RANGLE.IO

Rangle's Angular 2 Training Book

A^2

Table of Contents

Rangle's Angular 2 Training Book

Over the last three and a half years, AngularJS has become the leading open source JavaScript application framework for hundreds of thousands of programmers around the world. The "1.x" version of AngularJS has been widely used and became extremely popular for complex applications. The new "Angular 2" version of the framework is currently available as a release candidate.

About Rangle's Angular 2 Training Book

We developed this book to be used as course material for Rangle's Angular 2 training, but many people have found it to be useful for learning Angular 2 on their own. This book will cover the most important Angular 2 topics, from getting started with the Angular 2 toolchain to writing Angular 2 applications in a scalable and maintainable manner.

If you find this material useful, you should also consider registering for one of Rangle's training courses, which facilitate hands-on learning and are a great fit for companies that need to transition their technology to Angular 2, or individuals looking to upgrade their skills.

Rangle.io also has an Angular 1.x book which is geared towards writing Angular 1.x applications in an Angular 2 style. We hope you enjoy this book. We welcome your feedback in the Discussion Area.

License

Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

> This is a human-readable summary of (and not a substitute for) the license.

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Why Angular 2?

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 2 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 2.

Angular 2'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 2 was built on five years of community feedback.

Angular 2 Is Easier

The new Angular 2 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 2 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 2 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 2, the Angular team has added a `.component` method to Angular 1.5, which has been back-ported by community member Todd Motto to v1.3.

TypeScript

Angular 2 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 2 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 2. 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 2 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 2 than another library like React or framework like Ember.

Performance and Mobile

Angular 2 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 2.

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

Angular 2, 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 2 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 2 version of their product, providing *another* way to leverage native device features with Angular 2.

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 2 are:

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

Differences Between Angular 1 & 2

> 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 2
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

EcmaScript 6 and TypeScript Features

Figure: ES6 and TypeScript

The language we usually call "JavaScript" is formally known as "EcmaScript". The new version of JavaScript, "EcmaScript 6" or "ES6", offers a number of new features that extend the power of the language. ES6 is not widely supported in today's browsers, so it needs to be transpiled to ES5. You can choose between several transpilers, but we'll be using TypeScript, which is what the Angular team uses to write Angular 2. Angular 2 makes use of a number of features of ES6 and TypeScript.

ES6

JavaScript was created in 1995, but the language is still thriving today. There are subsets, supersets, current versions and the latest version, ES6, that brings a lot of new features.

Some of the highlights:

- Classes
- Arrow Functions
- Template Strings
- Inheritance
- Constants and Block Scoped Variables
- ...spread and ...rest
- Destructuring
- Modules

Classes

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

```
class Hamburger {
  constructor() {
    // This is the constructor.
  }
  listToppings() {
    // This is a method.
  }
}
```

Traditional class-based languages often reserve the word `this` to reference the current (runtime) instance of the class. This is also true in JavaScript, but JavaScript code can *optionally* supply `this` to a method at call time.

A Refresher on `this`

Inside a JavaScript class we'll be using `this` keyword to refer to the instance of the class. E.g., consider this case:

```
class Toppings {
  ...

  formatToppings() { /* implementation details */ }

  list() {
    return this.formatToppings(this.toppings);
  }
}
```

Here `this` refers to an instance of the `Toppings` class. As long as the `list` method is called using dot notation, like `myToppings.list()`, then `this.formatToppings(this.toppings)` invokes the `formatToppings()` method defined on the instance of the class. This will also ensure that inside `formatToppings`, `this` refers to the same instance.

However, `this` can also refer to other things. There are two basic cases that you should remember.

1. Method invocation:

   ```
   someObject.someMethod();
   ```

   Here, `this` used inside `someMethod` will refer to `someObject`, which is usually what you want.

2. Function invocation:

   ```
   someFunction();
   ```

   Here, `this` used inside `someFunction` can refer to different things depending on whether we are in "strict" mode or not. Without using the "strict" mode, `this` refers to the context in which `someFunction()` was called. This rarely what you want, and it can be confusing when `this` is not what you were expecting, because of where the function was called from. In "strict" mode, `this` would be undefined, which is slightly less confusing.

View Example

One of the implications is that you cannot easily detach a method from its object. Consider this example:

```
var log = console.log;
log('Hello');
```

In many browsers this will give you an error. That's because `log` expects `this` to refer to `console`, but the reference was lost when the function was detached from `console`.

This can be fixed by setting `this` explicitly. One way to do this is by using `bind()` method, which allows you to specify the value to use for `this` inside the bound function.

```
var log = console.log.bind(console);
log('Hello');
```

You can also achieve the same using `Function.call` and `Function.apply`, but we won't discuss this here.

Another instance where `this` can be confusing is with respect to anonymous functions, or functions declared within other functions. Consider the following:

```
class ServerRequest {
  notify() {
    ...
  }
  fetch() {
    getFromServer(function callback(err, data) {
       this.notify(); // this is not going to work
    });
  }
}
```

In the above case `this` will *not* point to the expected object: in "strict" mode it will be `undefined`. This leads to another ES6 feature - arrow functions, which will be covered next.

Arrow Functions

ES6 offers some new syntax for dealing with `this` : "arrow functions".
Arrow functions also make higher order functions much easier to work with.

The new "fat arrow" notation can be used to define anonymous functions in a simpler way.

Consider the following example:

```
items.forEach(function(x) {
  console.log(x);
  incrementedItems.push(x+1);
});
```

This can be rewritten as an "arrow function" using the following syntax:

```
items.forEach((x) => {
  console.log(x);
  incrementedItems.push(x+1);
});
```

Functions that calculate a single expression and return its values can be defined even simpler:

```
incrementedItems = items.map((x) => x+1);
```

The latter is *almost* equivalent to the following:

```
incrementedItems = items.map(function (x) {
  return x+1;
});
```

There is one important difference, however: arrow functions do not set a local copy of `this` , `arguments` , `super` , or `new.target` . When `this` is used inside an arrow function JavaScript uses the `this` from the outer scope. Consider the following example:

```
class Toppings {
  constructor(toppings) {
    this.toppings = Array.isArray(toppings) ? toppings : [];
  }
  outputList() {
    this.toppings.forEach(function(topping, i) {
      console.log(topping, i + '/' + this.toppings.length);  // no this
    })
  }
}


var ctrl = new Toppings(['cheese', 'lettuce']);


ctrl.outputList();
```

Let's try this code on ES6 Fiddle (http://www.es6fiddle.net/). As we see, this gives us an error, since `this` is undefined inside the anonymous function.

Now, let's change the method to use the arrow function:

```
class Toppings {
  constructor(toppings) {
    this.toppings = Array.isArray(toppings) ? toppings : [];
  }
  outputList() {
    this.toppings
      .forEach((topping, i) => console
        .log(topping, i + '/' + this.toppings.length)  // `this` works!
    )
  }
}


var ctrl = new Toppings(['cheese', 'lettuce']);
```

Here `this` inside the arrow function refers to the instance variable.

Warning arrow functions do *not* have their own `arguments` variable, which can be confusing to veteran JavaScript programmers. `super` and `new.target` are also scoped from the outer enclosure.

Template Strings

In traditional JavaScript, text that is enclosed within matching `"` or `'` marks is considered a string. Text within double or single quotes can only be on one line. There was no way to insert data into these strings. This resulted in a lot of ugly concatenation code that looked like:

```javascript
var name = 'Sam';
var age = 42;

console.log('hello my name is ' + name + ' I am ' + age + ' years old');
```

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

```javascript
var name = 'Sam';
var age = 42;

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

There are all sorts of places where this kind of string can come in handy, and front-end web development is one of them.

Inheritance

JavaScript's inheritance works differently from inheritance in other languages, which can be very confusing. ES6 classes provide a syntactic sugar attempting to alleviate the issues with using prototypical inheritance present in ES5. Our recommendation is still to avoid using inheritance or at least deep inheritance hierarchies. Try solving the same problems through delegation instead.

Constants and Block Scoped Variables

ES6 introduces the concept of block scoping. Block scoping will be familiar to programmers from other languages like C, Java, or even PHP.
In ES5 JavaScript and earlier, `var` s are scoped to `function` s, and they can "see" outside their functions to the outer context.

```
var five = 5;
var threeAlso = three; // error

function scope1() {
  var three = 3;
  var fiveAlso = five; // == 5
  var sevenAlso = seven; // error
}

function scope2() {
  var seven = 7;
  var fiveAlso = five; // == 5
  var threeAlso = three; // error
}
```

In ES5 functions were essentially containers that could be "seen" out of, but not into.

In ES6 `var` still works that way, using functions as containers, but there are two new ways to declare variables: `const` and `let` .

`const` and `let` use `{` and `}` blocks as containers, hence "block scope". Block scoping is most useful during loops. Consider the following:

```
var i;
for (i = 0; i < 10; i += 1) {
  var j = i;
  let k = i;
}
console.log(j); // 9
console.log(k); // undefined
```

Despite the introduction of block scoping, functions are still the preferred mechanism for dealing with most loops.

`let` works like `var` in the sense that its data is read/write. `let` is also useful when used in a for loop. For example, without let, the following example would output `5,5,5,5,5` :

```
for(var x=0; x<5; x++) {
  setTimeout(()=>console.log(x), 0)
}
```

However, when using `let` instead of `var`, the value would be scoped in a way that people would expect.

```
for(let x=0; x<5; x++) {
  setTimeout(()=>console.log(x), 0)
}
```

Alternatively, `const` is read-only. Once `const` has been assigned, the identifier cannot be reassigned; however the value itself is still mutable. For example:

```
const myName = 'pat';
let yourName = 'jo';

yourName = 'sam'; // assigns
myName = 'jan';   // error
```

The read-only nature can be demonstrated with any object:

```
const literal = {};

literal.attribute = 'test'; // fine
literal = []; // error;
```

...spread and ...rest

Spread takes a collection of something, like `[]` s or `{}` s, and applies them to something else that accepts `,` separated arguments, like `function` s, `[]` s and `{}` s.

For example:

```
const add = (a, b) => a + b;
let args = [3, 5];
add(...args); // same as `add(args[0], args[1])`, or `add.apply(null, args)`
```

Functions aren't the only place in JavaScript that makes use of comma separated lists - `[]` s can now be concatenated with ease:

```
let cde = ['c', 'd', 'e'];
let scale = ['a', 'b', ...cde, 'f', 'g']; // ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

Similarly, object literals can do the same thing:

```
let mapABC  = { a: 5, b: 6, c: 3};
let mapABCD = { ...mapABC, d: 7}; // { a: 5, b: 6, c: 3, d: 7 }
```

...rest arguments share the ellipsis like syntax of rest operators but are used for a different purpose. ...rest arguments are used to access a variable number of arguments passed to a function. For example:

```
function addSimple(a, b) {
  return a + b;
}

function add(...numbers) {
  return numbers[0] + numbers[1];
}

addSimple(3, 2);  // 5
add(3, 2);        // 5

// or in es6 style:
const addEs6 = (...numbers) => numbers.reduce((p, c) => p + c, 0);

addEs6(1, 2, 3);  // 6
```

Technically JavaScript already had an `arguments` variable set on each function (except for arrow functions), however `arguments` has a lot of issues, one of which is the fact that it is not technically an array.

...rest arguments are in fact arrays. The other important difference is that rest arguments only include arguments not specifically named in a function like so:

```javascript
function print(a, b, c, ...more) {
  console.log(more[0]);
  console.log(arguments[0]);
}

print(1, 2, 3, 4, 5);
// 4
// 1
```

Destructuring

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

To borrow from the MDN, destructuring can be used to turn the following:

```
let foo = ['one', 'two', 'three'];

let one   = foo[0];
let two   = foo[1];
let three = foo[2];
```

into

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

This is pretty interesting, but at first it might be hard to see the use case. ES6 also supports object destructuring, which might make uses more obvious:

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

let {drawSquare, drawText} = myModule;

drawSquare(5);
drawText('hello');
```

Destructuring can also be used for passing objects into a function, allowing you to pull specific properties out of an object in a concise manner. It is also possible to assign default values to destructured arguments, which can be a useful pattern if passing in a configuration object.

```
let jane = { firstName: 'Jane', lastName: 'Doe'};
let john = { firstName: 'John', lastName: 'Doe', middleName: 'Smith' }
function sayName({firstName, lastName, middleName = 'N/A'}) {
  console.log(`Hello ${firstName} ${middleName} ${lastName}`)
}


sayName(jane) // -> Hello Jane N/A Doe
sayName(john) // -> Helo John Smith Doe
```

There are many more sophisticated things that can be done with destructuring, and the MDN has some great examples, including nested object destructuring and dynamic destructuring with `for ... in` operators".

ES6 Modules

ES6 also introduces the concept of a module, which works similar to other languages. Defining an ES6 module is quite easy: each file is assumed to define a module and we specify its exported values using the `export` keyword.

Loading ES6 modules is a little trickier. In an ES6-compliant browser you use the `System` keyword to load modules asynchronously. To make our code work with current browsers, however, we will use the SystemJS library as a polyfill:

```html
<script src="/node_module/systemjs/dist/system.js"></script>
<script>
  var promise = System.import('app')
    .then(function() {
      console.log('Loaded!');
    })
    .then(null, function(error) {
      console.error('Failed to load:', error);
    });
</script>
```

TypeScript

ES6 is the current version of JavaScript. TypeScript is a superset of ES6, which means all ES6 features are part of TypeScript, but not all TypeScript features are part of ES6. Consequently, TypeScript must be transpiled into ES5 to run in most browsers.

One of TypeScript's primary features is the addition of type information, hence the name. This type information can help make JavaScript programs more predictable and easier to reason about.

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

Without TS:

```
function add(a, b) {
  return a + b;
}

add(1, 3);   // 4
add(1, '3'); // '13'
```

With TS:

```
function add(a: number, b: number) {
  return a + b;
}

add(1, 3);   // 4
// compiler error before JS is even produced
add(1, '3'); // '13'
```

Getting Started With TypeScript

Install the TypeScript transpiler using npm:

```
$ npm install -g typescript
```

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

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

Note About ES6 Examples

Our earlier ES6 class won't compile now. TypeScript is more demanding than ES6 and it expects instance properties to be declared:

```
class Pizza {
  toppings: string[];
  constructor(toppings: string[]) {
    this.toppings = toppings;
  }
}
```

Note that now that we've declared `toppings` to be an array of strings, TypeScript will enforce this. If we try to assign a number to it, we will get an error at compilation time.

If you want to have a property that can be set to a value of any type, however, you can still do this: just declare its type to be "any":

```
class Pizza {
  toppings: any;
  //...
}
```

Working With `tsc`

So far `tsc` has been used to compile a single file. Typically programmers have a lot more than one file to compile. Thankfully `tsc` can handle multiple files as arguments.

Imagine two ultra simple files/modules:

a.ts

```
export const A = (a) => console.log(a);
```

b.ts

```
export const B = (b) => console.log(b);
```

```
$ tsc ./a.ts ./b.ts
a.ts(1,1): error TS1148: Cannot compile modules unless the '--module' flag is provided
.
```

Hmmm. What's the deal with this module flag? TypeScript has a help menu, let's take a look:

```
$ tsc --help | grep module
 -m KIND, --module KIND           Specify module code generation: 'commonjs', 'amd',
 'system', 'umd' or 'es2015'
 --moduleResolution               Specifies module resolution strategy: 'node' (Node
.js) or 'classic' (TypeScript pre-1.6).
```

(TypeScript has more help than what we've shown; we filtered by `grep` for brevity.) There are two help entries that reference "module", and `--module` is the one TypeScript was complaining about. The description explains that TypeScript supports a number of different module schemes. For the moment `commonjs` is desirable. This will produce modules that are compatible with node.js's module system.

```
$ tsc -m commonjs ./a.ts ./b.ts
```

`tsc` should produce no output. In many command line traditions, no output is actually a mark of success. Listing the directory contents will confirm that our TypeScript files did in fact compile.

```
$ ls
a.js    a.ts    b.js    b.ts
```

Excellent - there are now two JavaScript modules ready for consumption.

Telling the `tsc` command what to compile becomes tedious and labor intensive even on small projects. Fortunately TypeScript has a means of simplifying this. `tsconfig.json` files let programmers write down all the compiler settings they want. When `tsc` is run, it looks for `tsconfig.json` files and uses their rules to compile JavaScript.

For Angular 2 projects there are a number of specific settings that need to be configured in a project's `tsconfig.json`

```json
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "noImplicitAny": false,
    "removeComments": false,
    "sourceMap": true
  },
  "exclude": [
    "node_modules",
    "dist/"
  ]
}
```

Target

The compilation target. TypeScript supports targeting different platforms depending on your needs. In our case, we're targeting modern browsers which support ES5.

Module

The target module resolution interface. We're integrating TypeScript through webpack which supports different interfaces. We've decided to use node's module resolution interface, `commonjs`.

Decorators

Decorator support in TypeScript hasn't been finalized yet but since Angular 2 uses decorators extensively, these need to be set to true. Decorators have not been introduced yet, and will be covered later in this section.

TypeScript with Webpack

We won't be running `tsc` manually, however. Instead, webpack's `ts-loader` will do the transpilation during the build:

```
// webpack.config.js
//...
loaders: [
  { test: /\.ts$/, loader: 'ts', exclude: /node_modules/ },
  //...
]
```

This loader calls `tsc` for us, and it will use our `tsconfig.json`.

Typings

Astute readers might be wondering what happens when TypeScript programmers need to interface with JavaScript modules that have no type information. TypeScript recognizes files labelled `*.d.ts` as *definition* files. These files are meant to use TypeScript to describe interfaces presented by JavaScript libraries.

There are communities of people dedicated to creating typings for JavaScript projects. There is also a utility called `typings` (`npm install --save-dev typings`) that can be used to manage third party typings from a variety of sources.

Linting

Many editors support the concept of "linting" - a grammar check for computer programs. Linting can be done in a programmer's editor and/or through automation.

For TypeScript there is a package called `tslint`, (`npm install --save-dev tslint`) which can be plugged into many editors. `tslint` can also be configured with a `tslint.json` file.

Webpack can run `tslint` before it attempts to run `tsc`. This is done by installing `tslint-loader` (`npm install --save-dev tslint-loader`) which plugs into webpack like so:

```
// ...
module: {
  preLoaders: [
    { test: /\.ts$/, loader: 'tslint' }
  ],
  loaders: [
    { test: /\.ts$/, loader: 'ts', exclude: /node_modules/ },
    // ...
  ]
  // ...
}
```

TypeScript Features

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

- Types
- Interfaces
- Shapes
- Decorators

Types

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

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

TypeScript also adds

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

Primitive type example:

```typescript
let isDone: boolean = false;
let height: number = 6;
let name: string = "bob";
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];
enum Color {Red, Green, Blue};
let c: Color = Color.Green;
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean

function showMessage(data: string): void {
  alert(data);
}
showMessage('hello');
```

This illustrates the primitive types in TypeScript, and ends by illustrating a `showMessage` function. In this function the parameters have specific types that are checked when `tsc` is run.

In many JavaScript functions it's quite common for functions to take optional parameters. TypeScript provides support for this, like so:

```typescript
function logMessage(message: string, isDebug?: boolean) {
  if (isDebug) {
    console.log('Debug: ' + message);
  } else {
    console.log(message);
  }
}
logMessage('hi');         // 'hi'
logMessage('test', true); // 'Debug: test'
```

Using a `?` lets `tsc` know that `isDebug` is an optional parameter. `tsc` will not complain if `isDebug` is omitted.

TypeScript Classes

TypeScript also treats `class` es as their own type:

```typescript
class Foo { foo: number; }
class Bar { bar: string; }

class Baz {
  constructor(foo: Foo, bar: Bar) { }
}

let baz = new Baz(new Foo(), new Bar()); // valid
baz = new Baz(new Bar(), new Foo());     // tsc errors
```

Like function parameters, `class` es sometimes have optional members. The same `?:` syntax can be used on a `class` definition:

```typescript
class Person {
  name: string;
  nickName?: string;
}
```

In the above example, an instance of `Person` is guaranteed to have a `name`, and might optionally have a `nickName`

Interfaces

Sometimes classes are "more" than a programmer wants. Classes end up creating code, in the form of transpiled ES6 classes or transpiled ES5 constructor functions.

Also, JavaScript is a *subset* of TypeScript, and in JavaScript functions are "first class" (they can be assigned to variables and passed around), so how can functions be described in TypeScript?

TypeScript's interfaces solve both of these problems. Interfaces are abstract descriptions of things, and can be used to represent any non-primitive JavaScript object. They produce no code (ES6 or ES5), and exist only to describe types to `tsc`.

Here is an example of an interface describing a function:

```typescript
interface Callback {
  (error: Error, data: any): void;
}

function callServer(callback: Callback) {
  callback(null, 'hi');
}
callServer((error, data) => console.log(data)); // 'hi'
callServer('hi');                               // tsc error
```

Sometimes JavaScript functions are "overloaded" - that is, they can have different call signatures. Interfaces can be used to specify this. (Methods in classes can also be overloaded):

```typescript
interface PrintOutput {
  (message: string): void;    // common case
  (message: string[]): void;  // less common case
}

let printOut: PrintOutput = (message) => {
  if (Array.isArray(message)) {
    console.log(message.join(', '));
  } else {
    console.log(message);
  }
}

printOut('hello');        // 'hello'
printOut(['hi', 'bye']); // 'hi, bye'
```

Here is an example of an interface describing an object literal:

```
interface Action {
  type: string;
}

let a: Action = {
    type: 'literal'
}
```

Shapes

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

Consider an expansion of the previous example:

```typescript
interface Action {
  type: string;
}

let a: Action = {
    type: 'literal'
}

class NotAnAction {
  type: string;
  constructor() {
    this.type = 'Constructor function (class)';
  }
}

a = new NotAnAction(); // valid TypeScript!
```

Despite the fact that `Action` and `NotAnAction` have different identifiers, `tsc` lets us assign an instance of `NotAnAction` to `a` which has a type of `Action` . This is because TypeScript only really cares that objects have the same shape. In other words if two objects have the same attributes, with the same typings, those two objects are considered to be of the same type.

Type Inference

One common misconception about TypeScript's types is that code needs to explicitly describe types at every possible opportunity. Fortunately this is not the case. TypeScript has a rich type inference system that will "fill in the blanks" for the programmer. Consider the following:

type-inference-finds-error.ts

```
let numbers = [2, 3, 5, 7, 11];
numbers = ['this will generate a type error'];
```

```
tsc ./type-inference-finds-error.ts
type-inference-finds-error.ts(2,1): error TS2322: Type 'string[]' is not assignable to
type 'number[]'.
  Type 'string' is not assignable to type 'number'.
```

The code contains no extra type information. In fact, it's valid ES6.
If `var` had been used, it would be valid ES5. Yet TypeScript is still able to determine type information.

Type inference can also work through context, which is handy with callbacks. Consider the following:

type-inference-finds-error-2.ts

```
interface FakeEvent {
  type: string;
}

interface FakeEventHandler {
  (e: FakeEvent): void;
}

class FakeWindow {
  onMouseDown: FakeEventHandler
}
const fakeWindow = new FakeWindow();

fakeWindow.onMouseDown = (a: number) => {
  // this will fail
};
```

```
tsc ./type-inference-finds-error-2.ts
type-inference-finds-error-2.ts(14,1): error TS2322: Type '(a: number) => void' is not
 assignable to type 'FakeEventHandler'.
  Types of parameters 'a' and 'e' are incompatible.
    Type 'number' is not assignable to type 'FakeEvent'.
      Property 'type' is missing in type 'Number'.
```

In this example the context is not obvious since the interfaces have been defined explicitly. In a browser environment with a real `window` object, this would be a handy feature, especially the type completion of the `Event` object.

Decorators

Decorators are proposed for a future version of JavaScript, but the Angular 2 team *really* wanted to use them, and they have been included in TypeScript.

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

Decorators are quite new in TypeScript, and most use cases demonstrate the use of existing decorators. However, decorators are just functions, and are easier to reason about after walking through a few examples.

Decorators are functions, and there are four things (`class`, parameter, method and property) that can be decorated; consequently there are four different function signatures for decorators:

- class: `declare type ClassDecorator = <TFunction extends Function>(target: TFunction) => TFunction | void;`
- property: `declare type PropertyDecorator = (target: Object, propertyKey: string | symbol) => void;`
- method: `declare type MethodDecorator = <T>(target: Object, propertyKey: string | symbol, descriptor: TypedPropertyDescriptor<T>) => TypedPropertyDescriptor<T> | void;`
- parameter: `declare type ParameterDecorator = (target: Object, propertyKey: string | symbol, parameterIndex: number) => void;`

Readers who have played with Angular 2 will notice that these signatures do not look like the signatures used by Angular 2 specific decorators like `@Component()`.

Notice the `()` on `@Component`. This means that the `@Component` is called once JavaScript encounters `@Component()`. In turn, this means that there must be a `Component` function somewhere that returns a function matching one of the decorator signatures outlined above. This is an example of the decorator factory pattern.

If decorators still look confusing, perhaps some examples will clear things up.

Property Decorators

Property decorators work with properties of classes.

```typescript
function Override(label: string) {
  return function (target: any, key: string) {
    Object.defineProperty(target, key, {
      configurable: false,
      get: () => label
    });
  }
}

class Test {
  @Override('test')      // invokes Override, which returns the decorator
  name: string = 'pat';
}

let t = new Test();
console.log(t.name);  // 'test'
```

The above example must be compiled with both the `--experimentalDecorators` and `--emitDecoratorMetadata` flags.

In this case the decorated property is replaced by the `label` passed to the decorator. It's important to note that property values cannot be directly manipulated by the decorator; instead an accessor is used.

Here's a classic property example that uses a *plain decorator*

```typescript
function ReadOnly(target: any, key: string) {
  Object.defineProperty(target, key, { writable: false });
}

class Test {
  @ReadOnly            // notice there are no `()`
  name: string;
}

const t = new Test();
t.name = 'jan';
console.log(t.name); // 'undefined'
```

In this case the name property is not `writable`, and remains undefined.

Class Decorators

```typescript
function log(prefix?: string) {
  return (target) => {
    // save a reference to the original constructor
    var original = target;

    // a utility function to generate instances of a class
    function construct(constructor, args) {
      var c: any = function () {
        return constructor.apply(this, args);
      }
      c.prototype = constructor.prototype;
      return new c();
    }

    // the new constructor behavior
    var f: any = function (...args) {
      console.log(prefix + original.name);
      return construct(original, args);
    }

    // copy prototype so instanceof operator still works
    f.prototype = original.prototype;

    // return new constructor (will override original)
    return f;
  };
}

@log('hello')
class World {
}

const w = new World(); // outputs "helloWorld"
```

In the example `log` is invoked using `@`, and passed a string as a parameter, `@log()` returns an anonymous function that is the actual decorator.

The decorator function takes a `class`, or constructor function (ES5) as an argument. The decorator function then returns a new class construction function that is used whenever `World` is instantiated.

This decorator does nothing other than log out its given parameter, and its `target`'s class name to the console.

Parameter Decorators

```
function logPosition(target: any, propertyKey: string, parameterIndex: number) {
  console.log(parameterIndex);
}

class Cow {
  say(b: string, @logPosition c: boolean) {
    console.log(b);
  }
}

new Cow().say('hello', false); // outputs 1 (newline) hello
```

The above demonstrates decorating method parameters. Readers familiar with Angular 2 can now imagine how Angular 2 implemented their `@Inject()` system.

The JavaScript Toolchain

In this section, we'll describe the tools that you'll be using for the rest of the course.

Figure: Hand Tools by M338 is licensed under Public Domain
(http://commons.wikimedia.org/wiki/File:Hand_tools.jpg)

Source Control: Git

`git` is a distributed versioning system for source code. It allows programmers to collaborate on the same codebase without stepping on each other's toes. It has become the de-facto source control system for open source development because of its decentralized model and cheap branching features.

The Command Line

JavaScript development tools are very command line oriented. If you come from a Windows background you may find this unfamiliar. However the command line provides better support for automating development tasks, so it's worth getting comfortable with it.

We will provide examples for all command line activities required by this course.

Command Line JavaScript: NodeJS

NodeJS is an environment that lets you write JavaScript programs that live outside the browser. It provides:

- the V8 JavaScript interpreter
- modules for doing OS tasks like file I/O, HTTP, etc.

While NodeJS was initially intended for writing server code in JavaScript, today it is widely used by JavaScript tools, which makes it relevant to front-end programmers too. A lot of the tools you'll be using in this course leverage NodeJS.

Back-End Code Sharing and Distribution: npm

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

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

Module Loading, Bundling and Build Tasks: Webpack

Webpack takes modules with dependencies and generates static assets representing those modules. It can bundle JavaScript, CSS, HTML or just about anything via additional loaders. Webpack can also be extended via plugins, for example minification and mangling can be done using the UglifyJS plugin for webpack.

Chrome

We use Google's Chrome browser for this course because of its cutting-edge JavaScript engine and excellent debugging tools.

However, code written with AngularJS should work on any modern web browser (Firefox, IE9+, Chrome, Safari).

Bootstrapping an Angular 2 Application

Bootstrapping is an essential process in Angular - it is where the application is built, and where Angular comes to life. Bootstrapping Angular 2 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 2 application with a single component. To do this we'll need the following files:

- *app/app.component.ts* - this is where we define our root component
- *index.html* - this is the page the component will be rendered in
- *app/boot.ts* - is the glue that combines the component and page together

app/app.component.ts

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

@Component({
    selector: 'app',
    template: '<b>Bootstrapping an Angular 2 Application</b>'
})

export class MyApp {}
```

index.html

```html
...
<body>
    <app>Loading...</app>
</body>
...
```

app/boot.ts

```typescript
import {bootstrap} from '@angular/platform/browser'
import {MyApp} from './app.component'

bootstrap(MyApp);
```

[View Example](#)

This is the main entry point of the application. The `App` component operates as the root component of our entire application and will be rendered on any `app` HTML element encountered. There is an `app` HTML element in the *index.html* file, and we use *app/boot.ts*

to import the `MyApp` component and the `bootstrap` function and kickstart the bootstrapping process, which will read the `App` metadata and then load the application wherever the `app` selector/tag-name is found.

Calling `bootstrap` will return a `Promise` that you can use to determine when the bootstrapping routine has completed. This is useful if you want to incorporate an initial UI loading state into your application.

Why does Angular 2 bootstrap itself in this way? Well there is actually a very good reason. Since Angular 2 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 2 applications. The magic is then in our bootstrapping process - we can import which `bootstrap` routine we would like to use, depending on the environment we're operating under. In our example, since we were running our Angular 2 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 *boot.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.

Bootstrapping Providers

Calling `bootstrap` also starts the dependency injection system in Angular 2. We won't go over Angular 2'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.

```
import {bootstrap} from '@angular/platform-browser-dynamic'
import {MyProvider} from './myprovider'
import {App} from './app.component'

bootstrap(MyApp, [MyProvider]);
```

[View Example](#)

We import our root Component, `App`, `bootstrap` and a custom provider, `MyProvider`. When we bootstrap our root component we can pass in application-wide providers that will be injected to any component that wants to use them.

Components in Angular 2
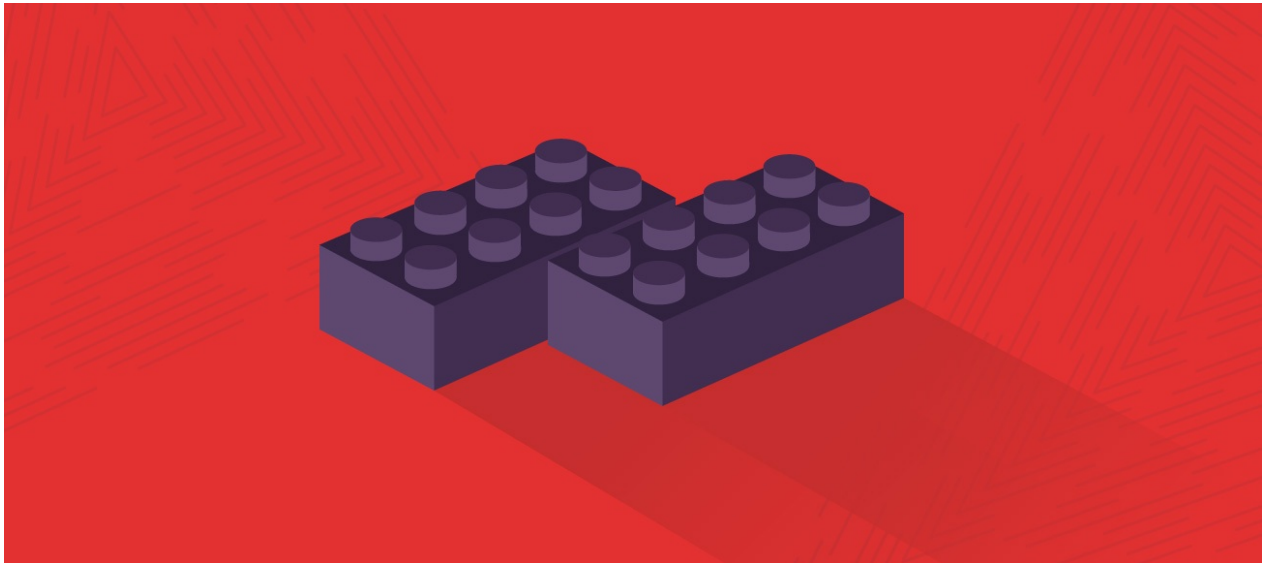
Figure: components

The core concept of any Angular 2 application is the *component*. In effect, the whole application can be modeled as a tree of these components.

This is how the Angular 2 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.

In Angular 1.x we had router views and directives which worked sort of like components. The idea of directive components became quite popular. They were created by using `directive` with a controller while relying on the `controllerAs` and `bindToController` properties. For example:

```
angular.module('ngcourse')
  .directive('ngcHelloComponent', () => ({
      restrict: 'E',
      scope: { name: '=' },
      template: '<span>Hello, {{ ctrl.name }}.</span>',
      controller: MyComponentCtrl,
      controllerAs: 'ctrl',
      bindToController: true
    })
  );
```

In fact, this concept became so popular that in Angular 1.5 the `.component` method was
introduced as syntactic sugar.

```
angular.module('ngcourse')
  .component('ngcHelloComponent', {
    bindings: { name: '=' },
    template: '<span>Hello, {{ $ctrl.name }}.</span>',
    controller: MyComponentCtrl
  });
```

Creating Components

Components in Angular 2 build upon this idea. We define a component's application logic inside a class. To this we then attach a selector and a template.

- *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 how to render this component.

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

@Component({
    selector: 'hello',
    template: '<p>Hello, {{name}}</p>'
})
export class Hello {
  name: string;

  constructor() {
    this.name = 'World';
  }
}
```

To use this component we simply add `<hello></hello>` to our HTML, and Angular will insert an instance of the `Hello` view between those tags.

View Example

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:

```
<TodoApp>
  <TodoList>
    <TodoItem></TodoItem>
    <TodoItem></TodoItem>
    <TodoItem></TodoItem>
  </TodoList>
  <TodoForm></TodoForm>
</TodoApp>
```

At the root we have `TodoApp` which consists of a `TodoList` and a `TodoForm` . Within the list we have several `TodoItem` s. Each of these components is visible to the user, who can interact with these components and perform actions.

Passing Data into a Component

The `inputs` attribute defines a set of parameters that can be passed down from the component's parent. For example, we can modify the `Hello` component so that `name` can be configured by the parent.

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

@Component({
  selector: 'hello',
  inputs: ['name'],
  template: '<p>Hello, {{name}}</p>'
})
export class Hello {
  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 -->
<hello name="World"></hello>
<!-- To bind to a variable in the parent scope -->
<hello [name]="name"></hello>
```

View Example

> Unlike Angular 1.x, this is one-way binding.

Responding to Component Events

Events in Angular 2 work similar to how they worked in Angular 1.x. The big change is template syntax.

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

@Component({
  selector: 'counter',
  template: `
    <div>
      <p>Count: {{ num }}</p>
      <button (click)="increment()">Increment</button>
    </div>
  `
})
export class Counter {
  num: number = 0;

  increment() {
    this.num++;
  }
}
```

[View Example](#)

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.

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

@Component({
  selector: 'counter',
  inputs: ['count'],
  outputs: ['result'],
  template: `
    <div>
      <p>Count: {{ count }}</p>
      <button (click)="increment()">Increment</button>
    </div>
  `
})
export default class Counter {
  count: number = 0;
  result: EventEmitter<number> = new EventEmitter<number>();

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

View Example

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`, for example:

```typescript
@Component({/*....*/})
export default class Counter {
  @Input() count: number = 0;
  @Output() countChange: EventEmitter<number> = new EventEmitter<number>();

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

@Component({
  template:'<counter [(count)]="myNumber"></counter>'
  directives:[Counter]
})
class SomeComponent {
  // ...
}
```

View Example

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/my-example.component.html

```html
<section >
  <form #myForm="ngForm" (ngSubmit)="submitForm(myForm)">
    <!-- ... -->
  </form>
</section>
```

app/my-example.component.ts

```typescript
// ...
import {NgForm} from '@angular/forms';
// ...
export class MyExampleComponent {
  submitForm (form: NgForm) {
    // ...
  }
}
```

View Example

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.

```html
<red-ball #myBall></red-ball>
The ball is {{ myBall.color }}.
```

```
@Component({
  selector: 'red-ball',
  template: '<div > · </div>'
})
export class RedBallComponent {
  color: string = 'red'
}
```

[View Example](#)

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.

```
@Component({
  selector: 'red-ball',
  template: '<div > · </div>'
```

Projection

Components by default support projection. You can use the `ngContent` directive to place the projected content in your template.

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

@Component({
  selector: 'child',
  template: `
    <h4>Child Component</h4>
    <ng-content></ng-content>
  `
})
class Child {}
```

[View Example](#)

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. For example, `TodoList` relies on `TodoItem` . To let a component know about the dependent components we use the directive attribute.

```typescript
import {Component} from '@angular/core';
import {TodoInput} from './components/todo-input';
import {TodoList} from './components/todo-list';

@Component({
  selector: 'todo-app',
  directives: [TodoInput, TodoList],
  template: `...`
})
export class TodoApp {}
```

The same idea applies to `pipes` .

Directives

Directives are entities that change the behavior of components or elements and are one of the core building blocks Angular 2 uses to build applications. In fact, Angular 2 components are in large part directives with templates. This is why components are passed in as children through the directives property.

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

There are two main types of directives in Angular 2:

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

Attribute Directives

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

For example, Angular 2 has built-in attribute directives such as `ngClass` and `ngStyle` that work on any component or element.

NgStyle Directive

Angular 2 provides a built-in directive, `ngStyle` , to modify a component or element's `style` attribute. Here's an example:

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

View Example

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 2 `ngStyle` directive are combined when styling the element.

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: '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 {
}
```

[View Example](#)

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: '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 {
}
```

[View Example](#)

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 2 applies each property name of that object to the component if that property is true.

```
@Component({
  selector: '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;
}
```

[View Example](#)

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 2 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: '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: '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 2 also has a built-in `template` directive that does the same thing:

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

NgIf Directive

The `ngIf` directive conditionally renders components or elements 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',
  directives: [IfExampleComponent],
  template: `
    <button type="button" (click)="toggleExists()">Toggle Component</button>
    <hr/>
    <if-example *ngIf="exists">
      Hello
    </if-example>
  `
})
export class AppComponent {
  exists: boolean = true;

  toggleExists() {
    this.exists = !this.exists;
  }
}
```

[View Example](#)

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',
  directives: [ForExampleComponent],
  template: `
    <for-example *ngFor="let episode of episodes" [episode]="episode">
      {{episode.title}}
    </for-example>
  `
})
export class AppComponent {
  episodes: any[] = [
    { 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' }
  ];
}
```

[View Example](#)

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',
  directives: [ForExampleComponent],
  template: `
    <template ngFor [ngForOf]="episodes" let-episode>
      <for-example [episode]="episode">
        {{episode.title}}
      </for-example>
    </template>
  `
})
```

[View Example](#)

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. `ngFor` also provides some other values that can be bound to:

- *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',
  directives: [ForExampleComponent],
  template: `
    <for-example
      *ngFor="let episode of episodes; let i = index; let isOdd = odd"
      [episode]="episode"
      [ngClass]="{ odd: isOdd }">
      {{i+1}}. {{episode.title}}
    </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 2 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 2 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',
  directives: [ForExampleComponent],
  template: `
  <button (click)="addOtherEpisode()" [disabled]="otherEpisodes.length === 0">Add Epis
ode</button>
    <for-example
      *ngFor="let episode of episodes;
      let i = index; let isOdd = odd;
      trackBy: trackById" [episode]="episode"
      [ngClass]="{ odd: isOdd }">
      {{episode.title}}
    </for-example>


  `
})
export class AppComponent {

  otherEpisodes: any[] = [
    { 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: any[] = [
    { 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)
  }
}
```

View Example

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.

View Example Without trackBy

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',
  directives: [DoorComponent],
  template: `
    <div [ngSwitch]="door">
      <door [id]="1" *ngSwitchCase="1">A new car!</door>
      <door [id]="2" *ngSwitchCase="2">A washer and dryer!</door>
      <door [id]="3" *ngSwitchCase="3">A trip to Tahiti!</door>
      <door [id]="4" *ngSwitchCase="4">25 000 dollars!</door>
      <door *ngSwitchDefault class="closed"></door>
    </div>

    <div class="options">
      <input type="radio" name="door" (click)="setDoor(1)" /> Door 1
      <input type="radio" name="door" (click)="setDoor(2)" /> Door 2
      <input type="radio" name="door" (click)="setDoor(3)" /> Door 3
      <input type="radio" selected="selected" name="door" (click)="setDoor()"/> Close
all
    </div>
  `
})
export class AppComponent {
  door: number;

  setDoor(num: number) {
    this.door = num;
  }
}
```

View Example

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

View Example

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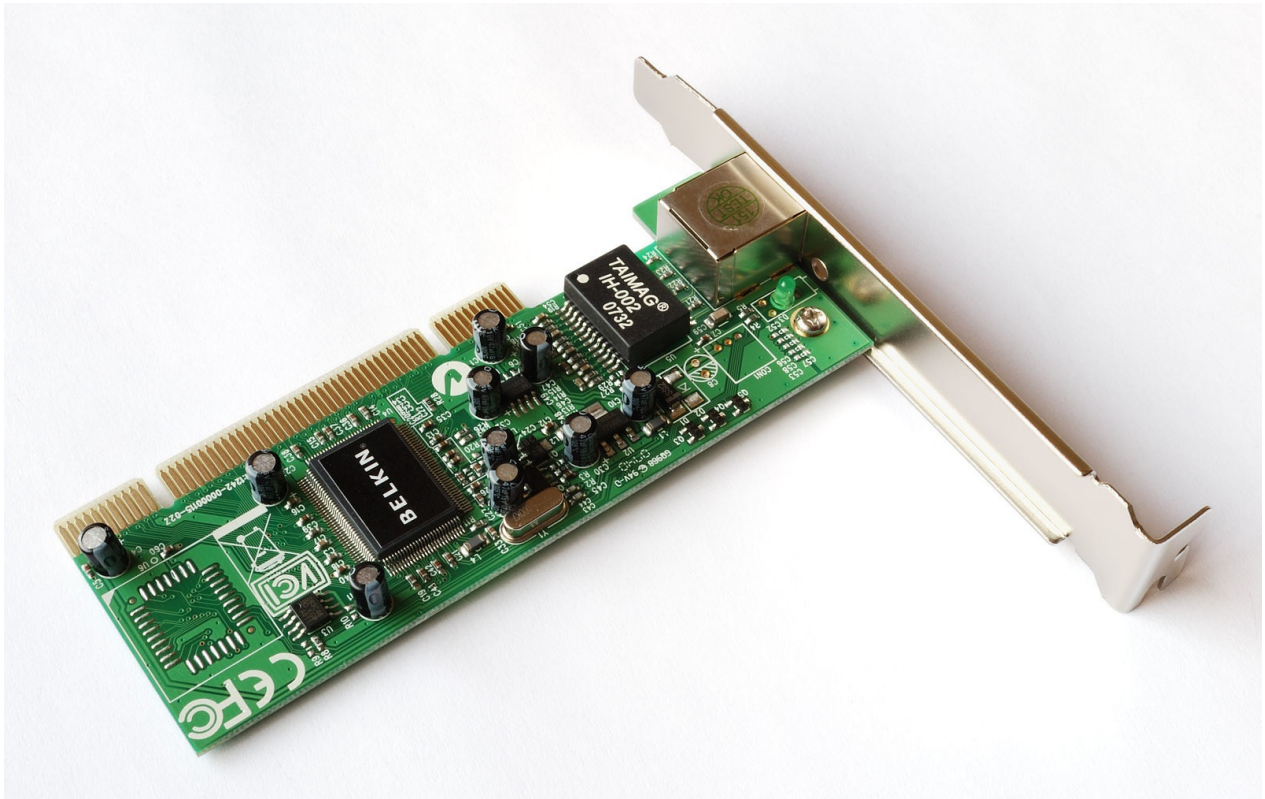
Figure: GB Network PCI Card by Harke is licensed under Public Domain (https://commons.wikimedia.org/wiki/File:GB_Network_PCI_Card.jpg)

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

from   Component Lifecycle

View Example

Accessing Child Component Classes

@ViewChild & @ViewChildren

The `@ViewChild` & `@ViewChildren` decorators allow access to the classes of child components of the current component.

`@ViewChild` selects one class instance of the given child component class when given a type.

For example, we can call `exampleFunction` which is on the child component `Hello` class:

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

@Component({
    selector: 'app',
    directives: [Hello],
    template: `
      <div>
        <hello></hello>
      </div>
      <button (click)="onClick()">Call Child function</button>`
})
export class App {
  @ViewChild(Hello) child: Hello;

  constructor() {}

  onClick() {
    this.child.exampleFunction();
  }
}
```

We can also use `@ViewChildren` to get a list of class instances if there are multiple, which selects a `QueryList` of the elements:

```
import {Component, ViewChildren} from '@angular/core';
import {Hello} from './hello.component';

@Component({
    selector: 'app',
    directives: [Hello],
    template: `
      <div>
        <hello></hello>
        <hello></hello>
        <hello></hello>
      </div>
      <button (click)="onClick()">Call Child function</button>`
})
export class App {
  @ViewChildren(Hello) helloChildren: QueryList<Hello>;

  constructor() {}

  onClick() {
    this.helloChildren.forEach((child) => child.exampleFunction());
  }
}
```

As shown above, when given a class type `@ViewChild` & `@ViewChildren` select child components based on type. However, they can also be passed selector strings.

In the below, only the middle `hello` element is selected & called:

```
import {Component, ViewChildren} from '@angular/core';
import {Hello} from './hello.component';

@Component({
    selector: 'app',
    directives: [Hello],
    template: `
      <div>
        <hello></hello>
        <hello #middle></hello>
        <hello></hello>
      </div>
      <button (click)="onClick()">Call Child function</button>`
})
export class App {
  @ViewChildren('middle') helloChildren: QueryList<Hello>;

  constructor() {}

  onClick() {
    this.helloChildren.forEach((child) => child.exampleFunction());
  }
}
```

View Example

@ContentChild & @ContentChildren

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

Also note that content children will not be set until `ngAfterContentInit` component lifecycle hook.

View Example

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.

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

View Example

ElementRef

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

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

@Component({
  selector: 'app',
  template: `
  <h1>My App</h1>
  <pre style="background: #eee; padding: 1rem; border-radius: 3px; overflow: auto;">
    <code>{{ node }}</code>
  </pre>
`
})
export class App {
  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;
  }

}
```

View Example

Observables

An exciting new feature used with Angular 2 is the `Observable`. This isn't an Angular 2 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 2 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 2 component:

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

@Component({
    selector: 'app',
    template: `
      <b>Angular 2 Component Using Observables!</b>
      <div>Values: {{values.toString()}}</div>
      <div>Errors? {{anyErrors}}</div>
      <div>Finished? {{finished}}</div>
      `
})
export class App {

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

  constructor() {
      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
      );
  }

}
```

[View Example](#)

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.

```typescript
export class App {

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

  constructor() {
      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");
  }

}
```

[View Example](#)

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.

```typescript
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
        );
    }
}
```

[View Example](#)

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.

```typescript
export class App {

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

    constructor() {

        this.data = new Observable(observer => {
            let timeoutId = setTimeout(() => {
                observer.next("You'll 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);
    }

}
```

View Example

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 {Http} from '@angular/http';

@Component({
    selector: 'app',
    template: `
      <b>Angular 2 HTTP requests using RxJs Observables!</b>

      <ul>
        <li *ngFor="let doctor of doctors">{{doctor}}</li>
      </ul>
      `
})

export class App {
  doctors: any;

  constructor(http: Http) {
    http.get('http://jsonplaceholder.typicode.com/posts/1')
        .subscribe((response) => {
            this.doctors = response.json();
        })
    }
  }
}
```

[View Example](#)

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 `Observables` are used in Angular 2 forms. Each field in a form is treated as an `Observable` that we can subscribe to and listen for any changes made to the value of the input field.

```typescript
import {Component} from '@angular/core';
import {Control, ControlGroup, FormBuilder} from '@angular/common';

@Component({
    selector: 'app',
    template: `
      <form [ngFormModel]="coolForm"><input ngControl="email"></form>
      <div><b>You Typed:</b> {{data}}</div>
    `
})

export class App {

    email: Control;
    coolForm: ControlGroup;
    data: string;

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

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

        this.email.valueChanges
            .subscribe(value => this.data = value);
    }
}
```

// View Example View Example // NOTE: using deprecated forms here! Must opt-in to new forms API.

Here we have created a new form by initializing a new `Control` field and grouped it into a `ControlGroup` 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 App {
  private doctors = [];

  constructor(http: Http) {
    http.get('http://jsonplaceholder.typicode.com/users/')
        .flatMap((response) => response.json())
        .filter((person) => person.id > 5)
        .map((person) => "Dr. " + person.name)
        .subscribe((data) => {
          this.doctors.push(data);
        });
  }
}
```

View Example

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.

rxmarbles.com is a helpful resource to understand how the operations work.

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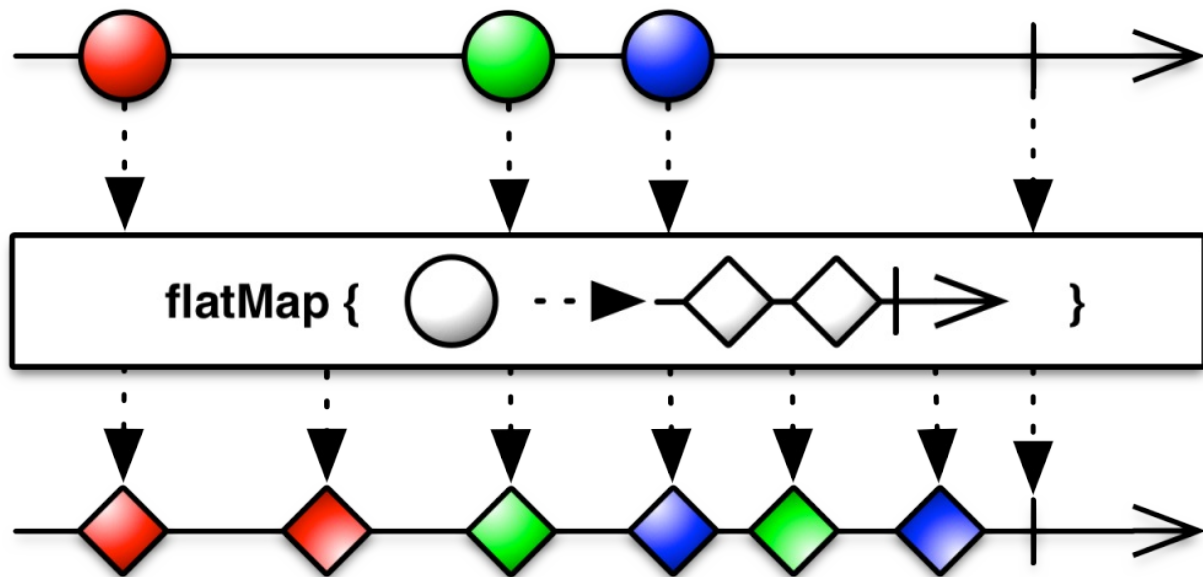
Figure: FlatMap created by ReactiveX licensed under CC-3
(http://reactivex.io/documentation/operators/flatmap.html)

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.ts

```
import {Http} from '@angular/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')
            .map((response) => response.json())
  }
}
```

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.ts

Combining Streams with flatMap

```typescript
import {Component} from '@angular/core';
import {Control, ControlGroup, FormBuilder} from '@angular/common';
import {SearchService} from './services/Search';
import 'rxjs/Rx';


@Component({
    selector: 'app',
    template: `
        <form [ngFormModel]="coolForm"><input ngControl="search" placeholder="Search S
potify artist"></form>

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

export class App {
    searchField: Control;
    coolForm: ControlGroup;

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

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

[View Example](#)

Here we have set up a basic form with a single field, `searchField`, 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.

Cold vs Hot Observables

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

```javascript
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);
```

View Example

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` . The example above is a cold `Observable` , while the example that uses `valueChanges` on our text field input is a hot `Observable` .

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);
```

View Example

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.

Summary

`Observables` offer a flexible set of APIs for composing and transforming asynchronous streams. They provide a multitude of functions to create streams from many other types, and to manipulate and transform them. We've taken a look at how Angular 2 uses `Observables` to create streams from many other types to read user input, perform asynchronous data fetches and set up custom emit/subscribe routines.

- rxjs 4 to 5 migration
- rxjs Observable API
- Which operator do I use?
- rxmarbles
- RxJS Operators by Example

Angular 2 Dependency Injection

Dependency Injection (DI) was a core feature in Angular 1.x, and that has not changed in Angular 2. 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 2 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 2.

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:

```typescript
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:

```typescript
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:

```typescript
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 2 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 2's DI

The last example introduced a hypothetical `Injector` object. Angular 2 simplifies DI even further. With Angular 2, programmers almost never have to get bogged down with injection details.

Angular 2's DI system is very subtle. It's not obvious, but calling Angular 2's bootstrap function initializes Angular 2's injection framework.

For example:

```
import {bootstrap} from '@angular/platform-browser-dynamic';
import {App} from './path/to/your/root/component';

bootstrap(App)
```

Believe it or not, the above example creates the root injector. This example is too primitive though; the injector is not told about anything.

```
import {bootstrap} from '@angular/platform-browser-dynamic';
import {Injectable} from '@angular/core';
import {App} from './path/to/your/root/component';

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

bootstrap(App, [Hamburger]);
```

In the above example the root injector is initialized, and told about the `Hamburger` class. Another way of saying this is that Angular 2 has been *provided* a `Hamburger`.

That seems pretty straightforward, but astute readers will be wondering how Angular 2 knows how to build `Hamburger`. What if `Hamburger` was a string, or a plain function?

Angular 2 *assumes* that it's being given a class.

What about `Bun`, `Patty` and `Toppings`? How is `Hamburger` getting those?

It's not, at least not yet. Angular 2 does not know about them yet. That can be changed easily enough:

```
import {bootstrap} from '@angular/platform-browser-dynamic';
import {Injectable} from '@angular/core';
import {App} from './path/to/your/root/component';

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

@Injectable()
class Patty {}

@Injectable()
class Bun {}

@Injectable()
class Toppings {}

// provide injectables to Angular 2
bootstrap(App, [Hamburger, Patty, Bun, Toppings]);
```

Okay, this is starting to look a little bit more complete. The key takeaway here is
`bootstrap(App, [Hamburger, Patty, Bun, Toppings])` . The second parameter is an array of
`providers` .

Although it's still unclear how `Hamburger` 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()` is a *manual* mechanism for letting Angular 2 know that a *parameter* must be injected. It can be used like so:

```typescript
import {Component, Inject} from '@angular/core';
import {Hamburger} from '../services/hamburger';

@Component({
  selector: 'app',
  template: `Bun Type: {{ bunType }}`
})
export class App {
  bunType: string;
  constructor(@Inject(Hamburger) h) {
    this.bunType = h.bun.type;
  }
}
```

When using TypeScript, `@Inject` is only needed for injecting *primitives*. TypeScript's types let Angular 2 know what to do in most cases. The above example would be simplified in TypeScript to:

```typescript
import {Component} from '@angular/core';
import {Hamburger} from '../services/hamburger';

@Component({
  selector: 'app',
  template: `Bun Type: {{ bunType }}`
})
export class App {
  bunType: string;
  constructor(h: Hamburger) {
    this.bunType = h.bun.type;
  }
}
```

View Example

`@Injectable()` lets Angular 2 know that a *class* can be used with the dependency injector. `@Injectable()` is not *strictly* required if the class has *other* Angular 2 decorators on it or does not have any dependencies.

What is important is that any class that is going to be injected with Angular 2 *is decorated*. However, best practice is to decorate injectables with `@Injectable()`, as it makes more sense to the reader.

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

```
import {Injectable} from '@angular/core';
import {Bun} from './bun';
import {Patty} from './patty';
import {Toppings} from './toppings';

@Injectable()
export class Hamburger {
  constructor(public bun: Bun, public patty: Patty, public toppings: Toppings) {
  }
}
```

In the above example Angular 2's injector determines what to inject into `Hamburger`'s constructor by using type information. This is possible because these particular dependencies are typed, and are *not primitive* types. In some cases Angular 2'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 2 is not limited to injecting classes. The concept of `providers` was also briefly touched upon.

So far `providers` have been passed to Angular 2's `bootstrap` function in an array. `providers` have also all been class identifiers. Angular 2 lets programmers specify providers with a more verbose "recipe". This is done with Angular 2's `provide` function, like so:

```
import {App} from './containers/app'; // hypothetical app component
import {Hamburger} from './services/hamburger';
import {bootstrap} from '@angular/platform-browser-dynamic';
import {provide} from '@angular/core';

bootstrap(App, [provide(Hamburger, { useClass: Hamburger })]);
```

This example is yet another example that `provide`s a class, but it does so with Angular 2's longer format. The second parameter to the `provide` function lets programmers "use" things other than classes for dependency injection.

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

```
import {App} from './containers/app'; // hypothetical app component
import {Hamburger} from './services/hamburger';
import {DoubleHamburger} from './services/double-hamburger';
import {bootstrap} from '@angular/platform-browser-dynamic';
import {provide} from '@angular/core';

bootstrap(App, [provide(Hamburger, { useClass: DoubleHamburger })]);
```

The best part of this implementation swap is that the injection system knows how to build `DoubleHamburgers`, 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 2 can use. For example:

```
import {App} from './containers/app'; // hypothetical app component
import {bootstrap} from '@angular/platform-browser-dynamic';
import {provide} from '@angular/core';

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

bootstrap(App, [provide('Random', randomDefinition)]);
```

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

```
import {Component, Inject, provide} from '@angular/core';
import {Hamburger} from '../services/hamburger';

@Component({
  selector: 'app',
  template: `Random: {{ value }}`
})
export class App {
  value: number;
  constructor(@Inject('Random') r) {
    this.value = r;
  }
}
```

View Example

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

The above example uses Angular 2's `useFactory` recipe. When Angular 2 is told to `provide` things using `useFactory`, Angular 2 expects the provided value to be a function. Sometimes functions and classes are even more than what's needed. Angular 2 has a "recipe" called `useValue` for these cases that works almost exactly the same:

```
import {bootstrap} from '@angular/platform-browser-dynamic';
import {provide} from '@angular/core';
import {App} from './containers/app';

const randomDefinition = { useValue: Math.random() };

bootstrap(App, [provide('Random', randomDefinition)]);
```

View Example

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

The Injector Tree

Angular 2 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 2 overcomes this limitation by using a tree of injectors.

In Angular 2 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 2's component tree.

Consider the following tree, which models a chat application consisting of two open chat windows, and a login/logout widget.
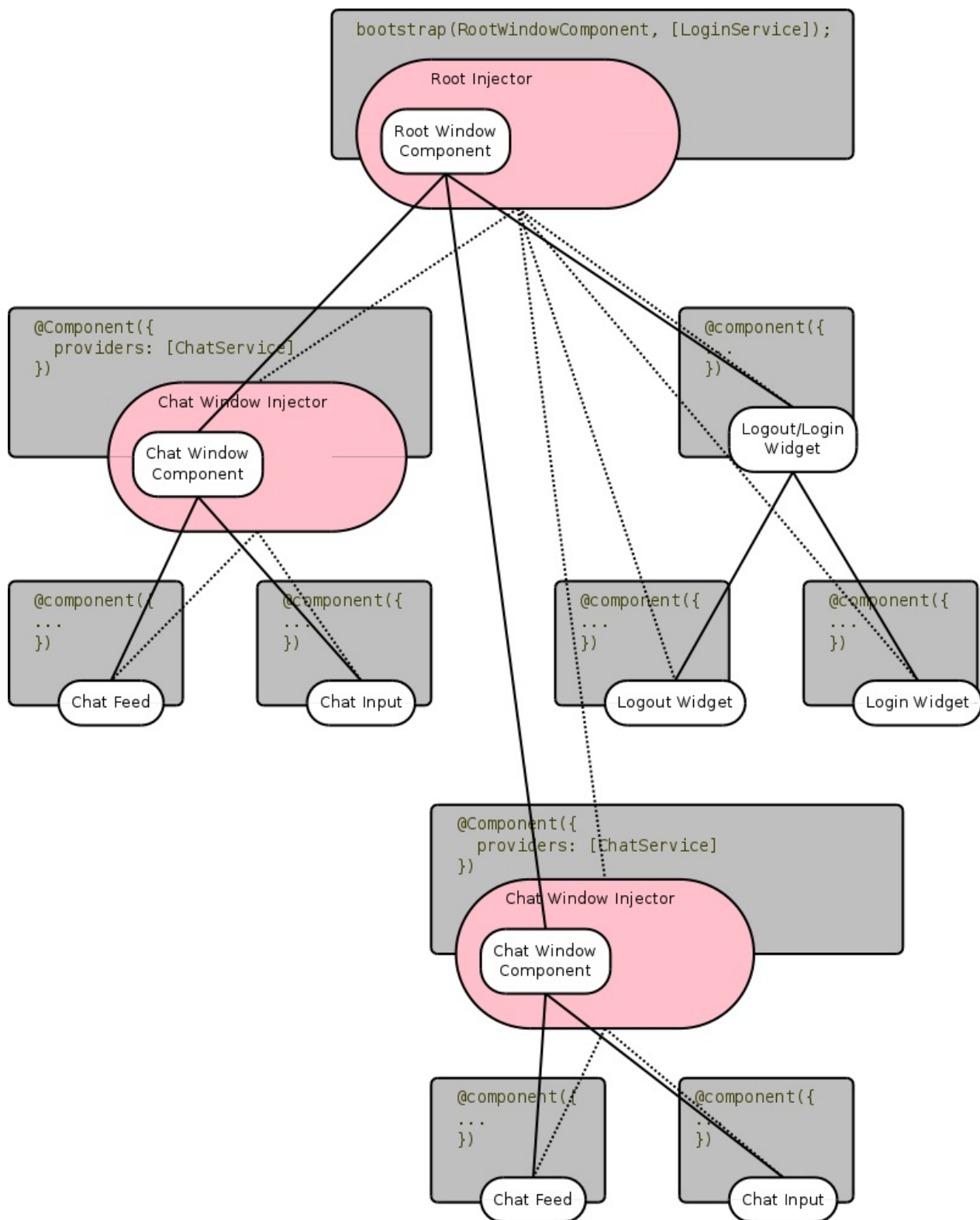
Figure: Image of a Component Tree, and a DI Tree

In the image above, there is one root injector, which is also the root component. This is also the application bootstrap area. There's a `LoginService` registered with the root injector.

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/boot.ts

```
import {bootstrap} from '@angular/platform-browser-dynamic';
import {provide} from '@angular/core';
import {App} from './containers/app';
import {Unique} from './services/unique';


bootstrap(App, [Unique]);
```

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: string;
  constructor() {
    this.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

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

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

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

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

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

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

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

```
import {Component, Inject, provide} from '@angular/core';
import {Hamburger} from '../services/hamburger';
import {ChildInheritor} from '../components/child-inheritor';
import {ChildOwnInjector} from '../components/child-own-injector';
import {Unique} from '../services/unique';

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

[View Example](#)

Change Detection

Figure: Change Detector by Vovka is licensed under Public Domain
(https://pixabay.com/en/coins-handful-russia-ruble-kopek-650779/)

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

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

Because by default Angular 1 implemented two way data binding, the flow of changes was pretty much chaotic, models were able to change directives, directives were able to change models, directives were able to change other directives and models were able to change other models.

In Angular 2, **the flow of information is unidirectional**, even when using `ngModel` to implement two way data binding, which is only syntactic sugar on top of the unidirectional flow. In this new version of the framework, our code is responsible for updating the models. Angular is only responsible for reflecting those changes in the components and the DOM by means of the selected change detection strategy.

Change Detection Strategies in Angular 1 vs Angular 2

Another difference between both versions of the framework is the way the nodes of an application (directives or components) are checked to see if the DOM needs to be updated.

Because of the nature of two-way data binding, in Angular 1 there was no guarantee that a parent node would always be checked before a child node. It was possible that a child node could change a parent node or a sibling or any other node in the tree, and that in turn would trigger new updates down the chain. This made it difficult for the change detection mechanism to traverse all the nodes without falling in a circular loop with the infamous message:

```
10 $digest() iterations reached. Aborting!
```

In Angular 2, changes are guaranteed to propagate unidirectionally. The change detector will **traverse each node only once**, always starting from the root. That means that a parent component is always checked before its children components.

Tree traversing in Angular 1 vs Angular 2
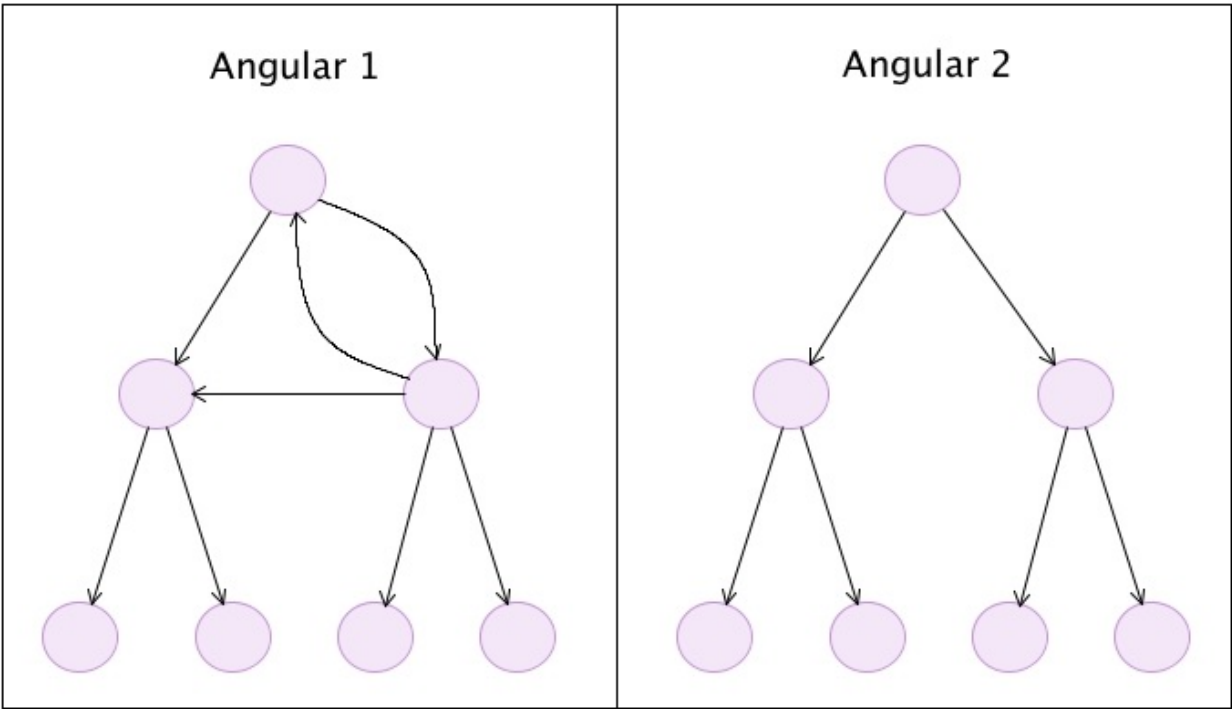
Figure: File Structure

How Change Detection Works

Let's see how change detection works with a simple example.

We are going to create a simple `MovieApp` to show information about one movie. This app is going to consist of only two components: the `MovieComponent` that shows information about the movie and the `MainComponent` which holds a reference to the movie with buttons to perform some actions.

As always, the first step is to create our `index.html` file using the HTML element defined in the root component of our app `MainComponent` .

index.html

```html
<html>
  <!-- ... -->
  <body>
    <main>Loading...</main>
  </body>
</html>
```

And the boot file to load the application.

app/boot.ts

```typescript
import {bootstrap} from 'angular2/platform/browser';
import {MainComponent} from './main.component';

bootstrap(MainComponent);
```

Our `MainComponent` will have three properties: the `slogan` of the app, the `title` of the movie and the lead `actor` . The last two properties will be passed to the `MovieComponent` element referenced in the template.

app/main.component.ts

```typescript
import {Component} from '@angular/core';
import {MovieComponent} from './movie.component';
import {Actor} from './actor.model';

@Component({
  selector: 'main',
  directives: [MovieComponent],
  template: `
    <h1>MovieApp</h1>
    <p>{{ slogan }}</p>
    <button type="button" (click)="changeActorProperties()">Change Actor Properties</button>
    <button type="button" (click)="changeActorObject()">Change Actor Object</button>
    <movie [title]="title" [actor]="actor"></movie>`
})
export class MainComponent {
  slogan: string = 'Just movie information';
  title: string = 'Terminator 1';
  actor: Actor = new Actor('Arnold', 'Schwarzenegger');

  changeActorProperties() {
    this.actor.firstName = 'Nicholas';
    this.actor.lastName = 'Cage';
  }

  changeActorObject() {
    this.actor = new Actor('Bruce', 'Willis');
  }
}
```

In the above code snippet, we can see that our component defines two buttons that trigger different methods. The `changeActorProperties` will update the lead actor of the movie by directly changing the properties of the `actor` object. In contrast, the method `changeActorObject` will change the information of the actor by creating a completely new instance of the `Actor` class.

The `Actor` model is pretty straightforward, it is just a class that defines the `firstName` and the `lastName` of an actor.

app/actor.model.ts

```typescript
export class Actor {
  constructor(
    public firstName: string,
    public lastName: string) {}
}
```

Finally, the `MovieComponent` shows the information provided by the `MainComponent` in its template.

app/movie.component.ts

```typescript
import {Component, Input} from '@angular/core';
import {Actor} from './actor.model';

@Component({
  selector: 'movie',
  styles: ['div {border: 1px solid black}'],
  template: `
    <div>
      <h3>{{ title }}</h3>
      <p>
        <label>Actor:</label>
        <span>{{actor.firstName}} {{actor.lastName}}</span>
      </p>
    </div>`
})
export class MovieComponent {
  @Input() title: string;
  @Input() actor: Actor;
}
```

The final result of the app is shown in the screenshot below:

MovieApp

Just movie information

| Change Actor Properties | Change Actor Object |

Terminator 1

Actor: Arnold Schwarzenegger

Figure: File Structure

Change Detector Classes

At runtime, Angular 2 will create special classes that are called *change detectors*, one for every component that we have defined. In this case, Angular will create two classes: `MainComponent_ChangeDetector` and `MovieComponent_ChangeDetector`.

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

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

In our example, because we only have one instance of the `MainComponent` and the `MovieComponent`, we will have only one instance of the `MainComponent_ChangeDetector` and the `MovieComponent_ChangeDetector`.

The code snippet below is a conceptual model of how the `MainComponent_ChangeDetector` class might look.

```
class MainComponent_ChangeDetector {

  constructor(
    public previousSlogan: string,
    public previousTitle: string,
    public previousActor: Actor,
    public movieComponent: MovieComponent
  ) {}

  detectChanges(slogan: string, title: string, actor: Actor) {
    if (slogan !== this.previousSlogan) {
      this.previousSlogan = slogan;
      this.movieComponent.slogan = slogan;
    }
    if (title !== this.previousTitle) {
      this.previousTitle = title;
      this.movieComponent.title = title;
    }
    if (actor !== this.previousActor) {
      this.previousActor = actor;
      this.movieComponent.actor = actor;
    }
  }
}
```

Because in the template of our `MainComponent` we reference three variables (`slogan` , `title` and `actor`), our change detector will have three properties to store the "old" values of these three properties, plus a reference to the `MainComponent` instance that it's supposed to "watch". When the change detection process wants to know if our `MainComponent` instance has changed, it will run the method `detectChanges` passing the current model values to compare with the old ones. If a change was detected, the component gets updated.

> Disclaimer: This is just a conceptual overview of how change detector classes work; the actual implementation may be different.

Change Detection Strategy: Default

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

app/movie.component.ts

```
// ...
import {ChangeDetectionStrategy} from '@angular/core';

@Component({
  // ...
  changeDetection: ChangeDetectionStrategy.Default
})
export class MovieComponent {
  // ...
}
```

View Example

Let's see what happens when a user clicks the button "Change Actor Properties" when using the `Default` strategy.

As noted previously, changes are triggered by events and the propagation of changes is done in two phases: the application phase and the change detection phase.

Phase 1 (Application):

In the first phase, the application (our code) is responsible for updating the models in response to some event. In this scenario, the properties `actor.firstName` and `actor.lastName` are updated.

Phase 2 (Change Detection):

Now that our models are updated, Angular must update the templates using change detection.

Change detection always starts at the root component, in this case the `MainComponent` , and checks if any of the model properties bound to its template have changed, comparing the old value of each property (before the event was triggered) to the new one (after the models were updated). The `MainComponent` template has a reference to three properties, `slogan` , `title` and `actor` , so the comparison made by its corresponding change detector will look like:

- Is `slogan !== previousSlogan` ? No, it's the same.
- Is `title !== previousTitle` ? No, it's the same.
- Is `actor !== previousActor` ? No, it's the same.

Notice that even if we change the properties of the `actor` object, we are always working with the same instance. Because we are doing a shallow comparison, the result of asking if `actor !== previousActor` will always be `false` even when its internal property values have indeed changed. Even though the change detector was unable to find any change, the **default strategy** for the change detection is **to traverse all the components of the tree** even if they do not seem to have been modified.

Next, change detection moves down in the component hierarchy and check the properties bound to the `MovieComponent` 's template doing a similar comparison:

- Is `title !== previousTitle` ? No, it's the same.
- Is `actorFirstName !== previousActorFirstName` ? **Yes**, it has changed.
- Is `actorLastName !== previousActorLastName` ? **Yes**, it has changed.

Finally, Angular has detected that some of the properties bound to the template have changed so it will update the DOM to get the view in sync with the model.

Performance Impact

Traversing all the tree components to check for changes could be costly. Imagine that instead of just having one reference to `<movie>` inside our `MainComponent` 's template, we have multiple references?

```
<movie *ngFor="let movie of movies" [title]="movie.title" [actor]="movie.actor"></movie
>`
```

If our movie list grows too big, the performance of our system will start degrading. We can narrow the problem to one particular comparison:

- Is `actor !== previousActor` ?

As we have learned, this result is not very useful because we could have changed the properties of the object without changing the instance, and the result of the comparison will always be `false` . Because of this, change detection is going to have to check every child component to see if any of the properties of that object (`firstName` or `lastName`) have changed.

What if we can find a way to indicate to the change detection that our `MovieComponent` depends only on its inputs and that these inputs are immutable? In short, we are trying to guarantee that when we change any of the properties of the `actor` object, we end up with a different `Actor` instance so the comparison `actor !== previousActor` will always return `true` . On the other hand, if we did not change any property, we are not going to create a new instance, so the same comparison is going to return `false` .

If the above condition can be guaranteed (create a new object every time any of its properties changes, otherwise we keep the same object) and when checking the inputs of the `MovieComponent` has this result:

- Is `title !== previousTitle` ? No, it's the same.
- Is `actor !== previousActor` ? No, it's the same.

then we can skip the internal check of the component's template because we are now certain that nothing has changed internally and there's no need to update the DOM. This will improve the performance of the change detection system because fewer comparisons have to be made to propagate changes through the app.

Change Detection Strategy: OnPush

To inform Angular that we are going to comply with the conditions mentioned before to improve performance, we will use the `OnPush` change detection strategy on the `MovieComponent` .

app/movie.component.ts

```
// ...

@Component({
  // ...
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class MovieComponent {
  // ...
}
```

[View Example](#)

This will inform Angular that our component only depends on its inputs and that any object that is passed to it should be considered immutable. This time when we click the "Change Actor Properties" button nothing changes in the view.

Let's follow the logic behind it again. When the user clicks the button, the method `changeActorProperties` is called and the properties of the `actor` object get updated.

When the change detection analyzes the properties bound to the `MainComponent` 's template, it will see the same picture as before:

- Is `slogan !== previousSlogan` No, it's the same.
- Is `title !== previousTitle` ? No, it's the same.
- Is `actor !== previousActor` ? No, it's the same.

But this time, we explicitly told Angular that our component only depends on its inputs and all of them are immutable. Angular then assumes that the `MovieComponent` hasn't changed and will skip the check for that component. Because we didn't force the `actor` object to be immutable, we end up with our model out of sync with the view.

Let's rerun the app but this time we will click the "Change Actor Object" button. This time, we are creating a new instance of the `Actor` class and assigning it to the `this.actor` object. When change detection analyzes the properties bound to the `MainComponent` 's template it will find:

- Is `slogan !== previousSlogan` No, it's the same.
- Is `title !== previousTitle` ? No, it's the same.
- Is `actor !== previousActor` ? **Yes**, it has changed.

Because change detection now knows that the `actor` object changed (it's a new instance) it will go ahead and continue checking the template for `MovieComponent` to update its view. At the end, our templates and models are in sync.

Enforcing Immutability

We cheated a little in the previous example. We told Angular that all of our inputs, including the `actor` object, were immutable objects, but we went ahead and updated its properties, violating the immutability principle. As a result we ended up with a sync problem between our models and our views. One way to enforce immutability is using the library Immutable.js.

Because in JavaScript primitive types like `string` and `number` are immutable by definition, we should only take care of the objects we are using. In this case, the `actor` object.

Here's an example comparing a mutable type like an `array` to an immutable type like a `string`:

```
var b = ['C', 'a', 'r'];
b[0] = 'B';
console.log(b) // ['B', 'a', 'r'] => The first letter changed, arrays are mutable

var a = 'Car';
a[0] = 'B';
console.log(a); // 'Car' => The first letter didn't change, strings are immutable
```

First we need to install the `immutable.js` library using the command:

```
npm install --save immutable
```

Then in our `MainComponent` we import the library and use it to create an actor object as an immutable.

app/main.component.ts

```
import {Component} from '@angular/core';
import {MovieComponent} from './movie.component';
import * as Immutable from 'immutable';

@Component({
  selector: 'main',
  directives: [MovieComponent],
  template: `
    <h1>MovieApp</h1>
    <p>{{ slogan }}</p>
    <button type="button" (click)="changeActor()">Change Actor</button>
    <movie [title]="title" [actor]="actor"></movie>`
})
export class MainComponent {
  slogan: string = 'Just movie information';
  title: string = 'Terminator 1';
  actor: Immutable.Map<string, string> = Immutable.Map({firstName: 'Arnold', lastName:
'Schwarzenegger'});

  changeActor() {
    this.actor = this.actor.merge({firstName: 'Nicholas', lastName: 'Cage'});
  }
}
```

Now, instead of creating an instance of an `Actor` class, we are defining an immutable object using `Immutable.Map`. Because `this.actor` is now an immutable object, we cannot change its internal properties (`firstName` and `lastName`) directly. What we can do however is create another object based on `actor` that has different values for both fields - that is exactly what the `merge` method does.

Because we are always getting a new object when we try to change the `actor`, there's no point in having two different methods in our component. We removed the methods `changeActorProperties` and `changeActorObject` and created a new one called `changeActor`.

Additional changes have to be made to the `MovieComponent` as well. First we need to declare the `actor` object as an immutable type, and in the template, instead of trying to access the object properties directly using a syntax like `actor.firstName`, we need to use the `get` method of the immutable.

app/movie.component.ts

```typescript
import {Component, Input} from '@angular/core';
import {ChangeDetectionStrategy} from '@angular/core';
import * as Immutable from 'immutable';

@Component({
  selector: 'movie',
  styles: ['div {border: 1px solid black}'],
  template: `
    <div>
      <h3>{{ title }}</h3>
      <p>
        <label>Actor:</label>
        <span>{{actor.get('firstName')}} {{actor.get('lastName')}}</span>
      </p>
    </div>`,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class MovieComponent {
  @Input() title: string;
  @Input() actor: Immutable.Map<string, string>;
}
```

[View Example](#)

Using this pattern we are taking full advantage of the "OnPush" change detection strategy and thus reducing the amount of work done by Angular to propagate changes and to get models and views in sync. This improves the performance of the application.

Additional Resources

To learn more about change detection, visit the following links (in order of relevance):

- NgConf 2014: Change Detection (Video)
- Angular API Docs: ChangeDetectionStrategy
- Victor Savkin Blog: Change Detection in Angular 2
- Victor Savkin Blog: Two Phases of Angular 2 Applications
- Victor Savkin Blog: Angular, Immutability and Encapsulation

Advanced Angular

Angular 2 gives us access to most of the core entities it uses in its architecture. Now that we understand the different parts involved in an Angular 2 application, let's dig deeper into some of these entities and take advantage of what we know.

Angular Directives

Angular 2 built-in directives cover a broad range of functionality, but sometimes creating our own directives will result in more elegant solutions.

Creating an Attribute Directive

Let's start with a simple button that moves a user to a different page.

```
@Component({
  selector: 'visit-rangle',
  template: `
    <button type="button" (click)="visitRangle()">Visit Rangle</button>
  `
})
export class VisitRangleComponent {
  visitRangle() {
    location.href = 'https://rangle.io';
  }
}
```

[View Example](#)

We're polite, so rather than just sending the user to a new page, we're going to ask if they're ok with that first by creating an attribute directive and attaching that to the button.

```
@Directive({
  selector: `[confirm]`
})
export class ConfirmDirective {

  @HostListener('click', ['$event'])
  confirmFirst(event: Event) {
    return window.confirm('Are you sure you want to do this?');
  }

}
```

[View Example](#)

Directives are created by using the `@Directive` decorator on a class and specifying a selector. For directives, the selector name must be camelCase and wrapped in square brackets to specify that it is an attribute binding. We're using the `@HostListener` decorator to listen in on events on the component or element it's attached to. In this case we're watching the `click` event and passing in the event details which are given by the special `$event` keyword. Next, we want to attach this directive to the button we created earlier.

```
directives: [ConfirmDirective],
template: `
  <button type="button" (click)="visitRangle()" confirm>Visit Rangle</button>
`
```

[View Example](#)

Notice, however, that the button doesn't work quite as expected. That's because while we're listening to the click event and showing a confirm dialog, the component's click handler runs before the directive's click handler and there's no communication between the two. To do this we'll need to rewrite our directive to work with the component's click handler.

```
@Directive({
  selector: `[confirm]`
})
export class ConfirmDirective {
  @Input('confirm') onConfirmed: Function = () => {};

  @HostListener('click', ['$event'])
  confirmFirst() {
    const confirmed = window.confirm('Are you sure you want to do this?');

    if(confirmed) {
      this.onConfirmed();
    }
  }
}
```

[View Example](#)

Here, we want to specify what action needs to happen after a confirm dialog's been sent out and to do this we create an input binding just like we would on a component. We'll use our directive name for this binding and our component code changes like this:

```
<button type="button" [confirm]="visitRangle">Visit Rangle</button>
```

[View Example](#)

Now our button works just as we expected. We might want to be able to customize the message of the confirm dialog however. To do this we'll use another binding.

```
@Directive({
  selector: `[confirm]`
})
export class ConfirmDirective {
  @Input('confirm') onConfirmed: Function = () => {};
  @Input() confirmMessage: string = 'Are you sure you want to do this?';

  @HostListener('click', ['$event'])
  confirmFirst() {
    const confirmed = window.confirm(this.confirmMessage);

    if(confirmed) {
      this.onConfirmed();
    }
  }
}
```

[View Example](#)

Our directive gets a new input property that represents the confirm dialog message, which we pass in to `window.confirm` call. To take advantage of this new input property, we add another binding to our button.

```
<button
  type="button"
  [confirm]="visitRangle"
  confirmMessage="Click ok to visit Rangle.io!">
  Visit Rangle
</button>
```

[View Example](#)

Now we have a button with a customizable confirm message before it moves you to a new url.

Creating a Structural Directive

We'll create a `delay` structural directive that delays instantiation of a component or element. This can potentially be used for cosmetic effect or for manually handling timing of when components are loaded, either for performance or UX.

```
@Directive({
  selector: '[delay]'
})
export class DelayDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainerRef: ViewContainerRef
  ) { }

  @Input('delay')
  set delayTime(time: number): void { }
}
```

[View Example](#)

We use the same `@Directive` class decorator as attribute directives and define a selector in the same way. One big difference here is that due to the nature of structural directives being bound to a template, we have access to `TemplateRef`, an object representing the `template` tag the directive is attached to. We also add an input property in a similar way, but this time with a `set` handler so we can execute some code when Angular 2 performs the binding. We bind `delay` in exactly the same way as the Angular 2 built-in structural directives.

```
@Component({
  selector: 'app',
  directives: [CardComponent,DelayDirective],
  template: `
    <div *ngFor="let item of [1,2,3,4,5,6]">
      <card *delay="500 * item">
        {{item}}
      </card>
    </div>
  `
})
export class App {
}
```

[View Example](#)

Notice that no content is being rendered however. This is due to Angular 2 simulating the html `template` tag and not rendering any child elements by default. To be able to get this content to render, we'll have to attach the template given by `TemplateRef` as an *embedded view* to a *view container*.

View Containers and Embedded Views

View Containers are containers where one or more *Views* can be attached. Views represent some sort of layout to be rendered and the context under which to render it. View containers are anchored to components and are responsible for generating its output so this means that changing which views are attached to the view container affect the final rendered output of the component.

Two types of views can be attached to a view container: *Host Views* which are linked to a *Component*, and *Embedded Views* which are linked to a *template*. Since structural directives interact with templates, we are interested in using *Embedded Views* in this case.

```
import {Directive, Input, TemplateRef, ViewContainerRef} from '@angular/core';

@Directive({
  selector: '[delay]'
})
export class DelayDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainerRef: ViewContainerRef
  ) { }

  @Input('delay')
  set delayTime(time: number): void {
    setTimeout(
      () => {
        this.viewContainerRef.createEmbeddedView(this.templateRef);
      },
      time);
  }
}
```

[View Example](#)

Directives get access to the view container by injecting a `ViewContainerRef` . Embedded views are created and attached to a view container by calling the `ViewContainerRef` 's `createEmbeddedView` method and passing in the template. We want to use the template our directive is attached to so we pass in the injected `TemplateRef` .

Providing Context Variables to Directives

Suppose we want to record some metadata on how our directive affected components and make this data available to them. For example, in our `delay` directive, we're making a `setTimeout` call, which in JavaScript's single-threaded asynchronous model means that it may not run after the exact time we provided. We'll capture the exact time it loads and make that variable available in the template.

```typescript
export class DelayContext {
  constructor(private loadTime: number) { }
}

@Directive({
  selector: '[delay]'
})
export class DelayDirective {
  constructor(
    private templateRef: TemplateRef<DelayContext>,
    private viewContainerRef: ViewContainerRef
  ) { }

  @Input('delay')
  set delayTime(time: number): void {
    setTimeout(
      () => {
        this.viewContainerRef.createEmbeddedView(
          this.templateRef,
          new DelayContext(performance.now())
        );
      },
      time);
  }
}
```

[View Example](#)

We've made a few changes to our `delay` directive. We've created a new `DelayContext` class that contains the context that we want to provide to our directive. In this case, we want to capture the actual time the `createEmbeddedView` call occurs and make that available as `loadTime` in our directive. We've also provided our new class as the generic argument to the `TemplateRef` function. This enables static analysis and lets us make sure our calls to `createEmbeddedView` pass in a variable of type `DelayContext`. In our `createEmbeddedView` call we pass in our variable which has captured the time of the method call.

In the component using `delay`, we access the `loadTime` context variable in the same way we access variables in `ngFor`.

```
@Component({
  selector: 'app',
  directives: [CardComponent,DelayDirective],
  template: `
    <div *ngFor="let item of [1,2,3,4,5,6]">
      <card *delay="500 * item; let loaded = loadTime">
        <div class="main">{{item}}</div>
        <div class="sub">{{loaded | number:'1.4-4'}}</div>
      </card>
    </div>
  `
})
```

View Example

Immutable.js

165

Immutable.js is a library that provides immutable generic collections.

Figure: Ayers Rock Uluru by Stefanoka is licensed under CC BY-SA 3.0
(https://commons.wikimedia.org/wiki/File:Ayers_Rock_Uluru.jpg)

What is Immutability?

Immutability is a design pattern where something can't be modified after being instantiated. If we want to change the value of that thing, we must recreate it with the new value instead. Some JavaScript types are immutable and some are mutable, meaning their value can change without having to recreate it. Let's explain this difference with some examples:

```
let movie = {
  name: 'Star Wars',
  episode: 7
};

let myEp = movie.episode;

movie.episode = 8;

console.log(myEp); // outputs 7
```

As you can see in this case, although we changed the value of `movie.episode` , the value of `myEp` didn't change. That's because `movie.episode` 's type, `number` , is immutable.

```
let movie1 = {
  name: 'Star Wars',
  episode: 7
};

let movie2 = movie1;

movie2.episode = 8;

console.log(movie1.episode); // outputs 8
```

In this case however, changing the value of episode on one object also changed the value of the other. That's because `movie1` and `movie2` are of the **Object** type, and Objects are mutable.

Of the JavaScript built-in types, the following are immutable:

- Boolean
- Number
- String
- Symbol
- Null

- Undefined

And the following are mutable:

- Object
- Array
- Function

String's an unusual case, since it can be iterated over using for...of and provides numeric indexers just like an array, but doing something like:

```
let message = 'Hello world';
message[5] = '-';
console.log(message); // writes Hello world
```

> This will throw an error in strict mode and fail silently in non-strict mode.

The Case for Immutability

One of the more difficult things to manage when structuring an application is managing its state. This is especially true when your application can execute code asynchronously. Let's say you execute some piece of code, but something causes it to wait (such as an HTTP request or user input). After it's completed, you notice the state it's expecting changed because some other piece of code executed asynchronously and changed its value.

Dealing with that kind of behavior on a small scale might be manageable, but this can show up all over an application and can be a real headache as the application gets bigger with more interactions and more complex logic.

Immutability attempts to solve this by making sure that any object referenced in one part of the code can't be changed by another part of the code unless they have the ability to rebind it directly.

JavaScript Solutions

Some new features have been added in ES6 that allow for easier implementation of immutable data patterns.

Object.assign

`Object.assign` lets us merge one object's properties into another, replacing values of properties with matching names. We can use this to copy an object's values without altering the existing one.

```js
let movie1 = {
  name: 'Star Wars',
  episode: 7
};

let movie2 = Object.assign({}, movie1);

movie2.episode = 8;

console.log(movie1.episode); // writes 7
console.log(movie2.episode); // writes 8
```

As you can see, although we have some way of copying an object, we haven't made it immutable, since we were able to set the episode's property to 8. Also, how do we modify the episode property in this case? We do that through the assign call:

```js
let movie1 = {
  name: 'Star Wars',
  episode: 7
};

let movie2 = Object.assign({}, movie1, { episode: 8 });

console.log(movie1.episode); // writes 7
console.log(movie2.episode); // writes 8
```

Object.freeze

`Object.freeze` allows us to disable object mutation.

```
let movie1 = {
  name: 'Star Wars',
  episode: 7
};

let movie2 = Object.freeze(Object.assign({}, movie1));

movie2.episode = 8; // fails silently in non-strict mode,
                    // throws error in strict mode

console.log(movie1.episode); // writes 7
console.log(movie2.episode); // writes 7
```

One problem with this pattern, however, is how much more verbose our code is and how difficult it is to read and understand what's actually going on with our data with all of the boilerplate calls to `Object.freeze` and `Object.assign`. We need some more sensible interface to create and interact with immutable data, and that's where Immutable.js fits in.

> `Object.freeze` is also very slow and should not be used with large arrays.

Immutable.js Basics

To solve our mutability problem, Immutable.js must provide immutable versions of the two core mutable types, **Object** and **Array**.

Immutable.Map

`Map` is the immutable version of JavaScript's object structure. Due to JavaScript objects having the concise object literal syntax, it's often used as a key-value store with `key` being type `string`. This pattern closely follows the map data structure. Let's revisit the previous example, but use `Immutable.Map` instead.

```
import * as Immutable from 'immutable';

let movie1 = Immutable.Map<string, any>({
  name: 'Star Wars',
  episode: 7
});

let movie2 = movie1;

movie2 = movie2.set('episode', 8);

console.log(movie1.get('episode')); // writes 7
console.log(movie2.get('episode')); // writes 8
```

Instead of binding the object literal directly to `movie1`, we pass it as an argument to `Immutable.Map`. This changes how we interact with movie1's properties.

To *get* the value of a property, we call the `get` method, passing the property name we want, like how we'd use an object's string indexer.

To *set* the value of a property, we call the `set` method, passing the property name and the new value. Note that it won't mutate the existing Map object - it returns a new object with the updated property, so we must rebind the `movie2` variable to the new object.

Map.merge

Sometimes we want to update multiple properties. We can do this using the `merge` method.

```
let baseButton = Immutable.Map<string, any>({
  text: 'Click me!',
  state: 'inactive',
  width: 200,
  height: 30
});

let submitButton = baseButton.merge({
  text: 'Submit',
  state: 'active'
});

console.log(submitButton);
// writes { text: 'Submit', state: 'active', width: 200, height: 30 }
```

Nested Objects

`Immutable.Map` wraps objects shallowly, meaning if you have an object with properties bound to mutable types then those properties can be mutated.

```javascript
let movie = Immutable.Map({
    name: 'Star Wars',
    episode: 7,
    actors: [
        { name: 'Daisy Ridley', character: 'Rey'},
        { name: 'Harrison Ford', character: 'Han Solo' }
    ],
    mpaa: {
        rating: 'PG-13',
        reason: 'sci-fi action violence'
    }
});

movie.get('actors').pop();
movie.get('mpaa').rating = 'PG';

console.log(movie.toObject());

/* writes
{ name: 'Star Wars',
  episode: 7,
  actors: [ { name: 'Daisy Ridley', character: 'Rey' } ],
  mpaa: { rating: 'PG', reason: 'sci-fi action violence' } }
  */
```

To avoid this issue, use `Immutable.fromJS` instead.

```
let movie = Immutable.fromJS({
    name: 'Star Wars',
    episode: 7,
    actors: [
        { name: 'Daisy Ridley', character: 'Rey'},
        { name: 'Harrison Ford', character: 'Han Solo' }
    ],
    mpaa: {
        rating: 'PG-13',
        reason: 'sci-fi action violence'
    }
});

movie.get('actors').pop();
movie.get('mpaa').rating = 'PG';

console.log(movie.toObject());

/* writes
{ name: 'Star Wars',
  episode: 7,
  actors: List [ Map { "name": "Daisy Ridley", "character": "Rey" }, Map { "name": "Ha
rrison Ford", "character": "Han Solo" } ],
  mpaa: Map { "rating": "PG-13", "reason": "sci-fi action violence" } }
*/
```

So let's say you want to modify `movie.mpaa.rating` . You might think of doing something like this: `movie = movie.get('mpaa').set('rating', 'PG')` . However, `set` will always return the calling Map instance, which in this case returns the Map bound to the `mpaa` key rather than the movie you wanted. We must use the `setIn` method to update nested properties.

```
let movie = Immutable.fromJS({
    name: 'Star Wars',
    episode: 7,
    actors: [
        { name: 'Daisy Ridley', character: 'Rey'},
        { name: 'Harrison Ford', character: 'Han Solo' }
    ],
    mpaa: {
        rating: 'PG-13',
        reason: 'sci-fi action violence'
    }
});

movie = movie
  .update('actors', actors => actors.pop())
  .setIn(['mpaa', 'rating'], 'PG');

console.log(movie.toObject());

/* writes
{ name: 'Star Wars',
  episode: 7,
  actors: List [ Map { "name": "Daisy Ridley", "character": "Rey" } ],
  mpaa: Map { "rating": "PG", "reason": "sci-fi action violence" } }
*/
```

We also added a call to `Map.update` which, unlike `set`, accepts a function as the second argument instead of a value. This function accepts the existing value at that key and must return the new value of that key.

Deleting Keys

Keys can be deleted from maps using the `Map.delete` and `Map.deleteIn` methods.

```javascript
let movie = Immutable.fromJS({
    name: 'Star Wars',
    episode: 7,
    actors: [
        { name: 'Daisy Ridley', character: 'Rey'},
        { name: 'Harrison Ford', character: 'Han Solo' }
    ],
    mpaa: {
        rating: 'PG-13',
        reason: 'sci-fi action violence'
    }
});

movie = movie.delete('mpaa');

console.log(movie.toObject());

/* writes
{ name: 'Star Wars',
  episode: 7,
  actors: List [ Map { "name": "Daisy Ridley", "character": "Rey" }, Map { "name": "Ha
rrison Ford", "character": "Han Solo" } ] }
*/
```

Maps are Iterable

Maps in Immutable.js are *iterable*, meaning that you can `map` , `filter` , `reduce` , etc. each key-value pair in the map.

```
let features = Immutable.Map<string, boolean>({
    'send-links': true,
    'send-files': true,
    'local-storage': true,
    'mirror-notifications': false,
    'api-access': false
});

let myFeatures = features.reduce((providedFeatures, provided, feature) => {
    if(provided)
        providedFeatures.push(feature);

  return providedFeatures;
}, []);

console.log(myFeatures); // [ 'send-links', 'send-files', 'local-storage' ]
```

```
const mapMap = Immutable.Map({ a: 0, b: 1, c: 2 });
mapMap.map(i => i * 30);

const mapFilter = Immutable.Map({ a: 0, b: 1, c: 2 });

mapFilter.filter(i => i % 2);

const mapReduce = Immutable.Map({ a: 10, b: 20, c: 30 });

mapReduce.reduce((acc, i) => acc + i, 0);
```

Immutable.List

`List` is the immutable version of JavaScript's array structure.

```javascript
let movies = Immutable.fromJS([ // again use fromJS for deep immutability
  {
    name: 'The Fellowship of the Ring',
    released: 2001,
    rating: 8.8
  },
  {
    name: 'The Two Towers',
    released: 2002,
    rating: 8.7
  }
]);

movies = movies.push(Immutable.Map({
    name: 'The Return of the King',
    released: 2003
}));

movies = movies.update(2, movie => movie.set('rating', 8.9)); // 0 based

movies = movies.zipWith(
  (movie, seriesNumber) => movie.set('episode', seriesNumber),
  Immutable.Range(1, movies.size + 1) // size property instead of length
);

console.log(movies);
/* writes
List [
  Map { "name": "The Fellowship of the Ring", "released": 2001, "rating": 8.8, "episod
e": 1 },
  Map { "name": "The Two Towers", "released": 2002, "rating": 8.7, "episode": 2 },
  Map { "name": "The Return of the King", "released": 2003, "rating": 8.9, "episode":
3 } ]
  */
```

Here we use the `Immutable.fromJS` call again since we have objects stored in the array. We call `push` to add items to the list, just like we would call it on an array. But since we're creating a new copy, we must rebind the variable. We have the same `set` and `update` calls when we want to update items at specific indexes. We also have access to array functions like `map`, `reduce` with support for extras like the one we're using here, `zipWith`.

Performance

Due to having to allocate memory and copy the data structure whenever any change is made, this can potentially lead to a large number of extra operations having to be performed depending on the type and number of changes. To demonstrate the difference, here is a test run. Doing memory allocation and copy on large strings can be expensive even on a shallow object.

Persistent and Transient Data Structures

Immutable data structures are also sometimes referred to as *persistent data structures*, since their values persist for their lifetime. Immutable.js provides the option for *transient data structures*, which is a mutable version of a persistent data structure during a transformation stage and returning a new immutable version upon completion. This is one approach to solving the performance issues we encountered earlier. Let's revisit the immutable case outlined in the Performance example, but using a transient data structure this time:

```javascript
import * as Immutable from 'immutable';

let list = Immutable.List();

list = list.withMutations(mutableList => {
  let val = "";

  return Immutable.Range(0, 1000000)
    .forEach(() => {
      val += "concatenation";
      mutableList.push(val);
  });
});

console.log(list.size); // writes 1000000
list.push('');
console.log(list.size); // writes 1000000
```

As we can see in this performance test, the transient list builder was still a lot slower than the fully mutable version, but much faster than the fully immutable version. Also, if you pass the mutable array to `Immutable.List` or `Immutable.fromJS`, you'll find the transient version closes the performance gap. The test also shows how slow `Object.freeze` can be compared to the other three.

Official documentation

For more information on Immutable.js, visit the official documentation at https://facebook.github.io/immutable-js/.

Pipes

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

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

Using Pipes

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

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

View Example

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: 'product-price',
    template: '<p>Total price of product is {{ price | currency: "CAD": true: "1.2-4"}
}</p>'
})
export class ProductPrice {
  price: number = 100.123456;
}
```

View Example

Chaining Pipes

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

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

@Component({
    selector: 'product-price',
    template: '<p>Total price of product is {{ price | currency: "CAD": true: "1.2-4"
| lowercase}}</p>'
})
export class ProductPrice {
  price: number = 100.123456;
}
```

[View Example](#)

Custom Pipes

Angular 2 allows you to create your own custom pipes:

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

@Pipe({name: 'length'})
export class LengthPipe implements PipeTransform {
  transform(value:string, displayMessage: boolean): string {
    return displayMessage ? `${value} ${value.length}` : `${value.length}`
  }
}
```

View Example

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';
import {LengthPipe} from './length.pipe';

@Component({
    selector: 'hello',
    template: '<div><p>{{ message | length: true }}</p><p>{{ message | length: false }
}</p></div>',
    pipes: [LengthPipe]
})
export class Hello {
  message: string = 'Hello There';
}
```

View Example

Stateful Pipes

There are two categories of pipes:

- *Stateless* pipes are pure functions that flow input data through without remembering anything or causing detectable side-effects. Most pipes are stateless. The `CurrencyPipe` we used and the length pipe we created are examples of a stateless pipe.

- *Stateful* pipes are those which can manage the state of the data they transform. A pipe that creates an HTTP request, stores the response and displays the output, is a stateful pipe. Stateful Pipes should be used cautiously.

Angular 2 provides `AsyncPipe`, which is stateful.

AsyncPipe

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

```
import {Component} from '@angular/core';
import {Observable} from 'rxjs/Observable';
@Component({
  selector: 'product-price',
  template: `
    <p>Total price of product is {{fetchPrice | async | currency: "CAD": true: "1.2-2"
}}</p>
    <p>Seconds: {{seconds | async}}</p>
  `
})
export class ProductPrice {
  count: number = 0;
  fetchPrice: Promise<number> = new Promise((resolve, reject) => {
    setTimeout(() => resolve(10), 500);
  });

  seconds: Observable<number> = new Observable(observer => {
    setInterval(() => { observer.next(this.count++); }, 1000);
  });
}
```

[View Example](#)

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.

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

@Pipe({
  name: 'delay',
  pure: false
})
export class DelayPipe implements PipeTransform {

  private fetchedValue: number;
  private fetchPromise: Promise<number>;

  transform(value: number): number {
    if (!this.fetchPromise) {
      this.fetchPromise = new Promise<number>((resolve, reject) => {
        setTimeout(() => resolve(value * 1000), value * 500);
      });

      this.fetchPromise.then((val: number) => this.fetchedValue = val);
    }
    return this.fetchedValue;
  }
}
```

View Example

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 2 is much more flexible than Angular 1 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 use built-in directives to create straightforward form components with minimal code.
- FormBuilder uses the provided APIs to handle more complex validation and subforms.

Getting Started

Opt-In APIs

Before we dive into any of the form features, we need to do a little bit of housekeeping. If you are using a current release candidate version of Angular 2, you must add two `Provider` imports to your `bootstrap` .

```
import {bootstrap} from "@angular/core";
// ...
import {disableDeprecatedForms, provideForms} from "@angular/forms";

bootstrap(MyApp, [
  // ...
  disableDeprecatedForms(),
  provideForms()
]);
```

> This override will be deprecated in future releases.

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="username">
```

Angular 2 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 2 is to take advantage of the directives provided for you.

First, consider a typical form:

```html
<form method="POST" action="/register" id="SignupForm">
  <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 2 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

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

signup-form.component.html

```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';

@Component({
  selector: 'signup-form',
  templateUrl: 'app/signup-form.component.html',
  providers: [NgForm]
})
export class SignupForm {
  registerUser (form: NgForm) {
    console.log(form.value);
    // {email: '...', password: '...'}
    // ...
  }
}
```

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

@Component({
```

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.

```html
<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 2 has a solution for.

`ngModelGroup`

When building a template-driven form in Angular 2, 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.

```html
<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

```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

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

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

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

Two-Way Binding

While Angular 2 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.

```html
<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>
```

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

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

  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.

Form Builder

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. The `FormBuilder`, 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.

FormBuilder Basics

Several steps beyond Angular 2's `ngForm` directive is the `FormBuilder` . There is a little bit of black-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.

To begin with FormBuilder, we first ensure we are working with the right directives, and the right classes. We need to load a different set of classes and directives, in order to take advantage of procedural forms. In our case, these are `FormBuilder` , `FormGroup` and `FormControl` . We get to take a bit of a shortcut, in that Angular allows us to import a list of all of the helpful directives for us.

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

app/login-form.component.ts

FormBuilder Basics

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

@Component({
  selector: 'login-form',
  templateUrl: 'app/login-form.component.html',
  directives: [REACTIVE_FORM_DIRECTIVES]
})
export class LoginForm {
  loginForm: FormGroup;
  username: FormControl;
  password: FormControl;

  constructor (builder: FormBuilder) {
    this.username = new FormControl('', []);
    this.password = new FormControl('', []);
    this.loginForm = builder.group({
      username: this.username,
      password: this.password
    });
  }

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

app/login-form.component.html

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

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

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

View Example

FormControl

You will 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 (eg: `username.root.controls.password`). Just be careful to make sure that `root` and `controls` exist, before they're used.

When constructing a `FormControl`, it takes 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 FormBuilder Forms

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

Angular 2 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 {
  // ...
  Validators
} from '@angular/forms';

@Component({ /* ... */ })
export class LoginForm {
  loginForm: FormGroup;
  username: FormControl;
  password: FormControl;

  constructor (builder: FormBuilder) {
    this.username = new FormControl('', [
      Validators.required,
      Validators.minLength(5)
    ]);
    this.password = new FormControl('', [Validators.required]);
    this.loginForm = builder.group({
      username: this.username,
      password: this.password
    });
  }

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

app/login-form.component.html

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

View Example

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.

FormBuilder Custom Validation

As useful as the built-in validators are, it is very useful to be able to include your own. Angular 2 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 };
}



// ...
this.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>
<!-- ... -->
```

[View Example](#)

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.

```javascript
function minLength (minimum) {
  return function (input) {
    return input.value >= 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

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

// ...

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

app/login-form.component.html

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

View Example

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
]);
```

View Example

Visual Cues for Users

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

```css
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 2 provides you with several classes, already accessible on your inputs.

```css
/* 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>
```

Routing

In this section we will discuss the role of routing in Single Page Applications and Angular 2'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 `RouteConfig` 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 `RouteConfig`, create a RouteConfig array which contains Route Definition Objects.

Below is the sample RouteConfig array definition:

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

See RouteConfig definition

provideRouter

`provideRouter` takes the `RouteConfig` array as an argument and returns the providers necessary for routing.

The providers returned by `provideRouter` *must* be provided when bootstrapping the application to register the routes. In the bootstrap file:

```
import { provideRouter, RouterConfig } from '@angular/router';

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

bootstrap(AppComponent, [
  provideRouter(routes)
]);
```

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

This tells the router to redirect to component-one when matching the empty path ('').

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

Defining Links Between Routes

RouterLink

Add links to routes using the `RouterLink` directive, which is part of `ROUTER_DIRECTIVES` along with `RouterOutlet` .

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

```html
<a [routerLink]="['/component-one']">Component One</a>
```

Navigating Programmatically

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

```js
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 2 dynamically adds the component for the route being activated into the `<router-outlet></router-outlet>` element.

In order to make use of the `RouterOutlet` we must give our component access to the router components by passing `ROUTER_DIRECTIVES` in the component directives array.

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

@Component({
  selector: 'app',
  directives: [ROUTER_DIRECTIVES],
  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.

View Example

> View examples running in full screen mode to see route changes in the URL.

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 `RouteConfig`:

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

Note `:id` in the path of the `product-details` route, which places the parameter in the path. For example, to see the product details page for product with ID 5, you must use the following URL: `localhost:3000/product-details/5`

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.

View Basic Example

View Example with Programmatic Route Navigation

View examples running in full screen mode to see route changes in the URL.

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 `RouteConfig` with children would look like:

```
export const routes: RouterConfig = [
  { 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 { ROUTER_DIRECTIVES, 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 -->
  `,
  directives: [ROUTER_DIRECTIVES]
})
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: RouterConfig = [
  { 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.

View Example with child routes

View Example with route params & child routes

> View examples running in full screen mode to see route changes in the URL.

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();
  }
}
```

View Example child routes accessing parent's route parameters

> View examples running in full screen mode to see route changes in the URL.

Links

Routes can be prepended with `/` , or `../` ; this tells Angular 2 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).

View Example with linking throughout route tree

> View examples running in full screen mode to see route changes in the URL.

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 { LoginRouteGuard } from './login-route-guard';
import { SaveFormsGuard } from './save-forms-guard';

import { provideRouter, RouterConfig } from '@angular/router';
import { HomePage } from './home-page';
import { AccountPage } from './account-page';

const routes: RouterConfig = [
  { path: 'home', component: HomePage },
  {
    path: 'accounts',
    component: AccountPage,
    canActivate: [LoginRouteGuard],
    canDeactivate: [SaveFormsGuard]
  }
];

bootstrap(AppComponent, [
  provideRouter(routes),
  LoginRouteGuard,
  SaveFormsGuard
]);
```

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.

See Official Definition for CanActivate

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();
  }

}
```

See Official Definition for CanDeactivate

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?');
}
```

View Example

See Official Documentation for Route Guards

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 { Router } from '@angular/router';

@Component({
  selector: 'product-list',
  template: `<!-- Show product list -->`
})
export default class ProductList {
  constructor(private router: Router) {}

  ngOnInit() {
    this.sub = this.router
      .routerState
      .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 } });
  }
}
```

View Example

See Official Documentation on Query Parameters

Redux and Ngrx

What is Redux?

Redux is an application state manager for JavaScript applications, and keeps with the core principles of the Flux-architecture by having a unidirectional data flow in your application.

Where Flux applications traditionally have multiple stores, Redux applications have only one global, read-only application state. This state is calculated by "reducing" over a collection or stream of actions that update it in controlled ways.

One popular Angular 2 specific implementation of the Redux pattern is Ng2-Redux, which we'll be using to describe how to use this approach in an application.

What is Ngrx?

Redux implementation has been very well received and has inspired the creation of ngrx, a set of modules that implement the same way of managing state as well as some of the middleware and tools in the Redux ecosystem. Ngrx was created to be used specifically with Angular 2 and RxJS, as it leans heavily on the observable paradigm.

Although we'll be using Ng2-Redux, a lot of the same lessons apply with regards to ngrx though the syntax may be different and have some slight differences in what abstractions are involved.

Resources

- Redux Documentation
- Getting Started with Redux - Egghead.io
- Ng2-Redux - Angular 2 Bindings for Redux
- ngrx - Redux style state management with RxJS and Angular 2
- Angular 2 Redux Starter Kit

Review of Reducers and Pure Functions

One of the core concepts of Redux is the *reducer*. A reducer is a function with the signature `(accumulator: T, item: U) => T`. Reducers are often used in JavaScript through the `Array.reduce` method, which iterates over each of the array's items and accumulates a single value as a result. Reducers should be *pure functions*, meaning they don't generate any side-effects.

A simple example of a reducer is the sum function:

```
let x = [1,2,3].reduce((value, state) => value + state, 0)
// x == 6
```

Reducers as State Management

This simple idea turns out to be very powerful. With Redux, you replay a series of events into the reducer and get your new application state as a result.

Reducers in a Redux application should not mutate the state, but return a copy of it, and be side-effect free. This encourages you to think of your application as UI that gets "computed" from a series of events in time.

Let's take a look at a simple counter reducer.

Simple Reducer

app/reducer/counter-reducer.ts

```
import { INCREMENT_COUNTER, DECREMENT_COUNTER } from '../actions/counter-actions';

export default function counter(state = 0, action) {
  switch (action.type) {
    case INCREMENT_COUNTER:
      return state + 1;
    case DECREMENT_COUNTER:
      return state - 1;
    default:
      return state;
  }
}
```

We can see here that we are passing in an initial state and an action. To handle each action we have set up a switch statement. Instead of each reducer needing to explicitly subscribe to the dispatcher, every action gets passed into each reducer, which handles the actions it's interested in and then returns the new state along to the next reducer.

Reducers should be side-effect free. This means that they should not modify things outside of their own scope. They should simply compute the next application state as a pure function of the reducer's arguments.

For this reason, side-effect causing operations, such as updating a record in a database, generating an id, etc. should be handled elsewhere in the application such as in the action creators, using middleware such as 'Epics' from redux-observable or ngrx/effects.

Another consideration when creating your reducers is to ensure that they are immutable and not modifying the state of your application. If you mutate your application state, it can cause unexpected behavior. There are a few ways to help maintain immutability in your reducers. One way is by using new ES6 features such as `Object.assign` or the spread operator for arrays.

```javascript
function immutableObjectReducer(state = { someValue: 'value'} , action) {
  switch(action.payload) {
    case SOME_ACTION:
      return Object.assign({}, state, { someValue: action.payload.value });
    default:
      return state;
  }
}

function immutableArrayReducer(state = [1,2,3], action) {
  switch(action.payload) {
    case ADD_ITEM:
      return [...state,action.payload.value]
    default:
      return state;
  }
}
```

However, when dealing with complex or deeply nested objects, it can be difficult to maintain immutability in your application using this syntax. This is where a library like ImmutableJS can help.

Redux Actions

Redux actions should generally be simple JSON objects. This is because they should be serializable and replayable into the application state. Even if your actions involve asynchronous logic, the final dispatched action should remain a plain JSON object.

Redux action creators are generally where side-effects should happen, such as making API calls or generating IDs. This is because when the final action gets dispatched to the reducers, we want to update the application state to reflect what has already happened.

Let's take a look at the actions that are used in this example. For now, let's just focus on some simple synchronous actions.

Synchronous Actions

Most Redux apps have a set of functions, called "action creators", that are used to set up and dispatch actions.

In Angular 2, it's convenient to define "action creator services" for your action creators to live in; these services can be injected into the components that need to dispatch the actions.

app/actions/counter-actions.ts

```typescript
import { Injectable } from '@angular/core';
import { NgRedux } from 'ng2-redux';

export const INCREMENT_COUNTER = 'INCREMENT_COUNTER';
export const DECREMENT_COUNTER = 'DECREMENT_COUNTER';

@Injectable()
export class CounterActions {
  constructor(private redux: NgRedux<any>) {}

  increment() {
    this.redux.dispatch({ type: INCREMENT_COUNTER });
  }

  decrement() {
    this.redux.dispatch({ type: DECREMENT_COUNTER });
  }
}
```

As you can see, the action creators are simple functions that (optionally) take parameters, and then dispatch a JSON object containing more information.

The `dispatch` function expects to be called with something that conforms to the "Action" interface from the Redux library:

```
import { Action } from 'redux';
```

This interface has the following properties:

- *type* - a string/enum representing the action
- *payload?* - optional, the data that you want to pass into the reducer if applicable
- *error?* - optional, indicates if this message is due to an error
- *metaData?* - optional - any extra information

Asynchronous Actions

This "ActionCreatorService" pattern comes in handy if you must handle asynchronous or conditional actions (users of react-redux may recognize this pattern as analogous to redux-thunk in a dependency-injected world).

app/actions/counter-actions.ts

```
import { Injectable } from '@angular/core';
import { NgRedux } from 'ng2-redux';

export const INCREMENT_COUNTER = 'INCREMENT_COUNTER';
export const DECREMENT_COUNTER = 'DECREMENT_COUNTER';

@Injectable()
export class CounterActions {
  constructor(private redux: NgRedux<any>) {}

  // ...

  incrementIfOdd() {
    const { counter } = this.redux.getState();

    if (counter % 2 === 0) return;
    this.redux.dispatch({ type: INCREMENT_COUNTER });
  }

  incrementAsync(timeInMs = 1000) {
    this.delay(timeInMs).then(() => this.redux.dispatch({ type: INCREMENT_COUNTER }));
  }

  private delay(timeInMs) {
    return new Promise((resolve, reject) => {
      setTimeout(() => resolve() , timeInMs);
    });
  }
}
```

In the `incrementIfOdd` action, we are using the `getState` function to get the current state of the application.

In the `incrementAsync` action, we are delaying the actual call to `dispatch`. For example, we have created a Promise that will resolve after the delay. Once the Promise resolves, we can then do a dispatch with the increase action.

View Ng2-Redux Example View Ngrx Example

Actions that Depend on Other Services

The ActionCreatorService pattern becomes necessary in cases where your action creators must use other Angular 2 services. Consider the following ActionCreatorService that handles a remote API call:

```typescript
import { Injectable } from '@angular/core';
import { NgRedux } from 'ng2-redux';
import { AuthService } from '../services/auth/';

@Injectable()
export class SessionActions {
  static LOGIN_USER_PENDING = 'LOGIN_USER_PENDING';
  static LOGIN_USER_SUCCESS = 'LOGIN_USER_SUCCESS';
  static LOGIN_USER_ERROR = 'LOGIN_USER_ERROR';
  static LOGOUT_USER = 'LOGOUT_USER';

  constructor(
    private ngRedux: NgRedux<any>,
    private authService: AuthService) {}

  loginUser(credentials) {
    const username = credentials.username;
    const password = credentials.password;

    this.ngRedux.dispatch({ type: SessionActions.LOGIN_USER_PENDING });

    this.authService.login(username, password)
      .then(result => this.ngRedux.dispatch({
          type: SessionActions.LOGIN_USER_SUCCESS,
          payload: result
      }))
      .catch(() => this.ngRedux.dispatch({
         type: SessionActions.LOGIN_USER_ERROR
      }));
  };

  logoutUser = () => {
    this.ngRedux.dispatch({ type: SessionActions.LOGOUT_USER });
  };
}
```

239

Configuring your Application to use Redux

Once you have the reducers and actions created, it is time to configure your Angular 2 application to make use of Ng2-Redux. For this, we will need to:

- Register Ng2-Redux with Angular 2
- Create our application reducer
- Create and configure a store

Registering Ng2-Redux with Angular 2

app/boot.ts

```
import { bootstrap } from '@angular/platform-browser-dynmic'
import { SimpleRedux } from './containers/app-container'
import { NgRedux } from 'ng2-redux'

bootstrap(
  SimpleRedux,
  [ NgRedux, /* ... */ ]);
```

Here, we're simply adding the `NgRedux` class as a provider in our bootstrap file.

Create our Application Reducer

app/reducers/index.ts

```
import { combineReducers } from 'redux';
import counter from './counter-reducer';

export default combineReducers({
  counter
});
```

`combineReducers` allows us to break out our application into smaller reducers with a single area of concern. Each reducer that you pass into it will control a property on the state. So when we are subscribing to our state changes with Ng2-Redux's `@select` decorator, we are able to select a counter property, or any other reducers you have provided.

Create and Configure a Store

Next we want Ng2-Redux to configure our store based on settings we provide. This should be done once, in the top-level component of your application.

app/containers/SimpleRedux.ts

```
import { Component } from '@angular/core';
import { NgRedux } from 'ng2-redux';
import logger from '../store/configure-logger';
import reducer from '../reducers/index';

@Component({
  // ...
})
class SimpleRedux {
  constructor(ngRedux: NgRedux) {
    const initialState = {};
    const middleware = [ logger ];
    ngRedux.configureStore(reducer, initialState, middleware);
  }
}
```

In this example we are creating a store that uses the `redux-logger` middleware, which will add some logging functionality to the application.

Using Redux with Components

We will use the select pattern from Ng2-Redux to bind our components to the store. To demonstrate how this works, let's take a look at a small example with a counter component.

Counter Example

Let's start by building out a counter component. The component will be responsible for keeping track of how many times it was clicked and displaying that amount.

app/components/counter-component.ts

```typescript
import { Component } from '@angular/core';
import { select } from 'ng2-redux';
import { CounterActions } from '../actions/counter-actions';

@Component({
  selector: 'counter',
  providers: [ CounterActions ],
  template: `
  <p>
    Clicked: {{ counter$ | async }} times
    <button (click)="actions.increment()">+</button>
    <button (click)="actions.decrement()">-</button>
    <button (click)="actions.incrementIfOdd()">Increment if odd</button>
    <button (click)="actions.incrementAsync()">Increment async</button>
  </p>
  `
})
export class Counter {
  @select() counter$: Observable<number>;

  constructor(public actions: CounterActions) {}
}
```

View Example

The template syntax should be familiar by now, displaying a `Observable` counter with the async pipe and handling some click events.

In this case, the click events are bound to expressions that call our action creators from the `CounterActions` ActionCreatorService.

Let's take a look at the use of `@select` .

`@select` is a feature of Ng2-Redux which is designed to help you attach your store's state to your components in a declarative way. You can attach it to a property of your component class and Ng2-Redux will create an `Observable` and bind it to that property for you.

In this case, `@select` has no parameters, so Ng2-Redux will look for a store property with the same name as the class variable. It omits the trailing `$` since that's simply a naming convention for `Observables`.

So now, any time `store.counter` is updated by a reducer, `counter$` will receive the new value and `| async` will update it in the template.

Note that `@select` supports a wide range of arguments to allow you to select portions of your Redux store with a great deal of flexibility. See the Ng2-Redux docs for more details.

The Ng2-Redux "select pattern" style differs a bit from the "connect" style used by `react-redux`; however by using Angular 2's DI and TypeScript's decorators, we can have a nicely declarative binding where most of the work is done in the template. We also get the power of `Observables` and `OnPush` change detection for better performance.

Either way, we still benefit from the Redux fundamentals of reducers and one-way data-flow.

Redux and Component Architecture

In the above example, our `counter` component is a smart component. It knows about Redux, the structure of the state and the actions it needs to call. In theory you can drop this component into any area of your application and just let it work. But it will be tightly bound to that specific slice of state and those specific actions. For example, what if we wanted to have multiple counters tracking different things on the page? For example, counting the number of red clicks vs blue clicks.

To help make components more generic and reusable, it's worth trying to separate them into *container* components and *presentational* components.

	Container Components	**Presentational Components**
Location	Top level, route handlers	Middle and leaf components
Aware of Redux	Yes	No
To read data	Subscribe to Redux state	Read state from @Input properties
To change data	Dispatch Redux actions	Invoke callbacks from @Output properties

[redux docs](#)

Keeping this in mind, let's refactor our `counter` to be a *presentational* component. First, let's modify our `app-container` to have two counter components on it as we currently have it.

```
import { Component } from '@angular/core';
import { Counter } from '../components/counter-component';

@Component({
  selector: 'simple-redux',
  directives: [ Counter ]
  template: `
    <div>
      <h1>Redux: Two components, one state.</h1>
      <div style="float: left; border: 1px solid red;">
        <h2>Click Counter</h2>
        <counter></counter>
      </div>
      <div style="float: left; border: 1px solid blue;">
        <h2>Curse Counter</h2>
        <counter></counter>
      </div>
    </div>
  `
})
export class SimpleRedux {}
```

[View Example](#)

As you can see in the example, when clicking on the buttons the numbers in both components will update in sync. This is because the counter component is coupled to a specific piece of state and action.

Looking at the example, you can see that there is already an *app/reducers/curse-reducer.ts* and *app/actions-curse-actions.ts*. They are pretty much the same as the counter actions and counter reducer, we just wanted to create a new reducer to hold the state of it.

To turn the counter component from a smart component into a dumb component, we need to change it to have data and callbacks passed down into it. For this, we will pass the data into the component using `@Input` properties, and the action callbacks as `@Output` properties.

We now have a nicely-reusable presentational component with no knowledge of Redux or our application state.

app/components/counter-component.ts

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

@Component({
  selector: 'counter',
  template: `
  <p>
    Clicked: {{ counter$ | async }} times
    <button (click)="increment.emit()">+</button>
    <button (click)="decrement.emit()">-</button>
    <button (click)="incrementIfOdd.emit()">Increment if odd</button>
    <button (click)="incrementAsync.emit()">Increment async</button>
  </p>
  `
})
export class Counter {
  @Input() counter$: Observable<number>;
  @Output() increment = new EventEmitter<void>();
  @Output() decrement = new EventEmitter<void>();
  @Output() incrementIfOdd = new EventEmitter<void>();
  @Output() incrementAsync = new EventEmitter<void>();
}
```

Next, let's modify the main app container to hook up these inputs and outputs to the template.

`@Component` *app/src/containers/app-containter.ts*

```
@Component({
  selector: 'simple-redux',
  providers: [ CounterActions, CurseActions ],
  directives: [ Counter ],
  template: `
<div>
  <h1>Redux: Presentational Counters</h1>
  <div style="float: left; border: 1px solid red;">
    <h2>Click Counter</h2>
    <counter [counter$]="counter$"
        (increment)="counterActions.increment()"
        (decrement)="counterActions.decrement()"
        (incrementIfOdd)="counterActions.incrementIfOdd()"
        (incrementAsync)="counterActions.incrementAsync()">
    </counter>
  </div>
  <div style="float: left; border: 1px solid blue;">
    <h2>Curse Counter</h2>
    <counter [counter$]="curse$"
        (increment)="curseActions.castCurse()"
        (decrement)="curseActions.removeCurse()"
        (incrementIfOdd)="curseActions.castIfOdd()"
        (incrementAsync)="curseActions.castAsync()">
    </counter>
  </div>
</div>
    `
})
```

At this point, the template is attempting to call actions on our two ActionCreatorServices,
`CounterActions` and `CurseActions` ; we just need to hook those up using Dependency
Injection:

app/src/containers/app-container.ts

```
import { Component, View, Inject, OnDestroy, OnInit } from '@angular/core';
import { select } from 'ng2-redux';
import { Counter } from '../components/counter-component';
import { CounterActions } from '../actions/counter-actions';
import { CurseActions } from '../actions/curse-actions';

@Component({ /* see above .... */})
export class SimpleRedux {
  @select() counter$: Observable<number>;
  @select() curse$: Observable<number>;

  constructor(
    public counterActions: CounterActions,
    public curseActions: CurseActions,
    redux: NgRedux) {
      const initialState = {};
      const middleware = [ logger ];
      redux.configureStore(reducer, initialState, middleware);
    }
}
```

View Ng2-Redux Example View Ngrx Example

Our two `Observable` s, `counter$` and `curse$` , will now get updated with a new value every time the relevant store properties are updated by the rest of the system.

Getting More From Redux and Ngrx

Redux

Redux has a number of tools and middleware available in its ecosystem to facilitate elegant app development.

- *Redux DevTools* - a tool that displays a linear timeline of actions that have interacted with its store. Allows for replaying actions and error handling
- *redux-thunk* - middleware that enables lazy evaluation of actions
- *redux-observable* - an RxJS-based model for handling side-effects on action streams.
- *ng2-redux-router - reactive glue between the Angular 2 router and your redux store.

Ngrx

Ngrx provides most of its Redux implementation through the ngrx/store module. Other modules are available for better integration and development.

- *ngrx/store-devtools* - an ngrx implementation of the Redux DevTools
- *ngrx/effects* - a model for performing side-effects similar to `redux-saga`
- *ngrx/router* and *ngrx/router-store* - a router for Angular 2 that can be connected to your ngrx store

TDD 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 2 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 2 application. Then, using that DOM instance that is running our Angular 2 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.

Test Setup

The repo angular2-redux-starter is a basic webpack-based Angular 2 application (with Redux) with the same testing toolchain outlined above. Let's take a look at how this project is set up.

Filename Conventions

Each unit test is put into its own separate file. The Angular 2 team recommends putting unit test scripts alongside the files they are testing and using a `.spec` filename extension to mark it as a testing script (this is a Jasmine convention). So if you had a component `/app/components/mycomponent.ts`, then your unit test for this component would be in `/app/components/mycomponent.spec.ts`. This is a matter of personal preference; you can put your testing scripts wherever you like, though keeping them close to your source files makes them easier to find and gives those who aren't familiar with the source code an idea of how that particular piece of code should work.

Karma Configuration

Karma is the foundation of our testing workflow. It brings together our other testing tools to define the framework we want to use, the environment to test under, the specific actions we want to perform, etc. In order to do this Karma relies on a configuration file *karma.config.js*. You can seed a new configuration file though the `karma init` command, which will guide you through a few basic questions to get a bare minimum setup running. If we take a look at the *karma.config.js* file in angular-redux-starter we'll see a few important points of interest:

Karma Configuration

```javascript
module.exports = (config) => {
  config.set({
    frameworks: [
      'jasmine',
      'source-map-support',
    ],

    files: ['./src/tests.entry.ts'],

    preprocessors: {
      './src/**/*.ts': [
        'webpack',
        'sourcemap',
      ],
      './src/**/!(*.test|tests.*).ts': [
        'coverage',
      ],
    },

    webpack: {...},

    webpackServer: {
      noInfo: true, // prevent console spamming when running in Karma!
    },

    reporters: [
      'spec',
      'coverage',
    ],
    coverageReporter: {
      reporters: [
        { type: 'json' },
        { type: 'html' },
      ],
      dir: './coverage/',
      subdir: (browser) => {
        return browser.toLowerCase().split(/[ /-]/)[0]; // returns 'chrome'
      },
    },

    port: 9999,
    browsers: ['Chrome'], // Alternatively: 'PhantomJS'
    singleRun: true,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    captureTimeout: 6000,
  });
};
```

The configuration file is put together by exporting a function that accepts the configuration object that Karma is going to work with. Modifying certain properties on this object will tell Karma what it is we want to do. Let's go over some of the key properties used in this configuration file:

- `frameworks` is a list of the testing frameworks we want to use. These frameworks must be installed through NPM as a dependency in our project or/and as a Karma plugin.
- `files` is a list of files to be loaded into the browser/testing environment. This can also take the form of a glob pattern as it becomes rather tedious to manually add in a new file for each new testing script created. In the angular2-redux-starter *karma.config.js* we have put the testing files we wish to include in a separate file - *src/tests.entry.ts*, which includes a `require` call using a regex pattern for importing files with the **.spec.ts** file extension. As a project grows larger and the number of files to include grows in complexity it is good practice to put file imports in a separate file - this keeps the *karma.config.js* file cleaner and more readable. Here is what our *src/tests.entry.ts* looks like:

```
let testContext = (<{ context?: Function }>require).context('./', true, /\.test\.ts/);
testContext.keys().forEach(testContext);
```

- `preprocessors` allow for some operation to be performed on the contents of a unit testing file before it is executed. These operations are carried out through the use of Karma plugins and are often used for transpiling operations. Since we are writing unit tests in TypeScript, *.ts* files must be transpiled into plain Javascript in order to run in a browser-based environment. In angular2-redux-starter this process is done with webpack, so we explicitly invoke the `webpack` processor on all of our testing files (those ending with *.spec.ts*). We also load any source map files originating from transpilation through the `sourcemap` processor.
- `coverageReporter` is used to configure the output of results of our code coverage tool (our toolchain uses Istanbul). Here we have specified to output the results in JSON and HTML. Reports will appear in the *coverage/* folder.
- `reporters` is a list of reporters to use in the test cycle. Reporters can be thought of as modular tools used to report on some aspect of the testing routine outside of the core unit tests. Code coverage is an example of a reporter - we want it to report on how much of our code is being tested. There are many more reporters available for Karma that can aid in crafting your testing workflow.
- If the project uses webpack, then the property `webpack` in the Karma configuration object is where we can configure webpack with Karma. Using webpack we can configure how to bundle our unit tests; that is, whether to pack all tests into a single bundle, or each unit test in its own bundle, etc. Regardless, unit tests should not be bundled with the rest of the applications code (especially in production!). In angular2-

redux-starter we have opted to bundle all unit tests together.

- `port` , `browsers` and `singleRun` configure the environment our unit tests will run under. The `browsers` property specifies which browser we want Karma to launch and capture output from. We can use Chrome, Firefox, Safari, IE or Opera (requires additional Karma launcher to be installed for each respective browser). For a browser-less DOM instance we can use PhantomJS (as outlined in the toolchain section). We can also manually capture output from a browser by navigating to `http://localhost:port` , where port is the number specified in the `port` property (the default value is 9876 if not specified). The property `singleRun` controls how Karma executes, if set to `true` , Karma will start, launch configured browsers, run tests and then exit with a code of either `0` or `1` depending on whether or not all tests passed.

This is just a sample of the core properties in *karma.config.js* being used by angular2-redux-starter project. There are many more properties that can be used to extend and configure the functionality of Karma - take a look at the official documentation for the full API breakdown.

Typings

Since our project and unit tests are written in TypeScript, we need type definitions for the libraries we'll be writing our tests with (Chai and Jasmine). In angular2-redux-starter we have included these type definitions in *typings.json*.

Executing Test Scripts

Our entire testing workflow is done through Karma. Run the command `karma start` to kickstart Karma into setting up the testing environment, running through each unit test and executing any reporters we have set up in the *karma.config.js* configuration file. In order to run Karma through the command line it must be installed globally (`npm install karma -g`).

A good practice is to amalgamate all the project's task/build commands through npm. This gives continuity to your build process and makes it easier for people to test/run your application without knowing your exact technology stack. In *package.json* the `scripts` field holds an object with key-value pairing, where the key is the alias for the command and the value is the command to be executed.

```
...
"scripts": {
    "test": "karma start",
    ...
}
...
```

Now running `npm test` will start Karma. Below is the output of our Karma test. As you can see we had one test that passed, running in a Chrome 48 browser.

Figure: image

Simple Test

To begin, let's start by writing a simple test in Jasmine.

```
import {
  describe,
  expect,
  it
} from '@angular/core/testing';

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.

Using Chai

Chai is an assertion library with some tasty syntax sugar that can be paired with any other testing framework. It lets us write tests in a TDD (Test Driven Development) style or BDD (Behavior Driven Development) style. We already know what TDD is (read the intro!), so what exactly is BDD? Well BDD is the combination of using TDD with natural language constructs (English-like sentences) to express the behavior and outcomes of unit tests. Jasmine already uses a TDD style, so we'll be using Chai for its BDD interfaces, mainly through the use of `should` and `expect`.

```
import {
  describe,
  expect,
  it
} from '@angular/core/testing';

describe('Testing math', () => {
  it('multiplying should work', () => {
    let testMe = 16;

    // Using the expect interface
    chai.expect(testMe).to.be.a('number');
    chai.expect(testMe).to.equal(16);

    // Using the should interface
    chai.should();
    testMe.should.be.a('number');
    testMe.should.equal(16);
  });
});
```

The `expect` and `should` interface both take advantage of chaining to construct English-like sentences for describing tests. Once you've decided on a style you should maintain that style for all your other tests. Each style has its own unique syntax; refer to the documentation for that specific API.

Testing Components

Testing Angular 2 components requires some insight into the Angular 2 `@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 2 components fairly easily - after all, Angular 2 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

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

@Component({
  selector: 'display-message',
  template: '<h1>{{message}}</h1>'
})

export class MessageComponent {
  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 {
  describe,
  expect,
  it
} from '@angular/core/testing';

import {MessageComponent} from './message.component';

describe('Testing message state in message.component', () => {
  beforeEach(() => {
    this.app = new MessageComponent();
  });

  it('should set new message', () => {
    this.app.setMessage('Testing');
    expect(this.app.message).toBe('Testing');
  });

  it('should clear message', () => {
    this.app.clearMessage();
    expect(this.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 2, 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

```typescript
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 to initiate the `QuoteComponent` class. The Angular testing library offers a way to inject any dependencies this class relies on, as well as a `TestComponentBuilder` object which will create the component for us and return a *component fixture* that we can perform testing operations on.

quote.spec.ts

```
class MockQuoteService {
  quote: 'Test quote';

  getQuote() {
    return Promise.resolve(this.quote);
  }
}

import { QuoteService } from './quote.service';
import { QuoteComponent } from './quote.component';
import { provide } from '@angular/core';
import {
  async,
  expect,
  it,
  describe,
  inject,
  beforeEach,
  addProviders,
} from '@angular/core/testing';

import {
  TestComponentBuilder,
  ComponentFixture,
}
from '@angular/compiler/testing';

describe('Testing Quote Component', () => {

  beforeEach(()=>
    addProviders([{provide: QuoteService, useClass: MockQuoteService }]));

  it('Should get quote',
    async(inject([TestComponentBuilder], (tcb: TestComponentBuilder) => {
      tcb.createAsync(QuoteComponent).then(fixture => {

        fixture.componentInstance.getQuote();

        fixture.whenStable().then(() => {
          fixture.detectChanges();
          let compiled = fixture.debugElement.nativeElement;
          expect(compiled.querySelector('div')).toHaveText('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 `addProviders` 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? Using `provide` we inject the `QuoteService` dependency using our mock class `MockQuoteService` , where we will provide mock data for the component to consume.

Next we use `inject` to inject the `TestComponentBuilder` into our test. We wrap this in `async()` so our tests run in an asynchronous test zone. Once we have a reference to `TestComponentBuilder` we call `createAsync` to create the component we will be testing (`QuoteComponent`). The `TestComponentBuilder` will then create a new instance of our component, fulfilling any Angular-specific routines like dependency injection. Using `async` creates a test zone which will ensure that all asynchronous functions have resolved prior to ending the test.

`TestComponentBuilder` will return a *fixture* for us to use in our tests. 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 for testing components and we use it extensively. Here we have gotten access to our component through the `fixture.debugElement.componentInstance` property where we call `getQuote` to kickstart our only action in the `QuoteComponent` . We run the test when the fixture is stable by using its `whenStable` method which will ensure the promise inside `getQuote()` resolves, 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.

Overriding Components for Testing

In some components, providers are not directly injected through the constructor but instead defined through a decorator. Consider the following component:

```
@Component({
  selector: 'example',
  template: '<div>Simple example</div>',
  providers: [ExampleService]
});
class SimpleComponent {}
```

This won't work when using `addProviders`. Instead we can use the `TestComponentBuilder` to explicitly inject the `ExampleService` provider through `overrideProviders`. As we did before, you should create a mocked version of the `ExampleService` to feed in data you expect.

```
it('Should work', async(inject(
  [TestComponentBuilder], (tcb: TestComponentBuilder) => {
    tcb.overrideProviders(SimpleComponent, [
      provide(ExampleService, {useClass: MockExampleService})
    ]).createAsync(SimpleComponent).then(fixture => {

      // test your fixture here

    });
  }))
);
```

`TestComponentBuilder` also lets you override a component's template. 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. Calling `overrideTemplate` will set the component's template to whatever you pass in.

```
it('Should work', async(inject(
  [TestComponentBuilder], (tcb: TestComponentBuider) => {
    tcb.overrideTemplate(SimpleComponent, '<span>{{message}}</span>')
      .createAsync(SimpleComponent).then(fixture => {

        // test all things relating to the message property here


      // test all things relating to the message property here


    });
  }))
);
```

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.

Suppose we had a component with a button that will trigger an action within a setTimeout of two seconds. Here's what our component and unit test might look like:

```
@Component({
  selector: 'example',
  template: `
      <span>{{message}}</span>
      <button (click)="performAction()">Click me</button>
    `})

class SampleComponent {
  message: string;

  performAction () {
    setTimeout(() => {
      this.message = 'My expected data';
    }, 2000);
  }
}
```

```
it('Should work',
  fakeAsync(inject([TestComponentBuilder], (tcb: TestComponentBuilder) => {
    tcb.createAsync(SampleComponent).then(fixture => {
      fixture.debugElement.nativeElement.querySelector('button')
        .click();

      tick(2000);

      fixture.detectChanges();
      expect(fixture.debugElement.nativeElement.querySelector('span'))
        .toHaveText('My expected data');
    });
  }));
);
```

Here we have a `SampleComponent` that has a button, when clicked a `setTimeout` of two seconds will be called to set the message property to 'My expected data'. Our unit test builds our component using the `TestComponentBuilder` . We have wrapped our entire test in `fakeAsync` which will allow us to test the asynchronous behavior of our component using synchronous function calls. We simulate a button click and then immediately call `tick(2000)` which simulates a two second delay. 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.

Testing Services

When testing services in Angular 2 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 calling `addProviders` in the `beforeEach` part of your unit test to feed in required dependencies like the HTTP module.

Mocking Dependencies

Since we want to isolate our service testing from outside network requests, we have to create mock requests where we can manually set up properties like expected responses, errors, headers, etc. Fortunately Angular 2 makes this an easy process by providing a way to make fake HTTP calls through the `MockBackend` class. This class can be injected wherever a service is expecting the HTTP module and used by the service as though it were the real HTTP module - except now the results/data are set up by us, without any network request taking place. Let's take a look at some code:

wikisearch.ts

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


@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. We want to use `MockBackend` to inject a mocked version of HTTP, so that any call to http.get will allow us to feed in whatever data we want - not the data from wikipedia.org. Lets take a look at our unit test:

wikisearch.spec.ts

```
import {
  BaseRequestOptions,
  Response,
  ResponseOptions,
  ConnectionBackend,
  Http
} from '@angular/http';

import {
  it,
  expect,
  describe,
  addProviders,
  inject
} from '@angular/core/testing';

import {MockBackend} from '@angular/http/testing';
import {provide} from '@angular/core';
import {SearchWiki} from './wikisearch';

describe('Testing the wikipedia search service', () => {
  beforeEach(() => {
    addProviders([
      MockBackend,
      BaseRequestOptions,
      SearchWiki,
      provide(
        Http, {
          useFactory: (
            mockbackend: ConnectionBackend,
            defaultOptions: BaseRequestOptions
          ) => {
            return new Http(mockbackend, defaultOptions);
          },
          deps: [MockBackend, BaseRequestOptions]
        }
      )
    ]);
  });
});
```

As you can see, we have quite a few imports here to set up our mocked HTTP module. We use `addProviders` to include `MockBackend` to create our mocked HTTP module. In order to properly set up our mocked module we also include `BaseRequestOptions` and `ConnectionBackend` . We create an instance of `Http` that our service will get by using a factory.

To create this instance the `Http` class takes two parameters: the back-end to use (`ConnectionBackend` type) and the default options to use (`RequestOptions` type). Since we are planning to use a mock back-end, we feed in `MockBackend` to the first parameter. The second parameter, default options, doesn't have any bearing on the mocking aspects we plan on using, so we use `BaseRequestOptions` , which is the default implementation.

Right now this test doesn't do anything aside from setting up a mockable HTTP module - let's take a look at some actual tests.

Testing HTTP Requests

Now that we have set up our service and created a mocked version of the HTTP module, we are ready to start testing. The basic strategy for testing this 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 (calling search will return the data we expect it to).

Asynchronous Testing

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.

In the `@angular/testing` module there are a few functions that allow us to fake asynchronous behavior. This is done by wrapping the entire test in a call to `fakeAsync`, and then using calls to `tick` to move the test along. FakeAsync essentially puts our test into a zone, and then listens for any asynchronous operations like setTimeouts, Promises, callbacks etc. Those functions are not actually called asynchronously; instead Angular will rely on us calling tick, which will call those functions immediately, wait for their execution to finish and then proceed, thus simulating time elapsing.

Now that we have an idea of how to test our asynchronous services, let's take a look at a test to verify the search method of our WikiSearch service:

wikisearch.spec.ts

```
describe('verify search', () => {
  it('searches for the correct term',
    fakeAsync(inject([SearchWiki, MockBackend], (searchWiki, mockBackend) => {
      const expectedUrl = 'https://en.wikipedia.org/w/api.php?' +
        'action=query&list=search&srsearch=Angular';

      mockBackend.connections.subscribe((conn) => {
        expect(conn.request.url).toBe(expectedUrl);

        let response = new ResponseOptions({
          body: {
            query: {
              searchInfo: { totalhits: 1 }
            },
            search: [
              {
                ns: 0,
                title: 'Angular',
                size: 840,
                wordcount: 115
              }
            ]
          }
        });

        conn.mockRespond(new Response(response));
      });

      let result;
      searchWiki.search('Angular').subscribe((res) => {
        result = res;
      });

      tick();

      expect(result.query.searchInfo.totalhits).toBe(1);
    }))));
});
```

Our testing strategy is fairly straightforward - we check to see that our service has requested the right URL, and once we've responded with mock data we want to verify that our service returns that same data.

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 make a call to `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 2 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. Notice that we call `tick` before checking the contents of the search result. This is to ensure that all `Observable`s, `Promise`s, etc. and any other asynchronous operations complete before proceeding. 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.

Testing JSONP and XHR Back-Ends

Some services take advantage of the JSONP or XHR module to fetch data instead of the traditional HTTP module. We use the same strategies for testing these services - create a mock back-end, initialize the service and test to see if the request our service made is correct and if the data mocked through the back-end makes its way successfully to the service. Fortunately services that rely on the XHR module are tested *exactly* the same way as services that use the HTTP module. The only difference is in which class is used to mock the back-end. In services that use the HTTP module, the `MockBackend` class is used; in those that use XHR, the `XHRBackend` is used instead. Everything else remains the same.

Unfortunately services that use the JSONP module use a significantly different class for mocking the back-end. The class `MockBrowserJsonp` is used for this scenario.

Executing Tests Asynchronously

Since services operate in an asynchronous manner it may be useful to execute a service's entire unit test asynchronously. This can speed up the overall time it takes to complete a full testing cycle since a particular long unit test will not block other unit tests from executing. We can set up our unit test to return a promise, which will resolve as either a success or failure depending on the activity of the test.

```javascript
describe('verify search', () => {
  it('searches for the correct term',
    fakeAsync(inject([SearchWiki, MockBackend], (searchWiki, mockBackend) => {
        return new Promise((pass, fail) => {
          ...
        });
    })));
});
```

Instead of only using `inject`, we use `fakeAsync` to wrap it and fulfill dependencies and execute the test in an asynchronous process. Using `fakeAsync` requires us to return a Promise, which we use to resolve the competition of our test by calling `pass`, or `fail`, depending on the results of our test.

Testing Redux

Unit testing Redux is a very straightforward process. There are two primary units:

- *Reducers* are pure functions that lend themselves well to testing.

- *Actions* trigger changes in a Redux system. There are two broad categories of actions: synchronous (which are quite simple to test) and asynchronous (which are slightly more involved).

The examples below should provide you with a strong foundation for testing Redux applications.

Testing Simple Actions

Consider the following simple actions, from the Redux chapter of this book:

```typescript
import { Injectable } from '@angular/core';
import { NgRedux } from 'ng2-redux';

export const INCREMENT_COUNTER = 'INCREMENT_COUNTER';
export const DECREMENT_COUNTER = 'DECREMENT_COUNTER';

@Injectable
export class CounterActions {
  constructor(private redux: NgRedux<any>) {}

  increment() {
    this.redux.dispatch({ type: INCREMENT_COUNTER });
  }

  decrement() {
    this.redux.dispatch({ type: DECREMENT_COUNTER });
  }
}
```

These are pretty straightforward to test:

```typescript
import { NgRedux } from 'ng2-redux';
import {
  CounterActions,
  INCREMENT_COUNTER,
  DECREMENT_COUNTER,
} from './counter';
import {
  beforeEach,
  describe,
  expect,
  it,
} from '@angular/core/testing';

// Mock out the NgRedux class with just enough to test what we want.
class MockRedux extends NgRedux<any> {
  constructor() {
    super(null);
  }
  dispatch = () => undefined;
}

describe('counter action creators', () => {
```

```
    let actions: CounterActions;
    let mockRedux: NgRedux<any>;

    beforeEach(() => {
      // Initialize mock NgRedux and create a new instance of the
      // ActionCreatorService to be tested.
      mockRedux = new MockRedux();
      actions = new CounterActions(mockRedux);
    });

    it('increment should dispatch INCREMENT_COUNTER action', () => {
      const expectedAction = {
        type: INCREMENT_COUNTER
      };

      spyOn(mockRedux, 'dispatch');
      actions.increment();

      expect(mockRedux.dispatch).toHaveBeenCalled();
      expect(mockRedux.dispatch).toHaveBeenCalledWith(expectedAction);
    });

    it('decrement should dispatch DECREMENT_COUNTER action', () => {
      const expectedAction = {
        type: DECREMENT_COUNTER
      };

      spyOn(mockRedux, 'dispatch');
      actions.decrement();

      expect(mockRedux.dispatch).toHaveBeenCalled();
      expect(mockRedux.dispatch).toHaveBeenCalledWith(expectedAction);
    });
  });
```

We just make sure that our action creators do indeed dispatch the correct actions.

Testing Complex Actions

Things get a little trickier when we want to test asynchronous or conditional action creators. Our goal is still the same: make sure that our operations are dispatching the actions we're expecting.

A Conditional Action

Consider the following conditional action (i.e., one that is fired depending on current state):

```
import { Injectable } from '@angular/core';
import { NgRedux } from 'ng2-redux';
export const INCREMENT_COUNTER = 'INCREMENT_COUNTER';

@Injectable()
export class MyActionService {
  constructor(private redux: NgRedux) {};

  // A conditional action
  incrementIfOdd() {
    const { counter } = this.redux.getState();

    if (counter % 2 === 0) return;
    this.redux.dispatch({ type: INCREMENT_COUNTER });
  }
}
```

Unit testing is similar to before, but we need to mock our state as well as dispatch:

```
import { NgRedux } from 'ng2-redux';
import { CounterActions } from './counter';
import {
  beforeEach,
  describe,
  expect,
  it,
} from '@angular/core/testing';

class MockRedux extends NgRedux<any> {
  constructor(private state: any) {
    super(null);
  }
  dispatch = () => undefined;
  getState = () => this.state;
}
```

```
describe('counter action creators', () => {
  let actions: CounterActions;
  let mockRedux: NgRedux<any>;
  let mockState: any = {};

  beforeEach(() => {
    // Our mock NgRedux can now accept mock state as a constructor param.
    mockRedux = new MockRedux(mockState);
    actions = new CounterActions(mockRedux);
  });

  it('incrementIfOdd should dispatch INCREMENT_COUNTER action if count is odd',
    () => {
      // Our tests can bake in the initial state they need.
      const expectedAction = {
        type: CounterActions.INCREMENT_COUNTER
      };

      mockState.counter = 3;
      spyOn(mockRedux, 'dispatch');
      actions.incrementIfOdd();

      expect(mockRedux.dispatch).toHaveBeenCalled();
      expect(mockRedux.dispatch).toHaveBeenCalledWith(expectedAction);
    });

  it('incrementIfOdd should not dispatch INCREMENT_COUNTER action if count is even',
    () => {
      mockState.counter = 2;
      spyOn(mockRedux, 'dispatch');
      actions.incrementIfOdd();

      expect(mockRedux.dispatch).not.toHaveBeenCalled();
    });
});
```

An Async Action

What about async actions like the following?

```
import { Injectable } from '@angular/core';
import { NgRedux } from 'ng2-redux';

export const INCREMENT_COUNTER = 'INCREMENT_COUNTER';
export const DECREMENT_COUNTER = 'DECREMENT_COUNTER';

@Injectable()
export class CounterActions {
  constructor(private redux: NgRedux<any>) {}

  // ...

  incrementAsync(timeInMs = 1000) {
    this.delay(timeInMs).then(() => this.redux.dispatch({ type: INCREMENT_COUNTER }));
  }

  private delay(timeInMs) {
    return new Promise((resolve,reject) => {
      setTimeout(() => resolve() , timeInMs);
    });
  }
}
```

We can test this using the normal techniques for async service functions:

- If we can make `incrementAsync` return a promise, we can just return a promise from the test and `jasmine` will wait until it settles.
- Alternately, we can use the `fakeAsync` technique discussed in the section on testing components.

The thing to remember is that if we follow the ActionCreatorService pattern, our actions are just functions on an Angular service. So we can mock out NgRedux (and any other dependencies) and just test it as we would any other Angular 2 service.

Testing Reducers

Luckily, testing reducers is a lot like testing our synchronous action creators, since all reducer operations are synchronous. This plays a big role in making our global state easy to keep track of, which is why we're big fans of Redux.

We'll test the counter reducer in angular2-redux-starter, as follows:

```
export default function counter(state = 0, action)
  switch (action.type) {
    case INCREMENT_COUNTER:
      return state + 1;
    case DECREMENT_COUNTER:
      return state - 1;
    default:
      return state;
  }
}
```

As you can see, there are three cases to test: the default case, the increment and the decrement. We want to test that our actions trigger the state changes we expect from the reducer.

```
import { INCREMENT_COUNTER, DECREMENT_COUNTER } from '../actions/counter';
import counter from './counter';

describe('counter reducers', () => {
  it('should handle initial state', () => {
    expect(
      counter(undefined, {})
    )
    .toEqual(0)
  });

  it('should handle INCREMENT_COUNTER', () => {
    expect(
      counter(0, {
        type: INCREMENT_COUNTER
      })
    )
    .toEqual(1)
  });

  it('should handle DECREMENT_COUNTER', () => {
    expect(
      counter(1, {
        type: DECREMENT_COUNTER
      })
    )
    .toEqual(0)
  });
});
```

Note that we're only testing the section of Redux state that the `counter` reducer is responsible for, and not the whole. We can see from these tests that Redux is largely built on pure functions.

Afterthoughts

The examples outlined above are just one approach to unit testing in Redux. During actual development it might prove to be too costly to maintain tests for every action and reducer, and in some cases even trivial (i.e. should I be paranoid about this JSON object with one property being returned?).

Another approach we've tried is to treat the overall state change in the store triggered by an action (or by a series of actions) as a single unit - in the Redux world reducers don't function without actions and vice versa, so why separate them? This leaves more flexibility when making changes to actions and reducers without losing scope of what Redux is doing for our app.

Migrating Angular 1.x Projects to Angular 2

Figure: Great Migration by gekkodigitalmedia licensed under Public Domain (https://pixabay.com/en/great-migration-africa-animal-1021460/)

Migration Prep

Before most Angular 1.x applications can be upgraded to Angular 2 there is preparatory work to do. This is especially true for Angular applications using style that predates Angular 1.3.

Upgrading to Angular 1.3+ Style

The first step of any migration is to upgrade the codebases style to conform to Angular 1.3+ style, ideally an Angular 1.5+ style. This means:

- All controllers should be in `controllerAs` form, and ideally should only exist on directives
- Use directives, specifically "component directives", that use the following properties:

  - `restrict: 'E'`
  - `scope: {}`
  - `bindToController: {}`
  - `controllerAs`
  - `template` or `templateUrl`
  - `transclude` (optional)
  - `require` (optional)
- Component directives should *not* use the following attributes:

  - `compile`
  - `replace: true`
  - `priority` / `terminal`
- Ideally have one component, or one *thing* per file
- Ideally have folders organized by feature

Migrating to TypeScript

TypeScript is a superset of ES6 and, as its name suggests, uses a type system. This can have an enormous impact on developer tools, providing richer auto-complete and static analysis.

Angular 2 was built using TypeScript, and supports decorators which provide meta information to Angular. While it is possible to use Angular 2 without these features, the syntax feels more "natural" with TypeScript's decorators.

Using Webpack

Using a module loader like webpack is essential for migrating to Angular 2, and should already be part of every modern programmer's tool set. Webpack will make it easy to manage all the different files that a modern, modular Angular 1.3+ project prescribes. This includes bundling the application for distribution or deployment.

Using webpack will also simplify a programmer's Angular 2 workflow, since the easiest way to work with Angular 2 is with TypeScript, or ES6, neither of which works natively in contemporary browsers.

Choosing an Upgrade Path

There are three ways to upgrade from Angular 1 to 2:

- Total conversion
- ng-upgrade
- ng-forward

Total Conversion

Completely converting an application from Angular 1 to Angular 2 is technically possible, but really only suitable for the smallest applications. Even small applications can be tricky to totally convert if they're not well structured.

Using ng-forward (Angular 1.x Using 2 Style)

The ng-forward approach is done with Angular 1.x dependencies and a few small helper libraries. ng-forward allows developers to use Angular 2 style TypeScript (annotations/decorators) *without* Angular 2. Unfortunately templates are still mostly in Angular 1.x style.

Once an application is converted to ng-forward style it is very close to Angular 2, but still requires refactoring. In most cases, ng-forward is not as efficient as ng-upgrade with respect to refactoring time. The payload of an ng-forwarded application is smaller, and porting to ng-forward can be done in an even more ad-hoc fashion than with ng-upgrade.

The general flow of ng-forwarding an application is:

- install ng-forward dependencies
- bootstrap root component
- upgrade components strategically
- refactor the codebase to Angular 2

Using ng-upgrade (Angular 1.x Co-Existing With Angular 2)

The ng-upgrade is done by running Angular 2 and Angular 1 together in the same application. In this scenario Angular 1.x controls the page, and Angular 2 controls the change detection mechanisms. Once the two Angulars co-exist, upgrading can be done in strategic pieces.

Bootstrapping ng-upgrade

- Use manual Angular 1.x bootstrapping, and remove `ng-app` / `ng-strict-di` references if they exist
- Add Angular 2 dependencies
- Add the upgrade adapter `import {UpgradeAdapter} from '@angular/upgrade'`
- Call the upgrade adapter's bootstrap

Once this is working the foundation is set for transitioning from Angular 1.x to Angular 2. It is important to note that the upgrade adapter's bootstrap mechanism is asynchronous. Additionally it's important to treat the upgrade adapter as a singleton.

The following file creates an instance of `UpgradeAdapter` and exports it.

```
// Angular 2 Vendor Import
import {UpgradeAdapter} from '@angular/upgrade';

// Instantiate an adapter
export const upgradeAdapter = new UpgradeAdapter();
```

The following file bootstraps an Angular 1/2 hybrid application:

```
// Angular 1 Vendor Import
import * as angular from 'angular';

// Import the upgradeAdapter singleton
import {upgradeAdapter} from './upgrade-adapter';



// Name the application
const APPNAME = 'angular-upgrade-example';

// Register classic Angular 1 modules
angular
  .module(APPNAME, []);

// Bootstrap Angular 1 manually
angular.bootstrap(document.body, [APPNAME]);

// Bootstrap Angular 2 - *note* this is asynchronous
upgradeAdapter.bootstrap(document.documentElement, [APPNAME], {strictDi: true});
```

The above example does not actually do anything other than bootstrap an empty application.

Upgrading/Downgrading Components

Once bootstrapping is complete, Angular 1.x components can be *upgraded* to work with Angular 2. Conversely, Angular 2 components can be *downgraded* to work with Angular 1.x.

Downgrading Components

Upgrading components sounds like it should happen before downgrading, but the point of upgrading is to make an Angular 1.x component work with Angular 2. For an Angular 2 component to use an Angular 1.x component in an ng-upgrade application there must first be a downgraded Angular 2 component. Consequently it's important to first learn how to downgrade Angular 2 components to work with Angular 1.x

All downgraded components operate like Angular 1.x `'E'` element directives.

Here is an example of a very simple Angular 2 component:

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

@Component({
  selector: 'a2-downgrade',
  template: '<p>{{ message }}</p>'
})
export class A2DowngradeComponent {
  message = `What you're seeing here is an Angular2 component ` +
    `running in an Angular1 app!`;
}
```

Registering the downgraded component with Angular 1.x:

```
// Angular 1 Vendor Import
import * as angular from 'angular';

// Angular 2 component from above
import {A2DowngradeComponent} from './components/a2-downgrade';

// Register classic Angular 1 modules
angular
  .module(APPNAME)
  .directive('a2Downgrade',
    upgradeAdapter.downgradeNg2Component(A2DowngradeComponent));
```

Upgrading Components

The only Angular 1.x components that can be upgraded and used in Angular 2 code are those that *strictly* follow the component pattern outlined at the top of this document. Wherever possible use Angular 1.5+'s `.component` .

Here is an Angular 1.x directive that conforms to ng-upgrade's "component directive" specification:

```
export function a1UpgradableDirective() {
  return {
    restrict: 'E',
    scope: {},
    bindToController: {},
    controller: Upgradable,
    controllerAs: 'a1Upgradable',
    template: `<span>{{ a1Upgradable.message }}</span>`
  };
}


class Upgradable {
  message = 'I am an Angular 1 Directive';
}
```

Here is an Angular 2 component that will use the upgraded Angular 1.x directive:

```
import {Component} from '@angular/core';
import {upgradeAdapter} from '../upgrade-adapter';

@Component({
  selector: 'a2-using-a1',
  directives: [upgradeAdapter.upgradeNg1Component('a1Upgradable')],
  template: `<p>{{ message }}<a1-upgradable></a1-upgradable></p>`
})
export class A2UsingA1Component {
  message = 'Angular 2 Using Angular 1: ';
}
```

Finally, let Angular 1.x know about the directive:

```typescript
import {a1UpgradableDirective} from './components/a1-upgradable';


// Angular 1 Vendor Import
import * as angular from 'angular';


// Register classic Angular 1 modules
angular
  .module(APPNAME)
  .directive('a1Upgradable', a1UpgradableDirective)
```

```typescript
import {a1UpgradableDirective} from './components/a1-upgradable';


// Angular 1 Vendor Import
import * as angular from 'angular';
```

Projecting Angular 1 Content into Angular 2 Components

In Angular 2 the concept of "transclusion" has been replaced by the concept of projection. ng-upgrade provides mechanisms for projecting/transcluding Angular 1.x content into Angular 2 components:

This is what a simple Angular 2 component that supports projection looks like:

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

@Component({
  selector: 'a2-projection',
  template: `
  <p>
    Angular 2 Outer Component (Top)
    <ng-content></ng-content>
    Angular 2 Outer Component (Bottom)
  </p>
  `
})
export class A2Projection { }
```

Here's a very simple Angular 1.x directive that will be projected into the Angular 2 component:

```
export function a1ProjectionContentsDirective() {
  return {
    restrict: 'E',
    scope: {},
    bindToController: {},
    controller: A1ProjectionContents,
    controllerAs: 'a1ProjectionContents',
    template: `<p>{{ a1ProjectionContents.message }}</p>`
  };
}

class A1ProjectionContents {
  message = 'I am an Angular 1 Directive "projected" into Angular 2';
}
```

Both the component and the directive must be registered with Angular 1.x:

```
import {A2Projection} from './components/a2-projection';
import {a1ProjectionContentsDirective} from
  './components/a1-projection-contents';

// Angular 1 Vendor Import
import * as angular from 'angular';

// Import the upgradeAdapter singleton
import {upgradeAdapter} from './upgrade-adapter';

// Name the application
const APPNAME = 'angular-upgrade-example';

// Register classic Angular 1 modules
angular
  .module(APPNAME)
  .directive('a2Projection',
    upgradeAdapter.downgradeNg2Component(A2Projection))
  .directive('a1ProjectionContent', a1ProjectionContentsDirective);
```

Finally, using the HTML selectors is as simple as:

```
<a2-projection>
  <a1-projection-content></a1-projection-content>
</a2-projection>
```

Transcluding Angular 2 Components into Angular 1 Directives

Angular 2 components can be transcluded into Angular 1.x directives.

Here is a very simple Angular 2 component:

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

@Component ({
  selector: 'a2-transclusion-contents',
  template: `<p>{{ message }}</p>`
})
export class A2Transclusion {
  message =
    'I am an Angular 2 Component "transcluded" into Angular 1.x';
}
```

Here is an Angular 1.x directive that supports transclusion:

```
export function a1TransclusionDirective() {
  return {
    restrict: 'E',
    transclude: true,
    scope: {},
    bindToController: {},
    controller: A1Transclusion,
    controllerAs: 'a1ProjectionContents',
    template: `
    <p>
      <ng-transclude></ng-transclude>
    </p>
    `
  };
}

class A1Transclusion {
}
```

Angular 1.x needs to know about both the component and the directive:

```
import {A2Transclusion} from './components/a2-transclusion-contents';
import {a1TransclusionDirective} from './components/a1-transclusion';

// Angular 1 Vendor Import
import * as angular from 'angular';

// Import the upgradeAdapter singleton
import {upgradeAdapter} from './upgrade-adapter';

// Name the application
const APPNAME = 'angular-upgrade-example';

// Register classic Angular 1 modules
angular
  .module(APPNAME)
  .directive('a2TransclusionContents',
    upgradeAdapter.downgradeNg2Component(A2Transclusion))
  .directive('a1Transclusion', a1TransclusionDirective);
```

Finally, Angular 2 content can be transcluded into Angular 1.x like so:

```
<a1-transclude>
  <a2-transclusion-contents></a2-transclusion-contents>
</a1-transclude>
```

Injecting Across Frameworks

Angular 1.x providers/services can be upgraded and injected into Angular 2.

Simple Angular 1.x service:

```
export class A1UpgradeService {
  data = 'Hello from Angular 1 service';
}
```

Simple Angular 2 component that will have an Angular 1.x service injected into it:

```
import {Component, Inject} from  '@angular/core';
import {A1UpgradeService} from '../services/a1-upgrade-service';

@Component({
  selector: 'a2-using-a1-service',
  template: `<p>{{ message }}</p>`
})
export class A2UsingA1Service {
  message = '';
  constructor(@Inject('a1UpgradeService') a1UpgradeService:A1UpgradeService) {
    this.message = a1UpgradeService.data;
  }
}
```

Attaching everything to Angular 1.x:

```
import {A2UsingA1Service} from './components/a2-using-a1-service';
import {A1UpgradeService} from './services/a1-upgrade-service';

// Angular 1 Vendor Import
import * as angular from 'angular';

// Import the upgradeAdapter singleton
import {upgradeAdapter} from './upgrade-adapter';

// Name the application
const APPNAME = 'angular-upgrade-example';

// Register classic Angular 1 modules
angular
  .module(APPNAME)
  .directive('a2UsingA1Service',
    upgradeAdapter.downgradeNg2Component(A2UsingA1Service))
  .service('a1UpgradeService', A1UpgradeService);
```

Angular 2.x services can be downgraded and injected into Angular 1. In normal operation, Angular 2.x services would be bootstrapped with the application, but because of ng-upgrade being a hybrid mode, this is not the case. The upgrade adapter comes with an `addProvider` method that must be used in the interim.

Here is a very simple Angular 2 service:

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

@Injectable()
export class A2DowngradeService {
  fetchData() {
    return 'some data';
  }
}
```

Since Angular 2 is bootstrapped with the upgrade adapter, there is no place to register Angular 2 services. Fortunately the upgrade adapter's `addProvider` method can do this:

```
upgradeAdapter.addProvider(Phones);
```

Lastly, Angular 1.x must be informed about the Angular 2 service:

```
// The service to downgrade
import {A2DowngradeService} from './services/a2-downgrade'

// Angular 1 Vendor Import
import * as angular from 'angular';

// Import the upgradeAdapter singleton
import {upgradeAdapter} from './upgrade-adapter';

// Name the application
const APPNAME = 'angular-upgrade-example';

// Register classic Angular 1 modules
angular
  .module(APPNAME)
  .factory('a2DowngradeService',
    upgradeAdapter.downgradeNg2Provider(A2DowngradeService));
```

Using this downgraded service in an Angular 1.x directive is as simple as:

```
import {A2DowngradeService} from '../services/a2-downgrade';

export function a1UsingA2ServiceDirective() {
  return {
    restrict: 'E',
    scope: {},
    bindToController: {},
    controller: A1UsingA2,
    controllerAs: 'a1UsingA2',
    template: `<span>{{ a1UsingA2.message }}</span>`
  };
}

class A1UsingA2 {
  message: string;
  constructor(private a2DowngradeService: A2DowngradeService) {
    this.message = this.a2DowngradeService.fetchData();
  }
}
```

Upgrading Components Strategically

Services that have no dependencies are excellent candidates for conversion. Once converted to Angular 2, services can be downgraded to work in Angular 1.x. Components can follow a similar strategy, with "leaf" components being converted before "root" components.

Possibly the most challenging component/service to upgrade is Angular 1.x's UI Router library. This process *might* be simplified in the future, but for the moment it's best to upgrade UI Router last.

Project Setup

Proper tooling and setup is good for any project, but it's especially important for Angular 2 due to all of the pieces that are involved. We've decided to use webpack, a powerful tool that attempts to handle our complex integrations. Due to the number of parts of our project that webpack touches, it's important to go over the configuration to get a good understanding of what gets generated client-side.

Webpack

A modern JavaScript web application includes a lot of different packages and dependencies, and it's important to have something that makes sense of it all in a simple way.

Angular 2 takes the approach of breaking your application apart into many different components, each of which can have several files. Separating application logic this way is good for the programmer, but can detract from user experience since doing this can increase page loading time. HTTP2 aims to solve this problem in one way, but until more is known about its effects we will want to bundle different parts of our application together and compress it.

Our platform, the browser, must continue to provide backwards compatibility for all existing code and this necessitates slow movement of additions to the base functionality of HTML/CSS/JS. The community has created different tools that transform their preferred syntax/feature set to what the browser supports to avoid binding themselves to the constraints of the web platform. This is especially evident in Angular 2 applications, where TypeScript is used heavily. Although we don't do this in our course, projects may also involve different CSS preprocessors (sass, stylus) or templating engines (jade, Mustache, EJS) that must be integrated.

Webpack solves these problems by providing a common interface to integrate all of these tools and that allows us to streamline our workflow and avoid complexity.

Installation

The easiest way to include webpack and its plugins is through NPM and save it to your `devDependencies` :

```
npm install -D webpack ts-loader html-webpack-plugin tslint-loader
```

Setup and Usage

The most common way to use webpack is through the CLI. By default, running the command executes `webpack.config.js` which is the configuration file for your webpack setup.

Bundle

The core concept of webpack is the *bundle*. A bundle is simply a collection of modules, where we define the boundaries for how they are separated. In this project, we have two bundles:

- `app` for our application-specific client-side logic
- `vendor` for third party libraries

In webpack, bundles are configured through *entry points*. Webpack goes through each entry point one by one. It maps out a dependency graph by going through each module's references. All the dependencies that it encounters are then packaged into that bundle.

Packages installed through NPM are referenced using *CommonJS* module resolution. In a JavaScript file, this would look like:

```
const app = require('./src/index.ts');
```

or TypeScript/ES6 file:

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

We will use those string values as the module names we pass to webpack.

Let's look at the entry points we have defined in our sample app:

```
{
  ...
  entry: {
    app: './src/index.ts',
    vendor: [
      'es6-shim',
      'angular2/bundles/angular2-polyfills',
      'angular2/bootstrap',
      'angular2/platform/browser',
      'angular2/platform/common_dom',
      '@angular/core',
      'angular2/router',
      'angular2/http',
      'redux',
      'redux-thunk',
      'ng2-redux',
      'redux-logger'
    ]
  }
  ...
}
```

The entry point for `app`, `./src/index.ts`, is the base file of our Angular 2 application. If we've defined the dependencies of each module correctly, those references should connect all the parts of our application from here. The entry point for `vendor` is a list of modules that we need for our application code to work correctly. Even if these files are referenced by some module in our app bundle, we want to separate these resources in a bundle just for third party code.

Output Configuration

In most cases we don't just want to configure how webpack generates bundles - we also want to configure how those bundles are output.

- Often, we will want to re-route where files are saved. For example into a `bin` or `dist` folder. This is because we want to optimize our builds for production.

- Webpack transforms the code when bundling our modules and outputting them. We want to have a way of connecting the code that's been generated by webpack and the code that we've written.

- Server routes can be configured in many different ways. We probably want some way of configuring webpack to take our server routing setup into consideration.

All of these configuration options are handled by the config's `output` property. Let's look at how we've set up our config to address these issues:

```
{
  ...
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].[hash].js',
    publicPath: "/",
    sourceMapFilename: '[name].[hash].js.map'
  }
  ...
}
```

> Some options have words wrapped in square brackets. Webpack has the ability to parse parameters for these properties, with each property having a different set of parameters available for substitution. Here, we're using `name` (the name of the bundle) and `hash` (a hash value of the bundle's content).

To save bundled files in a different folder, we use the `path` property. Here, `path` tells webpack that all of the output files must be saved to `path.resolve(__dirname, 'dist')`. In our case, we save each bundle into a separate file. The name of this file is specified by the `filename` property.

Linking these bundled files and the files we've actually coded is done using what's known as source maps. There are different ways to configure source maps. What we want is to save these source maps in a separate file specified by the `sourceMapFilename` property. The way the server accesses the files might not directly follow the filesystem tree. For us, we want to use the files saved under *dist* as the root folder for our server. To let webpack know this, we've set the `publicPath` property to `/`.

Loaders

TypeScript isn't core JavaScript so webpack needs a bit of extra help to parse the `.ts` files. It does this through the use of *loaders*. Loaders are a way of configuring how webpack transforms the outputs of specific files in our bundles. Our `ts-loader` package is handling this transformation for TypeScript files.

Inline

Loaders can be configured – inline – when requiring/importing a module:

```
const app = require('ts!./src/index.ts');
```

The loader is specified by using the `!` character to separate the module reference and the loader that it will be run through. More than one loader can be used and those are separated with `!` in the same way. Loaders are executed right to left.

```
const app = require('ts!tslint!./src/index.ts');
```

> Although the packages are named `ts-loader`, `tslint-loader`, `style-loader`, we don't need to include the `-loader` part in our config.

Be careful when configuring loaders this way – it couples implementation details of different stages of your application together so it might not be the right choice in a lot of cases.

Webpack Config

The preferred method is to configure loaders through the `webpack.config.js` file. For example, the TypeScript loader task will look something like this:

```
{
  test: /\.ts$/,
  loader: 'ts-loader',
  exclude: /node_modules/
}
```

This runs the typescript compiler which respects our configuration settings as specified above. We want to be able to handle other files and not just TypeScript files, so we need to specify a list of loaders. This is done by creating an array of tasks.

Tasks specified in this array are chained. If a file matches multiple conditions, it will be processed using each task in order.

```
{
  ...
  module: {
    preLoaders: [{
      test: /\.ts$/,
      loader: 'tslint'
    }],
    loaders: [
      { test: /\.ts$/, loader: 'ts', exclude: /node_modules/ },
      { test: /\.html$/, loader: 'raw' },
      { test: /\.css$/, loader: 'style!css?sourceMap' },
      { test: /\.svg/, loader: 'url' },
      { test: /\.eot/, loader: 'url' },
      { test: /\.woff/, loader: 'url' },
      { test: /\.woff2/, loader: 'url' },
      { test: /\.ttf/, loader: 'url' },
    ],
    noParse: [ /zone\.js\/dist\/.+/, /angular2\/bundles\/.+/ ]
  }
  ...
}
```

Each task has a few configuration options:

- *test* - The file path must match this condition to be handled. This is commonly used to test file extensions eg. `/\.ts$/` .

- *loader* - The loaders that will be used to transform the input. This follows the syntax specified above.

- *exclude* - The file path must not match this condition to be handled. This is commonly used to exclude file folders, e.g. `/node_modules/` .

- *include* - The file path must match this condition to be handled. This is commonly used to include file folders. eg. `path.resolve(__dirname, 'app/src')` .

Pre-Loaders

The preLoaders array works just like the loaders array only it is a separate task chain that is executed before the loaders task chain.

Non JavaScript Assets

Webpack also allows us to load non JavaScript assets such as: CSS, SVG, font files, etc. In order to attach these assets to our bundle we must require/import them within our app modules. For example:

```
import './styles/style.css';

// or

const STYLES = require('./styles/style.css');
```

Other Commonly Used Loaders

- *raw-loader* - returns the file content as a string.

- *url-loader* - returns a base64 encoded data URL if the file size is under a certain threshold, otherwise it just returns the file.

- *css-loader* - resolves `@import` and `url` references in CSS files as modules.

- *style-loader* - injects a style tag with the bundled CSS in the `<head>` tag.

Plugins

Plugins allow us to inject custom build steps during the bundling process.

A commonly used plugin is the `html-webpack-plugin` . This allows us to generate HTML files required for production. For example it can be used to inject script tags for the output bundles.

```
new HtmlWebpackPlugin({
  template: './src/index.html',
  inject: 'body',
  minify: false
});
```

Summary

When we put everything together, our complete `webpack.config.js` file looks something like this:

```javascript
'use strict';

const path = require("path");
const webpack = require('webpack');
const HtmlWebpackPlugin = require('html-webpack-plugin');

const basePlugins = [
  new webpack.optimize.CommonsChunkPlugin('vendor', '[name].[hash].bundle.js'),

  new HtmlWebpackPlugin({
    template: './src/index.html',
    inject: 'body',
    minify: false
  })
];

const envPlugins = {
    production: [
        new webpack.optimize.UglifyJsPlugin({
            compress: {
                warnings: false
            }
        })
    ],
    development: []
};

const plugins = basePlugins.concat(envPlugins[process.env.NODE_ENV] || []);

module.exports = {
  entry: {
    app: './src/index.ts',
    vendor: [
      'es6-shim',
      'angular2/bundles/angular2-polyfills',
      'angular2/bootstrap',
      'angular2/platform/browser',
      'angular2/platform/common_dom',
      '@angular/core',
      'angular2/router',
      'angular2/http',
      'redux',
      'redux-thunk',
      'ng2-redux',
```

```
        'redux-logger'
    ]
  },

  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].[hash].js',
    publicPath: "/",
    sourceMapFilename: '[name].[hash].js.map'
  },

  devtool: 'source-map',

  resolve: {
    extensions: ['', '.webpack.js', '.web.js', '.ts', '.js']
  },

  plugins: plugins,

  module: {
    preLoaders: [{
      test: /\.ts$/,
      loader: 'tslint'
    }],
    loaders: [
      { test: /\.ts$/, loader: 'ts', exclude: /node_modules/ },
      { test: /\.html$/, loader: 'raw' },
      { test: /\.css$/, loader: 'style!css?sourceMap' },
      { test: /\.svg/, loader: 'url' },
      { test: /\.eot/, loader: 'url' },
      { test: /\.woff/, loader: 'url' },
      { test: /\.woff2/, loader: 'url' },
      { test: /\.ttf/, loader: 'url' },
    ],
    noParse: [ /zone\.js\/dist\/.+/, /angular2\/bundles\/.+/ ]
  }
}
```

Going Further

Webpack also does things like hot code reloading and code optimization which we haven't covered. For more information you can check out the official documentation. The source is also available on Github.

NPM Scripts Integration

NPM allows us to define custom scripts in the `package.json` file. These can then execute tasks using the NPM CLI.

We rely on these scripts to manage most of our project tasks and webpack fits in as well.

The scripts are defined in the `scripts` property of the `package.json` file. For example:

```
...
  scripts: {
    "clean": "rimraf dist",
    "prebuild": "npm run clean",
    "build": "NODE_ENV=production webpack",
  }
...
```

NPM allows pre and post task binding by prepending the word `pre` or `post` respectively to the task name. Here, our `prebuild` task is executed before our `build` task.

> We can run an NPM script from inside another NPM script.

To invoke the `build` script we run the command `npm run build` :

1. The `prebuild` task executes.
2. The `prebuild` task runs the `clean` task, which executes the `rimraf dist` command.
3. `rimraf` (an NPM package) recursively deletes everything inside a specified folder.
4. The `build` task is executed. This sets the `NODE_ENV` environment variable to `production` and starts the webpack bundling process.
5. Webpack generates bundles based on the `webpack.config.js` available in the project root folder.

Angular CLI

With all of the new features Angular 2 takes advantage of, like static typing, decorators and ES6 module resolution, comes the added cost of setup and maintenance. Spending a lot of time with different build setups and configuring all of the different tools used to serve a modern JavaScript application can really take a lot of time and drain productivity by not being able to actually work on the app itself.

Seeing the popularity of ember-cli, Angular 2 decided they would provide their own CLI to solve this problem. Angular CLI is geared to be the tool used to create and manage your Angular 2 app. It provides the ability to:

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

> The Angular 2 CLI currently only generates scaffolding in TypeScript, with other dialects to come later.

Setup

Prerequisites

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

Installation

The Angular 2 CLI can be installed with the following command:

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

Creating a New App

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

File and Folder Setup
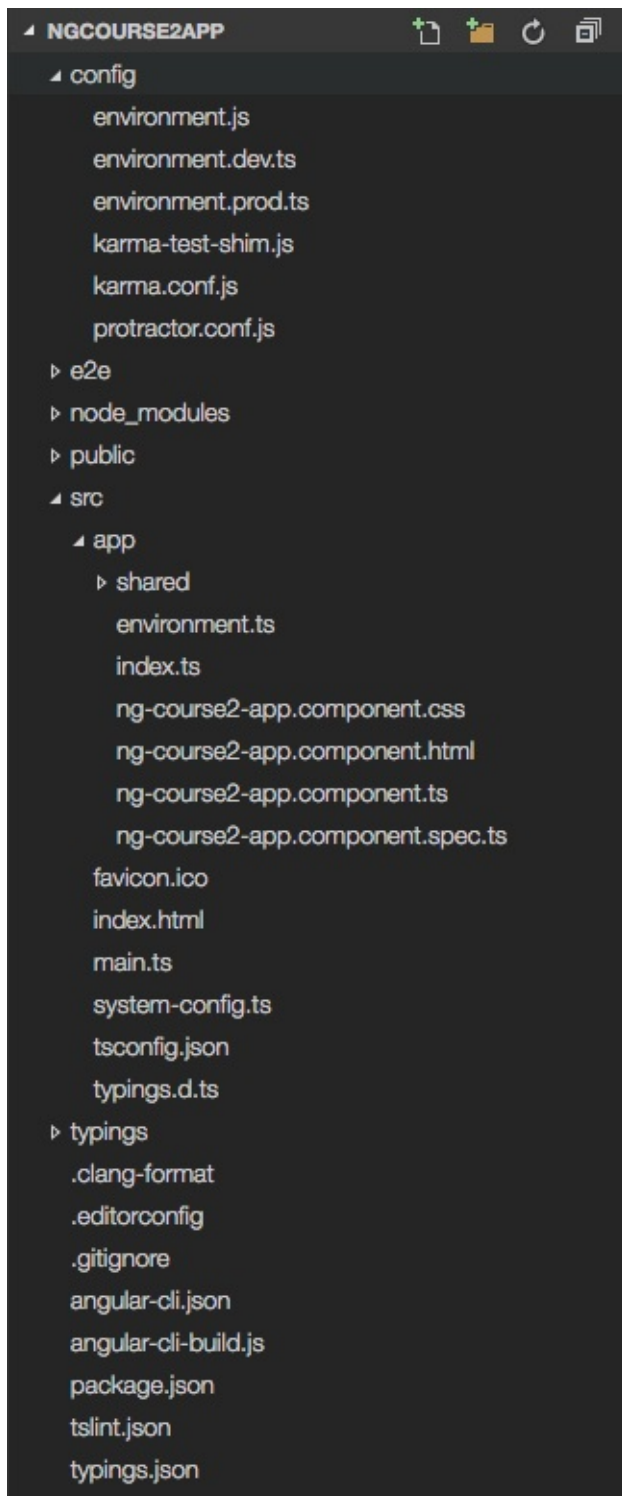
The generated app folder will look like this:

Figure: App folder

Application configuration is stored in different places, some located in the *config* folder, such as test configuration, and some being stored in the project root such as linting information and build information. The CLI stores application-specific files in the *src* folder and Angular 2-specific code in the *src/app* folder. Files and folders generated by the CLI will follow the official style guide.

> Warning: The CLI relies on some of the settings defined in the configuration files to be able to execute the commands. Take care when modifying them, particularly the package.json file.

The CLI has installed everything a basic Angular 2 application needs to run properly. To make sure everything has run and installed correctly we can run the server.

Serving the App

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

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

Creating Components

The CLI can scaffold Angular 2 components through the `generate` command. To create a new component run:

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

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

- `[component-name].component.ts` the component class file
- `[component-name].component.css` for styling the component
- `[component-name].component.html` component html
- `[component-name].component.spec.ts` tests for the component
- `index.ts` which exports the component

Creating Routes

The `ng g route [route-name]` command will spin up a new folder and route files for you. It will also insert the route into the parent component's `@Routes` decorator. For example, `ng g route myRoute` adds a route to the main component and creates the following files:

- +my-route/
  - shared/
  - index.ts
  - my-route.component.css
  - my-route.component.html
  - my-route.component.ts
  - my-route.component.spec.ts

It also adds the path to the root component's `@Route` decorator:

```
@Routes([
  {path: '/my-route', component: MyRouteComponent}
])
```

By default, routes are created as lazy routes (and indicated with a `+` in the folder name), meaning they will only be loaded into the browser when needed. You can turned this off using the flag `--lazy false`.

Creating Other Things

The CLI can scaffold other Angular 2 entities such as services, pipes and directives using the generate command.

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

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

Item	Command	Files generated
Component:	`ng g component [name]`	component, HTML, CSS, test spec files
Directive:	`ng g directive [name]`	component, test spec files
Pipe:	`ng g pipe [name]`	component, test spec files
Service:	`ng g service [name]`	component, test spec files
Class:	`ng g class [name]`	component, test spec files
Route:	`ng g route [name]`	component, HTML, CSS, test spec files (in new folder)

Testing

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

Unit Tests

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

End-to-End Tests

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

Linting

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

CLI Command Overview

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

Linting

`ng lint` lints the code in your project using [tslint](). You can customize the rules for your project by editing `tslint.json`.

> You can switch some of these to use your preferred tool by editing the scripts in `package.json`.

Testing

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

Build

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

Serve

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

> Include `--port [number]` to serve your app on a different HTTP port.

E2E

Once your app is served, you can run end-to-end tests using `ng e2e`. The CLI uses [Protractor]() for these tests.

Deploy

`ng deploy` deploys to GitHub pages or Firebase.

Adding Third Party Libraries

The CLI provides the ability to integrate third party libraries into the application. For step-by-step information, visit https://github.com/angular/angular-cli/wiki/3rd-party-libs.

Integrating an Existing App

Apps that were created without CLI can be integrated to use CLI later on. This is done by going to the existing app's folder and running `ng init` .

Since the folder structure for an existing app might not follow the same format as one created by the CLI, the `init` command has some configuration options.

- `--source-dir` identifies the relative path to the source files (default = src)
- `--prefix` identifies the path within the source dir that Angular 2 application files reside (default = app)
- `--style` identifies the path where additional style files are located (default = css).

Other Resources

- Generative Art with SVG And Angular 2
- Making your own Canvas Renderer with Angular 2
  - Github Repo
- RxJs - Which Operator do I use?
- Rx Marbels
- The Introduction to Reactive Programming you've been missing