

RANGLE.IO

Rangle's Angular 2 Training Book



Published with
GitBook



Table of Contents

Introduction	1.1
License	1.2
Why Angular?	1.3
The Architect's Guide to Angular	2.1
Creating Functional Forms	2.1.1
EcmaScript 6 and TypeScript Features	3.1
ES6	3.1.1
Classes	3.1.1.1
Refresher on 'this'	3.1.1.2
Arrow Functions	3.1.1.3
Template Strings	3.1.1.4
Inheritance	3.1.1.5
Delegation	3.1.1.6
Constants and Block Scoped Variables	3.1.1.7
...spread and ...rest	3.1.1.8
Destructuring	3.1.1.9
Modules	3.1.1.10
TypeScript	3.1.2
Getting Started With TypeScript	3.1.2.1
Working With tsc	3.1.2.2
Typings	3.1.2.3
Linting	3.1.2.4
TypeScript Features	3.1.2.5
TypeScript Classes	3.1.2.6
Interfaces	3.1.2.7
Shapes	3.1.2.8
Type Inference	3.1.2.9
Type Keyword	3.1.2.10
Decorators	3.1.2.11
Property Decorators	3.1.2.12

Class Decorators	3.1.2.13
Parameter Decorators	3.1.2.14
The JavaScript Toolchain	4.1
Source Control: git	4.1.1
The Command Line	4.1.2
Command Line JavaScript: NodeJS	4.1.3
Back-End Code Sharing and Distribution: npm	4.1.4
Module Loading, Bundling and Build Tasks: Webpack	4.1.5
Chrome	4.1.6
Bootstrapping an Angular Application	5.1
Understanding the File Structure	5.1.1
Bootstrapping Providers	5.1.2
Components in Angular	6.1
Creating Components	6.1.1
Application Structure with Components	6.1.2
Passing Data into a Component	6.1.2.1
Responding to Component Events	6.1.2.2
Using Two-Way Data Binding	6.1.2.3
Accessing Child Components from Template	6.1.2.4
Projection	6.1.3
Structuring Applications with Components	6.1.4
Using Other Components	6.1.5
Directives	7.1
Attribute Directives	7.1.1
NgStyle Directive	7.1.1.1
NgClass Directive	7.1.1.2
Structural Directives	7.1.2
NgIf Directive	7.1.2.1
NgFor Directive	7.1.2.2
NgSwitch Directives	7.1.2.3
Using Multiple Structural Directives	7.1.2.4
Advanced Components	8.1
Component Lifecycle	8.1.1
Accessing Other Components	8.1.2

View Encapsulation	8.1.3
ElementRef	8.1.4
Observables	9.1
Using Observables	9.1.1
Error Handling	9.1.2
Disposing Subscriptions and Releasing Resources	9.1.3
Observables vs Promises	9.1.4
Using Observables From Other Sources	9.1.5
Observables Array Operations	9.1.6
Cold vs Hot Observables	9.1.7
Summary	9.1.8
Angular Dependency Injection	10.1
What is DI?	10.1.1
DI Framework	10.1.2
Angular's DI	10.1.3
@Inject() and @Injectable	10.1.3.1
Injection Beyond Classes	10.1.3.2
Avoiding Injection Collisions: OpaqueToken	10.1.3.3
The Injector Tree	10.1.3.4
Http	11.1
Making Requests	11.1.1
Catching Rejections	11.1.2
Catch and Release	11.1.2.1
Cancel a Request	11.1.2.2
Retry	11.1.2.3
Search with flatMap	11.1.3
Enhancing Search with switchMap	11.1.4
Requests as Promises	11.1.5
Change Detection	12.1
Change Detection Strategies in Angular 1 vs Angular 2	12.1.1
How Change Detection Works	12.1.2
Change Detector Classes	12.1.3
Change Detection Strategy: OnPush	12.1.4

Enforcing Immutability	12.1.5
Additional Resources	12.1.6
Zone.js	13.1
Advanced Angular	14.1
Directives	14.1.1
Creating an Attribute Directive	14.1.1.1
Listening to an Element Host	14.1.1.1.1
Setting Properties in a Directive	14.1.1.1.2
Creating a Structural Directive	14.1.1.2
View Containers and Embedded Views	14.1.1.2.1
Providing Context Variables to Directives	14.1.1.2.2
AoT	14.1.2
AoT limitations	14.1.2.1
AoT Configuration	14.1.2.2
Immutable.js	15.1
What is Immutability?	15.1.1
The Case for Immutability	15.1.2
JavaScript Solutions	15.1.3
Object.assign	15.1.3.1
Object.freeze	15.1.3.2
Immutable.js Basics	15.1.4
Immutable.Map	15.1.4.1
Map.merge	15.1.4.1.1
Nested Objects	15.1.4.2
Deleting Keys	15.1.4.2.1
Maps are Iterable	15.1.4.2.2
Immutable.List	15.1.4.3
Performance and Transient Changes	15.1.4.4
Official Documentation	15.1.4.5
Pipes	16.1
Using Pipes	16.1.1
Custom Pipes	16.1.2
Stateful Pipes	16.1.3
Forms	17.1

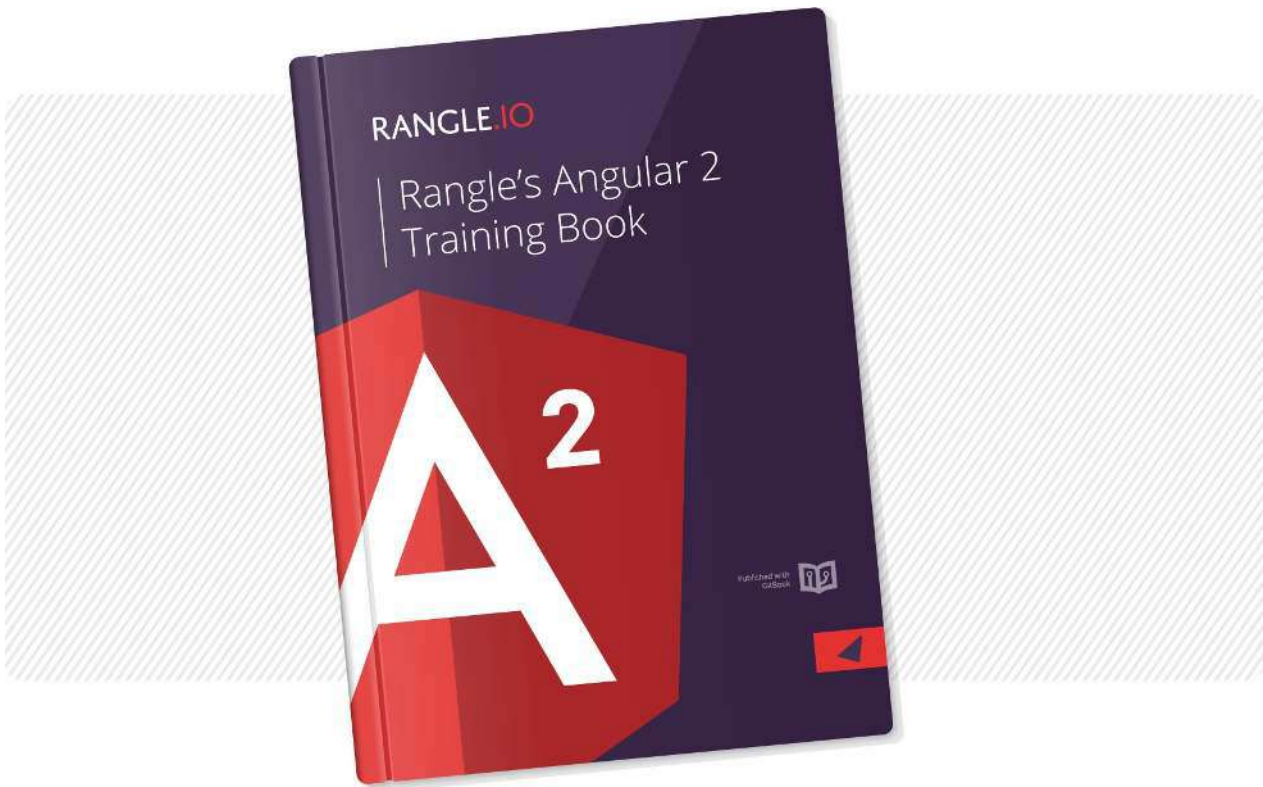
Getting Started	17.1.1
Template-Driven Forms	17.1.2
Nesting Form Data	17.1.2.1
Using Template Model Binding	17.1.2.2
Validating Template-Driven Forms	17.1.2.3
Reactive/Model-Driven Forms	17.1.3
FormBuilder Basics	17.1.3.1
Validating FormBuilder Forms	17.1.3.2
FormBuilder Custom Validation	17.1.3.3
Visual Cues for Users	17.1.4
Modules	18.1
What is an Angular Module?	18.1.1
Adding Components, Pipes and Services to a Module	18.1.2
Creating a Feature Module	18.1.3
Directive Duplications	18.1.4
Lazy Loading a Module	18.1.5
Lazy Loading and the Dependency Injection Tree	18.1.6
Shared Modules and Dependency Injection	18.1.7
Sharing the Same Dependency Injection Tree	18.1.8
Routing	19.1
Why Routing?	19.1.1
Configuring Routes	19.1.2
Redirecting the Router to Another Route	19.1.3
Defining Links Between Routes	19.1.4
Dynamically Adding Route Components	19.1.5
Using Route Parameters	19.1.6
Defining Child Routes	19.1.7
Controlling Access to or from a Route	19.1.8
Passing Optional Parameters to a Route	19.1.9
Using Auxiliary Routes	19.1.10
State Management	20.1
Redux and @ngrx	20.1.1
Adding @ngrx to your Project	20.1.1.1

Defining your Main Application State	20.1.1.2
Example Application	20.1.1.3
Reading your Application State using Selectors	20.1.1.4
Actions	20.1.1.5
Modifying your Application State by Dispatching Actions	20.1.1.6
Reducers and Pure Functions	20.1.1.7
Reducers as State Management	20.1.1.8
Creating your Application's Root Reducer	20.1.1.9
Configuring your Application	20.1.1.10
Implementing Components	20.1.1.11
Component Architecture	20.1.1.12
Side Effects	20.1.1.13
Getting More From Redux and @ngrx	20.1.1.14
TDD Testing	21.1
The Testing Toolchain	21.1.1
Test Setup	21.1.2
Filename Conventions	21.1.2.1
Karma Configuration	21.1.2.2
TestBed Configuration (Optional)	21.1.2.3
Typings	21.1.2.4
Executing Test Scripts	21.1.2.5
Simple Test	21.1.3
Using Chai	21.1.4
Testing Components	21.1.5
Verifying Methods and Properties	21.1.5.1
Injecting Dependencies and DOM Changes	21.1.5.2
Overriding Components for Testing	21.1.5.2.1
Testing Asynchronous Actions	21.1.5.3
Refactoring Hard-to-Test Code	21.1.5.4
Testing Services	21.1.6
Testing Strategies for Services	21.1.6.1
Testing HTTP Requests	21.1.6.2
Using MockBackend	21.1.6.2.1
Alternative Mocking Strategy	21.1.6.2.2

Testing JSONP and XHR Back-Ends	21.1.6.2.3
Executing Tests Asynchronously	21.1.6.3
Testing Redux	21.1.7
Testing Simple Actions	21.1.7.1
Testing Complex Actions	21.1.7.2
Testing Reducers	21.1.7.3
Afterthoughts	21.1.7.4
Migrating Angular 1.x Projects to Angular 2	22.1
Migration Prep	22.1.1
Upgrading To Angular 1.3+ Style	22.1.1.1
Using Webpack	22.1.1.2
Migrating To TypeScript	22.1.1.3
Choosing an Upgrade Path	22.1.2
Avoiding Total Conversion	22.1.3
Using ng-metadata (Angular 1.x Using 2 Style)	22.1.4
Bootstrapping ng-metadata	22.1.4.1
Components and Services	22.1.4.2
Using ng-upgrade (Angular 1.x Coexisting With Angular 2)	22.1.5
Order of Operations	22.1.5.1
Replacing Services with TypeScript Classes	22.1.5.2
Bootstrapping ng-upgrade	22.1.5.3
Downgrading Components	22.1.5.4
Upgrading Components	22.1.5.5
Projecting Angular 1 Content into Angular 2 Components	22.1.5.6
Transcluding Angular 2 Components into Angular 1 Directives	22.1.5.7
Injecting Across Frameworks	22.1.5.8
Project Setup	23.1
Webpack	23.1.1
Installation and Usage	23.1.1.1
Loaders	23.1.1.2
Plugins	23.1.1.3
Summary	23.1.1.4
NPM Scripts Integration	23.1.2

Angular CLI	24.1
Setup	24.1.1
Creating a New App	24.1.2
Serving the App	24.1.3
Creating Components	24.1.4
Creating Routes	24.1.5
Creating Other Things	24.1.6
Testing	24.1.7
Linting	24.1.8
CLI Command Overview	24.1.9
Adding Third Party Libraries	24.1.10
Integrating an Existing App	24.1.11
Accessibility in Angular	25.1
Why Make my Application Accessible?	25.1.1
Key Concerns of Accessible Web Applications	25.1.2
Semantic Markup	25.1.2.1
Keyboard Accessibility	25.1.2.2
Visual Assistance	25.1.2.3
Testing for Accessibility	25.1.3
Is my Application Readable?	25.1.3.1
Is my Application Predictable?	25.1.3.2
Is my Application Navigable?	25.1.3.3
Testing with Screen Readers	25.1.3.4
Additional Resources	25.1.4
Internationalization in Angular	26.1
What is the process like and how is involved?	26.1.1
Marking text in our templates	26.1.2
Extracting translation text using the Angular CLI	26.1.3
How to import the completed translation files	26.1.4
Using the AoT Compiler	26.1.4.1
Using the JiT Compiler	26.1.4.2
Glossary	27.1
Further Reading And Reference	27.2

Rangle's Angular Training Book



Over the last three and a half years, Angular has become the leading open source JavaScript application framework for hundreds of thousands of programmers around the world. The "1.x" version of Angular has been widely used and became extremely popular for complex applications. The new Angular 2.x has also announced its [final release version](#).

About Rangle's Angular Training Book

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

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

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

License

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

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

You are free to:

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

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

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

Under the following terms:

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

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

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

Why Angular?

There are many front-end JavaScript frameworks to choose from today, each with its own set of trade-offs. Many people were happy with the functionality that Angular 1.x afforded them. Angular 2 improved on that functionality and made it faster, more scalable and more modern. Organizations that found value in Angular 1.x will find more value in Angular 2.

Angular's Advantages

The first release of Angular provided programmers with the tools to develop and architect large scale JavaScript applications, but its age has revealed a number of flaws and sharp edges. Angular 2 was built on five years of community feedback.

Angular 2 Is Easier

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

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

Angular 2 is a more streamlined framework that allows programmers to focus on simply building JavaScript classes. Views and controllers are replaced with components, which can be described as a refined version of directives. Even experienced Angular programmers are not always aware of all the capabilities of Angular 1.x directives. Angular 2 components are considerably easier to read, and their API features less jargon than Angular 1.x's directives. Additionally, to help ease the transition to Angular 2, the Angular team has added a

`.component` method to Angular 1.5, which has been [back-ported by community member Todd Motto to v1.3](#).

TypeScript

Angular 2 was written in TypeScript, a superset of JavaScript that implements many new ES2016+ features.

By focusing on making the framework easier for computers to process, Angular 2 allows for a much richer development ecosystem. Programmers using sophisticated text editors (or IDEs) will notice dramatic improvements with auto-completion and type suggestions. These

improvements help to reduce the cognitive burden of learning Angular 2. Fortunately for traditional ES5 JavaScript programmers this does *not* mean that development must be done in TypeScript or ES2015: programmers can still write vanilla JavaScript that runs without transpilation.

Familiarity

Despite being a complete rewrite, Angular 2 has retained many of its core concepts and conventions with Angular 1.x, e.g. a streamlined, "native JS" implementation of dependency injection. This means that programmers who are already proficient with Angular will have an easier time migrating to Angular 2 than another library like React or framework like Ember.

Performance and Mobile

Angular 2 was designed for mobile from the ground up. Aside from limited processing power, mobile devices have other features and limitations that separate them from traditional computers. Touch interfaces, limited screen real estate and mobile hardware have all been considered in Angular 2.

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

Angular 2, like React and other modern frameworks, can leverage performance gains by rendering HTML on the server or even in a web worker. Depending on application/site design this isomorphic rendering can make a user's experience *feel* even more instantaneous.

The quest for performance does not end with pre-rendering. Angular 2 makes itself portable to native mobile by integrating with [NativeScript](#), an open source library that bridges JavaScript and mobile. Additionally, the Ionic team is working on an Angular 2 version of their product, providing *another* way to leverage native device features with Angular.

Project Architecture and Maintenance

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

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

Angular 2.x makes use of the ES2015 module system, and modern packaging tools like webpack or SystemJS. Modules are far less coupled to the "Angular way", and it's easier to write more generic JavaScript and plug it into Angular. The removal of minification workarounds and the addition of rigid prescriptions make maintaining existing applications simpler. The new module system also makes it easier to develop effective tooling that can reason better about larger projects.

New Features

Some of the other interesting features in Angular 2 are:

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

Differences Between Angular 1 & 2

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

	Old School Angular 1.x	Angular 1.x Best Practices	Transitional Architecture	Angular 2
Nested scopes ("\$scope", watches)	Used heavily	Avoided	Avoided	Gone
Directives vs controllers	Use as alternatives	Used together	Directives as components	Component directives
Controller and service implementation	Functions	Functions	ES6 classes	ES6 classes
Module system	Angular's modules	Angular's modules	ES6 modules	ES6 modules
Transpiler required	No	No	TypeScript	TypeScript

Software Architecture Using Angular

[Angular](#) has grown steadily more popular since its release, and [a growing number of books](#) are now available to teach programmers how to use it. But building an application involves a lot more than just writing code: in order to be performant and maintainable, the application must have some over-arching architectural plan that ties its pieces together and gives direction to future growth.

To help with this, [Rangle](#) has started to assemble a guide to software architecture using Angular. How should sets of components be organized so that they can easily be tested and re-used? How can Angular be combined with tools like Redux to simplify state management in an asynchronous, event-driven world? How can we design today to make server-side rendering less painful to add tomorrow (and should we be doing SSR at all)? Sooner or later, every growing application has to wrestle with these issues; our aim is to distill expert knowledge so that people can make the right decisions early on.

The first chapter of this new guide, based on work by Daniel Figueiredo and Renee Vratsidis, is now available. We would be grateful for feedback on its content and format, and for suggestions about other topics the community would like to see covered. And if you are interested in contributing, please [get in touch](#): as always, we are smarter together than any one of us is alone.

[Be the first to know when we release the next chapter of the Architect's Guide to Angular.](#)

Creating Functional Forms

Daniel Figueiredo and Renee Vrantsidis

The architecture of a software system is a description of how to decompose it into pieces, how those pieces interact, and how that decomposition enables and constrains further development. In this chapter, we will begin our exploration of the architecture of JavaScript applications by looking at how to use ideas borrowed from pure functional programming to simplify one of the most common tasks in web development: form handling. At first glance these two topics have nothing to do with each other, but paradoxically, acting as if the state of our system cannot be changed actually makes its changes easier to manage.

The Problem

You have probably built web applications that use forms. They are easy enough to manage if you only have a few, or a few dozen, but manual approaches fall apart by the time you have hundreds of distinct forms, each of which can update the application's state in different ways.

Angular provides some good tools for building forms, but these are not enough by themselves in very large programs. In particular, `NgForm` leaves state management entirely in the developer's hands, which quickly results in unmanageable client-side complexity: a modern single-page application (SPA) may have to keep track of hundreds of pieces of information, any of which may need to be updated based on the user's actions *and* kept in sync with permanent storage on the server.

As if that wasn't complicated enough, many of these interactions have to be done asynchronously in order to keep the user's FQ (frustration quotient) down. If, for example, a web page locks up for half a second every time the user types a single character because it's fetching possible auto-completes from the server, the user will quickly take her business somewhere else.

Our problem is therefore this:

How should we manage client-side state in an asynchronous web application that uses forms?

Our solution is:

Use Redux to represent state as a sequence of snapshots, each of which is created in response to a single action.

Redux in a Hurry

Luckily for us, people who use pure functional programming languages have been thinking about these issues for more than 30 years, and have developed some design patterns that we can use in conventional languages like JavaScript. A *pure functional* language is one in which data cannot be changed, or *mutated*, in place: once a value is defined, it is *immutable*. Rather than changing its state, a program written in a pure functional language creates an entirely new state on which to operate.

This may seem wasteful: after all, if I want to paint one wall of a room, I don't have to build an entirely new room that is identical to the original except for the color of the wall in question. However, making state immutable has a lot of advantages when we are dealing with concurrency. In particular, systems are a lot easier to reason about and test if they can't change under our feet. Pure functional programming therefore lets us substitute a cheap, plentiful resource—computer time—for one which is much more expensive—human brain power.

As we will see below, we can often avoid the need to copy all of the state. If we divide it into logically-separate chunks, we can re-use the chunks that *don't* change. In our experience, the amount of data that actually has to be duplicated will grow slowly with the size of the application so long as we think carefully about how to organize it.

The system we will use to illustrate this idea is [Redux](#), which is built on three architectural principles:

1. There is a **single source of truth**, which in practice means that the entire state of the application is stored in a single object tree. A good rule of thumb is that this object tree must hold everything needed to restore the state of the system after shutdown and restart. Storing all this information in one place makes debugging a lot easier, and as we shall see, also simplifies implementation of things like undo/redo.
2. **State is read-only**, i.e., the object tree mentioned above is *never* modified in place. Instead, the only way to change the state is to create an *action* object that describes what change is desired. Button click handlers and I/O callbacks never update the state themselves, but rather create an action and queue it up to be handled sequentially. As a bonus, recording the state changes as objects makes replay, debugging, and testing a lot simpler.
3. **Changes are made by pure functions**. Redux calls these functions *reducers*, since they reduce the combination of an existing state and an action to a new state. Reducers *never* have side effects: they do not modify global variables, write data to disk, or do anything else to change the world around them. This allows programmers to think about their effects one at a time, and makes it easy to combine and re-use them.

As a very short example, suppose we want to implement a traffic light that switches state between red, amber, and green. The state is an object with a single field showing the current color of the light:

```
let state = {color: 'red'};
```

and the reducer cycles between colors:

```
function changeColor(state, action) {
  switch(action.type) {

    case 'NEXT':
      if (state.color == 'red')
        return {color: 'green'};
      else if (state.color == 'green')
        return {color: 'amber'};
      else if (state.color == 'amber')
        return {color: 'red'};

    default:
      return state;
  }
}
```

It's very important that `changeColor` returns the original state without modification when it doesn't recognize the type of the action it is being asked to perform, since this allows us to combine reducer functions without worrying that they will trip over one another. And to simplify our program, we frequently fold the initialization of the state into the definition of the reducer by defining the initial state as the default value for the reducer's first parameter:

```
function changeColor(state = {color: 'red'}, action) {
  switch(action.type) {
    ...as before...
  }
}
```

Once this reducer is defined, our application wraps it up to create an object store, and then sends actions to that store to tell it when to move to the next state:

```
import { createStore } from 'redux';
let store = createStore(changeColor);

for (let i=0; i<10; i++) {
  store.dispatch({type: 'NEXT'});
  console.log(store.getState().color); // red, green, amber, red, ...
}
```

If we want to add new capabilities, we simply update our reducer:

```
function changeColor(state, action) {
  switch(action.type) {

    case 'NEXT':
      ...as before...

    case 'EMERGENCY':
      return {color: 'red'};

    default:
      return state;
  }
}
```

This may seem like a lot of work to manage a single traffic light, but that work pays off as soon as we have to worry about asynchronous updates. For example, if we want to do an emergency test of the light every night at 1:00 am, all we have to do is this:

```
let delay = until(tomorrow() + ONE_HOUR);
setTimeout(() => changeColor(state, {type: 'EMERGENCY'}), delay);
```

When the timeout callback is triggered, Redux will put the traffic light in the required state regardless of what else has gone on or is going on.

Setting Up Forms to Work with Redux

As our running example, we will build a set of forms that allow users to create a wizard for a role-playing game. We created this example for [a talk at NG Conf 2017](#), and the complete source is [available on GitHub](#).

Our overall goals are to take advantage of Redux state management to automatically store our form data in state as form values change, and if possible to use one mechanism to manage all the forms in our application. The first step is to plan the shape of our forms. We can structure the store however we want, but since we know that real applications evolve,

we will define things using TypeScript interfaces rather than relying on specific concrete classes. That will make it easier to swap things in and out for testing later on. (It also encourages us to avoid using `this`, which in turn encourages us to think more functionally.)

```
export interface IForm {
  character: ICharacter;          // our single top-level object (for now)
}

export interface ICharacter {
  name?: string;                // optional name
  bioSummary: IBioSummary;      // biographical information (see below)
  skills: string[];             // list of skills
}

export interface IBioSummary {
  age: number;                  // age in years
  alignment: string;           // good or bad, lawful or chaotic
  race: string;                 // character's species
}
```

These interfaces define the shape of a form in our Redux store, which is the part of the store our reducer will be concerned with in this example. Defining these interfaces does *not* actually create the store—we will do that in a moment—but *does* specify the shape of the objects our reducers will hand back to us.

The form contains a single object representing a character, rather than using the character itself as the store. That way, if we want to add other top-level items in future, we won't need to reorganize existing material. For example, the next version of our application might have this structure:

```
export interface IForm {
  character: ICharacter;        // character info
  equipment: IEquipment;       // character's equipment
}
```

Equally, if we want to add more forms, we won't need to reorganize the existing material, and we can re-use common actions that work for any form.

One consequence of this decision is that every action should contain the path to the particular part of the state it applies to. Agile purists might say that we shouldn't add this until we need it, and if our application consisted of just a login form and a message of the day, they'd be right. However, we have enough experience with applications of this kind to know that we're going to, and adding that to our architecture from the start will save us re-work later.

Having decided on the structure of our store, the next step is to define its initial state. We will do this by creating a literal object that conforms to the `ICharacter` interface, and another that conforms to the `IForm` interface:

```
const characterInitialState: ICharacter = {
  name: undefined,
  bioSummary: {
    age: undefined,
    size: undefined,
    alignment: undefined,
    race: undefined,
  },
  skills: []
};

const initialState: IForm = {
  character: characterInitialState
};
```

We could fold the definition of `characterInitialState` into `initialState` to create one large literal object, but again, since we're likely to add more state information (like the equipment the character is carrying), we have decided that `initialState` will only ever be a list of top-level objects conforming to separately-defined interfaces.

We have defined our initial state in a file of its own to make it easier to find and update. In a real application, it would probably be created programmatically so that we could swap in something else for testing. Note that in real life, it would *not* come from a database: instead, it's the blank slate that would be updated from a database as the application is being bootstrapped. (And yes, that bootstrap process would be implemented as a series of update actions.) However the initial state is created, it should be as empty as possible.

Another early decision is that we will define functions to create actions, and that each action will have two elements: a string called `type` (which Redux requires), and a sub-object called `payload` that will contain the path to the part of the store it applies to and whatever extra values are needed to update the state. Redux doesn't mandate this, but again, experience teaches us that creating literal objects in many different places throughout our program is just as risky as using magic numbers rather than defining a constant and referring to it.

The name `type` is a requirement: Redux mandates it by exporting an `Action` interface defined as:

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

The name `payload` is not required, but is widely used and strongly encouraged, since in some cases we need to add extra info for our actions to be completed:

```
interface PayloadAction extends Action { payload: any }
```

As an example of an action creation function, here's one that saves a form's value:

```
const saveForm = (path, value) => ({
  type: 'SAVE_FORM',
  payload: {
    path: path,
    value: value
  }
});
```

Now that we know the shapes of our store and our action, the first version of the reducer is easy to write. To keep it short, we rely on three helper functions from [Ramda](#), a library of functional utilities for JavaScript:

- `assocPath` : makes a shallow clone of an object, replacing the specified property with the specified value as it does so. (Think of this as "make me a copy of X, but with Y set to Z".)
- `merge` : create a shallow copy of one object with properties merged in from a second object. (This is like `assocPath`, but using a second object to get multiple changes at once.)
- `path` : retrieve the value at a specified location in a structured object.

(We could write replacements for these to avoid depending on Ramda, but we prefer to leverage the work they've put into testing and performance optimization.) With these helpers in hand, our `formReducer` looks like this:

```
import { path, assocPath, merge } from 'ramda';

export function formReducer(state = initialState: IForm, action) {
  switch (action.type) {

    case 'SAVE_FORM':
      return assocPath(
        action.payload.path,
        merge(path(action.payload.path, state), action.payload.value),
        state
      );

    default:
      return state;
  }
}
```

There is actually less going on here than first appears. Working from the outside in, if the action's type is *not* `SAVE_FORM`, then `formReducer` returns the original state without any changes. This means that we can safely chain this reducer together with others that handle other parts of our user interface: as long as they all use distinct keys for their actions, they will watch the state fly by without doing anything we don't want them to.

If the action does have the right type, `formReducer` uses `path` to get the part of the state that needs to be updated, then uses `merge` and `assocPath` to create a shallow copy of the state, replacing that element, *and only that element*, with a different value. The most important word in the previous sentence is "create": at this point, the state that was passed in is thrown away and a new one created. Some parts of the old state are recycled—`assocPath` does a shallow clone—but those are the parts that *weren't* changed, and if other reducers do their jobs properly, they will replace those parts rather than updating them in place when it's their turn to act.

Connecting the Dots

It's now time to turn our attention to the interface that will trigger this state change and reflect any changes to the state triggered by other interface components. Since we're using Angular 2, we will define a class and use the `@Component` decorator to weld some metadata to it:

```

@Component({
  selector: 'character-form',
  template: require('./character-form.html')
})
export class CharacterForm {
  @ViewChild(NgForm) ngForm: NgForm;
  public characterForm: IForm;
  private formSubs;

  constructor(private ngRedux: NgRedux<IAppState>) {}
}

```

There's a lot going on here, so we'll start with a high-level explanation and then dive into details. As the diagram shows, Angular will keep `ngForm` and `characterForm` in sync with each other using its own dark magic, which we will see in a moment. In order to synchronize that with our state, we create a standard observer/observable connection to make `characterForm` watch the character portion of our Redux state.

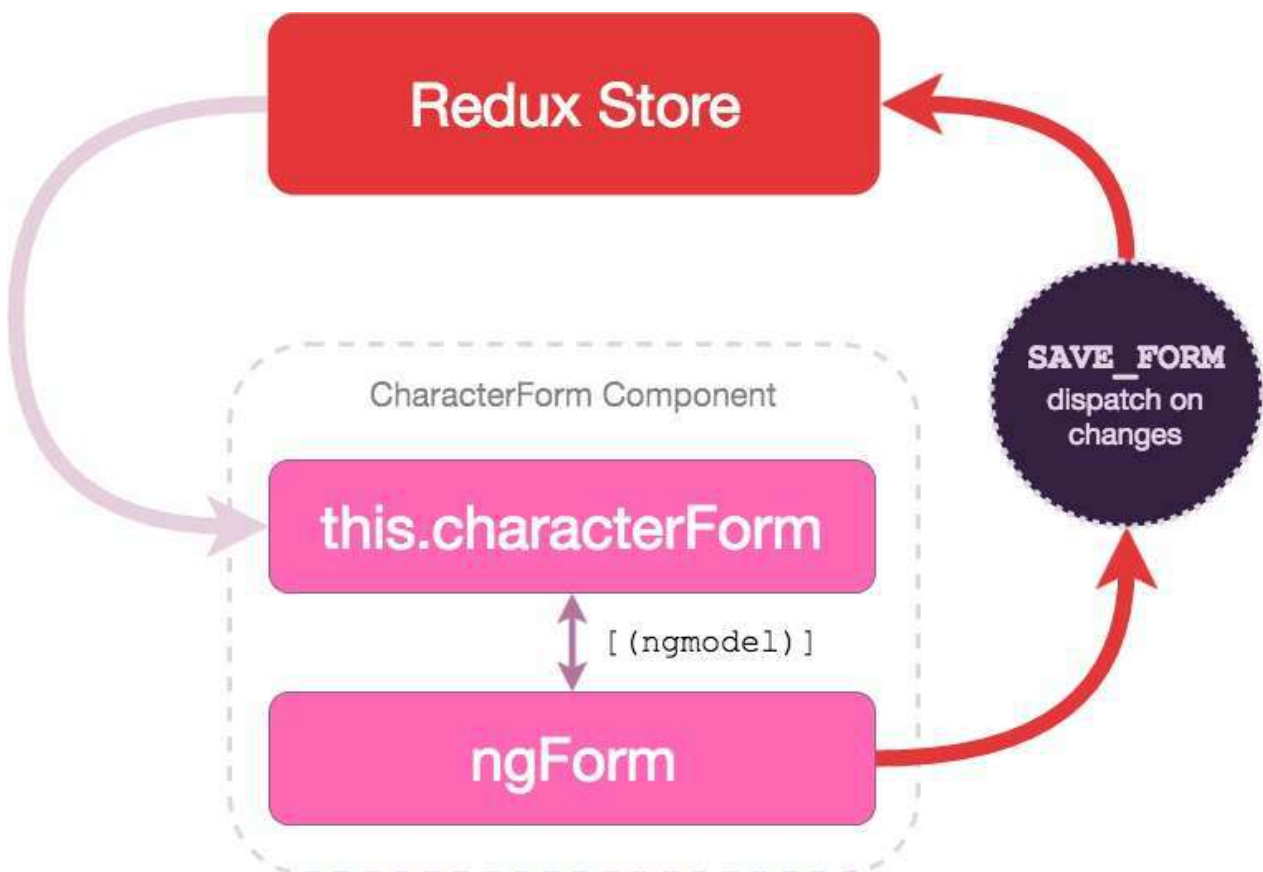


Figure: Redux Flow

Looking once more at the diagram, there is a potential circularity in the updates. Angular avoids an infinite loop by breaking the cycle when `characterForm` and the form's actual internal data are the same.

Looking more closely at the code used to create all of this:

- The `@Component` decorator in the code above tells Angular that this class is used to fill in `<character-form>` elements in the `character-form.html` template snippet.
- Using the `@ViewChild` decorator on `ngForm` gives this class an instance variable that watches a form. We need this because we are going to subscribe to event notifications from that form later on.
- The `characterForm` instance variable is our working copy of the form's state.
- Finally, `private ngRedux: NgRedux<IAppState>` triggers Angular's dependency injection and gives us access to the Redux store. When our application is busy doing other things, our data will live in this store, and when we're testing, we can inject a mock object here to give us more insight.

The `formSubs` instance variable is the odd one out in this class. Its job is to store the observer/observable subscription connecting our state to our form so that we can unsubscribe cleanly when this component is destroyed. Angular 2 will automatically unsubscribe on our behalf, but it's always safer to put our own toys away when we're done playing with them...

Since `CharacterForm` is an Angular component, it needs a template to define how it will be rendered. In keeping with Angular best practices, we will use a template-driven form and bind its inputs to objects retrieved from the Redux state. The first part of that form looks like this:

```
<form #form="ngForm">
  <label for="name">Character Name:</label>
  <input
    type="text"
    name="name"
    [(ngModel)]="characterForm.name">

  <label for="age">Age:</label>
  <input
    type="number"
    name="age"
    [(ngModel)]="characterForm.bioSummary.age">
</form>
```

`ngModel` is a magic word meaning "the value of this field in the form". The expression `[(ngModel)]` therefore makes a form control and binds it to the public data member of our form object. The final step in wiring all of this together is therefore to subscribe to the Redux

state so that whenever it changes, the changes are automatically pushed into the form (which triggers a DOM update and a re-rendering of the page). Equally, since we're listening to the form for changes and writing those to our state by creating and dispatching actions, anything the user does will be reflected in the state. The beauty of this is that if anyone else does an update anywhere, everything will keep itself in sync.

For the sake of simplicity we are synchronizing the whole form object in a single action, which sets the character attribute in the state with a brand new object every time. We could instead sync specific attributes to improve performance by using the `path` mechanism introduced earlier.

It's tempting to put this final piece of wiring in the constructor of our component, but that doesn't work because Angular has to construct several objects before any of them can be wired together. This is a key feature of Angular's architecture (and of the architectures of many other systems): object *construction* and object *initialization* are handled separately to free us from headaches related to cyclic references.

The right place to connect everything is `ngOnInit`, which is called after all the objects in the system have been created but before any of them are used. The `ngOnInit` for `CharacterForm` does the two pieces of wiring described above in an arbitrary order:

```
ngOnInit() {  
  
  // Subscribe to the form in state.  
  this.formSubs = this.ngRedux.select(state => state.form.character)  
    .subscribe(characterFormState => {  
      this.characterForm = characterFormState;  
    });  
  
  // Dispatch an action when the form is changed.  
  this.ngForm.valueChanges.debounceTime(0)  
    .subscribe(change =>  
      this.ngRedux.dispatch(  
        saveForm(  
          change,  
          ['character']  
        )  
      )  
    );  
}
```

The first half of this method says that when the state changes, we want to update the form. We use `ngRedux.select` with a fat arrow function as callback to pick out the character portion of the state, then subscribe to that with a another fat arrow callback that updates the form data (which as explained above triggers redisplay).

The second half of `ngOnInit` does the binding in the opposite direction: whenever the form changes, we dispatch an action created by `saveForm` that has `SAVE_FORM` as the change and `['character']` as the path to the part of the state we want to modify. `ngForm`'s `valueChanges` method automatically gives us a `change` value that contains all of the form values. These values arrive in a JavaScript object that mirrors the structure of the HTML, which is exactly what we want (because we defined the name attributes to get it).

And that's it: every change to our form triggers creation of a new state, and every change to our state triggered by anything else is immediately reflected in our form. We *aren't* saving the state to disk, but that's best left to some kind of middleware (which can be smart about only persisting the bits of state that have actually changed between updates).

Filling in the Gaps

The only meaningful way to assess an architectural decision is to ask whether it makes things comprehensible today and easy to change tomorrow. Given all of the plumbing introduced above, you might think that our architecture falls short on both counts, but it turns out that it actually simplifies application evolution. To see this, let's have a look at how it helps us handle multi-valued forms, which Angular doesn't support out of the box.

As the name suggests, a multi-valued form is one that can have many values at once, such as a multi-select dropdown list. The natural way to store these values is in an array, such as the one shown below for storing a character's skills:

```
character: {  
  skills: ['Drinking', 'Knowing Things'],  
}
```

The first step in adding support for multi-valued fields to our application is to define actions that manipulate arrays—or more precisely, to define functions that create actions that tell Redux how to generate a new array from an old one:

```
const addIntoArray = (path, value) => ({
  type: 'ADD_INDEXED_FORM_VALUE',
  payload: { path, value }
});

const putInArray = (path, value, index) => ({
  type: 'UPDATE_INDEXED_FORM_VALUE',
  payload: { path, value, index }
});

const removeFromArray = (index, path) => ({
  type: 'REMOVE_INDEXED_FORM_VALUE',
  payload: { index, path }
});
```

There's nothing exciting going on here: each function creates an action with a `type` (again, required by Redux) and a `payload` (our own term). To cut down on typing, we make use of the fact that `{a, b}` is a shorthand for `{a: a, b: b}`:

```
let a = 'A';
let b = 'B';
let both = {a, b};
console.log(both);
```

```
{a: 'A', b: 'B'}
```

The payload of the action returned by `addIntoArray` therefore has two keys called `path` and `value`, which are in turn bound to whatever was passed in for the parameters with the corresponding names.

Once we have these actions, the next step is to update our reducer by adding the following case:

```
case 'ADD_INDEXED_FORM_VALUE':
  const lensForProp = lensPath(action.payload.path);
  const propValue = <any[]> view(lensForProp, state);
  return assocPath(
    action.payload.path,
    concat(propValue, [action.payload.value]),
    state
  );
```

Again, we're using Ramda functions to create a new array of skills given an old array and the thing we want to add: `lensForProp` is the location of the skills array in our state, `propValue` is its current value, and `concat` wrapped up in `assocPath` gives us the old skills

with the new one appended. The additions to handle updating and removing skills have a similar shape.

Extending our user interface is equally easy. We start with a helper function that creates and dispatches the required action:

```
addSkill() {
  this.ngRedux.dispatch(addIntoArray({
    value: undefined,
    path: [ 'character', 'skills' ]
  }));
}
```

We have put this in the form component, but neither Angular nor good design principles strictly require this: we could have put it (and similar functions) in a file full of utilities.

Now that we have the functions we need, we can write the HTML needed to put everything in front of the user:

```
<label>Skills:</label>

<div *ngFor="let skillSlot of characterForm.skills; let i = index;">
  <select
    [value]="skillSlot"
    (change)="onSelectSkill($event, i)">
    <option *ngFor="let skill of skills" [value]="skill">
      {{ skill }}
    </option>
  </select>
  <button type="button" (click)="removeSkill(i)">Remove</button>
</div>

<button type="button" (click)="addSkill()">Add skill</button>
```

The last line of this HTML is the one that lets users add skills; the rest is to handle skill display and removal.

The Payoff: Validation

One way to see how these changes pays off is to look at validation, and in particular at the way in which we can use *selectors* to compute data by composing functions, and then use *memoization* to make state changes more efficient.

For those who wish to be more precise, we should distinguish between selectors as a general concept and selectors as implemented by the `reselect` library that we're using. The memoization benefits come from `reselect`, but the general benefit of using a selector for validation is that you can derive the validity of your form from the data you already have without having to dispatch extra actions like `VALIDATE_FIELD`.

Selectors chain functions together and pipe their return values into the last function in the chain. In the simple code below, the selector will return `form.character`:

```
const formStateSelector = (state: IAppState) => state.form;

const characterFormSelector = createSelector(
  formStateSelector,
  (form: IForm) => form.character
);
```

With that in hand, let's write a validation function that checks that all required fields are present and add it to our component:

```
export const isFormValid = createSelector(
  characterFormSelector,
  character => character.name
    && character.bioSummary.age
    && character.skills.length > 0
);

@select(isFormValidSelector)
isFormValid$: Observable<boolean>;
```

Now, after selecting, we can use `isFormValid$` with an asynchronous pipe in our template:

```
<button
  [disabled]="!(isFormValid$ | async)"
  type="submit">
  Save
</button>
```

(The `$` on the end of `isFormValid$` is a naming convention used in the RxJS library and elsewhere meaning, "This is an observable." We don't have to use it, but we find it helps make code easier to understand.)

We can now go ahead and create specific rules for specific fields of our form using a helper function called `isBetweenNumber` that does exactly what you'd expect:

```
const humanAgeValid = isBetweenNumber(14, 40);
const elfAgeValid = isBetweenNumber(80, 800);
const tieflingAgeValid = isBetweenNumber(35, 53);

const bioSummarySelector = createSelector(
  characterFormSelector,
  ({bioSummary}: ICharacter) => bioSummary
);

const ageValidationSelector = createSelector(
  bioSummarySelector,
  (bioSummary: IBioSummary) => {
    switch (bioSummary.race) {
      case 'Human': return humanAgeValid;
      case 'Elf': return elfAgeValid;
      case 'Tiefling': return tieflingAgeValid;
    }
  }
);

export const isAgeValidSelector = createSelector(
  bioSummarySelector,
  ageValidationSelector,
  ({age}, isAgeValid) => isAgeValid(age)
);
```

Once we have a pile of validation rules for our fields, we can chain those selectors together like this:

```
const isFormValidSelector = createSelector(
  isAgeValidSelector,
  isNameValidSelector,
  (ageValid, nameValid) => ageValid && nameValid
);
```

In larger applications, we can and should go further and make selectors out of reusable validation functions:

```
const maxStringLengthValidation = (value: string, max: number) =>
  value.length < max;

const minStringLengthValidation = (value: string, min: number) =>
  value.length > min;

const isNameValidSelector = createSelector(
  createFormFieldSelector(['character', 'name']),
  (name: string) => maxStringLengthValidation(name, 50)
  && minStringLengthValidation(name, 3)
);
```

We can make our application more user-friendly by subscribing to the selectors we've made and using them in our template:

```
@select(isAgeValidSelector)
isAgeValid$: Observable<boolean>;

@select(isNameValidSelector)
isNameValid$: Observable<boolean>;
```

Since we have access to Angular's `FormControl` states, we can use them to show error messages:

```
<input
  type="text"
  name="name"
  #nameField="ngModel"
  [(ngModel)]="characterForm.name">
<div
  [hidden]="isNameValid || nameField.control.pristine">
  Name must be between 3 and 50 characters
</div>
```

Conclusion

One of the things that makes it hard to discuss software architecture is that the solutions to big problems inevitably look like overkill when applied to small ones. If all we ever wanted to do was manage the names, ages, and skill sets of wizards, and if the entire application was only ever going to be maintained by its original author, we wouldn't want or need all of the machinery introduced above. However, these assumptions all too easily become self-fulfilling prophecies. If we don't architect our software to accommodate both expansion of function and expansion of the development team, both will be so painful that we won't do them.

Our experience is that combining Angular 2 forms with Redux for state management pays off sooner than you would think. Testing is a lot easier when all checks on changes to an application's state can be written as before-and-after rules. (To see how much easier, compare the [Redux testing examples](#) to whatever you're doing now.) Just as importantly, a single reducer can often be used for many forms, and generic actions can be re-used as well. For example, most of what we wrote above to handle character skills can be used to keep track of how many potions and scrolls they are carrying as well.

Of course, nothing is ever free. In large applications, it's important to design the structure of the Redux store so that new instances of it can be created quickly, and so that the next programmer to inherit the application can easily figure out which parts of the state to update when. We are also still learning how best to co-design reusable parts of forms and reusable parts of state so that reducers can be packaged and re-used as black boxes. Just as careful design of state makes the computer more efficient, careful co-design of components will, we hope, allow programmers to be more efficient. If you would like to share your experiences with this, we'd enjoy hearing from you.

EcmaScript 6 and TypeScript Features

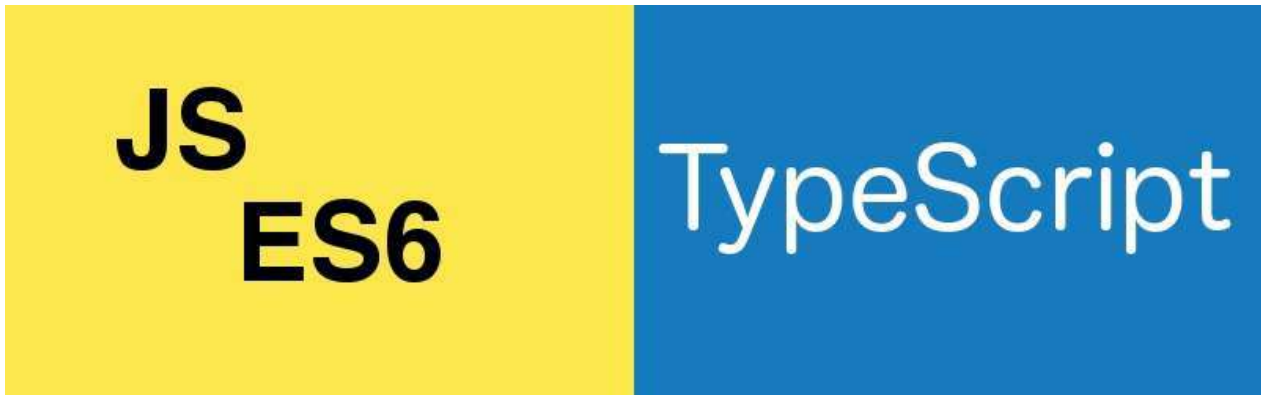


Figure: ES6 and TypeScript

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:

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

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

2. Function invocation:

```
someFunction();
```

Here, `this` used inside `someFunction` can refer to different things depending on whether we are in "strict" mode or not. Without using the "strict" mode, `this` refers to the context in which `someFunction()` was called. This 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, `this` would be undefined, which is slightly less confusing.

[View Example](#)

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

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

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

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

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

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

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

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

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

Arrow Functions

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

Arrow functions also make higher order functions much easier to work with.

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

Consider the following example:

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

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

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

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

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

The latter is *almost* equivalent to the following:

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

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

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

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

myToppings.outputList();
```

Let's try this code on <http://jsbin.com>. 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>. 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 imagine 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 available 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 constructor 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 `Penguin` class 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](#).

Constants and Block Scoped Variables

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

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

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

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

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

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

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

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

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

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

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

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

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

Alternatively, `const` is read-only. Once `const` has been assigned, the identifier cannot be reassigned.

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`.

[Ref.:](#)

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](#), [MDN-Rest Parameters](#), [ECMAScript Spec - Spread Syntax](#), [ECMAScript Spec - Rest Parameters](#)

Destructuring

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

To [borrow from the MDN](#), destructuring can be used to turn the following:

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

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

into

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

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

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

let {drawSquare, drawText} = myModule;

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

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

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

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

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

ES6 Modules

ES6 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](#)

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>
```

TypeScript

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

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

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

Without TS:

```
function add(a, b) {  
  return a + b;  
}  
  
add(1, 3); // 4  
add(1, '3'); // '13'
```

With TS:

```
function add(a: number, b: number) {  
  return a + b;  
}  
  
add(1, 3); // 4  
// compiler error before JS is even produced  
add(1, '3'); // '13'
```

Getting Started With TypeScript

Install the TypeScript transpiler using npm:

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

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

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

Note About ES6 Examples

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

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

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

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

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

Working With tsc

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

Imagine two ultra simple files/modules:

a.ts

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

b.ts

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

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](#)), 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](#) 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 `class` es 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, `class` es 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.

Class Decorators

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

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

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

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

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

@log('hello')
class World {
}

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

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

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

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

Parameter Decorators

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

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

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

The above demonstrates decorating method parameters. Readers familiar with Angular 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

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](#)

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

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 writing fast executing code on the server to handle events and non-blocking I/O efficiently.

- 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

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

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

Module Loading, Bundling and Build Tasks: **Webpack**

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

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](#) 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 } from '@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](#)

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

There is an `app` HTML element in the *index.html* file, and we use *app/main.ts* to import the `AppModule` component and the `platformBrowserDynamic().bootstrapModule` function and kickstart the process. 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 } from '@angular/core';
import { AppComponent } from './app.component'
import { GreeterService } from './greeter.service';

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

[View Example](#)

Components in Angular

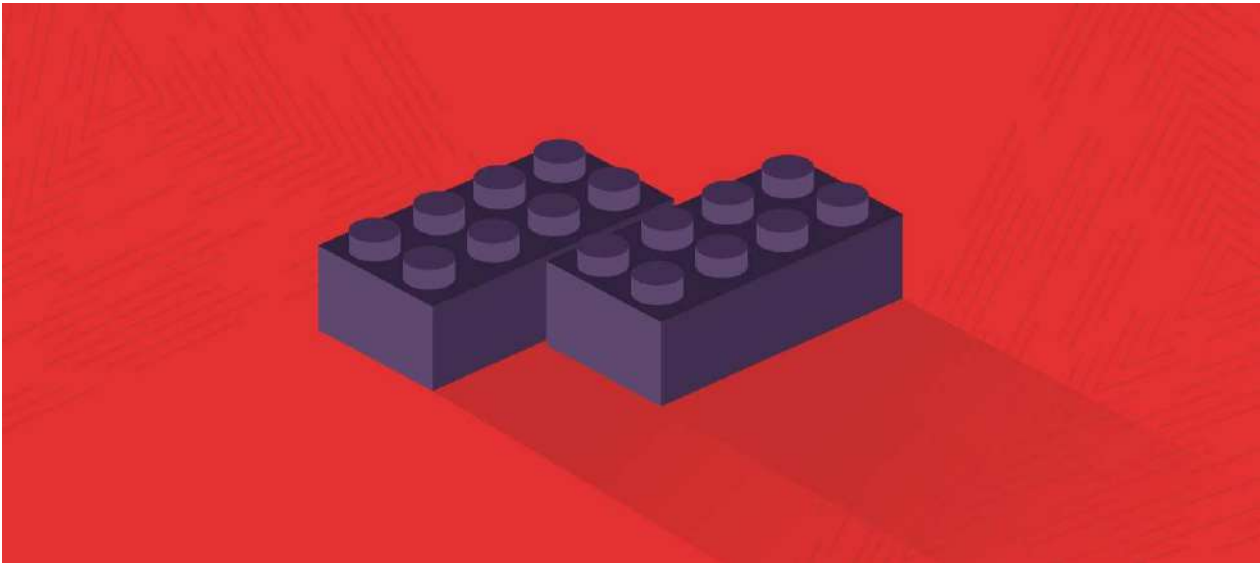


Figure: components

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 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.

[View Example](#)

Application Structure with Components

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

For example consider the following:

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

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

Passing Data into a Component

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

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

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

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

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

We can now use our component like so:

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

[View Example](#)

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](#)

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](#)

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

Using Two-Way Data Binding

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

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

What this is doing behind the scenes is equivalent to:

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

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

app/counter.component.ts

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

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

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

app/counter.component.html

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

[View Example](#)

Access Child Components From the Template

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

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

app/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](#)

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](#)

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

Projection

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

```
@Component({
  selector: 'rio-child',
  template: `
    <div>
      <h4>Child Component</h4>
      {{ childContent }}
    </div>
  `
})
export class ChildComponent {
  childContent = "Default content";
}
```

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

So, change `ChildComponent` to use projection:

```
app/child/child.component.ts
```

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

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

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

```
app/app.component.html
```

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

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

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

[View Example](#)

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](#)

Structuring Applications with Components

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

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

Using Other Components

Components depend on other components, directives and pipes. For example,

`TodoListComponent` relies on `TodoItemComponent`. To let a component know about these dependencies we group them into a module.

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

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

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

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

Please see the [Modules section](#) for more info about `NgModule`.

Directives

A Directive modifies the DOM to change appearance, 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': borderBottomStyle
      }">
      <ng-content></ng-content>
    </p>
  `
})
export class StyleExampleComponent {
  borderBottomStyle = '1px solid black';
}
```

[View Example](#)

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

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](#)

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](#)

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

Structural Directives

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

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

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

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

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

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

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

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

NgIf Directive

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

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

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

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

[View Example](#)

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

NgFor Directive

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

```
@Component({
  selector: 'app-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](#)

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

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

[View Example](#)

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

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](#)

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](#)

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

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

[View Example Without trackBy](#)

NgSwitch Directives

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

```
@Component({
  selector: 'app-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](#)

Here we see the `ngSwitch` attribute directive being attached to an element. This expression bound to the directive defines what will be 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](#)

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;

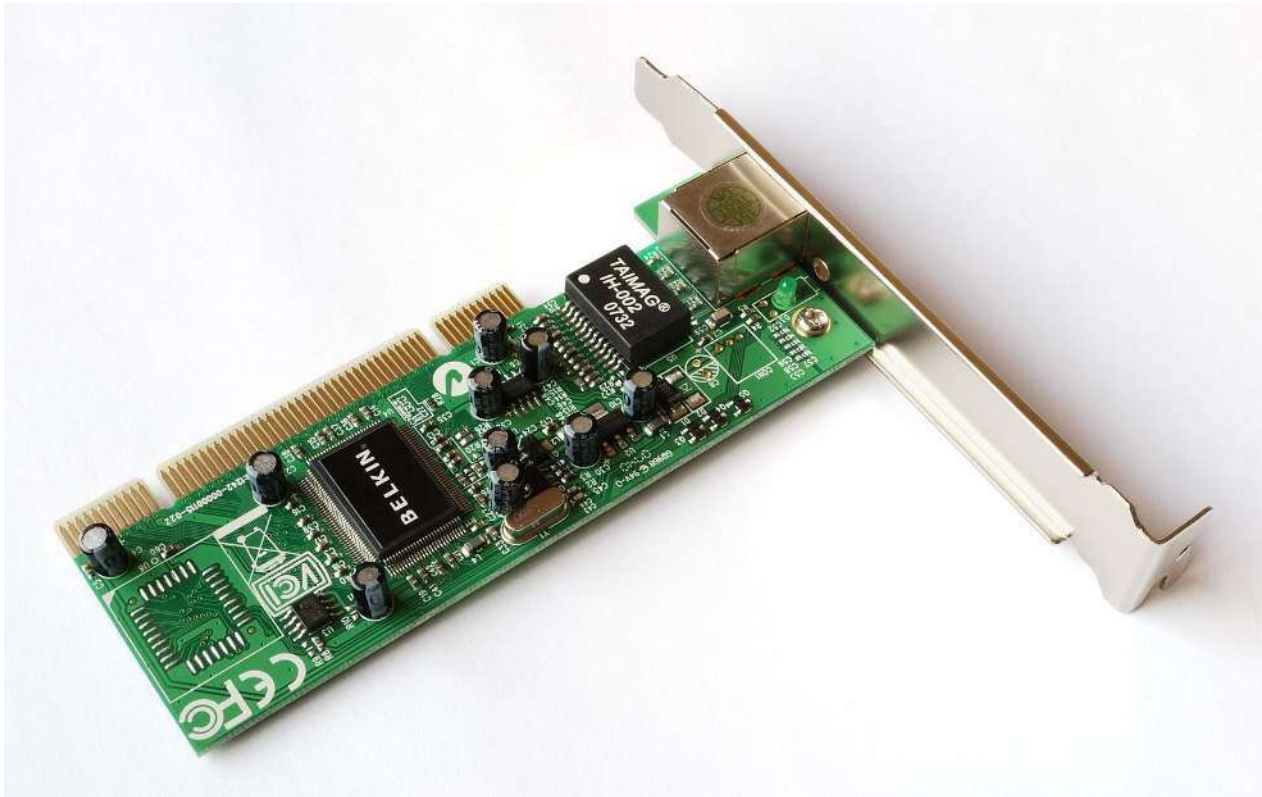
  tabs = [
    { title: 'Tab 1', content: 'Tab content 1' },
    { title: 'Tab 2', content: 'Tab content 2' },
    { title: 'Tab 3', content: 'Tab content 3' },
  ];

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

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

[View Example](#)

Advanced Components



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

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

Component Lifecycle

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

Here is the complete lifecycle hook interface inventory:

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

from [Component Lifecycle](#)

[View Example](#)

Accessing Child Component Classes

@ViewChild 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](#)

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](#)

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](#)

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

@ContentChild and @ContentChildren

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

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

[View Example](#)

View Encapsulation

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

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

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](#)

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](#)

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

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

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

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

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

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

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

```
export class MyApp {

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

  constructor() {
  }

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

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

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

      this.status = "Started";
    });

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

[View Example](#)

Error Handling

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

```
export class App {  
  
  values: number[] = [];  
  anyErrors: Error;  
  private data: Observable<number[]>;  
  
  constructor() {  
  
    this.data = new Observable(observer => {  
      setTimeout(() => {  
        observer.next(10);  
      }, 1500);  
      setTimeout(() => {  
        observer.error(new Error('Something bad happened!'));  
      }, 2000);  
      setTimeout(() => {  
        observer.next(50);  
      }, 2500);  
    });  
  
    let subscription = this.data.subscribe(  
      value => this.values.push(value),  
      error => this.anyErrors = error  
    );  
  }  
}
```

[View Example](#)

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

Disposing Subscriptions and Releasing Resources

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

```
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](#)

Calling `.unsubscribe()` will unhook a member's callbacks listening in on the `Observable` stream. When creating an `Observable` you can also return a custom callback, `onUnsubscribe`, that will be invoked when a member listening to the stream has unsubscribed. This is useful for any kind of cleanup that must be implemented. If we did not clear the `setTimeout` then values would still be emitting, but there would be no one listening. To save resources we should stop values from being emitted. An important thing to note is that when you call `.unsubscribe()` you are destroying the subscription object that is listening, therefore the on-complete event attached to that subscription object will not get called.

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

Observables vs Promises

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

- As seen in the example above, `observables` can define both the setup and teardown aspects of asynchronous behavior.
- `observables` are cancellable.
- Moreover, `observables` can be retried using one of the retry operators provided by the API, such as `retry` and `retryWhen`. On the other hand, `Promises` require the caller to have access to the original function that returned the promise in order to have a retry capability.

Using Observables From Other Sources

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

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

Observable HTTP Events

A common operation in any web application is getting or posting data to a server. Angular applications do this with the `Http` library, which previously used `Promises` to operate in an asynchronous manner. The updated `Http` library now incorporates `Observables` for triggering events and getting new data. Let's take a quick look at this:

```
import {Component} from '@angular/core';
import {Http} from '@angular/http';
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](#)

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](#)

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](#)

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

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

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

Note the chaining function style, and the optional static typing that comes with TypeScript, that we used in this example. Most importantly functions like `filter` return an `Observable`, as in `Observables` beget other `Observables`, similarly to promises. In order to use `map` and `filter` in a chaining sequence we have flattened the results of our `Observable` using

`flatMap` . Since `filter` accepts an `Observable` , and not an array, we have to convert our array of JSON objects from `data.json()` to an `Observable` stream. This is done with `flatMap` .

There are many other array operations you can employ in your `observables` ; look for them in the [RxJS API](#).

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

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](#)

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

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

latecomer must start watching the performance from where it is.

We have already encountered both kind of `observables`. The example above is a cold `observable`, while the example that uses `valueChanges` on our text field input is a hot `Observable`.

Converting from Cold Observables to Hot Observables

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

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

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

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

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

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

}).publish();

obsv.connect();

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

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

[View Example](#)

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

Another useful method to work with hot `observables` instead of `connect` is `refCount`. This is an auto connect method, that will start broadcasting as soon as there is more than one subscriber. Analogously, it will stop if the number of subscribers goes to 0; in other words, if everyone in the audience walks out, the performance will stop.

Summary

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

- [rxjs 4 to 5 migration](#)
- [rxjs Observable API](#)
- [Which operator do I use?](#)
- [rxmarbles](#)
- [RxJS Operators by Example](#)

Angular Dependency Injection

Dependency Injection (DI) was a core feature in Angular 1.x, and that has not changed in Angular 2. DI is a programming concept that predates Angular. The purpose of DI is to simplify dependency management in software components. By reducing the amount of information a component needs to know about its dependencies, unit testing can be made easier and code is more likely to be flexible.

Angular 2 improves on Angular 1.x's DI model by unifying Angular 1.x's two injection systems. Tooling issues with respect to static analysis, minification and namespace collisions have also been fixed in Angular 2.

What is DI?

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

Consider the following code:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

DI Framework

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

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

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

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

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

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

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

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

Angular's DI

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

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

For example:

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

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

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

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

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

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

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

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

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

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

@Component({
  // ...
})
class AuthWidget {}

@Injectable()
class AuthService {}

@Injectable()
class ChatSocket {}

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

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

@Inject and @Injectable

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

@Inject()

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

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

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

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

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

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

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

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

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

[View Example](#)

@Injectable()

`@Injectable()` lets Angular know that a *class* can be used with the dependency injector.

`@Injectable()` is not *strictly* required if the class has *other* Angular decorators on it or does not have any dependencies. What is important is that any class that is going to be injected with Angular *is decorated*. However, best practice is to decorate injectables with

`@Injectable()`, as it makes more sense to the reader.

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

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

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

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

Injection Beyond Classes

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

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

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

@NgModule({
  providers: [ { provide: ChatWidget, useClass: ChatWidget } ],
})
export class DiExample {};
```

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

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

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

@NgModule({
  providers: [ { provide: ChatWidget, useClass: MockChatWidget } ],
})
export class DiExample {};
```

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

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

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

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

@NgModule({
  providers: [ { provide: 'Random', useFactory: randomFactory } ],
})
export class DiExample {};
```

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

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

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

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

[View Example](#)

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

The above example uses Angular'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](#)

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](#)

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

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

OpaqueToken

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

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

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

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

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

app/email/email.module.ts

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

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

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

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

[View Example](#)

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

The Injector Tree

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

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

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

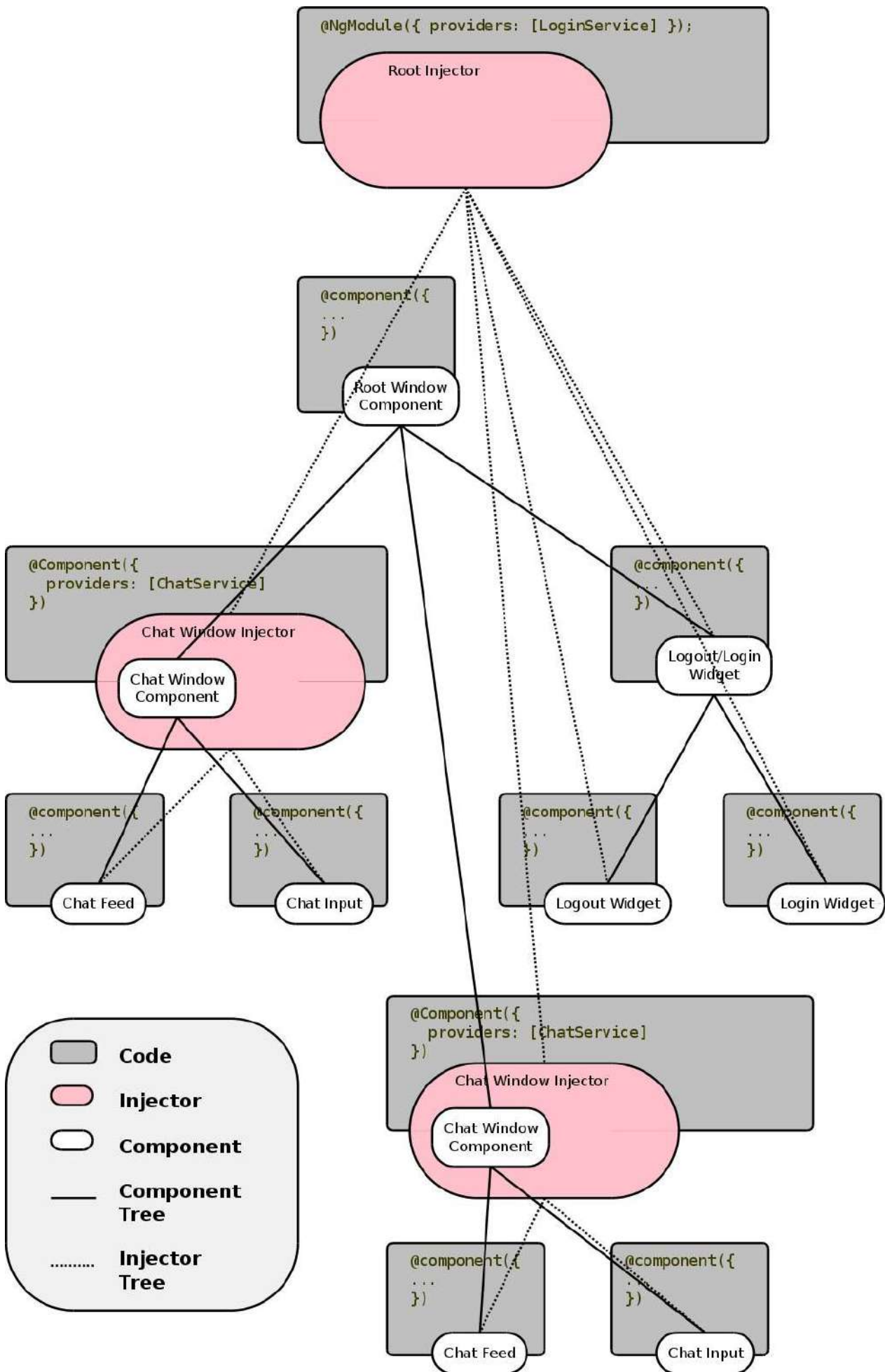


Figure: Image of a Component Tree, and a DI Tree

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

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

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

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

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

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

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

Consider the following example:

app/module.ts

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

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

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

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

app/services/unique.ts

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

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

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

app/components/child-inheritor.component.ts

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

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

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

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

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

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

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

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

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

app/containers/app.ts

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

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

[View Example](#)

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 `SearchService` to interact with the Spotify API.

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

@Injectable()
export class SearchService {

  constructor(private http: Http) {}

  search(term: string) {
    return this.http
      .get('https://api.spotify.com/v1/search?q=' + term + '&type=artist')
      .map(response => response.json());
  }
}
```

[View Example](#)

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](#)

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](#)

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.

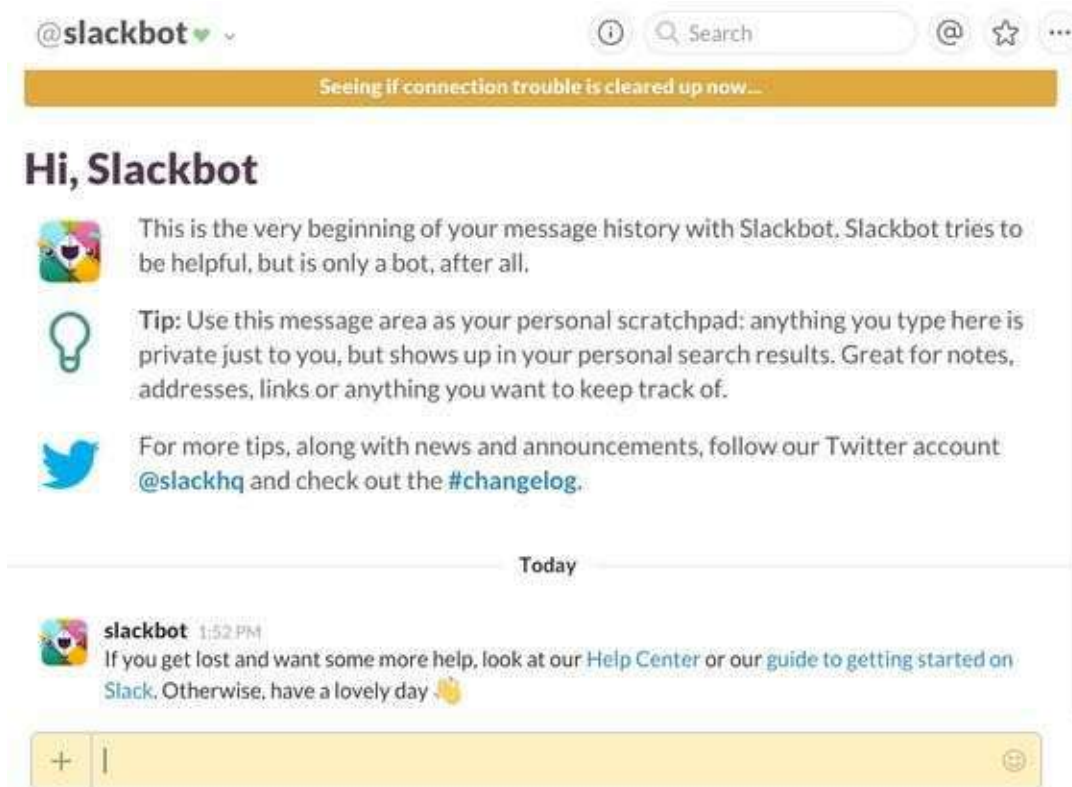


Figure: Retry example from Slack

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](#)

Combining Streams with flatMap

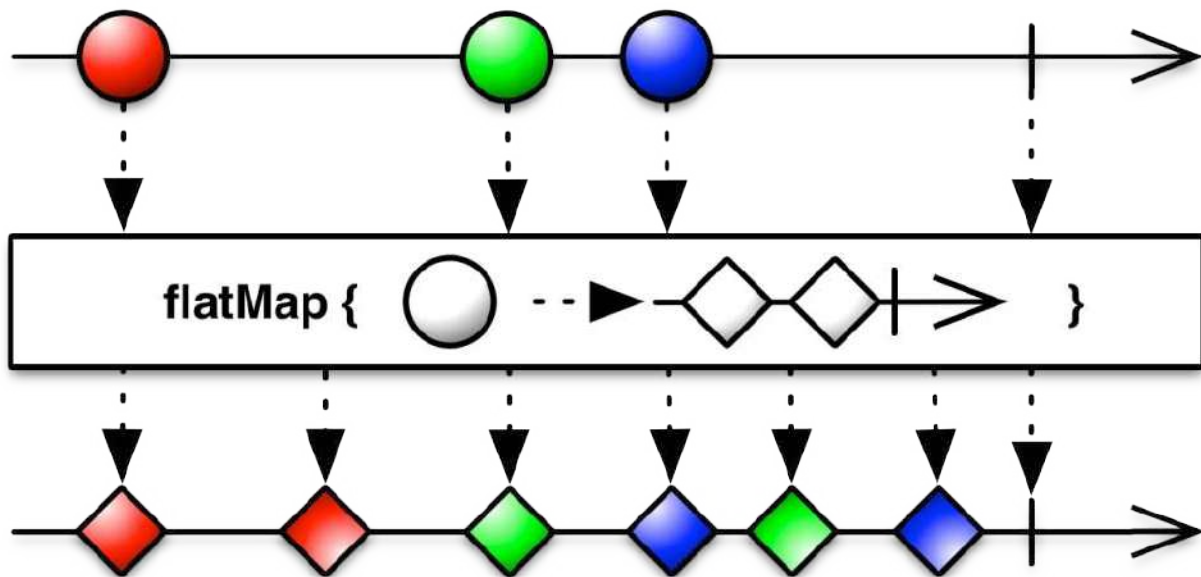


Figure: FlatMap created by ReactiveX licensed under CC-3
 (<http://reactivex.io/documentation/operators/flatmap.html>)

A case for FlatMap:

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

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

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

`app/services/search.service.ts`

```
import {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](#)

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`:

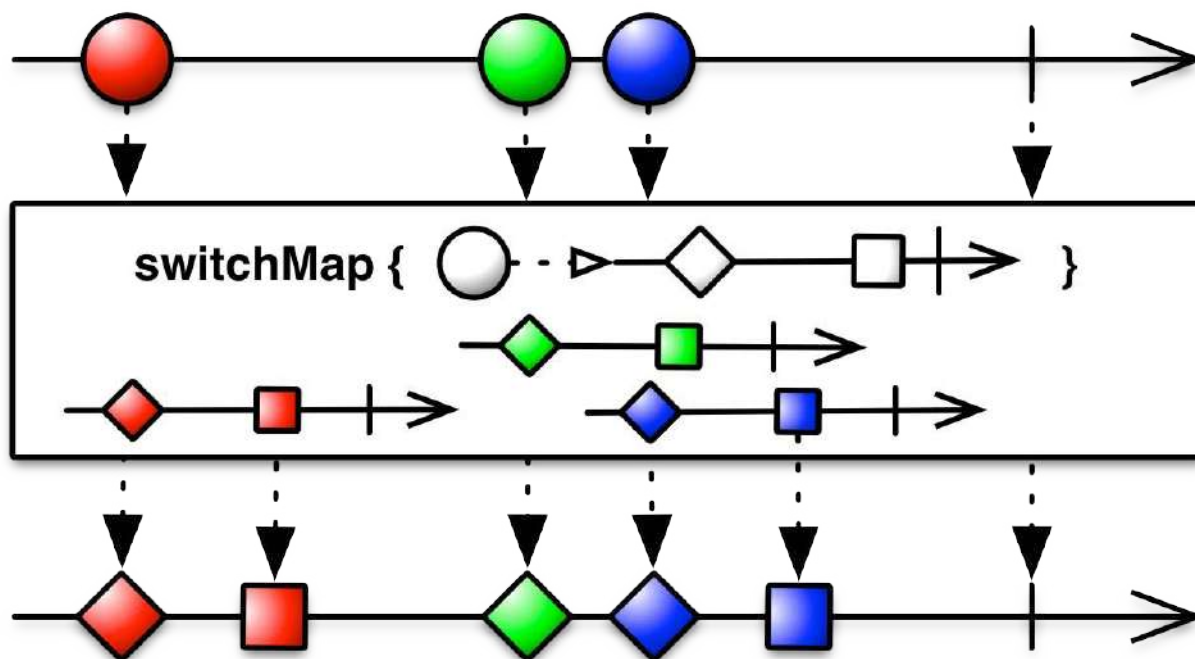


Figure: SwitchMap created by ReactiveX licensed under CC-3
 (<http://reactivex.io/documentation/operators/flatMap.html>)

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](#)

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](#)
- [Egghead Video Tutorial on SwitchMap](#)

- [RxJS Documentation for 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

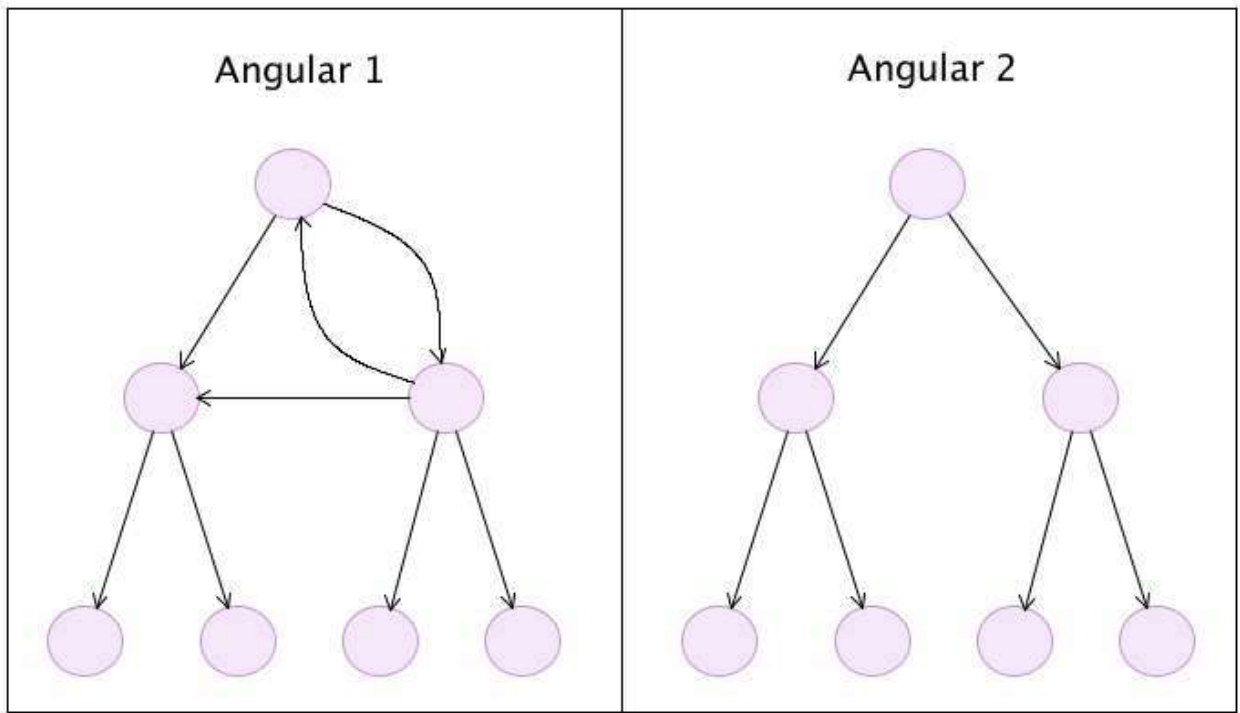


Figure: File Structure

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](#)

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 `detectChanges` passing the current model values to compare with the old ones. If a change was detected, the component gets updated.

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

Change Detection Strategy: Default

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

app/movie.component.ts

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

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

[View Example](#)

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

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

Phase 1 (Application):

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

Phase 2 (Change Detection):

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

Change detection always starts at the root component, in this case the `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 `AppComponent` template has a reference to three properties, `slogan`, `title` and `actor`, so the comparison made by its corresponding change detector will look like:

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

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

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

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

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

Performance Impact

Traversing all the tree components to check for changes could be costly. Imagine that instead of just having one reference to `<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](#)

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

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

When the change detection analyzes the properties bound to the `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 `actor` object, were immutable objects, but we went ahead and updated its properties, violating the immutability principle. As a result we ended up with a sync problem between our models and our views. One way to enforce immutability is using the library [Immutable.js](#).

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

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

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

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

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

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

Then in our `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 `actor` object as an immutable type, and in the template, instead of trying to access the object properties directly using a syntax like `actor.firstName`, we need to use the `get` method of the immutable.

app/movie.component.ts


```
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](#)

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

Additional Resources

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

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

Zones

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

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

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

It is important to know that Zone.js accomplishes these various interceptions by [Monkey Patching](#) common methods and elements in the browser, e.g. `setTimeout` and `HTMLInputElement.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](#) 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 occurring.

```
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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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.setStyle(this.el.nativeElement, 'backgroundColor', color);
  }
}
```

[View Example](#)

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 programmatically 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, let's 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](#)

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](#)

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

View Containers and Embedded Views

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

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

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

@Directive({
  selector: '[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

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 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](#)

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](#)

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](#) 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 `eval` ed. 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](#) 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](#).

(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](#) 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.

Immutable.js Basics

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

Immutable.Map

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

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

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

let movie2 = movie1;

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

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

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

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

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

Map.merge

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

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

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

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

Nested Objects

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

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

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

console.log(movie.toObject());

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

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

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

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

console.log(movie.toObject());

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

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

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

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

console.log(movie.toObject());

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

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

Deleting Keys

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

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

movie = movie.delete('mpaa');

console.log(movie.toObject());

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

Maps are Iterable

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

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

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

  return providedFeatures;
}, []);

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

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

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

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

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

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


Immutable.List

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

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

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

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

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

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

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

Performance and Transient Changes

Performance

Immutable data structures often have a performance penalty due to the costs of allocation new memory and copying data. Consider these two examples, one which uses a mutable array and one which uses an Immutable.js collection.

Mutable

```
const list = [];  
let val = "";  
  
Immutable.Range(0, 1000000)  
  .forEach(function() {  
    val += "concatenation";  
    list.push(val);  
  });
```

Immutable

```
const init = {  
  list: Immutable.List(),  
  val: ""  
};  
  
const list = Immutable.Range(0, 1000000)  
  .reduce(function(reduced) {  
    var next = reduced.val + "concatenation";  
  
    return {  
      list: reduced.list.push(next),  
      val: next  
    };  
  }, init).list
```

Here the fully immutable code runs around 90% slower than the mutable code! While immutable data can make code much easier to reason about, there is definitely a cost associated with that decision. As we can see here for iterative concat, this can have a major impact on usability. Fortunately, Immutable.js provides some features where the performance costs can be mitigated.

Persistent Data Structures and Transient Changes

Immutable data structures are also sometimes referred to as *persistent data structures*, since their values persist for their lifetime. Immutable.js provides the option for *transient changes*: operations during which an immutable data structure can perform mutable changes locally while returning an immutable result. This is one approach to solving the performance issues we encountered earlier. Let's revisit the immutable case outlined in the performance example, but using a transient data structure this time:

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

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

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

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

This transient list builder is still much slower than our fully mutable implementation but much faster than our fully immutable version.

Official documentation

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

Pipes



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

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

Using Pipes

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

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

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

[View Example](#)

Passing Parameters

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

```
pipeName: parameterValue
```

You can also pass multiple parameters this way:

```
pipeName: parameter1: parameter2
```

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

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

[View Example](#)

Chaining Pipes

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

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

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

[View Example](#)

Custom Pipes

Angular allows you to create your own custom pipes:

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

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

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

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

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

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

Each custom pipe implementation must:

- have the `@Pipe` decorator with pipe metadata that has a `name` property. This value will be used to call this pipe in template expressions. It must be a valid JavaScript identifier.
- implement the `PipeTransform` interface's `transform` method. This method takes the value being piped and a variable number of arguments of any type and return a transformed ("piped") value.

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

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

@Component({
  selector: 'app-root',
  template: `
    <div>
      <p *ngFor="let f of fileSizes">{{ f | formatFileSize }}</p>
      <p>{{ largeFileSize | formatFileSize:true }}</p>
    </div>`
})
export class AppComponent {
  fileSizes = [10, 100, 1000, 10000, 100000, 1000000, 1000000000];
  largeFileSize = Math.pow(10, 15)
}
```

[View Example](#)

Stateful Pipes

There are two categories of pipes:

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

Angular provides `AsyncPipe`, which is stateful.

AsyncPipe

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

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

  seconds = Observable.of(0).concat(Observable.interval(1000))
}
```

[View Example](#)

Implementing Stateful Pipes

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

```
// naive implementation assumes small number increments
@Pipe({
  name: 'animateNumber',
  pure: false
})
export class AnimateNumberPipe implements PipeTransform {
  private currentNumber: number = null; // intermediary number
  private targetNumber: number = null;

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

      const difference = this.targetNumber - this.currentNumber

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

    return this.currentNumber;
  }
}
```

[View Example](#)

Forms

An application without user input is just a page. Capturing input from the user is the cornerstone of any application. In many cases, this means dealing with forms and all of their complexities.

Angular 2 is much more flexible than Angular 1.x for handling forms — we are no longer restricted to relying solely on `ngModel`. Instead, we are given degrees of simplicity and power, depending on the form's purpose.

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

Getting Started

Opt-In APIs

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

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

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

platformBrowserDynamic().bootstrapModule(AppModule)
```

Input Labeling

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

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

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

```
<input> S:
```

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

Template-Driven Forms

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

First, consider a typical form:

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

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

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

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

index.html

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

signup-form.component.html

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

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

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

signup-form.component.ts

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

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


Nesting Form Data

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

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

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

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

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

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

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

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

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

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

  <!-- ... -->

</form>
```

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

ngModelGroup

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

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

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

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

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

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

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

Using Template Model Binding

One-Way Binding

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

app/signup-form.component.html

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

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

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

app/signup-form.component.ts

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

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

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

Two-Way Binding

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

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

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

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

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

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

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

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

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

Validating Template-Driven Forms

Validation

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

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

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

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

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

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

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

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

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

Reactive/Model-Driven Forms

While using directives in our templates gives us the power of rapid prototyping without too much boilerplate, we are restricted in what we can do. Reactive forms on the other hand, lets us define our form through code and gives us much more flexibility and control over data validation.

There is a little bit of magic in its simplicity at first, but after you're comfortable with the basics, learning its building blocks will allow you to handle more complex use cases.

Reactive Forms Basics

To begin, we must first ensure we are working with the right directives and the right classes in order to take advantage of procedural forms. For this, we need to ensure that the

`ReactiveFormsModule` was imported in the bootstrap phase of the application module.

This will give us access to components, directives and providers like `FormBuilder`, `FormGroup`, and `FormControl`

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

app/login-form.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: 'app/app.component.html'
})
export class AppComponent {
  username = new FormControl('')
  password = new FormControl('')

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

  constructor(private builder: FormBuilder) { }

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

app/login-form.component.html


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

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

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

[View Example](#)

FormControl

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

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

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

Validating Reactive Forms

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

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

app/login-form.component.ts

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

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

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

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

  constructor(private builder: FormBuilder) { }

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

app/login-form.component.html

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

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

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

    <div [hidden]="username.valid || username.untouched">
      <div>
        The following problems have been found with the username:
      </div>

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

    <div [hidden]="password.valid || password.untouched">
      <div>
        The following problems have been found with the password:
      </div>

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

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

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

[View Example](#)

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

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

Reactive Forms Custom Validation

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

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

app/login-form.component.ts

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

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

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

app/login-form.component.ts

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

[View Example](#)

Predefined Parameters

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

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

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

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

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

app/login-form.component.ts

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

// ...

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

app/login-form.component.html

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

[View Example](#)

Validating Inputs Using Other Inputs

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

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

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

// ...

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

[View Example](#)

Visual Cues for Users

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

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

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

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

```
/* field value is valid */  
.ng-valid {}  
  
/* field value is invalid */  
.ng-invalid {}  
  
/* field has not been clicked in, tapped on, or tabbed over */  
.ng-untouched {}  
  
/* field has been previously entered */  
.ng-touched {}  
  
/* field value is unchanged from the default value */  
.ng-pristine {}  
  
/* field value has been modified from the default */  
.ng-dirty {}
```

Note the three pairs:

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

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

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

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

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

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

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

Modules

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

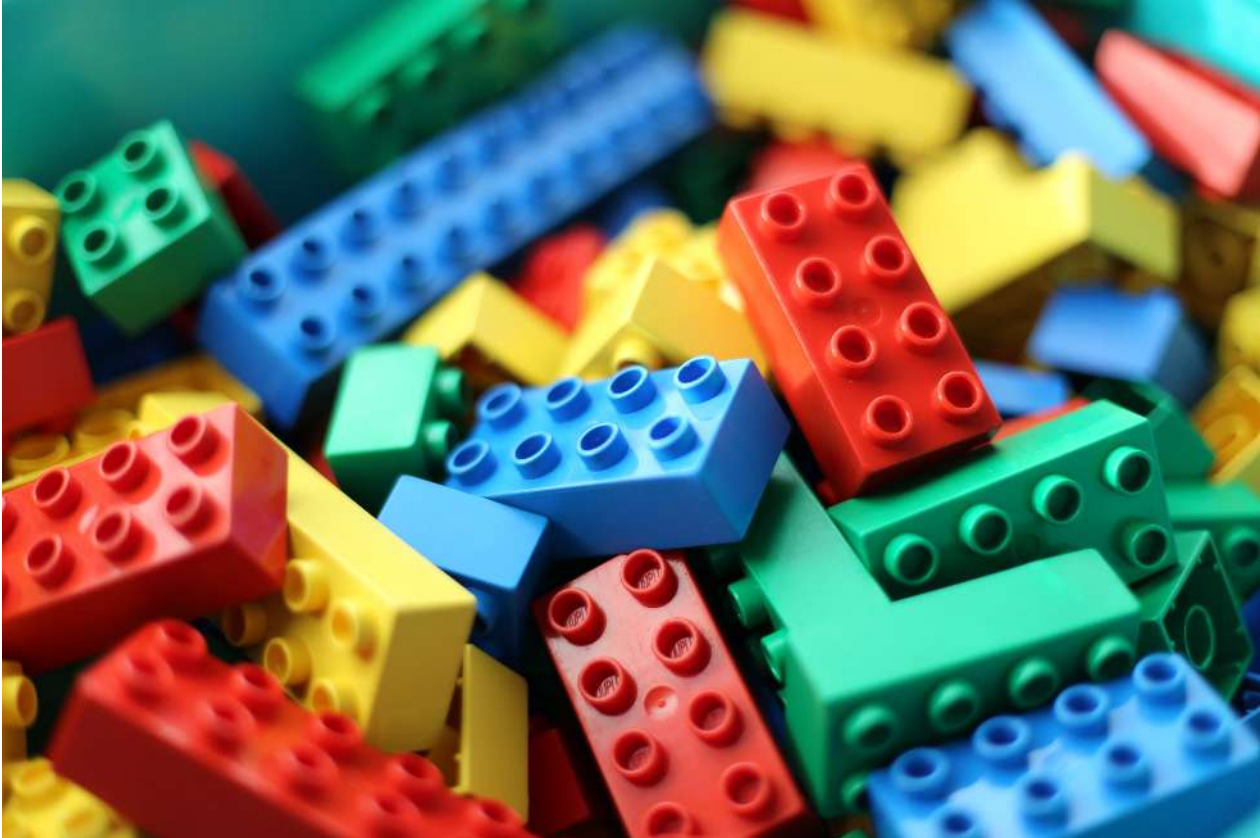


Figure: Used Lego Duplo Bricks by Arto Alanenpää is licensed under CC BY-SA 4.0 (https://commons.wikimedia.org/wiki/File:Lego_dublo_arto_alanenpaa_5.JPG)

What is an Angular Module?

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

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

A Basic Use of Modules

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

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

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

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

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

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

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

app/app.component.ts

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

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

app/app.module.ts

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

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

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

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

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

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

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

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

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

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

Bootstrapping an Application

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

main.ts

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

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

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

[View Example](#)

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

Adding Components, Pipes and Services to a Module

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

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

credit-card.component.ts

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

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

  constructor(private creditCardService: CreditCardService) {}

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

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

credit-card.service.ts

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

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

credit-card-mask.pipe.ts

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

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

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

app.component.ts

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

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

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

app.module.ts

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

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

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

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

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

[View Example](#)

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

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

Creating a Feature Module

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

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

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

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

Notice how each folder has its own module file: *app.module.ts* and *credit-card.module.ts*. Let's focus on the latter first.

credit-card/credit-card.module.ts

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

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

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

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

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

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

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

app/app.component.ts

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

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

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

app/app.module.ts

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

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

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

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

[View Example](#)

Services and Lazy Loaded Modules

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

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

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

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

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

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

credit-card/credit-card.module.ts

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

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

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

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

app/app.module.ts

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

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

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

[View Example](#)

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

Directive Duplications

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

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

This example is a variation of the code found in the [official documentation](#).

blue-highlight.directive.ts

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

@Directive({
  selector: '[appHighlight]'
})
export class BlueHighlightDirective {
  constructor(renderer: Renderer, el: ElementRef) {
    renderer.setStyle(el.nativeElement, 'backgroundColor', 'blue');
    renderer.setStyle(el.nativeElement, 'color', 'gray');
  }
}
```

yellow-highlight.directive.ts

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

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

These two directives are similar, they are trying to style an element. The

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

app.module.ts

```
// Imports

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

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

app.component.ts

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

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

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

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

[View Example](#)

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

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

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

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

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

Lazy Loading a Module

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

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

app/app.module.ts

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

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

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

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

app/app.component.ts

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

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

Our navigation system has only two paths: `eager` and `lazy`. To know what those paths are loading when clicking on them we need to take a look at the `routing` object that we passed to the root module.

app/app.routing.ts

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

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

const routes: Routes = [
  { path: '', redirectTo: 'eager', pathMatch: 'full' },
  { path: 'eager', component: EagerComponent },
  { path: 'lazy', loadChildren: 'lazy/lazy.module#LazyModule' }
];

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

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

app/eager.component.ts

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

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

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

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

There's a few important things to notice here:

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

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

app/lazy/lazy.module.ts

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

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

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

If we define the class `LazyModule` as the `default` export of the file, we don't need to define the class name in the `loadChildren` property as shown above.

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

app/lazy/lazy.routing.ts

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

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

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

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

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

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

app/lazy/lazy.component.ts

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

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

[View Example](#)

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

We have effectively lazily loaded a module.

Lazy Loading and the Dependency Injection Tree

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

To show this behaviour, let's continue with the example of the previous section and add a

`CounterService` to our `LazyModule` .

app/lazy/lazy.module.ts

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

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

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

app/lazy/counter.service.ts

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

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

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

app/lazy/lazy.component.ts

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

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

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

  constructor(public counterService: CounterService) {}

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

[View Example](#)

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

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

`EagerComponent` .

app/eager.component.ts

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

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

  constructor(public counterService: CounterService) {}

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

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

```
No provider for CounterService!
```

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

Shared Modules and Dependency Injection

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

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

app/shared/shared.module.ts

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

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

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

app/app.module.ts

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

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

app/lazy/lazy.module.ts


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

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

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

app/eager.component.ts

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

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

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

[View Example](#)

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

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

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

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

Sharing the Same Dependency Injection Tree

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

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

app/shared/shared.module.ts

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

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

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

app/app.module.ts

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

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

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

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

[View Example](#)

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

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

app/shared/shared.module.ts

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

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

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

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

app/lazy/lazy.module.ts

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

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

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

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

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

[View Example](#)

Routing

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

Why Routing?

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

Routing allows you to:

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

Configuring Routes

Base URL Tag

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

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

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

Route Definition Object

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

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

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

To use `Routes`, create an array of [route configurations](#).

Below is the sample `Routes` array definition:

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

[See Routes definition](#)

RouterModule

`RouterModule.forRoot` takes the `Routes` array as an argument and returns a *configured* router module. The following sample shows how we import this module in an `app.routes.ts` file.

app/app.routes.ts

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

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

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

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

app/app.module.ts

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

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

Redirecting the Router to Another Route

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

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

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

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

Defining Links Between Routes

RouterLink

Add links to routes using the `RouterLink` directive.

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

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

Navigating Programmatically

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

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

Dynamically Adding Route Components

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

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

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

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

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

[View Example](#)

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

Using Route Parameters

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

- add a route parameter ID
- link the route to the parameter
- add the service that reads the parameter.

Declaring Route Parameters

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

Routes :

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

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

Linking to Routes with Parameters

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

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

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

```
goToProductDetails(id) {  
  this.router.navigate(['/product-details', id]);  
}
```

Reading Route Parameters

The `ProductDetails` component must read the parameter, then load the product based on the ID given in the parameter.

The `ActivatedRoute` service provides a `params` Observable which we can subscribe to to get the route parameters (see [Observables](#)).

```
import { Component, OnInit, OnDestroy } from '@angular/core';  
import { ActivatedRoute } from '@angular/router';  
  
@Component({  
  selector: 'product-details',  
  template: `  
    <div>  
      Showing product details for product: {{id}}  
    </div>  
  `,  
})  
export class LoanDetailsPage implements OnInit, OnDestroy {  
  id: number;  
  private sub: any;  
  
  constructor(private route: ActivatedRoute) {}  
  
  ngOnInit() {  
    this.sub = this.route.params.subscribe(params => {  
      this.id = +params['id']; // (+) converts string 'id' to a number  
  
      // In a real app: dispatch action to load the details here.  
    });  
  }  
  
  ngOnDestroy() {  
    this.sub.unsubscribe();  
  }  
}
```

The reason that the `params` property on `ActivatedRoute` is an Observable is that the router may not recreate the component when navigating to the same component. In this case the parameter may change without the component being recreated.

[View Basic Example](#)

[View Example with Programmatic Route Navigation](#)

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

Defining Child Routes

When some routes may only be accessible and viewed within other routes it may be appropriate to create them as child routes.

For example: The product details page may have a tabbed navigation section that shows the product overview by default. When the user clicks the "Technical Specs" tab the section shows the specs instead.

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

```
localhost:3000/product-details/3/overview
```

When the user clicks "Technical Specs":

```
localhost:3000/product-details/3/specs
```

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

Our `Routes` with children would look like:

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

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


```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'product-details',
  template: `
    <p>Product Details: {{id}}</p>
    <!-- Product information -->
    <nav>
      <a [routerLink]="['overview']">Overview</a>
      <a [routerLink]="['specs']">Technical Specs</a>
    </nav>
    <router-outlet></router-outlet>
    <!-- Overview & Specs components get added here by the router -->
  `
})
export default class ProductDetails implements OnInit, OnDestroy {
  id: number;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this.sub = this.route.params.subscribe(params => {
      this.id = +params['id']; // (+) converts string 'id' to a number
    });
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }
}
```

Alternatively, we could specify `overview` route URL simply as:

```
localhost:3000/product-details/3
```

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

Since the `overview` child route of `product-details` has an empty path, it will be loaded by default. The `specs` child route remains the same.

[View Example with child routes](#)

[View Example with route params & child routes](#)

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

Accessing a Parent's Route Parameters

In the above example, say that the child routes of `product-details` needed the ID of the product to fetch the spec or overview information. The child route component can access the parent route's parameters as follows:

```
export default class Overview {
  parentRouteId: number;
  private sub: any;

  constructor(private router: Router,
              private route: ActivatedRoute) {}

  ngOnInit() {
    // Get parent ActivatedRoute of this route.
    this.sub = this.router.routerState.parent(this.route)
      .params.subscribe(params => {
        this.parentRouteId = +params["id"];
      });
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }
}
```

[View Example child routes accessing parent's route parameters](#)

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

Links

Routes can be prepended with `/`, or `../`; this tells Angular where in the route tree to link to.

Prefix	Looks in
/	Root of the application
none	Current component children routes
../	Current component parent routes

Example:

```
<a [routerLink]="['route-one']">Route One</a>  
<a [routerLink]="['../route-two']">Route Two</a>  
<a [routerLink]="['/route-three']">Route Three</a>
```

In the above example, the link for route one links to a child of the current route. The link for route two links to a sibling of the current route. The link for route three links to a child of the root component (same as route one link if current route is root component).

[View Example with linking throughout route tree](#)

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

Controlling Access to or from a Route

To control whether the user can navigate to or away from a given route, use route guards.

For example, we may want some routes to only be accessible once the user has logged in or accepted Terms & Conditions. We can use route guards to check these conditions and control access to routes.

Route guards can also control whether a user can leave a certain route. For example, say the user has typed information into a form on the page, but has not submitted the form. If they were to leave the page, they would lose the information. We may want to prompt the user if the user attempts to leave the route without submitting or saving the information.

Registering the Route Guards with Routes

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

For example, say we have an `accounts` route that only users that are logged in can navigate to. This page also has forms and we want to make sure the user has submitted unsaved changes before leaving the accounts page.

In our route config we can add our guards to that route:

```
import { Routes, RouterModule } from '@angular/router';
import { AccountPage } from './account-page';
import { LoginRouteGuard } from './login-route-guard';
import { SaveFormsGuard } from './save-forms-guard';

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

export const appRoutingProviders: any[] = [];

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

Now `LoginRouteGuard` will be checked by the router when activating the `accounts` route, and `SaveFormsGuard` will be checked when leaving that route.

Implementing CanActivate

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

```
import { CanActivate } from '@angular/router';
import { Injectable } from '@angular/core';
import { LoginService } from './login-service';

@Injectable()
export class LoginRouteGuard implements CanActivate {

  constructor(private loginService: LoginService) {}

  canActivate() {
    return this.loginService.isLoggedIn();
  }
}
```

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

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

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

[See Official Definition for CanActivate](#)

Implementing CanDeactivate

`CanDeactivate` works in a similar way to `CanActivate` but there are some important differences. The `canDeactivate` function passes the component being deactivated as an argument to the function:

```
export interface CanDeactivate<T> {
  canDeactivate(component: T, route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
    Observable<boolean>|Promise<boolean>|boolean;
}
```

We can use that component to determine whether the user can deactivate.

```
import { CanDeactivate } from '@angular/router';
import { Injectable } from '@angular/core';
import { AccountPage } from './account-page';

@Injectable()
export class SaveFormsGuard implements CanDeactivate<AccountPage> {

  canDeactivate(component: AccountPage) {
    return component.areFormsSaved();
  }
}
```

[See Official Definition for CanDeactivate](#)

Async Route Guards

The `canActivate` and `canDeactivate` functions can either return values of type `boolean`, or `observable<boolean>` (an Observable that resolves to `boolean`). If you need to do an asynchronous request (like a server request) to determine whether the user can navigate to or away from the route, you can simply return an `observable<boolean>`. The router will wait until it is resolved and use that value to determine access.

For example, when the user navigates away you could have a dialog service ask the user to confirm the navigation. The dialog service returns an `observable<boolean>` which resolves to true if the user clicks 'OK', or false if user clicks 'Cancel'.

```
canDeactivate() {
  return dialogService.confirm('Discard unsaved changes?');
}
```

[View Example](#)

[See Official Documentation for Route Guards](#)

Passing Optional Parameters

Query parameters allow you to pass optional parameters to a route such as pagination information.

For example, on a route with a paginated list, the URL might look like the following to indicate that we've loaded the second page:

```
localhost:3000/product-list?page=2
```

The key difference between query parameters and [route parameters](#) is that route parameters are essential to determining route, whereas query parameters are optional.

Passing Query Parameters

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

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

Alternatively, we can navigate programmatically using the `Router` service:

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

Reading Query Parameters

Similar to reading [route parameters](#), the `Router` service returns an [Observable](#) we can subscribe to to read the query parameters:

```
import { Component } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router';

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

  ngOnInit() {
    this.sub = this.route
      .queryParams
      .subscribe(params => {
        // Defaults to 0 if no query param provided.
        this.page = +params['page'] || 0;
      });
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }

  nextPage() {
    this.router.navigate(['product-list'], { queryParams: { page: this.page + 1 } });
  }
}
```

[View Example](#)

[See Official Documentation on Query Parameters](#)

Using Auxiliary Routes

Angular supports the concept of auxiliary routes, which allow you to set up and navigate multiple independent routes in a single app. Each component has one primary route and zero or more auxiliary outlets. Auxiliary outlets must have unique name within a component.

To define the auxiliary route we must first add a named router outlet where contents for the auxiliary route are to be rendered.

Here's an example:

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

@Component({
  selector: 'app',
  template: `
    <nav>
      <a [routerLink]="['/component-one']">Component One</a>
      <a [routerLink]="['/component-two']">Component Two</a>
      <a [routerLink]="[{ outlets: { 'sidebar': ['component-aux'] } }]">Component Aux<
    /a>
  </nav>

  <div style="color: green; margin-top: 1rem;">Outlet:</div>
  <div style="border: 2px solid green; padding: 1rem;">
    <router-outlet></router-outlet>
  </div>

  <div style="color: green; margin-top: 1rem;">Sidebar Outlet:</div>
  <div style="border: 2px solid blue; padding: 1rem;">
    <router-outlet name="sidebar"></router-outlet>
  </div>
  `
})
export class AppComponent {
}
```

Next we must define the link to the auxiliary route for the application to navigate and render the contents.

```
<a [routerLink]="[{ outlets: { 'sidebar': ['component-aux'] } }]">
  Component Aux
</a>
```

[View Example](#)

Each auxiliary route is an independent route which can have:

- its own child routes
- its own auxiliary routes
- its own route-params
- its own history stack

State Management

For larger Angular applications with a lot of asynchronous activity and where there's a lot of state that is being shared and manipulated across multiple components and modules, managing state can be quite challenging. In a typical application, we're managing things like:

- Data that comes from the server and whether it's pending or resulted in an error
- UI state like toggles, alerts and errors messages
- User input, such as form submissions, filters and search queries
- Custom themes, credentials and localization
- Many other types of state

As the application grows, how do we know that a state change in one module will consistently and accurately reflected in other modules? And what if these modifications result in even more state changes? Eventually, it becomes extremely difficult to reason about what's actually happening in your application, and be a large source of bugs.

In Angular, there are 3 main ways to solve this problem.

1. [Redux using @ngrx](#);
2. Redux using ng2-redux; and
3. Angular Services and RxJS.

Redux and @ngrx

What is Redux?

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

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

What is @ngrx?

Redux state managers have been very well received and have inspired the creation of [@ngrx](#), a set of modules that implement the same way of managing state as well as some of the middleware and tools in the Redux ecosystem. @ngrx was created to be used specifically with Angular and [RxJS](#), as it leans heavily on the observable paradigm.

We'll describe how to use this approach in an application.

For further on Redux and @ngrx see the [Further reading](#) section

Adding @ngrx to your Project

In your console, run the following command to add [@ngrx](#) to your list of dependencies in

`package.json` :

```
npm install @ngrx/core @ngrx/store --save
```

If you plan on using the [@ngrx/effects](#) extensions to add side-effect capabilities, then also run the following command:

```
npm install @ngrx/effects --save
```

Defining your Main Application State

When building an application using Redux, the first thing to think about is, "What state do I want to store?" It is generally a good idea to capture all of the application's state so that it can be accessible from anywhere and all in one place for easy inspection.

In the application state, we store things like:

- Data received through API calls
- User input
- Presentation state, such as menu and button toggles
- Application preferences
- Internationalization messages
- Themes and other customizable areas of your application

To define your application state, use an interface called `AppState` or `IAppState`, depending on the naming conventions used on your project.

Here's an example:

app/models/appState.ts

```
export interface AppState {
  readonly colors: Colors;
  readonly localization: Localization;
  readonly login: Login;
  readonly projectList: ProjectList;
  readonly registration: Registration;
  readonly showMainNavigation: boolean;
}
```

Note: We're using `readonly` to ensure compile-time immutability, and it provides the simplest immutable implementation without adding more dependencies to clutter the examples. However, feel free to use another approach on your project that makes sense for your team.

Example Application

In this chapter, you'll be creating a simple counter application using [@ngrx](#). Your app will allow users to increment and decrement a number by one, as well as reset that value back to zero. Here's the `AppState` that we'll be using throughout the example:

app/models/appState.ts

```
import {Counter} from './counter';

export interface AppState {
  readonly counter: Counter;
}
```

app/models/counter.ts

```
export interface Counter {
  readonly currentValue: number;
}
```

It's good practice to declare each interface in its own file, and create a logical directory structure if you have seven or more interfaces used by your application.

Reading your Application State using Selectors

To read your application state in Redux, we need to use the `select()` method on `@ngrx's` `Store` class. This method creates and returns an `Observable` that is bound to a specific property in your application state.

For example, here's how you would select the `counter` object:

```
store.select('counter'); // Returns Observable<Counter>
```

And to fetch the counter's `currentValue`, we can pass in a `string` array, where each string plucks a single property from the application state one at a time in the order specified:

```
store.select(['counter', 'currentValue']); // Returns Observable<number>
```

While `select()` allows for several variations of strings to be passed in, it has its shortcomings - namely you won't actually know if the plucking is working properly until you execute your code.

Because of that, `select()` allows you to select values using functions too, which makes things more type-safe and your selectors will be more refactorable by your IDE.

```
store.select(appState => appState.counter.currentValue);
```

Creating a Counter Service

While you could inject `Store` and select values directly in your Angular components, it's considered to be a best practice to wrap this functionality into separate services. This approach encapsulates all of the selection logic and eliminates any duplication where the selection path is repeated throughout your application.

Let's tie everything together by building out a `CounterService` example:

```
app/services/counter.service.ts
```



```
import {Injectable} from '@angular/core';
import {Store} from '@ngrx/store';
import {Observable} from 'rxjs/Observable';

import {AppState} from '../models';

@Injectable()
export class CounterService {

  constructor(private store: Store<AppState>) {}

  getCurrentValue(): Observable<number> {
    return this.store.select(appState => appState.counter.currentValue)
      .filter(Boolean);
  }
}
```

Because `select()` returns an `Observable`, the `getCurrentValue()` method also applies a `filter()` to ensure that subscribers do not receive any *falsy* values. This greatly simplifies the code and templates in your components, since they don't have to repeatedly consider the falsy case everywhere the value is used.

Actions

Redux uses a concept called Actions, which describe state changes to your application. Redux actions are simple JSON objects that implement the `Action` interface provided by [@ngrx](#):

```
export interface Action {  
  type: string;  
  payload?: any;  
}
```

The `type` property is a string used to uniquely identify your action to your application. It's a common convention to use *lisp-case* (such as `MY_ACTION`), however you are free to use whatever casing style that makes to your team, as long as it's consistent across the project.

The `payload` property provides a way to pass additional data to other parts of Redux, and it's entirely optional.

Here is an example:

```
const loginSendAction: Action = {  
  type: 'LOGIN_SEND',  
  payload: {  
    username: 'katie',  
    password: '35c0cd1ecbbb68c75498b83c4e79fe2b'  
  }  
};
```

Plain objects are used so that the actions are serializable and can be replayable into the application state. Even if your actions involve asynchronous logic, the final dispatched action will remain a plain JSON object.

To simplify action creation, you can create a factory function to take care of the repeating parts within your application:

app/store/createAction.ts

```
import {Action} from '@ngrx/store';  
  
export function createAction(type, payload?): Action {  
  return { type, payload };  
}
```

The resulting creation of the `LOGIN_SEND` action becomes much more succinct and cleaner:

```
const loginSendAction: Action = createAction('LOGIN_SEND', {
  username: 'katie',
  password: '35c0cd1ecbbb68c75498b83c4e79fe2b'
});
```

Modifying your Application State by Dispatching Actions

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

In Angular, it's convenient to define your action creators as `@Injectable()` services, decoupling the dispatch, creation and side-effect logic from the `@Component` classes in your application.

Synchronous Actions

Here is a simple example:

app/store/counter/counter.actions.ts

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

import {createAction} from '../createAction';
import {AppState} from '../../models/appState';

@Injectable()
export class CounterActions {

  static INCREMENT = 'INCREMENT';
  static DECREMENT = 'DECREMENT';
  static RESET = 'RESET';

  constructor(private store: Store<AppState>) {

  }

  increment() {
    this.store.dispatch(createAction(CounterActions.INCREMENT));
  }

  decrement() {
    this.store.dispatch(createAction(CounterActions.DECREMENT));
  }

  reset() {
    this.store.dispatch(createAction(CounterActions.RESET));
  }

}
```

As you can see, the action creators are simple functions that dispatch `Action` objects containing more information that describes the state modification.

Asynchronous Actions

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

app/store/counter/counter.actions.ts

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

import {createAction} from '../createAction';
import {AppState} from '../../models/appState';

@Injectable()
export class CounterActions {

  constructor(private store: Store<AppState>) {

  }

  incrementIfOdd() {
    this.store.select(appState => appState.counter.currentValue)
      .take(1)
      .subscribe(currentValue => {
        if (currentValue % 2 !== 0) {
          this.store.dispatch(createAction(CounterActions.INCREMENT));
        }
      });
  }

  incrementAsync(timeInMs: number = 1000) {
    this.delay(timeInMs).then(() => this.store.dispatch(createAction(CounterActions.INCREMENT)));
  }

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

In the `incrementIfOdd()` action creator, we create a one-time subscription to the counter's `currentValue` in the application state. From there, we check to see if it's odd before dispatching an action.

In the `incrementAsync()` action creator, we are delaying the actual call to `dispatch()`. We created a `Promise` that will resolve after the delay. Once the `Promise` resolves, we can then dispatch an action to increment the counter.

Actions that Depend on Other Services

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

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

import {createAction} from '../createAction';
import {AppState} from '../../models/appState';

@Injectable()
export class SessionActions {

    static LOGIN_USER_PENDING = 'LOGIN_USER_PENDING';
    static LOGIN_USER_SUCCESS = 'LOGIN_USER_SUCCESS';
    static LOGIN_USER_ERROR = 'LOGIN_USER_ERROR';
    static LOGOUT_USER = 'LOGOUT_USER';

    constructor(
        private store: Store<AppState>,
        private authService: AuthService
    ) {

    }

    loginUser(credentials: any) {
        this.store.dispatch(createAction(SessionActions.LOGIN_USER_PENDING));

        this.authService.login(credentials.username, credentials.password)
            .then(result => this.store.dispatch(createAction(SessionActions.LOGIN_USER_SUCCESS, result)))
            .catch(() => this.store.dispatch(createAction(SessionActions.LOGIN_USER_ERROR)));
    };

    logoutUser() {
        this.store.dispatch(createAction(SessionActions.LOGOUT_USER));
    };
}
```

Review of Reducers and Pure Functions

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

A simple example of a reducer is the sum function:

```
let x = [1, 2, 3].reduce((sum, number) => sum + number, 0);  
// x == 6
```


Reducers as State Management

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

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

Simple Reducer

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

app/store/counter/counter.reducer.ts

```
import {Action} from '@ngrx/store';

import {CounterActions} from './counter.actions';

export default function counterReducer(state: number = 0, action: Action): number {
  switch (action.type) {
    case CounterActions.INCREMENT:
      return state + 1;
    case CounterActions.DECREMENT:
      return state - 1;
    case CounterActions.RESET:
      return 0;
    default:
      return state;
  }
}
```

We can see here that we are passing in an initial state (the current number) and an `Action`. To handle each action, a common approach is to use a `switch` statement. Instead of each reducer needing to explicitly subscribe to the dispatcher, every action gets passed into each reducer, which handles the actions it's interested in and then returns the new state along to the next reducer.

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

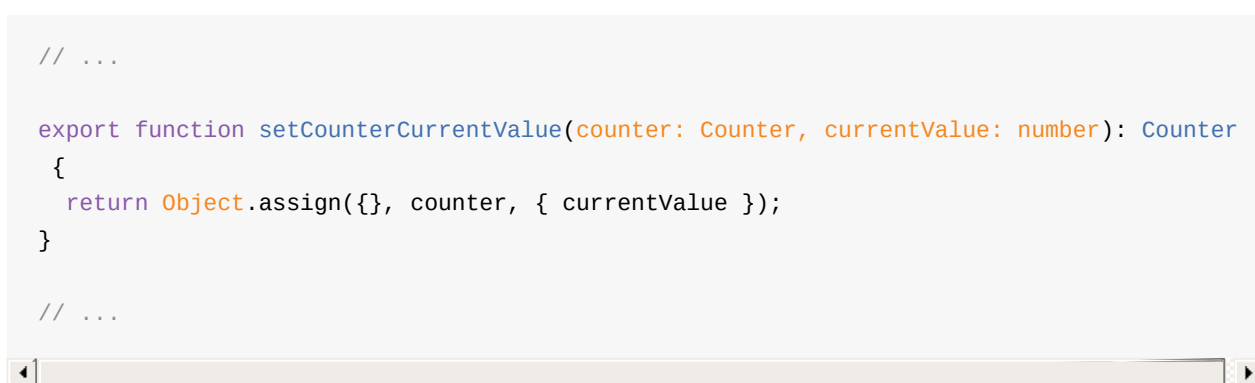
For this reason, side-effect causing operations, such as updating a record in a database, generating an id, etc. should be handled elsewhere in the application, like in your action creators or using [@ngrx/effects](#).

Complex Reducer

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

app/models/counter.ts

```
// ...  
  
export function setCounterCurrentValue(counter: Counter, currentValue: number): Counter  
{  
  return Object.assign({}, counter, { currentValue });  
}  
  
// ...
```



Here, the `setCounterCurrentValue()` function creates a new `Counter` object that overwrites the `counter.currentValue` property with a new value while maintaining the references and values of all of the other properties from `counter`.

Let's update our reducer to utilize this concept:

```
import {Action} from '@ngrx/store';

import {Counter, createDefaultCounter, setCounterCurrentValue} from '../../../models/counter';
import {CounterActions} from './counter.actions';

export function counterReducer(
  counter: Counter = { currentValue: 0 },
  action: Action
): Counter {
  switch (action.type) {
    case CounterActions.INCREMENT:
      return setCounterCurrentValue(counter, counter.currentValue + 1);

    case CounterActions.DECREMENT:
      return setCounterCurrentValue(counter, counter.currentValue - 1);

    case CounterActions.RESET:
      return setCounterCurrentValue(counter, 0);

    default:
      return counter;
  }
}
```

With each action, we take the existing `counter` state and create a new state with the updated value (such as `counter.currentValue + 1`).

When dealing with complex or deeply nested objects, it can be difficult to maintain immutability in your application using this syntax. This is where a library like [Ramda](#) can help.

Creating your Application's Root Reducer

[@ngrx](#) allows us to break our application into smaller reducers with a single area of concern.

We can combine these reducers by creating an object that mirrors the application's

`AppState` , where each property will point to one of those smaller reducers.

app/store/rootReducer.ts

```
import {counterReducer} from './counter/counter.reducer';

export const rootReducer = {
  counter: counterReducer
};
```

Configuring your application

Once you have your reducers created, it's time to configure your Angular application. In your main application module, simply add the `StoreModule.provideStore()` call to your

`@NgModule` 's imports:

app/app.module.ts

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {FormsModule} from '@angular/forms';
import {HttpClientModule} from '@angular/http';
import {StoreModule} from '@ngrx/store';
import {EffectsModule} from '@ngrx/effects';

import 'rxjs/Rx';

import {rootReducer} from './store/rootReducer';
import {CounterActions} from './store/actions';
import {CounterEffects} from './store/effects';
import {AppComponent, CounterComponent} from './components';
import {CounterService} from './services';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    StoreModule.provideStore(rootReducer)
  ],
  declarations: [
    AppComponent,
    CounterComponent
  ],
  providers: [
    CounterActions,
    CounterService
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```


Implementing Components

To demonstrate how to use the `CounterService` in your components, let's start by building out a small `CounterComponent`. The component will be responsible for incrementing and decrementing the counter by one, as well as allowing the user to reset the counter to zero.

app/components/counter.component.ts

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

import {CounterService} from '../services';
import {CounterActions} from '../store/counter/counter.actions';

@Component({
  selector: 'counter',
  templateUrl: './counter.component.html'
})
export class CounterComponent {

  private currentValue$: Observable<number>;

  constructor(
    counterService: CounterService,
    public actions: CounterActions
  ) {
    this.currentValue$ = counterService.getCurrentValue();
  }
}
```

app/components/counter.component.html

```
<p>
  Clicked: {{currentValue$ | async}} times
  <button (click)="actions.increment()">+</button>
  <button (click)="actions.decrement()">-</button>
  <button (click)="actions.reset()">Reset</button>
</p>
```

The template syntax should be familiar by now, displaying an `Observable` counter value with the `async` pipe. Any time `AppState.counter.currentValue` is updated by a reducer, `currentValue$` will receive the new value and `| async` will update it in the template.

The component also handles some click events. Each click event is bound to expressions that call our action creators from the `CounterActions` `ActionCreatorService`.

Component Architecture

Our previous `CounterComponent` example is called a **smart component** - it knew about Redux, the structure of the state and the actions it needed to call. In theory you can drop this component into any area of your application and just let it work, but it will be tightly bound to that specific slice of state and those specific actions.

For example, what if we wanted to have multiple counters tracking different things on the page? Or how about counting the number of red clicks vs blue clicks?

To help make components more generic and reusable, it's worth trying to separate them into *container* components and *presentational* components.

	Container Components	Presentational Components
Location	Top level, route handlers	Middle and leaf components
Aware of Redux	Yes	No
To read data	Subscribe to Redux state	Read state from <code>@Input</code> properties
To change data	Dispatch Redux actions	Invoke callbacks from <code>@Output</code> properties

[redux docs](#)

Keeping this in mind, let's refactor our `CounterComponent` to be a *presentational* component.

Modifying `AppComponent` to become a smart component

First, let's modify our top-level application component to use the `CounterService` and `CounterActions`, just as `CounterComponent` did:

`app/app.component.ts`

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

import {Counter} from '../../models/counter';
import {CounterService} from '../../services/counter.service';
import {CounterActions} from '../../store/counter/counter.actions';

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

  counter$: Observable<Counter>;

  constructor(
    counterService: CounterService,
    public actions: CounterActions
  ) {
    this.counter$ = counterService.getCounter();
  }
}
```

Now our `AppComponent` is a smart-component, because it's aware of Redux, it's presence in the application state and the underlying services. As with previous examples, we can use the `async` pipe to obtain the most recent `counter` value and pass it along to other components within the template.

And while we haven't looked at the `@Output()`'s on `CounterComponent` just yet, we'll want to delegate those events to our action creators in `CounterActions`.

app/app.component.html

```
<counter [counter]="counter$ | async"
  (onIncrement)="actions.increment()"
  (onDecrement)="actions.decrement()"
  (onReset)="actions.reset()">
</counter>
```

Modifying `CounterComponent` to become a presentation component

In turn, we need to make the `CounterComponent` from a *smart component* into a *dumb component*. For this, we will pass the data into the component using `@Input` properties and click events using `@Output()` properties, removing the use of `CounterService` and `CounterActions` entirely.

app/counter/counter.component.ts

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

import {Counter} from '../../models/counter';

@Component({
  selector: 'counter',
  templateUrl: './counter.component.html'
})
export class CounterComponent {

  @Input()
  counter: Counter;

  @Output()
  onIncrement: EventEmitter<void> = new EventEmitter<void>();

  @Output()
  onDecrement: EventEmitter<void> = new EventEmitter<void>();

  @Output()
  onReset: EventEmitter<void> = new EventEmitter<void>();

}
```

Our child components become much simpler and testable, because we don't have to use the `async` pipe to work with our state, which removes a lot of pain when dealing with lots of `@Input` 's or the need to use complex expressions with `observable` 's.

We can also now simply use core Angular features to emit values whenever a click event happens:

app/counter/counter.component.html

```
<p>
  Clicked: {{counter.currentValue}} times
  <button (click)="onIncrement.emit()">+</button>
  <button (click)="onDecrement.emit()">-</button>
  <button (click)="onReset.emit()">Reset</button>
</p>
```

We now have a nicely-reusable presentational component with no knowledge of Redux or our application state.

Side Effects

Often times, we need to perform some logic after an action has been dispatched and the store has been updated. Because reducers should be side-effect free, we need a way to handle these side-effects. Sometimes we can put this logic with our action creator services, and that works for simple cases, but often times the same block of logic needs to run in response to multiple action types.

[@ngrx](#) offers a library called [@ngrx/effects](#) to solve these problems.

Creating your first effects service

Let's say we have an application that supports customizable themes and other customization options. To support this functionality, we'll need to fetch these customizations from the server whenever a user logs into the application.

Let's build out a `CustomizationEffects` service to accomplish this:

```
import {Injectable} from '@angular/core';
import {Action} from '@ngrx/store';
import {Actions, Effect} from '@ngrx/effects';
import {Observable} from 'rxjs/Observable';

import {createAction} from '../createAction';
import {CustomizationActions, SessionActions} from '../session/session.actions.ts';
import {ApiService} from '../services';

@Injectable()
export class CustomizationEffects {

  constructor(
    private actions$: Actions,
    private apiService: ApiService
  ) {

  }

  @Effect()
  login$ = this.actions$
    .ofType(SessionActions.LOGIN_SEND_SUCCESS)
    .mergeMap<Action>(action => this.apiService.getCustomizations(action.payload.userId)
      .map(result => createAction(CustomizationActions.CUSTOMIZATIONS_RETRIEVE_SUCCESS, result.json()))
      .catch(error => Observable.of(createAction(CustomizationActions.CUSTOMIZATIONS_RETRIEVE_ERROR, error.json()))))
    );
}
```

[@ngrx/effects](#) provides an Angular `actions$` service (which is also an `Observable`) to emit every action that has been dispatched by your application in a single stream. Its `ofType()` method can be used to filter the one or more actions we're interested in before adding a side-effect.

In this example, when the `LOGIN_SEND_SUCCESS` action occurs from anywhere in the application, we make a request to the server to fetch the user's customizations given their `userId`, which has been attached to the payload of this action. Regardless if this request succeeds or fails, we need to create and return an `Observable` that is bound to the new action we'd like Redux to perform, such as storing the customizations when the request is successful, or processing the error returned by the server if there's an error.

To tell [@ngrx/effects](#) which `Observable` objects are side-effects to associate them with Redux, we need to provide a hint using the `@Effect()` decorator. Without it, your side-effect will not run.

Configuring your effects service

Lastly, we just need to add the `CustomizationEffects` service as a module to the `@NgModule`'s imports to start the effects:

```
// ...
import {EffectsModule} from '@ngrx/effects';

import {CustomizationEffects} from './store/customization/customization.effects';

@NgModule({
  imports: [
    // ...
    EffectsModule.run(CustomizationEffects)
  ],
  // ...
})
export class AppModule { }
```

Getting more from Redux and @ngrx

Redux

Redux has a number of tools and middleware available in its ecosystem to facilitate elegant app development.

- [Redux DevTools](#) - a tool that displays a linear timeline of actions that have interacted with its store. Allows for replaying actions and error handling
- [redux-thunk](#) - middleware that enables lazy evaluation of actions
- [redux-observable](#) - an RxJS-based model for handling side-effects on action streams.
- [*ng2-redux-router](#) - reactive glue between the Angular router and your redux store.

@ngrx

@ngrx provides most of its Redux implementation through the [ngrx/store](#) module. Other modules are available for better integration and development.

- [ngrx/store-devtools](#) - an ngrx implementation of the Redux DevTools
- [ngrx/effects](#) - a model for performing side-effects similar to `redux-saga`
- [ngrx/router](#) and [ngrx/router-store](#) - a router for Angular that can be connected to your ngrx store

TDD Testing

Test-Driven-Development is an engineering process in which the developer writes an initial automated test case that defines a feature, then writes the minimum amount of code to pass the test and eventually refactors the code to acceptable standards.

A *unit test* is used to test individual components of the system. An *integration test* is a test which tests the system as a whole, and how it will run in production.

Unit tests should only verify the behavior of a specific unit of code. If the unit's behavior is modified, then the unit test would be updated as well. Unit tests should not make assumptions about the behavior of other parts of your codebase or your dependencies. When other parts of your codebase are modified, your unit tests should not fail. (Any failure indicates a test that relies on other components and is therefore not a unit test.) Unit tests are cheap to maintain and should only be updated when the individual units are modified. For TDD in Angular, a unit is most commonly defined as a class, pipe, component, or service. It is important to keep units relatively small. This helps you write small tests which are "self-documenting", where they are easy to read and understand.

The Testing Toolchain

Our testing toolchain will consist of the following tools:

- Jasmine
- Karma
- Phantom-js
- Istanbul
- Sinon
- Chai

[Jasmine](#) is the most popular testing framework in the Angular community. This is the core framework that we will write our unit tests with.

[Karma](#) is a test automation tool for controlling the execution of our tests and what browser to perform them under. It also allows us to generate various reports on the results. For one or two tests this may seem like overkill, but as an application grows larger and the number of units to test grows, it is important to organize, execute and report on tests in an efficient manner. Karma is library agnostic so we could use other testing frameworks in combination with other tools (like code coverage reports, spy testing, e2e, etc.).

In order to test our Angular application we must create an environment for it to run in. We could use a browser like Chrome or Firefox to accomplish this (Karma supports in-browser testing), or we could use a browser-less environment to test our application, which can offer us greater control over automating certain tasks and managing our testing workflow.

[PhantomJS](#) provides a JavaScript API that allows us to create a headless DOM instance which can be used to bootstrap our Angular application. Then, using that DOM instance that is running our Angular application, we can run our tests.

[Istanbul](#) is used by Karma to generate code coverage reports, which tells us the overall percentage of our application being tested. This is a great way to track which components/services/pipes/etc. have tests written and which don't. We can get some useful insight into how much of the application is being tested and where.

For some extra testing functionality we can use the [Sinon](#) library for things like test spys, test subs and mock XHR requests. This is not necessarily required as Jasmine comes with the `spyOn` function for incorporating spy tests.

[Chai](#) is an assertion library that can be paired with any other testing framework. It offers some syntactic sugar that lets us write our unit tests with different verbiage - we can use a *should*, *expect* or *assert* interface. Chai also takes advantage of "function chaining" to form

English-like sentences used to describe tests in a more user friendly way. Chai isn't a required library for testing and we won't explore it much more in this handout, but it is a useful tool for creating cleaner, more well-written tests.

Test Setup

The repo [angular2-redux-starter](#) is a basic webpack-based Angular application (with Redux) with the same testing toolchain outlined above. Let's take a look at how this project is set up.

Filename Conventions

Each unit test is put into its own separate file. The Angular team recommends putting unit test scripts alongside the files they are testing and using a `.spec` filename extension to mark it as a testing script (this is a Jasmine convention). So if you had a component `/app/components/mycomponent.ts`, then your unit test for this component would be in `/app/components/mycomponent.spec.ts`. This is a matter of personal preference; you can put your testing scripts wherever you like, though keeping them close to your source files makes them easier to find and gives those who aren't familiar with the source code an idea of how that particular piece of code should work.

Karma Configuration

Karma is the foundation of our testing workflow. It brings together our other testing tools to define the framework we want to use, the environment to test under, the specific actions we want to perform, etc. In order to do this Karma relies on a configuration file named by default *karma.conf.js*.

You can seed a new configuration file though the `karma init` command, which will guide you through a few basic questions to get a bare minimum setup running.

Overview

The configuration file is put together by exporting a function that accepts the configuration object that Karma is going to work with. Modifying certain properties on this object will tell Karma what it is we want to do. Let's go over some of the key properties used in this configuration file:

```
module.exports = (config) => {
  const coverage = config.singleRun ? ['coverage'] : [];

  config.set({
    frameworks: [...],
    plugins: [ ... ],
    files: [ ... ],
    preprocessors: { ... },

    webpack: { ... },

    reporters: [ ... ],
    coverageReporter: { ... },

    port: 9999,
    browsers: ['Chrome'], // Alternatively: 'PhantomJS'
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    captureTimeout: 6000,
  });
};
```

frameworks

```
frameworks: [  
  'jasmine',  
],
```

`frameworks` is a list of the testing frameworks we want to use. These frameworks must be installed through NPM as a dependency in our project or/and as a Karma plugin.

plugins

```
plugins: [  
  'karma-jasmine',  
  'karma-webpack',  
  'karma-coverage',  
  'karma-remap-istanbul',  
  'karma-chrome-launcher',  
],
```

Plugins that integrate karma with testing frameworks like Jasmine or build systems like Webpack.

files

```
files: [  
  './src/tests.entry.ts',  
  {  
    pattern: '**/*.map',  
    served: true,  
    included: false,  
    watched: true,  
  },  
],
```

`files` is a list of files to be loaded into the browser/testing environment. These are loaded sequentially, so order matters. The file list can also take the form of a glob pattern as it becomes rather tedious to manually add in a new file for each new testing script created.

In the angular2-redux-starter *karma.conf.js* we have put the testing files we wish to include in a separate file - *src/tests.entry.ts*, which includes a `require` call using a regex pattern for importing files with the `.spec.ts` file extension. As a project grows larger and the number of files to include grows in complexity it is good practice to put file imports in a separate file - this keeps the *karma.conf.js* file cleaner and more readable. Here is what our *src/tests.entry.ts* looks like:

```
let testContext = (<{ context?: Function }>require).context('./', true, /\.test\.ts/);
testContext.keys().forEach(testContext);
```

preprocessors

```
preprocessors: {
  './src/tests.entry.ts': [
    'webpack',
    'sourcemap',
  ],
  './src/**/!(*.test|tests.*).(ts|js)': [
    'sourcemap',
  ],
}
```

`preprocessors` allow for some operation to be performed on the contents of a unit testing file before it is executed. These operations are carried out through the use of Karma plugins and are often used for transpiling operations. Since we are writing unit tests in TypeScript, `.ts` files must be transpiled into plain Javascript in order to run in a browser-based environment.

In `angular2-redux-starter` this process is done with `webpack`, so we explicitly invoke the `webpack` processor on all of our testing files (those ending with `.spec.ts`). We also load any source map files originating from transpilation through the `sourcemap` processor.

webpack


```
webpack: {
  plugins,
  entry: './src/tests.entry.ts',
  devtool: 'inline-source-map',
  resolve: {
    extensions: ['.webpack.js', '.web.js', '.ts', '.js'],
  },
  module: {
    rules:
      combinedLoaders().concat(
        config.singleRun
          ? [ loaders.istanbulInstrumenter ]
          : [ ]),
  },
  stats: { colors: true, reasons: true },
},
webpackServer: {
  noInfo: true, // prevent console spamming when running in Karma!
}
```

If the project uses webpack, then the property `webpack` in the Karma configuration object is where we can configure webpack with Karma. In the `angular2-redux-starter`, plugins and loaders are exported from their own files to be imported by both the webpack config and the karma config, making the configuration object smaller.

Using webpack, we can configure how to bundle our unit tests; that is, whether to pack all tests into a single bundle, or each unit test in its own bundle, etc. Regardless, unit tests should not be bundled with the rest of the applications code (especially in production!). In `angular2-redux-starter` we have opted to bundle all unit tests together.

coverageReporters and reporters

```
reporters: ['spec']
  .concat(coverage)
  .concat(coverage.length > 0 ? ['karma-remap-istanbul'] : []),

remapIstanbulReporter: {
  src: 'coverage/chrome/coverage-final.json',
  reports: {
    html: 'coverage',
  },
},

coverageReporter: {
  reporters: [
    { type: 'json' },
  ],
  dir: './coverage/',
  subdir: (browser) => {
    return browser.toLowerCase().split(/[-]/)[0]; // returns 'chrome'
  },
},
```

`coverageReporter` is used to configure the output of results of our code coverage tool (our toolchain uses Istanbul). Here we have specified to output the results in JSON and HTML. Reports will appear in the `coverage/` folder.

`reporters` is a list of reporters to use in the test cycle. Reporters can be thought of as modular tools used to report on some aspect of the testing routine outside of the core unit tests. Code coverage is an example of a reporter - we want it to report on how much of our code is being tested. [There are many more reporters available for Karma](#) that can aid in crafting your testing workflow.

Environment configuration

```
port: 9999,
browsers: ['Chrome'], // Alternatively: 'PhantomJS'
colors: true,
logLevel: config.LOG_INFO,
autoWatch: true,
captureTimeout: 6000,
```

`port`, `browsers` and `singleRun` configure the environment our unit tests will run under. The `browsers` property specifies which browser we want Karma to launch and capture output from. We can use Chrome, Firefox, Safari, IE or Opera (requires additional Karma launcher to be installed for each respective browser). For a browser-less DOM instance we can use PhantomJS (as outlined in the toolchain section).

We can also manually capture output from a browser by navigating to

`http://localhost:port` , where `port` is the number specified in the `port` property (the default value is 9876 if not specified). The property `singleRun` controls how Karma executes, if set to `true` , Karma will start, launch configured browsers, run tests and then exit with a code of either `0` or `1` depending on whether or not all tests passed.

Completed Configuration

The net result of customizing all of these properties is the [karma.conf.js file in angular-redux-starter](#).

Additional Resources

This is just a sample of the core properties in `karma.conf.js` being used by `angular2-redux-starter` project. There are many more properties that can be used to extend and configure the functionality of Karma - [take a look at the official documentation for the full API breakdown](#).

TestBed Configuration (Optional)

As you will see in [Testing Components](#), real-world component testing often relies on the Angular2 testing utility `TestBed`, which requires some configuration. Most significantly, we need to use `TestBed.initTestEnvironment` to create a testing platform before we can use unit tests with `TestBed`. This testing environment would have to be created, destroyed and reset as appropriate before every unit test.

In the `angular2-redux-starter`, this configuration is done in a `tests.configure.ts` file and imported into every unit test for easy re-use.

```
import {
  getTestBed,
  ComponentFixtureAutoDetect,
  TestBed,
} from '@angular/core/testing';

import {
  BrowserDynamicTestingModule,
  platformBrowserDynamicTesting,
} from '@angular/platform-browser-dynamic/testing';

export const configureTests = (configure: (testBed: TestBed) => void) => {
  const testBed = getTestBed();

  if (testBed.platform == null) {
    testBed.initTestEnvironment(
      BrowserDynamicTestingModule,
      platformBrowserDynamicTesting());
  }

  testBed.configureCompiler({
    providers: [
      {provide: ComponentFixtureAutoDetect, useValue: true},
    ]
  });

  configure(testBed);

  return testBed.compileComponents().then(() => testBed);
};
```

`tests.configure.ts` creates the testing platform if it doesn't already exist, compiles the template, and exports `configureTests` which can then be imported and used in our unit tests.

Here's a look at how it would be used:

```
import { TestBed } from '@angular/core/testing';
import { ExampleComponent } from './index';
import { configureTests } from '../tests.configure';
import { AppModule } from '../modules/app.module';

describe('Component: Example', () => {
  let fixture;

  beforeEach(done => {
    const configure = (testBed: TestBed) => {
      testBed.configureTestingModule({
        imports: [AppModule],
      });
    };

    configureTests(configure).then(testBed => {
      fixture = testBed.createComponent(ExampleComponent);
      fixture.detectChanges();
      done();
    });
  });
});
```

Typings

Since our project and unit tests are written in TypeScript, we need type definitions for the libraries we'll be writing our tests with (Chai and Jasmine). In [angular2-redux-example](#) we have included these type definitions from `@types` .

If you're following the example of `angular2-redux-starter` and using a `tests.entry` file to specify your project testing requirements, bear in mind you'll also need to add node typings to your dependencies.

Executing Test Scripts

Our entire testing workflow is done through Karma. Run the command `karma start` to kickstart Karma into setting up the testing environment, running through each unit test and executing any reporters we have set up in the `karma.config.js` configuration file. In order to run Karma through the command line it must be installed globally (`npm install karma -g`).

A good practice is to amalgamate all the project's task/build commands through npm. This gives continuity to your build process and makes it easier for people to test/run your application without knowing your exact technology stack. In `package.json` the `scripts` field holds an object with key-value pairing, where the key is the alias for the command and the value is the command to be executed.

```
...
"scripts": {
  "test": "karma start",
  ...
}
...
```

Now running `npm test` will start Karma. Below is the output of our Karma test. As you can see we had one test that passed, running in a Chrome 48 browser.

A terminal window showing the output of a Karma test. The output consists of four lines of log messages: 1. '[karma]: Karma v0.13.19 server started at http://localhost:9999/' 2. '[launcher]: Starting browser Chrome' 3. '[Chrome 48.0.2564 (Mac OS X 10.10.5)]: Connected on socket /#Jj7R4JfuvdbF4ibVAAAA with id 43976422' 4. 'Chrome 48.0.2564 (Mac OS X 10.10.5): Executed 1 of 1 SUCCESS (0.005 secs / 0.002 secs)'. The prompt shows the user is in a directory named 'angular2-redux-starter' on a 'tyler@tylers-MacBook-Pro' machine.

```
01 02 2016 17:06:05.914:INFO [karma]: Karma v0.13.19 server started at http://localhost:9999/
01 02 2016 17:06:05.924:INFO [launcher]: Starting browser Chrome
01 02 2016 17:06:06.600:INFO [Chrome 48.0.2564 (Mac OS X 10.10.5)]: Connected on socket /#Jj7R4JfuvdbF4ibVAAAA with id 43976422
Chrome 48.0.2564 (Mac OS X 10.10.5): Executed 1 of 1 SUCCESS (0.005 secs / 0.002 secs)
tyler@tylers-MacBook-Pro angular2-redux-starter$
```

Figure: image

Simple Test

To begin, let's start by writing a simple test in Jasmine.

```
describe('Testing math', () => {  
  it('multiplying should work', () => {  
    expect(4 * 4).toEqual(16);  
  });  
});
```

Though this test may be trivial, it illustrates the basic elements of a unit test. We explain what this test is for by using `describe`, and we use `it` to assert what kind of result we are expecting from our test. These are user-defined so it's a good idea to be as descriptive and accurate in these messages as possible. Messages like "should work", or "testing service" don't really explain exactly what's going on and may be confusing when running multiple tests across an entire application.

Our actual test is basic, we use `expect` to formulate a scenario and use `toEqual` to assert the resulting condition we are expecting from that scenario. The test will pass if our assertion is equal to the resulting condition, and fail otherwise. You always want your tests to pass - do not write tests that have the results you want in a failed state.

Using Chai

Chai is an assertion library with some tasty syntax sugar that can be paired with any other testing framework. It lets us write tests in a TDD (Test Driven Development) style or BDD (Behavior Driven Development) style. We already know what TDD is (read the intro!), so what exactly is BDD? Well BDD is the combination of using TDD with natural language constructs (English-like sentences) to express the behavior and outcomes of unit tests. Jasmine already uses a TDD style, so we'll be using Chai for its BDD interfaces, mainly through the use of `should` and `expect`.

```
describe('Testing math', () => {
  it('multiplying should work', () => {
    let testMe = 16;

    // Using the expect interface
    chai.expect(testMe).to.be.a('number');
    chai.expect(testMe).to.equal(16);

    // Using the should interface
    chai.should();
    testMe.should.be.a('number');
    testMe.should.equal(16);
  });
});
```

The `expect` and `should` interface both take advantage of chaining to construct English-like sentences for describing tests. Once you've decided on a style you should maintain that style for all your other tests. Each style has its own unique syntax; refer to the [documentation for that specific API](#).

Testing Components

Testing Angular components requires some insight into the Angular `@angular/core/testing` module. Though many features of Jasmine are used in Angular's testing module there are some very specific wrappers and routines that Angular requires when testing components.

Verifying Methods and Properties

We can test the properties and methods of simple Angular components fairly easily - after all, Angular components are simple classes that we can create and interface with. Say we had a simple component that kept a defined message displayed. The contents of the message may be changed through the `setMessage` function, and the `clearMessage` function would put an empty message in place. This is a very trivial component but how would we test it?

message.component.ts

```
import {Component} from '@angular/core';

@Component({
  selector: 'display-message',
  template: '<h1>{{message}}</h1>'
})

export class MessageComponent {
  public message: string = '';

  constructor() {}

  setMessage(newMessage: string) {
    this.message = newMessage;
  }

  clearMessage() {
    this.message = '';
  }
}
```

Now for our unit test. We'll create two tests, one to test the `setMessage` function to see if the new message shows up and another to test the `clearMessage` function to see if clearing the message works as expected.

message.spec.ts

```
import {MessageComponent} from './message.component';

describe('Testing message state in message.component', () => {
  let app: MessageComponent;

  beforeEach(() => {
    app = new MessageComponent();
  });

  it('should set new message', () => {
    app.setMessage('Testing');
    expect(app.message).toBe('Testing');
  });

  it('should clear message', () => {
    app.clearMessage();
    expect(app.message).toBe('');
  });
});
```

[View Example](#)

We have created two tests: one for `setMessage` and the other for `clearMessage`. In order to call those functions we must first initialize the `MessageComponent` class. This is accomplished by calling the `beforeEach` function before each test is performed.

Once our `MessageComponent` object is created we can call `setMessage` and `clearMessage` and analyze the results of those actions. We formulate an expected result, and then test to see if the result we were expecting came to be. Here we are testing whether or not the message we tried to set modified the `MessageComponent` property `message` to the value we intended. If it did, then the test was successful and our `MessageComponent` works as expected.

Injecting Dependencies and DOM Changes

In the previous example the class we were testing, `MessageComponent`, did not have any injected dependencies. In Angular, components will often rely on services and other classes (pipes/providers/etc.) to function, which will be injected into the constructor of the components class. When testing these components we have to inject the dependencies ourselves. Since this is an Angular-specific routine, there are no pure Jasmine functions used to accomplish this. Angular provides a multitude of functions in `@angular/core/testing` that allows us to effectively test our components. Let's take a look at a basic component:

quote.component.ts

```
import { QuoteService } from './quote.service';
import { Component } from '@angular/core';

@Component({
  selector: 'my-quote',
  template: '<h3>Random Quote</h3> <div>{{quote}}</div>'
})

export class QuoteComponent {
  quote: string;

  constructor(private quoteService: QuoteService){};

  getQuote() {
    this.quoteService.getQuote().then((quote) => {
      this.quote = quote;
    });
  };
}
```

This component relies on the `QuoteService` to get a random quote, which it will then display. The class is pretty simple - it only has the `getQuote` function that will modify the DOM, therefore it will be our main area of focus in testing.

In order to test this component we need initiate the `QuoteComponent` class. The Angular testing library offers a utility called `TestBed`. This allows us to configure a testing module where we can provide mocked dependencies. Additionally it will create the component for us and return a *component fixture* that we can perform testing operations on.

quote.spec.ts

```
import { QuoteService } from './quote.service';
```

```
import { QuoteComponent } from './quote.component';
import { provide, destroyPlatform } from '@angular/core';
import {
  async,
  inject,
  TestBed,
} from '@angular/core/testing';
import {
  BrowserDynamicTestingModule,
  platformBrowserDynamicTesting
} from '@angular/platform-browser-dynamic/testing';

class MockQuoteService {
  public quote: string = 'Test quote';

  getQuote() {
    return Promise.resolve(this.quote);
  }
}

describe('Testing Quote Component', () => {

  let fixture;

  beforeEach(() => destroyPlatform());

  beforeEach(() => {
    TestBed.initTestEnvironment(
      BrowserDynamicTestingModule,
      platformBrowserDynamicTesting()
    );

    TestBed.configureTestingModule({
      declarations: [
        QuoteComponent
      ],
      providers: [
        { provide: QuoteService, useClass: MockQuoteService }
      ]
    });
    fixture = TestBed.createComponent(QuoteComponent);
    fixture.detectChanges();
  });

  it('Should get quote', async(inject([], () => {
    fixture.componentInstance.getQuote();
    fixture.whenStable()
      .then(() => {
        fixture.detectChanges();
        return fixture.whenStable();
      })
      .then(() => {
        const compiled = fixture.debugElement.nativeElement;
```

```
    expect(compiled.querySelector('div').innerText).toEqual('Test quote');
  });
}));
});
```

[View Example](#)

Testing the `QuoteComponent` is a fairly straightforward process. We want to create a `QuoteComponent`, feed it a quote and see if it appears in the DOM. This process requires us to create the component, pass in any dependencies, trigger the component to perform an action and then look at the DOM to see if the action is what we expected.

Let's take a look at how this is accomplished with the above unit test.

We use `TestBed.initTestingEnvironment` to create a testing platform using `BrowserDynamicTestingModule` and `platformBrowserDynamicTesting` as arguments, which are also imported from angular and allow the application to be bootstrapped for testing. This is necessary for all unit tests that make use of `TestBed`. Notice that this platform is destroyed and reset before each test runs.

We use `TestBed.configureTestingModule` to feed in any dependencies that our component requires. Here our component depends on the `QuoteService` to get data. We mock this data ourselves thus giving us control over what value we expect to show up. It is good practice to separate component testing from service testing - this makes it easier to test as you are only focusing on a single aspect of the application at a time. If your service or component fails, how will you know which one was the culprit? We inject the `QuoteService` dependency using our mock class `MockQuoteService`, where we will provide mock data for the component to consume.

Next we use `TestBed.createComponent(QuoteComponent)` to create a *fixture* for us to use in our tests. This will then create a new instance of our component, fulfilling any Angular-specific routines like dependency injection. A fixture is a powerful tool that allows us to query the DOM rendered by a component, as well as change DOM elements and component properties. It is the main access point of testing components and we use it extensively.

In the `Should get quote` test we have gotten access to our component through the `fixture.componentInstance` property. We then call `getQuote` to kickstart our only action in the `QuoteComponent` component. We run the test when the fixture is stable by using its `whenStable` method which will ensure the promise inside the `getQuote()` has resolved, giving the component a chance to set the quote value. We call `fixture.detectChanges` to keep an eye out for any changes taking place to the DOM, and use the `fixture.debugElement.nativeElement` property to get access to those underlying DOM elements.

Now we can check to see if the DOM rendered by our `QuoteComponent` contains the quote that we mocked in through the `QuoteService`. The final line attempts to assert that the DOM's div tag contains the mocked quote 'Test Quote' inside. If it does, then our component passes the test and works as expected; if it doesn't, that means our component is not outputting quotes correctly.

We wrap `should get quote` test in `async()`. This is to allow our tests run in an asynchronous test zone. Using `async` creates a test zone which will ensure that all asynchronous functions have resolved prior to ending the test.

Overriding Dependencies for Testing

`TestBed` provides several functions to allow us to override dependencies that are being used in a test module.

- `overrideModule`
- `overrideComponent`
- `overrideDirective`
- `overridePipe`

For example, you might want to override the template of a component. This is useful for testing a small part of a large component, as you can ignore the output from the rest of the DOM and only focus on the part you are interested in testing.

```
import {Component} from '@angular/core';

@Component({
  selector: 'display-message',
  template: `
    <div>
      <div>
        <h1>{{message}}</h1>
      <div>
    </div>
  `
})
export class MessageComponent {
  public message: string = '';

  setMessage(newMessage: string) {
    this.message = newMessage;
  }
}
```

```
import {MessageComponent} from './message.component';
import { provide } from '@angular/core';
import {
  async,
  inject,
  TestBed,
} from '@angular/core/testing';

describe('MessageComponent', () => {

  let fixture;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [MessageComponent],
      providers: []
    });

    fixture = TestBed.overrideComponent(MessageComponent, {
      set: {
        template: '<span>{{message}}</span>'
      }})
      .createComponent(MessageComponent);

    fixture.detectChanges();
  });

  it('should set the message', async(inject([], () => {
    fixture.componentInstance.setMessage('Test message');
    fixture.detectChanges();
    fixture.whenStable().then(() => {
      const compiled = fixture.debugElement.nativeElement;
      expect(compiled.querySelector('span').innerText).toEqual('Test message');
    });
  })));

});
```

[View Example](#)

Testing Asynchronous Actions

Sometimes we need to test components that rely on asynchronous actions that happen at specific times. Angular provides a function called `fakeAsync` which wraps our tests in a zone and gives us access to the `tick` function, which will allow us to simulate the passage of time precisely.

Let's go back to the example of the `QuoteComponent` component and rewrite the unit test using `fakeAsync` :

```
import { Component } from '@angular/core';
import { QuoteService } from './quote.service';

@Component({
  selector: 'my-quote',
  template: '<h3>Random Quote</h3> <div>{{quote}}</div>'
})

export class QuoteComponent {
  quote: string;

  constructor(private quoteService: QuoteService){};

  getQuote() {
    this.quoteService.getQuote().then((quote) => {
      this.quote = quote;
    });
  };
}
```

```
import { QuoteService } from './quote.service';
import { QuoteComponent } from './quote.component';
import { provide } from '@angular/core';
import {
  async,
  TestBed,
  fakeAsync,
  tick,
} from '@angular/core/testing';

class MockQuoteService {
  public quote: string = 'Test quote';

  getQuote() {
    return Promise.resolve(this.quote);
  }
}

describe('Testing Quote Component', () => {

  let fixture;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [
        QuoteComponent
      ],
      providers: [
        { provide: QuoteService, useClass: MockQuoteService }
      ]
    });
    fixture = TestBed.createComponent(QuoteComponent);
    fixture.detectChanges();
  });

  it('Should get quote', fakeAsync(() => {
    fixture.componentInstance.getQuote();
    tick();
    fixture.detectChanges();
    const compiled = fixture.debugElement.nativeElement;
    expect(compiled.querySelector('div').innerText).toEqual('Test quote');
  }));
});
```

[View Example](#)

Here we have a `QuoteComponent` that has a `getQuote` which triggers an asynchronous update. We have wrapped our entire test in `fakeAsync` which will allow us to test the asynchronous behavior of our component (`getQuote()`) using synchronous function calls by

calling `tick()` . We can then run `detectChanges` and query the DOM for our expected result.

Refactoring Hard-to-Test Code

As you start writing unit tests, you may find that a lot of your code is hard to test. The best strategy is often to refactor your code to make it easy to test. For example, consider refactoring your component code into services and focusing on service tests or vice versa.

Testing Services

When testing services in Angular we employ many of the same techniques and strategies used for testing components. Services, like components, are classes with methods and properties that we want to verify. Data is the main emphasis in testing services - are we getting, storing and propagating data correctly.

Testing Strategies for Services

When testing services that make HTTP calls, we don't want to hit the server with real requests. This is because we want to isolate the testing of our service from any other outside points of failure. Our service may work, but if the API server is failing or giving values we aren't expecting, it may give the impression that our service is the one failing. Also, as a project grows and the number of unit tests increase, running through a large number of tests that make HTTP requests will take a long time and may put strain on the API server. Therefore, when testing services we'll be mocking out fake data with fake requests.

Injecting Dependencies

Like components, services often require dependencies that Angular injects through the constructor of the service's class. Since we are initializing these classes outside the bootstrapping process of Angular, we must explicitly inject these dependencies ourselves. This is accomplished by using the `TestBed` to configure a testing module and feed in required dependencies like the HTTP module.

Testing HTTP Requests

Services, by their nature, perform asynchronous tasks. When we make an HTTP request we do so in an asynchronous manner so as not to block the rest of the application from carrying out its operations. We looked a bit at testing components asynchronously earlier - fortunately a lot of this knowledge carries over into testing services asynchronously.

The basic strategy for testing such a service is to verify the contents of the request being made (correct URL) and ensure that the data we mock into the service is returned correctly by the right method.

Let's take a look at some code:

wikisearch.ts

```
import {Http} from '@angular/http';
import {Injectable} from '@angular/core';
import {Observable} from 'rxjs';
import 'rxjs/add/operator/map'

@Injectable()
export class SearchWiki {
  constructor (private http: Http) {}

  search(term: string): Observable<any> {
    return this.http.get(
      'https://en.wikipedia.org/w/api.php?' +
      'action=query&list=search&srsearch=' + term
    ).map((response) => response.json());
  }

  searchXML(term: string): Observable<any> {
    return this.http.get(
      'https://en.wikipedia.org/w/api.php?' +
      'action=query&list=search&format=xmlfm&srsearch=' + term
    );
  }
}
```

Here is a basic service. It will query Wikipedia with a search term and return an `Observable` with the results of the query. The search function will make a GET request with the supplied term, and the searchXML method will do the same thing, except request the response to be in XML instead of JSON. As you can see, it depends on the HTTP module to make a request to wikipedia.org.

Our testing strategy will be to check to see that the service has requested the right URL, and once we've responded with mock data we want to verify that it returns that same data.

Testing HTTP Requests Using MockBackend

To unit test our services, we don't want to make actual HTTP requests. To accomplish this, we need to mock out our HTTP services. Angular provides us with a `MockBackend` class that can be configured to provide mock responses to our requests, without actually making a network request.

The configured `MockBackend` can then be injected into HTTP, so any calls to the service, such as `http.get` will return our expected data, allowing us to test our service in isolation from real network traffic.

wikisearch.spec.ts

```
import {
  fakeAsync,
  inject,
  TestBed
} from '@angular/core/testing';
import {
  HttpClientModule,
  XHRBackend,
  ResponseOptions,
  Response,
  RequestMethod
} from '@angular/http';
import {
  MockBackend,
  MockConnection
} from '@angular/http/testing/mock_backend';

import {SearchWiki} from './wikisearch.service';

const mockResponse = {
  "batchcomplete": "",
  "continue": {
    "sroffset": 10,
    "continue": "-||"
  },
  "query": {
    "searchinfo": {
      "totalhits": 36853
    },
    "search": [{
      "ns": 0,
      "title": "Stuff",
```

```
    "snippet": "<span></span>",
    "size": 1906,
    "wordcount": 204,
    "timestamp": "2016-06-10T17:25:36Z"
  }]
}
};

describe('Wikipedia search service', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpModule],
      providers: [
        {
          provide: XHRBackend,
          useClass: MockBackend
        },
        SearchWiki
      ]
    });
  });

  it('should get search results', fakeAsync(
    inject([
      XHRBackend,
      SearchWiki
    ], (mockBackend: XHRBackend, searchWiki: SearchWiki) => {

      const expectedUrl = 'https://en.wikipedia.org/w/api.php?' +
        'action=query&list=search&srsearch=Angular';

      mockBackend.connections.subscribe(
        (connection: MockConnection) => {
          expect(connection.request.method).toBe(RequestMethod.Get);
          expect(connection.request.url).toBe(expectedUrl);

          connection.mockRespond(new Response(
            new ResponseOptions({ body: mockResponse })
          ));
        });

      searchWiki.search('Angular')
        .subscribe(res => {
          expect(res).toEqual(mockResponse);
        });
    })
  ));

  it('should set foo with a 1s delay', fakeAsync(
    inject([SearchWiki], (searchWiki: SearchWiki) => {
      searchWiki.setFoo('food');
      tick(1000);
    })
  ));
});
```

```
    expect(searchWiki.foo).toEqual('food');
  })
});

});
```

[View Example](#)

We use `inject` to inject the `SearchWiki` service and the `MockBackend` into our test. We then wrap our entire test with a call to `fakeAsync`, which will be used to control the asynchronous behavior of the `SearchWiki` service for testing.

Next, we `subscribe` to any incoming connections from our back-end. This gives us access to an object `MockConnection`, which allows us to configure the response we want to send out from our back-end, as well as test any incoming requests from the service we're testing.

In our example, we want to verify that the `SearchWiki`'s search method makes a GET request to the correct URL. This is accomplished by looking at the request object we get when our `SearchWiki` service makes a connection to our mock back-end. Analyzing the `request.url` property we can see if its value is what we expect it to be. Here we are only checking the URL, but in other scenarios we can see if certain headers have been set, or if certain POST data has been sent.

Now, using the `MockConnection` object we mock in some arbitrary data. We create a new `ResponseOptions` object where we can configure the properties of our response. This follows the format of a regular [Angular Response class](#). Here we have simply set the `body` property to that of a basic search result set you might see from Wikipedia. We could have also set things like cookies, HTTP headers, etc., or set the `status` value to a non-200 state to test how our service responds to errors. Once we have our `ResponseOptions` configured we create a new instance of a `Response` object and tell our back-end to start using this as a response by calling `.mockRespond`.

It is possible to use multiple responses. Say your service had two possible GET requests - one for `/api/users`, and another `/api/users/1`. Each of these requests has a different corresponding set of mock data. When receiving a new connection through the `MockBackend` subscription, you can check to see what type of URL is being requested and respond with whatever set of mock data makes sense.

Finally, we can test the `search` method of the `SearchWiki` service by calling it and subscribing to the result. Once our search process has finished, we check the result object to see if it contains the same data that we mocked into our back-end. If it is, then congratulations, your test has passed.

In the `should set foo with a 1s delay` test, you will notice that we call `tick(1000)` which simulates a 1 second delay.

Alternative HTTP Mocking Strategy

An alternative to using `MockBackend` is to create our own light mocks. Here we create an object and then tell TypeScript to treat it as `Http` using type assertion. We then create a spy for its `get` method and return an observable similar to what the real `Http` service would do.

This method still allows us to check to see that the service has requested the right URL, and that it returns that expected data.

wikisearch.spec.ts

```
import {
  fakeAsync,
  inject,
  TestBed
} from '@angular/core/testing';
import {
  HttpClientModule,
  Http,
  ResponseOptions,
  Response
} from '@angular/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/of';
import { SearchWiki } from './wikisearch.service';

const mockResponse = {
  "batchcomplete": "",
  "continue": {
    "sroffset": 10,
    "continue": "-||"
  },
  "query": {
    "searchinfo": {
      "totalhits": 36853
    },
    "search": [{
      "ns": 0,
      "title": "Stuff",
      "snippet": "<span></span>",
      "size": 1906,
      "wordcount": 204,
      "timestamp": "2016-06-10T17:25:36Z"
    }]
  }
};
```

```
describe('Wikipedia search service', () => {
  let mockHttp: Http;

  beforeEach(() => {
    mockHttp = { get: null } as Http;

    spyOn(mockHttp, 'get').and.returnValue(Observable.of({
      json: () => mockResponse
    }));

    TestBed.configureTestingModule({
      imports: [HttpModule],
      providers: [
        {
          provide: Http,
          useValue: mockHttp
        },
        SearchWiki
      ]
    });
  });

  it('should get search results', fakeAsync(
    inject([SearchWiki], searchWiki => {
      const expectedUrl = 'https://en.wikipedia.org/w/api.php?' +
        'action=query&list=search&srsearch=Angular';

      searchWiki.search('Angular')
        .subscribe(res => {
          expect(mockHttp.get).toHaveBeenCalledWith(expectedUrl);
          expect(res).toEqual(mockResponse);
        });
    })
  ));
});
```

[View Example](#)

Testing JSONP and XHR Back-Ends

Some services take advantage of the JSONP or XHR module to fetch data instead of the traditional HTTP module. We use the same strategies for testing these services - create a mock back-end, initialize the service and test to see if the request our service made is correct and if the data mocked through the back-end makes its way successfully to the service. Fortunately services that rely on the XHR module are tested *exactly* the same way as services that use the HTTP module. The only difference is in which class is used to mock the back-end. In services that use the HTTP module, the `MockBackend` class is used; in those that use XHR, the `XHRBackend` is used instead. Everything else remains the same.

Unfortunately services that use the JSONP module use a significantly different class for mocking the back-end. The class `MockBrowserJsonp` is used for this scenario.

Executing Tests Asynchronously

Since services operate in an asynchronous manner it may be useful to execute a service's entire unit test asynchronously. This can speed up the overall time it takes to complete a full testing cycle since a particular long unit test will not block other unit tests from executing. We can set up our unit test to return a promise, which will resolve as either a success or failure depending on the activity of the test.

```
describe('verify search', () => {
  it('searches for the correct term',
    fakeAsync(inject([SearchWiki, MockBackend], (searchWiki, mockBackend) => {
      return new Promise((pass, fail) => {
        ...
      });
    })));
});
```

Instead of only using `inject`, we use `fakeAsync` to wrap it and fulfill dependencies and execute the test in an asynchronous process. Using `fakeAsync` requires us to return a Promise, which we use to resolve the completion of our test by calling `pass`, or `fail`, depending on the results of our test.

Testing Redux

Unit testing Redux is a very straightforward process. There are two primary units:

- *Reducers* are pure functions that lend themselves well to testing.
- *Actions* trigger changes in a Redux system. There are two broad categories of actions: synchronous (which are quite simple to test) and asynchronous (which are slightly more involved).

The examples below should provide you with a strong foundation for testing Redux applications.

Testing Simple Actions

Consider the following simple actions, from the Redux chapter of this book:

```
import { Injectable } from '@angular/core';
import { NgRedux } from 'ng2-redux';

export const INCREMENT_COUNTER = 'INCREMENT_COUNTER';
export const DECREMENT_COUNTER = 'DECREMENT_COUNTER';

@Injectable()
export class CounterActions {
  constructor(private redux: NgRedux<any>) {}

  increment() {
    this.redux.dispatch({ type: INCREMENT_COUNTER });
  }

  decrement() {
    this.redux.dispatch({ type: DECREMENT_COUNTER });
  }
}
```

These are pretty straightforward to test:

```
import { NgRedux } from 'ng2-redux';
import {
  CounterActions,
  INCREMENT_COUNTER,
  DECREMENT_COUNTER,
} from './counter';

// Mock out the NgRedux class with just enough to test what we want.
class MockRedux extends NgRedux<any> {
  constructor() {
    super(null);
  }
  dispatch = () => undefined;
}

describe('counter action creators', () => {
  let actions: CounterActions;
  let mockRedux: NgRedux<any>;

  beforeEach(() => {
    // Initialize mock NgRedux and create a new instance of the
    // ActionCreatorService to be tested.
    mockRedux = new MockRedux();
    actions = new CounterActions(mockRedux);
  });

  it('increment should dispatch INCREMENT_COUNTER action', () => {
    const expectedAction = {
      type: INCREMENT_COUNTER
    };

    spyOn(mockRedux, 'dispatch');
    actions.increment();

    expect(mockRedux.dispatch).toHaveBeenCalled();
    expect(mockRedux.dispatch).toHaveBeenCalledWith(expectedAction);
  });

  it('decrement should dispatch DECREMENT_COUNTER action', () => {
    const expectedAction = {
      type: DECREMENT_COUNTER
    };

    spyOn(mockRedux, 'dispatch');
    actions.decrement();

    expect(mockRedux.dispatch).toHaveBeenCalled();
    expect(mockRedux.dispatch).toHaveBeenCalledWith(expectedAction);
  });
});
```

We just make sure that our action creators do indeed dispatch the correct actions.

Testing Complex Actions

Things get a little trickier when we want to test asynchronous or conditional action creators. Our goal is still the same: make sure that our operations are dispatching the actions we're expecting.

A Conditional Action

Consider the following conditional action (i.e., one that is fired depending on current state):

```
import { Injectable } from '@angular/core';
import { NgRedux } from 'ng2-redux';
export const INCREMENT_COUNTER = 'INCREMENT_COUNTER';

@Injectable()
export class MyActionService {
  constructor(private redux: NgRedux) {}

  // A conditional action
  incrementIfOdd() {
    const { counter } = this.redux.getState();

    if (counter % 2 === 0) return;
    this.redux.dispatch({ type: INCREMENT_COUNTER });
  }
}
```

Unit testing is similar to before, but we need to mock our state as well as dispatch:

```
import { NgRedux } from 'ng2-redux';
import { CounterActions } from './counter';

class MockRedux extends NgRedux<any> {
  constructor(private state: any) {
    super(null);
  }
  dispatch = () => undefined;
  getState = () => this.state;
}

describe('counter action creators', () => {
  let actions: CounterActions;
  let mockRedux: NgRedux<any>;
  let mockState: any = {};

  beforeEach(() => {
    // Our mock NgRedux can now accept mock state as a constructor param.
    mockRedux = new MockRedux(mockState);
    actions = new CounterActions(mockRedux);
  });

  it('incrementIfOdd should dispatch INCREMENT_COUNTER action if count is odd',
    () => {
    // Our tests can bake in the initial state they need.
    const expectedAction = {
      type: CounterActions.INCREMENT_COUNTER
    };

    mockState.counter = 3;
    spyOn(mockRedux, 'dispatch');
    actions.incrementIfOdd();

    expect(mockRedux.dispatch).toHaveBeenCalled();
    expect(mockRedux.dispatch).toHaveBeenCalledWith(expectedAction);
  });

  it('incrementIfOdd should not dispatch INCREMENT_COUNTER action if count is even',
    () => {
    mockState.counter = 2;
    spyOn(mockRedux, 'dispatch');
    actions.incrementIfOdd();

    expect(mockRedux.dispatch).not.toHaveBeenCalled();
  });
});
```

An Async Action

What about async actions like the following?

```
import { Injectable } from '@angular/core';
import { NgRedux } from 'ng2-redux';

export const INCREMENT_COUNTER = 'INCREMENT_COUNTER';
export const DECREMENT_COUNTER = 'DECREMENT_COUNTER';

@Injectable()
export class CounterActions {
  constructor(private redux: NgRedux<any>) {}

  // ...

  incrementAsync(timeInMs = 1000) {
    this.delay(timeInMs).then(() => this.redux.dispatch({ type: INCREMENT_COUNTER }));
  }

  private delay(timeInMs) {
    return new Promise((resolve, reject) => {
      setTimeout(() => resolve() , timeInMs);
    });
  }
}
```

We can test this using the normal techniques for async service functions:

- If we can make `incrementAsync` return a promise, we can just return a promise from the test and `jasmine` will wait until it settles.
- Alternately, we can use the `fakeAsync` technique discussed in the section on testing components.

The thing to remember is that if we follow the ActionCreatorService pattern, our actions are just functions on an Angular service. So we can mock out NgRedux (and any other dependencies) and just test it as we would any other Angular service.

Testing Reducers

Luckily, testing reducers is a lot like testing our synchronous action creators, since all reducer operations are synchronous. This plays a big role in making our global state easy to keep track of, which is why we're big fans of Redux.

We'll test the counter reducer in [angular2-redux-starter](#), as follows:

```
export default function counter(state = 0, action)
  switch (action.type) {
    case INCREMENT_COUNTER:
      return state + 1;
    case DECREMENT_COUNTER:
      return state - 1;
    default:
      return state;
  }
}
```

As you can see, there are three cases to test: the default case, the increment and the decrement. We want to test that our actions trigger the state changes we expect from the reducer.

```
import { INCREMENT_COUNTER, DECREMENT_COUNTER } from '../actions/counter';
import counter from './counter';

describe('counter reducers', () => {
  it('should handle initial state', () => {
    expect(
      counter(undefined, {})
    )
    .toEqual(0)
  });

  it('should handle INCREMENT_COUNTER', () => {
    expect(
      counter(0, {
        type: INCREMENT_COUNTER
      })
    )
    .toEqual(1)
  });

  it('should handle DECREMENT_COUNTER', () => {
    expect(
      counter(1, {
        type: DECREMENT_COUNTER
      })
    )
    .toEqual(0)
  });
});
```

Note that we're only testing the section of Redux state that the `counter` reducer is responsible for, and not the whole. We can see from these tests that Redux is largely built on pure functions.

Afterthoughts

The examples outlined above are just one approach to unit testing in Redux. During actual development it might prove to be too costly to maintain tests for every action and reducer, and in some cases even trivial (i.e. should I be paranoid about this JSON object with one property being returned?).

Another approach we've tried is to treat the overall state change in the store triggered by an action (or by a series of actions) as a single unit - in the Redux world reducers don't function without actions and vice versa, so why separate them? This leaves more flexibility when making changes to actions and reducers without losing scope of what Redux is doing for our app.

Migrating Angular 1.x Projects to Angular 2



*Figure: Great Migration by gekkodigitalmedia licensed under Public Domain
(<https://pixabay.com/en/great-migration-africa-animal-1021460/>)*

Migration Prep

Before most Angular 1.x applications can be upgraded to Angular 2 there is preparatory work to do. This is especially true for Angular applications using style that predates Angular 1.3.

Upgrading to Angular 1.3+ Style

The first step of any migration is to upgrade the codebases style to conform to Angular 1.3+ style, ideally an Angular 1.5+ style. This means:

- All controllers should be in `controllerAs` form, and ideally should only exist on directives
- Use directives, specifically "component directives", that use the following properties:
 - `restrict: 'E'`
 - `scope: {}`
 - `bindToController: {}`
 - `controllerAs`
 - `template` OR `templateUrl`
 - `transclude` (optional)
 - `require` (optional)
- Component directives should *not* use the following attributes:
 - `compile`
 - `replace: true`
 - `priority / terminal`
- Ideally have one component, or one *thing* per file
- Ideally have folders organized by feature

Using Webpack

Using a module loader like webpack is essential for migrating to Angular, and should already be part of every modern programmer's tool set. Webpack will make it easy to manage all the different files that a modern, modular Angular 1.3+ project prescribes. This includes bundling the application for distribution or deployment.

Using webpack will also simplify a programmer's Angular workflow, since the easiest way to work with Angular is with TypeScript, or ES6, neither of which works natively in contemporary browsers.

Migrating to TypeScript

TypeScript is a superset of ES6 and, as its name suggests, uses a type system. This can have an enormous impact on developer tools, providing richer auto-complete and static analysis.

Angular was built using TypeScript, and supports decorators which provide meta information to Angular. While it is possible to use Angular without these features, the syntax feels more "natural" with TypeScript's decorators.

Choosing an Upgrade Path

There are three ways to upgrade from Angular 1 to 2:

- Total conversion
- ng-upgrade
- ng-metadata

Total Conversion

Completely converting an application from Angular 1 to Angular 2 is technically possible, but really only suitable for the smallest applications. Even small applications can be tricky to totally convert if they're not well structured.

Using ng-metadata (Angular 1.x Using ≥ 2 Style)

The ng-metadata approach is done with Angular 1.x dependencies and a few small helper libraries. ng-metadata allows developers to use Angular 2 style TypeScript (annotations/decorators) *without* Angular 2. Unfortunately templates are still mostly in Angular 1.x style.

Once an application is converted to ng-metadata style it is very close to Angular 2, but still requires refactoring. In most cases, ng-metadata is not as efficient as ng-upgrade with respect to refactoring time. The payload of an ng-metadata application is smaller, and porting to ng-metadata can be done in an even more ad-hoc fashion than with ng-upgrade.

The general flow of using ng-metadata with an application is:

- Install ng-metadata dependencies
- Bootstrap root component
- Upgrade components strategically
- Refactor the codebase to Angular 2

ng-metadata is favored over the deprecated ng-forward.

Bootstrapping ng-metadata

ng-metadata provides `@NgModule` from Angular 2 to Angular 1. To use `@NgModule`, update your application bootstrap from `angular.bootstrap` to the example below.

Bootstrap (bootstrap.ts)

```
import { platformBrowserDynamic } from 'ng-metadata/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

App Module (app.module.ts)

```
import { NgModule } from 'ng-metadata/core';
import { AppComponent } from './app.component';
import { HeroComponent } from './hero.component';
import { HeroService } from './hero.service';

@NgModule({
  declarations: [AppComponent, HeroComponent],
  providers: [HeroService]
})
export class AppModule {}
```

Components and Services

ng-metadata lets us write code in an Angular style. Components written in this style are prime candidates for an eventual upgrade using ng-upgrade.

Components

Components use `@Component` from ng-metadata. Lifecycle hooks similar to those in Angular should work with ng-metadata.

```
import { Component, Inject, Input, Output, EventEmitter, OnInit } from 'ng-metadata/core';
import { HeroService } from './hero.service';

@Component({
  selector: 'hero',
  templateUrl: './hero.component.html'
})
export class HeroComponent implements OnInit {
  @Input() name: string;
  @Output() call = new EventEmitter<void>();

  // Services can be included using `@Inject` or by their class literal
  constructor(
    @Inject('$log') private $log: ng.ILogService,
    private heroSvc: HeroService
  ){
  }

  ngOnInit() {
    console.log('Component initialized');
  }
}
```

Services

Services use `@Injectable` from ng-metadata. This decorator is written preceding a TypeScript class. Angular 1 services can be added by using the `@Inject` decorator in the service constructor.

```
import { Injectable, Inject } from 'ng-metadata/core';

@Injectable()
export class HeroService {
  constructor(@Inject('$http') private $http: ng.IHttpService){
  }

  fetchAll(){
    return this.$http.get('/api/heroes');
  }
}
```

Using ng-upgrade (Angular 1.x Co-Existing With Angular 2)

The ng-upgrade is done by running Angular 2 and Angular 1 together in the same application. In this scenario Angular 1.x controls the page, and Angular 2 controls the change detection mechanisms. Once the two Angulars co-exist, upgrading can be done in strategic pieces.

Order of Operations

Migrating a large Angular 1 application to Angular 2 can be a big undertaking. We recommend the following order of operations during conversion.

- Webpack
- TypeScript
- Move as much code as possible into pure TypeScript modules
 - Write framework-agnostic unit tests for that code
 - Good candidates for this are stateless services
- Enable ngUpgrade
 - If used, replace the `ng-app` directive with `angular.bootstrap`.
 - Create `UpgradeAdapter` singleton and replace "bootstrap".
- Identify *components* (directives) of the app most likely to benefit from Angular 2
 - These could be parts of the app where performance is a problem, parts where there will be more active development or parts that could really benefit from Angular 2 libraries or components.
- Convert all service dependencies from Angular 1 to Angular 2
 - Move existing `.factory` Angular services to `.service`
 - Leverage TypeScript classes
 - Use `upgradeAdapter.downgradeNg2Provider(serviceName)` to expose Angular 2 service to Angular 1
- Repeat this process until all components have been converted to Angular 2

Replacing Services with TypeScript Classes

Early Angular 1 applications predate the widespread use of module loaders. The strategy of this era was to concatenate source files and rely on Angular 1's dependency injection as a poor-man's module loader. Often services were used to house *libraries* instead of *stateful services*.

During conversion, we will introduce Webpack as a module loader. For services that lack state and don't heavily rely on other dependency injected services, we recommend rewriting them using TypeScript modules. The advantages of writing code this way are:

- It becomes framework agnostic (doesn't rely on Angular 1 explicitly)
- It is easier to test
- It instantly works with both Angular 1 and Angular 2

Even services that depend on a limited set of Angular 1 services (e.g. `$http`) can be rewritten by depending on other libraries (e.g. `window.fetch`).

How do we get there?

- Convert services using `.factory` to `.service`
 - Angular 2's `@Injectable` expects an object it can use `new` with, similar to how `.service` works (e.g. `new CalculatorService()`)
- Replace constructor functions with TypeScript `class`
- Use the class directly by `export` ing it.

Example

`.factory` original

```
angular.module('calcapp', [])
  .factory('CalculatorService', function () {
    return {
      square: function (a) {
        return a*a;
      },
      cube: function (a) {
        return a*a*a;
      }
    };
  });
```

Conversion to `.service`

```
angular.module('calcapp', [])
  .service('CalculatorService', function () {
    this.square = function (a) {
      return a*a;
    };

    this.cube = function (a) {
      return a*a*a;
    }
  });
```

Conversion to TypeScript class

```
class CalculatorService {
  square (a) {
    return a*a;
  }

  cube (a) {
    return a*a*a;
  }
}

angular.module('calcapp', [])
  .service('CalculatorService', CalculatorService);
```

Skip the middleman

```
export class CalculatorService { ... }

// elsewhere
import {CalculatorService} from './calculator.service';
```

Bootstrapping ng-upgrade

- Use manual Angular 1.x bootstrapping, and remove `ng-app` / `ng-strict-di` references if they exist
- Add Angular 2 dependencies
- Add the upgrade adapter `import {UpgradeAdapter} from '@angular/upgrade'`
- Call the upgrade adapter's bootstrap

Once this is working the foundation is set for transitioning from Angular 1.x to Angular 2. It is important to note that the upgrade adapter's bootstrap mechanism is asynchronous. Additionally it's important to treat the upgrade adapter as a singleton.

The following file creates an instance of `UpgradeAdapter` and exports it.

```
// Angular 2 Vendor Import
import {UpgradeAdapter} from '@angular/upgrade';
import {NgModule, forwardRef} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';

// Instantiate an adapter with the AppModule
// Use forwardRef to pass AppModule reference at runtime
export const upgradeAdapter = new UpgradeAdapter(forwardRef(() => AppModule));

@NgModule({
  declarations: [],
  providers: [],
  imports: [BrowserModule]
})
export class AppModule {
}
```

The following file bootstraps an Angular 1/2 hybrid application:

```
// Import the upgradeAdapter singleton
import {upgradeAdapter} from './upgrade-adapter';

// Name the application
const APPNAME = 'angular-upgrade-example';

// Register classic Angular 1 modules
angular.module(APPNAME, []);

// Bootstrap Angular 2 - *note* this is asynchronous
upgradeAdapter.bootstrap(document.body, [APPNAME], {strictDi: true});
```

The above example does not actually do anything other than bootstrap an empty application.

Upgrading/Downgrading Components

Once bootstrapping is complete, Angular 1.x components can be *upgraded* to work with Angular 2. Conversely, Angular 2 components can be *downgraded* to work with Angular 1.x.

Downgrading Components

Upgrading components sounds like it should happen before downgrading, but the point of upgrading is to make an Angular 1.x component work with Angular 2. For an Angular 2 component to use an Angular 1.x component in an ng-upgrade application there must first be a downgraded Angular 2 component. Consequently it's important to first learn how to downgrade Angular 2 components to work with Angular 1.x

All downgraded components operate like Angular 1.x `'E'` element directives.

Here is an example of a very simple Angular 2 component:

```
import {Component} from '@angular/core';

@Component({
  selector: 'a2-downgrade',
  template: '<p>{{ message }}</p>'
})
export class A2DowngradeComponent {
  message = `What you're seeing here is an Angular2 component ` +
    `running in an Angular1 app!`;
}
```

Registering the downgraded component with Angular 1.x:

```
// Angular 1 Vendor Import
import * as angular from 'angular';

// Import the upgradeAdapter singleton
import {upgradeAdapter} from './upgrade-adapter';

// Angular 2 component from above
import {A2DowngradeComponent} from './components/a2-downgrade';

// Register classic Angular 1 modules
angular
  .module(APPNAME)
  .directive('a2Downgrade',
    upgradeAdapter.downgradeNg2Component(A2DowngradeComponent));
```

Upgrading Components

The only Angular 1.x components that can be upgraded and used in Angular 2 code are those that *strictly* follow the component pattern outlined at the top of this document.

Wherever possible use Angular 1.5+'s `.component` .

Here is an Angular 1.x directive that conforms to ng-upgrade's "component directive" specification:

```
angular.module('app').directive('a1Upgradable', function a1UpgradableDirective() {
  return {
    restrict: 'E',
    scope: {},
    bindToController: {},
    controller: Upgradable,
    controllerAs: 'a1Upgradable',
    template: `{{ a1Upgradable.message }}`
  };
});

class Upgradable {
  message = 'I am an Angular 1 Directive';
}
```

Equivalently this can be written using `.component` in Angular 1.5+:

```
angular.module('app').component('a1Upgradable', {
  controller: Upgradable,
  template: `{{ a1Upgradable.message }}`
});

class Upgradable {
  message = 'I am an Angular 1 Directive';
}
```

Below is an Angular 2 component that will use the upgraded Angular 1.x directive:


```
import {upgradeAdapter} from '../upgrade-adapter';
import {A2UsingA1Component} from './a2-using-a1.component';

@NgModule({
  declarations: [upgradeAdapter.upgradeNg1Component('a1Upgradable'), A2UsingA1Component],
  providers: [],
  imports: [BrowserModule]
})
export class AppModule {
}
```

```
import {Component} from '@angular/core';

@Component({
  selector: 'a2-using-a1',
  template: `

{{ message }}


```

Finally, let Angular 1.x know about the directive:

```
import {a1UpgradableDirective} from './components/a1-upgradable';

// Angular 1 Vendor Import
import * as angular from 'angular';

// Register classic Angular 1 modules
angular
  .module(APPNAME)
  .directive('a1Upgradable', a1UpgradableDirective)
```

Projecting Angular 1 Content into Angular 2 Components

In Angular 2 the concept of "transclusion" has been replaced by the concept of projection. ng-upgrade provides mechanisms for projecting/transcluding Angular 1.x content into Angular 2 components:

This is what a simple Angular 2 component that supports projection looks like:

```
import {Component, Input} from '@angular/core';

@Component({
  selector: 'a2-projection',
  template: `
    <p>
      Angular 2 Outer Component (Top)
      <ng-content></ng-content>
      Angular 2 Outer Component (Bottom)
    </p>
  `
})
export class A2Projection { }
```

Here's a very simple Angular 1.x directive that will be projected into the Angular 2 component:

```
export function a1ProjectionContentsDirective() {
  return {
    restrict: 'E',
    scope: {},
    bindToController: {},
    controller: A1ProjectionContents,
    controllerAs: 'a1ProjectionContents',
    template: `<p>{{ a1ProjectionContents.message }}</p>`
  };
}

class A1ProjectionContents {
  message = 'I am an Angular 1 Directive "projected" into Angular 2!';
}
```

Both the component and the directive must be registered with Angular 1.x:

```
import {A2Projection} from './components/a2-projection';
import {a1ProjectionContentsDirective} from
  './components/a1-projection-contents';

// Angular 1 Vendor Import
import * as angular from 'angular';

// Import the upgradeAdapter singleton
import {upgradeAdapter} from './upgrade-adapter';

// Name the application
const APPNAME = 'angular-upgrade-example';

// Register classic Angular 1 modules
angular
  .module(APPNAME)
  .directive('a2Projection',
    upgradeAdapter.downgradNg2Component(A2Projection))
  .directive('a1ProjectionContent', a1ProjectionContentsDirective);
```

Finally, using the HTML selectors is as simple as:

```
<a2-projection>
  <a1-projection-content></a1-projection-content>
</a2-projection>
```

Transcluding Angular 2 Components into Angular 1 Directives

Angular 2 components can be transcluded into Angular 1.x directives.

Here is a very simple Angular 2 component:

```
import {Component} from '@angular/core';

@Component ({
  selector: 'a2-transclusion-contents',
  template: `<p>{{ message }}</p>`
})
export class A2Transclusion {
  message =
    'I am an Angular 2 Component "transcluded" into Angular 1.x';
}
```

Here is an Angular 1.x directive that supports transclusion:

```
export function a1TransclusionDirective() {
  return {
    restrict: 'E',
    transclude: true,
    scope: {},
    bindToController: {},
    controller: A1Transclusion,
    controllerAs: 'a1ProjectionContents',
    template: `
      <p>
        <ng-transclude></ng-transclude>
      </p>
    `;
  };
}

class A1Transclusion {
}
```

Angular 1.x needs to know about both the component and the directive:

```
import {A2Transclusion} from './components/a2-transclusion-contents';
import {a1TransclusionDirective} from './components/a1-transclusion';

// Angular 1 Vendor Import
import * as angular from 'angular';

// Import the upgradeAdapter singleton
import {upgradeAdapter} from './upgrade-adapter';

// Name the application
const APPNAME = 'angular-upgrade-example';

// Register classic Angular 1 modules
angular
  .module(APPNAME)
  .directive('a2TransclusionContents',
    upgradeAdapter.downgradeNg2Component(A2Transclusion))
  .directive('a1Transclusion', a1TransclusionDirective);
```

Finally, Angular 2 content can be transcluded into Angular 1.x like so:

```
<a1-transclude>
  <a2-transclusion-contents></a2-transclusion-contents>
</a1-transclude>
```

Injecting Across Frameworks

Angular 1.x providers/services can be upgraded and injected into Angular 2.

Simple Angular 1.x service:

```
export class A1UpgradeService {  
  data = 'Hello from Angular 1 service';  
}
```

Simple Angular 2 component that will have an Angular 1.x service injected into it:

```
import {Component, Inject} from '@angular/core';  
import {A1UpgradeService} from '../services/a1-upgrade-service';  
  
@Component({  
  selector: 'a2-using-a1-service',  
  template: `

{{ message }}

`  
})  
export class A2UsingA1Service {  
  message = '';  
  constructor(@Inject('a1UpgradeService') a1UpgradeService:A1UpgradeService) {  
    this.message = a1UpgradeService.data;  
  }  
}
```

Attaching everything to Angular 1.x:

```
import {A2UsingA1Service} from './components/a2-using-a1-service';
import {A1UpgradeService} from './services/a1-upgrade-service';

// Angular 1 Vendor Import
import * as angular from 'angular';

// Import the upgradeAdapter singleton
import {upgradeAdapter} from './upgrade-adapter';

// Name the application
const APPNAME = 'angular-upgrade-example';

// Register classic Angular 1 modules
angular
  .module(APPNAME)
  .directive('a2UsingA1Service',
    upgradeAdapter.downgradeNg2Component(A2UsingA1Service))
  .service('a1UpgradeService', A1UpgradeService);
```

Angular 2.x services can be downgraded and injected into Angular 1. In normal operation, Angular 2.x services would be bootstrapped with the application, but because of ng-upgrade being a hybrid mode, this is not the case. The upgrade adapter comes with an `addProvider` method that must be used in the interim.

Here is a very simple Angular 2 service:

```
import {Injectable} from '@angular/core';

@Injectable()
export class A2DowngradeService {
  fetchData() {
    return 'some data';
  }
}
```

Since Angular 2 is bootstrapped with the upgrade adapter, there is no place to register Angular 2 services. Fortunately the upgrade adapter's `addProvider` method can do this:

```
upgradeAdapter.addProvider(Phones);
```

Lastly, Angular 1.x must be informed about the Angular 2 service:

```
// The service to downgrade
import {A2DowngradeService} from './services/a2-downgrade'

// Angular 1 Vendor Import
import * as angular from 'angular';

// Import the upgradeAdapter singleton
import {upgradeAdapter} from './upgrade-adapter';

// Name the application
const APPNAME = 'angular-upgrade-example';

// Register classic Angular 1 modules
angular
  .module(APPNAME)
  .factory('a2DowngradeService',
    upgradeAdapter.downgradeNg2Provider(A2DowngradeService));
```

Using this downgraded service in an Angular 1.x directive is as simple as:

```
import {A2DowngradeService} from './services/a2-downgrade';

export function a1UsingA2ServiceDirective() {
  return {
    restrict: 'E',
    scope: {},
    bindToController: {},
    controller: A1UsingA2,
    controllerAs: 'a1UsingA2',
    template: `<span>{{ a1UsingA2.message }}</span>`
  };
}

class A1UsingA2 {
  message: string;
  constructor(private a2DowngradeService: A2DowngradeService) {
    this.message = this.a2DowngradeService.fetchData();
  }
}
```


Project Setup

Proper tooling and setup is good for any project, but it's especially important for Angular due to all of the pieces that are involved. We've decided to use [webpack](#), a powerful tool that attempts to handle our complex integrations. Due to the number of parts of our project that webpack touches, it's important to go over the configuration to get a good understanding of what gets generated client-side.

Webpack

A modern JavaScript web application includes a lot of different packages and dependencies, and it's important to have something that makes sense of it all in a simple way.

Angular takes the approach of breaking your application apart into many different components, each of which can have several files. Separating application logic this way is good for the programmer, but can detract from user experience since doing this can increase page loading time. HTTP2 aims to solve this problem in one way, but until more is known about its effects we will want to bundle different parts of our application together and compress it.

Our platform, the browser, must continue to provide backwards compatibility for all existing code and this necessitates slow movement of additions to the base functionality of HTML/CSS/JS. The community has created different tools that transform their preferred syntax/feature set to what the browser supports to avoid binding themselves to the constraints of the web platform. This is especially evident in Angular applications, where [TypeScript](#) is used heavily. Although we don't do this in our course, projects may also involve different CSS preprocessors (sass, stylus) or templating engines (jade, Mustache, EJS) that must be integrated.

Webpack solves these problems by providing a common interface to integrate all of these tools and that allows us to streamline our workflow and avoid complexity.

Installation

The easiest way to include webpack and its plugins is through NPM and save it to your

`devDependencies` :

```
npm install -D webpack ts-loader html-webpack-plugin tslint-loader
```

Setup and Usage

The most common way to use webpack is through the CLI. By default, running the command executes `webpack.config.js` which is the configuration file for your webpack setup.

Bundle

The core concept of webpack is the *bundle*. A bundle is simply a collection of modules, where we define the boundaries for how they are separated. In this project, we have two bundles:

- `app` for our application-specific client-side logic
- `vendor` for third party libraries

In webpack, bundles are configured through *entry points*. Webpack goes through each entry point one by one. It maps out a dependency graph by going through each module's references. All the dependencies that it encounters are then packaged into that bundle.

Packages installed through NPM are referenced using *CommonJS* module resolution. In a JavaScript file, this would look like:

```
const app = require('./src/index.ts');
```

or TypeScript/ES6 file:

```
import { Component } from '@angular/core';
```

We will use those string values as the module names we pass to webpack.

Let's look at the entry points we have defined in our sample app:

```
{
  ...
  entry: {
    app: './src/index.ts',
    vendor: [
      '@angular/core',
      '@angular/compiler',
      '@angular/common',
      '@angular/http',
      '@angular/platform-browser',
      '@angular/platform-browser-dynamic',
      '@angular/router',
      'es6-shim',
      'redux',
      'redux-thunk',
      'redux-logger',
      'reflect-metadata',
      'ng2-redux',
      'zone.js',
    ]
  }
  ...
}
```

The entry point for `app`, `./src/index.ts`, is the base file of our Angular application. If we've defined the dependencies of each module correctly, those references should connect all the parts of our application from here. The entry point for `vendor` is a list of modules that we need for our application code to work correctly. Even if these files are referenced by some module in our app bundle, we want to separate these resources in a bundle just for third party code.

Output Configuration

In most cases we don't just want to configure how webpack generates bundles - we also want to configure how those bundles are output.

- Often, we will want to re-route where files are saved. For example into a `bin` or `dist` folder. This is because we want to optimize our builds for production.
- Webpack transforms the code when bundling our modules and outputting them. We want to have a way of connecting the code that's been generated by webpack and the code that we've written.
- Server routes can be configured in many different ways. We probably want some way of configuring webpack to take our server routing setup into consideration.

All of these configuration options are handled by the config's `output` property. Let's look at how we've set up our config to address these issues:

```
{
  ...
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].[hash].js',
    publicPath: "/",
    sourceMapFilename: '[name].[hash].js.map'
  }
  ...
}
```

Some options have words wrapped in square brackets. Webpack has the ability to parse parameters for these properties, with each property having a different set of parameters available for substitution. Here, we're using `name` (the name of the bundle) and `hash` (a hash value of the bundle's content).

To save bundled files in a different folder, we use the `path` property. Here, `path` tells webpack that all of the output files must be saved to `path.resolve(__dirname, 'dist')`. In our case, we save each bundle into a separate file. The name of this file is specified by the `filename` property.

Linking these bundled files and the files we've actually coded is done using what's known as source maps. There are different ways to configure source maps. What we want is to save these source maps in a separate file specified by the `sourceMapFilename` property. The way the server accesses the files might not directly follow the filesystem tree. For us, we want to use the files saved under `dist` as the root folder for our server. To let webpack know this, we've set the `publicPath` property to `/`.

Loaders

TypeScript isn't core JavaScript so webpack needs a bit of extra help to parse the `.ts` files. It does this through the use of *loaders*. Loaders are a way of configuring how webpack transforms the outputs of specific files in our bundles. Our `ts-loader` package is handling this transformation for TypeScript files.

Inline

Loaders can be configured – inline – when requiring/importing a module:

```
const app = require('ts!./src/index.ts');
```

The loader is specified by using the `!` character to separate the module reference and the loader that it will be run through. More than one loader can be used and those are separated with `!` in the same way. Loaders are executed right to left.

```
const app = require('ts!tslint!./src/index.ts');
```

Although the packages are named `ts-loader`, `tslint-loader`, `style-loader`, we don't need to include the `-loader` part in our config.

Be careful when configuring loaders this way – it couples implementation details of different stages of your application together so it might not be the right choice in a lot of cases.

Webpack Config

The preferred method is to configure loaders through the `webpack.config.js` file. For example, the TypeScript loader task will look something like this:

```
{
  test: /\.ts$/,
  loader: 'ts-loader',
  exclude: /node_modules/
}
```

This runs the typescript compiler which respects our configuration settings as specified above. We want to be able to handle other files and not just TypeScript files, so we need to specify a list of loaders. This is done by creating an array of tasks.

Tasks specified in this array are chained. If a file matches multiple conditions, it will be processed using each task in order.

```
{
  ...
  module: {
    rules: [
      { test: /\.ts$/, loader: 'tslint' },
      { test: /\.ts$/, loader: 'ts', exclude: /node_modules/ },
      { test: /\.html$/, loader: 'raw' },
      { test: /\.css$/, loader: 'style!css?sourceMap' },
      { test: /\.svg/, loader: 'url' },
      { test: /\.eot/, loader: 'url' },
      { test: /\.woff/, loader: 'url' },
      { test: /\.woff2/, loader: 'url' },
      { test: /\.ttf/, loader: 'url' },
    ],
    noParse: [ /zone\.js\/dist\/.+/ , /angular2\/bundles\/.+/ ]
  }
  ...
}
```

Each task has a few configuration options:

- *test* - The file path must match this condition to be handled. This is commonly used to test file extensions eg. `/\.ts$/` .
- *loader* - The loaders that will be used to transform the input. This follows the syntax specified above.
- *exclude* - The file path must not match this condition to be handled. This is commonly used to exclude file folders, e.g. `/node_modules/` .
- *include* - The file path must match this condition to be handled. This is commonly used to include file folders. eg. `path.resolve(__dirname, 'app/src')` .

Pre-Loaders

The preLoaders array works just like the loaders array only it is a separate task chain that is executed before the loaders task chain.

Non JavaScript Assets

Webpack also allows us to load non JavaScript assets such as: CSS, SVG, font files, etc. In order to attach these assets to our bundle we must require/import them within our app modules. For example:

```
import './styles/style.css';  
  
// or  
  
const STYLES = require('./styles/style.css');
```

Other Commonly Used Loaders

- *raw-loader* - returns the file content as a string.
- *url-loader* - returns a base64 encoded data URL if the file size is under a certain threshold, otherwise it just returns the file.
- *css-loader* - resolves `@import` and `url` references in CSS files as modules.
- *style-loader* - injects a style tag with the bundled CSS in the `<head>` tag.

Plugins

Plugins allow us to inject custom build steps during the bundling process.

A commonly used plugin is the `html-webpack-plugin`. This allows us to generate HTML files required for production. For example it can be used to inject script tags for the output bundles.

```
new HtmlWebpackPlugin({
  template: './src/index.html',
  inject: 'body',
  minify: false
});
```

Summary

When we put everything together, our complete `webpack.config.js` file looks something like this:

```
'use strict';

const path = require("path");
const webpack = require('webpack');
const HtmlWebpackPlugin = require('html-webpack-plugin');

const basePlugins = [
  new webpack.optimize.CommonsChunkPlugin('vendor', '[name].[hash].bundle.js'),

  new HtmlWebpackPlugin({
    template: './src/index.html',
    inject: 'body',
    minify: false
  })
];

const envPlugins = {
  production: [
    new webpack.optimize.UglifyJsPlugin({
      compress: {
        warnings: false
      }
    })
  ],
  development: []
};

const plugins = basePlugins.concat(envPlugins[process.env.NODE_ENV] || []);

module.exports = {
  entry: {
    app: './src/index.ts',
    vendor: [
      '@angular/core',
      '@angular/compiler',
      '@angular/common',
      '@angular/http',
      '@angular/platform-browser',
      '@angular/platform-browser-dynamic',
      '@angular/router',
      'es6-shim',
      'redux',
      'redux-thunk',
      'redux-logger',
    ]
  }
};
```

```
'reflect-metadata',
'ng2-redux',
'zone.js',
]
},

output: {
  path: path.resolve(__dirname, 'dist'),
  filename: '[name].[hash].js',
  publicPath: "/",
  sourceMapFilename: '[name].[hash].js.map'
},

devtool: 'source-map',

resolve: {
  extensions: ['.webpack.js', '.web.js', '.ts', '.js']
},

plugins: plugins,

module: {
  rules: [
    { test: /\.ts$/, loader: 'tslint' },
    { test: /\.ts$/, loader: 'ts', exclude: /node_modules/ },
    { test: /\.html$/, loader: 'raw' },
    { test: /\.css$/, loader: 'style!css?sourceMap' },
    { test: /\.svg/, loader: 'url' },
    { test: /\.eot/, loader: 'url' },
    { test: /\.woff/, loader: 'url' },
    { test: /\.woff2/, loader: 'url' },
    { test: /\.ttf/, loader: 'url' },
  ],
  noParse: [ /zone\.js\/dist\/.+/, /angular2\/bundles\/.+/ ]
}
}
```

Going Further

Webpack also does things like hot code reloading and code optimization which we haven't covered. For more information you can check out the [official documentation](#). The source is also available on [Github](#).

NPM Scripts Integration

NPM allows us to define custom scripts in the `package.json` file. These can then execute tasks using the NPM CLI.

We rely on these scripts to manage most of our project tasks and webpack fits in as well.

The scripts are defined in the `scripts` property of the `package.json` file. For example:

```
...
  scripts: {
    "clean": "rimraf dist",
    "prebuild": "npm run clean",
    "build": "NODE_ENV=production webpack",
  }
...
```

NPM allows pre and post task binding by prepending the word `pre` or `post` respectively to the task name. Here, our `prebuild` task is executed before our `build` task.

We can run an NPM script from inside another NPM script.

To invoke the `build` script we run the command `npm run build`:

1. The `prebuild` task executes.
2. The `prebuild` task runs the `clean` task, which executes the `rimraf dist` command.
3. `rimraf` (an NPM package) recursively deletes everything inside a specified folder.
4. The `build` task is executed. This sets the `NODE_ENV` environment variable to `production` and starts the webpack bundling process.
5. Webpack generates bundles based on the `webpack.config.js` available in the project root folder.

Angular CLI

With all of the new features Angular takes advantage of, like static typing, decorators and ES6 module resolution, comes the added cost of setup and maintenance. Spending a lot of time with different build setups and configuring all of the different tools used to serve a modern JavaScript application can really take a lot of time and drain productivity by not being able to actually work on the app itself.

Seeing the popularity of [ember-cli](#), Angular decided they would provide their own CLI to solve this problem. [Angular CLI](#) is geared to be the tool used to create and manage your Angular app. It provides the ability to:

- create a project from scratch
- scaffold components, directives, services, etc.
- lint your code
- serve the application
- run your unit tests and end to end tests.

The Angular CLI currently only generates scaffolding in TypeScript, with other dialects to come later.

Setup

Prerequisites

Angular CLI is currently only distributed through npm and requires Node version 4 or greater.

Installation

The Angular CLI can be installed with the following command:

```
npm install -g angular-cli
```

Creating a New App

Use the `ng new [app-name]` command to create a new app. This will generate a basic app in the folder of the app name provided. The app has all of the features available to work with the CLI commands. Creating an app may take a few minutes to complete since npm will need to install all of the dependencies. The directory is automatically set up as a new **git** repository as well. If git is not your version control of choice, simply remove the `.git` folder and `.gitignore` file.

File and Folder Setup

The generated app folder will look like this:

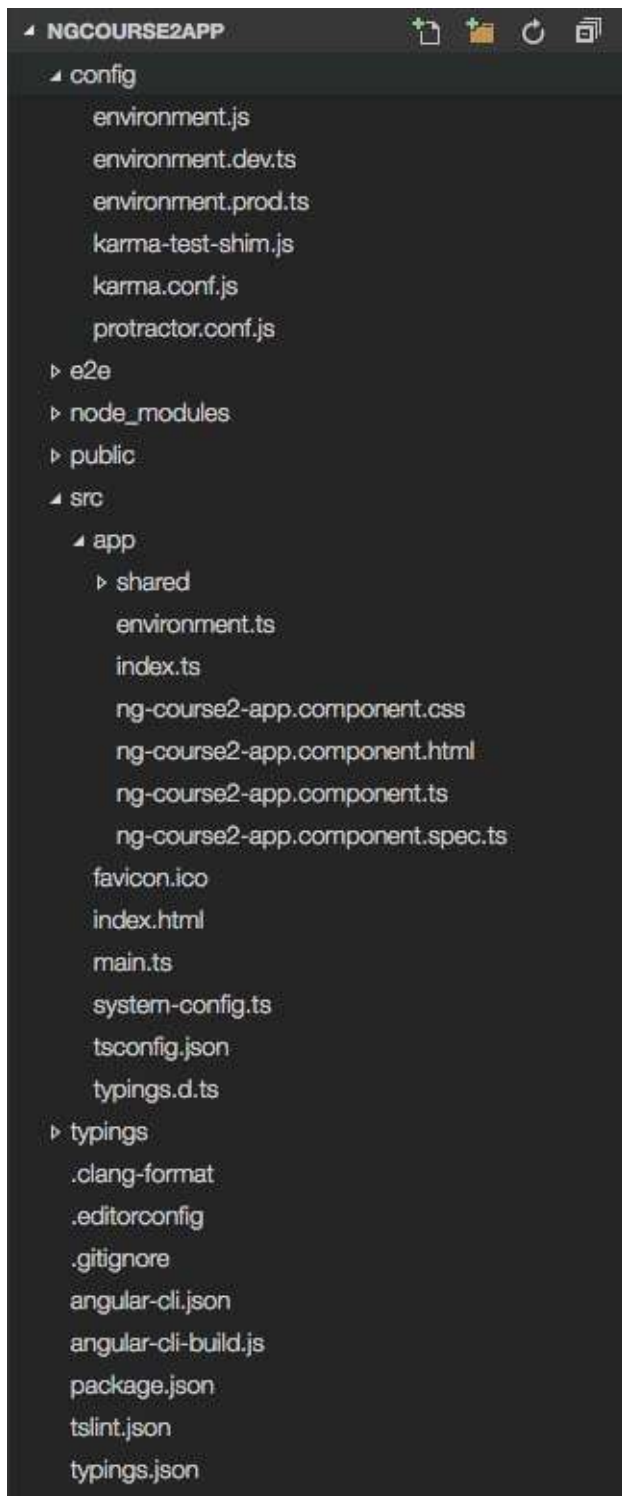


Figure: App folder

Application configuration is stored in different places, some located in the *config* folder, such as test configuration, and some being stored in the project root such as linting information and build information. The CLI stores application-specific files in the *src* folder and Angular-specific code in the *src/app* folder. Files and folders generated by the CLI will follow the [official style guide](#).

Warning: The CLI relies on some of the settings defined in the configuration files to be able to execute the commands. Take care when modifying them, particularly the `package.json` file.

The CLI has installed everything a basic Angular application needs to run properly. To make sure everything has run and installed correctly we can run the server.

Serving the App

The CLI provides the ability to serve the app with live reload. To serve an application, simply run the command `ng serve`. This will compile the app and copy all of the application-specific files to the *dist* folder before serving.

By default, `ng serve` serves the application locally on port 4200 (<http://localhost:4200>) but this can be changed by using a command line argument: `ng serve --port=8080`.

Creating Components

The CLI can scaffold Angular components through the `generate` command. To create a new component run:

```
ng generate component [component-name]
```

Executing the command creates a folder, `[component-name]`, in the project's `src/app` path or the current path the command is executed in if it's a child folder of the project. The folder has the following:

- `[component-name].component.ts` the component class file
- `[component-name].component.css` for styling the component
- `[component-name].component.html` component html
- `[component-name].component.spec.ts` tests for the component
- `index.ts` which exports the component

Creating Routes

The `ng g route [route-name]` command will spin up a new folder and route files for you.

At the time of writing this feature was temporarily disabled due to ongoing changes happening with Angular routing.

Creating Other Things

The CLI can scaffold other Angular entities such as services, pipes and directives using the generate command.

```
ng generate [entity] [entity-name]
```

This creates the entity at `src/app/[entity-name].[entity].ts` along with a spec file, or at the current path if the command is executed in a child folder of the project. The CLI provides blueprints for the following entities out of the box:

Item	Command	Files generated
Component:	<code>ng g component [name]</code>	component, HTML, CSS, test spec files
Directive:	<code>ng g directive [name]</code>	component, test spec files
Pipe:	<code>ng g pipe [name]</code>	component, test spec files
Service:	<code>ng g service [name]</code>	component, test spec files
Class:	<code>ng g class [name]</code>	component, test spec files
Route:	<code>ng g route [name]</code>	component, HTML, CSS, test spec files (in new folder)

Testing

Apps generated by the CLI integrate automated tests. The CLI does this by using the [Karma test runner](#).

Unit Tests

To execute unit tests, run `ng test`. This will run all the tests that are matched by the Karma configuration file at `config/karma.conf.js`. It's set to match all TypeScript files that end in `.spec.ts` by default.

End-to-End Tests

End-to-end tests can be executed by running `ng e2e`. Before end-to-end tests can be performed, the application must be served at some address. Angular CLI uses protractor. It will attempt to access `localhost:4200` by default; if another port is being used, you will have to update the configuration settings located at `config/protractor.conf.js`.

Linting

To encourage coding best practices Angular CLI provides built-in linting. By default the app will look at the project's `tslint.json` for configuration. Linting can be executed by running the command `ng lint`.

For a reference of tslint rules have a look at: <https://palantir.github.io/tslint/rules/>.

CLI Command Overview

One of the advantages of using the Angular CLI is that it automatically configures a number of useful tools that you can use right away. To get more details on the options for each task,

USE `ng --help` .

Linting

`ng lint` lints the code in your project using [tslint](#). You can customize the rules for your project by editing `tslint.json` .

You can switch some of these to use your preferred tool by editing the scripts in `package.json` .

Testing

`ng test` triggers a build and then runs the unit tests set up for your app using [Karma](#). Use the `--watch` option to rebuild and retest the app automatically whenever source files change.

Build

`ng build` will build your app (and minify your code) and place it into the default output path, `dist/` .

Serve

`ng serve` builds and serves your app on a local server and will automatically rebuild on file changes. By default, your app will be served on <http://localhost:4200/>.

Include `--port [number]` to serve your app on a different HTTP port.

E2E

Once your app is served, you can run end-to-end tests using `ng e2e` . The CLI uses [Protractor](#) for these tests.

Deploy

`ng deploy` deploys to GitHub pages or Firebase.

Adding Third Party Libraries

The CLI generates development automation code which has the ability to integrate third party libraries into the application. Packages are installed using `npm` and the development environment is setup to check the installed libraries mentioned in `package.json` and bundle these third party libraries within the application. For more information see <https://github.com/angular/angular-cli#3rd-party-library-installation>

Integrating an Existing App

Apps that were created without CLI can be integrated to use CLI later on. This is done by going to the existing app's folder and running `ng init`.

Since the folder structure for an existing app might not follow the same format as one created by the CLI, the `init` command has some configuration options.

- `--source-dir` identifies the relative path to the source files (default = `src`)
- `--prefix` identifies the path within the source dir that Angular application files reside (default = `app`)
- `--style` identifies the path where additional style files are located (default = `css`).

Web Accessibility in Angular

Accessibility is something that web developers need to always be aware of, and Angular applications are no exception. While the approach and key concepts are largely the same as with other frameworks, it's important to examine what differences Angular adds to the equation.

Why Accessibility

While making websites accessible can lead to more time spent in development, there are many reasons why you should be making your application accessible.

Reaching out to everyone

Multiple studies suggest that around 15-20% of the population are living with a disability of some kind¹. In comparison, that number is higher than any single browser demographic currently, other than Chrome². Not considering those users when developing an application means excluding a large number of people from being able use it comfortable or at all.

Overlap with User Experience and SEO

Sites that employ techniques to make their site accessible will guarantee that it matches a minimum standard of usability for everyone. Features originally intended for accessibility are often appreciated by all users, voice control modes for phones being one such example.

In addition to this, accessibility techniques such as semantic markup help search engines understand the application better, leading to improved visibility.

Accessibility Laws

Countries around the world have different accessibility rules centered around compliance of two main sets of guidelines: [Web Content Accessibilities \(WCAG\) 2.0](#) and [Section 508](#). Not being in compliance of these rules exposes you to liability.

-
1. WHO. "[Report on Disability](#)". 2011
Eurostat. "[Prevalence of basic activity difficulties or disability](#)". 2012
US Census Bureau. "[Anniversary of Americans with Disabilities Act: July 26](#)". 2012
 2. [Statcounter](#). September 2016

Key Concerns of Accessible Web Applications

It's best to focus on the following three areas when making a web application accessible:

- [Semantic Markup](#) - Allows the application to be understood on a more general level rather than just details of whats being rendered
- [Keyboard Accessibility](#) - Applications must still be usable when using only a keyboard
- [Visual Assistance](#) - color contrast, focus of elements and text representations of audio and events

Semantic Markup

Using Proper HTML Elements and Attributes

A common trap when structuring html is using too many divs and marking them up with classes or ids to indicate their role, for example:

```
@Component({
  selector: 'ngc2-app'
  template: `
    <div class="header">
      <div class="navigation">
        <div class="item"><a [routerLink]="['']"></a></div>
        <div class="item"><a [routerLink]="['services']">Services</a></div>
        <div class="item"><a [routerLink]="['process']">Process</a></div>
        <div class="item"><a [routerLink]="['work']">Work</a></div>
      </div>
    </div>
    <router-outlet class="page-content">
    </router-outlet>
  `
})
```

[View Example](#)

While it might be obvious to someone reading the HTML or using the application what the purpose of each element is, the class names don't have any semantic meaning for browsers and screen readers. We can give them more information by using the proper HTML elements instead.

```
@Component({
  selector: 'ngc2-app',
  template: `
    <header>
      <nav>
        <ul>
          <li><a [routerLink]="['']"></a></li>
          <li><a [routerLink]="['services']">Services</a></li>
          <li><a [routerLink]="['process']">Process</a></li>
          <li><a [routerLink]="['work']">Work</a></li>
        </ul>
      </nav>
    </header>
    <main aria-live="polite">
      <router-outlet>
    </router-outlet>
    </main>
  `
})
```

View Example

Here, we use the `header` element instead of a `div`, which lets the browser know that the elements within provide information about the site as a whole rather than about the specific page.

We replace another `div` with the `nav` element, which lets the browser know the elements within are related to accessing different parts of the page or site.

We also nest `router-outlet` within a `main` element, which tells the browser that the content loaded into the router outlet is the main content of the page.

There are a couple of new attributes on different elements as well to give the browser even more information. The `alt` attribute has been added to the image to let the browser know that it's a logo image. There's also an `aria-live` attribute on the `main` element.

This attribute is part of larger spec known as [Accessible Rich Internet Applications \(WAI-ARIA\)](#) which we'll go over in detail. This is something that lets screen readers know that the content within the `main` tag will be updated on the client-side after the page has loaded and needs to be watched for updates.

Roles and ARIA

The ARIA spec was created as a way for content authors a way to provide additional context to the semantics of their application rather than just details on how to render the content. This allows assistive technology to understand what's going on inside an application and relay that information in a structured and streamlined format for users with disabilities.

One of the main concepts in the ARIA spec is the *role*. A [role](#) defines what the purpose of an html element is within the context of that document or application. Roles are defined by adding an attribute to an html element ie. `role="main"` or are defined by default depending on the html element.

Some examples of roles are [list](#), [button](#) or [navigation](#) which are the default roles of `ul`, `button` and `nav` respectively. Sometimes however, you may not want or be able to use the standard html element to represent these objects in your application, for example, you may want to create your own button component with it's own distinct logic. In this case you can make use of the `role` attribute:

```
@Component({
  selector: 'ngc2-notification-button',
  template: `
    <span>{{label}}</span>
  `,
  styles: [
    `
      :host {
        display: flex;
        width: 80px;
        height: 80px;
        justify-content: center;
        align-items: center;
        background-color: yellow;
        border-radius: 40px;
      }

      :host:hover {
        cursor: pointer;
      }
    `
  ])
export class NotificationButtonComponent {
  @Input()
  message = 'Alert!';

  @Input()
  label = 'Notify';

  constructor(private notification: NotificationService) { }

  @HostListener('click', [])
  notify() {
    this.notification.notify(this.message)
  }
}
```

[View Example](#)

This lets you create a component that has the same semantics as a `button` element to screen readers and browsers, but now have the opportunity to fully control the styling of that component as well as inject your own custom logic.

ARIA attributes

Some native HTML tags have attributes that provides extra context on what's being displayed on the browser. For example, the `img` tag's `alt` attribute lets the reader know what is being shown using a short description.

However, native tags don't cover all cases. This is where ARIA fits in. ARIA attributes can provide context on what roles specific elements have in the application or on how elements within the document relate to each other.

One example of this is modals. Native modals provided by different platforms such as web browsers often have limited customization options, which can make for a poor experience. This necessitates the creation of custom modals.

A modal component can be given the `role` of `dialog` or `alertdialog` to let the browser know that that component is acting as a modal. The modal component template can use the ARIA attributes `aria-labelledby` and `aria-describedby` to describe to readers what the title and purpose of the modal is.

app.component.ts

```
@Component({
  selector: 'ngc2-app',
  template: `
    <ngc2-notification-button
      message="Hello!"
      label="Greeting"
      role="button">
    </ngc2-notification-button>
    <ngc2-modal
      [title]="modal.title"
      [description]="modal.description"
      [visible]="modal.visible"
      (close)="modal.close()">
    </ngc2-modal>
  `
})
export class AppComponent {
  constructor(private modal: ModalService) { }
}
```

notification-button.component.ts

```
@Component({
  selector: 'ngc2-modal',
  template: `
    <div
      role="dialog"
      aria-labelledby="modal-title"
      aria-describedby="modal-description">
      <div id="modal-title">{{title}}</div>
      <p id="modal-description">{{description}}</p>
      <button (click)="close.emit()">OK</button>
    </div>
  `
})
export class ModalComponent {
  ...
}
```

[View Example](#)

ARIA tags can enhance the accessibility of an application, but should by no means be the only accessibility consideration. More information is available in the [WAI-ARIA specification](#).

Keyboard Accessibility

Keyboard accessibility is the ability of your application to be interacted with using just a keyboard. The more streamlined the site can be used this way, the more keyboard accessible it is. Keyboard accessibility is one of the largest aspects of web accessibility since it targets:

- those with motor disabilities who can't use a mouse
- users who rely on screen readers and other assistive technology, which require keyboard navigation
- those who prefer not to use a mouse

Focus

Keyboard interaction is driven by something called *focus*. In web applications, only one element on a document has focus at a time, and keypresses will activate whatever function is bound to that element. The currently focused element can be accessed programmatically through the `document.activeElement` DOM method.

Visually, an element with focus is represented by default with a glowing border around the element:

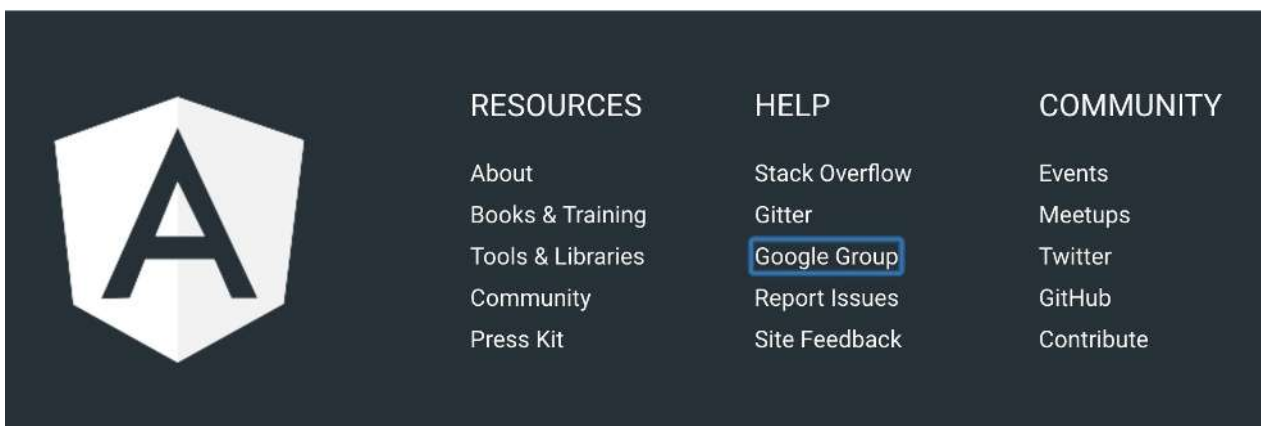


Figure: Focus border

This border can be styled with CSS using the `outline` property, but it should **not be removed**. Elements can also be styled using the `:focus` pseudo-selector.

Tabbing

The most common way of moving focus along the page is through the `tab` key. Elements will be traversed in the order they appear in the document outline - so that order must be carefully considered during development. By default, only links, buttons and form controls can receive keyboard focus.

Whenever possible, developers should bind behaviour to elements that can natively receive focus, such as using a `button` rather than a `div`. They should also adjust the source order of elements to change the tab traversal order.

There may, however, be cases where you'll want to change the default behaviour or tab order. This can be done through the `tabindex` attribute. The `tabindex` can be given the values:

- *less than zero* - to let readers know that an element should be focusable but not keyboard accessible
- *0* - to let readers know that that element should be accessible by keyboard
- *greater than zero* - to let readers know the order in which the focusable element should be reached using the keyboard. Order is calculated from lowest to highest.

Altering `tabindex` [should be done carefully](#), and must also be paired with keypress support for `space` and `enter`.

Transitions

The majority of transitions that happen in an Angular application will not involve a page reload. This means that developers will need to carefully manage what happens to focus in these cases.

It's important that if some action involves a transition away from the natural page flow, then focus should be handled as well.

Modals are one example of this:

```
@Component({
  selector: 'ngc2-modal',
  template: `
    <div
      role="dialog"
      aria-labelledby="modal-title"
      aria-describedby="modal-description">
      <div id="modal-title">{{title}}</div>
      <p id="modal-description">{{description}}</p>
      <button (click)="close.emit()">OK</button>
    </div>
  `,
})
export class ModalComponent {
  constructor(private modal: ModalService, private element: ElementRef) { }

  ngOnInit() {
    this.modal.visible$.subscribe(visible => {
      if(visible) {
        setTimeout(() => {
          this.element.nativeElement.querySelector('button').focus();
        }, 0);
      }
    })
  }
}
```

[View Example](#)

In this example, we see that when the modal becomes visible, the `OK` button immediately receives focus. This streamlines the experience for keyboard users or screen readers to match the experience given to mouse users or those without screen readers.

Visual Assistance

One large category of disability is visual impairment. This includes not just the blind, but those who are [color blind](#) or partially sighted, and require some additional consideration.

Color Contrast

When choosing colors for text or elements on a website, the contrast between them needs to be considered. For WCAG 2.0 AA, this means that the contrast ratio for text or visual representations of text needs to be at least [4.5:1](#).

There are tools online to measure the contrast ratio such as this [color contrast checker](#) from WebAIM or be checked with using automation tests.

Visual Information

Color can help a user's understanding of information, but it should never be the only way to convey information to a user. For example, a user with red/green color-blindness may have trouble discerning at a glance if an alert is informing them of success or failure.

Always be sure that color is not the only way information is being conveyed to the user.

Audiovisual Media

Audiovisual elements in the application such as video, sound effects or audio (ie. podcasts) need related textual representations such as transcripts, captions or descriptions. They also should never auto-play and playback controls should be provided to the user.

Testing for Accessibility

Automated accessibility checkers can help you find problems with your application, but they can also produce noisy, overwhelming results that may not surface the most important issues affecting the usability of your app. It's therefore important to know how to test for accessibility yourself.

There are many different concerns to consider when testing your application for accessibility. The following questions can guide your accessibility testing and provide concrete items for evaluation:

- [Is my Application Readable?](#)
- [Is my Application Predictable?](#)
- [Is my Application Navigable?](#)

One excellent method for covering the bases of an accessible application is to [test with screen readers](#).

Is my Application Readable?

Users should be able to read and understand the information presented by your application. Your application should also be flexible enough to meet your user's reading needs. Check in particular:

- Can the page be resized to 200% in the browser and still be usable?
- Are text sizes appropriate across variable device widths?
- Can the page still be read if the user is overriding your fonts with their own? (Try testing this using the Chrome addon [OpenDyslexic](#))
- Does the text need to be adjusted for colour or weight to provide adequate contrast? (Try your most used fonts in a [contrast checker](#))

Is my Application Predictable?

Users should be able to reasonably anticipate how an application behaves, and they should remain in control of their experience at all times. Check for the following:

- Has auto-play been turned off on all sliders, video and audio? Are playback controls available?
- Will the user be warned if a context switch occurs (e.g. a new window opening)?
- Can interruptions (such as alerts or page updates) be postponed or suppressed by the user?
- Does the user have enough time to read alerts? Does the user's screen reader have enough time?

Is my Application Navigable?

One of the key accessibility features for any application is keyboard navigability. Because many forms of [assistive technology](#) rely on keyboard controls exclusively, testing for keyboard support is a good thing to prioritize. Check in particular:

- Can basic tasks and workflows be completed using only a keyboard?
- Can all forms be completed using only a keyboard?
- When an element has tab focus, is there a visual indicator?
- If a modal or overlay opens, is keyboard focus trapped inside the new context until it is closed?

Testing with Screen Readers

The only way to be certain that your application is usable by screen readers and other assistive technologies is to check for yourself.

To test with a screen reader, you will need to use a screen reading program and navigate your page using keyboard commands specific to that program. *When testing with a screen reader, do not rely on Chrome.* Most [screen reader users do not use Chrome](#) as their main browser, and browser-specific differences will occur in testing. Internet Explorer or Firefox are preferred.

Testing on Mac using VoiceOver

VoiceOver is OSX's built-in screen reader, and therefore often the easiest option for developers.

VoiceOver can be turned on and off with the keyboard shortcut `CMD + F5` and also through the `System Preferences` menu. In the System Preferences menu, you can also adjust the speed at which VoiceOver will read to you. While testing, it is helpful to set it to a higher speed.

[WebAIM's guide for testing with VoiceOver](#) provides detailed usage instructions.

Testing on Windows using NVDA

[NVDA](#) is a free (but donation funded) screen reader for Windows platforms.

Once NVDA has been downloaded and installed, you can start it with the keyboard command `Ctrl + Alt + N`.

[WebAIM's guide for testing with NVDA](#) provides detailed usage instructions.

What to look for

When testing with a screen reader, you are testing a different form of user experience. As with any UX consideration, your objective should be to ensure that screen reader users will find your application sensible and usable.

It is likely that problems will jump out at you when you start, but check in particular for the following:

- Image names are not read out by the screen reader; they have appropriate descriptive text
- Elements that are inaccessible to the screen reader are not necessary to the user's understanding
- Blocks of repeating text (e.g. menus) can be skipped
- The page is read in the intended/visual order (not source order)
- Forms can be navigated, completed and understood clearly
- Validation errors can be understood clearly
- The screen reader understands and reads toasts and alerts in a comprehensible way

Additional Notes

When testing with a screen reader, it is important to remember that you will not necessarily have a 100% authentic experience due to difference in screen reading programs and user proficiency. The [most popular screen readers](#) are not free, and though many developers use Macs, most screen reader users will use Windows.

Additional Resources

While this section goes over the key aspects of making an accessible website with respect to Angular, it is not a comprehensive guide for everything related to web accessibility. For more information, please visit one of the resources below:

- [WAI-ARIA Specification](#)
- [Web Content Accessibility Guidelines \(WCAG\) 2.0](#)
- [Section 508](#)
- [WebAIM](#) - Organization with the goal of providing accessibility information
- [The A11y Project](#) - A community driven site to help make web accessibility easier
- [WCAG 2.0 checklists](#)

Testing Resources

- [Web Content Accessibility Toolkit](#)
- [Guide for Evaluating Accessibility](#)
- [pa11y](#) - Automated Accessibility Monitoring

What is the purpose of i18n?

Internationalization (or *i18n*, which stands for *i--n*) is the process of adapting software and web applications to multiple languages, allowing their application to be used by multiple language-speaking users. By ensuring that your application supports multiple languages, you can reach a broader audience and ensure a smooth user experience for multilingual users.

Process & Roles

In order for an application to successfully support multiple languages, multiple team members must closely dedicate their efforts to working together. Translators must be introduced to the team as they will be the ones translating the content and will need guidance on the intent and meaning of the words being used. Using Angular's i18n facilities, the process would look like this:

- Mark text messages in your templates for translation.
- Use an angular i18n tool to extract the messages into an industry standard translation source file.
- Provide the industry standard translation source files to a translator, who will translate the files and give them back to you.
- Import the completed translation files using the Angular compiler

Marking text in our templates for translation

To get started, we will have to specify the text in our templates that we would like to translate. To ensure text is successfully translated, we will have to not only specify **what** text needs to be translated, but we will also need to provide a **description** and a **contextual meaning**. Below is an example of the changes we would have to make to an `<h1>` tag to mark the the text for translation, and provide a description and contextual meaning:

```
<h1 i18n="User welcome|An introduction header for this sample">Hello {{name}}</h1>
```

We use the pipe (`|`) to separate the description (left) and contextual meaning (right).

Using the Angular i18n tool

Now that we've marked our text, let's download an Angular CLI tool called `ng-xi18n` that will extract this text and place it into an [XLIFF](#) or [XMB](#) translation file, depending on your preference.

Once this is done in your templates, you will need to install the CLI and its platform-server peer dependency if you haven't already and then execute the `ng-xi18n` command to generate a translation file:

```
> npm install @angular/compiler-cli @angular/platform-server --save
> ./node_modules/.bin/ng-xi18n
```

By default, an XLIFF file is created but you use can append `--i18nFormat=xmb` if you would prefer the XMB format. The file created would be the file that you would share with translators who would fill in the translations using an XLIFF file editor. Once the translations are done, the translation files are returned back to you.

You can also specify the output folder with the `--project` or `-p` tag. This folder must exist for it to work.

```
> ./node_modules/.bin/ng-xi18n -p locale --i18nFormat=xmb
```

The above command will output a *.xmb file in the `locale` folder.

Running `ng-xi18n` will compile your code before extracting translation texts. It will output `.js` and `.metadata.json` files in your src folders. It might be a good idea to ignore these files when uploading your git repository.

Importing the completed translations files into your Application

There are two ways to do this: You can use the **JiT** (Just in Time) Compiler or **AoT** (Ahead-of-Time) Compiler. Regardless of the approach, you will need to provide the Angular compiler with

- the translation file(s)
- the translation file format
- the Locale Id (autogenerated by `ng-xi18n` and can be found in the translation file)

Importing the Translation Files with the AoT Compiler

To Internationalize with the AoT (Ahead of time) compiler, you will have to:

- pre-build a separate application package for each language
- determine which language the user needs
- serve the appropriate application package

To pre-build a separate application, you will have to ensure that you have the tools required to setup AoT. Refer to the [AoT cookbook](#) for details on how to do this.

Once your ready, use the `ngc` compile command providing the compiler with the following 3 options:

- `--i18nFile` : the path to the translation file
- `--locale` : the name of the locale
- `--i18nFormat` : the format of the localization file

For example, the French language command would look something like this:

```
./node_modules/.bin/ngc --i18nFile=./locale/messages.fr.xlf --locale=fr --i18nFormat=xlf
```

Importing translation files into your application with the JiT Compiler

The JiT (Just-in-time) compiler compiles the application dynamically, as the application loads. To do this, we will need to rely on 3 providers that tell the JiT compiler how to translate the template texts for a particular language:

- `TRANSLATIONS` is a string containing the content of the translation file.
- `TRANSLATIONS_FORMAT` is the format of the file.
- `LOCALE_ID` is the locale of the target language.

Here's how to bootstrap the app with the translation providers for French. We're assuming the translation file is `messages.fr.xlf`.

app/index.ts:

```
import { NgModule, TRANSLATIONS, TRANSLATIONS_FORMAT, LOCALE_ID } from '@angular/core'
;
import { BrowserModule } from '@angular/platform-browser';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { Hello } from './app.component.ts';

// Using SystemJs' text plugin
import translations from './messages.fr.xlf!text';
const localeId = 'fr';

@NgModule({
  imports: [
    BrowserModule
  ],
  declarations: [
    Hello
  ],
  bootstrap: [ Hello ]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule, {
  providers: [
    { provide: TRANSLATIONS, useValue: translations },
    { provide: TRANSLATIONS_FORMAT, useValue: 'xlf' },
    { provide: LOCALE_ID, useValue: localeId }
  ]
});
```

[View Example](#)

We're using SystemJS text plugin to import raw xlf files. We could alternately use webpack and `raw-loader` to achieve the same effect. Better yet, we could make an http call based on which language we're interested in, and asynchronously bootstrap the app once its loaded.

Glossary

Decorators

[@Component](#) [more](#)

[@Directive](#) [more](#)

[@HostListener](#) [more](#)

[@Inject](#) [more](#)

[@Injectable](#) [more](#)

[@Input](#) [more](#)

[@NgModule](#) [more](#)

[@Output](#) [more](#)

[@Pipe](#) [more](#)

[@ViewChild](#) [more](#)

[@ViewChildren](#) [more](#)

Further Reading and Reference

Angular

- [Angular.io API Reference](#) - Angular Reference Material with easy access to different Angular items
- [Angular Style Guide](#) - Opinions from the Angular team
- [Angular Module Github](#) - Source code is written in readable TypeScript
- [Angular Material Github](#) - Official repo for Angular implementation in material design

TypeScript

- [tsconfig options](#) - information on how to configure the TypeScript compiler
- [TypeScript Playground](#) - In-browser TypeScript editor with live reload
- [TypeScript Handbook](#)
- [TypeScript Deep Dive](#) - Additional learning material

General Coding Practice and Functional Programming

- [Mostly Adequate Guide to Functional Programming](#)

RxJS, Reactive Programming and Observables

Observables are known for having a steep learning curve due to the fact that it requires a different way of thinking. Here are some helpful topics about working with and understanding reactive programming using the observable model.

- [RxJS 5 Observables Reference](#) - Reference material on RxJS 5 Observables. There are many breaking changes from RxJS 4-> 5 so please use documentation on version 5.
- [RxMarbles](#) - Quick references for visualizing observable value fulfillment through the use of marble diagrams
- [How to debug RxJS code](#) - Blog post explaining how to read and use marble diagrams
- [RxJS 5 Thinking Reactively](#) - Talk from RxJS 5 product lead on how to approach

development using RxJS 5

- [Learn RxJS](#) - Example driven guide to RxJS

Redux and ngrx

- [A comprehensive guide to ngrx](#)
- [ng2-redux Github](#)
- [ngrx Github](#) - Includes links to ngrx scoped libraries including: store, effects, router
- [Redux Documentation](#)

Keeping up to date

- [Angular Weekly Notes](#)
- [Angular blog](#) - Includes blog posts for Angular 1.x
- [Angular Air](#) - Angular podcast
- [Adventures in Angular](#) - Angular podcast
- [Angular Changelog](#) - Technical Changelog

Other

- [Webpack 2 Official Docs](#)