Rangle.io: Angular 1.x Training

Rangle.io

Published with GitBook

Table of Contents

Introduction to Angular 1.x and ngCourse

AngularJS is the leading open source JavaScript application framework backed by Google. The "1.x" version of AngularJS is in wide use.

This book walks the reader through everything they need to know from getting started with the Angular toolchain to writing applications with scalable front end architectures.

Setup

Each attendee should come to the course with their own computer, so that they can follow the steps. We also ask all attendees to complete the following installation items before the first day of the course.

The Command Line Terminal

The AngularJS tool-chain is quite command-line oriented, so we recommend installing a good command-line terminal program. For **Mac OS X** we recommend iTerm2.

For **Windows**, we'll be using `Git Bash`, which comes with `git`, below.

Git

We'll be using `git` from the command line:

http://git-scm.com/download/

Windows Users: Just select the default options in the installation wizard.

Windows Users: The rest of this course assumes you are using the terminal that comes with `git` (`Git Bash`) to run all command-line examples.

There are tools that provide a GUI interface for `git`, for both OS X and Windows. Attendees who are proficient with `git` and prefer to use those tools can do so if they wish. However, all of our instructions will be for the command line, so we recommend having this available as a backup.

Node.JS

http://nodejs.org/download/

This should install two commands: `node` and `npm`.

`npm` may require some extra configuration to set permissions properly on your system.

On **Mac OS X**, do the following:

```
npm config set prefix ~/.npm
echo 'export PATH="$PATH:~/.npm/bin"' >> ~/.bashrc
~/.bashrc
```

On **Windows**, fire up `Git Bash` and type the following:

```
mkdir /c/Users/$USER/AppData/Roaming/npm
echo 'export PATH="$PATH:/c/Program Files/nodejs"' >> ~/.bashrc
~/.bashrc
```

Note: if you installed the 32-bit Windows version of node, the second line above should instead read:

```
echo 'export PATH="$PATH:/c/Program Files (x86)/nodejs"' >> ~/.bashrc
```

Google Chrome and Batarang

Please install a recent version of Google Chrome, since we'll be using tools that assume it.

Once you have installed Chrome, please install Angular Batarang, which is an Google Chrome extension that makes it easier to debug and inspect Angular applications. You can get it from the Google Webstore

Postman HTTP Client

Postman is our preferred tool for interacting with a REST API server.

http://www.getpostman.com/

Postman is less essential and if you have another HTTP client that you prefer, this is fine too.

A Code Editor

Any text editor will work. At Rangle.io the most popular editors/IDEs are:

- Sublime Text
- Atom
- Visual Code
- WebStorm
- Vim

Verifying the Setup

Once you have all the tools set up correctly, you should be able to do the following steps through the command line. This is the best way to check that `git`, `node`, and `npm` were installed correctly.

Clone the training repository:

```
git clone https://github.com/rangle/ngcourse-next.git
cd ngcourse-next
```

Install the project's `npm` modules:

```
npm install
```

Fire up the development server:

```
npm start
```

Once you've run those commands, you should be able to access the server at http://localhost:8080. If you see a login form at that point, then you did everything correctly and are ready for the course.

Proxy Issues

While running the above commands you might get errors related to proxies. If this is the case, you'll need to configure your command-line tools to handle proxies.

Git

This most likely means that you are behind a proxy that blocks SSH access to Github. In this case, run the following command:

```
git config --global url."https://".insteadOf git://
```

Then re-run the failed command.

It's a somewhat blunt weapon, but should work in most cases. If this doesn't work, and if you know the URL of your proxy server, you can try this:

```
git config –global http.proxy <proxy server url>
```

npm

```
npm config set proxy <proxy server url>
```

Introduction to AngularJS and ngCourse-Next

AngularJS is the leading open source JavaScript application framework backed by Google. The "1.x" version of AngularJS has been used quite widely. The new "Angular 2" version of the framework is currently available as a preview.

This course ("ngCourse-Next") provides an introduction to AngularJS based on our experience at rangle.io and with an eye to eventual transition to Angular 2. In other words, we will be learning how to build single page applications using Angular 1.4, but in such a way as to make the eventual transition to Angular 2 as easy as possible.

AngularJS, the Good Parts

Douglas Crockford's seminal book *JavaScript, The Good Parts* has this to say about JavaScript:

> Most languages contain good parts and bad parts. I discovered that I could be a better programmer by using only the good parts and avoiding the bad parts... JavaScript is a language with more than its share of bad parts.

Crockford goes on to point out that JavaScript also has lots of *good* parts which can make it a great language to use. The key is avoiding the bad parts.

(Crockford's book should be required reading for all JavaScript developers. At rangle.io, we keep a few extra copies on hand on the off-chance that one of our developers has not already read this great book.)

What Crockford says about programming languages applies equally well to frameworks: most have good parts and bad parts. Our approach to AngularJS 1.x has somewhat resembled Crockford's approach to JavaScript. AngularJS 1.x does not have nearly the same kind of warts as JavaScript, but it does have features that will help you shoot yourself in the foot, even as many of its other features help you build highly scalable software. When we've taught this course, we always tried to highlight "the good parts" and to show how you can use them to your advantage.

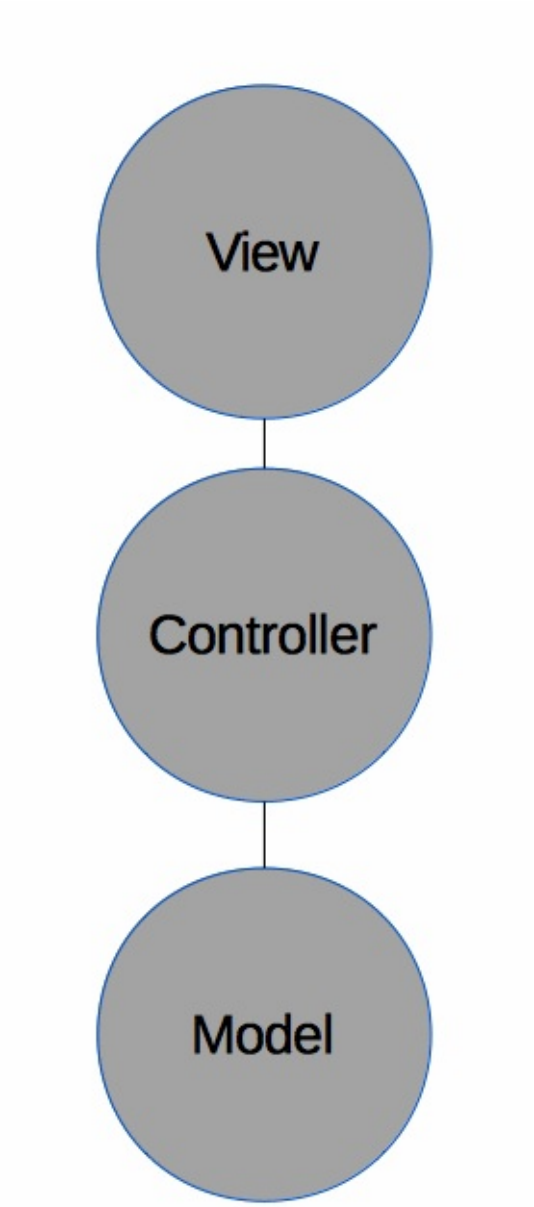
The upcoming Angular 2 brings a number of changes. Some of those changes involve removing those parts of Angular that have proved to not work well. Other changes involve borrowing the best ideas from outside the Angular ecosystem. Finally, another large set of changes aims to take advantage of new features that are becoming available in ES6, the new version of JavaScript.

Angular 2 is not yet ready for real-world use, but anyone who is starting to work on an Angular application today should be planning an eventual transition to Angular 2.

This course teaches you to write applications using Angular 1.4 the Angular 2 way.

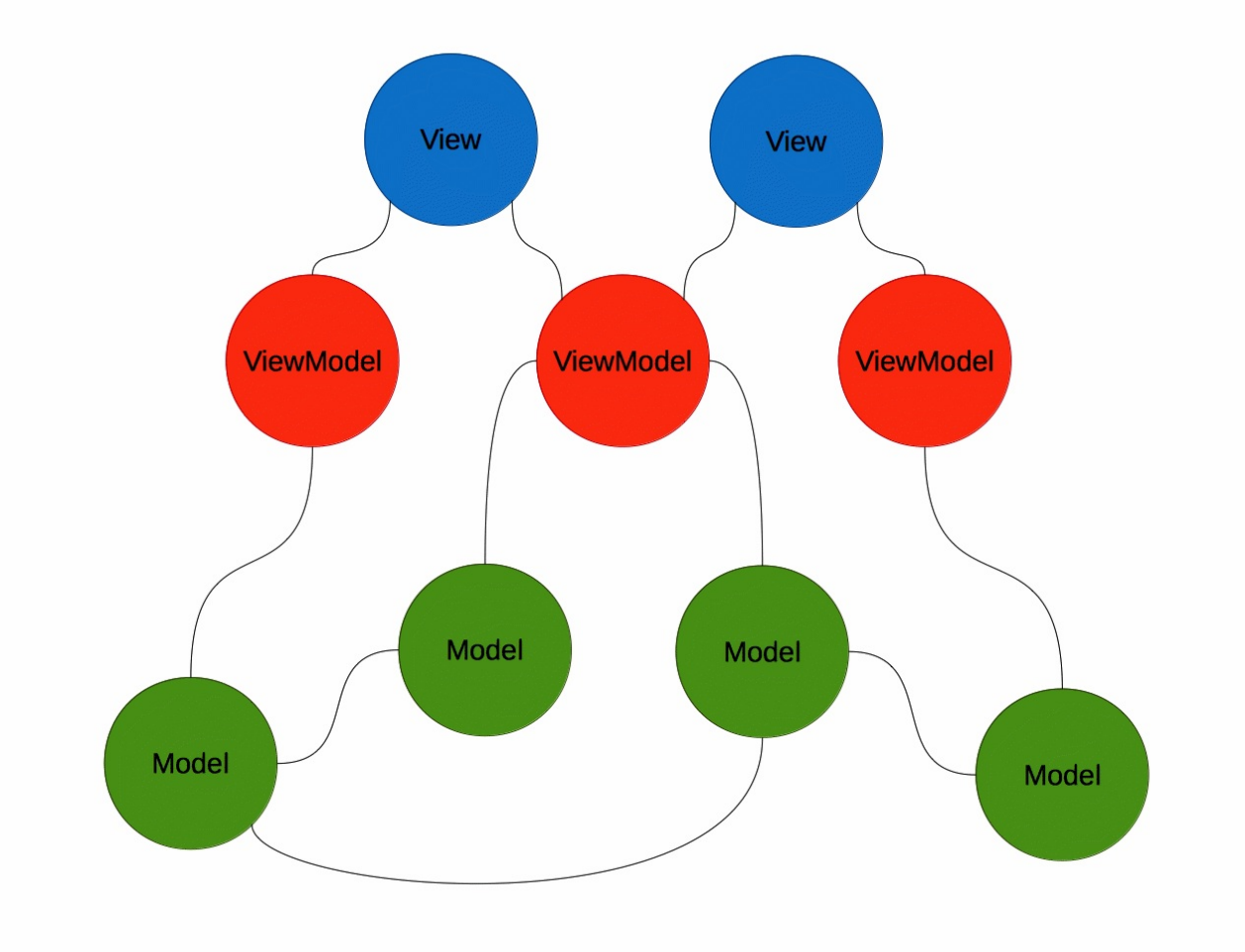
MVC and MVVM

AngularJS is often described as an MVC ("Model-View-Controller") framework. Here is how this is often illustrated:



This picture, however, is far too simple.

First, only the most trivial applications can be understood as consisting of a single model, a single view and a single controller. More commonly, an application will include multiple views, multiple controllers, and multiple data models. So, it might look more like this:

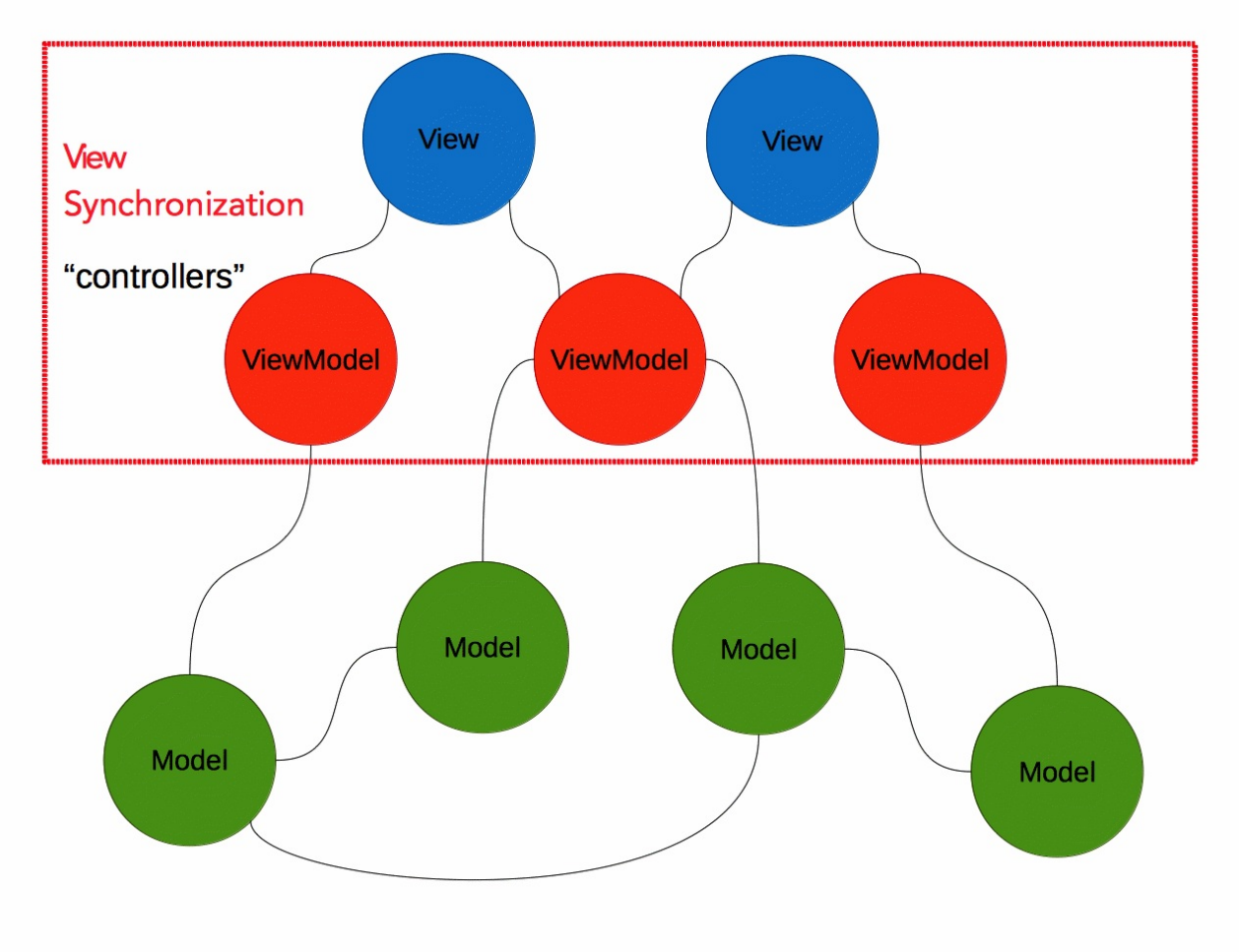


The figure above makes another important substitution, however. "Controllers" are replaced with "view models". Angular can be better understood as a "MVVM" ("Model-View-ViewModel") framework. In this approach, we have "view models" mediating between views and (data) models. While this may seem like just a minor change of terminology, the idea of "view model" helps clarify the path towards better AngularJS architecture. A view model is a mediating object that takes data from a data model and presents it to a view in a "digested" form. Because of that, the view model superficially looks like a model. It should not be confused with the application's real data models. Misusing the view model as the model is one of the most common sources of problems in AngularJS.

Now let's see how MVVM model is realized in AngularJS.

View Synchronization

Most introductions to Angular start with a look at the "front-end" of the framework. Let's do the same here, even though most of your AngularJS code should be in the model layer.



AngularJS views are HTML templates that are extended with custom elements and attributes called "directives". AngularJS provides you with a lot of directives and you will also be developing some yourself.

Views are linked with view models that take the form of "controllers" and custom "directives". In either case we are looking at some code that controls JavaScript objects (the actual "view model") that are referenced in the templates. Angular 2 merges "controllers" and "directives" into a single concept of a "component". In this course we will be building our application as a collection of components, though we will be practically implementing each component as a combination of a controller and a directive.

AngularJS automatically synchronizes DOM with view models when the view model changes. It also allows us to associate function handlers with DOM events. Both of those methods are preserved and generalized in Angular 2. Angular 1.x also has a concept of

"two-way databinding", where properties of the view model can be automatically changed to reflect changes to DOM properties. This approach is deprecated in Angular 2 and we will avoid it.

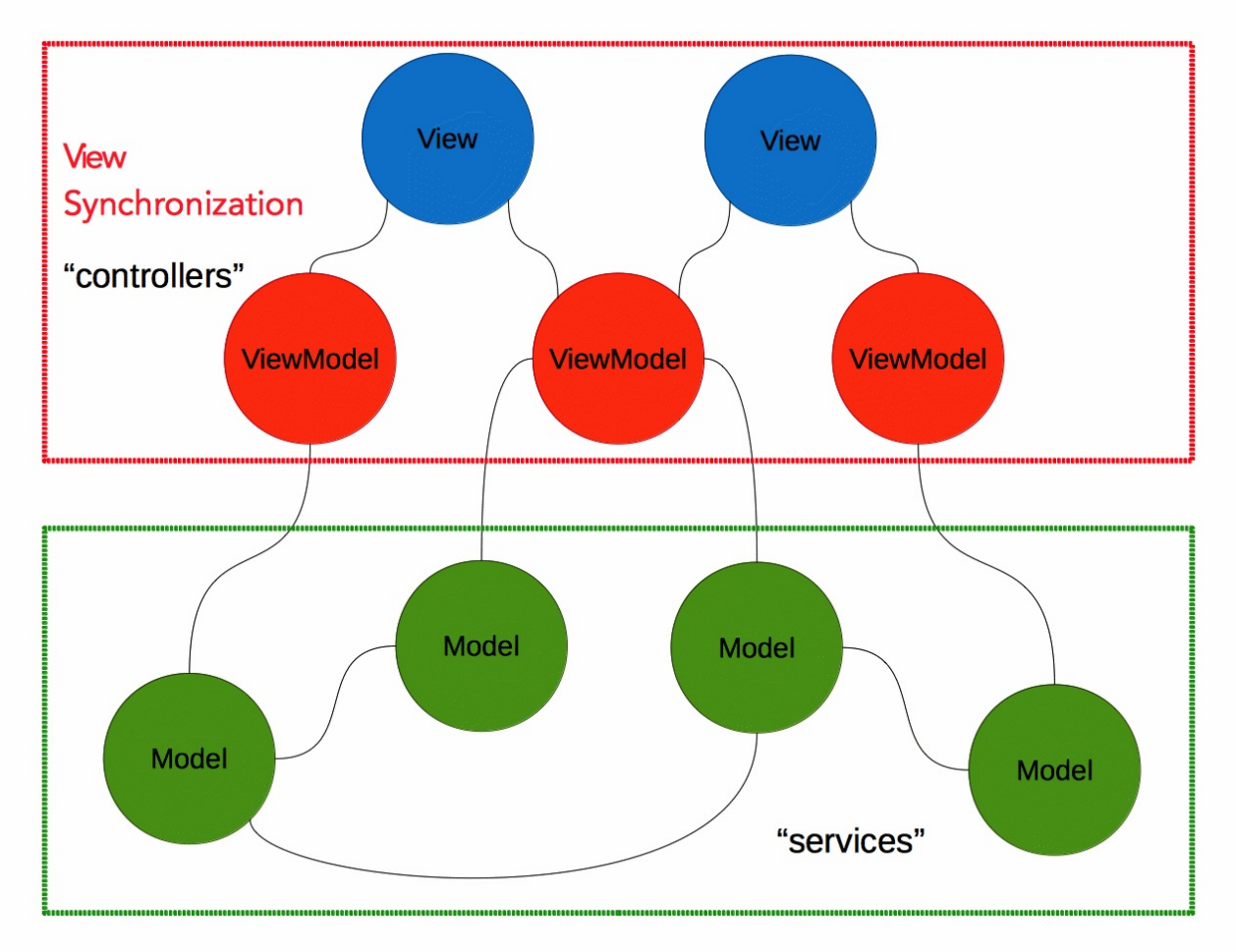
Angular's approach to view synchronization makes it very "designer-friendly": designers can modify HTML templates without worrying too much about the code. The reverse is also true: as long as there is a designer on the team, developers are largely freed from worrying about HTML and CSS. Angular 2 preserves this feature.

Angular 1.x allows you to organize your view models into a hierarchy of "scopes" that partly mirrors DOM structure. This approach has proven problematic and we've usually recommended against it. It is being dropped entirely in Angular 2. For this reason, we will avoid it completely in this course. Instead, we will aim to make our components fully isolated. We'll also aim to have them do as little work as possible. Instead, most of the work (in particular, all of the business logic) should be moved to the lower "model" level.

More generally, it is important to understand that view models are a temporary staging area for your data on the way to the view. They should not be used as your primary model.

Models in Services

AngularJS does provide us with a great way to implement our data models at arm's length from the views using a mechanism called "services".



Services are singleton objects that normally do not concern themselves with the DOM but instead take care of your data. The bulk of your application's business logic should belong in services. We'll spend a lot of time talking about this.

Services get linked together through an approach that AngularJS calls "dependency injection". This is also how they are exposed to view models (controllers and custom directives, or "components" going forward).

In the case of Angular, what "dependency injection" practically means is that components do not get to create and define their dependencies. Instead, services are created *first*, before any components are instantiated. Each component's definition specifies what dependencies should be provided to the component.

Angular's dependency injection is one of the best things about the framework. This approach makes your code more modular, reusable, and easier to test. Those features are essential when building larger applications.

ES6 and TypeScript

Angular 2 makes use of a number of features of ES6 and TypeScript. Using Angular 2 with ES5 (the current version of JavaScript) is possible but cumbersome. It is possible, however, to write Angular 1.4 code using TypeScript. So, that's what we will be doing in this course.

The Summary of Different Approaches

	Old School Angular 1.x	Angular 1.x Best Practices	Angular Next	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
Requires a transpiler	No	No	**TypeScript**	TypeScript

The JavaScript Toolchain

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

Source Control: Git

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

The Command Line

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

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

Command-line JavaScript: Node.JS

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

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

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

Back-End Code Sharing and Distribution: npm

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

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

Module Loading, Bundling and Build Tasks: Webpack

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

Chrome

Chrome is the web browser from Google. We will be using it for this course because of its cutting-edge JavaScript engine and excellent debugging tools.

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

Getting the Code

Before we proceed, get the code from Git if you have not done so:

```
git clone https://github.com/rangle/ngcourse-next.git
cd ngcourse-next
```

Now, create, and switch to the branch that we'll be using for today's course:

```
git branch --track YYYY-DD-MM origin/YYYY-DD-MM
git checkout YYYY-DD-MM
```

Replace "YYYY-MM-DD" above with the start date of the course, e.g., "2015-06-17".

This gives us a hollowed-out version of the application we'll be building.

Install NPM and Bower packages:

```
npm install
```

Start webpack's development server

```
npm start
```

We can also serve the app from a build directory using something like `http-server`.

Install it if you have not yet done so:

```
npm install -g http-server
```

```
http-server app/__build
```

EcmaScript 5 for AngularJS Developers

JavaScript is an untyped, interpreted programming language that can accomodate a variety of programming paradigms. Among other things, a lot of modern JavaScript code heavily leverages functional programming style. The combination of weak typing and functional methods can make JavaScript code a bit hard to understand for those coming from strongly typed object-oriented languages such as Java.

This module is intended for an audience who is new to JavaScript, or one that simply needs a refresher. We will walk through the basic JavaScript constructs that make up an AngularJS application.

Let's begin by dissecting the following example of AngularJS code:

```
angular.module('ngcourse')

.controller('MainCtrl', function($scope) {
  $scope.username = 'alice';
  $scope.numberOfTasks = 0;
});
```

This code defines an AngularJS controller called `MainCtrl` as a part of a module `ngcourse`. The way it does it may seem counterintuitive at first.

Function Chaining / Fluent interfaces

If we look at files that make up an AngularJS application, a typical file would often consist of a single giant JavaScript statement where multiple method calls are chained together: we call a method on an object, get another object, then call a method on that object, get another method, etc. JavaScript allows us to insert white space before the `.` that precedes a method invocation, so this:

In the previous example,

```
angular.module('ngcourse')

.controller(...);
```

is equivalent to this:

```
angular.module('ngcourse').controller(...);
```

To support this `fluent` programming style, Angular ensures that methods such as `.controller()` return the object to which the method belongs. This allows us to define another controller after the first one, while still working with the same expression:

```
angular.module('ngcourse')

.controller('MainCtrl', function($scope) {
  ...
})

.controller('TaskListCtrl', function($scope) {
  ...
});
```

To understand what is happening here, let's consider an alternative style where we actually save the objects in variables:

```
var ngCourseModule  = angular.module('ngcourse');

ngCourseModule.controller('MainCtrl', function($scope){ ... });

ngCourseModule.controller('TaskListCtrl', function($scope){ ... });
```

Note that the return value of `ngCourseModule.controller()` is the same object as `ngCourseModule`.

This style may seem more intuitive to those coming to JavaScript from other languages. There is a reason, however, why we do not use this style. The example above defines a new variable `ngCourseModule`. Unfortunately, a variable defined outside of a function becomes *global* in JavaScript. We'll come back to global variables and ways to avoid them in a short while.

For now, however, let's note that chained method calls provide us with a way to avoid defining new variables. We will see this pattern often in AngularJS applications, for example when defining services or directives.

JavaScript Functions

To define a controller we call module's `.controller` method with two arguments. The first is the name of the controller, the second one is a function that implements it:

```
angular.module('ngcourse')
.controller('MainCtrl', function($scope) {...});
```

In languages such as Java, arguments to functions and methods can be objects or primitives, but not functions. In JavaScript, however, functions are first class citizens. A function can be passed into another function as a parameter. A function can return a function. Functions can also be assigned to variables.

JavaScript allows two ways of defining a function. In the first method, called "function declaration", a new function is defined in the current scope and is given a name:

```
function foo() {
  // do something
}
```

We will be able to refer to this function as `foo` in the current scope. It is worth noting that functions defined using function declarations are "hoisted" in JavaScript. Regardless of where you define a function in the current scope, JavaScript would act as if the function was defined up front. So, this is perfectly valid:

```
// Call a function.
foo();

// Now provide a definition.
function foo() {
  // do something
}
```

An alternative method of defining a function is a "function expression". In this case, provide a function definition in a context where JavaScript would expect to see an expression:

```
var foo = function foo() {
  // do something
}
```

Functions defined this way are *not* hoisted, so this would be invalid:

```
// This call will fail because the value of "foo" is undefined at this point.
foo();

// The function is defined, but it's too late.
var foo = function foo() {
  // do something
}
```

Functions defined as function expressions do not need to have names:

```
var foo = function() {
  // do something
}
```

If we do provide a name in a function expression, we won't be able to call the function by this name, but the function will use name when reporting errors.

```
var foo = function bar() {
  // This function thinks it's called "bar" and will use this name when
  // reporting errors. We cannot call it by this name, however.
}

bar(); // This will fail.
```

In our case of defining an Angular controller, we use a function expression to define an anonymous function and then pass it as the second argument argument to `controller()`:

```
.controller('MainCtrl', function($scope) {
  ...
})
```

Variable Scope

In JavaScript, variables are *global* unless declared inside a function. Global variables can make code very hard to debug and maintain, so you must always be careful not to create global variables unintentionally.

The only way to create local variables in JavaScript is to define them inside a function. Doing so, however, requires defining a function. If we were to use a function declaration, we would end up polluting the global name space with the function itself.

Consider this example:

```
function foo() {
  var bar = 1;
  // do something with bar;
}
```

Here we succeeded in making `bar` local, but we created a global function `foo()`!

We can solve this "catch-22" situation by using a function expression:

```
(function() {
  var bar = 1;
  // do something with bar;
})();
```

Here we defined `bar` inside an *anonymous* function that we call right away. This pattern is very common in JavaScript and is called "an immediately-invoked function expression" or "IIFE". We do *not* usually use this style in AngularJS, however. Instead, we normally rely on functions that we define as arguments to module methods:

```
.controller('MainCtrl', function($scope) {
  var bar = 1; // This will be local to this function.
  ...
})
```

It is worth noting that unlike some languages, JavaScript does not support "lexical scoping": variables are defined at the level of a function, not a block. (This is being fixed in ES6 with the keyword "let".)

Nested Scopes

In JavaScript, a function can be declared within another function. The function has a local (function) scope, it has access to the scope of the outer function, and it also has access to a global scope.

Here is a pure JavaScript example with multiple scopes:

```
var x = 1;
var y = 2;

console.log('in global scope: ', x, y);

function func() {
  var x = 2;
  y = 2;
  console.log('in func(): ', x, y);

  function innerFunc() {
    var x = 3;
    y = 3;
    console.log('in innerFunc(): ', x, y);
  }
  innerFunc();

  console.log('in func() again: ', x, y);
};
func();
console.log('in global scope again: ', x, y);
```

Executing the code above will produce the following:

```
in global scope:  1 2
in func():  2 2
in innerFunc():  3 3
in func() again:  2 3
in global scope again:  1 3
```

Notice that here we declare `x` again in each scope. This makes `x` act as different variables in different scopes. Changing the value of `x` in inner scopes does not affect its value in the outer scopes.

For `y`, the behavior is quite different. We do not declare it again in the nested scopes, which means the outer scope's variable is used. Setting the value of `y` inside a nested scope changes its value in the outer scope.

EcmaScript 6 and TypeScript

The new version of JavaScript, "EcmaScript 6" or "ES6", offers a number of new features that extend the power of the language. (The language we usually call "JavaScript" is actually formally known as "EcmaScript".) 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 Angular team uses to write Angular 2.

Classes

ES5 has objects, but it has no concept of class. ES6 introduces classes.

```
class LoginFormController {
  constructor() {
    // This is the constructor.
  }
  submit() {
    // This is a method.
  }
}
```

This is pretty straightforward, until we run into the oddities of how `this` keyword works in JavaScript.

A Refresher on `this`

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

```
class LoginFormController {
  ...
  submit() {
    var form = {
      data: this
    };
    this.fireSubmit(form);
  }
}
```

Here `this` refers to an instance of the `LoginFormController` class. As long as the submit method is called using dot notation, like `myComponent.submit()`, then `this.fireSubmit(form)` invokes the `fireSubmit()` method defined on the instance of the class. This will also ensure that inside `fireSubmit`, that `this` refers to the same instance.

However, `this` can also refer to other things.

There are two basic cases that you would want to remember.

The first is "method invocation":

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

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

The second case is "function invocation":

```
  someFunction();
```

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

View Example

One of the implications of this is that you cannot easily detach a method from its object. E.g., 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.

Arrow Functions

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

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 share the value of `this` with the function in the context of which they are defined. Consider the following example:

```
class LoginFormController {
  constructor() {
    this.errorMessage = 'All good.';
  }
  submit() {
    [1, 2].forEach(function() {
      console.log(this.errorMessage); // this is undefined here
    })
  }
}

var ctrl = new LoginFormController();

ctrl.submit();
```

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

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

```
class LoginFormController {
  constructor() {
    this.errorMessage = 'All good.';
  }
  submit() {
    [1, 2].forEach(() => console.log(this.errorMessage));
  }
}
```

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

Inheritance

JavaScript's inheritance works differently from inheritance in other languages, which can be very confusing. The ES6 `class` operator is syntactic sugar that attempts to alleviate the issues with using prototypical inheritance present in ES5. Our recommendation is still to avoid using inheritance or at least deep inheritance hierarchies. Try solving the same problems through delegation, aggregation, or functional composition instead.

TypeScript

As of right now, no browser comes even close to supporting all of ES6. Support for classes is especially poor. Instead, we will need to rely on a transpiler to convert our ES6 code to ES5.

Several ES6 transpilers are available. One popular option is Babel. However, we will be using TypeScript transpiler, which is what Angular team users to write Angular 2.

TypeScript transpiler is different from other ES6 transpilers in that it expects as input not standard EcmaScript 6, but rather TypeScript, an extension of ES6 that adds support for optional typing.

We can install the TypeScript transpiler using npm:

```
npm install -g typescript
```

We can then use `tsc` to manually compile a TypeScript source file into ES5:

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

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

```
class LoginFormController {
  errorMessage: string;
  constructor() {
    this.errorMessage = 'All good.';
  }
  submit() {
    [1, 2].forEach(() => console.log(this.errorMessage));
  }
}
```

Note that now that we've declared `errorMessage` to be a string, 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 it's type to be "any":

```
class LoginFormController {
  errorMessage: any;
  ...
}
```

TypeScript with Webpack

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

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

Loading ES6 Modules

ES6 also introduces the concept of a module, which works in a similar way to other languages. Defining an ES6 module is quite easy: each file is assumed to define a module and we'll specify it's exported values using the `export` keyword. Loading ES6 modules is a little trickier.

In a ES6 compliant browser you would be using the `System` keyword to load modules asynchronously. To make our code work with the browsers of today, however, we will use 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>
```

Additional ES6 Features

In addition to classes and arrow functions, ES6 offers numerous other features not available in ES5. Most of those are currently supported by TypeScript. We won't discuss those features here, though we will come across some of them in the course. We encourage you to learn more about them on your own, however. This repository provides a good overview: https://github.com/lukehoban/es6features.

Getting Started with the Client

This course will be organized around building a collaborative task manager. We will start by building a client app, which we will later connect to a REST API. Our first task is to setup a simple Angular app consisting of a few **components**, and to understand how they fit together.

We'll be making use of common built-in directives such as `ng-model`, `ng-show`, `ng-hide`, `ng-cloak`, `ng-if`, `ng-repeat`. We will also discuss Angular's dependency injection and the use of `$log` for logging.

The Most Trivial Angular App

Let's start by setting up a really simple angular app -- so simple in fact that it won't do anything at all. Here is what we'll put in our *index.html* file.

```
<!DOCTYPE html>
<html>

  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1

    <title>NgCourse-Next Demo Application</title>
  </head>
  <body>
    <div>Hello World!</div>
  </body>
</html>
```

We'll also need a very simple TypeScript file - our "app":

```
angular.module('ngcourse', []);

angular.element(document).ready(
  () => angular.bootstrap(document, ['ngcourse'])
);
```

Angular 1.x Basics

This app doesn't do anything at all. To make it do something remotely interesting we'll need to define a directive.

Directives

In Angular 1.x, directives are the building blocks of your application. Directives can be described as markers on the DOM tree that allow to define custom behaviour and/or transformations on that DOM element.

Let's define a basic directive in our *app/src/app.ts* file to see this in action,

```
...
angular.module('ngcourse', [])
  .directive('ngcMain', () => ({
      restrict: 'E', // vs 'A', 'AE'
      replace: true,
      scope: {}, // vs 'true', 'null'
      template: '<span>Hello World from Directive!</span>'
    })
  );
...
```

Note the way `angular.module()` is invoked in these two files. The `module` function can be used in two ways.

1. `angular.module('ngcourse', ['ngcourse.directives'])` defines a new module with a name of 'ngcourse' that has dependencies on other modules specified in the dependency array pointing to other modules by name. *(More on AngularJS' dependency injection will be covered later)*.
2. `angular.module('ngcourse')` which accesses a module that has already been defined.

We already saw code that is similar, so we recognize JavaScript's "fluent" chaining style and the use of a function expression in the second argument to `directive()` .

And now we can use our directive in our *index.html* as follows:

```
...
  <body>
    <ngc-main></ngc-main>
    ...
  </body>
```

Note that we used "camelCase" when we defined this directive in our Angular application, but we used hyphens when inserting them into the HTML.

Angular will figure out that `<ngc-main></ngc-main>` refers to the directive that we defined as `ngcMain` .

Transclusion with `ng-transclude`

The way we defined our `ngc-main` directive above will ignore anything between the directive tags, as illustrated by example below.

```
<ngc-main>This text will be thrown away.</ngc-main>
```

In some situation we would like the content to be preserved and shown on the DOM. To achieve this we will need to modify the transclude property of our directive like so:

```
...
  .directive('ngcMain', () => ({
      ...
      transclude: true,
      template: '<span>Hello World from Directive! <div ng-transclude/></span>'
    })
  );
...
```

Controllers

Our application still does not do very much. In order to add behaviour to our directive, lets define a controller class with some simple logic.

```
class MainDirectiveCtrl {
  private userDisplayName;
  constructor() {
    this.userDisplayName = 'Mike Tyson';
  }
}

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

Note Angular's `{{ }}` syntax, referred to as the double moustache, used here to bind controller's property to the template.

Let's recap:

Template

The template is just an HTML snippet defining a view that represent this directive. Templates have access to any properties or functions defined on the directive's controller scope.

Controller

The controller is just an ES6 class that backs component's view represented by a template. The template above binds the `userDisplayName` property defined on the `MainDirectiveCtrl` controller class using the double-moustache syntax `{{ ctrl.userDisplayName }}` .

Using an External Template

Our templates are usually too complex to include as a string. So, instead we often provide a URL to the template file by using `templateUrl` instead of the `template` option in our Directive Definition Object (DDO).

Let's create a new directory *app/src/components/main/* and extract our template into a html file called *main-component.html*. Our templateUrl option should now point to *components/main/main-component.html*.

Using an External Controller Class

In the same fashion we should extract our controller class into a separate file, as we want to avoid clutter when our application grows.

Create a file called *main-component.ts* in the *app/src/components/main/* folder and move our controller class definition there.

Import and Export

Moving the controller class into a separate file is not enough as we need to reference it within our main *app.ts* file. That is what the ES6 import/export syntax is useful for.

Change the controller class definition in the *main-component.ts* file to include the export keyword as follows:

```
export class MainDirectiveCtrl {
  private userDisplayName;
  constructor() {
    this.userDisplayName = 'Mike Tyson';
  }
}
```

Now in *app.ts* lets import our class as follows:

```
import {MainDirectiveCtrl} from './components/main/main-component';
...
angular.module('ngcourse', []);

angular.module('ngcourse')
  .directive('ngcMain', () => ({
      restrict: 'E',
      scope: {},
      template: require('./main-component.html'),
      controller: MainDirectiveCtrl,
      controllerAs: 'ctrl',
      bindToController: true
    })
  );

angular.element(document).ready(
  () => angular.bootstrap(document, ['ngcourse'])
);
...
```

Using `require` to load an external template

Let's discuss the use of `require` to load an external template and how this is achieved using webpack's raw loader.

Note: Even though it is technically possible to use ES6 style imports to load the template, `require` ends up being cleaner.

With Angular 1.x basics out of the way we can start talking about...

Components

As directive in Angular 1.x, in Angular 2, components are the building blocks of your application. As a matter of fact what we built in the previous section is referred to as **Component Directive** within Angular 1.x context.

Angular 2's components are conceptually similar to component directives from Angular 1.x. The structure of Angular 2 application can be viewed as the tree of components, with a root element of that tree being the entry point of your application.

In summary, component is an object that structures and represents a UI element. It consists of two parts, component **controller** in charge of view logic and component **template** representing the view.

With that in mind let's define a basic component in a separate typescript file located in *app/src/components/main/main-component.ts*:

```
export class MainComponent {

  private username;
  private numberOfTasks;

  static selector = 'ngcMain';

  static directiveFactory: ng.IDirectiveFactory = () => {
    return {
      restrict: 'E',
      scope: {},
      template: require('./main-component.html'),
      controller: MainComponent,
      controllerAs: 'ctrl',
      bindToController: true
    };
  };

  constructor() {
    this.username = 'alice';
    this.numberOfTasks = 0;
  }
}
```

Using Components in your Angular 1.x Application

As a result of important similarities between components in Angular 2 and component directives from Angular 1.x, we can write Angular 2 style components today and future proof our applications.

In the previous section we learned how to define a component, now we need to use this component within our Angular 1.x application context. Components in Angular 2 share many of important similarities with component directives from Angular 1.x, and as a result it only makes sense to use `.directive()` function to instantiate them today.

Lets change our *app.ts* and let Angular know about our component via the `.directive()` function.

```
...
angular.module('ngcourse')
  .directive(
    MainComponent.selector,
    MainComponent.directiveFactory);
});
```

Finally, we can now use this component in our *index.html* as follows:

```
  <body>
    <ngc-main></ngc-main>
    ...
  </body>
```

Handling Events with Components

If we put functions onto the component's scope, we can attach those functions to DOM events.

Let's add a `addTask()` method to our `MainComponent` class:

```
...
export class MainComponent {
  ...
  public addTask() {
    this.numberOfTasks += 1;
  }
};
...
```

We need to modify our component's template and add a button element with an `addTask()` function attached to it's click event:

```
<div>
  <span>
    Hello, {{ ctrl.username }}!
    You've got {{ ctrl.numberOfTasks }} tasks.
    <button ng-click="ctrl.addTask()">Add task</button>
  </span>
</div>
```

Note the use of `ng-click` directive here.

A Look at Dependency Injection (DI)

Dependency Injection (DI) is a design pattern that allows software components to get references to their dependencies. DI allows to structure software in a way where components are decoupled from each other. This results in modular software structure with independent components which are much more unit-test friendly.

Injecting Dependencies into Components

Let's start with injecting Angular's `$log` service into our component:

```
...
export class MainComponent {
  ...
  static $inject = ['$log'];
  ...
  constructor( private $log ) {
    ...
  }

  public addTask() {
    this.$log.debug('Current number of tasks:', this.numberOfTasks);
    this.numberOfTasks += 1;
  }
}
...
```

In the code above we are injecting a `$log` service into our component by adding `$inject` static property to our component class. The reference to `$log` is available in the constructor.

Note that a `private $log` parameter in the constructor automatically creates a property of the same name on the class scope, accessible using `this.$log`.

Injecting Multiple Dependencies

This:

```
...
export class MainComponent {
  ...
  static $inject = ['$log', '$scope'];
  ...
  constructor(private $log, private $scope) {
  ...
}
...
```

is equivalent to this (Don't do this!):

```
...
export class MainComponent {
  ...
  static $inject = ['$log', '$scope'];
  ...
  constructor(private $a, private $b) {
  ...
}
...
```

But this:

```
...
export class MainComponent {
  ...
  static $inject = ['$log', '$scope'];
  ...
  constructor(private $scope, private $log) {
  ...
}
...
```

Will not work at all.

In short, the order of parameters in the `$inject` property relative to the class constructor is important.

Two-Way Data Binding with `ng-model`

We can also control our component's property value from within the HTML. Modify the template of our component to include the following:

```
<div>
  Enter username: <input ng-model="ctrl.username"/>
  <br/>
  <span>
    Hello, {{ ctrl.username }}!
    You've got {{ ctrl.numberOfTasks }} tasks.
    <button ng-click="ctrl.addTask()">Add task</button>
  </span>
</div>
```

In the above example, the `ng-model` directive bi-directionally binds an element to our component's class property. Note that if the property does not exist on the controller, it will be created.

Important Note: Two way binding in Angular 2 is one of the biggest changes compared to Angular 1.x. Angular 2 provides a mechanism allowing us to achieve 2-way data binding similarly to today, however this is mostly syntactic sugar while the underlying framework is different. A more detailed look into this will be provided in one of subsequent chapter of this course.

Implementing "Login"

Let's modify our component's template to hide the login form upon login and show the task counter.

```
<div>
  <div ng-hide="ctrl.isAuthenticated">
    Enter username: <input ng-model="ctrl.username"/><br/>
    Password: <input type="password" ng-model="ctrl.password"/><br/>
    <button ng-click="ctrl.login()">Login</button>
  </div>
  <div ng-show="ctrl.isAuthenticated">
    Hello, {{ ctrl.username }}!
    You've got {{ ctrl.numberOfTasks }} tasks<br/>
    <button ng-click="ctrl.addTask()">Add task</button>
  </div>
</div>
```

Note the use of `ng-hide` and `ng-show` directives here.

We'll also need to modify our component's controller as follows:

```
export class MainComponent {
  ...
  private isAuthenticated: any;
  private numberOfTasks: any;
  ...
  constructor(private $log ) {
    this.numberOfTasks = 0;
    this.isAuthenticated = false;
  }

  public login() {
    this.isAuthenticated = true;
  }

  public addTask() {
    this.$log.debug('Current number of tasks:', this.numberOfTasks);
    this.numberOfTasks += 1;
  }

};
```

Splitting Up the Components

By this point our component is getting unwieldy. Let's split it into two separate components.

The first component will be located in *app/src/components/task-list/task-list-component.ts* and will implement our simple task counter.

```
export class TaskListComponent {

  private numberOfTasks;

  static selector = 'ngcTasks';

  static directiveFactory: ng.IDirectiveFactory = () => ({
    restrict: 'E',
    controllerAs: 'ctrl',
    scope: {},
    bindToController: true,
    controller: TaskListComponent,
    template: require('./task-list-component.html')
  });

  constructor(private $log ) {
    this.numberOfTasks = 0;
  }

  public addTask() {
    this.$log.debug('Current number of tasks:', this.numberOfTasks);
    this.numberOfTasks += 1;
  }

};
```

We should also create a template file for this component with the familiar markup:

```
<div>
  <span>
    Hello, {{ ctrl.username }}!
    You've got {{ ctrl.numberOfTasks }} tasks.
  </span>
  <button ng-click="ctrl.addTask()">Add task</button>
</div>
```

The second component will be remain at *components/main/main-component.ts* and will be responsible for user authentication.

```
export class MainComponent {

  private isAuthenticated;
  ...
  constructor(private $log) {
    this.isAuthenticated = false;
  }

  public login() {
    this.isAuthenticated = true;
  }

};
```

```
<div>
  <div ng-hide="ctrl.isAuthenticated">
    Enter username: <input ng-model="ctrl.username"/><br/>
    Password: <input type="password" ng-model="ctrl.password"/><br/>
    <button ng-click="ctrl.login()">Login</button>
  </div>
  <div ng-show="ctrl.isAuthenticated">
    <ngc-tasks></ngc-tasks>
  </div>
</div>
```

The last thing remaining is to wire up our components within Angular application context.

```
...
import {MainComponent} from './components/main/main-component';
import {TaskListComponent} from './components/task-list/task-list-component';
...
angular.module('ngcourse')
  .directive(
    MainComponent.selector,
    MainComponent.directiveFactory)
  .directive(
    TaskListComponent.selector,
    TaskListComponent.directiveFactory);

angular.element(document).ready(
  () => angular.bootstrap(document, ['ngcourse'])
);
```

Simplifying `import`s

We can simplify our `import` statements further to make our life, just a little bit easier. Let's create a new file, *app/src/components/index,ts* and put the following code in there.

```
export * from './task-list/task-list-component';
export * from './main/main-component';
```

Now in our app.ts we can change our import statement to the following:

```
import {MainComponent, TaskListComponent} from './components/index';
...
```

ES6's module system is smart enough to figure out that *index.ts* is the default file in the directory. So we can simplofy this even further.

```
import {MainComponent, TaskListComponent} from './components';
...
```

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.

Let's try adding another `<ngc-tasks></ngc-tasks>` element to the template of a component we defined in *app/src/components/main/main-component.ts* and observe what happens in the browser.

Passing Data Between Components

We have introduced a bug during our re-factoring, the username is not displayed when `TaskListComponent` is shown. Let's modify *task-list-component.ts* and fix it:

```
...
export class TaskListComponent {

  static directiveFactory: ng.IDirectiveFactory = () => ({
    restrict: 'E',
    controllerAs: 'ctrl',
    scope: {},
    bindToController: {
      username: '=username'
    },
    controller: TaskListComponent,
    template: require('./task-list-component.html')
  });
  ...
}
```

and in *main-component.ts* let's change our template as follows:

```
...
<ngc-tasks username="ctrl.username"></ngc-tasks>
...
```

Now the `username` property is passed from `MainComponent` to `TaskListComponent` and this is how we can pass data "into" a child component.

Responding to Component Events

Let's restructure our code further and create a new component to handle the login form for us. We will put this component in a new file *app/src/components/login-form/login-form-component.ts* and create an html template file for it as well.

```
export class LoginFormComponent {

  static selector = 'ngcLoginForm';

  static directiveFactory: ng.IDirectiveFactory = () => {
    return {
      restrict: 'E',
      scope: {},
      controllerAs: 'ctrl',
      bindToController: {
        fireSubmit: '&onSubmit'
      },
      controller: LoginFormComponent,
      template: require('./login-form-component.html')
    };
  };

  private username;
  private password;
  private fireSubmit: Function;

  constructor() {
    //
  }

  public submit() {
    this.fireSubmit({
      data: this
    });
  }
}
```

```
<div>
  <form>
    <h1>ngCourse App</h1>

    <label>Enter username</label>
    <input
      type="text"
      ng-model="ctrl.username">

    <label>Password</label>
    <input
      type="password"
      ng-model="ctrl.password">

    <button
      type="submit"
      ng-click="ctrl.submit()">
      Login
    </button>
  </form>
</div>
```

Note the use of

```
bindToController: {
  fireSubmit: '&onSubmit'
}
```

This is how we will pass data out of the component, through events.

And change our wiring in `app.ts`

```
...
.directive(
  LoginFormComponent.selector,
  LoginFormComponent.directiveFactory)
...
```

Let's change *main-component.ts* and its template to accomodate this change:

```
export class MainComponent {

  static selector = 'ngcMain';

  static directiveFactory: ng.IDirectiveFactory = () => {
    return {
      transclude: true,
      restrict: 'E',
      scope: {},
      controllerAs: 'ctrl',
      bindToController: true,
      controller: MainComponent,
      template: require('./main-component.html')
    };
  };

  private isAuthenticated;
  private username;

  constructor(private $log) {
    this.isAuthenticated = false;
  }

  public login(data) {
    this.username = data.username;
    this.isAuthenticated = true;
  }

};
```

```
<div>
  <div ng-hide="ctrl.isAuthenticated">
    <ngc-login-form on-submit="ctrl.login(data)"></ngc-login-form>
  </div>
  <div ng-show="ctrl.isAuthenticated">
    <ngc-tasks username="ctrl.username"></ngc-tasks>
  </div>
</div>
```

Passing Data Between Components Summary

In the above sections we have seen 2 ways to pass data between components using the `bindToController` options.

`=` to pass variables from the current component into the component.

`&` to register a callback for component to invoke (with data if necessary) in order to pass data out of the component.

`=` and `&` are the mechanism that allow our component to have a "public API".

Note, if the attribute name and the property of the component class match the name can be omitted. i.e instead of `username: '=username'` we can just write `username: '='`, with the same shortcut applying to `&`.

Iteration with `ng-repeat`

When we have a list of items, we can use `ng-repeat` directive within our component's template to create identical DOM element for each item.

Let's modify the temaplate in *task-list-component.ts*

```
<div>
  <span>
    Hello, {{ ctrl.username }}!
    You've got {{ ctrl.tasks.length }} tasks.
  </span>
  <button ng-click="ctrl.addTask()">Add task</button>
</div>

<div>
  <div ng-repeat="task in ctrl.tasks" >
    <p>{{ task.owner }} | {{ task.description }}</p>
  </div>
</div>
```

In the TaskListComponent all we do is set `tasks` to an array:

```
...
  export class TaskListComponent {
    ...
    private tasks;

    constructor(private $log) {
      this.tasks = [
        {
          owner: 'alice',
          description: 'Build the dog shed.'
        },
        {
          owner: 'bob',
          description: 'Get the milk.'
        },
        {
          owner: 'alice',
          description: 'Fix the door handle.'
        }
      ];
    }

    public addTask() {
      this.$log.debug('Current number of tasks:', this.tasks.length);
    }
  };
```

Note that in the template of this component we also change `{{ ctrl.numberOfTasks }}` to `{{ ctrl.tasks.length }}`.

Structuring Applications with Components

For the sake of a simple application our `TaskListComponent` class is fine, but as the complexity and size of our application grow we want to divide responsibilities among our components further.

How should we divide responsibilities between our components? Let's start with our task list example above.

`TaskListComponent` will be responsible with retrieving and maintaining the list of tasks from the domain model. It should be able to retrieve the tasks, and it should be able to add a new task to the domain model (abstracted our in later sections).

`TaskComponent` will be responsible for a single task and displaying just that task interacting with it's parent through it's "public API"

With the above in mind, let's create the `TaskComponent` class.

```
...
export class TaskComponent {

  static selector = 'ngcTask';

  static directiveFactory: ng.IDirectiveFactory = () => {
    return {
      restrict: 'E',
      scope: {},
      controllerAs: 'ctrl',
      bindToController: {
        task: '='
      },
      controller: TaskComponent,
      template: require('./task-component.html')
    };
  };

  private task;
  constructor(private $log) {

  }
};
```

and its corresponding template

```
<p>{{ ctrl.task.owner }} | {{ ctrl.task.description }}</p>
```

What is left is to modify our *task-list-component.html*

```
<div>
  <span>
    Hello, {{ ctrl.username }}!
    You've got {{ ctrl.tasks.length }} tasks.
  </span>
  <button ng-click="ctrl.addTask()">Add task</button>
</div>

<div>
  <ngc-task ng-repeat="task in ctrl.tasks"
      task="task">
  </ngc-task>
</div>
```

The refactoring above illustrates and important categorization between components, as it allows us to think of components in the following ways.

Macro Components: which are application specific, higher-level, container components, with access to the application's domain model.

Micro Components: which are components responsible for UI rendering and/or behaviour of specific entities passed in via components API (i.e component properties and events). Those components are more inline with the upcoming Web Component standards.

Introduction to Unit Testing

The Rationale

Why bother with unit tests?

Unit Tests vs Integration Tests

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 behaviour of a specific unit of code. If the unit's behaviour is modified, then the unit test would be updated as well. Unit tests should not make assumptions about the behaviour of *other* parts of your codebase or your dependencies. When other parts of your codebase are modified, your unit tests **should not fail**. (If they do fail, you have written a test that relies on other components, it is therefore not a unit test.) Unit tests are cheap to maintain, and should only be updated when the individual units are modified.

The Toolchain

Let's talk about some of the available tools. Our preferred toolchain consists of:

- Mocha - the actual test framework.
- Chai - an assertion library.
- Sinon - a spy library.
- Karma - a test "runner".
- Gulp - a task automation tool.

Why Mocha?

While we see this as the best combination of tools, a common alternative is Jasmine, a somewhat older tool that combines features of Mocha, Chai and Sinon.

Mocha provides better support for asynchronous testing by adding support for the `done()` function. If you use it, your test doesn't pass until the `done()` function is called. This is a nice to have when testing asynchronous code. Mocha also allows for use of any assertion library that throws exceptions on failure, such as Chai.

A Basic Mocha Test

First, let's write a simple test and run it. Put this code into `app/src/simple.test.js`.

```
describe('Simple Test', () => {
  it('2*2 should equal 4', () => {
    let x;
    // Do something.
    x = 2 * 2;
    // Check that the results are what we expect and throw an error if something is off
    if (x!==4) {
      throw new Error('Failure of basic arithmetics.');
    }
  });
});
```

Running Mocha Tests with Karma

We can now run this code from the command line using karma (more in subsequent chapters):

```
karma start
```

This will run **all** tests under client.

Karma configuration is in `karma.conf.js`.

The Importance of Test Documentation

The first argument to `it()` should explain what your test aims to verify. Beyond that, consider adding additional information through comments. Well-documented tests can serve as documentation for your code and can simplify maintenance.

Mocha with Chai

Chai is an assertion library. It makes it easy to throw errors when things are not as we expect them to be. Chai has two styles: "TDD" and "BDD". We'll be using the "BDD" style.

We have already installed Chai when we ran `npm install` and it has also been configured in the karma config file. So, now we can go straight to using it.

```
describe('Simple Test', () => {
  it('2*2 should equal 4', () => {
    let x = 2 * 2;
    let y = 4;
    // Assert that x is defined.
    chai.expect(x).to.not.be.undefined;
    // Assert that x equals to specific value.
    chai.expect(x).to.equal(4);
    // Assert that x equals to y.
    chai.expect(x).to.equal(y);
    // See http://chaijs.com/api/bdd/ for more assertion options.
  });
});
```

Unit Testing Simple Components

Let's see how we can apply what we learned so far and unit test our `TaskComponent`. Create a new file *app/src/components/task/task-component.test.ts*, and copy the following unit-test code.

```
import 'angular';
import 'angular-mocks';

import {TaskComponent} from './task-component';

describe('TaskComponent', () => {

  let _$scope;
  let _$compile;

  angular.module('tasks', [])
    .directive(
      TaskComponent.selector,
      TaskComponent.directiveFactory);

  beforeEach(() => {
    angular.mock.module('tasks');
    angular.mock.inject(($compile, $rootScope) => {
      _$scope = $rootScope.$new();
      _$compile = $compile;
    });
  });

  it('generate the appropriate HTML', () => {
    _$scope.task = {
      owner: 'alice',
      description: 'Fix the door handle.',
      done: true
    };

    _$scope.user = {
      displayName: 'Alice'
    };

    let element = angular.element(
      `<ngc-task
        task="task"
        user="user">
      </ngc-task>`);

    let compiled = _$compile(element)(_$scope);

    _$scope.$digest();

    chai.expect(compiled.html()).to.contain('Fix the door handle.');
  });
});
```

There's a lot going on here, so let's break it down a bit.

Before Each

This is simple, as the name implies, this block defines a set of operations to be run before each test defined using the `it` block.

Inject

Also, we're using `inject()` to get access to two Angular services:

- $compile - used for evaluating the directive's template HTML.
- $rootScope - used to create an isolated $scope object for passing test data into the directive. Each time a new test is run (defined using it), a new isolated scope for our component will be created using `$rootScope.$new()`.

The Test

In the test itself, we:

1. Put the mock task data to on the scope to be passed into our component in the next step.
2. Create the directive using its HTML form, the way it would be used in real code.
3. Tell Angular to evaluate the template HTML based on the $scope we constructed.
4. Manually trigger Angular's digest cycle, which causes any angular expressions (`\{\{ }}` blocks) in the directive's template HTML to be evaluated.
5. Verify that the expected markup was generated by the directive.

Making HTTP Requests.

Most AngularJS apps will need to interact with a backend system to achieve their objectives. By far the most common protocol for this is HTTP (or, preferably, it's secure version, HTTPS). While this is the same protocol that was employed by the previous generation of web applications, it tends to be used rather differently in a context of a modern single-page application framework such as AngularJS.

Talking to a RESTful Server via Postman

In most cases, the Angular application and the server will not be exchanging HTML. Instead, they will be exchanging data using an approach that is usually referred to as "RESTful". The key idea is that each URL is normally understood to represent a data "resource" in some collection of resources. The application can use HTTP to create a resource, retrieve an existing one, update it, or delete it.

Before we begin talking to the server from our web application, let's get comfortable interacting with the REST API using Postman.

If you do not have Postman installed, get it here: http://http://www.getpostman.com/

Our server is setup at http://ngcourse.herokuapp.com/. Here is our `tasks` endpoint:

```
http://ngcourse.herokuapp.com/api/v1/tasks
```

HTTP Methods

An HTTP client interacts with server resources using HTTP methods:

GET to retrieve some resource (or resources). E.g.:

```
GET http://ngcourse.herokuapp.com/api/v1/tasks/5491e9da995e33455a7307e2
```

The server will respond with the a set of objects representing matching resources:

```
[{
  "_id":"5491e9da995e33455a7307e2",
  "owner":"alice",
  "description":"Create the horse shed.",
  "__v":0
}]
```

GET requests often also include a query, encoded as a query string.

POST to create a new resource.

```
POST http://ngcourse.herokuapp.com/api/v1/tasks/ + data
```

PUT to update an existing resources.

```
PUT http://ngcourse.herokuapp.com/api/v1/tasks/5491e9da995e33455a7307e2 + data
```

DELETE to delete an existing resource.

```
DELETE http://ngcourse.herokuapp.com/api/v1/tasks/5491e9da995e33455a7307e2
```

Content Type

RESTful APIs can exchange data using a variety of encoding formats. By far the most common format today is JSON. In a typical Angular application, this is the format used by the server for sending data in response to a GET request and by the client when sending data with a POST or PUT request. Even though JSON is quite common, the client and the server are supposed to notify each other that this is the format being used by using header. For JSON, the proper content-type is:

```
Content-Type: application/json
```

for compatibility reasons, however, many servers will specify the type as:

```
Content-Type: text/plain
```

The good news, however, is that Angular will assume that you are using JSON and will handle the details for you.

Status Codes

An HTTP response must contain a numeric status code that indicates whether the request was handled successfully. Legacy web applications often use such codes in a sloppy way, responding to all requests with the same code "200 Success." REST APIs usually approach status codes more seriously. This allows your application to rely on them to determine what happened and how to proceed.

There are many status codes in use and not all servers use them in the same way. The most important thing is to properly handle codes in each of the distinct *ranges* show below:

200-299: All codes in this range signify that the server did what it thought you asked it to do. The most common success codes are "200 Success" and "201 Created" (usually sent in response to a successful POST request). Some servers will just use "200" for all success responses.

400-499: All codes in this range mean that the server didn't do what you asked because it found some flaw with your request. This normally indicates that *you* (or the user of the application) are doing something wrong. The most common 400-level codes are "400 Bad Request" (generic), "401 Unauthorized" (the client needs to be authenticated), "403 Forbidden" (you are not authorized to do what you are asking to do), and "404 Not Found" (the resource you are asking for does not exist, or you are not allowed to know whether it exists). Some servers will send additional 400-level status code for certain specific problems, while others will fall back on "400" if none of the options above apply. It is important for your application to identify the different 400-level codes used by your server, since they may help you determine what to do next. For example, a 401 response would normally require that you ask the user to re-login, while a 403 response would usually mean telling the user that they aren't allowed to do whatever they were trying to do.

500-599: These codes indicate *server* errors. The server cannot do what you asked because something is wrong on the server. Normally, this means that it's *not* your application's fault and there is little you can do other than try again later (or get in touch with the administrators for the server). There are many specific 500-level status codes, but from your application's perspective it usually doesn't matter what exactly happened. You do, however, want to handle such errors somehow, perhaps by showing an error message saying "Sorry, we are having trouble talking to the server. Let's try again in a little bit."

CORS

Most browsers today follow a "same-origin policy", which stops your web application from making HTTP requests to servers other than the one from which the application itself was loaded. In our case right now, the web app is being loaded from localhost, so under this policy it would only be allowed to send HTTP requests to localhost. While this made sense in the past, in most modern single-page applications this is impractical. Usually, your app will need to talk to a server on some other host. (In our case, it's "ngcourse.herokuapp.com.") When you try to do this, however, you may get an error that would look roughly like this:

> XMLHttpRequest cannot load [some URL]. Origin [some domain] is not allowed by Access-Control-Allow-Origin.

There are two solutions to this problem.

CORS. The correct solution is for your server to implement "CORS", which is a mechanism for telling your web browser that it's ok for your application to talk to this server. It's a fairly simple thing to enable with most modern server technologies, and once it's done on the server, there is nothing more for you to do on the client if you are using AngularJS and a modern web browser.

Disable web security. The second option is to disable cross-origin security policy (among other things) in the browser. This makes sense if same-origin policy is not going to be a problem in deployment and the server is not going to support CORS. If using Google Chrome, you can achieve this by running it with `--disable-web-security` flag. For example, on OSX:

```
open -a /Applications/Google\ Chrome.app --args --disable-web-security
```

Or on Windows:

```
chrome.exe --disable-web-security
```

Same-origin policy does *not* apply to apps built with Cordova. However, you will run into this option when testing those apps in a web browser.

$http

Now we are ready to start making HTTP calls from our Angular app.

While AngularJS provides a service called `$resource` for interacting with RESTful APIs, we won't be using it, because use of `$resource` often leads to an "optimistic" approach to asynchronous code, which results in bugs that are hard to track. Instead, we will be using a more basic Angular server, `$http`.

Let's start by just getting a list of tasks:

```
...
export class TaskListComponent {

  ...

  static $inject = ['$log', '$http'];

  constructor(private $log, private $http) {

    this.$http.get('http://ngcourse.herokuapp.com/api/v1/tasks')
      .success((data, status) => {
        this.$log.info(data);
        this.tasks = data;
      })
      .error((data, status) => this.$log.error(status, data));
  }

  public addTask(task) {

  }
};
```

Here we are making an HTTP GET request and providing two callbacks: one to handle successful responses (200-level status codes or redirects that eventually lead to a 200-level status code) and another to handle errors (400- and 500-level status codes).

When the callbacks are called, they will be passed the data, expanded into a full JavaScript object or array and the specific status code.

We'll focus on a somewhat different different approach, though:

```
...
  this.$http.get('http://ngcourse.herokuapp.com/api/v1/tasks')
    .then(response => {
      this.$log.info(response.data);
      this.tasks = response.data;
    })
    .then(null,
      error => this.$log.error(status, error));
...
```

This takes advantage of the fact that `$http.get()` returns an object that can act as a standard "promise".

In addition to `$http.get()`, there is also `$http.post()`, `$http.put()`, `$http.delete()`, etc. They all return promises. To get more mileage out of those methods, we'll have to spend some time looking closely at promises.

Introduction to Promises.

In the previous section we saw that `$http` methods give us promises. But what exactly is a promise?

Promises vs Callbacks

JavaScript is single threaded, so we can't really ever "wait" for a result of a task such as an HTTP request. Our baseline solution is callbacks:

```
request(url, (error, response) => {
  // handle success or error.
});
doSomethingElse();
```

A few problems with that. One is the "Pyramid of Doom":

```
queryTheDatabase(query, (error, result) => {
  request(url, (error, response) => {
    doSomethingElse(response, (error, result) => {
      doAnotherThing(result, (error, result) => {
        request(anotherUrl, (error, response) => {
          ...
        })
      });
    })
  });
});
```

And this is without any error handling! A larger problem, though: hard to decompose.

The essence of the problem is that this pattern requires us to specify the task and the callback at the same time. In contrast, promises allow us to specify and dispatch the request in one place:

```
promise = $http.get(url);
```

and then to add the callback later, and in a different place:

```
promise.then(response => {
  // handle the response.
});
```

This also allows us to attach multiple handlers to the same task:

```
promise.then(response => {
  // handle the response.
});
promise.then(response => {
  // do something else with the response.
});
```

Unchaining Promises

You might have seen chained promises:

```
$http.get('http://ngcourse.herokuapp.com/api/v1/tasks')
  .then(response => response.data)
  .then(tasks => {
    $log.info(tasks);
    vm.tasks = tasks;
  })
  .then(null, error => $log.error(error));
```

We could also make this more complicated:

```
$http.get('http://ngcourse.herokuapp.com/api/v1/tasks')
  .then(response => {
    let tasks = response.data;
    return filterTasks(tasks);
  })
  .then(tasks => {
    $log.info(tasks);
    vm.tasks = tasks;
  })
  .then(null, error => $log.error(error));
```

Or even:

```
$http.get('http://ngcourse.herokuapp.com/api/v1/tasks')
  .then(response => response.data)
  .then(tasks => filterTasksAsynchronously(tasks))
  .then(tasks => {
    $log.info(tasks);
    vm.tasks = tasks;
  })
  .then(null, error => $log.error(error));
```

To make sense, let's "unchain" this using variables:

```
let responsePromise = $http.get('http://ngcourse.herokuapp.com/api/v1/tasks');
let tasksPromise = responsePromise.then(
  response => response.data);

let filteredTasksPromise = tasksPromise.then(
  tasks => filterTasksAsynchronously(tasks));

let vmUpdatePromise = filteredTasksPromise.then(tasks => {
  $log.info(tasks);
  vm.tasks = tasks;
})

let errorHandlerPromise = vmUpdatePromise.then(
  null, error => $log.error(error));
```

Let's work through this example.

Promises Beget Promises (via `.then()`)

A key point to remember: unless your promise library is buggy, `.then()` always returns a promise. Always.

```
p1 = getDataAsync(query);

p2 = p1.then(
  results => transformData(results));
```

`p2` is now a promise regardless of what transformData() returned. Even if something fails.

If the callback function returns a value, the promise resolves to that value:

```
p2 = p1.then(results => 1);
```

`p2` will resolve to “1”.

If the callback function returns a promise, the promise resolves to a functionally equivalent promise:

```
p2 = p1.then(results => {
  let newPromise = getSomePromise();
  return newPromise;
});
```

`p2` is now functionally equivalent to newPromise. It's not the same object, however. Let's discuss why not.

```
p2 = p1.then(
  results => throw Error('Oops'));

p2.then(results => {
  // You will be wondering why this is never
  // called.
});
```

`p2` is still a promise, but now it will be rejected with the thrown error.

Why won't the second callback ever be called?

Catching Rejections

So, catch rejections:

```
$http.get('http://ngcourse.herokuapp.com/api/v1/tasks')
  .then(response => response.data)
  .then(tasks => filterTasksAsynchronously(tasks))
  .then(
    tasks => {
      $log.info(tasks);
      vm.tasks = tasks;
    },
    error => $log.error(error));
```

What's the problem with this code?

So, the following is better.

```
$http.get('http://ngcourse.herokuapp.com/api/v1/tasks')
  .then(response => response.data)
  .then(tasks => filterTasksAsynchronously(tasks))
  .then(tasks => {
    $log.info(tasks);
    vm.tasks = tasks;
  })
  .then(
    null,
    error => log.error(error)
  );
```

Note that one catch at the end is often enough.

Using an Existing Function As a Handler

```
    .then(null, error => $log.error(error));
```

can be replaced with:

```
    .then(null, $log.error);
```

Let's make sure we understand why.

Returning Promises

There is one (common) case when it's ok to not catch the rejection:

```
  return $http.get('http://ngcourse.herokuapp.com/api/v1/tasks')
    .then(response => response.data);
```

That's passing the buck. But remember: the buck stops with the component's function that is triggered by Angular.

Catch and Release

Or you can catch, do something, and still pass the exception onwards:

```
  .then(null, error => {
    $log.error(error); // Log the error
    throw error; // Then re-throw it.
  });
```

Sometimes we may want to re-throw conditionally.

Promise Chains Considered Harmful

A better approach is to break them up into meaningful functions.

```
function getTasks() {
  return $http.get('http://ngcourse.herokuapp.com/api/v1/tasks')
    .then(response => response.data);
}

function getMyTasks() {
  return getTasks()
    .then(tasks => filterTasks(tasks, {owner: user.username}));
}

getMyTasks()
  .then(tasks => {
    $log.info(tasks);
    vm.tasks = tasks;
  })
  .then(null, $log.error);
```

Services

In the previous section we wrote functions that call `$http` methods and process resulting promises. We started by putting those functions in our controller. This, however, is a poor practice. Those functions are working with business logic and they should be kept out of controllers.

Services

Instead, we'll put those functions in an AngularJS service. Let's put this in *src/services/tasks/tasks-service.ts*

```
export class TasksService {

  static $inject = ['$log', '$http'];

  constructor(private $log, private $http) { }

  public getTasks () {
    return this.$http.get('http://ngcourse.herokuapp.com/api/v1/tasks')
      .then(response => response.data);
  };
}
```

Note we have added a new module definition and need to update *app.ts*.

```
  ...
  import {TasksService} from './services/tasks/tasks-service';
  ...
  angular.module('ngcourse', [])
    ...
    .service('tasksService', TasksService)

  ...
```

We can now simplify our code to use this service:

```
  ...
  export class TaskListComponent {
    ...
    static $inject = ['$log', 'tasksService'];

    constructor(private $log, private tasksService) {
        this.tasksService.getTasks()
          .then(tasks => this.tasks = tasks);
    }
```

Note that we've injected our newly created service.

Advantages of Keeping Code in Services

Let's discuss some of the advantages of keeping this type of code in services rather than in components.

The rule of thumb: code that can be written without referring to a controller's scope should be written this way and should be placed in a service.

More Services

When it comes to services, the more the better. Let's refactor some of the code from our `tasks` service into a new `server` service *app/src/services/server/server-service.ts*.

```
export class ServerService {

  private baseUrl = 'http://ngcourse.herokuapp.com';

  static $inject = ['$http'];

  constructor(private $http) { }

  public get(path) {
    return this.$http.get(this.baseUrl + path)
      .then(response => response.data);
  }
}
```

Again, let's add a new definition in *app.ts*.

```
...
import {ServerService} from './services/server/server-service';
...
angular.module('ngcourse', [])
  ...
  .service('server', ServerService);
...
```

While our `TaskService` code gets simplified to:

```
export class TasksService {

  static $inject = ['serverService'];
  constructor(private serverService) { }

  public getTasks () {
    return this.serverService.get('/api/v1/tasks');
  };
}
```

And we have a layered service architecture with the tasks service calling the server service.

But why bother, you might ask? Lets go over some of the benefits.

Using `.constant()` and `.value()`

We could decompose yet more, though:

```
  angular.module('ngcourse', [])
    ...
    .constant('API_BASE_URL', 'http://ngcourse.herokuapp.com')
    .service('serverService', ServerService);
```

and

```
export class ServerService {

  static $inject = ['$http', 'API_BASE_URL'];

  constructor(private $http, private API_BASE_URL) { }

  public get(path) {
    return this.$http.get(this.API_BASE_URL + path)
      .then(response => response.data);
  }
}
```

Alternatively, we can use `.value()` instead of `.constant()`. However, when in doubt, use `.constant()`.

Modules

At this point we may want to consider breaking our code up into modules. E.g., let's make `server` its own module:

```
angular.module('ngcourse.server', [])
  ...
  .constant('API_BASE_URL', 'http://ngcourse.herokuapp.com')
  .service('serverService', ServerService);
```

We can then make it a dependency in our `ngcourse` module (in `app.ts`):

```
angular.module('ngcourse', [
  'ngcourse.server'
]);
```

Note that an Angular "app" is basically just an Angular module.

Each module can define `.config()` and `.run()` sections. You will rarely see `.config()` except when setting up routes. (We'll discuss it in that context.) Your `.run()` is essentially you modules's equivalent of the "main" block.

```
angular.module('ngcourse', [
  'ngcourse.server'
])

.run($log => $log.info('All ready!'));
```

Keep in mind, though, that Angular's modules are somewhat of a fiction.

More Promises

Now that we've got our code better organized using services, we can look at some of the more interesting things we can do with promises.

Reusing Promises

```
...
export class TasksService {

  private taskPromise;
  static $inject = ['serverService'];
  constructor(private serverService) { }

  public getTasks () {

    this.taskPromise = this.taskPromise ||
      this.serverService.get('/api/v1/tasks');
    return taskPromise;
  };
}
```

Postponing the Requests

```
...
export class ServerService {

  private baseUrl = 'http://ngcourse.herokuapp.com';

  static $inject = ['$http', 'userService'];
  constructor(
    private $http,
    private userService) { }

  public get(path) {
    return this.userService.whenAuthenticated()
      .then(() => this.$http.get(API_BASE_URL + path))
      .then(response => response.data);
  }
}
```

A Brand New Promise

Suppose we have a function that uses callbacks. We have a few options for turning it into a function that returns a promise.

The first approach is to use `$q.defer()`:

```
function getFooPromise(param) {
  let deferred = $q.defer();
  getFooWithCallbacks(param, (error, result) => {
    if (error) {
      deferred.reject(error);
    } else {
      deferred.resolve(result);
    }
  });
  return deferred.promise;
}
```

An alternative is to use the ES6-style constructor:

```
function getFooPromise(name) {
  return $q((resolve, reject) => {
    getFooWithCallbacks(param, (error, result) => {
      if (error) {
        reject(error);
      } else {
        resolve(result);
      }
    });
  });
}
```

However, beware of a very common JavaScript anti-pattern that involves unnecessary creation of new promises. This is often called the "Deferred Anti-pattern", but it applies equally to the ES6-style promise constructor. In most cases, you do not need to create new promises from scratch and resolve them yourself. If you got a promise from another function, you should use that promise's `.then()` method to create further promises. It is almost never a good idea to create a manually resolve a promise inside of a promise callback.

Converting a callback-style function to one returning a promise is about the only valid case for using a promise constructor. Even in this case, however, the result can be better achieved using a dedicated conversion function. For example, if you have a function that

relies on Node-style callbacks as in the example above above, you can convert it using angular-promisify like so:

```
let getFooPromise = denodeify(getFooWithCallbacks);
```

While manually resolving promises is rarely a good idea, `$q` offers two promise-creation methods that often come in very handy. You can use `$q.when(value)` to create a promise that resolves to `value` and `$q.reject(error)` to create a promise that rejects with `error`. Use those methods when you want to avoid calling a function that would have given you a promise.

```
function get(path) {
  if (!networkInformation.isOnline) {
    return $q.reject('offline');
  } else {
    return userService.whenAuthenticated()
      .then(() => $http.get(API_BASE_URL + path))
      .then(response => response.data);
  }
};
```

Or, better yet:

```
function waitForPreconditions() {
  if (!networkInformation.isOnline) {
    return $q.reject('offline');
  } else {
    return userService.whenAuthenticated();
  }
}

function get(path) {
  return waitForPreconditions()
    .then(() => $http.get(API_BASE_URL + path))
    .then(response => response.data);
  }
};
```

Promises from `$timeout()`

Angular's `$timeout()` service also returns a promise. The same is true for a number of other AngularJS functions.

Promises vs Events

While promises are nearly always better than Node- style callbacks, the choice of when to use promises vs a publish-subscribe approach is a bit more complex. Here are the key differences between the two approaches:

Promises	Events (aka “Publish – Subscribe”)
Things that happen ONCE	Things that happen MANY TIMES
Same treatment for past and future	Only care about the future*
Easily matched with requests	Detached from requests

Promises are often the best approach for handling responses to an explicit request, such as an HTTP call. Publish-subscribe often works better for handling actions initiated by the user (except with modals).

Next Steps

We now know most of what we need to know about services. Before we write more code, however, we need to get setup for unit testing.

Unit Testing Services

Testing Services

Our tests are not really "unit tests" if they make use of many layers of dependencies - and especially if they make server calls. So, we need to "mock" our dependencies.

Since our services are classes with all of their dependencies injected through their constructors, testing them is simple. We just need to mock out all of their dependencies and pass them to our service class.

Let's create some unit tests for our `TasksService` class, starting by mocking out it's only dependency, `ServerService`.

```
import {TasksService} from './tasks-service';

describe('TasksService', () => {

  let _mockServerService;

  let _mockTasks = [{
    owner: 'alice',
    description: 'Build the dog shed.',
    done: true
  }, {
    owner: 'bob',
    description: 'Get the milk.',
    done: false
  }, {
    owner: 'alice',
    description: 'Fix the door handle.',
    done: true
  }];

  beforeEach(() => {
    _mockServerService = {
      get: () => Promise.resolve(_mockTasks)
    };
  });

  it('should get loaded', function() {
    let tasksService = new TasksService(_mockServerService);
    chai.expect(tasksService.getTasks()).to.not.be.undefined;
  });
});
```

Let's save this to `app/src/services/tasks/tasks-service.test.js`.

The sad truth, though, is that we have only established that `tasksService.getTasks()` does return a promise. We can't really judge the success of this test until we know what that promise resolves to.

An Asynchronous Test

So, we want to check what the promise resolves too, but this only will happen *later*. So, we need to use an asynchronous test that would wait for the promise to resolve.

```
...
beforeEach(() => {
  _mockServerService = {
    get: () => Promise.resolve(_mockTasks)
  };
});

it('should get loaded', function() {
  let tasksService = new TasksService(_mockServerService);
  chai.expect(tasksService.getTasks()).to.not.be.undefined;
});

it('should get tasks', (done) => {
  // Notice that we've specified that our function takes a 'done' argument.
  // This tells Mocha this is an asynchronous test. An asynchronous test will
  // not be considered 'successful' until done() is called without any
  // arguments. If we call done() with an argument the test fails, treating
  // that argument as an error.
  let tasksService = new TasksService(_mockServerService);

  return tasksService.getTasks()
    .then(tasks => {
      // Assertions thrown here will result to a failed promise downstream.
      expect(tasks).to.deep.equal(_mockTasks);
      done();
    })
    // Remember to call done(), otherwise the test will time out (and fail).
    .then(null, error => {
      done(error);
    });
});
...
```

Note the use of `deep.equal` assertion to check that the tasks received by calling `TasksService.getTasks()` method. Let's run.

A Simplified Use of done()

If all we want to do in case of error is to pass it to done, we don't actually need to define a new function in the handler. We can just provide `done` as the handler.

```
    .then(null, error => done(error));
```

is equivalent to:

```
    .then(null, done);
```

Mocha's Support for Promises

Mocha's tests can alternatively just accept a promise. In most case this is what you want to use.

```
...
it('should get tasks', () => {
  let tasksService = new TasksService(_mockServerService);
  return tasksService.getTasks()
    .then(tasks => chai.expect(tasks).to.deep.equal(_mockTasks));
});
...
```

Spying with Sinon

A test spy is a function that records arguments, return value, the value of `this`, and exception thrown (if any) for all its calls. A test spy can be an anonymous function or it can wrap an existing function. When using Sinon, we'll wrap the existing function with `sinon.spy()`:

```
...
beforeEach(() => {
  _mockServerService = {
    get: sinon.spy(() => Promise.resolve(_mockTasks))
  };
  _mockServerService.get.reset();
});
...
```

When spying on existing functions, the original function will behave as normal, but we will be proxied through the spy, which will collect information about the calls. For example, we can check if the function has been called:

```
...
it('should only call server service get once', () => {
  let tasksService = new TasksService(_mockServerService);
  return tasksService.getTasks() // Call getTasks the first time.
    .then(() => tasksService.getTasks())
    .then(() => chai.expect(_mockServerService.get.calledOnce).to.be.true);
});
...
```

Note that here we created a new test to verify that `serverService.get` is only getting called once. In between each test we are resetting the data gathered by the spy with `_mockServerService.get.reset()`.

Finally, we do not attempt to verify in this test that the promise returned by `getTasks()` actually resolves to the value we expect, since this is already being verified by another test. Keeping tests small and focused greatly facilitates test maintenance.

Refactor 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 so as to make it easy to test. For example, consider refactoring your component code into services and focusing on service tests.

Reactive Programming with RxJs.

In chapters 9 and 11, we have seen on how to deal with asynchronicity within your application by using callbacks and promises, this chapter will focus on the concept of Observable that provide a paradigm for dealing with asynchronous data streams.

What is Reactive Programming

Reactive programming is programming with asynchronous data streams represented by Observables. The concept here is a mix of Observer and Iterable design patterns. Specifically, observable can be conceptualized as an immutable collection of data ordered in time, and iterated over similarly to collections such as arrays or lists.

Creating an Observable from Scratch

First let's jump into a basic example to illustrate the concepts behinds RxJS observables.

```
let source = Rx.Observable.create(observer => {
  setTimeout(() => {
    observer.onNext(42);
    observer.onCompleted();
  }, 2000);

  console.log('Starting Observable Sequence!');
});

let subscription = source.subscribe(
  value => console.log('Value: ' + value),
  error => console.log(error),
  () => console.log('Completed Observable Sequence!')
);
```

First, we create an observable sequence `source`. This sequence emits a single value asynchronously using `setTimeout()` and then completes.

The second part of the code subscribes to the observable sequence `source`, and provides an *Observer* represented by the 3 callbacks provided. Those callbacks are:

1. `onNext`: represents a function to be invoked when a new value is emitted onto an observable sequence `source`.
2. `onError`: represents a function to be invoked if an error occurs within an observable sequence.
3. `onComplete`: represents a function to be invoked when the observable sequence completes.

Handling Errors

So far we created a basic observable example and attached the `onError` callback. Let's see how it can be put to better use.

```
let source = Rx.Observable.create(observer => {
  setTimeout(() => {
    try {
      //throw 'My Error';
      observer.onNext(42);
      observer.onCompleted();
    }
    catch (error) {
      observer.onError(error);
    }
  }, 2000);

  console.log('Starting Observable Sequence!');
});

let subscription = source.subscribe(
  value => console.log('Value: ' + value),
  error => console.log(error),
  () => console.log('Completed Observable Sequence!')
);
```

Running the above example without the `try...catch` and throwing an error will produce an uncaught error. Adding the try catch and handling the error by using the `onError` method allows us to emit the error via an observable and propagate it properly to the observer end.

Disposing Subscriptions

In some cases we might want to unsubscribe early from our observable. To achieve that we need to dispose of our subscription. Luckily, our call to `subscribe` on our observable actually returns an instance of a `Disposable`. This allows us to call `dispose` on our subscription should we decide to stop listening.

When we call `dispose` method on our subscription, our observer will stop listening to observable for data. In addition, we can return a function within our observable's implementation (i.e. `create` method above) that will be invoked when we call the `dispose` method on our subscription. This is useful for any kind of clean-up that might be required.

Let's modify our example and see how it works out.

```
let source = Rx.Observable.create(observer => {
  setTimeout(() => {
    try {

      console.log('In Timeout!');

      observer.onNext(42);
      observer.onCompleted();
    }
    catch (error) {
      observer.onError(error);
    }
  }, 2000);

  console.log('Starting Observable Sequence!');

  return onDispose = () => console.log('Stoping to Listen to this Observable Sequence')

});

let subscription = source.subscribe(
  value => console.log('Value: ' + value),
  error => console.log(error),
  () => console.log('Completed Observable Sequence!')
);

setTimeout(() => subscription.dispose(), 1000);
```

The last line of the code above, calls `dispose` method on our subscription after 1000ms (our observable is supposed to emit after 2000ms). If you run this example you will notice that this observable did not emit any values since we have called `dispose` method.

In most of the cases we will not need to explicitly call the `dispose` method on our subscription unless we want to cancel early or our observable has a longer life span than our subscription. The default behaviour of an observable operators is to dispose of the subscription as soon as `onCompleted` or `onError` messages are published. Keep in mind that RxJS was designed to be used in a "fire and forget" fashion most of the time.

Releasing Resources

Note, however that our log statement within `setTimeout` was still called. This implies that even though our subscription was disposed of, the code within the `setTimeout` was still executed. All that we achieved is that no values were emitted onto for the observer to see, but our code block was still put on the even queue to be executed. In other words we did not release our resources properly which is the main use of the function we are returning in our `create` block. We are half way there.

The correct implementation in this case would be to cancel the timeout instead, like so.

```
let source = Rx.Observable.create(observer => {
  let timeoutId = setTimeout(() => {
    try {

      console.log('In Timeout!');

      observer.onNext(42);
      observer.onCompleted();
    }
    catch (error) {
      observer.onError(error);
    }
  }, 2000);

  console.log('Starting Observable Sequence!');

  return onDispose = () => {
    console.log('Releasing Resources of this Observable Sequence');
    clearTimeout(timeoutId);
  };

});

let subscription = source.subscribe(
  value => console.log('Value: ' + value),
  error => console.log(error),
  () => console.log('Completed Observable Sequence!')
);

setTimeout(() => subscription.dispose(), 1000);
```

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 behaviour.
- 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 in the case of promises, the caller must have access to the original function that returned the promise in order to have a retry capability.

Creating Observable Sequences

In the example above we have been creating observables from scratch which is especially useful in understanding the anatomy of an observable.

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

Now that we got the anatomy and structure of observables understood, let's look at some of the many other ways to create observables.

`interval` and `take`

```
Rx.Observable.interval(1000).take(5).subscribe(
  element => console.info(element),
  error => console.info(error),
  () => console.info('I am done!')
);
```

The observable above will produce a value every 1000ms, only the first 5 values will be emitted due to the use of `take`, otherwise the sequence will emit values indefinitely. There are many more operators available within the API, such as `range`, `timer` etc.

fromArray

```
Rx.Observable.fromArray([1, 2, 3]).subscribe(
  element => console.info(element),
  error => console.info(error),
  () => console.info('I am done!')
);
```

fromPromise

```
let promise = new Promise((resolve, reject) => resolve(42));

Rx.Observable.fromPromise(promise)
  .subscribe((value) => console.log(value));
```

`fromEvent`

```
Rx.Observable.fromEvent(document, 'click').subscribe(
  clickEvent => console.info(
    clickEvent.clientX + ', ' + clickEvent.clientY
  )
);
```

The first line in the example above creates an observable of mouse click events on our document, ordered in time. Another way to refer to an observable is to call it an asynchronous collection.

The point to take home from this, is that EVERYTHING can be made into a stream using observables.

Using Observables Array Style

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 RxJS a API. This is what bridges 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:

```
Rx.Observable.fromEvent(document, 'click')
  .filter(clickEvent: MouseEvent => clickEvent.altKey)
  .subscribe(clickEvent: MouseEvent => console.info(
    clickEvent.clientX + ', ' + clickEvent.clientY
  )
);
```

Note the chaining function style, and the optional static typing that comes with TypeScript we have used in this example.

Most Importantly functions like `filter` return an observable, as in *observables beget other observables*, similarly to promises.

Asynchronous Requests Using Observables

A lot of the time the asynchronous nature of our application will surface when dealing with UI events as illustrated in examples above, or making asynchronous server requests. Observables are well equipped to deal with both.

We have already seen the code below in Chapter 8 about REST APIs.

```
this.$http.get('http://ngcourse.herokuapp.com/api/v1/tasks')
  .then((response) => {
    this.$log.info(response.data);
    this.tasks = response.data;
  })
  .then(null,
    (error) => this.$log.error(status, error));
```

Let's see how we can make an asynchronous server call using an Observable instead.

```
let responseStream = Rx.Observable.create(observer => {
  $http.get('http://ngcourse.herokuapp.com/api/v1/tasks')
    .then(response => observer.onNext(response))
    .then(null, error => observer.onError(error));
});

responseStream.subscribe(
  (response) => $log.info('Data: ', response),
  (error) => $log.info('Error: ', error)
);
```

In the code snippet above we are creating a custom response data stream, and notifying the observers of the stream when the data arrived.

```
let responseStream = Rx.Observable.fromPromise(
  $http.get('http://ngcourse.herokuapp.com/api/v1/tasks'));

responseStream.subscribe(
  (response) => $log.info('Data: ', response),
  (error) => $log.info('Error: ', error)
);
```

Combining Streams with `flatMap`

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

We want to make something a bit more useful and attach our server request to a button click. How can that be done with streams? Let's observe the example below.

```
let eventStream =
  Rx.Observable.fromEvent(document.getElementById('refreshBtn'), 'click');

let responseStream = eventStream
  .flatMap(() => Rx.Observable.fromPromise(
    $http.get('http://ngcourse.herokuapp.com/api/v1/tasks')));

responseStream.subscribe(
  (response) => $log.info('Async Data: ', response),
  (error) => $log.info('Async Error: ', error)
);
```

First we create an observable of button click events on some button. Then we use the `flatMap` function to transform our event stream into our response stream.

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.

Alternatively, if we were to use `map` instead, we would create a meta stream, i.e. a stream of stream.

```
...
let metaStream = eventStream
  .map(() => Rx.Observable.fromPromise(
    $http.get('http://ngcourse.herokuapp.com/api/v1/tasks')));

// We would have to subscribe to each stream received below
// to achieve the behaviour we want
metaStream.subscribe(
  (stream) => $log.info('Async Data: ', stream),
  (error) => $log.info('Async Error: ', error)
);
```

This is not very useful in our current example as we would have to subscribe to an observable received from an observable stream.

View Example

Cold vs. Hot Observables

Observables in RxJS can be classified into 2 main groups, Hot and Cold Observables. Let's start with a cold observables

View Example

```
let source = Rx.Observable.interval(1000).take(7);

setTimeout(() => {
  source.subscribe(
    value => console.log('subscription A: ' + value));
}, 0);

setTimeout(() => {
  source.subscribe(
    value => console.log('   subscription B: ' + value));
}, 2000);
```

In the above case subscriber B subscribes 2000ms after subscriber A. Yet subscriber B is starting to get the value from 0 to 6 just like subscriber A only time shifted. This behaviour is referred to as a **Cold Observable**. A useful analogy is watching a pre-recorded video, let's say 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 you have to start watching the performance from where it is.

We have already encountered both kind of observables, the example above is a cold observable, while an example that uses `fromEvent` on our mouse clicks is a hot observable.

Converting from Cold to Hot Observables

A useful method within RxJS API, is the `publish` method. This method takes in a cold observable as it's 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.

```
let source =
    Rx.Observable.interval(1000).take(7).publish();

setTimeout(() => {
  source.connect();
}, 1000);

setTimeout(() => {
  source.subscribe(
    value => console.log('subscription A: ' + value));
}, 0);

setTimeout(() => {
  source.subscribe(
    value => console.log('   subscription B: ' + value));
}, 5000);
```

In the case above, the live performance starts at 1000ms, subscriber A arrived to the concert hall 1000ms early to get a good seat, and our subscriber B arrived to the performance 4000ms late and missed a bunch of songs.

Another useful method to work with hot observables instead of `connect` is `refCount`. This is auto connect method, that will start broadcasting as soon as there are more than one subscriber. Analogously, it will stop if the number of subscribers goes to 0, in other words no performance will happen if there is no one in the audience.

View Example

Summary

RxJS is a flexible set of APIs for composing and transforming asynchronous streams. It provides multitude of functions to create streams from absolutely anything, and more to manipulate and transform them..

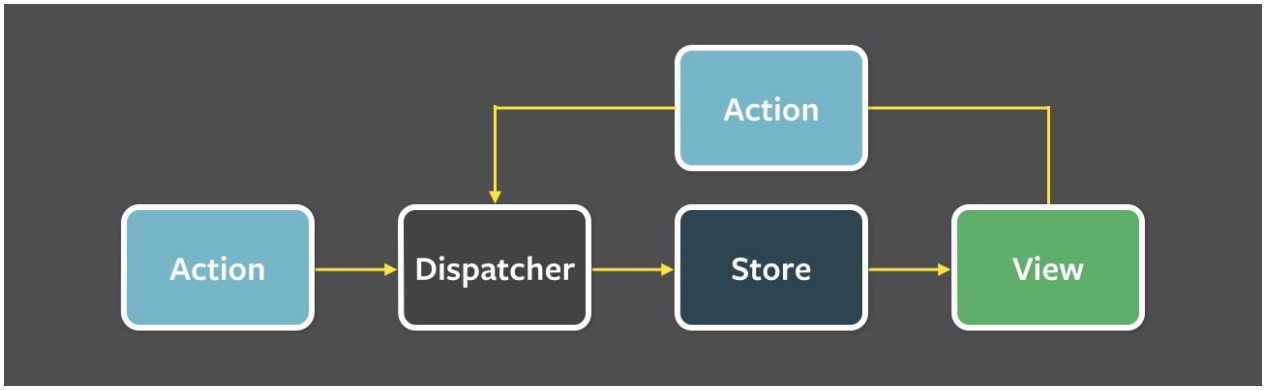
Flux Architecture

Flux is an architectural design pattern introduced and used by Facebook as an alternative to traditional MVC patterns to build modern web applications. Flux is not a library and implementing it within your application can be done without depending on any 3rd party code. Moreover Flux is language and framework agnostic and can be used outside single web-page application and JavaScript context. The core ideas behind Flux are inspired by game programming, specifically rendering of the changes of your domain model.

Core Concepts

The core idea behind Flux pattern, is unidirectional data flow implemented using the four major "machines" in the Flux "pipeline": Actions, Dispatcher, Stores and the Views.

The diagram below illustrates the flow of data through the Flux "pipeline".



When the user interacts with the View (i.e. a component), the view propagates an Action through a Dispatcher (i.e. a message bus) to the Stores where your domain model or state and the application logic are stored. Once the Store updates its domain model it propagates a "change" event to which Components in your application can subscribe, and re-render themselves accordingly to the new data.

Let's have a more in depth look at each of those parts in our "Flux machine".

Dispatcher

Dispatcher acts as a central message bus responsible for distributing the incoming Actions to the appropriate stores. It has no logic of its own and simply acts as a hub for Action creator methods to push Actions to. Store will listen on the dispatcher for incoming Actions responding to relevant ones.

Stores

As we mentioned above, Stores contain and manage your application's model and logic pertaining to a particular domain of your application. As a result an application can have many stores, each focused on a specific aspect of your domain. Stores are subscribed to the Dispatcher, our event bus, and listen to Actions that are relevant to them. The important part, is that nothing outside the store can modify the application state stored within it directly. Nothing outside the Store should know how the store manages its state, and the Store can be accessed in read-only fashion, i.e. by providing getters to the outside world.

In summary, the only way for a Store to modify the domain model of your application is by responding to an action coming through a Dispatcher.

Actions

Actions are a simple objects representing an action. They usually contain an action name and an optional payload. Most of the time actions are emitted from the views.

Views

Our views are our components

Implementing Dispatcher and Actions

Our dispatcher is very simple, let's modify our *app/src/app.ts* file

```
angular.module('ngcourse.dispatcher', [])
  .service('dispatcher', Rx.Subject);
```

In Chapter 13 - RxJS we have subscribed to Observables with Observers that implemented the `onNext`, `onError` and `onCompleted` methods. Rx.Subject is a combination of an Observer and Observable in one class. It is used here for convenience as you will see later.

Now, let add some actions to push onto our dispatcher later. Create a new file in *app/src/actions/task-actions.ts*

```
import {TASK_ACTIONS} from '../action-constants';

export class TaskActions {

  static $inject = ['dispatcher', 'tasksService'];
  constructor(
    private dispatcher: Rx.Subject<any>,
    private tasksService: TasksService
  ) { }

  getTasks() {
    this.dispatcher.onNext({
      actionType: TASK_ACTIONS.GET_TASKS_RESPONSE,
      tasks: []
    })
  }
}
```

It is a poor practice to use hard-coded strings like `'GET_TASKS'`. Therefore, we extracted our constants into a separate file, *app/src/actions/action-constants.ts* and are using those constants in *task-actions.ts* instead.

```
export const TASK_ACTIONS = {
  GET_TASKS_RESPONSE: 'GET_TASKS_RESPONSE',
  GET_TASKS_RESPONSE_ERROR: 'GET_TASKS_RESPONSE_ERROR'
};
```

Great! Now we can create our first store.

Implementing a Store

Now that we have a dispatcher and actions defined, lets start on our first Store. Let's create a new file for it in *app/src/stores/tasks/tasks-store.ts*

```
  export class TasksStore {

    private _tasks;

    static $inject = ['dispatcher'];
    constructor(private dispatcher) {

    }
  }
```

Our Store's will be responsible for providing the list of tasks that we will receive from the server. The Dispatcher is available to us from the `constructor` injection.

Before we do anything we need to listen to incoming Actions relevant to this store. Let's listen to incoming actions from our Dispatcher.

```
import {TASK_ACTIONS} from '../../actions/action-constants';

export class TasksStore {

  private _tasks;

  static $inject = ['dispatcher'];
  constructor(private dispatcher) {
      this.registerActionHandlers();
  }

  private registerActionHandlers() {
    this.dispatcher.filter(
      action => action.actionType === TASK_ACTIONS.GET_TASKS_RESPONSE)
        .subscribe(
          (tasks) => /* Handle action here */);
  }

  private getTasks() {
    // TODO
  }
}
```

For convenience we have created a new method on our Store called `registerActionHandlers()` where we will listen to incoming actions. Notice how we use `filter()` method to filter the actions that are relevant to this store, in this case we are only responding to `TASK_ACTIONS.GET_TASKS`.

Getting the tasks from the server

We start by modifying the `getTasks` method in the `TaskActions` service:

```
export class TaskActions {
  ...

  getTasks() {
    this.tasksService.getTasks()
      .then(tasks => this.dispatcher.onNext({
        actionType: TASK_ACTIONS.GET_TASKS_RESPONSE,
        tasks: tasks
      }))
      .then(null, error => this.dispatcher.onNext({
        actionType: TASK_ACTIONS.GET_TASKS_RESPONSE_ERROR,
        error: error
      }));
  }
}
```

Here we trigger the `TasksService` to fetch the tasks. And based on whether the request resolves/rejects we dispatch the appropriate action.

Also, we have added a call to `getTasks()` method within the run block for the app module. This will initialize our store with the data at the beginning of it's life-cycle.

```
angular.module('ngcourse', [...])
  ...
  .run((tasksActions) => {
    tasksActions.getTasks();
  });
```

Notifying Store Subscribers of State Change

So far we have managed to get the data we need from the server and make it available to the store. The next step is to publish this data to for our components. The components will also be listening to the changes to the store. We will implement this mechanism using what we learned about RxJS.

```
export class TasksStore {

  private _tasks: Rx.ReplaySubject<any[]>;
  private _error: Rx.ReplaySubject<any>;

  static $inject = ['dispatcher'];

  constructor(private dispatcher: Rx.Subject<any>) {

    this._tasks = new Rx.ReplaySubject<any[]>(1);
    this._error = new Rx.ReplaySubject(1);

    this.registerActionHandlers();
  }

  get tasks() {
    return this._tasks;
  }

  get error() {
    return this._error;
  }

  ...

}
```

Let go through this code step by step:

1. We created two `Rx.ReplaySubject(1)` s: `_tasks` & `_error` . These will be used as our change notification subjects. One for the data and the other for any errors.
2. `ReplaySubject` is a special subject that will replay a value from its buffer when a new subscriber is added.
3. We have added getters for both `_tasks` and `_error` . These can be used by an observer to subscribe to and be notified whenever a change occurs to our store.

The last step is to notify our observer of changes within the store. For this we modify the action handlers as follows:

```
...
private registerActionHandlers() {
  this.dispatcher.filter(
    action => action.actionType === TASK_ACTIONS.GET_TASKS_RESPONSE)
      .subscribe(action => this._tasks.onNext(action.tasks));

  this.dispatcher.filter(
    action => action.actionType === TASK_ACTIONS.GET_TASKS_RESPONSE_ERROR)
      .subscribe(action => this._error.onNext({
        type: action.actionType,
        error: action.error
      }));
}
...
```

Adding Getters to our Store

Being notified of a change within the store is not enough. We need to provide our observers with a mechanism to get the data from the Store. Let's add a getter method to get our list of tasks.

```
...
get tasks() {
  return this._tasks;
}
...
```

Using Stores within Components

Our Store is one part of a bigger picture, let's use our store within our `TaskListComponent` class.

```
export class TaskListComponent {

  tasks: Task[];
  users: {};
  user;
  errorMessage: String;

  ...

  constructor(
    private tasksStore: TasksStore
    ...
  ) {

    this.tasks = [];

    tasksStore.tasks
      .subscribe(tasks => this.tasks = tasks);
  }

  ...
}
```

All we are doing here is providing an observer to listen to store change events or error coming from the store. When a change occurs within the Store our Component gets notified and updates it's own (view related) list of tasks. If an error has occurred within the store the component get notified as well and can display it to the user (in some friendly form).

Since we used a `ReplaySubject` with a buffer of 1, we are guaranteed to get notified of the data within the store. Even if our component got instantiated after this event has already occurred.

Note, we did not have to change our `TaskComponent` implementation. This is because the `TaskListComponent` is passing it data down to it. It is recommended that only top level components subscribe to stores. Then pass that data down to other components.

Clean up

Even though our application is relatively small, and RxJS subscriptions are relatively lightweight on resources, we should cleanup our subscription when it is no longer needed. How do we know if the subscription is no longer needed?

There is another part of Angular we need to cover, and that is component life cycle events accessible to us via the `$scope` service.

Let's inject this service into out component as shown below:

```
...
constructor(private $scope, ...)
...
```

Since we will need to dispose of our subscriptions we need to keep a reference to them when we subscribe:

```
...
let tasksSubscription = tasksStore.tasks
    .subscribe(tasks => this.tasks = tasks);
...
```

Finally, let's use `$scope` 's `$on` method to subscribe to the `$destroy` even of the component in question:

```
...
constructor(
  private $log,
  private $scope,
  private tasksStore
  ) {

  ...

  this.$scope.$on('$destroy', () =>{
    tasksSubscription.dispose();
  }
}
...
```

A Case for Immutable Data

It is important that Stores, provide read-only access to its data. Stores should not provide setters to change their state. In our app so far we aren't actually maintaining state in the store. We fetch data from the server and publish it as is. In a more real world scenario you might actually want to maintain some data in the store. And then a reference to that state is pushed to the observers. This introduces certain complexities.

What if a component wants to make a change to the data retrieved from a store. This could be for UI purposes, re-shaping, etc. If that were to happen it would change the state of the store.

This is why our stores should maintain their state in immutable data structures. To illustrate on how this can be achieved, let's refactor our `TasksStore` to use immutable for it's model. Even though there could be several ways we could achieve this, for the purposes of this example we are going to be using the `Immutable.js` library: https://facebook.github.io/immutable-js/

```
...
import {fromJS} from 'immutable';

export class TasksStore {

  private _tasks: Rx.ReplaySubject<Task[]>;
  private _error: Rx.ReplaySubject<any>;
  private _data;

  static $inject = ['dispatcher'];

  ...

  constructor(private dispatcher: Rx.Subject<any>) {

    this._tasks = new Rx.ReplaySubject<Task[]>(1);
    this._error = new Rx.ReplaySubject(1);
    this._data = fromJS([]);

    this.registerActionHandlers();
  }

  ...

  private registerActionHandlers() {
    this.dispatcher.filter(
      action => action.actionType === TASK_ACTIONS.GET_TASKS_RESPONSE)
        .subscribe(action => {
          this._data = fromJS(tasks);
          this._tasks.onNext(_data)
        });
    ...
  }

  ...

}
```

Emitting Actions from Views

So far we have responded to changes within our Stores. Let's make a view that emits an Action.

Create a new file in *app/src/components/task-add/task-add-component.ts*

```
export class TaskAddComponent {

  static selector = 'ngcTaskAdd';

  static directiveFactory: ng.IDirectiveFactory = () => {
    return {
      restrict: 'E',
      scope: {},
      controllerAs: 'ctrl',
      bindToController: {},
      controller: TaskAddComponent,
      template: require('./task-add-component.html')
    };
  };

  static $inject = ['$log', 'tasksActions'];
  constructor(
    private $log,
    private tasksActions
   ) {
     //
  }

  save(task) {
    this.tasksActions.addTask(task);
  }
}
```

and a corresponding *task-add-component.html*

```
<div>
  <div>
    <h4>Add Task</h4>
  </div>
  <form>
    <label>Owner</label>
    <input
      type="text"
      ng-model="newTask.owner">
    <label>Description</label>
    <input
      type="text"
      ng-model="newTask.description">
    <button
      ng-click="ctrl.save(newTask)">
      Save
    </button>
  </form>
</div>
```

define a new action in *app/src/actions/task-actions.ts*

```
  ...
  addTask(newTask) {
    this.tasksService.addTask(newTask)
      .then(() => this.getTasks())
      .then(null, error => this.dispatcher.onNext({
        actionType: TASK_ACTIONS.GET_TASKS_RESPONSE_ERROR,
        error: error
      }));
  }
  ...
```

Note that we are calling `getTasks()` method from within `addTask()`, which will fetch the new tasks data from the server and emit a change on successful retrieval.

We should also take this time to go and add `addTasks` to *app/src/services/tasks/tasks-service.ts*

```
...
addTask(task) {
   return this.serverService.post('/api/v1/tasks', task);
  }
...
```

And also add the ability to post data in *app/src/services/server/server-service.ts*

```
...
  public post(path, data) {
   return this.$http.post(`${this.API_BASE_URL}${path}`, data);
 }
...
```

Unit Testing Stores

In the previous chapter we learned about Flux architecture by implementing our `TaskStore` and our `TaskListComponent`. But how do we unit test our Flux components that use RxJS?

Unit Testing a Component that depends on a Store

Let's start by having a look at out test for `TaskListComponent`. Create a new file *app/src/components/task-list/task-list-component.test.ts*, and copy the code below.

```
import {TaskListComponent} from './task-list-component';
import {TaskActions} from '../../actions/task/task-actions';

import 'angular';
import 'angular-mocks';
import * as Rx from 'rx';


let _$scope;
let _tasksStoreMock;

let _tasksMock = [{
  owner: 'alice',
  description: 'Build the dog shed.',
  done: true
}, {
  owner: 'bob',
  description: 'Get the milk.',
  done: false
}, {
  owner: 'alice',
  description: 'Fix the door handle.',
  done: true
}];

describe('TaskListComponent', () => {

  beforeEach(() => {
    angular.mock.inject($rootScope => {
      _$scope = $rootScope.$new();
    });

  });

  it('should get data from stores', () => {

    let scheduler = new Rx.TestScheduler();

    _tasksStoreMock = {
      tasks: scheduler.createHotObservable(
        Rx.ReactiveTest.onNext(200, _tasksMock))
    };

    let taskListComponent = new TaskListComponent(_$scope, _tasksStoreMock);

    scheduler.advanceTo(220);

    chai.expect(taskListComponent.tasks).to.equal(_tasksMock);
  });

});
```

The top part of the test should be familiar, we are just creating mock data to use within our test. We should look at the anatomy of the test defined in our only `it` block:

1. First, we create a TestScheduler object. TestScheduler is a virtual scheduler that allows us to control the timing of our test.
2. Then we create a mock `tasks` observable using the test scheduler. And define a virtual time for the `onNext` to be called with our mock data at 200 "ticks".
3. The code from (1) and (2) above allows us to create a mock store and instantiate our `TaskListComponent` class to be tested.
4. Then we advance out test scheduler to 220 ticks and test the state of our `TaskListComponent` instance at that time.

Note, that `done` callback is not required here, as our test is purely synchronous.

Unit Testing a Store

There are some similarities between testing services and stores. Let's have a look at the store test below, and copy the code into *app/src/stores/tasks/tasks-store.test.ts*.

```
import {TasksStore} from '../../stores';
import {TASK_ACTIONS} from '../../actions';

import * as Rx from 'rx';

describe('TasksStore', () => {

  let _scheduler;
  let _mockDispatcher;
  let _$log;

  let _mockTasks;

  beforeEach(() => {

    _mockTasks = [{
      _id: 2,
      owner: 'alice',
      description: 'Build the dog shed.'
    }, {
      _id: 5,
      owner: 'bob',
      description: 'Get the milk.'
    }, {
      _id: 7,
      owner: 'alice',
      description: 'Fix the door handle.'
    }];

    _scheduler = new Rx.TestScheduler();
  });

  it('should get a task by id', (done) => {

    _mockDispatcher = _scheduler.createColdObservable(
      Rx.ReactiveTest.onNext(10, {
        actionType: TASK_ACTIONS.GET_TASKS_RESPONSE,
        tasks: _mockTasks
      }));

    let tasksStore = new TasksStore(_mockDispatcher);

    tasksStore.getTaskById(5).subscribe(
      task => {
```

```
          chai.expect(task).to.not.be.undefined;
          chai.expect(task).to.deep.equal(_mockTasks[1]);
          done();
        }
      );

      _scheduler.advanceTo(25);

    });
  });
```

Most of the code above should be familiar by this point. The main goal of the unit test is to instantiate a store while creating a mock dispatcher using RxJS. This dispatcher is used to schedule an action to be piped into a store. Note that we are using an asynchronous test here, since we are working with observable streams.

UI Router

Routing allows us to express some aspects of the app's state in the URL. Unlike with server-side front-end solutions, this is optional - we can build the full app without ever changing the URL. Adding routing, however, allows the user to go straight into certain aspects of the app, which can be very convenient.

UI-Router

Angular's built in routing solution ('ng-route') has been de facto superseded by ui-router. We'll be using that. To use UI-Router you'll need to update your `app/src/app.ts` to inject the new module into your main module:

```
angular.module('ngcourse', [
  'ngcourse.tasks',
  'ngcourse.server',
  'ngcourse.router'
])
```

Creating a Router Service.

Let's start by adding our own "router" module which will serve as a wrapper around ui-router. Our module will have a `.config()` section.

This goes in `app/src/services/router/router-service.js`

```
export class RouterConfig {

  static $inject = ['$stateProvider', '$urlRouterProvider', '$locationProvider'];
  constructor(
    private $stateProvider,
    private $urlRouterProvider,
    private $locationProvider
  ) {

    $urlRouterProvider.otherwise('/tasks');
    $locationProvider.html5Mode(false);

    $stateProvider
      .state('tasks', {
        url: '/tasks',
        views: {
          '': {
            template: 'my tasks'
          }
        }
      });
  }
}
```

Let's configure the router within our *app.ts* file as follows:

```
angular.module('ngcourse.router', ['ui.router'])
  .config(RouterConfig);
```

We'll also need to add a directive for ui-router in index.html:

```
  <div ui-view></div>
```

Your index.html main section should now look like this:

```
...
<ngc-main>
  <div ui-view></main>
</ngc-main>
...
```

`ui-view` is a directive that `ui-router` uses to manage its views. It will be replaced by the template or templateURL that is configured for each ui-router state.

`ui-router` will insert the content in the `ui-view` element based on the current application state defined in `$stateProvider` above.

Let's talk about why this need to happen in the "config" section.

.config() and Providers

Up until now, we've mostly been dealing with services and controllers. However the example above introduces a couple of new concepts: providers and `.config()` blocks.

When an AngularJS application starts up, it goes through several 'phases':

1. Angular executes your code that contains calls to methods such as `.service()`, `.constant()`, `.config()`, etc. However, at this point all of those entities are only defined. They are not yet instantiated. In other words, Angular takes note of the fact that will want to create a service 'tasks' using the provided function. It does *not* however, call this function at this point.
2. Constants defined in `.constant()` blocks are set.
3. Registered providers are created. The order is determined by dependencies between providers.
4. All `.config()` blocks are executed in order in which they were defined. The code in config blocks can refer to constants and can call methods on providers.
5. Values are set.
6. Services are instantiated. This includes services defined by `.factory()` and `.service()`, as well as those defined via providers. The order depends on the dependencies between the services.
7. All .run() blocks are executed in the order in which they were defined.

A provider is something Angular's dependency injector can use to create a service. The provider can be configured with various data in the app's `config` phase, *before any services are instantiated*. When services are instantiated later, they can customize them based on configuration requests it received in the config phase.

In practical terms, any JavaScript object that exposes a function called $get() can serve as a provider.

So in this example, we're configuring ui-router's `$stateProvider`, `$urlRouterProvider`, and `$locationProvider` with settings that will later be used to generate state, route, and location services with the correct path and parameter data for use in our own controllers.

More States

```
$stateProvider
  .state('tasks', {
    url: '/tasks',
    template: 'my tasks'
  })
  .state('tasksDetail', {
    url: '/tasks/details',
    template: 'task details'
  })
  .state('account', {
    url: '/my-account',
    template: 'my account'
  });
```

States with Parameters

```
  .state('tasksDetailById', {
    url: '/tasks/{_id}',
    template: 'task details with id'
  })
```

This can include regular expressions:

```
  .state('tasksDetailByRegex', {
    url: '/tasks/{_id:[A-Za-z0-9-_]{0,}}',
    template: 'task details with regex'
  })
```

Now we are going to rebuild our view around ui-router. First, let's do tasks.

Components and Routing

The rule of thumb when using routing is that routes should be defined for top-level components. Generally, micro components should not be used in routing but instead used within the templates of macro component who pass data into them from above.

So in our case, `TaskListComponent` is a good candidate while `TaskComponent` is not.

There are 2 ways we can add a component as a routing state:

The inline template way

```
.state('tasks', {
  url: '/tasks',
  views: {
    '': {
      template: '<ngc-tasks></ngc-tasks>'
    }
  }
})
...
```

or the "controller" way

```
import {TaskListComponent} from 'components/task-list/task-list-component';
...

.state('tasks', {
  url: '/tasks',
  views: {
    '': {
      controller: TaskListComponent,
      controllerAs: 'ctrl',
      template: TaskListComponent.template
    }
  }
})
...
```

Both are equivalent, but with the former approach there is no need to define this component on a module using `.directive()`

Let's take a moment to review a few other aspects of ui-router.

Adding "Resolves"

We'll add a resolve: parameter to our state to demonstrate the user of resolves. Note that resolves isn't used to often and we'll remove this once we've learned the functionality.

```
  .state('account', {
      url: '/my-account',
      template: 'My account',
      resolve: {
        greeting: function($timeout) {
          return $timeout(function() {
            return 'Hello';
          }, 3000);
        }
      }
    });
```

The "resolve" property in a state configuration allows us to specify a set of dependencies that will need to resolved prior to transitioning to the new state. Those dependencies become injectable in the route's controller. In the example above, `greeting` property of the resolve is set to a function that returns a promise that resolves to 'Hello' after 3000 msec. (We generate this property using `$timeout`.) The UI-Router will wait until the promise resolves, then make the transition. The state's controller will be able to dependency-inject 'greeting', which will be set to 'Hello' by the time the controller is initialized.

This approach can simplify controller code, but does so at the cost of terrible user experience: after the user clicks on a button, nothing happens for 3 seconds, leaving the user wondering what happened.

A better approach is to not rely on "resolve" and instead make the transition immediately. The receiving controller can then decide what parts of the view can be displayed right away and what parts will need to be displayed with a short delay. For example, if the state involves displaying a list of objects that need to be retrieved from the server, the app can display everything other than the list, then make add the list items when they arrive. This usually produces a more natural experience for the user.

Nesting Views

One of the most powerful features of ui-router versus the out-of-the-box AngularJS router is nested views. To allow `ui-router` to know what view it's updating, we can add a name to the view as seen in `ui-view="child@parent"` below.

```
.state('parent', {
  url: '/parent',
  views: {
    'parent': {
      template: 'parent view <div ui-view="child@parent"></div>'
    }
  },
})
.state('parent.child1', {
  url: '/child1',
  views: {
    'child@parent': {
      template: 'child 1'
    }
  }
})
.state('parent.child2', {
  url: '/child2',
  views: {
    'child@parent': {
      template: 'child 2'
    }
  }
});
```

Update the parent index.html to be named using `<div ui-view="parent"/>`.

Nesting views allows sophisticated routing where parts of the view are defined by the parent state and parts are defined (or, overridden) by child states. Note that states get nested implicitly, based on names: "parent.child1" will be a child of "parent". (UI-Router also provides a facility for nesting states explicitly.) Child state's URL is understood to be relative to the parents. So, since "parent.child1" is a child of "parent" and parent's URL is "/parent", the URL for "child1" is "/parent/child1".

In the example above, the parent view provides part of the view (the text "parent view") and a placeholder where child states would go. When we visit child1 and child2, the parent's part of the view remains unchanged, while the child's section changes.

Alternatively, however, the child can override the parent's part of the view:

```
  .state('parent.child2.grandchild', {
    url: '/grandchild',
    views: {
      'child@parent': {
        template: 'parent overriden'
      }
    }
  })
```

In this case the "grandchild" overrides the view earlier defined by child2.

When overriding parents views we need to refer to them using the ..@.. which allows us to specify an absolute path to the view.

Transition Using `ui-sref`

We can easily transition between states using `ui-sref` directive:

```
<button ui-sref="tasks">Go to tasks</button>
```

Transitions Using `$state.go()`.

We can also transition using `$state.go()`:

```
$state.go('tasks.details', {_id: taskId});
```

However, let's wrap this in a service, we can use the same *router-service.ts* file for convinience:

```
...
export class RouterService {

  static $inject = ['$state'];
  constructor(private $state) { }

  goToTask(taskId) {
    this.$state.go('tasks.details', {
      _id: taskId
    });
  }

  goToTaskList() {
    this.$state.go('tasks', {}, {
      reload: true
    });
  }
};
```

Accessing Parameters Using `$stateParams`

`$stateParams` can be injected into your components using the `$inject` and used as follows:

```
$stateParams._id
```

But again, let's wrap it:

```
...
getTaskId() {
  return this.$stateParams._id;
};
```

Update Param Without a Reload

If we want to change the value of the parameters *without* triggering a state transition, we need to update the values in thre three different places where the UI route keeps them.

```
// Updates a state param without triggering a reload.
function quietlyUpdateParam(key, value) {
  $state.params[key] = value;
  $stateParams[key] = value;
  $state.$current.params[key] = value;
}
```

An example of where this would be useful is a Google Maps style UI, where the URL is continuously updated as the user moves around the map.

MEAN Stack Security Considerations

As our SPA's get more complex, and more business logic, rules, and validation move up to the client side - there are some new security considerations that need to be kept in mind.

However, a common misconception is that just because business rules and security checks have been moved up to the client side, negates the need to reimplement the same considerations on the server side. The old adage:

> Never Trust The Client

Still remains true. While validating data, applying security rules can lead to a more enjoyable user experience on the front-end, it does not negate the need to reinforce these checks on the server side.

The next sections of this course will discuss some steps that can be taken to help secure the front-end of your application, while also talking about some of the considerations that still need to be applied on the back-end.

Principals

0. When in doubt, ask for help.

Security is hard and it is easy to make a mistake. If you find yourself in the slightest doubt about a security topic, ask for assistance or a second opinion.

1. Code-review all security-related code.

All security-related code should be reviewed. This means that such code should be written to be easy to audit.

In other words, it is not enough for the code to *be* secure. It should also be easy to *see* that it is in fact secure.

2. Avoid DIY security.

When it comes to security, do-it-yourself is rarely a good idea, since it is very hard to verify that your solution actually delivers. It's better to go with battle-tested third-party solutions.

3. Choose third-party software wisely.

Third-party software brings the risk of security vulnerabilities. Keep this in mind when choosing third-party software, and choose solutions that are well-regarded by security experts. If you are not sure whether a piece of software is secure, ask someone who does.

4. Keep third-party software up to date.

Established solutions only offer more security if you keep up with security fixes. For this reason, keep third-party software up to date.

You can use tools such as "retire.js" to check for vulnerable NPM packages. Consider adding this check to your CI setup.

5. Ensure authentication and authorization.

Most apps require a method for establishing who the user is (authentication) and what they are allowed to do (authorization). You need to ensure that those methods actually work and cannot be circumvented.

6. Do not trust the client.

Anything that is stored in the browser's memory can be accessed by the user. Proceed with this assumption.

When you send JavaScript to a browser, you have no way of knowing if the browser will run this code unmodified. So, any security that you implement client-side can be easily circumvented. Do not send to the browser any data the user should not see. Do not accept from a browser any requests that the user should not be able to make.

7. Practice "courtesy security" client-side.

Client-side security is "courtesy security". It's not there to stop the user from misbehaving – that should be done by the server. The client's role is to guide the user towards choices that won't lead to a disappointingly rude response from the server.

E.g., disable the button that the user shouldn't click. They can still go into the console and use it to ask the server to do something they are not allowed to do. But at that point, a 400-level response is fair game. A well-behaved user, however, should be guided away from prohibited actions.

Client-side "courtesy security" frees your server from the need to be polite and simplifies your architecture.

8. Secure data in transit.

Data should be encrypted when traveling along channels where someone could intercept it. There are good, standard solutions for that.

9. Secure data at rest.

Data should be secure while it is being stored.

10. Secure the keys.

If you encrypt the data, it is only secure as long as the keys are secure. Consider how keys are being managed in your system.

11. Protect data from yourself.

In many cases, users' data needs to be protected not only from outsiders or other users, but also from our own and client's staff. Among other things, this usually means that you need to set things up in such a way that *you* wouldn't have access to that data either.

12. Plan to deploy to the cloud.

Leveraging cloud infrastructure makes it easier to ensure security compared to running your own servers. Our preferred cloud provider is Amazon AWS.

Note that we love Heroku for *development*, but prefer Amazon AWS for production deployment because of the wider set of security options. This means that when developing with Heroku it is important to avoid making too many commitments to it.

XSS

0. Understand XSS.

Cross-site scripting (XSS) involves someone finding a way to get your user's browser to run JavaScript code you didn't intend your app to run. The most common way of doing this is by submitting HTML through one of the forms and counting on your app to add this HTML to the page. Since injected code is run by the browser in the context of *your* application, the opportunities for abuse are endless.

Suppose we are writing a blog that accepts comments. When we display comments we do so by setting `div.innerHTML`. Someone posts a comment that says:

```
<script src="http//h4kk3rs.com/code/that/does/something/evil.js"></script>
```

Your app will add this HTML to the DOM, which will cause the browser to run the code.

1. Do not assume you can make HTML safe.

HTML is a beast. You can't parse it with regular expressions. You can't count on knowing all the crazy corners in all versions for all browsers. So, don't count on running HTML through a regexp and fully sanitizing it.

The only moderately safe way of handling HTML in a browser is to escape it, though even that will leave you open to XSS on some browsers.

2. Set Content Security Policy headers.

Content Security Policy headers tell modern browsers to block some of the most common holes. Don't count on it to prevent all XSS, but do it for a good measure.

3. Avoid attaching external HTML into DOM.

Since external HTML cannot be made safe, the best strategy is to avoid inserting it into DOM. Angular gives you some methods of doing this if you really insist, plus you always attach it to DOM directly. This doesn't make it a good idea, however.

AngularJS will escape HTML fed through `{{...}}` and `ng-bind`. This is for a good reason.

If you really insist, however, you *can* add HTML to a view by using `ng-bind-html`. Angular will require you to push the content through `$sce.trustAsHtml()` before you can do this. Note that this does not make it safe. `trustAsHtml()` is just your way of saying to Angular: "Trust me, I know what I am doing." The question is: do you?

If you really have to incorporate untrusted HTML, run it through `$sanitize`. Make sure to keep your fingers crossed.

Authentication

0. Never store users' passwords. Store "hashes" instead.

End users' passwords should never be stored in cleartext. There are no exceptions to this rule. Instead, store a "hash" generated by Bcrypt or another similar library.

Use Bcrypt to generate a "hash" of the user's password and store this in the database. When the user submits a password for authentication, use Bcrypt to verify that submitted password matches the stored hash.

The main problem with storing users' password in cleartext is not even that this will allow someone to steal the password and perhaps login into your application. Rather, it is the fact that the user may well be using the same password for all sorts of other applications. Maybe they are using it to log into their bank. For this reason, storing users' passwords in cleartext is never the right solution.

In most cases the best solution is to use a key derivation function (KDF) to generate what is colloquially referred to as a "hash" of the password. This "hash" is stored instead of the password used for checking if the password supplied by the user is correct. A key produced by a KDF is often referred to as a "hash" because KDFs are often based on cryptographic hash functions and are conceptually similar. Hash functions such as MD5, SHA1, SHA256, etc, are often used to implement "poor man's" KDFs. A simple hash function does not make the best KDF, however.

Proper commonly-used KDFs include Bcrypt, PBKDF2-SHA256, and Scrypt. We usually use Bcrypt. Note that PBKDF2-SHA256 is based on SHA256, but it's not the same thing as just using SHA256.

When using a KDF, it's important to use a different salt for every password to protect against dictionary attacks. If you were to try to implement your own KDF based on a hash function, you would need to make sure you use this salt properly. However, you shouldn't write your own KDFs. Use a standard one, and it will handle salts for you.

Bcrypt has one convenient feature that makes it easy to use well. Bcrypt can generate an appropriate salt automatically and you can tweak the strength by changing the "work factor" parameter. The salt and the chosen work factor are both stored in the resulting hash, so you

don't need to worry about storing them separately. This also makes it easy to upgrade the strength of your "hashes" later, as you can have the users do this whenever they change passwords.

In rare cases you will need to be able to retrieve the original password. In those cases you will need to encrypt it and worry about how you manage the keys.

1. Do not recover passwords, reset them.

Storing a hash means that user's original password cannot be recovered. Consider this a feature. However, offer the user a way of resetting their password if they have forgotten it.

Password reset by email is a standard solution for password reset. You send the user a link with a token that allows them to set a new password. This solution should be implemented in such a way that the token can only be used once and that it would expire in a short period of time.

The token should be chosen to be hard to guess, for example by a cryptographically secure random generator (for example, NodeJS's `crypto.randomBytes()`).

2. Ensure minimum password quality.

The needed strength of the password depends on the application. However, it is important to require some minimal strength.

You probably shouldn't allow your users to use "123456" or "password" as their password.

3. Count failed login attempts.

Your server should not let the user keep trying different username and password combinations until they find a match. You should either lock them out after some time or use captcha.

This needs to be done on the *server*. A user who is trying to brute-force a login is not going to do it using your client code. They'll probably be using `curl` or something like this.

Your server should respond to this by blocking the user, either permanently or for a short period of time. A short lockout (say, 5 minutes) can be fairly effective in stopping a brute force attack. You can also require captcha at this point.

If you lock the user out permanently, you will need to make sure there is a way for them to reinstate their account.

4. Consider "social" authentication.

Authentication using Google, Facebook, etc., allows you to push some of the work onto those applications and the user has one less password to remember.

5. Use token-based authentication.

Once the user is authenticated, you will need to either issue them an authentication token or a cookie. Tokens are vastly preferred.

Cookies are easier to steal and are hard to use in a multi-server setup. Tokens solve both of those problems. They can also be used with a completely stateless server architecture.

Our standard solution is JWT-style bearer tokens.

6. Consider how you store the tokens.

If you want the user to stay logged in after they close your application window, you will need to store the token somehow on the client. `localStorage` is a common option and is suitable for most applications, but you need to consider the specific requirements of your app. You need to remember to remove the tokens upon logout.

Tokens stored in `localStorage` can be accessed by any JavaScript loaded from the same domain. This becomes a problem if your application is vulnerable to an XSS attack (see XSS). Some applications may require a more elaborate approach to storing authentication tokens, though it's important to remember that there is no silver bullet here.

There are two routes towards securing `localStorage` further. One is to encrypt the token before storing. This step creates a hurdle for an attacker, but you now face the challenge of where to store the key. There are a few options, none of them entirely satisfying.

Another approach is to take advantage of the fact that `localStorage` is domain specific to ensure that the token is written by a domain that we are certain is safe from XSS. In this case we need to figure out how to move the token between the domains, which can be done using an iframe and `postMessage()`.

7. Consider the personal information stored in the token.

When using JWT tokens, consider that the token carries a data payload that is **signed** but is not **encrypted**. If the content of that payload is sensitive, the server should be encrypting it before sending it to the client.

A JWT token is encoded and **looks** encrypted, but it can be trivially converted into a JSON object.

8. When using cookies, protect against CSRF.

If you *must* use cookies, make sure your app is safe against cross-site request forgery (CSRF or XSRF). But really, just avoid cookies.

CSRF is an attack in which someone causes the user's browser to call your server using that user's credentials. In this case, the user is someone who have authenticated with your server using your application, but the call is being made by another application on that user's computer.

This can involve making an action or getting data. Requests for JSON content are in theory protected from CSRF by the same-origin policy, but in practice this can be circumvented. The JSON Array" vulnerability is just one of the ways this can be done.

Angular provides a partial solution for this via the use of `X-XSRF-TOKEN` header. However, in order for this to work this needs to be supported on the server. You would also need to verify that this is actually working in your case.

If implementing CSRF in Node.js, beware of this vulnerability in `methodOverride` middleware: "Bypass Connect CSRF protection by abusing methodOverride Middleware."

And again, ask yourself if you really want to be using cookies.

9. Pick a sensible session expiration policy.

When you authenticated the user, they should not stay authenticated forever. The specific policy that makes most sense depends on the application.

A common strategy is to set an expiration time and provide an endpoint for renewing the token. In this case, the client should keep track of whether the user is still active and renew the token prior to its expiration.

For example, if the server issued a token that lasts for 1 week, the client wait for 3 days then renew the token when the user interacts with the system again.

10. Require re-authentication for sensitive features.

It is generally a good idea to ask the users to re-authenticate before accessing features that may be particularly sensitive. Among other things, the user should always be asked to re-authenticate before changing their password.

If the user is resetting their password because they forgot it, they would need to use an alternative way of establishing their identity, e.g., the password reset token that was sent to them by email (see §8.1.1).

This becomes particularly important when using an authentication strategy that allows the user to stay logged in indefinitely.

HTTP does not provide standard vocabulary for such a re-authentication request, but this can be communicated by a simple 401, as long as the client knows how to handle it.

11. Consider multi-factor authentication.

Multi-factor authentication means asking the user to identify themselves with something they *have* in addition to providing the password. The most common solution today is having the user enter a code that you send to their phone.

Authorization

0. Do not confuse authorization and authentication.

Just the fact that the user is authenticated (you know who they are) doesn't mean you should do whatever they ask you to do.

1. Consider resource-based authorization.

In many cases it makes sense to model authorization as a matter of certain users having special access to specific resources. E.g., we could designate some users as being "owners" or a particular resource and give them special access to this resource based on that fact.

2. Consider role-based authorization.

In other cases it makes more sense to assign users to "roles" and link permissions with those roles.

It's usually hard to get away from resource-based authorization entirely, so in practice role-based authorization often ends up being used in combination with resource-based authorization.

Credentials

0. Pick good passwords.

Make sure to use good passwords for securing servers, databases, etc. Good passwords are random and look like this: "XRexQTMPvE4VjxnejG9qPg3U". If you can remember it, it's probably not a good password.

You can use a secure password manager such as LastPass to keep track of those passwords. Or store them in a file on your machine.

1. Never commit credentials or keys.

Do not commit credentials and keys, such as database passwords, to the repository. Load them from environment variables or a key management system.

If your software loads credentials from environment variables, you can load them into your environment from a bash script that you keep in a folder called "private" that is added to your `.gitignore` file to prevent you from committing it by accident.

An alternative approach is to encrypt individual configurations, commit the encrypted values, and store the decryption key in an environment variable.

The best approach is to use a proper key management system.

While production credentials and keys are the most important ones to secure, it is best to avoid committing development and staging credentials as well, as a matter of establishing good habits.

2. Do not log credentials and keys.

Do not log passwords and other credentials either to the console or to the file.

3. You shouldn't have the production credentials.

If you know production credentials for a system and you are not a member of the operations team, something is not right.

4. Consider a proper key management system.

To truly secure the keys you would need to use a proper key management system, i.e., a software solution that will deliver the right keys to authorized parties. AWS's Key Management Service (KMS) is one example of such a system. A proper key management system will be able to provide an audit of access to the keys.

Data in Transit

0. Use strong SSL in production between server and client.

Use of HTTPS/SSL/TLS is essential for communication between the browser and the server and should make use of the latest version of SSL/TLS available. This does not necessarily need to be done in the server code - it can be done with an SSL proxy in production. It does need to be done, however.

This is not necessarily something we need to provide, since this can be handled in production by the operations team via a HTTPS proxy or a load balancer that handles SSL termination. However, we need to ensure that the client understands this. Also, in cases where rangle.io is handling deployment, we absolutely need to enable SSL. Some applications may require an SSL connection between the server and the load balancer.

SSL and TLS are basically the same thing. Strictly speaking, TLS is a newer version of SSL. In practice, people usually say SSL when they mean TLS. HTTPS is HTTP with TLS.

1. Enable HTTP Strict Transport Security in production.

HTTP Strict Transport Security (HSTS) is a mechanism by which a server can tell the client that the client should never connect to this server without SSL. This can be done in production but we need to ensure that it's done.

This can be done by a proxy. We need to verify that client's operations team is aware of this. In cases where we do deployment, we need to make sure this is done correctly.

This can be easily done using helmet.

2. Ensure a private connection to the database in production.

To ensure privacy of the data passing between the server and the database, either use SSL or put the two on a secure network. Better yet do both.

MongoDB can now support SSL connections, but this requires a custom build and is not supported by all providers. When not using SSL, ensure that the traffic between the server and the database only passes through a private network.

This can also be set up in production, but we need to ensure that the client's operations teams understands this requirement.

Data at Rest

0. Consider drive encryption.

Many applications require encrypting data while it is being stored. The simplest way to achieve this is by using full hard-drive encryption.

AWS makes it easy to encrypt EBS volumes with no performance loss, but you have to be mindful of where you store the keys.

1. Consider application-level encryption.

Application level encryption means your application encrypts the data before saving it in the database. This is more work than full drive encryption, but offers more protection for your data.

With good application-level encryption, someone who manages to get into your database will find encrypted data.

Application-level encryption has an important downside: since your database essentially does not have access to the original data, many types of queries become hard or impossible to carry out. For this reason, application-level encryption is often applied selectively.

2. Consider public-key encryption.

Classic symmetric encryption uses the same key for encrypting and decrypting the data. This makes sense where when the data is being encrypted for later consumption by the same entity. In cases where the data is being sent from one entity to another, however, public-key encryption makes more sense.

E.g., if process A writes data into the database and process B read it, we can have process A write the data using process B's public key. This eliminates the need to share the private key with staff members who are supporting process A.

3. Use standard libraries.

NodeJS's crypto module should be the first place to look.

The main functions to consider are `crypto.publicEncrypt()` and `crypto.privateDecrypt()`.

4. Have a plan for how to manage keys.

Encryption is of little use if you don't secure the keys.

Privacy

0. Make privacy possible.

As application developers we usually do not control what our clients will ultimately do with their users' data, but we can make it easier for them to follow best practices in regards to user data. We can also advise them about such best practices.

1. Don't collect data you don't need.

The easiest way to avoid disclosing sensitive user data is to not collect it in the first place. Consider alternative solutions.

In the age of big data it may be tempting to collect everything you can, but every bit of information you collect is a bit that you can later inadvertently compromise. If the client wants to collect sensitive data, try to understand their ultimate objectives and consider suggesting alternative ways of achieving them.

2. Disclose and ask for consent.

When collecting data, you would usually want to ensure that the user understands what data is being collected and why. It is also helpful to indicate whether the data is required.

3. Consider partitioning your data.

Partitioning your data can help you manage privacy in a number of ways. First, you can separate sensitive data from non-sensitive data, which allows you to focus on securing the smaller data set better. Alternatively, you can anonymize your data by storing identifying information separately.

Partitioning your data into two different data stores (e.g., two different MongoDB databases) makes it possible for the client organization to give specific people access to a part of the data without giving them access to all of them. If user's accounts and other identifying information are stored in one database while their private details (medical exam results, financial assets) are stored in a different database, a person can be tasked with doing maintenance or analytics on one of the databases without being granted access to the other.

4. Guard anonymized data.

It is really hard to truly anonymize data. Do not assume that removing names is enough. Instead, assume that any "anonymized" data set can be potentially de-anonymized. Consequently, take care to preserve even "anonymized" data.

AOL learned this the hard way in 2006. So did Facebook in 2011.

5. Only expose what needs to be exposed.

When working with sensitive data, ensure that your endpoints only send out data that needs to be sent out.

This is also a good idea when working with less sensitive data. First, most data is potentially sensitive. Second, don't waste bandwidth.

6. Consider individual access rights.

Many jurisdictions mandate that a user should be given access to their personal information stored by an organization. You should consider how such access would be practically realized in your system.

Depending on your database design, complying with such a request could either be trivial or very costly.

Once again, the less data you collect, the less data is there to provide to the user upon demand.

7. Consider access monitoring.

Many jurisdictions require monitoring of access to private data. This can sometimes be done as parts of the operations setup, but application support may also be needed.

Node

0. Stay abreast of vulnerabilities.

New lines of attack are discovered all the time, so it is important to keep up to date. When you learn about new threats, alert others.

1. Detect vulnerabilities in your dependencies and upgrade.

Check for known vulnerabilities throughout the stack using a tool such as retire.js. Upgrade to the newer versions as soon as fixes are available.

2. Use a tool to block common attacks.

NPM's module helmetjs provides solutions for a number of common problems.

3. Don't trust submitted data.

Don't trust the data submitted from the client, whether it comes in the body or as parameters. This data may not be what you think it is. Vectors of attack include database query injection and ReDoS.

MongoDB is vulnerable to query injection much like SQL, but the fact that MongoDB's queries are JavaScript object makes this injection potentially even harder to notice. A query for `{username: query.username}` may have rather unexpected results when `query.username` is an object rather than a string.

Keep in mind that Node's `qs` module expands query parameters into JavaScript objects for you. So, check the type of the values you accept from the client.

Another risk that falls under the same header is "Regular Expression Denial of Service" (ReDoS) attack. Certain regular expressions can take a really really long time to compute when presented with certain kinds of long input. This can allow an attacker to bring your

server down by submitting a query with a parameter that is a few hundred characters longer than you expected. If you expect user's input to be under certain length, consider truncating it at that length.

4. Prevent internal implementation disclosure.

Do not let Express tell the world your server is running express. This is one of the things helmet can do for you.

5. Avoid "chatty" error messages.

Error messages that provide a lot of context are great for development but may provide an attacker with unnecessary assistance. As a general rule, 400-level error messages should state what's wrong with the request so that the caller could correct it, but 500-level error messages shouldn't provide any details as to what happened.

In most cases, a 500-level error code should not provide any further information beyond the actual status code. If something went terribly wrong on the server, "500 Internal Server Error" is all the client needs to know.

6. Disable development endpoints by default.

If you add any endpoints that are meant to be used in development mode only, make sure that they are disabled by default and must be actively enabled in development mode with a flag.

7. Handle recognized errors, but crash on unhandled ones.

You should handle errors that you recognize and know how to recover from. Do not attempt to recover from unrecognized errors, however. Let your process die.

If you do nothing, your NodeJS process dies upon an unhandled exception. This is the recommended practice. You can try to catch unhandled errors and log them, but the best thing to do after that is still to kill the process.

An unhandled error is by definition a bug. Something happened that the developer did not anticipate. At this point you do not know what state your process is in. Proceeding despite this brings a number of risks that you don't want to run. Instead, accept the defeat and let your process die. Use a service such as forever to restart your failed processes.

Crashing on unhandled exceptions highlights the importance of handling the exceptions that you know you can handle. You don't want people to stage denial-of-service attacks against your server by forcing it to crash repeatedly with simple requests.

Node Modules

Keeping in mind with the principles of *Avoid DIY Security* and *Choose third-party software wisely.*, there are two commonly used node modules that can be used to help secure your node servers - helmet, and lusca by PayPal. Both of these offer similar, but different sets of functionality.

Helmet

Helmet is a collection of middleware that can be included with your express server to help protect against a few common attack vectors, and consists of.

- contentSecurityPolicy for setting Content Security Policy
- dnsPrefetchControl controls browser DNS prefetching
- frameguard to prevent clickjacking
- hidePoweredBy to remove the X-Powered-By header
- hpkp for HTTP Public Key Pinning
- hsts for HTTP Strict Transport Security
- ieNoOpen sets X-Download-Options for IE8+
- noCache to disable client-side caching
- noSniff to keep clients from sniffing the MIME type
- xssFilter adds some small XSS protections source

Helmet out of the box without an extra configuration will make use of most of the above middleware with the exception of

- contentSecurityPolicy
- dnsPrefetchControl
- hpkp
- noCache

Which require some additional configuration, and/or may not be appropriate in all environments.

Lusca

lusca is another collection of middleware to protect against some common attacks, and is part of the Kraken framework by PayPal. However, it can be used independently of Kraken, as Kraken is a framework that is built on-top of Express.

While there is some overlap between Lusca and Helmet, the features provided by Lusca include:

- Cross Site Request Forgery (CSRF)
- X-Frame for clickjacking
- P3P Headers
- HTP Strict Transport Security
- X-XSS-Protection headers

For all of the features of Lusca to work, you will also need to use either express-session, or cookie-session.

Lucsa and CSRF

One of the noteworthy features of Lucasa, and their CSRF support, is an Angular specific that will et lusca up to use the default settings for CSRF validation according to the AngularJS docs.

Node

Error handling in Node applications is an important aspect to get right. One way of ensuring that all of the errors in your controllers are handled properly, is to create a controller factory that will handle the response for you, and allowing your controllers to simply return resolved or rejected promises.

The role of error handling and logging can then be delegated to the controller factory.

controller factory

```
'use strict';

const _ = require('lodash');

function logToConsole (data) {
  console.log('------------------');
  console.log(JSON.stringify(data, null, ' '));
}

module.exports = function (promiseFn, opts) {
  const _opts = _.merge({
    errorCode: 1000,
    successHTTPCode: 200,
    errorHTTPCode: 500,
    errorHTTPResponse: 'Internal error'
  }, opts);

  return function (req, res, next) {
    // Wrap every controller
    new Promise(function(resolve, reject){
      // Controllers are expected to return either a promise or value
      return promiseFn(req).then(resolve, reject);
    })
      .then(function (data) {

        if (process.env.NODE_ENV === 'development') {
          logToConsole(data);
        }

        res.status(_opts.successHTTPCode).json(data);

      }, function (e) {
        // Handled error
        if (e.safe) {
```

```
            const errCode = e.httpCode || _opts.errorHTTPCode;
            const errData = process.env.NODE_ENV === 'development' && e || {
              code: e.code || _opts.errorCode,
              msg: e.httpResponse || _opts.errorHTTPResponse
            };

            if (process.env.NODE_ENV === 'development') {
              logToConsole(errData);
            }

            res.status(errCode).json(errData);
          } else {
            // Unhandled error, let the error handling middleware do its job
            next(e);
          }
        });
    };
};
```

When setting up the routes for your express application, pass the controller code through the controller factory:

Example Controller

```
const error = require('./error-codes');
const build = require('./controller-factory');
const router = require('express').Router();
const routeController = function(req) {
  const ids = req.query.ids;

  if (!_.isArray(ids)) {
   throw new XError(error.generic.input);
 }

  return ExampleMongoQuery.find({'id': {$in: ids}}).lean().exec();
}

router.get('/route', build(routeController));
```

error-codes

To help with handling errors, creating a file of common errors, the type of status code you want to return them as and if they are safe or not can be a useful approach to managing sending errors to the client, and for logging purposes.

An example of an `error-codes` file:

```
module.exports = {
  // Generic errors
  generic: {
    internal: {
      code: 1001,
      httpCode: 500,
      httpResponse: 'Internal error',
      safe: true
    },
    input: {
      code: 1002,
      httpCode: 400,
      httpResponse: 'Invalid input',
      safe: true
    }
  }
}
```

This can give you a centralized way to handle expected errors within your application. If you recall in our controller factory, there is a check for `e.safe` - as these are errors that we have knowingly raised within our application, and trust that we can recover from.

If the source of the error is something that we have not anticipated or planned for, it can be considered an unsafe error. This is where the generic error handling middleware should take over, and return a status `500` and restart the system.

Unhandled Errors

```
// other middleware ...
// mount routes ...

// Catch-all error handler
app.use(function (err, req, res, next) {

  if (process.env.NODE_ENV === 'development') {
    console.log(err.message);
    console.log(err.stack);
  }

  res.status(500).json({
    code: 1000,
    msg: 'Internal error'
  });

  // Restart the server.
  setTimeout(function () {
    throw err;
  });
});
```

In the case of critical errors, and unhandled exceptions it is better to let the server restart, and have an external process monitor such as forver or pm2 monitor and restart your node servers.

In summary

This allows you to delegate the error handling code to one location, and ensure that all controllers will have the same error handling. An important aspect of handling these errors, is knowing if you are in a production or development environment.

When in a development environment, you often want to have more robust logging details to help debug the source of the issue. However, when in production environments you do not want to leak details to the end user that could expose information that could compromise the security of your system.

JSON Web Tokens - JWT

JWT is a specification for storing in a stateless way the information about a user, and what permissions a user can have within a system. Since JWT is a JSON object, it provides a great deal of flexibility in how you store, manage and make use of this information.

Since JWTs can be signed and encrypted, it is also possible to store them on the client side. A JWT is a Base64 encoded string that consists of three parts - a header that describes a token, a body with the details of the user, their roles, token expiry information, and a third section that is a has that verifies the integrity of the token.

However, where and how to store your JWT, handling expiry and token refresh are still areas that need to be considered. Incorrect use of JWT can leave your tokens vulnerable to man in the middle attacks, cross-site forgery attacks. In the next section, we will discuss the security considerations of where to store your tokens, and how to handle refresh and/or revocation.

Storing Tokens

The options available for storing your JWTs are

- HTML5 sessionStorage/localStorage
- Cookies

HTML5 session/localStorage

When using sessionStorage or localStorage, once a user has authenticated the response body would contain the token, and you could persist this to local storage, and have it sent out with every request with your application.

However, HTML5 storage is vulnerable to XSS attacks where it is possible for a malicious script to read the the entirety of what is stored in localStorage, and is becoming recommended to not store sensitive information

Cookies

The other option to storing JWT information in Cookies, this however comes with its own set of security challenges, such as still being vulnerable to CSRF attacks. If storing your JWT in a Cookie, it is a good idea to have them be HTTPS only to ensure they are not transmitted

across an insecure connection. However, when making a Cookie HTTPS only - your JavaScript application can no longer read the security details from it.

One approach to work around this, is generating a token that can give access to an endpoint such as `/user/whoami` which will contain the payload of the JWT in the response body. This can be used during the bootstrapping of your application.

Token Revoke & Expiry

Once a token is generated, it is not a good idea to let token live forever. If someone else is able to gain access to the token somehow, it can allow unauthorized users to gain and maintain access to the system. Another consideration is when a users roles and permissions have changed - you do not want to allow a stale token with now incorrect permissions to still have access to the system.

Token Refresh

One approach is to have the access token have a shorter duration, and another refresh token that can be used to generate a new JWT to automatically refresh.

However, the implementation of this approach can be clunky, as once a request fails - you will need to issue a request for a new token, and then attempt the request again.

Token Revoke

Another approach is to revoke the token. While this does add some state to your JWT tokens, it might be the appropriate compromise to help ensure the security of your system.

For example, you could attach a status field to the account attached to user, and when verifying the token - see if the user is still active and grant access depending on that.

Or, you could leverage the `jti` - JWT ID Claim of the token.

The `jti` should be a unique identifier for the token, and should also be stored in a database in a way that associates a user with the token. When a users access has been changed, or revoked - the `jti` should be added to a blacklist of tokens, and prevent access to the system.

Security with Angular $http

As with any SPA application, network requests consuming your restful API will be exposed to users that know how to monitor the network traffic in their browser. As such, it is still important to implement appropriate security checks on the server side to prevent users from being able to make requests for data that they should not have access to.

However, Angular does provide the ability to help prevent against cross-site request forgery - CSRF, or sometimes referred to as XSRF. If you are using the lusca middleware that was described in the node modules section, it will set the appropriate defaults for the header and cookie settings.

This will create an JavaScript readable `XSRF-TOKEN` cookie on the first HTTP GET request, and this will then be sent as an `X-XSRF-TOKEN` header with subsequent requests.

The Angular `$http` service does this as part of the default transform request and response transforms. If you have the need to create your own custom request and response transformations, it is important to append your transformations instead of overwriting the default ones provided.

```
function appendTransform(defaults, transform) {

  // We can't guarantee that the default transformation is an array
  defaults = angular.isArray(defaults) ? defaults : [defaults];

  // Append the new transformation to the defaults
  return defaults.concat(transform);
}

$http({
  url: '...',
  method: 'GET',
  transformResponse: appendTransform($http.defaults.transformResponse, function(value) {
    return doTransform(value);
  })
});
```

source

Restricting Angular Routes

There may be routes in your application that you want to restrict based on user roles and permissions. To help restrict access to routes in your application, you can use the `resolve` hook of ui-router.

```
// ....

$stateProvider.state('admin', {
  url: '/admin',
  template: 'Secured Area, requires role admin.' +
    '<a ui-sref="home">Home</a><br/>' +
      '<ui-view></ui-view>',
  resolve: {
    hasAccess: function(securityCheck) {
      return securityCheck.checkAccess('admin')
    }
  })
```

While resolve can be used to prefetch data before loading a controller, if one of the resolved dependencies is rejected it will raise a `$stateChangeError` event and not navigate to the route.

The implementation of `checkAccess` will depend on your needs, but it could consume an API to check for permissions, or make use of the user session object.

If a user does not have permissions, you should provide a rejection reason so you can inspect this in your `$stateChangeError` handler and do the appropriate logging, and redirection to an error page.

```
angular
.module('ngCourse')
.service('securityCheck', function($q, $window) {
    this.checkAccess = function(permissions) {

      if ($window.userRoles.indexOf(permissions) >= 0) {
        return $q.when(true)
      } else {
        return $q.reject({
          type: 'ACCESS_DENIED'
        })
      }
    }
  })
/// ...
.run(function($rootScope, $state, $log, $window) {

    $rootScope.$on('$stateChangeError', function(
      event,
      toState,
      toParams,
      fromState,
      fromParams,
      error) {
      if (error && error.type === 'ACCESS_DENIED') {
        $log.error('User attempted to navigate to a restricted route')
        $state.go('error')
      }

    })
```

example

When `ui-router` is resolving the routes, it will attempt to resolve the parents dependencies before loading the child-routes, so you can easily perform a permission check for an entire area of the application.

However, keep in mind that users may be able to bypass these checks and navigate to the routes. The API requests that these secured areas use will need to have the appropriate server side checks in place.

Restricting Views and Functionality

- Depending on how granular your roles are, you may want to be able to re-use the same views and controllers for all users, but restrict which fields are visible, or which actions can be taken
- Can create a directive that hooks into your permission system to enable/disable, or hide parts of the UI
- Keep in mind, this is to help with the UI/UX

Possible implementation:

```
'use strict';
angular
  .module('ngcCourse', [
    'ngcCourse.permissions'
  ])
  .directive('ngcRequires', function () {
    return {
      restrict: 'EA',
      transclude: true,
      replace: false,
      template: `<span
                  ng-if="ngcRequires.hasPermission"
                  ng-transclude></span>
`,
      scope: {
        permissions: '=',
        anyPermission: '='
      },
      controller: 'NgcRequiresController',
      controllerAs: 'ngcRequires'
    };
  })
  .controller('NgcRequiresController', function ($scope, contentPermissionFilter) {

    (function init(vm) {
      angular.extend(vm, {
        hasPermission: false,
      });

      $scope.$watchGroup(['permissions','anyPermission'], function (newValue) {
        if (newValue) {

          var permissionRequest = {
            requiredPermissions: newValue[0],
            anyPermissions: newValue[1]
          };
          // Create a custom service that can handle checking
          // permissions based on your needs. Could be done
          // client side, or possibly make an API request

          contentPermissionFilter.checkPermissionRequest(permissionRequest)
            .then(function (permissionCheckResult) {
                vm.hasPermission = permissionCheckResult;
            });
        }
      });
    })(this);
  });
```

Example use:

```
<button ngc-requires permissions="['tasks.canAdd']" ng-click="addTask()">
Add Task
</button>
```