChatWidget`.

That seems pretty straightforward, but astute readers will be wondering how Angular knows how to build `ChatWidget`. What if `ChatWidget` was a string, or a plain function?

Angular *assumes* that it's being given a class.

What about `AuthService`, `AuthWidget` and `ChatSocket`? How is `ChatWidget` getting those?

It's not, at least not yet. Angular does not know about them yet. That can be changed easily enough:

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

@Component({
  //  ...
})
class ChatWidget {
  constructor(private authService: AuthService, private authWidget: AuthWidget, private chatSocket:
    ChatSocket) {}
}
@Component({
  //  ...
})
class AuthWidget {}

@Injectable()
class AuthService {}

@Injectable()
class ChatSocket {}

@NgModule({
  declarations[ ChatWidget, AuthWidget ]
  providers: [ AuthService, ChatSocket ],
})
```

Okay, this is starting to look a little bit more complete. Although it's still unclear how `ChatWidget` is being told about its dependencies. Perhaps that is related to those odd `@Injectable` statements.

@Inject and @Injectable

Statements that look like `@SomeName` are decorators. Decorators are a proposed extension to JavaScript. In short, decorators let programmers modify and/or tag methods, classes, properties and parameters. There is a lot to decorators. In this section the focus will be on decorators relevant to DI: `@Inject` and `@Injectable` . For more information on Decorators please see the EcmaScript 6 and TypeScript Features section.

@Inject()

`@Inject()` is a *manual* mechanism for letting Angular know that a *parameter* must be injected. It can be used like so:

```
import { Component, Inject } from '@angular/core'; import { ChatWidget }
from '../components/chat-widget';

@Component({
    selector: 'app-root',
    template: `Encryption: {{ encryption }}`
})
export class AppComponent {
    encryption = this.chatWidget.chatSocket.encryption;

    constructor(@Inject(ChatWidget) private chatWidget) { }
}
```

In the above we've asked for `chatWidget` to be the singleton Angular associates with the `class` symbol `ChatWidget` by calling `@Inject(ChatWidget)` . It's important to note that we're using `ChatWidget` for its typings *and* as a *reference* to its singleton. We are *not* using `ChatWidget` to instantiate anything, Angular does that for us behind the scenes.

When using TypeScript, `@Inject` is only needed for injecting *primitives*. TypeScript's types let Angular know what to do in most cases. The above example would be simplified in TypeScript to:

```
import { Component } from '@angular/core';
import { ChatWidget } from '../components/chat-widget';

@Component({
    selector: 'app',
    template: `Encryption: {{ encryption }}`
})
export class App {
    encryption = this.chatWidget.chatSocket.encryption;

    constructor(private chatWidget: ChatWidget) { }
}
```

@Injectable()

`@Injectable()` lets Angular know that a *class* can be used with the dependency injector. `@Injectable()` is not *strictly* required if the class has *other* Angular decorators on it or does not have any dependencies. What is important is that any class that is going to be injected with Angular *is decorated*. However, best practice is to decorate injectables with `@Injectable()`, as it makes more sense to the reader.

Here's an example of `ChatWidget` marked up with `@Injectable`:

```
import { Injectable } from '@angular/core';
import { AuthService } from './auth-service';
import { AuthWidget } from './auth-widget';
import { ChatSocket } from './chat-socket';

@Injectable()
export class ChatWidget {
    constructor(
        public authService: AuthService,
        public authWidget: AuthWidget,
        public chatSocket: ChatSocket) { }
}
```

In the above example Angular's injector determines what to inject into `ChatWidget`'s constructor by using type information. This is possible because these particular dependencies are typed, and are *not primitive* types. In some cases Angular's DI needs more information than just types.

Injection Beyond Classes

So far the only types that injection has been used for have been classes, but Angular is not limited to injecting classes. The concept of `providers` was also briefly touched upon.

So far `providers` have been used with Angular's `@NgModule` meta in an array. `providers` have also all been class identifiers. Angular lets programmers specify providers with a more verbose "recipe". This is done with by providing Angular an Object literal (`{}`):

```
import { NgModule } from '@angular/core';
import { App } from './containers/app'; // hypothetical app component import { ChatWidget }
from './components/chat-widget';

@NgModule({
    providers: [ { provide: ChatWidget, useClass: ChatWidget } ], })

export class DiExample {};
```

This example is yet another example that `provide` s a class, but it does so with Angular's longer format.

This long format is really handy. If the programmer wanted to switch out `ChatWidget` implementations, for example to allow for a `MockChatWidget` , they could do this easily:

```
import { NgModule } from '@angular/core';
import { App } from './containers/app'; // hypothetical app component import { ChatWidget }
from './components/chat-widget';
import { MockChatWidget } from './components/mock-chat-widget';

@NgModule({
    providers: [ { provide: ChatWidget, useClass: MockChatWidget } ], })

export class DiExample {};
```

The best part of this implementation swap is that the injection system knows how to build `MockChatWidget` , and will sort all of that out.

The injector can use more than classes though. `useValue` and `useFactory` are two other examples of `provider` "recipes" that Angular can use. For example:

```
import { NgModule } from '@angular/core';
import { App } from './containers/app'; // hypothetical app component

const randomFactory = () => { return Math.random(); };

@NgModule({
    providers: [ { provide: 'Random', useFactory: randomFactory } ], })

export class DiExample {};
```

In the hypothetical app component, 'Random' could be injected like:

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

@Component({
    selector: 'app-root',
    template: `Random: {{ value }}`
})
export class AppCompoennt {
    value: number;

    constructor(@Inject('Random') r) {
        this.value = r;
    }
}
```

One important note is that 'Random' is in quotes, both in the consumer. This is because as a factory we have no `Random provide` function and in the identifier anywhere to access.

The above example uses Angular's `useFactory` recipe. When Angular is told to `provide` things using `useFactory` , Angular expects the provided value to be a function. Sometimes functions and classes are even more than what's needed. Angular has a "recipe" called `useValue` for these cases that works almost exactly the same:

```
import { NgModule } from '@angular/core';
import { AppComponent } from './containers/app.component'; // hypothetical app compone nt


@NgModule({
    providers: [ { provide: 'Random', useValue: Math.random() } ], })

export class DiExample {};
```

In this case, the product of `Math.random` is assigned to the `useValue` property passed to the `provider` .

Avoiding Injection Collisions: OpaqueToken

Since Angular allows the use of tokens as identifiers to its dependency injection system, one of the potential issues is using the same token to represent different entities. If, for example, the string `'token'` is used to inject an entity, it's possible that something totally unrelated also uses `'token'` to inject a different entity. When it comes time for Angular to resolve one of these entities, it might be resolving the wrong one. This behavior might happen rarely or be easy to resolve when it happens within a small team - but when it comes to multiple teams working separately on the same codebase or 3rd party modules from different sources are integrated these collisions become a bigger issue.

Consider this example where the main application is a consumer of two modules: one that provides an email service and another that provides a logging service.

app/email/email.service.ts

```
export const apiConfig = 'api-config';

@Injectable()
export class EmailService {
    constructor(@Inject(apiConfig) public apiConfig) { }
}
```

app/email/email.module.ts

```
@NgModule({
    providers: [ EmailService ],
})
export class EmailModule { }
```

The email service api requires some configuration settings, identified by the string `api-config` , to be provided by the DI system. This module should be flexible enough so that it can be used by different modules in different applications. This means that those settings should be determined by the application characteristics and therefore provided by the `AppModule` where the `EmailModule` is imported.

app/logger/logger.service.ts

```
export const apiConfig = 'api-config';

@Injectable()
export class LoggerService {
    constructor(@Inject(apiConfig) public apiConfig) { }
}
```

app/logger/logger.module.ts

```
@NgModule({
    providers: [ LoggerService ],
})
export class LoggerModule { }
```

The other service, `LoggerModule` , was created by a different team than the one that created `EmailModule` , and it that also requires a configuration object. Not surprisingly, they decided to use the same token for their configuration object, the string `api-config` . In an effort to avoid a collision between the two tokens with the same name, we could try to rename the imports as shown below. In an effort to avoid a collision between the two tokens with the same name, we could try to rename the imports as shown below.

app/app.module.ts

```
import { apiConfig as emailApiConfig } from './email/index'; import { apiConfig as
loggerApiConfig } from './logger/index';

@NgModule({
    ...
    providers: [
        { provide: emailApiConfig, useValue: { apiKey: 'email-key', context: 'registration' } },

        { provide: loggerApiConfig, useValue: { apiKey: 'logger-key' } },
    ],
    ...
})
export class AppModule { }
```

When the application runs, it encounters a collision problem resulting in both modules getting the same value for their configuration, in this case `{ apiKey: 'logger-key' }` . When it comes time for the main application to specify those settings, Angular overwrites the first `emailApiConfig` value with the `loggerApiConfig` value, since that was provided last. In this

case, module implementation details are leaking out to the parent module. Not only that, those details were obfuscated through the module exports and this can lead to problematic debugging. This is where Angular's OpaqueToken comes into play.

OpaqueToken

OpaqueToken s are unique and immutable values which allow developers to avoid collisions of dependency injection token ids.

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

const name = 'token';
const token1 = new OpaqueToken(name);
const token2 = new OpaqueToken(name);

console.log(token1 === token2); // false
```

Here, regardless of whether or not the same value is passed to the constructor of the token, it will not result in identical symbols.

app/email/email.module.ts

```
export const apiConfig = new OpaqueToken('api-config');

@Injectable()
export class EmailService {
    constructor(@Inject(apiConfig) public apiConfig: EmailConfig) { }
}
```

```
export const apiConfig = new OpaqueToken('api-config');

@Injectable()
export class LoggerService {
    constructor(@Inject(apiConfig) public apiConfig: LoggerConfig) { }
}
```

View Example

After turning the identifying tokens into OpaqueToken s without changing anything else, the collision is avoided. Every service gets the correct configuration object from the root module and Angular is now able to differentiate two tokens that uses the same string.

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.

Consider the following tree, which models a chat application consisting of two open chat windows, and a login/logout widget.

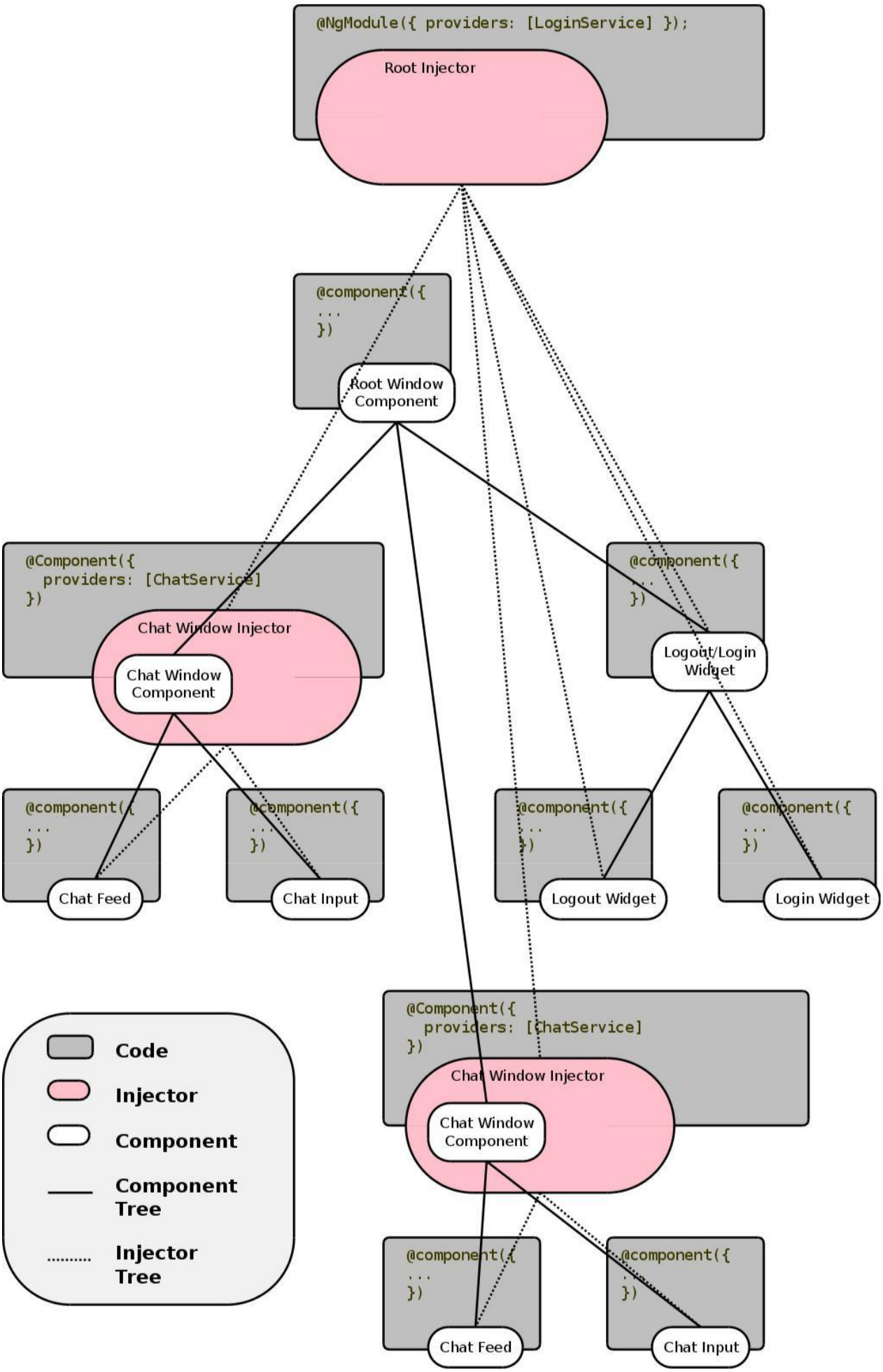


Figure: Image of a Component Tree, and a DI Tree

In the image above, there is one root injector, which is established through `@NgModule`'s `providers` array. There's a `LoginService` registered with the root injector.

Below the root injector is the root `@Component`. This particular component has no `providers` array and will use the root injector for all of its dependencies.

There are also two child injectors, one for each `ChatWindow` component. Each of these components has their own instantiation of a `ChatService`.

There is a third child component, `Logout/Login`, but it has no injector.

There are several grandchild components that have no injectors. There are `ChatFeed` and `ChatInput` components for each `ChatWindow`. There are also `LoginWidget` and `LogoutWidget` components with `Logout/Login` as their parent.

The injector tree does not make a new injector for every component, but does make a new injector for every component with a `providers` array in its decorator. Components that have no `providers` array look to their parent component for an injector. If the parent does not have an injector, it looks up until it reaches the root injector.

Warning: Be careful with `provider` arrays. If a child component is decorated with a `providers` array that contains dependencies that were *also* requested in the parent component(s), the dependencies the child receives will shadow the parent dependencies. This can have all sorts of unintended consequences.

Consider the following example:

app/module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser'; import {
AppComponent } from './app.component';
import { ChildInheritorComponent, ChildOwnInjectorComponent } from './components/index'
;
import { Unique } from './services/unique';


const randomFactory = () => { return Math.random(); };

@NgModule({
    imports: [BrowserModule],
    declarations: [
        AppComponent,
        ChildInheritorComponent,
        ChildOwnInjectorComponent,
    ],
    /** Provide dependencies here */
    providers: [Unique],
    bootstrap: [AppComponent],
})
export class AppModule {}
```

In the example above, `Unique` is bootstrapped into the root injector.

app/services/unique.ts

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

@Injectable()
export class Unique {
    value = (+Date.now()).toString(16) + '.' +
        Math.floor(Math.random() * 500);
}
```

The `Unique` service generates a value unique to *its* instance upon instantiation.

app/components/child-inheritor.component.ts

```
import { Component, Inject } from '@angular/core'; import { Unique }
from '../services/unique';

@Component({
    selector: 'app-child-inheritor',
    template: `<span>{{ value }}</span>`
})
export class ChildInheritorComponent {
    value = this.u.value;

    constructor(private u: Unique) { }
}
```

The child inheritor has no injector. It will traverse the component tree upwards looking for an injector.

app/components/child-own-injector.component.ts

```
import { Component, Inject } from '@angular/core'; import { Unique }
from '../services/unique';

@Component({
    selector: 'child-own-injector',
    template: `<span>{{ value }}</span>`,
    providers: [Unique]
})
export class ChildOwnInjectorComponent {
    value = this.u.value;

    constructor(private u: Unique) { }
}
```

The child own injector component has an injector that is populated with its own instance of `Unique`. This component will not share the same value as the root injector's `Unique` instance.

app/containers/app.ts

```
@Component({
    selector: 'app-root',
    template: `
        <p>
            App's Unique dependency has a value of {{ value }}
        </p>
        <p>
            which should match
        </p>
        <p>
            ChildInheritor's value:
            <app-child-inheritor></app-child-inheritor>
        </p>
        <p>
            However,
        </p>
        <p>
            ChildOwnInjector should have its own value:
            <app-child-own-injector></app-child-own-injector>
        </p>
        <p>
            ChildOwnInjector's other instance should also have its own value:
            <app-child-own-injector></app-child-own-injector>
        </p>`,
})
export class AppComponent {
    value: number = this.u.value;

    constructor(private u: Unique) { }
}
```

HTTP

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

Setting up angular/common/http

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

```
...
import { AppComponent } from './app.component'
import { HttpClientModule } from '@angular/common/http';

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

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 { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';

@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');
    }
}
```

Here we are making an HTTP GET request which is exposed to us as an observable.

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

Catching Rejections

To catch rejections we use the subscriber's `error` and `complete` callbacks.

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

@Injectable()
export class AuthService {

    constructor(private http: Http) {}

    login(username, password) {
        const payload = {
            username: username,
            password: password
        };

        this.http.post(`${ BASE_URL }/auth/login`, payload)
            .subscribe(
                authData => this.storeToken(authData.id_token),
                (err) => console.error(err),
```

```
            () => console.log('Authentication Complete')
        );
    }
}
```

Catch and Release

We also have the option of using the `.catch` operator. It allows us to catch errors on an existing stream, do something, and pass the exception onwards.

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

@Injectable()
export class SearchService {

    constructor(private http: Http) {}

    search(term: string) {
        return this.http.get('https://api.spotify.com/v1/dsds?q=' + term + '&type=artist')
            .pipe(catchError((e) => {
                return Observable.throw(
                    new Error(`${ e.status } ${ e.statusText }`)
                );
            }));
    }
}
```

It also allows us to inspect the error and decide which route to take. For example, if we encounter a server error then use a cached version of the request otherwise re-throw.

```
@Injectable()
export class SearchService {

    ...

    search(term: string) {
        return this.http.get(`https://api.spotify.com/v1/dsds?q=${term}&type=artist`)
            .pipe(catchError(e => {
                if (e.status >==              500) {
                    return cachedVersion();
                } else {
                    return Observable.throw(
                        new Error(`${ e.status } ${ e.statusText }`) );

                }
            }));
    }
```

Cancel a Request

Cancelling an HTTP request is a common requirement. For example, you could have a queue of requests where a new request supersedes a pending request and that pending request needs to be cancelled.

To cancel a request we call the `unsubscribe` function of its subscription.

```
@Component({ /* ... */ })
export class AppComponent {
  /* ... */

  search() {
    const request = this.searchService.search(this.searchField.value)
      .subscribe(
        result => { this.result = result.artists.items; },
        err => { this.errorMessage = err.message; }, () => {
        console.log('Completed'); }
      );

    request.unsubscribe();
  }
}
```

82

Retry

There are times when you might want to retry a failed request. For example, if the the user is offline you might want to retry a few times or indefinitely.

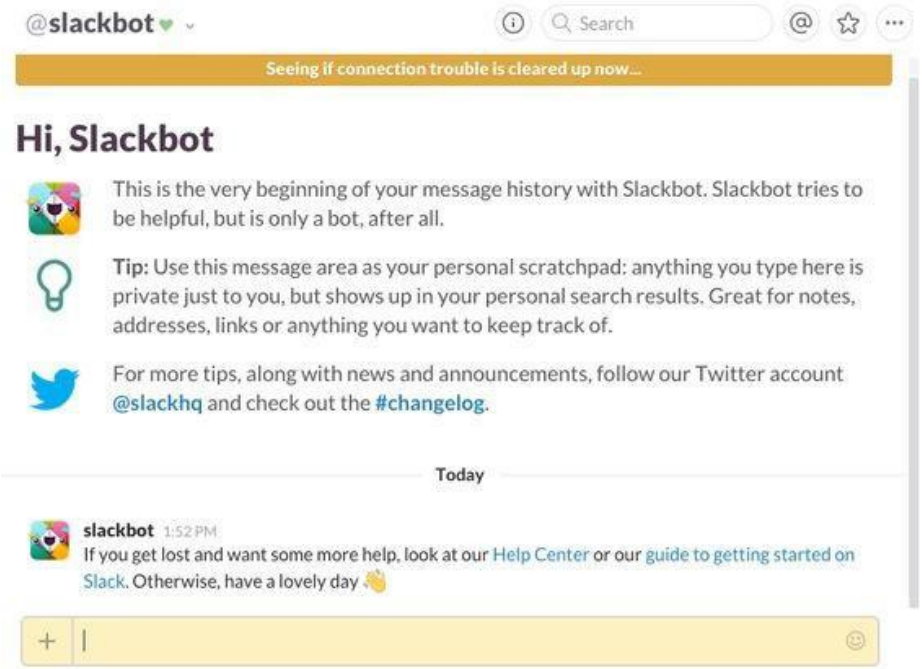


Figure: Retry example from Slack

Use the RxJS `retry` operator. It accepts a `retryCount` argument. If not provided, it will retry the sequence indefinitely.

Note that the error callback is not invoked during the retry phase. If the request fails it will be retried and only after all the retry attempts fail the stream throws an error.

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

@Injectable()
export class SearchService {

    constructor(private http: Http) {}

    search(term: string) {
        let tryCount = 0;
        return this.http.get('https://api.spotify.com/v1/dsds?q=' + term + '&type=artist')
            .pipe(retry(3));
    }
}
```

Combining Streams with `flatMap`

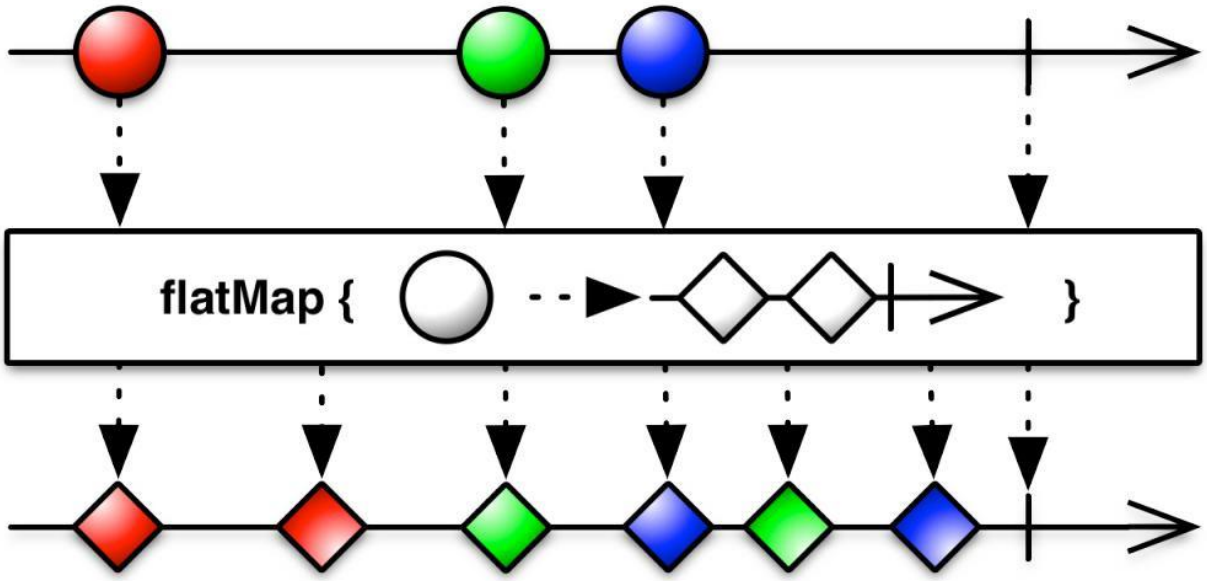


Figure: FlatMap created by ReactiveX licensed under CC-3 (http://reactivex.io/documentation/operators/flatmap.html)

A case for FlatMap:

- A simple observable stream
- A stream of arrays
- Filter the items from each event
- Stream of filtered items
- Filter + map simplified with flatMap

Let's say we wanted to implement an AJAX search feature in which every keypress in a text field will automatically perform a search and update the page with the results. How would this look? Well we would have an `Observable` subscribed to events coming from an input field, and on every change of input we want to perform some HTTP request, which is also an `Observable` we subscribe to. What we end up with is an `Observable` of an `Observable` .

By using `flatMap` we can transform our event stream (the keypress events on the text field) into our response stream (the search results from the HTTP request).

app/services/search.service.ts

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

@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')
    }
}
```

Here we have a basic service that will undergo a search query to Spotify by performing a get request with a supplied search term. This `search` function returns an `Observable` that has had some basic post-processing done (turning the response into a JSON object).

OK, let's take a look at the component that will be using this service.

app/app.component.ts

```
import { Component } from '@angular/core';
import { FormControl,
        FormGroup,
        FormBuilder } from '@angular/forms';
import { SearchService } from './services/search.service'; import 'rxjs/Rx';


@Component({
        selector: 'app-root',
        template: `
                <form [formGroup]="coolForm"><input formControlName="search" placeholder="Sear ch Spotify
artist"></form>

                <div *ngFor="let artist of result">
                    {{artist.name}}
                </div>
        `
})

export class AppComponent {
        searchField: FormControl;
        coolForm: FormGroup;

        constructor(private searchService:SearchService, private fb:FormBuilder) { this.searchField = new
                FormControl();
                this.coolForm = fb.group({search: this.searchField});

                this.searchField.valueChanges
                    .pipe(
                        debounceTime(400),
```

```
            flatMap(term => this.searchService.search(term)))
            .subscribe((result) => {
                this.result = result.artists.items
            });
    }
}
```

Here we have set up a basic form with a single field, `search`, which we subscribe to for event changes. We've also set up a simple binding for any results coming from the `SearchService`. The real magic here is `flatMap` which allows us to flatten our two separate subscribed `Observables` into a single cohesive stream we can use to control events coming from user input and from server responses.

Note that flatMap flattens a stream of `Observables` (i.e `Observable` of `Observables`) to a stream of emitted values (a simple `Observable`), by emitting on the "trunk" stream everything that will be emitted on "branch" streams.

Enhancing Search with `switchMap`

There is a problem with our previous implementation of incremental search.

What if the server, for some reason, takes a very long time to respond to a particular query? If we use `flatMap`, we run the risk of getting results back from the server in the wrong order. Let's illustrate this with an example.

What is `switchMap`?

`switchMap` is very similar to `flatMap`, but with a very important distinction. Any events to be merged into the trunk stream are ignored if a new event comes in. Here is a marble diagram showing the behavior of `switchMap`:

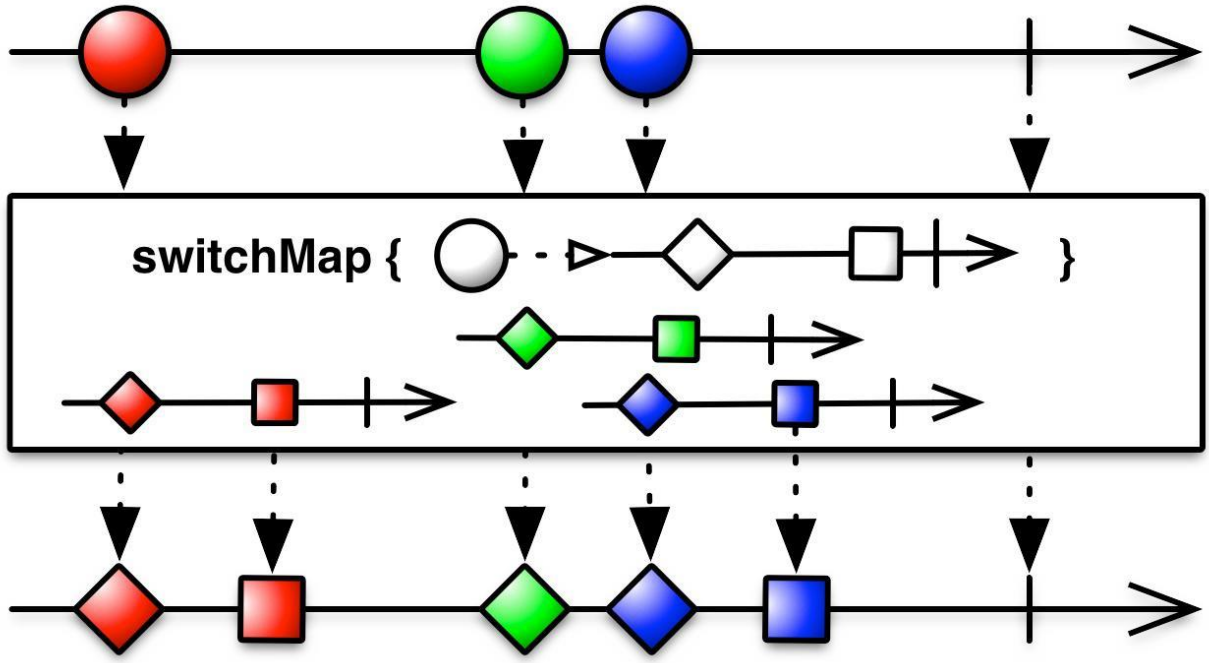


Figure: SwitchMap created by ReactiveX licensed under CC-3 (http://reactivex.io/documentation/operators/flatmap.html)

In short, every time an event comes down the stream, `flatMap` will subscribe to (and invoke) a new observable without unsubscribing from any other observable created by a previous event. `switchMap` on the other hand will automatically unsubscribe from any previous observable when a new event comes down the stream.

In the diagram above, the round "marbles" represent events in the originating stream. In the resulting stream, "diamonds" mark the creation (and subscription) of an inner observable (that is eventually merged onto the trunk stream) and "squares" represent values emitted from that same inner observable.

Just like `flatMap` , the red marble gets replaced with a red diamond and a subsequent red square. The interaction between the green and blue marble events are more interesting. Note that the green marble gets mapped to a green diamond immediately. And if enough time had passed, a green square would be pushed into the trunk stream but we do not see that here.

Before the green square event is able to happen, a blue marble comes through and gets mapped to a blue diamond. What happened is that the green square is now ignored and do not get merged back into the trunk stream. The behavior of `switchMap` can be likened to a `flatMap` that "switches" to the more immediate incoming event and ignores all previously created event streams.

In our case, because the blue marble event happened very quickly after the green marble, we "switched" over to focus on dealing with the blue marble instead. This behavior is what will prevent the problem we described above.

If we apply `switchMap` to the above example, the response for `ABC` would be ignored and the suggestions for `ABCX` would remain.

Enhanced Search with `switchMap`

Here is the revised component using `switchMap` instead of `flatMap`.

app/app.component.ts

```
import { Component } from '@angular/core';
import { FormControl,
        FormGroup,
        FormBuilder } from '@angular/forms';
import { SearchService } from './services/search.service'; import 'rxjs/Rx';


@Component({
        selector: 'app-root',
        template: `
                <form [formGroup]="coolForm"><input formControlName="search" placeholder="Sear ch Spotify
artist"></form>

                <div *ngFor="let artist of result">
                   {{artist.name}}
                </div>
        `
})

export class AppComponent {
        searchField: FormControl;
        coolForm: FormGroup;

        constructor(private searchService:SearchService, private fb:FormBuilder) { this.searchField = new
                FormControl();
                this.coolForm = fb.group({search: this.searchField});

                this.searchField.valueChanges
                   .pipe(debounceTime(400),
                      switchMap(term => this.searchService.search(term)))
                      .subscribe((result) => {
                            this.result = result.artists.items
                      });
        }
}
```

This implementation of incremental search with switchMap is more robust than the one we saw on the previous page with flatMap . The suggestions that the user sees will always eventually reflect the last thing the user typed. Thanks to this, we can guarantee a great user experience regardless of how the server responds.

Requests as Promises

The observable returned by Angular http client can be converted it into a promise.

> We recommend using observables over promises. By converting to a promise you will be lose the ability to cancel a request and the ability to chain RxJS operators.

```
import { Http } from '@angular/http';
import { Injectable } from '@angular/core';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/toPromise';

@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`).toPromise();

    }
}
```

We would then consume it as a regular promise in the component.

```
@Component({ /* ... */ })
export class AppComponent {
        /* ... */

        search() {
                this.searchService.search(this.searchField.value)
                    .then((result) => {
                this.result = result.artists.items;
            })
            .catch((error) => console.error(error));
        }
}
```

Pipes

Figure: Pipes by Life-Of-Pix is licensed under Public Domain (https://pixabay.com/en/pipe-plumbing-connection-pipeline-406906/)

Angular 6 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 6 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:

```
pipeName: parameterValue
```

You can also pass multiple parameters this way:

```
pipeName: parameter1: parameter2
```

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

@Component({
        selector: 'app-root',
        template: '<p>Total price of product is {{ price | currency: "CAD": true: "1.2-4" }}</p>'

})
export class AppComponent {
    price = 100.123456;
}
```

Chaining Pipes

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

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

@Component({
        selector: 'app-root',
        template: '<p>Total price of product is {{ price | currency: "CAD": true: "1.2-4"
|    lowercase }}</p>' })

export class ProductPrice { price =
    100.123456;
}
```

Custom Pipes

Angular allows you to create your own custom pipes:

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

const FILE_SIZE_UNITS = ['B', 'KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'];
const FILE_SIZE_UNITS_LONG = ['Bytes', 'Kilobytes', 'Megabytes', 'Gigabytes', 'Pettaby tes', 'Exabytes', 'Zettabytes',
'Yottabytes'];

@Pipe({
    name: 'formatFileSize'
})
export class FormatFileSizePipe implements PipeTransform {
    transform(sizeInBytes: number, longForm: boolean): string {
        const units = longForm
            ?   FILE_SIZE_UNITS_LONG :
            FILE_SIZE_UNITS;

        let power = Math.round(Math.log(sizeInBytes) / Math.log(1024)); power =
        Math.min(power, units.length - 1);

        const size = sizeInBytes / Math.pow(1024, power); // size in new units const formattedSize =
        Math.round(size * 100) / 100; // keep up to 2 decimals const unit = units[power];


        return `${formattedSize} ${unit}`;
    }
}
```

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.

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

@Component({
  selector: 'app-root',
  template: `
    <div>
      <p *ngFor="let f of fileSizes">{{ f | formatFileSize }}</p> <p>{{ largeFileSize |
      formatFileSize:true }}</p>
    </div>`
})
export class AppComponent {
  fileSizes = [10, 100, 1000, 10000, 100000, 10000000, 10000000000]; largeFileSize =
  Math.pow(10, 15)
}
```

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

  we used and the length pipe we created are examples of a stateless

- *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.

```
@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))
}
```

Implementing Stateful Pipes

Pipes are stateless by default. We must declare a pipe to be stateful by setting the pure property of the @Pipe decorator to false. This setting tells Angular's change detection system to check the output of this pipe each cycle, whether its input has changed or not.

```
//    naive implementation assumes small number increments

@Pipe({
   name: 'animateNumber',
   pure: false })

export class AnimateNumberPipe implements PipeTransform { private
   currentNumber: number = null; // intermediary number private targetNumber:
   number = null;

   transform(targetNumber: number): string {
      if (targetNumber !== this.targetNumber) {
         this.currentNumber = this.targetNumber || targetNumber; this.targetNumber
         = targetNumber;

         const difference = this.targetNumber - this.currentNumber

         Observable.interval(100)
            .take(difference)
            .subscribe(() => {
               this.currentNumber++;
```

```
            })
        }

        return this.currentNumber;
    }
}
```

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 6 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

Getting Started

Opt-In APIs

Before we dive into any of the form features, we need to do a little bit of housekeeping. We need to bootstrap our application using the FormsModule or ReactiveFormsModule .

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic' import { FormsModule
} from '@angular/forms'; import { AppComponent } from './components'


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

platformBrowserDynamic().bootstrapModule(AppModule)
```

Input Labeling

Most of the form examples use the following HTML5 style for labeling inputs:

```
<label for="name">Name</label>
<input type="text" name="username" id="name">
```

Angular also supports the alternate HTML5 style, which precludes the necessity of `id` s on `<input>` s:

```
<label>
    Name
    <input type="text" name="username">
</label>
```

Template-Driven Forms

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

First, consider a typical form:

```
<form method="POST" action="/register" id="signup-form"> <label
    for="email">Email</label>
    <input type="text" name="email" id="email">

    <label for="password">Password</label>
    <input type="password" name="password" id="password">

    <button type="submit">Sign Up</button>
</form>
```

Angular has already provided you a `form` directive, and form related directives such as input, etc which operates under the covers. For a basic implementation, we just have to add a few attributes and make sure our component knows what to do with the data.

index.html

```
<signup-form>Loading...</signup-form>
```

signup-form.component.html

```
<form #signupForm="ngForm" (ngSubmit)="registerUser(signupForm)"> <label
    for="email">Email</label>
    <input type="text" name="email" id="email" ngModel>

    <label for="password">Password</label>
    <input type="password" name="password" id="password" ngModel>

    <button type="submit">Sign Up</button>
</form>
```

signup-form.component.ts

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

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

      export class SignupFormComponent {
          registerUser(form: NgForm) {
              console.log(form.value);
        //    {email: '...', password: '...'}
        //    ...
    }
}
```

Nesting Form Data

If you find yourself wrestling to fit nested trees of data inside of a flat form, Angular has you covered for both simple and complex cases.

Let's assume you had a payment endpoint which required data, similar to the following:

```
{
    "contact": {
        "firstname": "Bob",
        "lastname": "McKenzie",
        "email": "BobAndDoug@GreatWhiteNorth.com",
        "phone": "555-TAKE-OFF"
    },
    "address": {
        "street": "123 Some St",
        "city": "Toronto",
        "region": "ON",
        "country": "CA",
        "code": "H0H 0H0"
    },
    "paymentCard": {
        "provider": "Credit Lending Company Inc",
        "cardholder": "Doug McKenzie",
        "number": "123 456 789 012",
        "verification": "321",
        "expiry": "2020-02"
    }
}
```

While forms are flat and one-dimensional, the data built from them is not. This leads to complex transforms to convert the data you've been given into the shape you need.

Worse, in cases where it is possible to run into naming collisions in form inputs, you might find yourself using long and awkward names for semantic purposes.

```
<form>
  <fieldset>
    <legend>Contact</legend>

    <label for="contact_first-name">First Name</label>
    <input type="text" name="contact_first-name" id="contact_first-name">

    <label for="contact_last-name">Last Name</label>
    <input type="text" name="contact_last-name" id="contact_last-name">

    <label for="contact_email">Email</label>
    <input type="email" name="contact_email" id="contact_email">

    <label for="contact_phone">Phone</label>
    <input type="text" name="contact_phone" id="contact_phone">
  </fieldset>

  <!-- ... -->

</form>
```

A form handler would have to convert that data into a form that your API expects. Thankfully, this is something Angular has a solution for.

ngModelGroup

When building a template-driven form in Angular, we can lean on the ngModelGroup directive to arrive at a cleaner implementation, while Angular does the heavy lifting of converting form-fields into nested data.

```
<form #paymentForm="ngForm" (ngSubmit)="purchase(paymentForm)"> <fieldset
    ngModelGroup="contact">
        <legend>Contact</legend>

        <label>
            First Name <input type="text" name="firstname" ngModel>
        </label>
        <label>
            Last Name <input type="text" name="lastname" ngModel>
        </label>
        <label>
            Email <input type="email" name="email" ngModel>
        </label>
        <label>
            Phone <input type="text" name="phone" ngModel> </label>

    </fieldset>

    <fieldset ngModelGroup="address">
        <!-- ... -->
    </fieldset>

    <fieldset ngModelGroup="paymentCard">
        <!-- ... -->
    </fieldset>
</form>
```

> - Using the alternative HTML5 labeling format; IDs have no bearing on the `ngForm` / `ngModel` paradigm
> - Aside from semantic purposes, `ngModelGroup` does not have to be used on `<fieldset>` — it would work just as well on a `<div>` .

If we were to fill out the form, it would end up in the shape we need for our API, and we can still rely on the HTML field validation if we know it’s available.

Using Template Model Binding

One-Way Binding

If you need a form with default values, you can start using the value-binding syntax for ngModel.

app/signup-form.component.html

```
<form #signupForm="ngForm" (ngSubmit)="register(signupForm)"> <label
    for="username">Username</label>
    <input type="text" name="username" id="username" [ngModel]="generatedUser">

    <label for="email">Email</label>
    <input type="email" name="email" id="email" ngModel>

    <button type="submit">Sign Up</button>
</form>
```

app/signup-form.component.ts

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

@Component({
    //     ...
})
export class SignupFormComponent { generatedUser: string =
    generateUniqueUserID();

    register(form: NgForm) {
        console.log(form.value);
        // ...
    }
}
```

Two-Way Binding

While Angular assumes one-way binding by default, two-way binding is still available if you need it.

In order to have access to two-way binding in template-driven forms, use the “Banana-Box” syntax (`[(ngModel)]="propertyName"`).

Be sure to declare all of the properties you will need on the component.

```
<form #signupForm="ngForm" (ngSubmit)="register(signupForm)"> <label
    for="username">Username</label>
    <input type="text" name="username" id="username" [(ngModel)]="username">

    <label for="email">Email</label>
    <input type="email" name="email" id="email" [(ngModel)]="email">

    <button type="submit">Sign Up</button>
</form>
```

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

@Component({
    //    ...
})
export class SignUpFormComponent { username: string =
    generateUniqueUserID(); email = '';


    register(form: NgForm) {
        console.log(form.value.username);
        console.log(this.username);
        // ...
    }
}
```

Validating Template-Driven Forms

Validation

Using the template-driven approach, form validation is a matter of following HTML5 practices:

```
<!-- a required field -->
<input type="text" required>

<!-- an optional field of a specific length --> <input type="text"
pattern=".{3,8}">

<!-- a non-optional field of specific length --> <input type="text"
pattern=".{3,8}" required>

<!-- alphanumeric field of specific length --> <input type="text"
pattern="[A-Za-z0-9]{0,5}">
```

Note that the `pattern` attribute is a less-powerful version of JavaScript's RegEx syntax.

There are other HTML5 attributes which can be learned and applied to various types of input; however in most cases they act as upper and lower limits, preventing extra information from being added or removed.

```
<!-- a field which will accept no more than 5 characters -->
<input type="text" maxlength="5">
```

You can use one or both of these methods when writing a template-driven form. Focus on the user experience: in some cases, it makes sense to prevent accidental entry, and in others it makes sense to allow unrestricted entry but provide something like a counter to show limitations.

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`

In our case, to build a login form, we're looking at something like the following:

app/login-form.component.ts

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, FormBuilder } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: 'app/app.component.html'
})
export class AppComponent {
```

```
    username = new FormControl('')
    password = new FormControl('')

    loginForm: FormGroup = this.builder.group({
      username: this.username,
      password: this.password
    });

    constructor(private builder: FormBuilder) { }

    login() {
      console.log(this.loginForm.value);
      // Attempt Logging in...
    }
}
```

app/login-form.component.html

```
<form [formGroup]="loginForm" (ngSubmit)="login()"> <label
    for="username">username</label>
    <input type="text" name="username" id="username" [formControl]="username"> <br>


    <label for="password">password</label>
    <input type="password" name="password" id="password" [formControl]="password"> <br>

    <button type="submit">log in</button>
</form>
```

FormControl

Note that the `FormControl` class is assigned to similarly named fields, both on `this` and in the `FormBuilder#group({ })` method. This is mostly for ease of access. By saving references to the `FormControl` instances on `this`, you can access the inputs in the template without having to reference the form itself. The form fields can otherwise be reached in the template by using `loginForm.controls.username` and `loginForm.controls.password`. Likewise, any instance of `FormControl` in this situation can access its parent group by using its `.root` property (e.g. `username.root.controls.password`).

> Make sure that `root` and `controls` exist before they're used.

A `FormControl` requires two properties: an initial value and a list of validators. Right now, we have no validation. This will be added in the next steps.

Validating Reactive Forms

Building from the previous login form, we can quickly and easily add validation.

Angular provides many validators out of the box. They can be imported along with the rest of dependencies for procedural forms.

app/login-form.component.ts

```
import { Component } from '@angular/core';
import { Validators, FormBuilder, FormControl } from '@angular/forms';

@Component({
    //     ...
})
export class AppComponent { username =
    new FormControl('', [
        Validators.required,
        Validators.minLength(5)
    ]);

    password = new FormControl('', [Validators.required]);

    loginForm: FormGroup = this.builder.group({
        username: this.username,
        password: this.password
    });

    constructor(private builder: FormBuilder) { }

    login () {
        console.log(this.loginForm.value);
        // Attempt Logging in...
    }
}
```

app/login-form.component.html

```
<form [formGroup]="loginForm" (ngSubmit)="login()">

    <div>
        <label for="username">username</label>

        <input
            type="text"
            name="username"
            id="username"
            [formControl]="username">

        <div [hidden]="username.valid || username.untouched"> <div>

            The following problems have been found with the username:
```

```
        </div>

          <div [hidden]="!username.hasError('minlength')"> Username can
                                                not be shorter than 5 characters.
        </div>
        <div [hidden]="!username.hasError('required')"> Username is
            required.
        </div>
    </div>
</div>
<div >
    <label for="password">password</label>
    <input
        type="password"
        name="password"
        id="password" [formControl]="password">

    <div [hidden]="password.valid || password.untouched"> <div>

            The following problems have been found with the password:
        </div>

        <div [hidden]="!password.hasError('required')"> The password is
            required.
        </div>
    </div>
</div>

<button type="submit" [disabled]="!loginForm.valid">Log In</button> </form>
```

Note that we have added rather robust validation on both the fields and the form itself, using nothing more than built-in validators and some template logic.

We are using `.valid` and `.untouched` to determine if we need to show errors - while the field is required, there is no reason to tell the user that the value is wrong if the field hasn't been visited yet.

For built-in validation, we are calling `.hasError()` on the form element, and we are passing a string which represents the validator function we included. The error message only displays if this test returns true.

Reactive Forms Custom Validation

As useful as the built-in validators are, it is very useful to be able to include your own. Angular allows you to do just that, with minimal effort.

Let's assume we are using the same Login Form, but now we also want to test that our password has an exclamation mark somewhere in it.

app/login-form.component.ts

```
function hasExclamationMark(input: FormControl) { const hasExclamation =
    input.value.indexOf('!') >= 0; return hasExclamation ? null : { needsExclamation:
    true };
}

password = new FormControl('', [
    Validators.required,
    hasExclamationMark
]);
```

A simple function takes the `FormControl` instance and returns null if everything is fine. If the test fails, it returns an object with an arbitrarily named property. The property name is what will be used for the `.hasError()` test.

app/login-form.component.ts

```
<!-- ... -->
<div [hidden]="!password.hasError('needsExclamation')">
    Your password must have an exclamation mark!
</div>
<!-- ... -->
```

Predefined Parameters

Having a custom validator to check for exclamation marks might be helpful, but what if you need to check for some other form of punctuation? It might be overkill to write nearly the same thing over and over again.

Consider the earlier example `Validators.minLength(5)` . How did they get away with allowing an argument to control the length, if a validator is just a function? Simple, really. It's not a trick of Angular, or TypeScript - it's simple JavaScript closures.

```
function minLength(minimum) {
    return function(input) {
        return input.value.length >= minimum ? null : { minLength: true };
    };
}
```

Assume you have a function which takes a "minimum" parameter and returns another function. The function defined and returned from the inside becomes the validator. The closure reference allows you to remember the value of the minimum when the validator is eventually called.

Let's apply that thinking back to a `PunctuationValidator`.

app/login-form.component.ts

```
function hasPunctuation(punctuation: string, errorType: string) {
    return function(input: FormControl) {
        return input.value.indexOf(punctuation) >= 0 ?
                null :
                { [errorType]: true };
    };
}

// ...

password = new FormControl('', [
    Validators.required,
    hasPunctuation('&', 'ampersandRequired')
]);
```

app/login-form.component.html

```
<!-- ... -->
<div [hidden]="!password.hasError('ampersandRequired')">
    You must have an &amp;amp; in your password.
</div>
<!-- ... -->
```

Validating Inputs Using Other Inputs

Keep in mind what was mentioned earlier: inputs have access to their parent context via `.root`. Therefore, complex validation can happen by drilling through the form, via root.

```
function duplicatePassword(input: FormControl) { if (!input.root ||
    !input.root.controls) {
        return null;
    }

    const exactMatch = input.root.controls.password === input.value; return exactMatch ?
    null : { mismatchedPassword: true };
}

// ...

this.duplicatePassword = new FormControl('', [
    Validators.required,
    duplicatePassword
]);
```

Visual Cues for Users

HTML5 provides `:invalid` and `:valid` pseudo-selectors for its input fields.

```
                input[type="text"]:valid {
                border: 2px solid green;
}

input[type="text"]:invalid {
    border: 2px solid red;
}
```

Unfortunately, this system is rather unsophisticated and would require more manual effort in order to work with complex forms or user behavior.

Rather than writing extra code, and creating and enforcing your own CSS classes, to manage these behaviors, Angular provides you with several classes, already accessible on your inputs.

```
/* field value is valid */
.ng-valid {}

/* field value is invalid */
.ng-invalid {}

/* field has not been clicked in, tapped on, or tabbed over */
.ng-untouched {}

/* field has been previously entered */
.ng-touched {}

/* field value is unchanged from the default value */
.ng-pristine {}

/* field value has been modified from the default */
```

```
.ng-dirty {}
```

Note the three pairs:

- valid / invalid
- untouched / touched
- pristine / dirty

These pairs can be used in many combinations in your CSS to change style based on the three separate flags they represent. Angular will toggle between the pairs on each input as the state of the input changes.

```
/* field has been unvisited and unchanged */
input.ng-untouched.ng-pristine {}

/* field has been previously visited, and is invalid */ input.ng-touched.ng-
invalid {}
```

> `.ng-untouched` will not be replaced by `.ng-touched` until the user *leaves* the input for the first time

For templating purposes, Angular also gives you access to the unprefixed properties on the input, in both code and template:

```
<input name="myInput" [formControl]="myCustomInput">
<div [hidden]="myCustomInput.pristine">I've been changed</div>
```

Modules

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

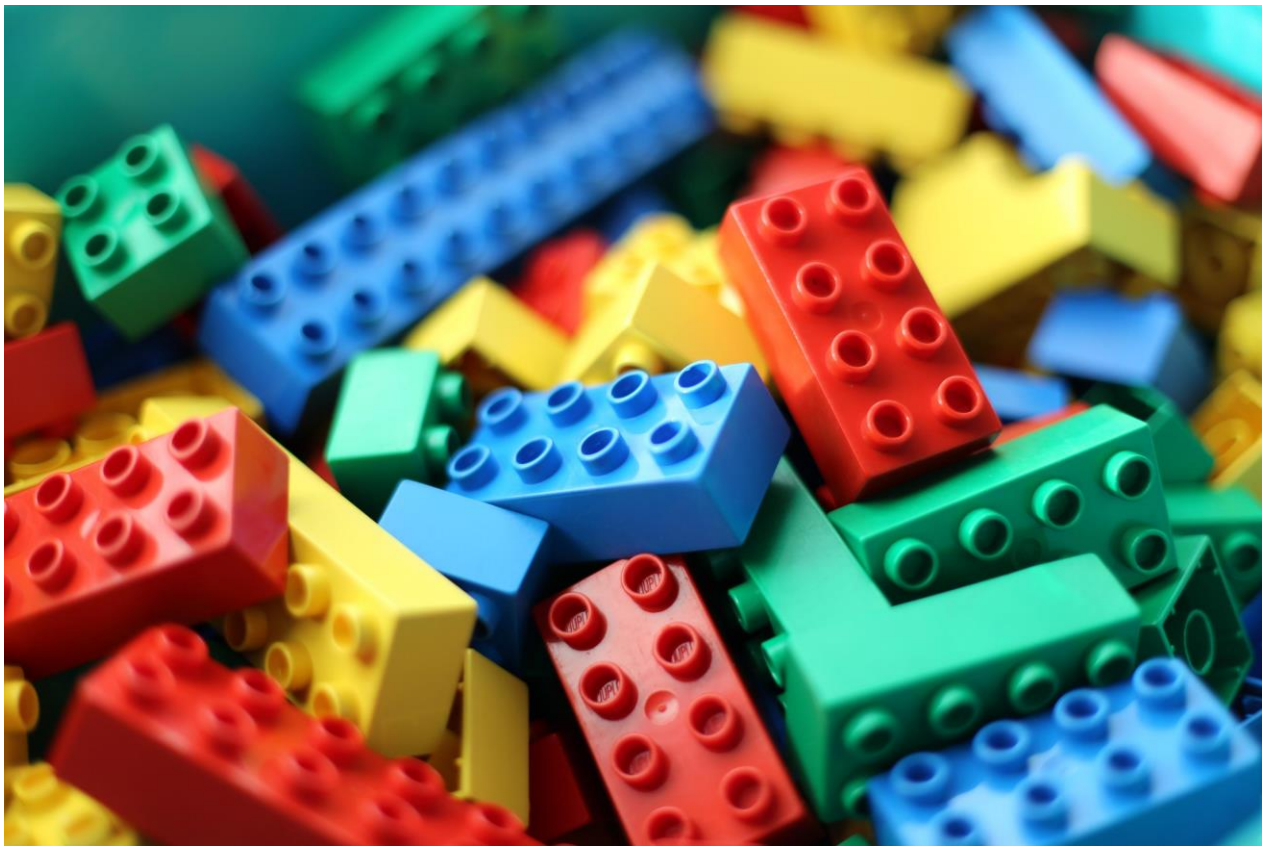


Figure: Used Lego Duplo Bricks by Arto Alanenpää is licensed under CC BY-SA 4.0 (https://commons.wikimedia.org/wiki/File:Lego_dublo_arto_alanenpaa_5.JPG)

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`.

The property `imports` expects an array of modules. Here's where we define the pieces of our puzzle (our application). The property `declarations` expects an array of components, directives and pipes that are part of the module. The `bootstrap` property is where we define the root component of our module. Even though this property is also an array, 99% of the time we are going to define only one component.

> There are very special circumstances where more than one component may be required to bootstrap a module but we are not going to cover those edge cases here.

Here's how a basic module made up of just one component would look like:

app/app.component.ts

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

@Component({
    selector: 'app-root',
    template: '<h1>My Angular App</h1>'
})
export class AppComponent {}
```

app/app.module.ts

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

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

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

The file *app.component.ts* is just a "hello world" component, nothing interesting there. In the other hand, the file *app.module.ts* is following the structure that we've seen before for defining a module but in this case, we are defining the modules and components that we are going to be using.

The first thing that we notice is that our module is importing the `BrowserModule` as an explicit dependency. The `BrowserModule` is a built-in module that exports basic directives, pipes and services. Unlike previous versions of Angular, we have to explicitly import those dependencies to be able to use directives like `*ngFor` or `*ngIf` in our templates.

Given that the root (and only) component of our module is the `AppComponent` we have to list it in the `bootstrap` array. Because in the `declarations` property we are supposed to define **all** the components or pipes that make up our application, we have to define the `AppComponent` again there too.

Before moving on, there's an important clarification to make. **There are two types of modules, root modules and feature modules**.

In the same way that in a module we have one root component and many possible secondary components, **in an application we only have one root module and zero or many feature modules**. To be able to bootstrap our application, Angular needs to know which one is the root module. An easy way to identify a root module is by looking at the `imports` property of its `NgModule` decorator. If the module is importing the `BrowserModule` then it's a root module, if instead is importing the `CommonModule` then it is a feature module.

> As developers, we need to take care of importing the `BrowserModule` in the root module and instead, import the `CommonModule` in any other module we create for the same application. Failing to do so might result in problems when working with lazy loaded modules as we are going to see in following sections.

By convention, the root module should always be named `AppModule`.

Bootstrapping an Application

To bootstrap our module based application, we need to inform Angular which one is our root module to perform the compilation in the browser. This compilation in the browser is also known as "Just in Time" (JIT) compilation.

main.ts

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

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

> It is also possible to perform the compilation as a build step of our workflow. This method is called "Ahead of Time" (AOT) compilation and will require a slightly different bootstrap process that we are going to discuss in another section.

In the next section we are going to see how to create a module with multiple components, services and pipes.

Adding Components, Pipes and Services to a Module

In the previous section, we learned how to create a module with just one component but we know that is hardly the case. Our modules are usually made up of multiple components, services, directives and pipes. In this chapter we are going to extend the example we had before with a custom component, pipe and service.

Let's start by defining a new component that we are going to use to show credit card information.

credit-card.component.ts

```
import { Component, OnInit } from '@angular/core';
import { CreditCardService } from './credit-card.service';

@Component({
    selector: 'app-credit-card',
    template: `
        <p>Your credit card is: {{ creditCardNumber | creditCardMask }}</p>
    `
})
export class CreditCardComponent implements OnInit {
    creditCardNumber: string;

    constructor(private creditCardService: CreditCardService) {}

    ngOnInit() {
        this.creditCardNumber = this.creditCardService.getCreditCard();
    }
}
```

This component is relying on the `CreditCardService` to get the credit card number, and on the pipe `creditCardMask` to mask the number except the last 4 digits that are going to be visible.

credit-card.service.ts

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

@Injectable()
export class CreditCardService {
    getCreditCard(): string {
        return '2131313133123174098';
    }
}
```

credit-card-mask.pipe.ts

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

@Pipe({
    name: 'creditCardMask'
})
export class CreditCardMaskPipe implements PipeTransform {
    transform(plainCreditCard: string): string {
        const visibleDigits = 4;
        let maskedSection = plainCreditCard.slice(0, -visibleDigits); let visibleSection =
        plainCreditCard.slice(-visibleDigits); return maskedSection.replace(/./g, '*') +
        visibleSection;
    }
}
```

With everything in place, we can now use the `CreditCardComponent` in our root component.

app.component.ts

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

@Component({
    selector: 'app-root',
    template: `
        <h1>My Angular App</h1>
        <app-credit-card></app-credit-card>
    `
})
export class AppComponent {}
```

Of course, to be able to use this new component, pipe and service, we need to update our module, otherwise Angular is not going to be able to compile our application.

app.module.ts

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

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

import { CreditCardMaskPipe } from './credit-card-mask.pipe'; import {
CreditCardService } from './credit-card.service'; import { CreditCardComponent } from
'./credit-card.component';

@NgModule({
    imports: [BrowserModule],
    providers: [CreditCardService],
    declarations: [
        AppComponent,
        CreditCardMaskPipe,
```

```
        CreditCardComponent
    ],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Notice that we have added the component `CreditCardComponent` and the pipe `CreditCardMaskPipe` to the `declarations` property, along with the root component of the module `AppComponent` . In the other hand, our custom service is configured with the dependency injection system with the `providers` property.

Be aware that this method of defining a service in the `providers` property **should only be used in the root module**. Doing this in a feature module is going to cause unintended consequences when working with lazy loaded modules.

In the next section, we are going to see how to safely define services in feature modules.

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".

In our previous example, we started to see that. Our root module has a component, a pipe and a service that its only purpose is to deal with credit cards. What if we extract these three elements to their own **feature module** and then we import it into our **root module**?

We are going to do just that. The first step is to create two folders to differentiate the elements that belong to the root module from the elements that belong to the feature module.

```
.
├── 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*. Let's focus on the latter first.

credit-card/credit-card.module.ts

```
import { NgModule } from '@angular/core'; import {
CommonModule } from '@angular/common';

import { CreditCardMaskPipe } from './credit-card-mask.pipe'; import {
CreditCardService } from './credit-card.service'; import { CreditCardComponent } from
'./credit-card.component';

@NgModule({
    imports: [CommonModule],
    declarations: [
        CreditCardMaskPipe,
        CreditCardComponent
    ],
    providers: [CreditCardService],
    exports: [CreditCardComponent]
})
export class CreditCardModule {}
```

Our feature `CreditCardModule` it's pretty similar to the root `AppModule` with a few important differences:

- We are not importing the `BrowserModule` but the `CommonModule` . If we see the documentation of the `BrowserModule` here, we can see that it's re-exporting the `CommonModule` with a lot of other services that helps with rendering an Angular application in the browser. These services are coupling our root module with a particular platform (the browser), but we want our feature modules to be platform independent. That's why we only import the `CommonModule` there, which only exports common directives and pipes.

> When it comes to components, pipes and directives, every module should import its own dependencies disregarding if the same dependencies were imported in the root module or in any other feature module. In short, even when having multiple feature modules, each one of them needs to import the `CommonModule` .

- We are using a new property called `exports` . Every element defined in the `declarations` array is **private by default**. We should only export whatever the other modules in our application need to perform its job. In our case, we only need to make the `CreditCardComponent` visible because it's being used in the template of the `AppComponent` .

app/app.component.ts

```
...
@Component({
    ...
    template: `
        ...
        <app-credit-card></app-credit-card>
    `
})
export class AppComponent {}
```

> We are keeping the `CreditCardMaskPipe` private because it's only being used inside the `CreditCardModule` and no other module should use it directly.

We can now import this feature module into our simplified root module.

app/app.module.ts

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

import { CreditCardModule } from '../credit-card/credit-card.module'; import { AppComponent }
from './app.component';

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

At this point we are done and our application behaves as expected.

Services and Lazy Loaded Modules

Here's the tricky part of Angular modules. While components, pipes and directives are scoped to its modules unless explicitly exported, services are globally available unless the module is lazy loaded.

It's hard to understand that at first so let's try to see what's happening with the `CreditCardService` in our example. Notice first that the service is not in the `exports` array but in the `providers` array. With this configuration, our service is going to be available

everywhere, even in the `AppComponent` which lives in another module. So, even when using modules, there's no way to have a "private" service unless... the module is being lazy loaded.

When a module is lazy loaded, Angular is going to create a child injector (which is a child of the root injector from the root module) and will create an instance of our service there.

Imagine for a moment that our `CreditCardModule` is configured to be lazy loaded. With our current configuration, when the application is bootstrapped and our root module is loaded in memory, an instance of the `CreditCardService` (a singleton) is going to be added to the root injector. But, when the `CreditCardModule` is lazy loaded sometime in the future, a child injector will be created for that module **with a new instance** of the `CreditCardService` . At this point we have a hierarchical injector with **two instances** of the same service, which is not usually what we want.

Think for example of a service that does the authentication. We want to have only one singleton in the entire application, disregarding if our modules are being loaded at bootstrap or lazy loaded. So, in order to have our feature module's service **only** added to the root injector, we need to use a different approach.

credit-card/credit-card.module.ts

```
import { NgModule, ModuleWithProviders } from '@angular/core'; /* ...other imports...
*/

@NgModule({
    imports: [CommonModule],
    declarations: [
        CreditCardMaskPipe,
        CreditCardComponent
    ],
    exports: [CreditCardComponent]
})
export class CreditCardModule {
    static forRoot(): ModuleWithProviders {
        return {
            ngModule: CreditCardModule,
            providers: [CreditCardService]
        }
    }
}
```

Different than before, we are not putting our service directly in the property `providers` of the `NgModule` decorator. This time we are defining a static method called `forRoot` where we define the module **and** the service we want to export.

With this new syntax, our root module is slightly different.

app/app.module.ts

```
/* ...imports... */

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

Can you spot the difference? We are not importing the `CreditCardModule` directly, instead what we are importing is the object returned from the `forRoot` method, which includes the `CreditCardService` . Although this syntax is a little more convoluted than the original, it will guarantee us that only one instance of the `CreditCardService` is added to the root module. When the `CreditCardModule` is loaded (even lazy loaded), no new instance of that service is going to be added to the child injector.

As a rule of thumb, **always use the `forRoot` syntax when exporting services from feature modules**, unless you have a very special need that requires multiple instances at different levels of the dependency injection tree.

Directive Duplications

Because we no longer define every component and directive directly in every component that needs it, we need to be aware of how Angular modules handle directives and components that target the same element (have the same selector).

Let's assume for a moment that by mistake, we have created two directives that target the same property:

> This example is a variation of the code found in the official documentation.

blue-highlight.directive.ts

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

@Directive({
    selector: '[appHighlight]'
})
export class BlueHighlightDirective {
    constructor(renderer: Renderer, el: ElementRef) {
```

```
        renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'blue');
        renderer.setElementStyle(el.nativeElement, 'color', 'gray');
    }
}
```

yellow-highlight.directive.ts

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

@Directive({
    selector: '[appHighlight]'
})
export class YellowHighlightDirective { constructor(renderer:
    Renderer, el: ElementRef) {
        renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'yellow');
    }
}
```

These two directives are similar, they are trying to style an element. The `BlueHighlightDirective` will try to set the background color of the element to blue while changing the color of the text to gray, while the `YellowHighlightDirective` will try only to change the background color to yellow. Notice that both are targeting any HTML element that has the property `appHighlight` . What would happen if we add both directives to the same module?

app.module.ts

```
// Imports

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

Let's see how we would use it in the only component of the module.

app.component.ts

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

@Component({
    selector: 'app-root',
    template: '<h1 appHighlight>My Angular App</h1>' })
export class AppComponent {}
```

We can see that in the template of our component, we are using the directive `appHighlight` in our `h1` element but, which styles are going to end up being applied?

The answer is: the text is going to be gray and the background yellow.

We are allowed to define multiple directives that target the same elements in the same module. What's going to happen is that Angular is going to do every transformation **in order**.

```
declarations: [
    ...,
    BlueHighlightDirective,
    YellowHighlightDirective
]
```

Because we have defined both directives in an array, and **arrays are ordered collection of items**, when the compiler finds an element with the property `appHighlight` , it will first apply the transformations of `BlueHighlightDirective` , setting the text gray and the background

blue, and then will apply the transformations of `YellowHighlightDirective` , changing again the background color to yellow.

In summary, **when two or more directives target the same element, they are going to be applied in the order they were defined**.

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.

To show this relationship, let's start by defining a simple module that will act as the root module of our example application.

app/app.module.ts

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

import { AppComponent } from './app.component'; import {
EagerComponent } from './eager.component'; import { routing } from
'./app.routing';
```

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

So far this is a very common module that relies on the BrowserModule , has a routing mechanism and two components: AppComponent and EagerComponent . For now, let's focus on the root component of our application (AppComponent) where the navigation is defined.

app/app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
    <h1>My App</h1>
    <nav>
      <a routerLink="eager">Eager</a>
      <a routerLink="lazy">Lazy</a>
    </nav>
    <router-outlet></router-outlet>
  `
})
export class AppComponent {}
```

Our navigation system has only two paths: eager and lazy loading when clicking on them we need to take a look at the to the root module.

. To know what those paths are routing object that we passed

app/app.routing.ts

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

import { EagerComponent } from './eager.component';

const routes: Routes = [
  { path: '', redirectTo: 'eager', pathMatch: 'full' },
  { path: 'eager', component: EagerComponent },
```

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

export const routing: ModuleWithProviders = RouterModule.forRoot(routes);
```

Here we can see that the default path in our application is called `eager` which will load `EagerComponent`.

app/eager.component.ts

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

@Component({
  template: '<p>Eager Component</p>'
})
export class EagerComponent {}
```

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.

There's nothing special about `LazyModule` other than it has its own `routing` and a component called `LazyComponent`.

app/lazy/lazy.module.ts

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

import { LazyComponent }        from './lazy.component';
import { routing } from './lazy.routing';

@NgModule({
  imports: [routing],
  declarations: [LazyComponent]
})
export class LazyModule {}
```

> If we define the class `LazyModule` as the `default` export of the file, we don't need to

> define the class name in the `loadChildren` property as shown above.

The `routing` object is very simple and only defines the default component to load when navigating to the `lazy` path.

app/lazy/lazy.routing.ts

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

import { LazyComponent } from './lazy.component';

const routes: Routes = [
    { path: '', component: LazyComponent } ];


export const routing: ModuleWithProviders = RouterModule.forChild(routes);
```

Notice that we use the method call `forChild` instead of `forRoot` to create the routing object. We should always do that when creating a routing object for a feature module, no matter if the module is supposed to be eagerly or lazily loaded.

Finally, our `LazyComponent` is very similar to `EagerComponent` and is just a placeholder for some text.

app/lazy/lazy.component.ts

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

@Component({
    template: '<p>Lazy Component</p>'
})
export class LazyComponent {}
```

When we load our application for the first time, the `AppModule` along the `AppComponent` will be loaded in the browser and we should see the navigation system and the text "Eager Component". Until this point, the `LazyModule` has not being downloaded, only when we click the link "Lazy" the needed code will be downloaded and we will see the message "Lazy Component" in the browser.

We have effectively lazily loaded a module.

Lazy Loading and the Dependency Injection Tree

Lazy loaded modules create their own branch on the Dependency Injection (DI) tree. This means that it's possible to have services that belong to a lazy loaded module, that are not accessible by the root module or any other eagerly loaded module of our application.

To show this behaviour, let's continue with the example of the previous section and add a `CounterService` to our `LazyModule`.

app/lazy/lazy.module.ts

```
...
import { CounterService } from './counter.service';

@NgModule({
    ...
    providers: [CounterService]
})
export class LazyModule {}
```

Here we added the `CounterService` to the `providers` array. Our `CounterService` is a simple class that holds a reference to a `counter` property.

app/lazy/counter.service.ts

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

@Injectable()
export class CounterService {
    counter = 0;
}
```

We can modify the `LazyComponent` to use this service with a button to increment the `counter` property.

app/lazy/lazy.component.ts

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

import { CounterService } from './counter.service';

@Component({
    template: `
        <p>Lazy Component</p>
        <button (click)="increaseCounter()">Increase Counter</button> <p>Counter: {{
        counterService.counter }}</p>
    `
})
export class LazyComponent {

    constructor(public counterService: CounterService) {}

    increaseCounter() {
        this.counterService.counter += 1;
    }
}
```

The service is working. If we increment the counter and then navigate back and forth between the `eager` and the `lazy` routes, the `counter` value will persist in the lazy loaded module.

But the question is, how can we verify that the service is isolated and cannot be used in a component that belongs to a different module? Let's try to use the same service in the `EagerComponent`.

app/eager.component.ts

```
import { Component } from '@angular/core';
import { CounterService } from './lazy/counter.service';

@Component({
    template: `
        <p>Eager Component</p>
        <button (click)="increaseCounter()">Increase Counter</button> <p>Counter: {{
        counterService.counter }}</p>
    `
})
export class EagerComponent {
    constructor(public counterService: CounterService) {}

    increaseCounter() {
        this.counterService.counter += 1;
    }
}
```

If we try to run this new version of our code, we are going to get an error message in the browser console:

```
No provider for CounterService!
```

What this error tells us is that the `AppModule`, where the `EagerComponent` is defined, has no knowledge of a service called `CounterService`. `CounterService` lives in a different branch of the DI tree created for `LazyModule` when it was lazy loaded in the browser.

Shared Modules and Dependency Injection

Now that we have proven that lazy loaded modules create their own branch on the Dependency Injection tree, we need to learn how to deal with services that are imported by means of a shared module in both an eager and lazy loaded module.

Let's create a new module called `SharedModule` and define the `CounterService` there.

app/shared/shared.module.ts

```
import { NgModule } from '@angular/core';
import { CounterService } from './counter.service';

@NgModule({
  providers: [CounterService]
})
export class SharedModule {}
```

Now we are going to import that `SharedModule` in the `AppModule` and the `LazyModule`.

app/app.module.ts

```
...
import { SharedModule } from './shared/shared.module';

@NgModule({
  imports: [
    SharedModule,
    ...
  ],
  declarations: [
    EagerComponent,
    ...
  ]
  ...
})
export class AppModule {}
```

app/lazy/lazy.module.ts

```
...
import { SharedModule } from '../shared/shared.module';

@NgModule({
  imports: [
    SharedModule,
    ...
  ],
  declarations: [LazyComponent]
})
export class LazyModule {}
```

With this configuration, the components of both modules will have access to the `CounterService`. We are going to use this service in `EagerComponent` and `LazyComponent` in exactly the same way. Just a button to increase the internal `counter` property of the service.

app/eager.component.ts

```
import { Component } from '@angular/core';
import { CounterService } from './shared/counter.service';

@Component({
  template: `
    <p>Eager Component</p>
    <button (click)="increaseCounter()">Increase Counter</button> <p>Counter: {{
    counterService.counter }}</p>
  `
})
export class EagerComponent {
  constructor(public counterService: CounterService) {}

  increaseCounter() {
    this.counterService.counter += 1;
  }
}
```

If you play with the live example, you will notice that the `counter` seems to behave independently in `EagerComponent` and `LazyComponent`, we can increase the value of one counter without altering the other one. In other words, we have ended up with two instances of the `CounterService`, one that lives in the root of the DI tree of the `AppModule` and another that lives in a lower branch of the DI tree accessible by the `LazyModule`.

This is not neccessarily wrong, you may find situations where you could need different instances of the same service, but I bet most of the time that's not what you want. Think for example of an authentication service, you need to have the same instance with the same

information available everywhere disregarding if we are using the service in an eagerly or lazy loaded module.

In the next section we are going to learn how to have only one instance of a shared service.

Sharing the Same Dependency Injection Tree

So far our problem is that we are creating two instances of the same services in different levels of the DI tree. The instance created in the lower branch of the tree is shadowing the one defined at the root level. The solution? To avoid creating a second instance in a lower level of the DI tree for the lazy loaded module and only use the service instance registered at the root of the tree.

To accomplish that, we need to modify the definition of the `SharedModule` and instead of defining our service in the `providers` property, we need to create a static method called `forRoot` that exports the service along with the module itself.

app/shared/shared.module.ts

```
import { NgModule, ModuleWithProviders } from '@angular/core'; import {
CounterService } from './counter.service';

@NgModule({})
export class SharedModule {
    static forRoot(): ModuleWithProviders {
        return {
            ngModule: SharedModule,
            providers: [CounterService]
        };
    }
}
```

With this setup, we can import this module in our root module `AppModule` calling the `forRoot` method to register the module and the service.

app/app.module.ts

```
...
import { SharedModule } from './shared/shared.module';

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

We should **only** call `forRoot` in the root application module and no where else. This ensures that only a single instance of your service exists at the root level. Calling `forRoot` in another module can register the service again in a different level of the DI tree.

Since `SharedModule` only consists of a service that Angular registers in the root app injector, we do not need to import it in `LazyModule` . This is because the lazy loaded module will already have access to services defined at the root level.

This time, whenever we change the value of the `counter` property, this value is shared between the `EagerComponent` and the `LazyComponent` proving that we are using the same instance of the `CounterService` .

However it is very likely that we may have a component, pipe or directive defined in `SharedModule` that we'll need in another module. Take the following for example.

app/shared/shared.module.ts

```
import { NgModule, ModuleWithProviders } from '@angular/core'; import {
CounterService } from './counter.service';

import { HighlightDirective } from './highlight.directive';

@NgModule({
    declarations: [HighlightDirective],
    exports: [ HighlightDirective ]
})
export class SharedModule {
    static forRoot(): ModuleWithProviders {
        return {
            ngModule: SharedModule,
            providers: [CounterService]
        };
    }
}
```

In here, we declare and export HighlightDirective so other modules that import SharedModule can use it in their templates. This means we can just import the module in LazyModule normally.

app/lazy/lazy.module.ts

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

import { SharedModule } from '../shared/shared.module';

import { LazyComponent } from './lazy.component';
import { routing } from './lazy.routing';

@NgModule({
  imports: [
    SharedModule,
    routing
  ],
  declarations: [LazyComponent]
})
export class LazyModule {}
```

Now we can use this directive within LazyModule without creating another instance of CounterService.

Routing

In this section we will discuss the role of routing in Single Page Applications and Angular's new component router.

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 (covered later in this chapter).

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. The following sample shows how we import this module in an `app.routes.ts` file.

app/app.routes.ts

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

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


export const routing = RouterModule.forRoot(routes);
```

We then import our routing configuration in the root of our application.

app/app.module.ts

```
...
import { routing } from './app.routes';

@NgModule({
    imports: [
        BrowserModule,
        routing
```

```
    ],
    declarations: [
        AppComponent,
        ComponentOne,
        ComponentTwo
    ],
    bootstrap: [ AppComponent ]
})
export class AppModule {
}
```

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 `<router-outlet></router-outlet>` if the entire URL matches the empty path ('').

When starting the application, it will now automatically navigate to the route for `component-one` .

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']);
```

Dynamically Adding Route Components

Rather than define each route's component separately, use `RouterOutlet` which serves as a component placeholder; Angular dynamically adds the component for the route being activated into the element.

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

@Component({
  selector: 'app',
  template: `
    <nav>
      <a routerLink="/component-one">Component One</a> <a
      routerLink="/component-two">Component Two</a>
    </nav>

    <router-outlet></router-outlet>
    <!-- Route components are added by router here -->
  `
})
export class AppComponent {}
```

In the above example, the component corresponding to the route specified will be placed after the `<router-outlet></router-outlet>` element when the link is clicked.

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 } ];
```

Linking to Routes with Parameters

In the `ProductList` component you could display a list of products. Each product would have a link to the `product-details` route, passing the ID of the product:

```
<a *ngFor="let product of products"
    [routerLink]="['/product-details', product.id]">
    {{ product.name }} </a>
```

Note that the `routerLink` directive passes an array which specifies the path and the route parameter. Alternatively we could navigate to the route programmatically:

```
goToProductDetails(id) {
    this.router.navigate(['/product-details', id]);
}
```

Reading Route Parameters

The `ProductDetails` component must read the parameter, then load the product based on the ID given in the parameter.

The `ActivatedRoute` service provides a `params` Observable which we can subscribe to to get the route parameters (see Observables).

```
import { Component, OnInit, OnDestroy } from '@angular/core'; import {
ActivatedRoute } from '@angular/router';

@Component({
    selector: 'product-details',
    template: `
        <div>
            Showing product details for product: {{id}}
        </div>
    `,
})
export class LoanDetailsPage implements OnInit, OnDestroy { id: number;

    private sub: any;

    constructor(private route: ActivatedRoute) {}

    ngOnInit() {
        this.sub = this.route.params.subscribe(params => {
             this.id = +params['id']; // (+) converts string 'id' to a number

             //    In a real app: dispatch action to load the details here.
        });
    }

    ngOnDestroy() {
        this.sub.unsubscribe();
    }
}
```

> The reason that the `params` property on `ActivatedRoute` is an Observable is that the router may not recreate the component when navigating to the same component. In this case the parameter may change without the component being recreated.

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:3000/product-details/3/overview
```

When the user clicks "Technical Specs":

```
localhost:3000/product-details/3/specs
```

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

Our `Routes` with children would look like:

```
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` .

```
import { Component, OnInit, OnDestroy } from '@angular/core'; import {
ActivatedRoute } from '@angular/router';

@Component({
    selector: 'product-details',
    template: `
        <p>Product Details: {{id}}</p>
        <!-- Product information -->
        <nav>
            <a [routerLink]="['overview']">Overview</a>
            <a [routerLink]="['specs']">Technical Specs</a>
        </nav>
        <router-outlet></router-outlet>
        <!-- Overview & Specs components get added here by the router -->
    `
})
export default class ProductDetails implements OnInit, OnDestroy { id: number;


    constructor(private route: ActivatedRoute) {}

    ngOnInit() {
        this.sub = this.route.params.subscribe(params => {
            this.id = +params['id']; // (+) converts string 'id' to a number });

    }
```

```
    ngOnDestroy() {
        this.sub.unsubscribe();
    }
}
```

Alternatively, we could specify `overview` route URL simply as:

```
localhost:3000/product-details/3
```

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

            { path: '', component: Overview },
            { path: 'specs', component: Specs }
        ]
    }
];
```

Since the `Overview` child route of `product-details` has an empty path, it will be loaded by default. The `specs` child route remains the same.

Accessing a Parent's Route Parameters

In the above example, say that the child routes of `product-details` needed the ID of the product to fetch the spec or overview information. The child route component can access the parent route's parameters as follows:

```
export default class Overview {
    parentRouteId: number;
    private sub: any;

                    constructor(private router: Router,
                      private route: ActivatedRoute) {}

    ngOnInit() {
        // Get parent ActivatedRoute of this route.
        this.sub = this.router.routerState.parent(this.route)
            .params.subscribe(params => {
                this.parentRouteId = +params["id"];
            });
    }

    ngOnDestroy() {
        this.sub.unsubscribe();
    }
}
```

Links

Routes can be prepended with `/` , or `../` ; this tells Angular where in the route tree to link to.

Prefix	Looks in
`/`	Root of the application
none	Current component children routes
`../`	Current component parent routes

Example:

```
<a [routerLink]="['route-one']">Route One</a> <a
[routerLink]="['../route-two']">Route Two</a> <a
[routerLink]="['/route-three']">Route Three</a>
```

In the above example, the link for route one links to a child of the current route. The link for route two links to a sibling of the current route. The link for route three links to a child of the root component (same as route one link if current route is root component).

Controlling Access to or from a Route

To control whether the user can navigate to or away from a given route, use route guards.

For example, we may want some routes to only be accessible once the user has logged in or accepted Terms & Conditions. We can use route guards to check these conditions and control access to routes.

Route guards can also control whether a user can leave a certain route. For example, say the user has typed information into a form on the page, but has not submitted the form. If they were to leave the page, they would lose the information. We may want to prompt the user if the user attempts to leave the route without submitting or saving the information.

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.

For example, say we have an `accounts` route that only users that are logged in can navigate to. This page also has forms and we want to make sure the user has submitted unsaved changes before leaving the accounts page.

In our route config we can add our guards to that route:

```
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);
```

Now `LoginRouteGuard` will be checked by the router when activating the `accounts` route, and `SaveFormsGuard` will be checked when leaving that route.

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.

Implementing CanDeactivate

CanDeactivate works in a similar way to CanActivate but there are some important differences. The canDeactivate function passes the component being deactivated as an argument to the function:

```
export interface CanDeactivate<T> {
    canDeactivate(component: T, route: ActivatedRouteSnapshot, state: RouterStateSnapsho t):

            Observable<boolean>|Promise<boolean>|boolean;
}
```

We can use that component to determine whether the user can deactivate.

```
import { CanDeactivate } from '@angular/router'; import {
Injectable } from '@angular/core'; import { AccountPage } from
'./account-page';

@Injectable()
export class SaveFormsGuard implements CanDeactivate<AccountPage> {

    canDeactivate(component: AccountPage) {
        return component.areFormsSaved();
    }

}
```

Async Route Guards

The canActivate and canDeactivate functions can either return values of type boolean , or Observable<boolean> (an Observable that resolves to boolean). If you need to do an asynchronous request (like a server request) to determine whether the user can navigate to or away from the route, you can simply return an Observable<boolean> . The router will wait until it is resolved and use that value to determine access.

For example, when the user navigates away you could have a dialog service ask the user to confirm the navigation. The dialog service returns an `Observable<boolean>` which resolves to true if the user clicks 'OK', or false if user clicks 'Cancel'.

```
canDeactivate() {
  return dialogService.confirm('Discard unsaved changes?');
}
```

Passing Optional Parameters

Query parameters allow you to pass optional parameters to a route such as pagination information.

For example, on a route with a paginated list, the URL might look like the following to indicate that we've loaded the second page:

```
localhost:3000/product-list?page=2
```

> The key difference between query parameters and route parameters is that route parameters are essential to determining route, whereas query parameters are optional.

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 to read the query parameters:

```
import { Component } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router';

@Component({
    selector: 'product-list',
    template: `<!-- Show product list -->`
})
export default class ProductList {
    constructor(
        private route: ActivatedRoute,
        private router: Router) {}

    ngOnInit() {
        this.sub = this.route
            .queryParams
            .subscribe(params => {
                //    Defaults to 0 if no query param provided. this.page =
                +params['page'] || 0;
            });
    }

    ngOnDestroy() {
        this.sub.unsubscribe();
    }

    nextPage() {
        this.router.navigate(['product-list'], { queryParams: { page: this.page + 1 } });
    }
}
```

Using Auxiliary Routes

Angular supports the concept of auxiliary routes, which allow you to set up and navigate multiple independent routes in a single app. Each component has one primary route and zero or more auxiliary outlets. Auxiliary outlets must have unique name within a component.

To define the auxiliary route we must first add a named router outlet where contents for the auxiliary route are to be rendered.

Here's an example:

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

@Component({
    selector: 'app',
    template: `
        <nav>
            <a [routerLink]="['/component-one']">Component One</a> <a
            [routerLink]="['/component-two']">Component Two</a>
            <a [routerLink]="[{ outlets: { 'sidebar': ['component-aux'] } }]">Component Aux<
/a>
```

```
        </nav>

        <div style="color: green; margin-top: 1rem;">Outlet:</div>
        <div style="border: 2px solid green; padding: 1rem;"> <router-
            outlet></router-outlet>
        </div>

        <div style="color: green; margin-top: 1rem;">Sidebar Outlet:</div>
        <div style="border: 2px solid blue; padding: 1rem;"> <router-outlet
            name="sidebar"></router-outlet>
        </div>
    `
})
export class AppComponent {

}
```

Next we must define the link to the auxiliary route for the application to navigate and render the contents.

```
<a [routerLink]="[{ outlets: { 'sidebar': ['component-aux'] } }]">
    Component Aux
</a>
```

Each auxiliary route is an independent route which can have:

- its own child routes
- its own auxiliary routes
- its own route-params
- its own history stack

Testing

Test-Driven-Development is an engineering process in which the developer writes an initial automated test case that defines a feature, then writes the minimum amount of code to pass the test and eventually refactors the code to acceptable standards.

A *unit test* is used to test individual components of the system. An *integration test* is a test which tests the system as a whole, and how it will run in production.

Unit tests should only verify the behavior of a specific unit of code. If the unit's behavior is modified, then the unit test would be updated as well. Unit tests should not make assumptions about the behavior of other parts of your codebase or your dependencies. When other parts of your codebase are modified, your unit tests should not fail. (Any failure indicates a test that relies on other components and is therefore not a unit test.) Unit tests are cheap to maintain and should only be updated when the individual units are modified. For TDD in Angular, a unit is most commonly defined as a class, pipe, component, or service. It is important to keep units relatively small. This helps you write small tests which are "self-documenting", where they are easy to read and understand.

The Testing Toolchain

Our testing toolchain will consist of the following tools:

- Jasmine
- Karma
- Phantom-js
- Istanbul
- Sinon
- Chai

Jasmine is the most popular testing framework in the Angular community. This is the core framework that we will write our unit tests with.

Karma is a test automation tool for controlling the execution of our tests and what browser to perform them under. It also allows us to generate various reports on the results. For one or two tests this may seem like overkill, but as an application grows larger and the number of units to test grows, it is important to organize, execute and report on tests in an efficient manner. Karma is library agnostic so we could use other testing frameworks in combination with other tools (like code coverage reports, spy testing, e2e, etc.).

In order to test our Angular application we must create an environment for it to run in. We could use a browser like Chrome or Firefox to accomplish this (Karma supports in-browser testing), or we could use a browser-less environment to test our application, which can offer us greater control over automating certain tasks and managing our testing workflow. PhantomJS provides a JavaScript API that allows us to create a headless DOM instance which can be used to bootstrap our Angular application. Then, using that DOM instance that is running our Angular application, we can run our tests.

Istanbul is used by Karma to generate code coverage reports, which tells us the overall percentage of our application being tested. This is a great way to track which components/services/pipes/etc. have tests written and which don't. We can get some useful insight into how much of the application is being tested and where.

For some extra testing functionality we can use the Sinon library for things like test spys, test subs and mock XHR requests. This is not necessarily required as Jasmine comes with the `spyOn` function for incorporating spy tests.

Chai is an assertion library that can be paired with any other testing framework. It offers some syntactic sugar that lets us write our unit tests with different verbiage - we can use a *should*, *expect* or *assert* interface. Chai also takes advantage of "function chaining" to form

English-like sentences used to describe tests in a more user friendly way. Chai isn't a required library for testing and we won't explore it much more in this handout, but it is a useful tool for creating cleaner, more well-written tests.

Simple Test

To begin, let's start by writing a simple test in Jasmine.

```
describe('Testing math', () => {
    it('multiplying should work', () => {
        expect(4 * 4).toEqual(16);
    });
});
```

Though this test may be trivial, it illustrates the basic elements of a unit test. We explain what this test is for by using `describe` , and we use `it` to assert what kind of result we are expecting from our test. These are user-defined so it's a good idea to be as descriptive and accurate in these messages as possible. Messages like "should work", or "testing service" don't really explain exactly what's going on and may be confusing when running multiple tests across an entire application.

Our actual test is basic, we use `expect` to formulate a scenario and use `toEqual` to assert the resulting condition we are expecting from that scenario. The test will pass if our assertion is equal to the resulting condition, and fail otherwise. You always want your tests to pass - do not write tests that have the results you want in a failed state.

Testing Components

Testing Angular components requires some insight into the Angular `@angular/core/testing` module. Though many features of Jasmine are used in Angular's testing module there are some very specific wrappers and routines that Angular requires when testing components.

Verifying Methods and Properties

We can test the properties and methods of simple Angular components fairly easily - after all, Angular components are simple classes that we can create and interface with. Say we had a simple component that kept a defined message displayed. The contents of the message may be changed through the `setMessage` function, and the `clearMessage` function would put an empty message in place. This is a very trivial component but how would we test it?

message.component.ts

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

@Component({
    selector: 'display-message',
    template: '<h1>{{message}}</h1>'
})

export class MessageComponent {
    public message: string = '';

    constructor() {}

    setMessage(newMessage: string) {
            this.message = newMessage;
    }

    clearMessage() {
        this.message = '';
    }
}
```

Now for our unit test. We'll create two tests, one to test the `setMessage` function to see if the new message shows up and another to test the `clearMessage` function to see if clearing the message works as expected.

message.spec.ts

```
import {MessageComponent} from './message.component';

describe('Testing message state in message.component', () => { let app:
    MessageComponent;

    beforeEach(() => {
        app = new MessageComponent();
    });

    it('should set new message', () => {
        app.setMessage('Testing');
        expect(app.message).toBe('Testing');
    });

    it('should clear message', () => {
        app.clearMessage();
        expect(app.message).toBe('');
    });
});
```

We have created two tests: one for `setMessage` and the other for `clearMessage` . In order to call those functions we must first initialize the `MessageComponent` class. This is accomplished by calling the `beforeEach` function before each test is performed.

Once our `MessageComponent` object is created we can call `setMessage` and `clearMessage` and analyze the results of those actions. We formulate an expected result, and then test to see if the result we were expecting came to be. Here we are testing whether or not the message we tried to set modified the `MessageComponent` property `message` to the value we intended. If it did, then the test was successful and our `MessageComponent` works as expected.

Injecting Dependencies and DOM Changes

In the previous example the class we were testing, `MessageComponent` , did not have any injected dependencies. In Angular, components will often rely on services and other classes (pipes/providers/etc.) to function, which will be injected into the constructor of the components class. When testing these components we have to inject the dependencies ourselves. Since this is an Angular-specific routine, there are no pure Jasmine functions used to accomplish this. Angular provides a multitude of functions in `@angular/core/testing` that allows us to to effectively test our components. Let's take a look at a basic component:

quote.component.ts

```
import { QuoteService } from './quote.service'; import {
Component } from '@angular/core';

@Component({
    selector: 'my-quote',
    template: '<h3>Random Quote</h3> <div>{{quote}}</div>' })


export class QuoteComponent {
    quote: string;

    constructor(private quoteService: QuoteService){};

    getQuote() {
        this.quoteService.getQuote().then((quote) => {
            this.quote = quote;
        });
    };
}
```

This component relies on the `QuoteService` to get a random quote, which it will then display. The class is pretty simple - it only has the `getQuote` function that will modify the DOM, therefore it will be our main area of focus in testing.

In order to test this component we need initiate the `QuoteComponent` class. The Angular testing library offers a utility called `TestBed` . This allows us to configure a testing module where we can provided mocked dependencies. Additionally it will create the component for us and return a *component fixture* that we can perform testing operations on.

quote.spec.ts

```
import { QuoteService } from './quote.service';

import { QuoteComponent } from './quote.component';
 import { provide, destroyPlatform } from '@angular/core';
import {

    async,
    inject,
    TestBed,
} from '@angular/core/testing';
import {

    BrowserDynamicTestingModule,
    platformBrowserDynamicTesting
} from '@angular/platform-browser-dynamic/testing';

class MockQuoteService {
    public quote: string = 'Test quote';

    getQuote() {
        return Promise.resolve(this.quote);
    }
}

describe('Testing Quote Component', () => {

    let fixture;

    beforeEach(() => destroyPlatform());

    beforeEach(() => {
        TestBed.initTestEnvironment(
            BrowserDynamicTestingModule,
            platformBrowserDynamicTesting()
        );

        TestBed.configureTestingModule({
            declarations: [
                QuoteComponent
            ],
            providers: [
                 { provide: QuoteService, useClass: MockQuoteService }
             ]
           });
        fixture = TestBed.createComponent(QuoteComponent);
        fixture.detectChanges();
    });
```

```
    it('Should get quote', async(inject([], () => {
        fixture.componentInstance.getQuote(); fixture.whenStable()

            .then(() => {
                fixture.detectChanges();
                return fixture.whenStable();
            })
            .then(() => {
                const compiled = fixture.debugElement.nativeElement;
                expect(compiled.querySelector('div').innerText).toEqual('Test quote');
            });
    })));
});
```

Testing the QuoteComponent is a fairly straightforward process. We want to create a QuoteComponent , feed it a quote and see if it appears in the DOM. This process requires us to create the component, pass in any dependencies, trigger the component to perform an action and then look at the DOM to see if the action is what we expected.

Let's take a look at how this is accomplished with the above unit test.

We use TestBed.initTestingEnvironment to create a testing platform using BrowserDynamicTestingModule and platformBrowserDynamicTesting as arguments, which are also imported from angular and allow the application to be bootstrapped for testing. This is necessary for all unit tests that make use of TestBed . Notice that this platform is destroyed and reset before each test runs.

We use TestBed.configureTestingModule to feed in any dependencies that our component requires. Here our component depends on the QuoteService to get data. We mock this data ourselves thus giving us control over what value we expect to show up. It is good practice to separate component testing from service testing - this makes it easier to test as you are only focusing on a single aspect of the application at a time. If your service or component fails, how will you know which one was the culprit? We inject the QuoteService dependency using our mock class MockQuoteService , where we will provide mock data for the component to consume.

Next we use TestBed.createComponent(QuoteComponent) to create a *fixture* for us to use in our tests. This will then create a new instance of our component, fulfilling any Angular-specific routines like dependency injection. A fixture is a powerful tool that allows us to query the DOM rendered by a component, as well as change DOM elements and component properties. It is the main access point of testing components and we use it extensively.

In the test we have gotten access to our component through the fixture.componentInstance property. We then call getQuote to kickstart our only action in the QuoteComponent component. We run the test when the fixture is stable by using its whenStable method which will ensure the promise inside the getQuote() has resolved, giving the component a chance to set the quote value. We call fixture.detectChanges to keep an eye

out for any changes taking place to the DOM, and use the `fixture.debugElement.nativeElement` property to get access to those underlying DOM elements.

Now we can check to see if the DOM rendered by our `QuoteComponent` contains the quote that we mocked in through the `QuoteService` . The final line attempts to assert that the DOM's div tag contains the mocked quote 'Test Quote' inside. If it does, then our component passes the test and works as expected; if it doesn't, that means our component is not outputting quotes correctly.

We wrap `Should get quote` test in `async()` . This is to allow our tests run in an asynchronous test zone. Using `async` creates a test zone which will ensure that all asynchronous functions have resolved prior to ending the test.

Overriding Dependencies for Testing

`TestBed` provides several functions to allow us to override dependencies that are being used in a test module.

- `overrideModule`
- `overrideComponent`
- `overrideDirective`
- `overridePipe`

For example, you might want to override the template of a component. This is useful for testing a small part of a large component, as you can ignore the output from the rest of the DOM and only focus on the part you are interested in testing.

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

@Component({
    selector: 'display-message',
    template: `
        <div>
            <div>
                <h1>{{message}}</h1>
            <div>
        </div>
    `
})
export class MessageComponent {
    public message: string = '';

    setMessage(newMessage: string) {
            this.message = newMessage;
    }
}
```

```
import {MessageComponent} from './message.component'; import {
provide } from '@angular/core'; import {

    async,
    inject,
    TestBed,
} from '@angular/core/testing';

describe('MessageComponent', () => {

    let fixture;

    beforeEach(() => {
        TestBed.configureTestingModule({
            declarations: [MessageComponent],
            providers: []
        });

        fixture = TestBed.overrideComponent(MessageComponent, { set: {

                template: '<span>{{message}}</span>'
            }})
            .createComponent(MessageComponent);

        fixture.detectChanges();
    });

    it('should set the message', async(inject([], () => {
        fixture.componentInstance.setMessage('Test message');
        fixture.detectChanges();
        fixture.whenStable().then(() => {
            const compiled = fixture.debugElement.nativeElement;
            expect(compiled.querySelector('span').innerText).toEqual('Test message');
          });
         })));

});
```

Testing Asynchronous Actions

Sometimes we need to test components that rely on asynchronous actions that happen at specific times. Angular provides a function called `fakeAsync` which wraps our tests in a zone and gives us access to the `tick` function, which will allow us to simulate the passage of time precisely.

Let's go back to the example of the `QuoteComponent` component and rewrite the unit test using `fakeAsync`:

```
import { Component } from '@angular/core'; import {
QuoteService } from './quote.service';

@Component({
    selector: 'my-quote',
    template: '<h3>Random Quote</h3> <div>{{quote}}</div>' })


export class QuoteComponent {
    quote: string;

    constructor(private quoteService: QuoteService){};

    getQuote() {
        this.quoteService.getQuote().then((quote) => {
            this.quote = quote;
        });
    };
}



import { QuoteService } from './quote.service';
import { QuoteComponent } from './quote.component';
import { provide } from '@angular/core'; import {
    async,
    TestBed,
    fakeAsync,
    tick,
} from '@angular/core/testing';

class MockQuoteService {
    public quote: string = 'Test quote';

    getQuote() {
        return Promise.resolve(this.quote);
    }
}

describe('Testing Quote Component', () => {

    let fixture;

    beforeEach(() => {
        TestBed.configureTestingModule({
            declarations: [
                QuoteComponent
            ],
            providers: [
                { provide: QuoteService, useClass: MockQuoteService }
             ]
           });
        fixture = TestBed.createComponent(QuoteComponent);
        fixture.detectChanges();
    });
```

```
    it('Should get quote', fakeAsync(() => {
      fixture.componentInstance.getQuote();
      tick();
      fixture.detectChanges();
      const compiled = fixture.debugElement.nativeElement;
      expect(compiled.querySelector('div').innerText).toEqual('Test quote');
    }));
});
```

Here we have a `QuoteComponent` that has a `getQuote` update. We have wrapped our entire test in `fakeAsync` asynchronous behavior of our component (`getQuote()` which triggers an asynchronous which will allow us to test the) using synchronous function calls by calling `tick()` . We can then run `detectChanges` and query the DOM for our expected result.

Refactoring Hard-to-Test Code

As you start writing unit tests, you may find that a lot of your code is hard to test. The best strategy is often to refactor your code to make it easy to test. For example, consider refactoring your component code into services and focusing on service tests or vice versa.390

Testing Services

When testing services in Angular we employ many of the same techniques and strategies used for testing components. Services, like components, are classes with methods and properties that we want to verify. Data is the main emphasis in testing services - are we getting, storing and propagating data correctly.

Testing Strategies for Services

When testing services that make HTTP calls, we don't want to hit the server with real requests. This is because we want to isolate the testing of our service from any other outside points of failure. Our service may work, but if the API server is failing or giving values we aren't expecting, it may give the impression that our service is the one failing. Also, as a project grows and the number of unit tests increase, running through a large number of tests that make HTTP requests will take a long time and may put strain on the API server. Therefore, when testing services we'll be mocking out fake data with fake requests.

Injecting Dependencies

Like components, services often require dependencies that Angular injects through the constructor of the service's class. Since we are initializing these classes outside the bootstrapping process of Angular, we must explicitly inject these dependencies ourselves. This is accomplished by using the `TestBed` to configure a testing module and feed in required dependencies like the HTTP module.

Testing HTTP Requests

Services, by their nature, perform asynchronous tasks. When we make an HTTP request we do so in an asynchronous manner so as not to block the rest of the application from carrying out its operations. We looked a bit at testing components asynchronously earlier - fortunately a lot of this knowledge carries over into testing services asynchronously.

The basic strategy for testing such a service is to verify the contents of the request being made (correct URL) and ensure that the data we mock into the service is returned correctly by the right method.

Let's take a look at some code:

wikisearch.ts

```
import {Http} from '@angular/http';
import {Injectable} from '@angular/core';
import {Observable} from 'rxjs';
import 'rxjs/add/operator/map'

@Injectable()
export class SearchWiki {
    constructor (private http: Http) {}

    search(term: string): Observable<any> {
        return this.http.get(
            'https://en.wikipedia.org/w/api.php?' +
            'action=query&list=search&srsearch=' + term
        ).map((response) => response.json());
    }

    searchXML(term: string): Observable<any> {
        return this.http.get(
            'https://en.wikipedia.org/w/api.php?' +
            'action=query&list=search&format=xmlfm&srsearch=' + term
        );
    }
}
```

Here is a basic service. It will query Wikipedia with a search term and return an `Observable` with the results of the query. The search function will make a GET request with the supplied term, and the searchXML method will do the same thing, except request the response to be in XML instead of JSON. As you can see, it depends on the HTTP module to make a request to wikipedia.org.

Our testing strategy will be to check to see that the service has requested the right URL, and once we've responded with mock data we want to verify that it returns that same data.

Testing HTTP Requests Using MockBackend

To unit test our services, we don't want to make actual HTTP requests. To accomplish this, we need to mock out our HTTP services. Angular provides us with a `MockBackend` class that can be configured to provide mock responses to our requests, without actually making a network request.

The configured `MockBackend` can then be injected into HTTP, so any calls to the service, such as `http.get` will return our expected data, allowing us to test our service in isolation from real network traffic.

wikisearch.spec.ts

```
import {
    fakeAsync,
    inject,
    TestBed
} from '@angular/core/testing'; import {

    HttpModule,
    XHRBackend,
    ResponseOptions,
    Response,
    RequestMethod
} from '@angular/http';
import {
    MockBackend,
    MockConnection
} from '@angular/http/testing/mock_backend'; import {SearchWiki}

from './wikisearch.service';

const mockResponse = {
    "batchcomplete": "",
    "continue": {
        "sroffset": 10,
        "continue": "-||"
    },
    "query": {
        "searchinfo": {
            "totalhits": 36853
        },
        "search": [{
            "ns": 0,
            "title": "Stuff",
            "snippet": "<span></span>",
            "size": 1906,
            "wordcount": 204,
            "timestamp": "2016-06-10T17:25:36Z"
        }]
    }
};

describe('Wikipedia search service', () => {

    beforeEach(() => {
        TestBed.configureTestingModule({
            imports: [HttpModule],
            providers: [
                {
                    provide: XHRBackend,
                    useClass: MockBackend
```

```
            },
            SearchWiki
        ]
    });
});

it('should get search results', fakeAsync(
    inject([
        XHRBackend,
        SearchWiki
    ], (mockBackend: XHRBackend, searchWiki: SearchWiki) => {

        const expectedUrl = 'https://en.wikipedia.org/w/api.php?' +
            'action=query&list=search&srsearch=Angular';

        mockBackend.connections.subscribe(
            (connection: MockConnection) => {
                expect(connection.request.method).toBe(RequestMethod.Get);
                expect(connection.request.url).toBe(expectedUrl);

                connection.mockRespond(new Response(
                    new ResponseOptions({ body: mockResponse })
                ));
            });

        searchWiki.search('Angular')
            .subscribe(res => {
                expect(res).toEqual(mockResponse);
            });
    })
));

it('should set foo with a 1s delay', fakeAsync( inject([SearchWiki],
    (searchWiki: SearchWiki) => {
        searchWiki.setFoo('food');
        tick(1000);
        expect(searchWiki.foo).toEqual('food');
    })
));
});
```

We use `inject` to inject the `SearchWiki` service and the `MockBackend` into our test. We then wrap our entire test with a call to `fakeAsync` , which will be used to control the asynchronous behavior of the `SearchWiki` service for testing.

Next, we `subscribe` to any incoming connections from our back-end. This gives us access to an object `MockConnection` , which allows us to configure the response we want to send out from our back-end, as well as test any incoming requests from the service we're testing.

In our example, we want to verify that the `SearchWiki` 's search method makes a GET request to the correct URL. This is accomplished by looking at the request object we get when our `SearchWiki` service makes a connection to our mock back-end. Analyzing the `request.url` property we can see if its value is what we expect it to be. Here we are only

checking the URL, but in other scenarios we can see if certain headers have been set, or if certain POST data has been sent.

Now, using the `MockConnection` object we mock in some arbitrary data. We create a new `ResponseOptions` object where we can configure the properties of our response. This follows the format of a regular Angular Response class. Here we have simply set the `body` property to that of a basic search result set you might see from Wikipedia. We could have also set things like cookies, HTTP headers, etc., or set the `status` value to a non-200 state to test how our service responds to errors. Once we have our `ResponseOptions` configured we create a new instance of a `Respond` object and tell our back-end to start using this as a response by calling `.mockRespond`.

It is possible to use multiple responses. Say your service had two possible GET requests - one for `/api/users`, and another `/api/users/1`. Each of these requests has a different corresponding set of mock data. When receiving a new connection through the `MockBackend` subscription, you can check to see what type of URL is being requested and respond with whatever set of mock data makes sense.

Finally, we can test the `search` method of the `SearchWiki` service by calling it and subscribing to the result. Once our search process has finished, we check the result object to see if it contains the same data that we mocked into our back-end. If it is, then congratulations, your test has passed.
In the `should set foo with a 1s delay` test, you will notice that we call `tick(1000)` which simulates a 1 second delay.

Alternative HTTP Mocking Strategy

An alternative to using `MockBackend` is to create our own light mocks. Here we create an object and then tell TypeScript to treat it as `Http` using type assertion. We then create a spy for its `get` method and return an observable similar to what the real `Http` service would do.

This method still allows us to check to see that the service has requested the right URL, and that it returns that expected data.

wikisearch.spec.ts

```
import {
    fakeAsync,
    inject,
    TestBed
} from '@angular/core/testing'; import {

    HttpModule,
    Http,
```

```
    ResponseOptions,
    Response
} from '@angular/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/of';
import {SearchWiki} from './wikisearch.service';

const mockResponse = {
    "batchcomplete": "",
    "continue": {
        "sroffset": 10,
        "continue": "-||"
    },
    "query": {
        "searchinfo": {
            "totalhits": 36853
        },
        "search": [{
            "ns": 0,
            "title": "Stuff",
            "snippet": "<span></span>",
            "size": 1906,
            "wordcount": 204,
            "timestamp": "2016-06-10T17:25:36Z"
        }]
    }
};

describe('Wikipedia search service', () => {
    let mockHttp: Http;

    beforeEach(() => {
        mockHttp = { get: null } as Http;

        spyOn(mockHttp, 'get').and.returnValue(Observable.of({ json: () =>
            mockResponse
        }));

        TestBed.configureTestingModule({
            imports: [HttpModule],
            providers: [
                {
                    provide: Http,
                    useValue: mockHttp
                },
                SearchWiki
            ]
        });
    });

    it('should get search results', fakeAsync(
        inject([SearchWiki], searchWiki => {
            const expectedUrl = 'https://en.wikipedia.org/w/api.php?' +
                'action=query&list=search&srsearch=Angular';
```

```
        searchWiki.search('Angular')
          .subscribe(res => {
            expect(mockHttp.get).toHaveBeenCalledWith(expectedUrl);
            expect(res).toEqual(mockResponse);
          });
      })
  ));

});
```

Introduction to Angular animations

Animation provides the illusion of motion: HTML elements change styling over time. Well-designed animations can make your application more fun and easier to use, but they aren't just cosmetic. Animations can improve your app and user experience in a number of ways:

- Without animations, web page transitions can seem abrupt and jarring.
- Motion greatly enhances the user experience, so animations give users a chance to detect the application's response to their actions.
- Good animations intuitively call the user's attention to where it is needed.

Typically, animations involve multiple style transformations over time. An HTML element can move, change color, grow or shrink, fade, or slide off the page. These changes can occur simultaneously or sequentially. You can control the timing of each transformation.

Angular's animation system is built on CSS functionality, which means you can animate any property that the browser considers animatable. This includes positions, sizes, transforms, colors, borders, and more.

Enabling the animations module

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

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

Importing animation functions into component files

If you plan to use specific animation functions in component files, import those functions from

```
import {
  trigger,
  state,
  style,
  animate,
  transition,
  // ...
} from '@angular/animations';
```

Adding the animation metadata property

In the component file, add a metadata property called animations: within the @Component() decorator. You put the trigger that defines an animation within the animations metadata property.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  animations: [
    // animation triggers go here
  ]
})
```

Animating a simple transition

Let's animate a simple transition that changes a single HTML element from one state to another. For example, you can specify that a button displays either Open or Closed based on the user's last action. When the button is in the open state, it's visible and yellow. When it's the closed state, it's transparent and green.

In HTML, these attributes are set using ordinary CSS styles such as color and opacity. In Angular, use the style() function to specify a set of CSS styles for use with animations. You can collect a set of styles in an animation state, and give the state a name, such as open or closed.

Animation state and styles

Use Angular's state() function to define different states to call at the end of each transition. This function takes two arguments: a unique name like open or closed and a style() function.

Use the style() function to define a set of styles to associate with a given state name. Note that the style attributes must be in camelCase.

Let's see how Angular's state() function works with the style() function to set CSS style attributes. In this code snippet, multiple style attributes are set at the same time for the state. In the open state, the button has a height of 200 pixels, an opacity of 1, and a background color of yellow.

```
// ...
state('open', style({
        height: '200px',
        opacity: 1,
        backgroundColor: 'yellow'
      })),
```

In the closed state, shown below, the button has a height of 100 pixels, an opacity of 0.5, and a background color of green.

```
state('closed', style({
        height: '100px',
        opacity: 0.5,
        backgroundColor: 'green'
      })),
```

Transitions and timing

In Angular, you can set multiple styles without any animation. However, without further refinement, the button instantly transforms with no fade, no shrinkage, or other visible indicator that a change is occurring.

To make the change less abrupt, we need to define an animation transition to specify the changes that occur between one state and another over a period of time. The transition() function accepts two arguments: the first argument accepts an expression that defines the direction between two transition states, and the second argument accepts an animate() function.

Use the animate() function to define the length, delay, and easing of a transition, and to designate the style function for defining styles while transitions are taking place. You can also use the animate() function to define the keyframes() function for multi-step animations. These definitions are placed in the second argument of the animate() function.

Animation metadata: duration, delay, and easing

The animate() function (second argument of the transition function) accepts the timings and styles input parameters.

The timings parameter takes a string defined in three parts.

animate ('duration delay easing')

The first part, duration, is required. The duration can be expressed in milliseconds as a simple number without quotes, or in seconds with quotes and a time specifier. For example, a duration of a tenth of a second can be expressed as follows:

- As a plain number, in milliseconds: 100
- In a string, as milliseconds: '100ms'
- In a string, as seconds: '0.1s'

The second argument, delay, has the same syntax as duration. For example:

- Wait for 100ms and then run for 200ms: '0.2s 100ms'

The third argument, easing, controls how the animation accelerates and decelerates during its runtime. For example, ease-in causes the animation to begin slowly, and to pick up speed as it progresses.

- Wait for 100ms, run for 200ms. Use a deceleration curve to start out fast and slowly decelerate to a resting point: '0.2s 100ms ease-out'
- Run for 200ms, with no delay. Use a standard curve to start slow, accelerate in the middle, and then decelerate slowly at the end: '0.2s ease-in-out'
- Start immediately, run for 200ms. Use a acceleration curve to start slow and end at full velocity: '0.2s ease-in'

This example provides a state transition from open to closed with a one second transition between states.

```
transition('open => closed', [
        animate('1s')
      ]),
```

In the code snippet above, the => operator indicates unidirectional transitions, and <=> is bidirectional. Within the transition, animate() specifies how long the transition takes. In this case, the state change from open to closed takes one second, expressed here as 1s.

This example adds a state transition from the closed state to the open state with a 0.5 second transition animation arc.

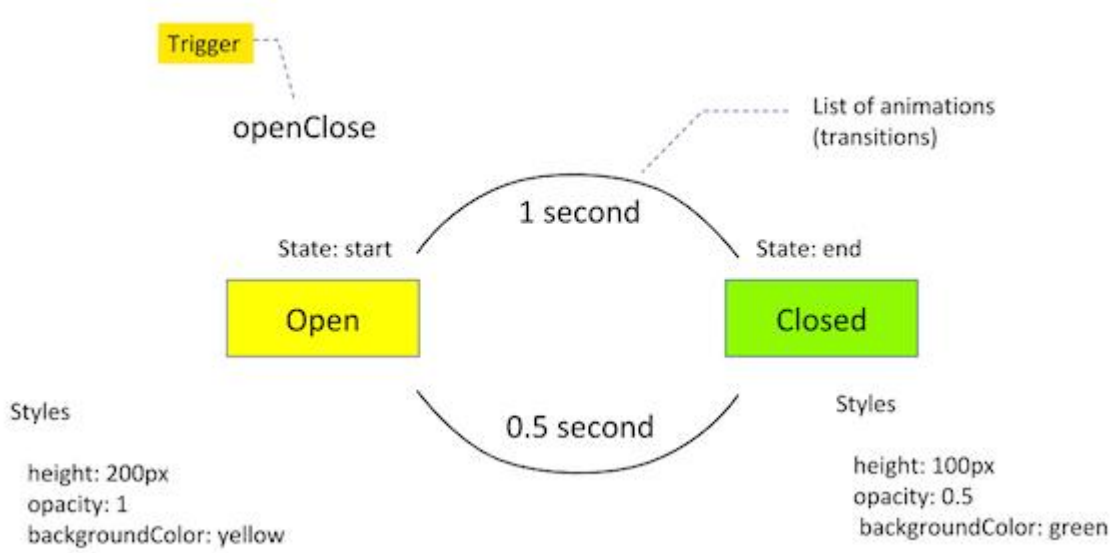
```
transition('closed => open', [
        animate('0.5s')
      ]),
```

Triggering the animation

An animation requires a trigger, so that it knows when to start. The trigger() function collects the states and transitions, and gives the animation a name, so that you can attach it to the triggering element in the HTML template.

The trigger() function describes the property name to watch for changes. When a change occurs, the trigger initiates the actions included in its definition. These actions can be transitions or other functions, as we'll see later on.

In this example, we'll name the trigger openClose, and attach it to the button element. The trigger describes the open and closed states, and the timings for the two transitions.



Defining animations and attaching them to the HTML template

Animations are defined in the metadata of the component that controls the HTML element to be animated. Put the code that defines your animations under the animations: property within the @Component() decorator.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  animations: [
    trigger('openClose', [
      state('open', style({
        height: '200px',
        opacity: 1,
        backgroundColor: 'yellow'
      })),
      state('closed', style({
        height: '100px',
        opacity: 0.5,
        backgroundColor: 'green'
      })),
      transition('open => closed', [
        animate('1s')
      ]),
      transition('closed => open', [
        animate('0.5s')
      ]),
    ])
  ]
})
export class AppComponent {
  isOpen = true;

  toggle() {
    this.isOpen = !this.isOpen;
  }
}
```

When you've defined an animation trigger for a component, you can attach it to an element in that component's template by wrapping the trigger name in brackets and preceding it with an @ symbol. Then, you can bind the trigger to a template expression using standard Angular property binding syntax as shown below, where triggerName is the name of the trigger, and expression evaluates to a defined animation state.

```
<div [@triggerName]="expression">…</div>
```

The animation is executed or triggered when the expression value changes to a new state.

The following code snippet binds the trigger to the value of the isOpen property.

```
<div [@openClose]="isOpen ? 'open' : 'closed'" class="open-close-container">
  <p>The box is now {{ isOpen ? 'Open' : 'Closed' }}!</p>
</div>
```

In this example, when the isOpen expression evaluates to a defined state of open or closed, it notifies the trigger openClose of a state change. Then it's up to the openClose code to handle the state change and kick off a state change animation.

For elements entering or leaving a page (inserted or removed from the DOM), you can make the animations conditional. For example, use *ngIf with the animation trigger in the HTML template.