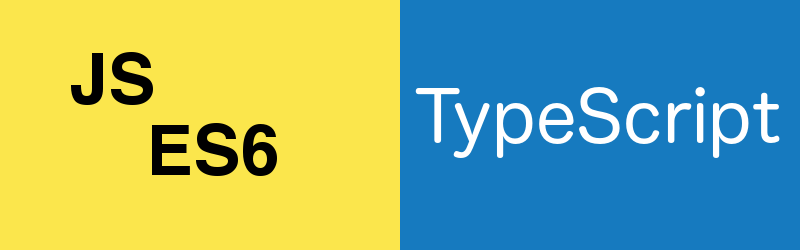
**EcmaScript 6 and TypeScript Features**

Figure: ES6 and TypeScript

The language we usually call "JavaScript" is formally known as "EcmaScript". The new version of JavaScript, known as "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. Angular 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 operators
* 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. In Javascript this refers to the calling context and therefore can change to be something other than the object.

## Object

An object is an instance of a class which is created using the new operator. When using a dot notation to access a method on the object, this will refer to the object to the left of the dot.

let burger = new Hamburger();

burger.listToppings();

In the snippet above, whenever this is used from inside class Hamburger, it will refer to object burger.

## Changing Caller Context

JavaScript code can optionally supply this to a method at call time using one of the following.

* Function.prototype.call(object [,arg, ...])
* Function.prototype.bind(object [,arg, ...])
* Function.prototype.apply(object [,argsArray])

# 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:
2. someObject.someMethod();

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

1. Function invocation:
2. 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 is 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, thiswould be undefined, which is slightly less confusing.

[View Example](http://jsbin.com/vekawimihe/2/edit?js,console)

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); // `this` will be undefined

});

}

}

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

myToppings.outputList();

Let's try this code on [http://jsbin.com](http://jsbin.com/qakigoqulo/edit?js,console). 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 myToppings = new Toppings(['cheese', 'lettuce']);

myToppings.outputList();

Let's try this code on [http://jsbin.com](http://jsbin.com/tulikutife/edit?js,console). 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:

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 string interpolation for inserting variables into strings:

var name = 'Sam';

var age = 42;

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

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.

To illustrate this, let's image we have a zoo application where types of birds are created. In classical inheritance, we define a base class and then subclass it to create a derived class.

## Subclassing

The example code below shows how to derive Penguin from Bird using the **extends** keyword. Also pay attention to the **super** keyword used in the subclass constructor of Penguin, it is used to pass the argument to the base class Bird's constructor.

The Bird class defines the method walk which is inherited by the Penguin class and is available for use by instance of Penguin objects. Likewise the Penguin class defines the method swim which is not avilable to Bird objects. Inheritance works top-down from base class to its subclass.

## Object Initialization

The class constructor is called when an object is created using the **new** operator, it will be called before the object is fully created. A consturctor is used to pass in arguments to initialize the newly created object.

The order of object creation starts from its base class and then moves down to any subclass(es).

// Base Class : ES6

class Bird {

constructor(weight, height) {

this.weight = weight;

this.height = height;

}

walk() {

console.log('walk!');

}

}

// Subclass

class Penguin extends Bird {

constructor(weight, height) {

super(weight, height);

}

swim() {

console.log('swim!');

}

}

// Penguin object

let penguin = new Penguin(...);

penguin.walk(); //walk!

penguin.swim(); //swim!

Below we show how prototypal inheritance was done before class was introduced to JavaScript.

// JavaScript classical inheritance.

// Bird constructor

function Bird(weight, height) {

this.weight = weight;

this.height = height;

}

// Add method to Bird prototype.

Bird.prototype.walk = function() {

console.log("walk!");

};

// Penguin constructor.

function Penguin(weight, height) {

Bird.call(this, weight, height);

}

// Prototypal inheritance (Penguin is-a Bird).

Penguin.prototype = Object.create( Bird.prototype );

Penguin.prototype.constructor = Penguin;

// Add method to Penguin prototype.

Penguin.prototype.swim = function() {

console.log("swim!");

};

// Create a Penguin object.

let penguin = new Penguin(50,10);

// Calls method on Bird, since it's not defined by Penguin.

penguin.walk(); // walk!

// Calls method on Penguin.

penguin.swim(); // swim!

# Delegation

In the inheritance section we looked at one way to extend a class functionality, there is second way using delegation to extend functionality. With delegation, one object will contain a reference to a different object that it will hand off a request to perform the functionality.

The code below shows how to use delegation with the Bird class and Penguin class. The Penguinclass has a reference to the Bird class and it delegates the call made to it's walk method over to Bird's walk method.

// ES6

class Bird {

constructor(weight, height) {

this.weight = weight;

this.height = height;

}

walk() {

console.log('walk!');

}

}

class Penguin {

constructor(bird) {

this.bird = bird;

}

walk() {

this.bird.walk();

}

swim() {

console.log('swim!');

}

}

const bird = new Bird(...);

const penguin = new Penguin(bird);

penguin.walk(); //walk!

penguin.swim(); //swim!

A good discussion on 'behaviour delegation' can be found [here](https://github.com/getify/You-Dont-Know-JS/blob/master/this%20%26%20object%20prototypes/ch6.md).

# 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, vars are scoped to functions, 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.

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;

However there are two cases where **const** does not work as you think it should.

1. A const object literal.
2. A const reference to an object.

## Const Object Literal

const person = {

name: 'Tammy'

};

person.name = 'Pushpa'; // OK, name property changed.

person = null; // "TypeError: Assignment to constant variable.

The example above demonstrates that we are able to change the **name** property of object person, but we are unable to reset the reference **person** since it has been marked as const.

## Const Reference To An Object

Something similar to the above code is using a const reference, below we've switch to using **let** for the literal object.

let person = {

name: 'Tammy'

};

const p = person;

p.name = 'Pushpa'; // OK, name property changed.

p = null; // "TypeError: Assignment to constant variable.

Take away, marking an object reference **const** does not make properties inside the object const.

# Spread Syntax (Spread Element) and Rest parameters

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

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

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

## Spread Syntax

Spread 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 - arrays 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 parameter

Rest parameters share the ellipsis like syntax of spread syntax but are used for a different purpose. Rest parameters are used to access indefinite 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 parameters are in fact arrays which provides access to methods like map, filter, reduce and more. The other important difference is that rest parameters only include arguments not specifically named in a function like so:

function print(a, b, c, ...more) {

console.log(more[0]);

console.log(arguments[0]);

}

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

// 4

// 1

Note: Commonly spread syntax and rest parameters are referenced as Spread and Rest operators but they aren't operators according to ECMAScript specifications. Few references[*MDN-Spread Syntax*](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator),[*MDN-Rest Parameters*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters),[*ECMAScript Spec - Spread Syntax*](http://www.ecma-international.org/ecma-262/6.0/#sec-array-initializer),[*ECMAScript Spec - Rest Parameters*](http://www.ecma-international.org/ecma-262/6.0/#sec-function-definitions)

# Destructuring

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

To [borrow from the MDN](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment), 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](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) has some great examples, including nested object destructuring and dynamic destructuring with for ... inoperators".

# ES6 Modules

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

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

// File: circle.js

export const pi = 3.141592;

export const circumference = diameter => diameter \* pi;

The code above uses the Arrow function for circumference. Read more about arrow functions [here](https://angular-2-training-book.rangle.io/handout/features/arrow_functions.html)

## Module Systems

Using a module on the backend(server side) is relatively straightforward, you simply make use of the **import** keyword. However Web Browsers have no concept of modules or import, they just know how to load javascript code. We need a way to bring in a javascript module to start using it from other javascript code. This is where a module loader comes in.

We won't get into the various module systems out there, but it's worth understanding there are various module loaders available. The popular choices out there are:

* RequireJS
* SystemJS
* Webpack

## Loading a Module From a Browser

Below we make use of SystemJS to load a module. The script first loads the code for the SystemJS library, then the function call **System.import** is use to import(load) the app module.

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:

<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>

# ypeScript

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);

Before TypeScript@1.8.2:

$ 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

Since TypeScript@1.8.2, tsc has a default rule for --module option: target === 'ES6' ? 'ES6' : 'commonjs' (more details can be found [here](https://www.typescriptlang.org/docs/handbook/compiler-options.html)), so we can simply run:

$ tsc ./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 projects there are a number of specific settings that need to be configured in a project's tsconfig.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](http://rbuckton.github.io/ReflectDecorators/typescript.html) but since Angular 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

//...

rules: [

{ 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. (Deprecated in TypeScript 2.0)

In TypeScript 2.0, users can get type files directly from @types through npm (for example, npm install --save @types/lodash will install lodash type file).

# 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:

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:

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 classes as their own type:

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, classes sometimes have optional members. The same ?: syntax can be used on a class definition:

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

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

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

Here is an example of an interface describing a function API:

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 can accept multiple types as well as varying arguments, that is, they can have different call signatures. Interfaces can be used to specify this.

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 classes and interfaces.

Consider an expansion of the previous example:

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.

# Type Keyword

The type keyword defines an alias to a type.

type str = string;

let cheese: str = 'gorgonzola';

let cake: str = 10; // Type 'number' is not assignable to type 'string'

At first glance, this does not appear to be very useful (even the error mentions the original type), but as type annotations become more complex, the benefits of the type keyword become apparent.

### Union Types

Union types allow type annotations to specify that a property should be one of a set of types (either/or).

function admitAge (age: number|string): string {

return `I am ${age}, alright?!`;

}

admitAge(30); // 'I am 30, alright?!'

admitAge('Forty'); // 'I am Forty, alright?!'

The type keyword simplifies annotating and reusing union types.

type Age = number | string;

function admitAge (age: Age): string {

return `I am ${age}, alright?!`;

}

let myAge: Age = 50;

let yourAge: Age = 'One Hundred';

admitAge(yourAge); // 'I am One Hundred, alright?!'

A union type of string literal types is a very useful pattern, creating what is basically an enum with string values.

type PartyZone = "pizza hut" | "waterpark" | "bowling alley" | "abandoned warehouse";

function goToParty (place: PartyZone): string {

return `lets go to the ${place}`;

}

goToParty("pizza hut");

goToParty("chuck e. cheese"); // Argument of type `"chuck e. cheese"' is not assignable to parameter of type 'PartyZone'

### Intersection Types

Intersection types are the combination of two or more types. Useful for objects and params that need to implement more than one interface.

interface Kicker {

kick(speed: number): number;

}

interface Puncher {

punch(power: number): number;

}

// assign intersection type definition to alias KickPuncher

type KickPuncher = Kicker & Puncher;

function attack (warrior: KickPuncher) {

warrior.kick(102);

warrior.punch(412);

warrior.judoChop(); // Property 'judoChop' does not exist on type 'KickPuncher'

}

### Function Type Definitions

Function type annotations can get much more specific than typescripts built-in Function type. Function type definitions allow you to attach a function signature to it's own type.

type MaybeError = Error | null;

type Callback = (err: MaybeError, response: Object) => void;

function sendRequest (cb: Callback): void {

if (cb) {

cb(null, {});

}

}

The syntax is similar to ES6 fat-arrow functions. ([params]) => [return type].

To illustrate the how much the type keyword improved the readability of the previous snippet, here is the function type defined inline.

function sendRequest (cb: (err: Error|null, response: Object) => void): void {

if (cb) {

cb(null, {});

}

}

# Decorators

Decorators are proposed for a future version of JavaScript, but the Angular 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 will notice that these signatures do not look like the signatures used by Angular 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.

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

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.

# 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 can now imagine how Angular 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](http://git-scm.com/)

A source control, sometimes called a version control brings change management to saving files at different points in the development process. A **Version control system (VCS)** that will we make use of is Git.

Git is a decentralized distributed versioning system, 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.

For more information on how to use Git, head over to [Pro Git](https://www.gitbook.com/book/gitbookio/progit/details)

# 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](http://nodejs.org/)

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

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

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

# Back-End Code Sharing and Distribution: [npm](https://www.npmjs.com/)

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](http://webpack.github.io/docs/what-is-webpack.html)

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

## Web Browsers

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

However you are free to use other browsers. Not well known, there is a Mozilla Firefox [Developer Edition](https://www.mozilla.org/en-US/firefox/developer/) available with support for great development and debugging tools. Code written with JavaScript should work on any modern web browser (Firefox, IE9+, Chrome, Safari, Opera).

# Bootstrapping an Angular Application

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

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

# Understanding the File Structure

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

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

app/app.component.ts

import { Component } from '@angular/core'

@Component({

selector: 'app-root',

template: '<b>Bootstrapping an Angular Application</b>'

})

export class AppComponent { }

index.html

<body>

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

</body>

app/app.module.ts

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

import { NgModule } '@angular/core';

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

@NgModule({

imports: [BrowserModule],

declarations: [AppComponent],

bootstrap: [AppComponent]

})

export class AppModule {

}

app/main.ts

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);

If you're making use of Ahead-of-Time (AoT) compilation, you would code main.ts as follows.

import { platformBrowser} from '@angular/platform-browser';

import { AppModuleNgFactory } from '../aot/app/app.module.ngfactory';

platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);

[View Example](https://plnkr.co/edit/X0EBXA?p=preview)

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

There is an app HTML element in the index.html file, and we use app/main.ts to import the AppModulecomponent and the platformBrowserDynamic().bootstrapModule function and kickstart the process. As shown above, you may optionally use **AoT** in which case you will be working with Factories, in the example, AppModuleNgFactory and bootstrapModuleFactory.

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

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

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

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

# Bootstrapping Providers

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

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

app/app.module.ts

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

import { NgModule } '@angular/core';

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

import { GreeterService } from './greeter.service';

@NgModule({

imports: [BrowserModule],

providers: [GreeterService],

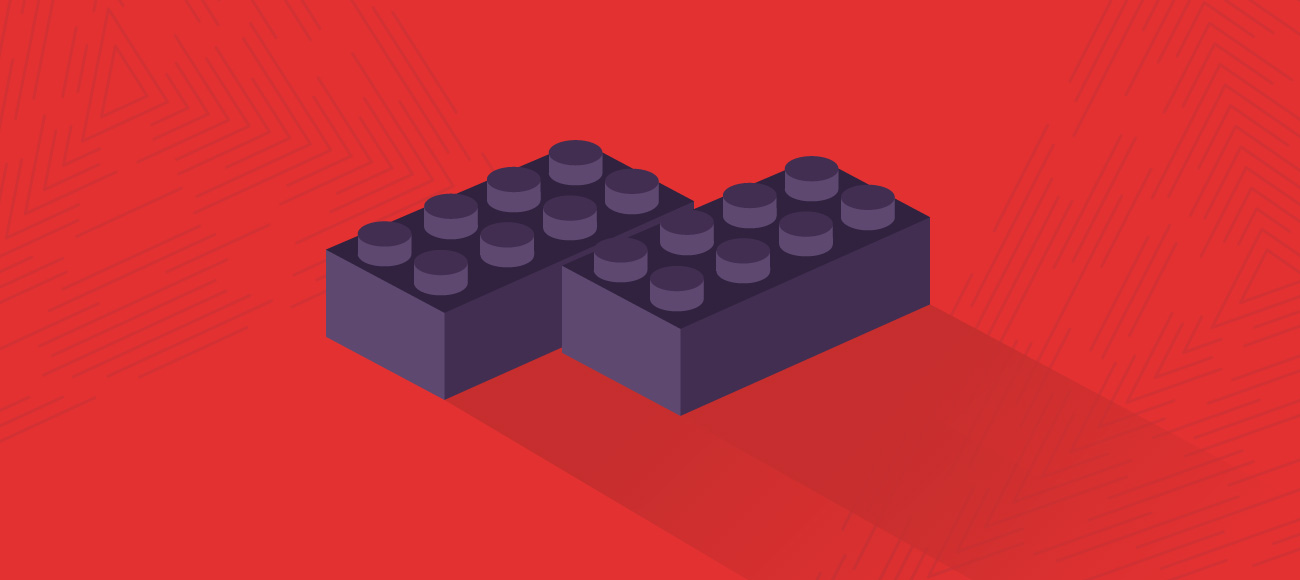
declarations: [AppComponent],

bootstrap: [AppComponent]

})

export class AppModule { }

**Components in Angular**

Figure: components

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

This is how the Angular team defines a component:

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

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

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

* selector is the element property that we use to tell Angular to create and insert an instance of this component.
* template is a form of HTML that tells Angular what needs to be to rendered in the DOM.

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

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

@Component({

selector: 'rio-hello',

template: '<p>Hello, {{name}}!</p>',

})

export class HelloComponent {

name: string;

constructor() {

this.name = 'World';

}

}

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

# Application Structure with Components

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

For example consider the following:

<rio-todo-app>

<rio-todo-list>

<rio-todo-item></rio-todo-item>

<rio-todo-item></rio-todo-item>

<rio-todo-item></rio-todo-item>

</rio-todo-list>

<rio-todo-form></rio-todo-form>

</rio-todo-app>

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

# Passing Data into a Component

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

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

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

@Component({

selector: 'rio-hello',

template: '<p>Hello, {{name}}!</p>',

})

export class HelloComponent {

@Input() name: string;

}

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

We can now use our component like so:

<!-- To bind to a raw string -->

<rio-hello name="World"></rio-hello>

<!-- To bind to a variable in the parent scope -->

<rio-hello [name]="helloName"></rio-hello>

[View Example](http://plnkr.co/edit/LEtEN9?p=preview)

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

# Responding to Component Events

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

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

@Component({

selector: 'rio-counter',

template: `

<div>

<p>Count: {{num}}</p>

<button (click)="increment()">Increment</button>

</div>

`

})

export class CounterComponent {

num = 0;

increment() {

this.num++;

}

}

[View Example](http://plnkr.co/edit/l4FweMxodN8I26OeqhGH?p=preview)

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

app/counter.component.ts

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

@Component({

selector: 'rio-counter',

templateUrl: 'app/counter.component.html'

})

export class CounterComponent {

@Input() count = 0;

@Output() result = new EventEmitter<number>();

increment() {

this.count++;

this.result.emit(this.count);

}

}

app/counter.component.html

<div>

<p>Count: {{ count }}</p>

<button (click)="increment()">Increment</button>

</div>

app/app.component.ts

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

@Component({

selector: 'rio-app',

templateUrl: 'app/app.component.html'

})

export class AppComponent implements OnChange {

num = 0;

parentCount = 0;

ngOnChange(val: number) {

this.parentCount = val;

}

}

app/app.component.html

<div>

Parent Num: {{ num }}<br>

Parent Count: {{ parentCount }}

<rio-counter [count]="num" (result)="ngOnChange($event)">

</rio-counter>

</div>

[View Example](http://plnkr.co/edit/5RYLZ0?p=preview)

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 ngModeldirective.

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

What this is doing behind the scenes is equivalent to:

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

To create your own component that supports two-way binding, you must define an @Output property to match an @Input, but suffix it with the Change. The code example below, inside class CounterComponent shows how to make property count support two-way binding.

app/counter.component.ts

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

@Component({

selector: 'rio-counter',

templateUrl: 'app/counter.component.html'

})

export class CounterComponent {

@Input() count = 0;

@Output() countChange = EventEmitter<number>();

increment() {

this.count++;

this.countChange.emit(this.count);

}

}

app/counter.component.html

<div>

<p>

<ng-content></ng-content>

Count: {{ count }} -

<button (click)="increment()">Increment</button>

</p>

</div>

[View Example](http://plnkr.co/edit/nkww1Ov2AWZRMHFyjhjl?p=preview)

**Access Child Components From the Template**

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

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

*app/app.component.html*

<section >

<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">

<label for="name">Name</label>

<input type="text" name="name" id="name" ngModel>

<button type="submit">Submit</button>

</form>

Form Value: {{formValue}}

</section>

*app/app.component.ts*

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

@Component({

selector: 'rio-app',

templateUrl: 'app/app.component.html'

})

export class AppComponent {

formValue = JSON.stringify({});

onSubmit (form: NgForm) {

this.formValue = JSON.stringify(form.value);

}

}

[View Example](https://plnkr.co/edit/hfv5RC?p=preview)

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

*app/app.component.html*

<rio-profile #profile></rio-profile>

My name is {{ profile.name }}

*app/profile.component.ts*

@Component({

selector: 'rio-profile',

templateUrl: 'app/profile.component.html'

})

export class ProfileComponent {

name = 'John Doe';

}

[View Example](https://plnkr.co/edit/wEFOta?p=preview)

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

+

# Projection

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

@Component({

selector: 'rio-child',

template: `

<div>

<h4>Child Component</h4>

{{ childContent }}

</div>

`

})

export class ChildComponent {

childContent = "Default content";

}

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

So, change ChildComponent to use projection:

app/child/child.component.ts

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

@Component({

selector: 'rio-child',

template: `

<div style="border: 1px solid blue; padding: 1rem;">

<h4>Child Component</h4>

<ng-content></ng-content>

</div>

`

})

export class ChildComponent {

}

Then, when we use ChildComponent in the template:

app/app.component.html

...

<rio-child>

<p>My <i>projected</i> content.</p>

</rio-child>

...

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

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

[View Example](http://plnkr.co/edit/QAQ6BFuwuzEDVvqAmN9L?p=preview)

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

app/child-select.component.html

<div style="...">

<h4>Child Component with Select</h4>

<div style="...">

<ng-content select="header"></ng-content>

</div>

<div style="...">

<ng-content select="section"></ng-content>

</div>

<div style="...">

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

</div>

<div style="...">

<ng-content select="footer"></ng-content>

</div>

</div>

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

app/app.component.html

...

<rio-child-select>

<section>Section Content</section>

<div class="class-select">

div with .class-select

</div>

<footer>Footer Content</footer>

<header>Header Content</header>

</rio-child-select>

...

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

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

app/app.component.html

<div class="class-select">

div with .class-select

</div>

[View Example](http://plnkr.co/edit/rH2vGgFluLXHCsgfkNjF?p=preview)

# Structuring Applications with Components

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

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

# Using Other Components

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

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

import {TodoListComponent} from './components/todo-list.component';

import {TodoItemComponent} from './components/todo-item.component';

import {TodoInputComponent} from './components/todo-input.component';

@NgModule({

imports: [ ... ],

declarations: [

TodoListComponent,

TodoItemComponent,

TodoInputComponent

],

bootstrap: [ ... ]

})

export class ToDoAppModule {

}

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

Please see the [Modules section](https://angular-2-training-book.rangle.io/handout/modules/) for more info about NgModule.

# Directives

A Directive modifies the DOM to change apperance, behavior or layout of DOM elements. Directives are one of the core building blocks Angular uses to build applications. In fact, Angular components are in large part directives with templates.

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

There are three main types of directives in Angular:

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

# Attribute Directives

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

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

# NgStyle Directive

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

@Component({

selector: 'app-style-example',

template: `

<p style="padding: 1rem"

[ngStyle]="{

'color': 'red',

'font-weight': 'bold',

'borderBottom': borderStyle

}">

<ng-content></ng-content>

</p>

`

})

export class StyleExampleComponent {

borderStyle = '1px solid black';

}

[View Example](https://plnkr.co/edit/akK3Gw8W6EgUQ4PRQp4h?p=preview)

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

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

@Component({

selector: 'app-style-example',

template: `

<p style="padding: 1rem"

[ngStyle]="alertStyles">

<ng-content></ng-content>

</p>

`

})

export class StyleExampleComponent {

borderStyle = '1px solid black';

alertStyles = {

'color': 'red',

'font-weight': 'bold',

'borderBottom': this.borderStyle

};

}

# NgClass Directive

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

## Binding a string

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

@Component({

selector: 'app-class-as-string',

template: `

<p ngClass="centered-text underlined" class="orange">

<ng-content></ng-content>

</p>

`,

styles: [`

.centered-text {

text-align: center;

}

.underlined {

border-bottom: 1px solid #ccc;

}

.orange {

color: orange;

}

`]

})

export class ClassAsStringComponent {

}

[View Example](https://plnkr.co/edit/uUtjY1Qlkx5dOB8gsqCm?p=preview)

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

## Binding an array

@Component({

selector: 'app-class-as-array',

template: `

<p [ngClass]="['warning', 'big']">

<ng-content></ng-content>

</p>

`,

styles: [`

.warning {

color: red;

font-weight: bold;

}

.big {

font-size: 1.2rem;

}

`]

})

export class ClassAsArrayComponent {

}

[View Example](https://plnkr.co/edit/uUtjY1Qlkx5dOB8gsqCm?p=preview)

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

## Binding an object

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

@Component({

selector: 'app-class-as-object',

template: `

<p [ngClass]="{ card: true, dark: false, flat: flat }">

<ng-content></ng-content>

<br>

<button type="button" (click)="flat=!flat">Toggle Flat</button>

</p>

`,

styles: [`

.card {

border: 1px solid #eee;

padding: 1rem;

margin: 0.4rem;

font-family: sans-serif;

box-shadow: 2px 2px 2px #888888;

}

.dark {

background-color: #444;

border-color: #000;

color: #fff;

}

.flat {

box-shadow: none;

}

`]

})

export class ClassAsObjectComponent {

flat: boolean = true;

}

[View Example](https://plnkr.co/edit/uUtjY1Qlkx5dOB8gsqCm?p=preview)

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

# Structural Directives

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

Note: For those who are unfamiliar with the*template*tag, it is an HTML element with a few special properties. Content nested in a template tag is not rendered on page load and is something that is meant to be loaded through code at runtime. For more information on the*template*tag, visit the[*MDN documentation*](https://developer.mozilla.org/en/docs/Web/HTML/Element/template).

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

@Component({

selector: 'app-directive-example',

template: `

<p \*structuralDirective="expression">

Under a structural directive.

</p>

`

})

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

@Component({

selector: 'app-directive-example',

template: `

<template [structuralDirective]="expression">

<p>

Under a structural directive.

</p>

</template>

`

})

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

@Component({

selector: 'app-directive-example',

template: `

<p template="structuralDirective expression">

Under a structural directive.

</p>

`

})

# NgIf Directive

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

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

@Component({

selector: 'app-root',

template: `

<button type="button" (click)="toggleExists()">Toggle Component</button>

<hr>

<app-if-example \*ngIf="exists">

Hello

</app-if-example>

`

})

export class AppComponent {

exists = true;

toggleExists() {

this.exists = !this.exists;

}

}

[View Example](https://plnkr.co/edit/Kb0KW89265F0e9pYJ118?p=preview)

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

# NgFor Directive

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

@Component({

selector: 'app-root',

template: `

<app-for-example \*ngFor="let episode of episodes" [episode]="episode">

{{episode.title}}

</app-for-example>

`

})

export class AppComponent {

episodes = [

{ title: 'Winter Is Coming', director: 'Tim Van Patten' },

{ title: 'The Kingsroad', director: 'Tim Van Patten' },

{ title: 'Lord Snow', director: 'Brian Kirk' },

{ title: 'Cripples, Bastards, and Broken Things', director: 'Brian Kirk' },

{ title: 'The Wolf and the Lion', director: 'Brian Kirk' },

{ title: 'A Golden Crown', director: 'Daniel Minahan' },

{ title: 'You Win or You Die', director: 'Daniel Minahan' },

{ title: 'The Pointy End', director: 'Daniel Minahan' }

];

}

[View Example](https://plnkr.co/edit/dXU4K13piTYotDX5Nhi6?p=preview)

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

@Component({

selector: 'app',

template: `

<template ngFor [ngForOf]="episodes" let-episode>

<app-for-example [episode]="episode">

{{episode.title}}

</app-for-example>

</template>

`

})

[View Example](https://plnkr.co/edit/dXU4K13piTYotDX5Nhi6?p=preview)

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

## Local Variables

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

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

@Component({

selector: 'app-root',

template: `

<app-for-example

\*ngFor="let episode of episodes; let i = index; let isOdd = odd"

[episode]="episode"

[ngClass]="{ odd: isOdd }">

{{i+1}}. {{episode.title}}

</app-for-example>

<hr>

<h2>Desugared</h2>

<template ngFor [ngForOf]="episodes" let-episode let-i="index" let-isOdd="odd">

<for-example [episode]="episode" [ngClass]="{ odd: isOdd }">

{{i+1}}. {{episode.title}}

</for-example>

</template>

`

})

[View Example](https://plnkr.co/edit/58A5p8cWpVIY7Ne4O7aO?p=preview)

## trackBy

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

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

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

@Component({

selector: 'app-root',

template: `

<button

(click)="addOtherEpisode()"

[disabled]="otherEpisodes.length === 0">

Add Episode

</button>

<app-for-example

\*ngFor="let episode of episodes;

let i = index; let isOdd = odd;

trackBy: trackById" [episode]="episode"

[ngClass]="{ odd: isOdd }">

{{episode.title}}

</app-for-example>

`

})

export class AppComponent {

otherEpisodes = [

{ title: 'Two Swords', director: 'D. B. Weiss', id: 8 },

{ title: 'The Lion and the Rose', director: 'Alex Graves', id: 9 },

{ title: 'Breaker of Chains', director: 'Michelle MacLaren', id: 10 },

{ title: 'Oathkeeper', director: 'Michelle MacLaren', id: 11 }]

episodes = [

{ title: 'Winter Is Coming', director: 'Tim Van Patten', id: 0 },

{ title: 'The Kingsroad', director: 'Tim Van Patten', id: 1 },

{ title: 'Lord Snow', director: 'Brian Kirk', id: 2 },

{ title: 'Cripples, Bastards, and Broken Things', director: 'Brian Kirk', id: 3 },

{ title: 'The Wolf and the Lion', director: 'Brian Kirk', id: 4 },

{ title: 'A Golden Crown', director: 'Daniel Minahan', id: 5 },

{ title: 'You Win or You Die', director: 'Daniel Minahan', id: 6 }

{ title: 'The Pointy End', director: 'Daniel Minahan', id: 7 }

];

addOtherEpisode() {

// We want to create a new object reference for sake of example

let episodesCopy = JSON.parse(JSON.stringify(this.episodes))

this.episodes=[...episodesCopy,this.otherEpisodes.pop()];

}

trackById(index: number, episode: any): number {

return episode.id;

}

}

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

export class ForExampleComponent {

@Input() episode;

ngOnInit() {

console.log('component created', this.episode)

}

ngOnDestroy() {

console.log('destroying component', this.episode)

}

}

[View Example](https://plnkr.co/edit/jQmozF?p=preview)

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](https://plnkr.co/edit/hC2cIK?p=preview)

# NgSwitch Directives

ngSwitch is actually comprised of two directives, an attribute directive and a structural directive. It's very similar to a [switch statement](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Statements/switch) in JavaScript and other programming languages, but in the template.

@Component({

selector: 'app-root',

template: `

<div class="tabs-selection">

<app-tab [active]="isSelected(1)" (click)="setTab(1)">Tab 1</app-tab>

<app-tab [active]="isSelected(2)" (click)="setTab(2)">Tab 2</app-tab>

<app-tab [active]="isSelected(3)" (click)="setTab(3)">Tab 3</app-tab>

</div>

<div [ngSwitch]="tab">

<app-tab-content \*ngSwitchCase="1">Tab content 1</app-tab-content>

<app-tab-content \*ngSwitchCase="2">Tab content 2</app-tab-content>

<app-tab-content \*ngSwitchCase="3"><app-tab-3></app-tab-3></app-tab-content>

<app-tab-content \*ngSwitchDefault>Select a tab</app-tab-content>

</div>

`

})

export class AppComponent {

tab: number = 0;

setTab(num: number) {

this.tab = num;

}

isSelected(num: number) {

return this.tab === num;

}

}

[View Example](https://plnkr.co/edit/QWxD0DIZi6QiISafwfgu?p=preview)

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

# Using Multiple Structural Directives

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

@Component({

selector: 'app-root',

template: `

<template ngFor [ngForOf]="[1,2,3,4,5,6]" let-item>

<div \*ngIf="item > 3">

{{item}}

</div>

</template>

`

})

[View Example](https://plnkr.co/edit/V2nWlGOwIITPrUDksGNG?p=preview)

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

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

@Component({

selector: 'app-root',

template: `

<div class="tabs-selection">

<tab

\*ngFor="let tab of tabs; let i = index"

[active]="isSelected(i)"

(click)="setTab(i)">

{{ tab.title }}

</tab>

</div>

<div [ngSwitch]="tabNumber">

<template ngFor [ngForOf]="tabs" let-tab let-i="index">

<tab-content \*ngSwitchCase="i">

{{tab.content}}

</tab-content>

</template>

<tab-content \*ngSwitchDefault>Select a tab</tab-content>

</div>

`

})

export class AppComponent {

tabNumber: number = -1;

tabs = [

{ title: 'Tab 1', content: 'Tab content 1' },

{ title: 'Tab 2', content: 'Tab content 2' },

{ title: 'Tab 3', content: 'Tab content 3' },

];

setTab(num: number) {

this.tabNumber = num;

}

isSelected(num: number) {

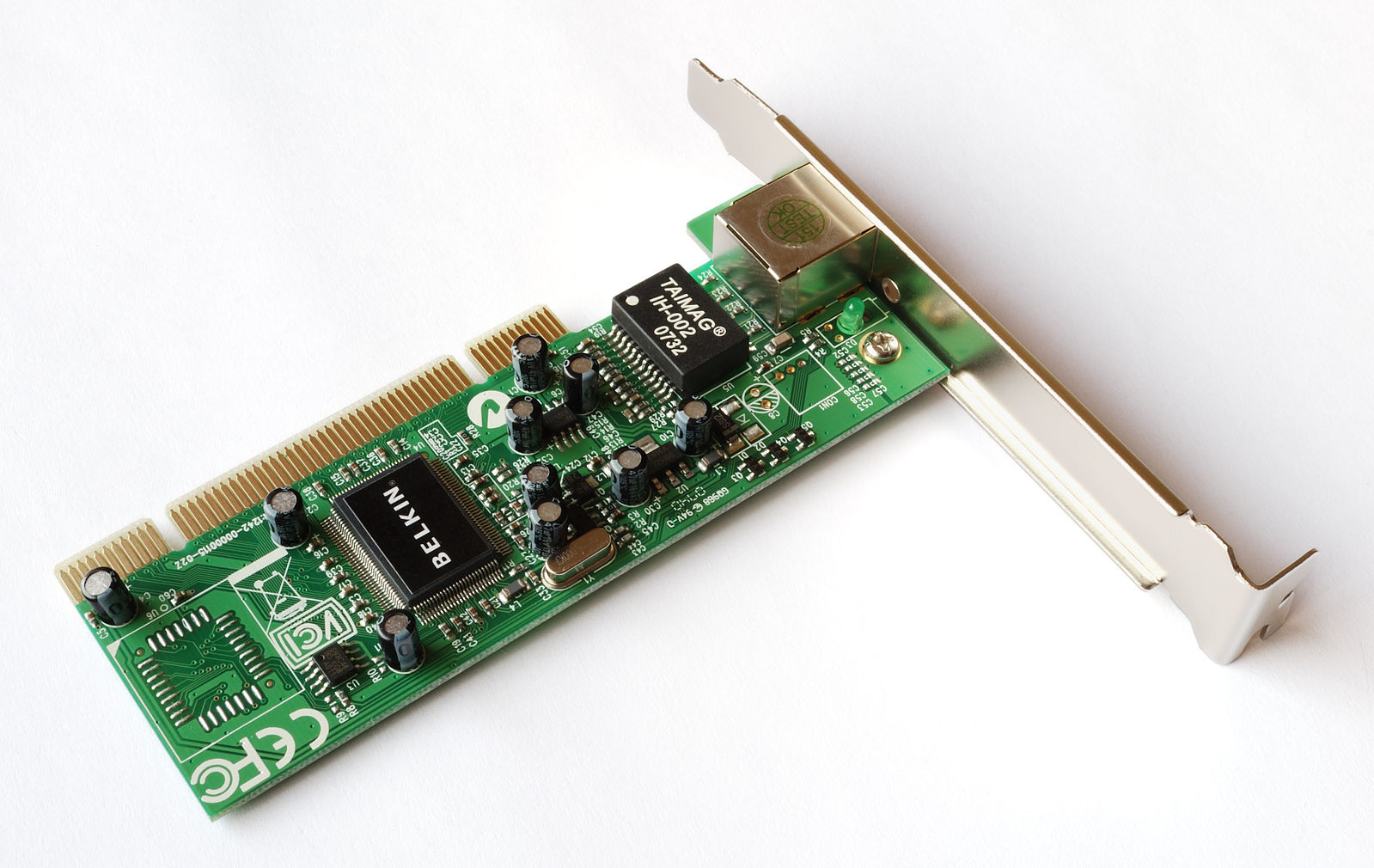
return this.tabNumber === i;

}

}

[View Example](https://plnkr.co/edit/YOT4G4buUZduwvVi8cMA?p=preview)

**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](https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html)

[View Example](http://plnkr.co/edit/kBHV6AximaHAC26kYEOA?p=preview)

# Accessing Child Component Classes

## @ViewChild and @ViewChildren

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

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

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

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

import { AlertComponent } from './alert.component';

@Component({

selector: 'app-root',

template: `

<app-alert>My alert</app-alert>

<button (click)="showAlert()">Show Alert</button>`

})

export class AppComponent {

@ViewChild(AlertComponent) alert: AlertComponent;

showAlert() {

this.alert.show();

}

}

[View Example](http://plnkr.co/edit/NEeEPfkHsYBbVuuAxz5z?p=preview)

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

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

import { Component, QueryList, ViewChildren } from '@angular/core';

import { AlertComponent } from './alert.component';

@Component({

selector: 'app-root',

template: `

<app-alert ok="Next" (close)="showAlert(2)">

Step 1: Learn angular

</app-alert>

<app-alert ok="Next" (close)="showAlert(3)">

Step 2: Love angular

</app-alert>

<app-alert ok="Close">

Step 3: Build app

</app-alert>

<button (click)="showAlert(1)">Show steps</button>`

})

export class AppComponent {

@ViewChildren(AlertComponent) alerts: QueryList<AlertComponent>;

alertsArr = [];

ngAfterViewInit() {

this.alertsArr = this.alerts.toArray();

}

showAlert(step) {

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

}

}

[View Example](http://plnkr.co/edit/zPtb3ZJLx7CWJa7RptxZ?p=preview)

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

@Component({

selector: 'app-root',

template: `

<app-alert #first ok="Next" (close)="showAlert(2)">

Step 1: Learn angular

</app-alert>

<app-alert ok="Next" (close)="showAlert(3)">

Step 2: Love angular

</app-alert>

<app-alert ok="Close">

Step 3: Build app

</app-alert>

<button (click)="showAlert(1)">Show steps</button>`

})

export class AppComponent {

@ViewChild('first') alert: AlertComponent;

@ViewChildren(AlertComponent) alerts: QueryList<AlertComponent>;

// ...

}

[View Example](http://plnkr.co/edit/EnOxkmJy7Y1LIPN4wUKc?p=preview)

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

## @ContentChild and @ContentChildren

@ContentChild and @ContentChildren work the same way as the equivalent @ViewChild and @ViewChildren, however, the key difference is that @ContentChild and @ContentChildren select from the [projected content](https://angular-2-training-book.rangle.io/handout/components/projection.html) within the component.

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

[View Example](http://plnkr.co/edit/SkX3kkAA4uprtwfjDZ6y?p=preview)

# View Encapsulation

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

* Emulated (default) - styles from main HTML propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.
* Native - styles from main HTML do not propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.
* None - styles from the component propagate back to the main HTML and therefore are visible to all components on the page. Be careful with apps that have None and Native components in the application. All components with None encapsulation will have their styles duplicated in all components with Native encapsulation.
* @Component({
* // ...
* encapsulation: ViewEncapsulation.None,
* styles: [
* // ...
* ]
* })
* export class HelloComponent {
* // ...
* }

[View Example](http://plnkr.co/edit/E5Hb6B5dRN0llz3JuO57?p=preview)

# ElementRef

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

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

@Component({

selector: 'app-root',

template: `

<h1>My App</h1>

<pre>

<code>{{ node }}</code>

</pre>

`

})

export class AppComponent implements AfterContentInit {

node: string;

constructor(private elementRef: ElementRef) { }

ngAfterContentInit() {

const tmp = document.createElement('div');

const el = this.elementRef.nativeElement.cloneNode(true);

tmp.appendChild(el);

this.node = tmp.innerHTML;

}

}

[View Example](https://plnkr.co/edit/TY7SrMXs8XoV6AOYwn9k?p=preview)

# Observables

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

# Using Observables

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

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

import {Observable} from 'rxjs/Observable';

@Component({

selector: 'app',

template: `

<b>Angular Component Using Observables!</b>

<h6 style="margin-bottom: 0">VALUES:</h6>

<div \*ngFor="let value of values">- {{ value }}</div>

<h6 style="margin-bottom: 0">ERRORs:</h6>

<div>Errors: {{anyErrors}}</div>

<h6 style="margin-bottom: 0">FINISHED:</h6>

<div>Finished: {{ finished }}</div>

<button style="margin-top: 2rem;" (click)="init()">Init</button>

`

})

export class MyApp {

private data: Observable<Array<number>>;

private values: Array<number> = [];

private anyErrors: boolean;

private finished: boolean;

constructor() {

}

init() {

this.data = new Observable(observer => {

setTimeout(() => {

observer.next(42);

}, 1000);

setTimeout(() => {

observer.next(43);

}, 2000);

setTimeout(() => {

observer.complete();

}, 3000);

});

let subscription = this.data.subscribe(

value => this.values.push(value),

error => this.anyErrors = true,

() => this.finished = true

);

}

}

[View Example](http://plnkr.co/edit/SA25mG?p=preview)

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 numbertype. This illustrates the stream of data that Observables offer as well as giving us the ability to maintain integrity of the type of data we are expecting to receive.

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

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

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

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

new Promise((resolve, reject) => {

Observable.of(1, 2, 3).subscribe(

doSomething,

reject,

resolve);

});

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

export class MyApp {

private data: Observable<Array<number>>;

private values: Array<number> = [];

private anyErrors: boolean;

private finished: boolean;

constructor() {

}

init() {

this.data = new Observable(observer => {

setTimeout(() => {

observer.next(42);

}, 1000);

setTimeout(() => {

observer.next(43);

}, 2000);

setTimeout(() => {

observer.complete();

}, 3000);

this.status = "Started";

});

let subscription = this.data.forEach(v => this.values.push(v))

.then(() => this.status = "Ended");

}

}

[View Example](http://plnkr.co/edit/eJWIJd?p=preview)

# Error Handling

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

export class App {

values: number[] = [];

anyErrors: Error;

private data: Observable<number[]>;

constructor() {

this.data = new Observable(observer => {

setTimeout(() => {

observer.next(10);

}, 1500);

setTimeout(() => {

observer.error(new Error('Something bad happened!'));

}, 2000);

setTimeout(() => {

observer.next(50);

}, 2500);

});

let subscription = this.data.subscribe(

value => this.values.push(value),

error => this.anyErrors = error

);

}

}

[View Example](http://plnkr.co/edit/09rodT?p=preview)

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

# Disposing Subscriptions and Releasing Resources

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

export class MyApp {

private data: Observable<Array<string>>;

private value: string;

private subscribed: boolean;

private status: string;

init() {

this.data = new Observable(observer => {

let timeoutId = setTimeout(() => {

observer.next('You will never see this message');

}, 2000);

this.status = 'Started';

return onUnsubscribe = () => {

this.subscribed = false;

this.status = 'Finished';

clearTimeout(timeoutId);

}

});

let subscription = this.data.subscribe(

value => this.value = value,

error => console.log(error),

() => this.status = 'Finished';

);

this.subscribed = true;

setTimeout(() => {

subscription.unsubscribe();

}, 1000);

}

}

[View Example](http://plnkr.co/edit/0MfW5d?p=preview)

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 Observableoperators 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';

import 'rxjs/Rx';

@Component({

selector: 'app',

template: `

<b>Angular HTTP requests using RxJs Observables!</b>

<ul>

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

</ul>

`

})

export class MyApp {

private doctors = [];

constructor(http: Http) {

http.get('http://jsonplaceholder.typicode.com/users/')

.flatMap((data) => data.json())

.subscribe((data) => {

this.doctors.push(data);

});

}

}

[View Example](http://plnkr.co/edit/AikZi1?p=preview)

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 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.

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

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

import 'rxjs/add/operator/map';

@Component({

selector: 'app',

template: `

<form [formGroup]="coolForm">

<input formControlName="email">

</form>

<div>

<b>You Typed Reversed:</b> {{data}}

</div>

`

})

export class MyApp {

email: FormControl;

coolForm: FormGroup;

data: string;

constructor(private fb: FormBuilder) {

this.email = new FormControl();

this.coolForm = fb.group({

email: this.email

});

this.email.valueChanges

.map(n=>n.split('').reverse().join(''))

.subscribe(value => this.data = value);

}

}

[View Example](http://plnkr.co/edit/vCdjZM?p=preview)

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

# Observables Array Operations

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

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

export class MyApp {

private doctors = [];

constructor(http: Http) {

http.get('http://jsonplaceholder.typicode.com/users/')

.flatMap((response) => response.json())

.filter((person) => person.id > 5)

.map((person) => "Dr. " + person.name)

.subscribe((data) => {

this.doctors.push(data);

});

}

}

[View Example](http://plnkr.co/edit/a0JuHC?p=preview)

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 Observablestream. This is done with flatMap.

There are many other array operations you can employ in your Observables; look for them in the [RxJS API](https://github.com/Reactive-Extensions/RxJS).

[rxmarbles.com](http://rxmarbles.com/) is a helpful resource to understand how the operations work.

# Cold vs Hot Observables

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

const obsv = new Observable(observer => {

setTimeout(() => {

observer.next(1);

}, 1000);

setTimeout(() => {

observer.next(2);

}, 2000);

setTimeout(() => {

observer.next(3);

}, 3000);

setTimeout(() => {

observer.next(4);

}, 4000);

});

// Subscription A

setTimeout(() => {

obsv.subscribe(value => console.log(value));

}, 0);

// Subscription B

setTimeout(() => {

obsv.subscribe(value => console.log(`>>>> ${value}`));

}, 2500);

[View Example](http://jsbin.com/felanu/46/edit?js,console)

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](http://jsbin.com/fewotud/3/edit?js,console)

In the case above, the live performance starts at 1000ms, subscriber A arrived to the concert hall at 0sto 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 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](https://github.com/ReactiveX/rxjs/blob/master/MIGRATION.md)
* [rxjs Observable API](http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html)
* [Which operator do I use?](https://xgrommx.github.io/rx-book/content/which_operator_do_i_use/instance_operators.html)
* [rxmarbles](http://rxmarbles.com/)
* [RxJS Operators by Example](https://gist.github.com/btroncone/d6cf141d6f2c00dc6b35#file-rxjs_operators_by_example-md)

# Angular 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 Hamburgerconsists of a Bun, Patty and Toppings. The class is also responsible for making the Bun, Pattyand Toppings. This is a bad thing. What if a vegetarian burger were needed? One naive approach might be:

class VeggieHamburger {

private bun: Bun;

private patty: Patty;

private toppings: Toppings;

constructor() {

this.bun = new Bun('withSesameSeeds');

this.patty = new Patty('tofu');

this.toppings = new Toppings(['lettuce', 'pickle', 'tomato']);

}

}

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

class Hamburger {

private bun: Bun;

private patty: Patty;

private toppings: Toppings;

constructor(bunType: string, pattyType: string, toppings: string[]) {

this.bun = new Bun(bunType);

this.patty = new Patty(pattyType);

this.toppings = new Toppings(toppings);

}

}

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

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

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

class Hamburger {

private bun: Bun;

private patty: Patty;

private toppings: Toppings;

constructor(bun: Bun, patty: Patty, toppings: Toppings) {

this.bun = bun;

this.patty = patty;

this.toppings = toppings;

}

}

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

class Hamburger {

constructor(private bun: Bun, private patty: Patty,

private toppings: Toppings) {}

}

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

However there is still a problem. How can the instantiation of Bun, Patty and Toppings best be managed?

This is where dependency injection as a framework can benefit programmers, and it is what Angular provides with its dependency injection system.

# DI Framework

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

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

That's a lot of work to create a Hamburger, and now all the different pieces of code that make Hamburgers 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 Hamburgers need to be created this factory will become confusing. The factory is also responsible for knowing how to create four different components. This is a lot for one function.

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

const injector = new Injector([Hamburger, Bun, Patty, Toppings]);

const burger = injector.get(Hamburger);

# Angular's DI

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

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

For example:

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

@Component({

// ...

})

class ChatWidget {

constructor(private authService: AuthService, private authWidget: AuthWidget,

private chatSocket: ChatSocket) {}

}

@NgModule({

declarations: [ ChatWidget ]

})

export class AppModule {

};

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

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

Angular assumes that it's being given a class.

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

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

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

@Component({

// ...

})

class ChatWidget {

constructor(private authService: AuthService, private authWidget: AuthWidget,

private chatSocket: ChatSocket) {}

}

@Component({

// ...

})

class AuthWidget {}

@Injectable()

class AuthService {}

@Injectable()

class ChatSocket {}

@NgModule({

declarations[ ChatWidget, AuthWidget ]

providers: [ AuthService, ChatSocket ],

})

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

# @Inject and @Injectable

Statements that look like @SomeName are decorators. [Decorators](http://blog.wolksoftware.com/decorators-reflection-javascript-typescript) 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](https://angular-2-training-book.rangle.io/handout/features/).

## @Inject()

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

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

import { ChatWidget } from '../components/chat-widget';

@Component({

selector: 'app-root',

template: `Encryption: {{ encryption }}`

})

export class AppComponent {

encryption = this.chatWidget.chatSocket.encryption;

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

}

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

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

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

import { ChatWidget } from '../components/chat-widget';

@Component({

selector: 'app',

template: `Encryption: {{ encryption }}`

})

export class App {

encryption = this.chatWidget.chatSocket.encryption;

constructor(private chatWidget: ChatWidget) { }

}

[View Example](https://plnkr.co/edit/BAYoY7W6tUkbnczk3Lsg?p=preview)

## @Injectable()

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

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

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

import { AuthService } from './auth-service';

import { AuthWidget } from './auth-widget';

import { ChatSocket } from './chat-socket';

@Injectable()

export class ChatWidget {

constructor(

public authService: AuthService,

public authWidget: AuthWidget,

public chatSocket: ChatSocket) { }

}

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

**Injection Beyond Classes**

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

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

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

import { App } from './containers/app'; // hypothetical app component

import { ChatWidget } from './components/chat-widget';

@NgModule({

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

})

export class DiExample {};

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

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

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

import { App } from './containers/app'; // hypothetical app component

import { ChatWidget } from './components/chat-widget';

import { MockChatWidget } from './components/mock-chat-widget';

@NgModule({

providers: [ { provide: ChatWidget, useClass: MockChatWidget } ],

})

export class DiExample {};

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

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

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

import { App } from './containers/app'; // hypothetical app component

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

@NgModule({

providers: [ { provide: 'Random', useFactory: randomFactory } ],

})

export class DiExample {};

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

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

@Component({

selector: 'app-root',

template: `Random: {{ value }}`

})

export class AppCompoennt {

value: number;

constructor(@Inject('Random') r) {

this.value = r;

}

}

[View Example](http://plnkr.co/edit/BKMZYlAviRhauCzxMnx6?p=preview)

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

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

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

@NgModule({

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

})

export class DiExample {};

[View Example](http://plnkr.co/edit/xGMOsHn1v3tTbc9RkuDz?p=preview)

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

+

# Avoiding Injection Collisions: OpaqueToken

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

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

app/email/email.service.ts

export const apiConfig = 'api-config';

@Injectable()

export class EmailService {

constructor(@Inject(apiConfig) public apiConfig) { }

}

app/email/email.module.ts

@NgModule({

providers: [ EmailService ],

})

export class EmailModule { }

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

app/logger/logger.service.ts

export const apiConfig = 'api-config';

@Injectable()

export class LoggerService {

constructor(@Inject(apiConfig) public apiConfig) { }

}

app/logger/logger.module.ts

@NgModule({

providers: [ LoggerService ],

})

export class LoggerModule { }

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

app/app.module.ts

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

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

@NgModule({

...

providers: [

{ provide: emailApiConfig, useValue: { apiKey: 'email-key', context: 'registration' } },

{ provide: loggerApiConfig, useValue: { apiKey: 'logger-key' } },

],

...

})

export class AppModule { }

[View Example](https://plnkr.co/edit/QrvjsucT6lF6dnFUb2ag?p=preview)

When the application runs, it encounters a collision problem resulting in both modules getting the same value for their configuration, in this case { apiKey: 'logger-key' }. When it comes time for the main application to specify those settings, Angular overwrites the first emailApiConfig value with the loggerApiConfig value, since that was provided last. In this case, module implementation details are leaking out to the parent module. Not only that, those details were obfuscated through the module exports and this can lead to problematic debugging. This is where Angular's OpaqueToken comes into play.

## OpaqueToken

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

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

const name = 'token';

const token1 = new OpaqueToken(name);

const token2 = new OpaqueToken(name);

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

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

app/email/email.module.ts

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

@Injectable()

export class EmailService {

constructor(@Inject(apiConfig) public apiConfig: EmailConfig) { }

}

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

@Injectable()

export class LoggerService {

constructor(@Inject(apiConfig) public apiConfig: LoggerConfig) { }

}

[View Example](https://plnkr.co/edit/SHfTH9R6JVDwJKnzRFSH?p=preview)

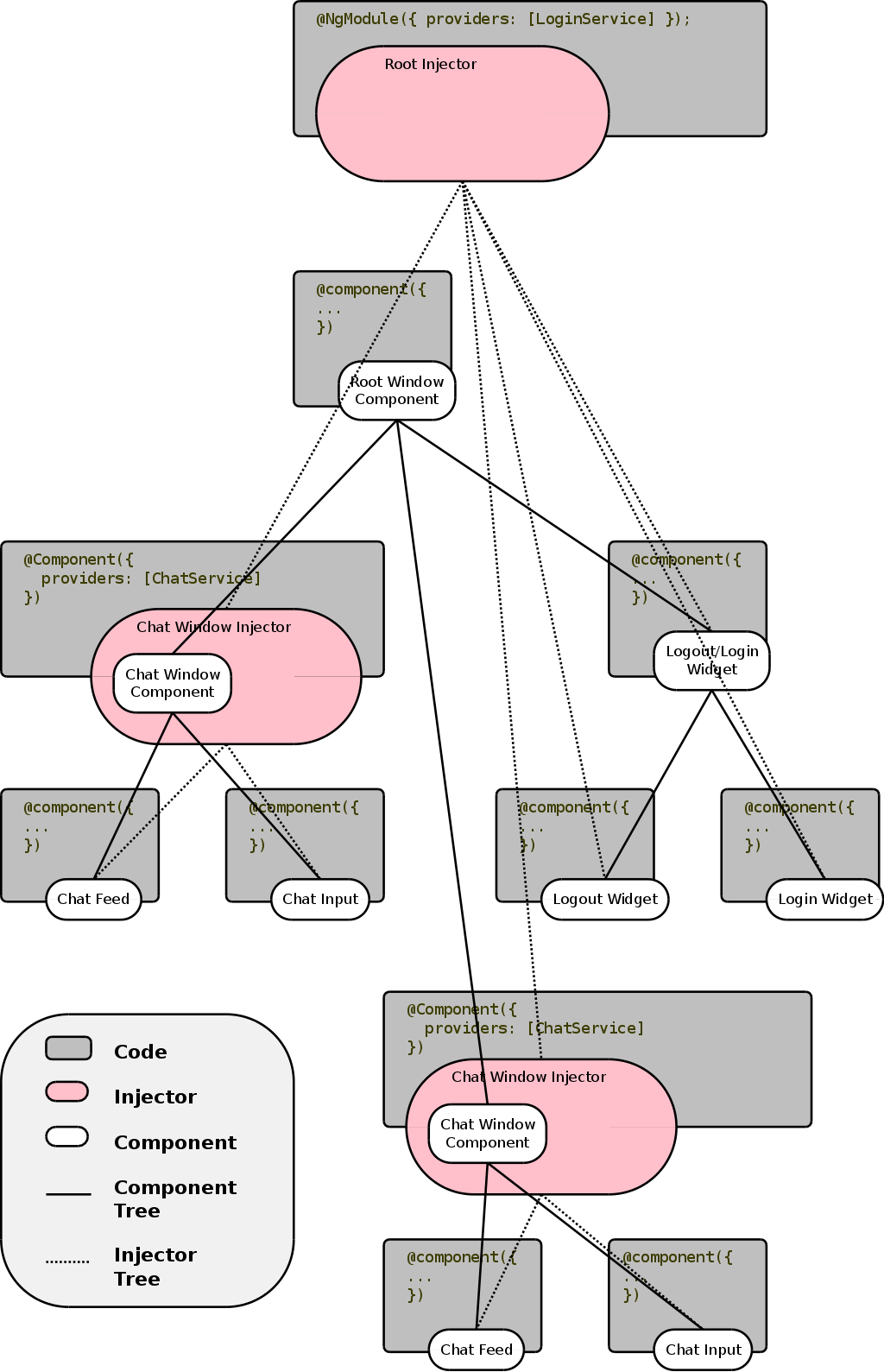
After turning the identifying tokens into OpaqueTokens without changing anything else, the collision is avoided. Every service gets the correct configuration object from the root module and Angular is now able to differentiate two tokens that uses the same string.

**The Injector Tree**

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

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

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

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

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

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

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

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

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

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

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

Consider the following example:

*app/module.ts*

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

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

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

import { ChildInheritorComponent, ChildOwnInjectorComponent } from './components/index';

import { Unique } from './services/unique';

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

@NgModule({

imports: [BrowserModule],

declarations: [

AppComponent,

ChildInheritorComponent,

ChildOwnInjectorComponent,

],

/\*\* Provide dependencies here \*/

providers: [Unique],

bootstrap: [AppComponent],

})

export class AppModule {}

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

*app/services/unique.ts*

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

@Injectable()

export class Unique {

value = (+Date.now()).toString(16) + '.' +

Math.floor(Math.random() \* 500);

}

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

*app/components/child-inheritor.component.ts*

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

import { Unique } from '../services/unique';

@Component({

selector: 'app-child-inheritor',

template: `<span>{{ value }}</span>`

})

export class ChildInheritorComponent {

value = this.u.value;

constructor(private u: Unique) { }

}

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

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

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

import { Unique } from '../services/unique';

@Component({

selector: 'child-own-injector',

template: `<span>{{ value }}</span>`,

providers: [Unique]

})

export class ChildOwnInjectorComponent {

value = this.u.value;

constructor(private u: Unique) { }

}

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

*app/containers/app.ts*

@Component({

selector: 'app-root',

template: `

<p>

App's Unique dependency has a value of {{ value }}

</p>

<p>

which should match

</p>

<p>

ChildInheritor's value:

<app-child-inheritor></app-child-inheritor>

</p>

<p>

However,

</p>

<p>

ChildOwnInjector should have its own value:

<app-child-own-injector></app-child-own-injector>

</p>

<p>

ChildOwnInjector's other instance should also have its own value:

<app-child-own-injector></app-child-own-injector>

</p>`,

})

export class AppComponent {

value: number = this.u.value;

constructor(private u: Unique) { }

}

[View Example](http://plnkr.co/edit/abeUOuG8AdHDUcvjial8?p=preview)

## HTTP

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

### Setting up angular/http

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

...

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

import { HttpModule } from '@angular/http';

@NgModule({

imports: [

BrowserModule,

ReactiveFormsModule,

FormsModule,

HttpModule

],

providers: [SearchService],

declarations: [AppComponent],

bootstrap: [AppComponent]

})

export class AppModule {}

## Making HTTP Requests

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

import { Http } from '@angular/http';

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

import { Observable } from 'rxjs/Observable';

import 'rxjs/add/operator/map';

@Injectable()

export class SearchService {

constructor(private http: Http) {}

search(term: string) {

return this.http

.get('https://api.spotify.com/v1/search?q=' + term + '&type=artist')

.map(response => response.json());

}

}

[View Example](http://plnkr.co/edit/C8Zv9i?p=preview)

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

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

## Catching Rejections

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

import { Http } from '@angular/http';

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

@Injectable()

export class AuthService {

constructor(private http: Http) {}

login(username, password) {

const payload = {

username: username,

password: password

};

this.http.post(`${ BASE\_URL }/auth/login`, payload)

.map(response => response.json())

.subscribe(

authData => this.storeToken(authData.id\_token),

(err) => console.error(err),

() => console.log('Authentication Complete')

);

}

}

## Catch and Release

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

import { 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/dsds?q=' + term + '&type=artist')

.map((response) => response.json())

.catch((e) => {

return Observable.throw(

new Error(`${ e.status } ${ e.statusText }`)

);

});

}

}

[View Example](http://plnkr.co/edit/3lCaeI?p=preview)

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

@Injectable()

export class SearchService {

...

search(term: string) {

return this.http.get(`https://api.spotify.com/v1/dsds?q=${term}&type=artist`)

.map(response => response.json())

.catch(e => {

if (e.status >== 500) {

return cachedVersion();

} else {

return Observable.throw(

new Error(`${ e.status } ${ e.statusText }`)

);

}

});

}

}

## Cancel a Request

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

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

@Component({ /\* ... \*/ })

export class AppComponent {

/\* ... \*/

search() {

const request = this.searchService.search(this.searchField.value)

.subscribe(

result => { this.result = result.artists.items; },

err => { this.errorMessage = err.message; },

() => { console.log('Completed'); }

);

request.unsubscribe();

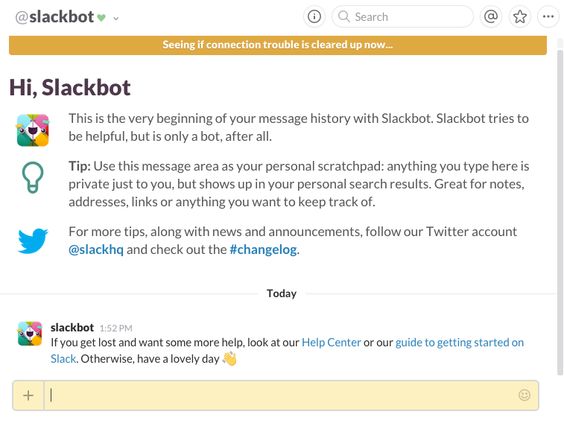
}

}

[View Example](http://plnkr.co/edit/XQL8v9?p=preview)

## Retry

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

Figure: Retry example from Slack

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

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

import { Http } from '@angular/http';

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

import { Observable } from 'rxjs/Rx';

@Injectable()

export class SearchService {

constructor(private http: Http) {}

search(term: string) {

let tryCount = 0;

return this.http.get('https://api.spotify.com/v1/dsds?q=' + term + '&type=artist')

.map(response => response.json())

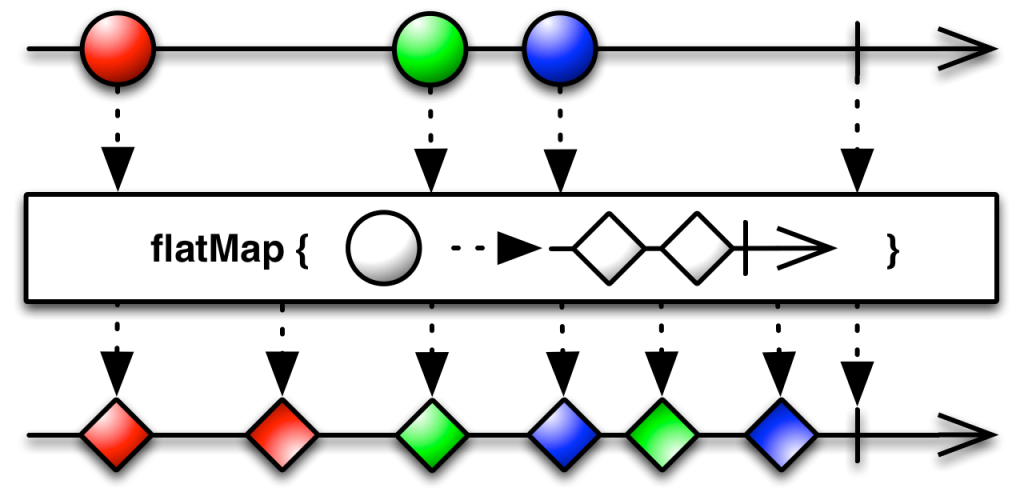
.retry(3);

}

}

[View Example](http://plnkr.co/edit/zSAWwV?p=preview)

# 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](http://jsbin.com/nutegi/36/edit?js,console)
* [A stream of arrays](http://jsbin.com/lerake/3/edit?js,console)
* [Filter the items from each event](http://jsbin.com/widadiz/2/edit?js,console)
* [Stream of filtered items](http://jsbin.com/reyoja/2/edit?js,console)
* [Filter + map simplified with flatMap](http://jsbin.com/sahiye/2/edit?js,console)

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

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

app/services/search.service.ts

import {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.component.ts

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

import { FormControl,

FormGroup,

FormBuilder } from '@angular/forms';

import { SearchService } from './services/search.service';

import 'rxjs/Rx';

@Component({

selector: 'app-root',

template: `

<form [formGroup]="coolForm"><input formControlName="search" placeholder="Search Spotify artist"></form>

<div \*ngFor="let artist of result">

{{artist.name}}

</div>

`

})

export class AppComponent {

searchField: FormControl;

coolForm: FormGroup;

constructor(private searchService:SearchService, private fb:FormBuilder) {

this.searchField = new FormControl();

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

this.searchField.valueChanges

.debounceTime(400)

.flatMap(term => this.searchService.search(term))

.subscribe((result) => {

this.result = result.artists.items

});

}

}

[View Example](http://plnkr.co/edit/L6CLXo?p=preview)

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

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

# Enhancing Search with switchMap

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

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

## A Quick Example

Consider a situation where we first type in the letters ABC, and suppose the string ABC is actually a special string where it will take the server a few extra seconds to reply.

Meanwhile, after we paused for a bit (more than the debounce time), we decide to type in another letter (the letter X) and our app sends a request to the server for the string ABCX. Since ABCX is not considered a special string, the server replies very quickly and our app sets the suggestions for ABCX.

A few seconds later, however, the server finally replies with the response for the ABC string, and our app receives that response and sets the search suggestions for ABC, overwriting the suggestions for the ABCX string, even though the request for that actually came afterwards.

Here is a simple diagram to illustrate the issue:

// A1: Request for `ABC`

// A2: Response for `ABC`

// B1: Request for `ABCX`

// B2: Response for `ABCX`

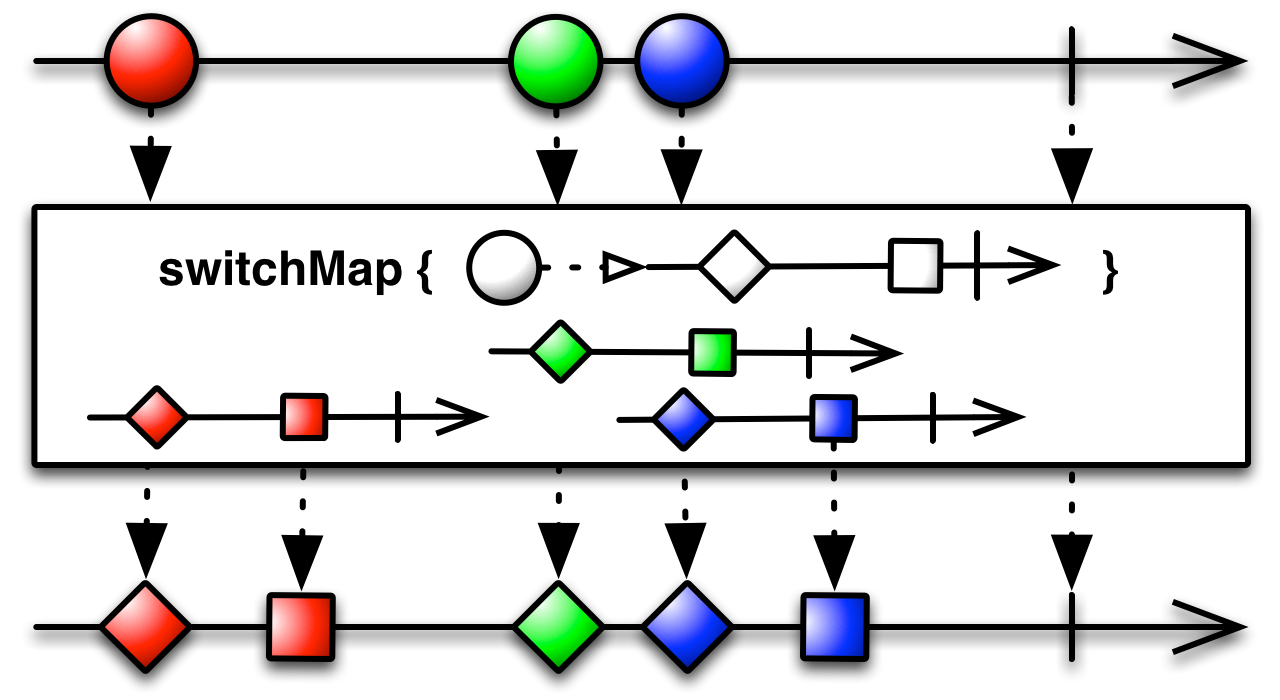
--A1----------A2-->

------B1--B2------>

You can see that A2 arrives after B2 even though the A1 request began first. This will end up showing the wrong results to the user. "If the last input in the search was ABCX why am I seeing the results for ABC?" the user might think. To get around this problem we need to replace flatMap with switchMap.

## What is switchMap?

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

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

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

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

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

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

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

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

## Enhanced Search with switchMap

Here is the revised component using switchMap instead of flatMap.

app/app.component.ts

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

import { FormControl,

FormGroup,

FormBuilder } from '@angular/forms';

import { SearchService } from './services/search.service';

import 'rxjs/Rx';

@Component({

selector: 'app-root',

template: `

<form [formGroup]="coolForm"><input formControlName="search" placeholder="Search Spotify artist"></form>

<div \*ngFor="let artist of result">

{{artist.name}}

</div>

`

})

export class AppComponent {

searchField: FormControl;

coolForm: FormGroup;

constructor(private searchService:SearchService, private fb:FormBuilder) {

this.searchField = new FormControl();

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

this.searchField.valueChanges

.debounceTime(400)

.switchMap(term => this.searchService.search(term))

.subscribe((result) => {

this.result = result.artists.items

});

}

}

[View Example](http://plnkr.co/edit/FYLTcx?p=preview)

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

## Further Resources

* [SwitchMap Examples](https://www.learnrxjs.io/operators/transformation/switchmap.html)
* [Egghead Video Tutorial on SwitchMap](https://egghead.io/lessons/rxjs-starting-a-stream-with-switchmap?course=step-by-step-async-javascript-with-rxjs)
* [RxJS Documentation for SwitchMap](http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html#instance-method-switchMap)

## Requests as Promises

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

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

import { Http } from '@angular/http';

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

import 'rxjs/add/operator/map';

import 'rxjs/add/operator/toPromise';

@Injectable()

export class SearchService {

constructor(private http: Http) {}

search(term: string) {

return this.http

.get(`https://api.spotify.com/v1/search?q=${term}&type=artist`)

.map((response) => response.json())

.toPromise();

}

}

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

@Component({ /\* ... \*/ })

export class AppComponent {

/\* ... \*/

search() {

this.searchService.search(this.searchField.value)

.then((result) => {

this.result = result.artists.items;

})

.catch((error) => console.error(error));

}

}

**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, **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.x vs Angular 2

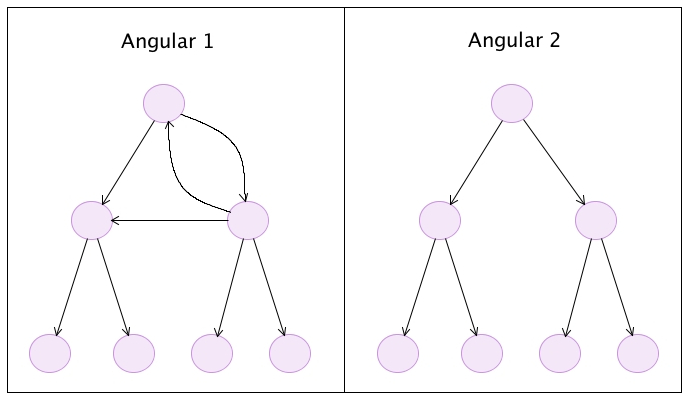
Another difference between both versions of the framework is the way the nodes of an application (directives or components) are checked to see if the DOM needs to be updated.

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, 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.x 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 AppComponent which holds a reference to the movie with buttons to perform some actions.

Our AppComponent 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/app.component.ts

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

import {MovieComponent} from './movie.component';

import {Actor} from './actor.model';

@Component({

selector: 'app-root',

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>

<app-movie [title]="title" [actor]="actor"></app-movie>`

})

export class AppComponent {

slogan = 'Just movie information';

title = 'Terminator 1';

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

export class Actor {

constructor(

public firstName: string,

public lastName: string) {}

}

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

app/movie.component.ts

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

import { Actor } from './actor.model';

@Component({

selector: 'app-movie',

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;

}

[View Example](http://plnkr.co/edit/RKfTH5xSEA9KhuY9quSa?p=preview)

# Change Detector Classes

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

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

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

In our example, because we only have one instance of the AppComponent and the MovieComponent, we will have only one instance of the AppComponent\_ChangeDetector and the MovieComponent\_ChangeDetector.

The code snippet below is a conceptual model of how the AppComponent\_ChangeDetector class might look.

class AppComponent\_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 AppComponent 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 AppComponent instance that it's supposed to "watch". When the change detection process wants to know if our AppComponent instance has changed, it will run the method detectChangespassing 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](http://plnkr.co/edit/yBekf1Do7UeB8F4EQzxU?p=preview)

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 AppComponent, 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 AppComponenttemplate 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 !== previousActorwill 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 <app-movie> inside our AppComponent'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](http://plnkr.co/edit/yjr8R6LhWpOKcGnAwYNS?p=preview)

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 changeActorPropertiesis called and the properties of the actor object get updated.

When the change detection analyzes the properties bound to the AppComponent'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 AppComponent'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 actorobject, 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](https://facebook.github.io/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 AppComponent we import the library and use it to create an actor object as an immutable.

app/app.component.ts

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

import { MovieComponent } from './movie.component';

import \* as Immutable from 'immutable';

@Component({

selector: 'app-root',

template: `

<h1>MovieApp</h1>

<p>{{ slogan }}</p>

<button type="button" (click)="changeActor()">

Change Actor

</button>

<app-movie [title]="title" [actor]="actor"></app-movie>`

})

export class AppComponent {

slogan = 'Just movie information';

title = 'Terminator 1';

actor = 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 actorobject 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

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

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

import \* as Immutable from 'immutable';

@Component({

selector: 'app-movie',

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](http://plnkr.co/edit/0Qp7ynAcZCqcv67OvsSD?p=preview)

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)](https://www.youtube.com/watch?v=jvKGQSFQf10)
* [Angular API Docs: ChangeDetectionStrategy](https://angular.io/docs/ts/latest/api/core/index/ChangeDetectionStrategy-enum.html)
* [Victor Savkin Blog: Change Detection in Angular 2](http://victorsavkin.com/post/110170125256/change-detection-in-angular-2)
* [Victor Savkin Blog: Two Phases of Angular 2 Applications](http://victorsavkin.com/post/114168430846/two-phases-of-angular-2-applications)
* [Victor Savkin Blog: Angular, Immutability and Encapsulation](http://victorsavkin.com/post/133936129316/angular-immutability-and-encapsulation)

# Zones

[Zone.js](https://github.com/angular/zone.js) provides a mechanism, called zones, for encapsulating and intercepting asynchronous activities in the browser (e.g. setTimeout, , promises).

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

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

It is important to know that Zone.js accomplishes these various interceptions by [Monkey Patching](https://en.wikipedia.org/wiki/Monkey_patch) common methods and elements in the browser, e.g. setTimeout and HTMLElement.prototype.onclick. These interceptions can cause unexpected behaviour between external libraries and Angular. In some cases, it may be preferential to execute third party methods outside of Angular's zone (see below).

## In The Zone

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

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

To subscribe to these we inject **NgZone** into our components/services/etc. and subscribe to the public Observables.

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

@Injectable()

export class OurZoneWatchingService() {

constructor(private ngZone: NgZone) {

this.ngZone.onStable.subscribe(this.onZoneStable);

this.ngZone.onUnstable.subscribe(this.onZoneUnstable);

this.ngZone.onError.subscribe(this.onZoneError);

}

onZoneStable() {

console.log('We are stable');

}

onZoneUnstable() {

console.log('We are unstable');

}

onZoneError(error) {

console.error('Error', error instanceof Error ? error.message : error.toString());

}

}

Subscribing to these can help you determine if your code is unexpectedly triggering change detection as a result of operations that do not affect application state.

## Change Detection

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

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

In this [example](http://plnkr.co/edit/d3KGMh?p=preview) you will see in the log what happens with Angular's zone when code is run in and outside of it.

You will notice that in both cases clicking the button causes the Angular zone to become unstable due to Zone.js patching and watching **HTMLElement.prototype.onclick**, however the **setInterval** executing outside of Angular's zone does not affect its stability and does not trigger change detection.

## Debugging

Generally, exceptions thrown during a chain of asynchronous events will only include the current method in their stack trace.

With Zone.js tracking all of our asynchronous calls it can provide us a longer, more detailed, stack trace of the events and calls that occurred leading up to our exception.

To enable long stack traces in development, you should include the **long-stack-trace-zone** module in your code. It is a good idea not to include this in your production build but Angular will skip setting up longer stack traces when in production mode (enableProdMode from @angular/core).

Angular will take care of forking and extending its own zone to display more meaningful stack traces.

if (\_\_PRODUCTION\_\_) {

enableProdMode();

} else {

require('zone.js/dist/long-stack-trace-zone');

}

With the following code, we start by calling startAsync which triggers a chain of setTimeouts leading up to an uncaught error.

function startAsync() {

setTimeout(stepOne, 100);

}

function stepOne() {

setTimeout(stepTwo, 100);

}

function stepTwo() {

throw new Error('Finished');

}

### Simple Stack trace

This is a typical stack trace that you would see in this scenario, without Zone, showing only the function where the unhandled exception occurred.

Uncaught Error: Finished(…)

stepTwo @ debugging.html:28

### Detailed "Long" Stack trace

In the stack trace below, you can see the order of events that occurred within this asynchronous chain of function calls, '>>' has been added to point out our functions.

You'll notice this stack trace includes much more information, including Zone's own task management (e.g. onScheduleTask), as well as the time that elapsed between when the function was queued and when it was executed.

Having this longer stack trace may aide you with debugging which feature of Angular your code is interacting with asynchronously and help you narrow down where your problem is occuring.

debugging.html:16 Error: Finished

>> at stepTwo (http://localhost:3030/examples/debugging.html:28:15)

at ZoneDelegate.invokeTask (http://localhost:3030/node\_modules/zone.js/dist/zone.js:265:35)

at Zone.runTask (http://localhost:3030/node\_modules/zone.js/dist/zone.js:154:47)

at ZoneTask.invoke (http://localhost:3030/node\_modules/zone.js/dist/zone.js:335:33)

at data.args.(anonymous function) (http://localhost:3030/node\_modules/zone.js/dist/zone.js:970:25)

------------- Elapsed: 101 ms; At: Wed Nov 16 2016 08:23:17 GMT-0500 (EST) -------------

at Object.onScheduleTask (http://localhost:3030/node\_modules/zone.js/dist/long-stack-trace-zone.js:83:18)

at ZoneDelegate.scheduleTask (http://localhost:3030/node\_modules/zone.js/dist/zone.js:242:49)

at Zone.scheduleMacroTask (http://localhost:3030/node\_modules/zone.js/dist/zone.js:171:39)

at http://localhost:3030/node\_modules/zone.js/dist/zone.js:991:33

at setTimeout (eval at createNamedFn (http://localhost:3030/node\_modules/zone.js/dist/zone.js:927:17), <anonymous>:3:37)

>> at stepOne (http://localhost:3030/examples/debugging.html:23:9)

at ZoneDelegate.invokeTask (http://localhost:3030/node\_modules/zone.js/dist/zone.js:265:35)

at Zone.runTask (http://localhost:3030/node\_modules/zone.js/dist/zone.js:154:47)

------------- Elapsed: 105 ms; At: Wed Nov 16 2016 08:23:17 GMT-0500 (EST) -------------

at Object.onScheduleTask (http://localhost:3030/node\_modules/zone.js/dist/long-stack-trace-zone.js:83:18)

at ZoneDelegate.scheduleTask (http://localhost:3030/node\_modules/zone.js/dist/zone.js:242:49)

at Zone.scheduleMacroTask (http://localhost:3030/node\_modules/zone.js/dist/zone.js:171:39)

at http://localhost:3030/node\_modules/zone.js/dist/zone.js:991:33

at setTimeout (eval at createNamedFn (http://localhost:3030/node\_modules/zone.js/dist/zone.js:927:17), <anonymous>:3:37)

>> at startAsync (http://localhost:3030/examples/debugging.html:33:9)

at ZoneDelegate.invoke (http://localhost:3030/node\_modules/zone.js/dist/zone.js:232:26)

at Zone.run (http://localhost:3030/node\_modules/zone.js/dist/zone.js:114:43)

# Advanced Angular

Angular 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 application, let's dig deeper into some of these entities and take advantage of what we know.

# Angular Directives

Angular 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: 'app-visit-rangle',

template: `

<button

type="button"

(click)="visitRangle()">

Visit Rangle

</button>

`

})

export class VisitRangleComponent {

visitRangle() {

location.href = 'https://rangle.io';

}

}

[View Example](https://plnkr.co/edit/9ANDvP9C1p2jSZW2s4LX?p=preview)

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: `[appConfirm]`

})

export class ConfirmDirective {

@HostListener('click', ['$event'])

confirmFirst(event: Event) {

return window.confirm('Are you sure you want to do this?');

}

}

[View Example](https://plnkr.co/edit/KMfnzrmSx0ywKp6ztaNN?p=preview)

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.

template: `

<button

type="button"

(click)="visitRangle()"

appConfirm>

Visit Rangle

</button>

`

[View Example](https://plnkr.co/edit/KMfnzrmSx0ywKp6ztaNN?p=preview)

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: `[appConfirm]`

})

export class ConfirmDirective {

@Input() appConfirm = () => {};

@HostListener('click', ['$event'])

confirmFirst() {

const confirmed = window.confirm('Are you sure you want to do this?');

if(confirmed) {

this.appConfirm();

}

}

}

[View Example](https://plnkr.co/edit/OBuN06R0hmcnpGu01Z7I?p=preview)

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"

[appConfirm]="visitRangle">

Visit Rangle

</button>

[View Example](https://plnkr.co/edit/OBuN06R0hmcnpGu01Z7I?p=preview)

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: `[appConfirm]`

})

export class ConfirmDirective {

@Input() appConfirm = () => {};

@Input() confirmMessage = 'Are you sure you want to do this?';

@HostListener('click', ['$event'])

confirmFirst() {

const confirmed = window.confirm(this.confirmMessage);

if(confirmed) {

this.appConfirm();

}

}

}

[View Example](https://plnkr.co/edit/S8pkKyrdF4jB7HlVQ76n?p=preview)

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"

[appConfirm]="visitRangle"

confirmMessage="Click ok to visit Rangle.io!">

Visit Rangle

</button>

[View Example](https://plnkr.co/edit/S8pkKyrdF4jB7HlVQ76n?p=preview)

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

# Listening to an Element Host

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

@Directive({

selector: '[appMyDirective]'

})

class MyDirective {

@HostListener('click', ['$event'])

onClick() {}

}

We can also respond to external events, such as from window or document, by adding the target in the listener.

@Directive({

selector: `[appHighlight]`

})

export class HighlightDirective {

constructor(private el: ElementRef, private renderer: Renderer) { }

@HostListener('document:click', ['$event'])

handleClick(event: Event) {

if (this.el.nativeElement.contains(event.target)) {

this.highlight('yellow');

} else {

this.highlight(null);

}

}

highlight(color) {

this.renderer.setElementStyle(this.el.nativeElement, 'backgroundColor', color);

}

}

[View Example](https://plnkr.co/edit/iJvMpPYDQmiwqvUTKSU8?p=preview)

Although less common, we can also use @HostListener if we'd like to register listeners on the host element of a Component.

## Host Elements

The concept of a host element applies to both directives and components.

For a directive, the concept is fairly straight forward. Whichever template tag you place your directive attribute on is considered the host element. If we were implementing the HighlightDirective above like so:

<div>

<p appHighlight>

<span>Text to be highlighted</span>

</p>

</div>

The <p> tag would be considered the host element. If we were using a custom TextBoxComponent as the host, the code would look like this:

<div>

<app-my-text-box appHighlight>

<span>Text to be highlighted</span>

</app-my-text-box>

</div>

In the context of a Component, the host element is the tag that you create through the selector string in the component configuration. For the TextBoxComponent in the example above, the host element in the context of the component class would be the <app-my-text-box> tag.

# Setting Properties with a Directive

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

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

For example, lets say that we want to create a directive for buttons that dynamically adds a class when we press on it. That could look something like:

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

@Directive({

selector: '[appButtonPress]'

})

export class ButtonPressDirective {

@HostBinding('attr.role') role = 'button';

@HostBinding('class.pressed') isPressed: boolean;

@HostListener('mousedown') hasPressed() {

this.isPressed = true;

}

@HostListener('mouseup') hasReleased() {

this.isPressed = false;

}

}

Notice that for both use cases of @HostBinding we are passing in a string value for which property we want to affect. If we don't supply a string to the decorator, then the name of the class member will be used instead.

In the first @HostBinding, we are statically setting the role attribute to button. For the second example, the pressed class will be applied when isPressed is true.

Tip: Though less common, @HostBinding can also be applied to Components if required.

# Creating a Structural Directive

We'll create an appDelay 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: '[appDelay]'

})

export class DelayDirective {

constructor(

private templateRef: TemplateRef<any>,

private viewContainerRef: ViewContainerRef

) { }

@Input()

set appDelay(time: number): void { }

}

[View Example](https://plnkr.co/edit/80AGn8bR4CiyH0ceP8ws?p=preview)

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 performs the binding. We bind delay in exactly the same way as the Angular built-in structural directives.

@Component({

selector: 'app-root',

template: `

<div \*ngFor="let item of [1,2,3,4,5,6]">

<card \*appDelay="500 \* item">

{{item}}

</card>

</div>

`

})

export class AppComponent {

}

[View Example](https://plnkr.co/edit/80AGn8bR4CiyH0ceP8ws?p=preview)

Notice that no content is being rendered however. This is due to Angular simulating the html templatetag 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: '[appDelay]'

})

export class DelayDirective {

constructor(

private templateRef: TemplateRef<any>,

private viewContainerRef: ViewContainerRef

) { }

@Input()

set appDelay(time: number): void {

setTimeout(

() => {

this.viewContainerRef.createEmbeddedView(this.templateRef);

},

time);

}

}

[View Example](https://plnkr.co/edit/UIyFaG6VyHeeGlCKM76L?p=preview)

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 appDelay 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.

export class DelayContext {

constructor(private loadTime: number) { }

}

@Directive({

selector: '[appDelay]'

})

export class DelayDirective {

constructor(

private templateRef: TemplateRef<DelayContext>,

private viewContainerRef: ViewContainerRef

) { }

@Input()

set appDelay(time: number): void {

setTimeout(

() => {

this.viewContainerRef.createEmbeddedView(

this.templateRef,

new DelayContext(performance.now())

);

},

time);

}

}

[View Example](https://plnkr.co/edit/GmjxiDSbv78zbBFqw8yv?p=preview)

We've made a few changes to our appDelay 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 appDelay, we access the loadTime context variable in the same way we access variables in ngFor.

@Component({

selector: 'app-root',

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](https://plnkr.co/edit/pSv4JsGhxxwzJOh9qSNj?p=preview)

**AoT in Angular**

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

The initial compiler in Angular 1.x and Angular 2 is called JiT (Just-in-Time) compiler. As for AoT, it stands for the Ahead-of-Time compiler that was recently introduced in Angular. Compared to the JiT compilation performed by Angular at run-time, AoT provides a smaller bundle with faster rendering in the browser. Using AoT, we can reduce the [angular2-starter](https://github.com/rangle/angular2-starter/pull/149) to 428.8 kb compared to the original 1.2 MB and reduce loading times by skipping compilation in the browser.

| **Characteristic** | **JiT** | **AoT** |
| --- | --- | --- |
| Compilation target | Browser | Server |
| Compilation context | Runtime | Build |
| Bundle size | Huge (~1.2 MB) | Smaller (~400 KB) |
| Execution Performance | - | Better |
| Startup time | - | Shorter |

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

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

Besides, AoT compiler also inlines HTML templates and CSS files and help reduce the amount of asynchronous requests sent by the application. (Note: this caused a config bug that we will mention in a latter section)

# AoT limitations

However, AoT is not perfect. The main limitation is that AoT, due to the way it compiles the raw code, cannot be used with common code patterns, for example, default exports from modules, template literals for templates, and functions in providers, routes, or declarations. Currently, we do not have a complete list of "AoT Do's and Don'ts" and the Angular team has not released anything regarding this issue. Rangle made its own list [here](https://github.com/rangle/angular-2-aot-sandbox) and also provides a sandbox for testing features with AoT.

Another problem with AoT is that when the application reaches certain complexity, the AoT bundle compared to JiT bundle can actually takes up more space. As an trade-off of having a simpler logic for browser (therefore faster rendering speed), the code generated by AoT is actually more verbose compared to "dynamic" JiT.

# AoT Configuration

To enable AoT in Angular, there are two possible methods:

* using ngc directly
* using @ngtools/webpack

We recommend the second way because it fits the Angular + Webpack toolchain the best. One problem of using raw ngc is that ngc tries to inline CSS while lacking necessary context. For example, the @import 'basscss-basic' statement in index.css would cause an error like Error: Compilation failed. Resource file not found with ngc. It lacks the information that 'basscss-basic' is actually a node module inside node\_modules. On the other hand, @ngtools/webpack provides AotPlugin and loader for Webpack which shares the context with other loaders/plugins. So when ngc is called by @ngtools/webpack, it can gather necessary informations from other plugins like postcss-import to correctly understand things like @import 'basscss-basic'.

## Config @ngtools/webpack

First, get @ngtools/webpack from npm and save it as a development dependency:

npm install -D @ngtools/webpack

Then, inside the Webpack configuration file (usually named as webpack.config.js), add following code:

import {AotPlugin} from '@ngtools/webpack'

exports = { /\* ... \*/

module: {

rules: [

{

test: /\.ts$/,

loader: '@ngtools/webpack',

}

]

},

plugins: [

new AotPlugin({

tsConfigPath: 'path/to/tsconfig.json',

entryModule: 'path/to/app.module#AppModule'

})

]

}

Here @ngtools/webpack replaces other typescript loader like ts-loader or awesome-typescript-loader. It works with AotPlugin together to enable AoT compilation. More details can be found [here](https://github.com/angular/angular-cli/tree/master/packages/webpack).

(Note, for project generated by angular-cli, turning on AoT can be simple as ng build --aot, but since angular-cli does not allow customized webpack configuration for complex use cases, it may be insufficient.)

**Immutable.js**

[Immutable.js](https://facebook.github.io/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.

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:

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.