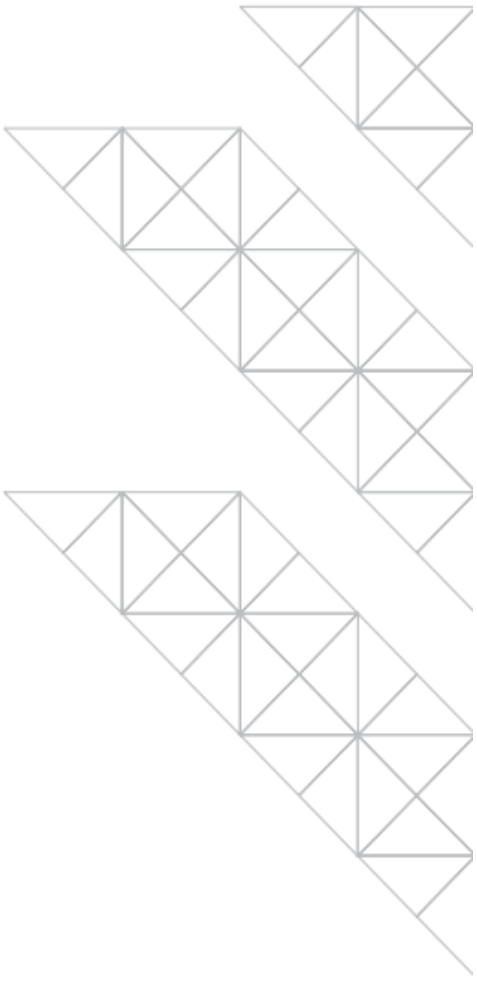
AngularJS - Done Right



Front Edge Digital

consult. design. develop.

Contents

Angular
Alternative Frameworks
Angular Popularity
Prerequisites
Single Page Web-apps
Introducing AngularJS
Declaring an Angular Application
Modules
MVC / MVVM
Packages / Frameworks
Controllers
Digest Cycle
$Apply
Dependency Injection
Directives
Custom Directives
Services
Asynchronous
Routing
UI-Router
Filters
Angular Forms
Translation
References

Angular

What is Angular?

- Structural framework for dynamic javascript web apps
- Complete client-side solution
- Provides a high-level of abstraction - built for CRUD applications
- Open source: maintained by google team, and large community
- First beta release: 2010 / First stable release: 2012
- Currently on version 1.4
- Major changes coming with version 2.0 next year

How does Angular work?

- Turns static content into dynamic content
- Directives attach behavior to DOM elements

Why Angular?

- Clear separation between the model, view and controller
- Decouple client-side from server-side
- Extremely modular
- Data Binding out of the box
- Extend HTML by adding specified behavior to DOM elements
- Grouping of HTML into reusable components
- Declarative approach is easy to read, great for team collaboration
- Data models in Angular are plain old JavaScript objects
- Highly Testable

Alternative Frameworks

1. Backbone
2. Ember
3. Knockout
4. React

Angular Popularity

- More Github contributors
- More job opportunities
- Bigger community on StackOverflow

Prerequisites

- JavaScript
- HTML
- Understand DOM manipulation
- Data structures, e.g. JSON
- Understand asynchronous behavior
- Bonus knowledge: Promises
- Bonus knowledge: Development design patterns

Single Page Web-apps

Single page web applications eliminate page loads after initialization, creating an uninterrupted user experience (if coded well).

Rather than the backend serving views to the client, the client requests data by interaction with the user interface. This essential hands the view back to client-side developers who can update the DOM by requesting data from the server.

JavaScript is a single-threaded language meaning events are placed in a queue and are processed one after the other.

Asynchronous (async) behavior means being able to do multiple things at the same time without locking the main thread. In node and JavaScript in general, this applies particularly to ajax file requests. Interaction fires off several async calls to retrieve data, and the main thread will not be locked while it renders the content. For example, you may have a program of 1000 instructions but you cannot say what the order of its execution is. What happens depends on what buttons and options the user clicks. Event handlers are called in many different orders.

The semantics of Angular dictate that you use promises as a sort of 'callback handle'. Do something asynchronous in a service, return a promise, and when the asynchronous work is done, the promise callback function is triggered. This is how an Angular single page application works.

Introducing AngularJS

Scope

Scope is the glue between an application controller and the view.

A Scope object can be bound to an element in the HTML (there are various approaches for binding scope to elements in the view covered in this document). 'ng-model' provides the ability to bind Scope to form elements. When the form element is updated, the value of the Scope property updates. Double curly braces provide means of displaying Scope properties as text in the view.

Below, 'ng-model' binds the Scope property 'name' to the form input and the curly braces displays the value of 'name' in the span.

JavaScript:

```
$scope.name = 'Batman';
```

HTML:

```
<input type="text" ng-model="name" />
<span>Hi! My name is {{name}}</span>
```

When an HTML element is updated (via a text input for instance), the Scope object bound to that element is updated. The Scope object is accessible in Angular JavaScript functions and reflects the changes made in the view. The $scope Object is really a glorified ViewModel.

Initial Scope value:

Batman

Hi! My name is Batman

Updated via input field:

Robin

Hi! My name is Robin

Modules

Angular is comprised of Modules. At it's most simplest, a Module is a container for code. 'angular.module' is a global place for creating, registering and retrieving Angular Modules.

The 'ng' Module is loaded by default when an AngularJS application is started. The Module itself contains the essential 'components' for an Angular application to function.

There are several other default angular modules you can choose to load with your application, each providing specific functionality. And of course, you can create your own modules, which we will discuss as we continue on.

Declaring an Angular Application

1. Include the Angular JavaScript library in the main html file.
    - `<script src="libs/angular/angular.js"></script>`
2. Define an application by specifying a value for 'ng-app'
    - `<body ng-app="myApp">`
    - 'ng-app' is a directive of the 'ng' module. This 'ng-app' directive is responsible for automatically setting up your AngularJS application.
    - It defines the DOM bounds of the application, and is usually placed on the <body> or <html> tag.
    - When a browser renders a page, it essentially reads the HTML markup, creates a DOM and broadcasts an event when the DOM is ready. AngularJS listens for that event and as soon as it hears it, it starts traversing the DOM, looking for an ng-app attribute on one of the elements.
    - There can be only one 'ng-app' directive initialized on your app
3. Include a link to your main application module.
    - `<script src="myApp.js"></script>`

```
<!doctype html>

  <body ng-app="myApp">

    <script src="libs/angular/angular.js"></script>

    <script src="myApp.js"></script>

  </body>

</html>
```

Modules

The Angular framework gives us access to a range of methods. The first you will encounter is ‘module’. As mentioned earlier, a module is a container for your code.

We define an Angular module, by providing it a name. The main module of your application must match the value given to the ‘ng-app’ directive in your HTML.

- `angular.module('myApp', []);`

When defining a new module, an array is passed in as a second argument. This is for injecting other modules into our application.

An Angular application may be comprised of 1 or multiple modules. There is always a main module for your application which is specified by ‘ng-app’, however, your application may be separated into additional modules. These sub-modules are made available to your application by ‘injecting’ them into the main module via the second argument (the array).

- `angular.module('myApp', ['secondModule', 'anotherModule']);`

The sub-modules may have dependencies of their own and can be injected in just as with the main module.

- `angular.module('scondModule', ['mapsModule', 'coolModule'])`
  `angular.module('anotherModule', ['myBestModuleEver'])`

To access an existing module in your application, ignore the second argument (meaning the array).

- `angular.module('myApp');`

Module provides an API for using Angular components. This API includes methods for creating controllers, directives, services, filters and configurations.

```
angular.module('myApp').controller(...);
angular.module('myApp').directive(...);

// Services
angular.module('myApp').service(...);
angular.module('myApp').factory(...);
angular.module('myApp').provider(...);
angular.module('myApp').value(...);
angular.module('myApp').constant(...);

angular.module('myApp').filters(...);
angular.module('myApp').config(...);
```

These ‘methods’ on the module are essentially different components you can use. These methods include:

Controllers

- A JavaScript constructor function that is used to augment the Angular Scope
- Used to set up the initial state of the $scope object
- Used to add behavior to the $scope object

Directives

- Directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell Angulars HTML compiler to attach a specified behavior to that DOM element or even transform the DOM element and its children.
- ‘ng-model’ is a directive on the ‘ng’ module

Services

- Angular services are substitutable objects that are wired together
- You can use services to organize and share code across your app
- Services are singletons, which are objects that are instantiated only once per app

- They provide an interface to keep together methods that relate to a specific function

Filters

- A filter formats the value of an expression for display to the user

Configurations

- Used to configure services in the application
- Occurs before Angular compiles the DOM

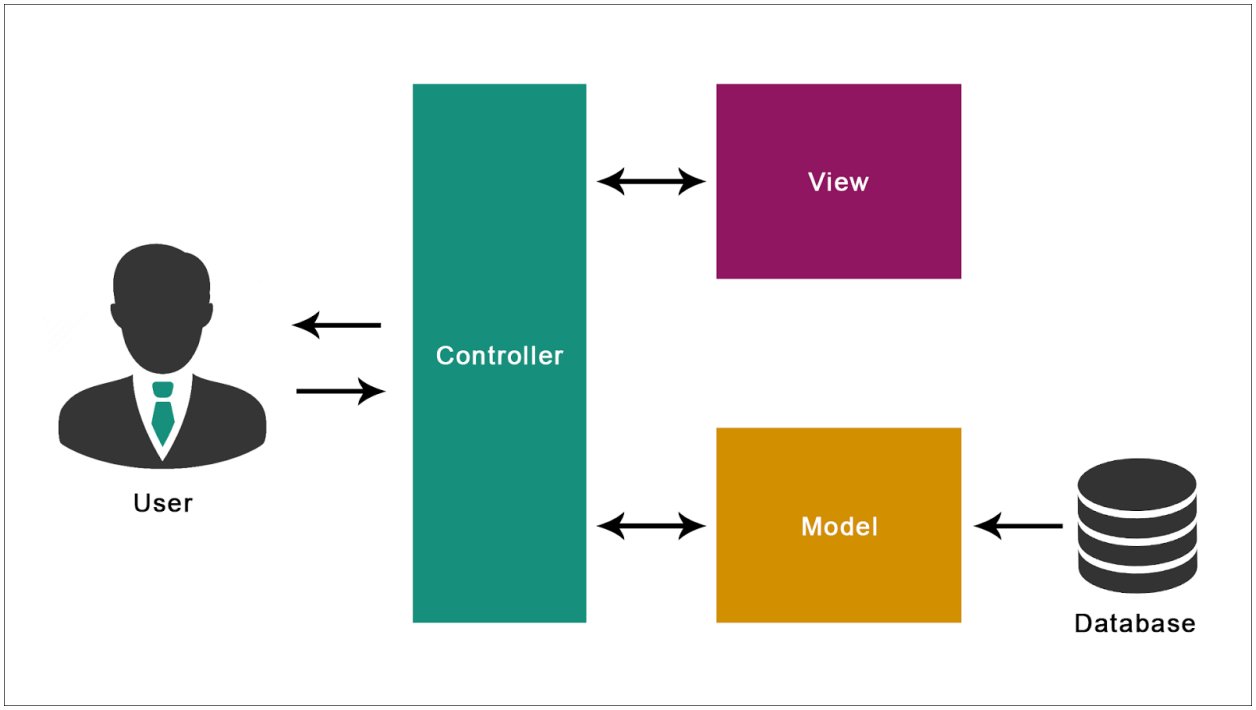
MVC / MVVM

MVC

"MVC is not a design pattern, it is an Architectural pattern that describes a way to structure our application and the responsibilities and interactions for each part in that structure"

An easy way to understand MVC:

- The model is the data
    - Handles data and business logic
- The view is the window on the screen
    - Presents data to the user in any supported format and layout
- The controller is the glue between the Model and the View
    - Receives user requests and calls appropriate resource to carry them out



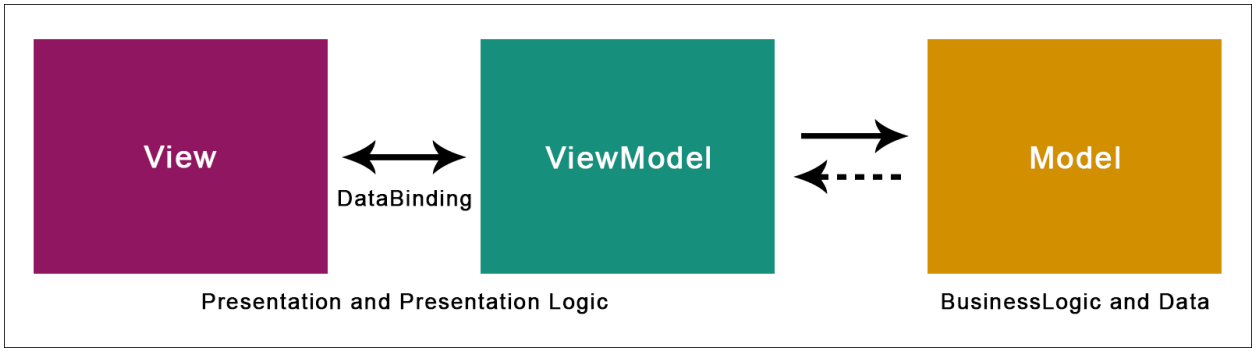
Angular MVC

- **Model**: A model in angular can be a primitive type such as string, number, boolean or a complex type such as object. In short model is just a plain javascript object

- **View**: This is what the users see, in angular the view is the Document Object Model (DOM)
- **Controller**: This is the place where we put our application logic. In angular, controller is simply formed by javascript classes

MVVM

"AngularJS provides Two-Way Data Binding. It is two-way because the data can go from the model to the DOM (view) and from the DOM (view) to the model. If somethings changes in the model, it will be automatically reflected in the DOM. If you interact with the view (e.g. mouse click), the model will be automatically notified."



- **MVVM**: An architectural pattern based on MVC
- **$scope object**: Represents ViewModel
- **$scope**: Allows to define both the data and methods that will be available for the view

Packages / Frameworks

Front Edge Digital uses a number of frameworks and packages to help with development and build processes. These include (but are by no means limited to): Grunt, Bower, Node / NPM and Yoeman.

Grunt

A JavaScript task-runner. It automates processes, such as minification and compiling. Basically, it does the hard work for you.

Bower

Web sites are made of lots of things — frameworks, libraries, assets, utilities. Bower works by fetching and installing packages from all over, taking care of hunting, finding, downloading, and saving the stuff you're looking for. Bower keeps track of these packages in a manifest file, bower.json.

Node

As an asynchronous event driven framework, Node.js is designed to build scalable network applications. We are not necessarily writing Node modules but we quite often use existing modules including some common Grunt Tasks.

NPM

Like Bower, NPM (Node Package Manager), takes care of fetching packages for us but in the form of node modules.

Yeoman

Helps to kickstart new projects, prescribing best practices and tools to help you stay productive. Use the default, community created packages, or customize to create your own.

Controllers

“In Angular, a Controller is a JavaScript constructor function that is used to augment the Angular Scope.”

Defining

The first method we will inspect on an Angular Module is ‘Controller’. A controller is used to augment Scope, therefore, you need to make the Scope object available to the controller. You do this by ‘injecting’ Scope into the controller constructor function.

```
angular.module('myApp')
    .controller('myController', function($scope) {
        $scope.greeting = 'hello, Angular converts!';
    });
```

- We are accessing an existing module, ‘myApp’ (notice the absence of square brackets)
- Define a controller method by passing in a name for your controller, e.g. ‘myController’
- The second argument is the constructor function
    - The $scope object is injected into the controller constructor function
- In the above example, $scope is being augmented with the addition of a ‘greeting’ property

Accessing

Scope is the glue between the DOM and the JavaScript. As controllers are used for binding data to the Scope, it makes sense that we should be able to access Angular controllers from our HTML.

```
<body ng-app="myApp">
    <div ng-controller="myController">
        <p>I just want to say {{greeting}}</p>
```

- We define an Angular application using ‘ng-app=”myApp”’

    - This gives us access to the methods defined on the ‘myApp’ module
- ‘myController’ is a controller of the ‘myApp’ module
- Access the controller using the ‘ng-controller’ directive
- ‘ng-controller’ is a directive of the ‘ng’ module provided as default by the Angular framework
- Pass the name of your controller to the ‘ng-controller’ directive and you will have access to that controllers scope

To print primitive scope properties to the DOM, you can use double curly braces **{{ value }}.**

I just want to say hello, Angular converts!

The ‘ngController’ directive actually creates new scope. Each child scope inherits from it’s parent scope. However a child scope will of course override parent scope variables, which can lead to unintended results and confusions. In order to have direct access to parent scopes, you must use $parent.

```
<body ng-app="myApp">
    <div ng-controller="myController">
        <div ng-controller="yourController">
            <p>I just want to say {{$parent.greeting}}</p>
```

This can get extremely confusing with a deeply nested structure.

```
<div ng-controller="myController">
    <div ng-controller="yourController">
        <div ng-controller="anotherController">
            <p>I just want to say {{$parent.$parent.greeting}}</p>
```

Controller As Syntax

To try and reduce or eliminate scope inheritance issues. Since Angular 1.2, we can use a slightly different approach for accessing controller data.

Instead of augmenting Scope and accessing Scope data, we can assign properties to our controller instances themselves, using ‘this’. This reflects a cleaner, ‘class’ based structure.

```
angular.module('myApp')
    .controller('myController', function() {
        this.greeting = 'hello, Angular converts!';
    });
```

Q. With Scope being the glue between our view and out controller, how do we access data without it?

A. We assign a variable name for the controller and we access data through this variable, or namespace.

[

```
<body ng-app="myApp">
    <div ng-controller="myController as myCtrl">
        <p>I just want to say {{myCtrl.greeting}}</p>
    </div>
```

The above approach assigns a variable name of ‘myCtrl’ to the controller ‘myController’. We can then access ‘greeting’ by referencing ‘myCtrl.greeting’.

Note: We can’t access Scope data using this variable name approach, only data assigned to ‘this’.

This approach has been implemented to move away from Scope. The available information we have about the future release of Angular 2.0 indicates that Scope will be eliminated from the framework, so to assist with the inevitable future migration, we can implement this approach as of minor release 1.2.

Digest Cycle

AngularJS creates a ‘watch’ internally when you create a data binding. A ‘watch’ means that AngularJS ‘watches’ changes to the variable on the $scope object.

At key points in your application AngularJS calls the **$scope.$digest()** function. This function iterates through all watches and checks if any of the watched variables have changed.

Watches are created using the **$scope.$watch()** function. If a watched variable has changed, Angular will update both the view and model to keep them in sync. The $digest() function is what triggers the data binding to update.

Most of the time AngularJS will call the $scope.$watch() and $scope.$digest() functions for you, but in some situations you may have to call them yourself.

```
$scope.$watch('myVar', function(newValue, oldValue) {
    alert('myVar has changed from oldValue to newValue!');
});
```

- ‘myVar’ is the name of a Scope property. The callback function is called when the ‘watched’ Scope property is changed
- You can access the new value and the old value of the watched Scope property

Q. With attempting to eliminate Scope from our application, how do we ‘watch’ properties directly assigned to the controller using ‘this’?

```
angular.module('myApp')
    .controller('myController', function() {
        this.greeting = 'hello, Angular converts!';
    });
```

A. Unfortunately, with the current minor release of Angular, we still need to use Scope to access the $watch function, however, we can bind our controller to the watch and return the property we want to be notified about.

```
app.controller('MainCtrl', function ($scope) {
    this.title = 'Some title';

    $scope.$watch(angular.bind(this, function () {
        return this.title;
    }), function (newVal, oldVal) {
        console.log('Title changed to ' + newVal);
    });
});
```

$Apply

Most of the time Angular will call $scope.$digest() for you. You can also call $scope.$apply() to manually trigger a $diegest cycle.

$Apply is needed when a variable change has been made outside of Angular.

- Timeout Events
- XHR Callbacks
- 3rd Party Libraries, that often use jQuery for DOM manipulation

Dependency Injection

Angulars dependency injection helps in making components reusable, maintainable and testable. Components are given their dependencies instead of hardcoding them within the component.

We have discussed how Modules can be given other Modules as dependencies to build applications:

- `angular.module('myApp', ['secondModule', 'anotherModule']);`

We have also shown you how to inject $scope into your controller (even though we one day plan to eliminate this entirely).

Services are a big part of Angular (we will explain in detail the various service types later) to share data and common functionality. To help you build reusable components and an extremely modular application, Services can be injected into Controllers and other Services.

```
angular.module('myApp')
    .controller('myController', function(dataService) {
        this.values = dataService.data;
    });
```

Here, we are injecting a service called ‘dataService’ into a controller. We can then access methods of this service directly in our controller.

When creating distribution builds of applications, it is best practice to minify your JavaScript, removing unnecessary white spaces, comments and redundant code. Minification scripts also shorten argument names, for example, ‘dataService’ in the above may be shortened to ‘a’:

```
angular.module('myApp')
    .controller('myController', function(a) {
        this.values = a.data;
        // 'a' is undefined
    });
```

‘a’ will be undefined as Angular will unsuccessfully look for a service called ‘a’ in the distribution build of the application. There are ‘Grunt’ tasks that will take care of this issue for you, however, Angular provides you with a way to do ‘explicit dependency injection’.

```
angular.module('myApp')
    .controller('myController', ['dataService', function(dataService) {
        this.values = dataService.data;
    }]);
```

```
angular.module('myApp')
    .controller('myController', ['dataService', function(a) {
        this.values = a.data;
        // 'a' is resolved to dataService
    }]);
```

With this array based syntax we pass an array whose elements consist of a list of names of the dependencies followed by the function itself.

The list of names order in the array, and the order of names injected into the function need to match, meaning if ‘dataService’ is the first name in the list, it needs to be the first argument in the function. If an additional service was to be injected in, the service would be resolved based on the order of injection:

e.g.
Before minification: [‘dataService’, ‘anotherService’, function(dataService, anotherService){}]
After minification: [‘dataService’, ‘anotherService’, function(a, b){}]

When minified, Angular will refer to the list of names to return the name of the service. In the above situation, ‘a’ is resolved to ‘dataService’ and ‘b’ is resolved to ‘anotherService’.

Directives

"Directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell Angular's HTML compiler to attach a specified behavior to that DOM element or even transform the DOM element and its children."

ng Module Directives

You have already been introduced to a few directives made available by Angulars 'ng' Module:

- **ngModel**:
    - The ngModel directive binds an input, select, textarea, (or custom form control) to a property on the scope
- **ngApp**:
    - Responsible for automatically setting up your AngularJS application
    - It defines the DOM bounds of the application, and is usually placed on the <body> or <html> tag
    - There can be only one 'ng-app' directive initialized on your app

Some additional directives of the 'ng' Module you will frequently use are:

- **ngClick**:
    - Binds functionality or expressions to the users click event on that element

```
<button ng-click="clickFn()">Click Me</button>
```

```
$scope.clickFn = function(){
    alert("Who touched Me?");
}
```

    - When the button is clicked, an alert box is displayed in the browser

- **ngRepeat**:
    - Allows you to create a for loop in the HTML binding to scope data

```
$scope.items = [
    {'name': 'Javascript'},
    {'name': 'HTML'},
    {'name': 'CSS'},
    {'name': 'Angular'}
]
```

```
<li ng-repeat="item in items">
    {{item.name}}
</li>
```

    - Results in:
        - Javascript
        - HTML
        - CSS
        - Angular

- **ngShow / ngHide**:
    - Allows you to make elements visible or hidden to the user by evaluating scope variables or functions to true or false

```
$scope.showMe = true;
$scope.hideMe = false;
```

```
<label>Show:</label>
<input type="checkbox" ng-click="showMe = !showMe" checked="checked" />
<span ng-show="showMe">Show when checked</span>

<label>Hide:</label>
<input type="checkbox" ng-click="hideMe = !hideMe" />
<span ng-hide="hideMe">Hide when checked</span>
```

    - ‘ngShow’ will display the span the directive is initialized on if ‘showMe’ resolves to true and hidden when false

- ‘ngHide’ will hide the span the directive is initialized on if ‘hideMe’ resolves to true and is displayed when false
- Both checkboxes toggle the value of ‘showMe’ and ‘hideMe’ respectively when clicked, resulting in the related element being either visible or hidden

- **ngInclude**:

  - Fetches, compiles and includes an external HTML fragment
  - Has a number of methods associated including ‘onload’ allowing you to trigger a function when the template loads

```
<div ng-include="template.url"></div>
```

  - In this situation ‘template’ is a Scope property which in turn has a property for ‘url’. The ‘url’ property is a string path to an HTML file

Custom Directives

The Directive method takes a name followed by a factory function. This factory function should return an object.

```
angular.module('myApp')
    .directive('sayHello', function() {
        return {
            restrict: 'AEC',
            template: '<p>Hello Everyone</p>'
        };
    });
```

The above directive is a directive of the custom built Module ‘myApp’. The directive has a name **‘sayHello’** and returns an object with various options configured.

Restrict

There are many options available for configuration on directives, one of which is **‘restrict’**. Angular initializes directive when it digests the application and finds the directive name on an element in the DOM.

‘restrict’ tells Angular when to initialize the directive, e.g. only initialize it when the Directive name is an attribute of an element, is the element name itself or is a class on the element:

- Attribute: `<div say-hello></div>`
- Element: `<say-hello></say-hello>`
- Class: `<div class="sayHello"></div>`

When a directive is initialized by attribute or element, the directive name is hyphenated in the HTML. This does not affect the name of the directive itself, this will continue to be camel cased.

‘restrict’ helps you setup a development pattern and aligns multiple developers working on the same application. e.g. “Initialize directives by attribute and use our company name to prefix the directive name”.

- `<div company-x-say-hello></div>`

Template

Another directive option is ‘template’. The HTML specified by the ‘template’ replaces the contents (if any) of the container element where the directive was initialized.

- ```
  <div company-x-say-hello>
        This content will be replaced by the template

  ```

If the directive template has more than a single line of HTML, you may consider using the **‘templateUrl’** option instead of ‘template’. This allows you to set a path to an HTML file to be used as the template.

Directive Controllers

Adding business logic to your directive will require you to create a controller function function.

Directives can specify controllers using the unsurprisingly named controller option. Just like ngController, this option attaches a controller to the template of the directive.

```
angular.module('myApp')
    .directive('sayHello', function() {
        return {
            restrict: 'AEC',
            template: ' Hello {{name}} ',

            controller: function($scope){
                $scope.name = "Everyone";
            }

        };
    });
```

We can also assign properties to the controller itself and access them by a controller variable name using the **‘controllerAs’** option.
```

```
angular.module('myApp')
    .directive('sayHello', function() {
        return {
            restrict: 'AEC',
            template: '<p>Hello {{sayHelloCtrl.name}}</p>',

            controller: function(){
                this.name = "Everyone";
            },
            controllerAs: 'sayHelloCtrl'

        };
    });
```

Directives have an option for 'scope'. No scope declaration (**scope: false** by default) means the directive will share the scope of its parent. All other components that share this scope will see any properties / changes made to the scope in your directive.

```
.directive('sayHello', function() {
    return {
        template: '<p>{{greeting}} {{name}}</p>',
        controller: function($scope){
            $scope.name = "Everyone";
        }
    }
});
```

This can prove to be hard to manage as your application grows. Scope properties might end up being set and modified in many different files and in many different controllers.

Setting the scope option to true (**scope: true**) creates a child scope for this directive. This directive's child scope will inherit from the parent scope just like the child scope of a ngController. Changes to Scope properties in the parent will be visible to your directive and children of your directive.

Setting the scope option to an empty object (**scope: {}**) creates an isolated child scope. This scope will not inherit from it's parent. Often this is beneficial so that you can have isolated modular components which have no dependencies.

```
.directive('sayHello', function() {
    return {

        scope: {},

        template: '<p>Hello {{name}}</p>',
        controller: function($scope){
            $scope.name = "Everyone";
        }
    }
});
```

Quite often however, you will want to access some properties of the parent Scope without having access to its entirety. We can add properties into this isolated scope object in order to pass data from the parent to the directive scope, which we will cover next.

@ For One-Way Binding

We can pass primitive values through to an isolated Scope using the '@' symbol. The below example has a Scope property 'greeting' set in the parent controller. By assigning that Scope property to an attribute on the directive element, we can access the value in our isolated Scope by referencing the attribute name followed by the '@' symbol in our scope option.

```
.controller('myController',['$scope', function($scope) {
    $scope.greeting = "Welcome";
}]);
```

```
<div ng-controller="myController">
    <say-hello greeting="{{greeting}}"></say-hello>
</div>
```

```
.directive('sayHello', function() {
    return {
        scope: {
            greeting: '@'
        },
        template: '<p>{{greeting}} {{name}}</p>',
        controller: function($scope){
            $scope.name = "Everyone";
        }
```

This only allows one-way data binding, meaning the directive will pick-up any changes to property in the parent Scope but changes to the property in the directive will not reflect in the parent Scope.

= For Two-Way Binding

The '=' symbol allows you to create a two-way data-binding with the parent. Any changes to 'greeting' on the directive Scope will be reflected in the parent.

```
.directive('sayGreeting', function() {
    return {
        scope: {
            greeting: '='
        },
        template: '<input type="text" ng-model="greeting" />'
```

& TO EXECUTE FUNCTIONS IN THE PARENT SCOPE

The '&' symbol allows you to bind functionality set on the parent Scope to your isolated directive Scope. In the below example. 'doGreeting' is a function set in the parent controller. If we trigger that function from the directive, the parent function will be called.

```
<div ng-controller="myController">
    <say-hi do-greeting="doGreeting()"></say-hi>
</div>
```

```
.directive('sayHi', function() {
    return {
        scope: {
            doGreeting: '&'
        },
        template: '<a ng-click="doGreeting()">Say Hi</a>',
```

Passing through data with ControllerAs Syntax

With the current minor release of Angular 1.3, we still need to use the directive scope option to pass through data into our directive, so eliminating scope from our application is not quite there yet, however, each minor release enables us to move towards this with new features and functionality.

For now, we pass data through using the scope option but we can bind this to our controller using the 'bindToController' option.

```
.directive('parentHi', function() {
    return {
        scope: {
            parentString: '@'
        },
        controller: function () {},
        controllerAs: 'ctrl',
        bindToController: true,
        template: '<span>{{ctrl.parentString}}</span>'
```

The above example adds the 'parentString' property to out controller and can then be accessed via the controller variable name.

Directive Link Function

Adding functionality to your directive will require you to create a link function.

The compile function is called once for each references to a given directive. The compilation process happens in two phases.

1. **Compile**: traverse the DOM and collect all of the directives. The result is a linking function.
2. **Link**: combine the directives with a scope and produce a live view.

The ngRepeat directive for instance will have to look up the element it is attached to, extract the html fragment that it is attached to and create a template function. You pass data to the template function and the return value of that function is the html with the data in the right place. The compilation phase is that step in Angular which returns the template function.

After the compilation phase is complete, the directive link function is called. As this happens after the template has been linked with data and inserted into the DOM, the link function has access to both the element where the directive was initialized and the attributes on that directive. Perhaps the link function would be better named as ‘post-link’ function?

```
angular.module('myApp')
    .directive('sayHello', function() {
        return {
            restrict: 'AEC',
            template: '<p>Hello {{sayHelloCtrl.name}}</p>',
            controller: function(){
                this.name = "Everyone";
            },
            controllerAs: 'sayHelloCtrl',
            link: function(scope, element, attrs){

            }
        };
    });
```

- **Scope**: is an Angular scope object
- **Element**: is the jqLite-wrapped element that this directive matches
- **Attrs**: is a hash object with key-value pairs of normalized attribute names and their corresponding attribute values

```
<link-example color="red">Click Me!</link-example>
```

```
myApp.directive('linkExample', function () {
    return {
        link: function ($scope, element, attrs) {
            element.bind('click', function () {
                element.html('You clicked me!');
            });
            element.bind('mouseenter', function () {
                element.css({'color': 'black',
                             'background-color': 'yellow'});
            });
            element.bind('mouseleave', function () {
                element.css({'color': attrs.color,
                             'background-color': 'white'});
            });;
```

The above example binds functionality to the element where the directive was initialized. The ‘mouseleave’ example uses the attribute ‘color’ defined on the element to assign a color value to the element when the user hovers off the element.

If a new $scope is created in the controller function, the link function can access the same scope and it properties and methods, however, Scope variables set in the link function are not accessible in the controller function due to the link function happening after the controller function has run.

```
<link-ctrl-example>Click Me!</link-ctrl-example>
```

```
myApp.directive('linkCtrlExample', function () {
    return {
        controller:function($scope){
            $scope.newHtml = 'text from controller';
        },
        link: function (scope, element, attrs) {
            element.bind('click', function () {
                element.html(scope.newHtml);
            });
        }
    };
});
```

We can eliminate Scope from this process with a fourth argument in the link function allowing us to access the directive controller.

```
myApp.directive('linkCtrlExample', function () {
    return {
        controller:function(){
            this.newHtml = 'text from controller';
        },
        link: function (scope, element, attrs, ctrl) {
            element.bind('click', function () {
                element.html(ctrl.newHtml);
            });
        }
    };
});
```

Require Option

You will find situations where you will have a directive inside of a directive. The directive option **'require'** allows you to access the parent directive controller from the child directive link function.

The example below illustrates accessing the parent controller using 'require', however, when you require a parent controller, the fourth argument of the link function is how you access that controller.

In this situation, when you want to access both the parent directive controller and the current directive controller, you can access the current directive controller through the element, passing in the controller name to the **element.controller** method.

```
<parent-ctrl-example>
    <child-ctrl-example>Click Me!</child-ctrl-example>
</parent-ctrl-example>
```

```
myApp.directive('parentCtrlExample', function () {
    return {
        controller:function(){
            this.parentText = 'I am your Father';
```

```
myApp.directive('childCtrlExample', function () {
    return {
        require: 'parentCtrlExample',
        controller:function(){
            this.childText = 'Who are you?';
        },
        link: function ($scope, element, attrs, ctrl) {
            // ctrl is the parent controller
            // element.controller('childCtrlExample')
                //is the current controller
        }
```

Transclusion

When a directive has a template, the contents of the element where the directive was initialized will be replaced by the directive template.

There are often situations where you want to retain the contents of the directive element but also provide a template for the directive.

The first code block below shows a directive being initialized inside of another directive. The parent directive has a template but we don't want to replace the child directive. The middle code block below is the template of the parent directive.

Setting the directive **'transclude'** option to true, we essentially have access to the element contents which can then be placed within the parent directive template.

You specify where you would like the HTML contents to be placed within the parent template by using the **'ngTranslude'** directive. In the below setup, the 'childDirective' will be initialized in the 'contentSection' div.

```
<directive-example>
    <child-directive></child-directive>
</directive-example>
```

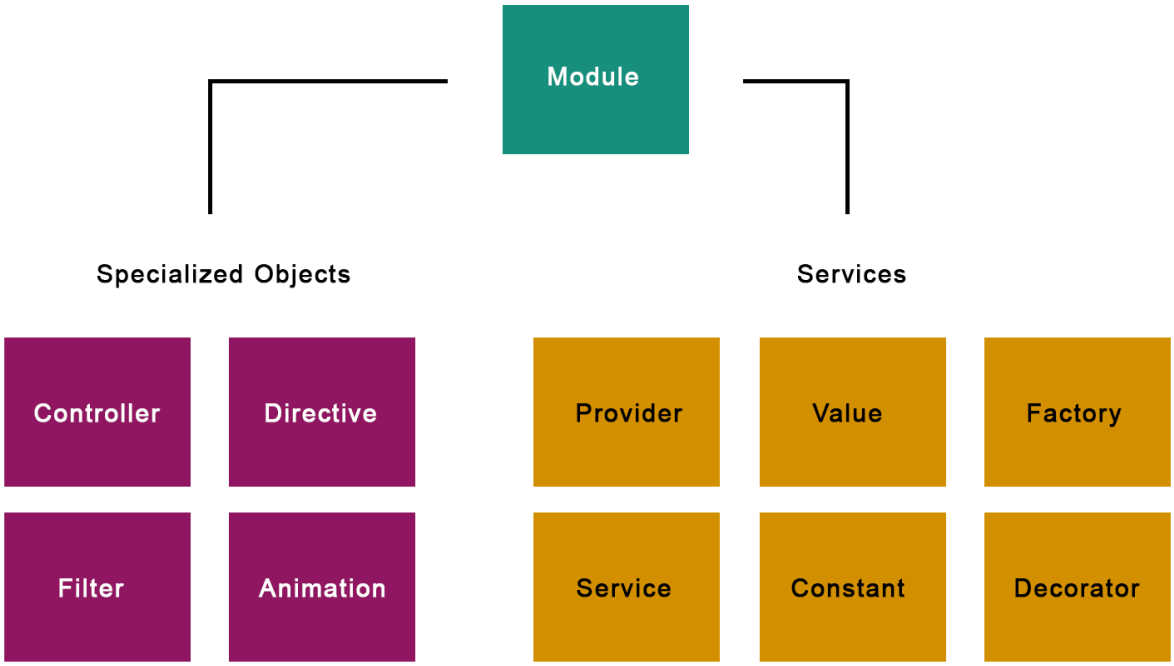
```
<div class="container">
    <p>This is some parent directive content</p>
    <div class="contentSection" ng-transclude=""></div>
</div>
```

```
myApp.directive('directiveExample', function () {
    return {
        transclude: true,
        templateUrl: 'path_to_file.html'
    }
});
```

Services

Specialized Objects & Services

An Angular application is comprised of both ‘Specialized Objects’ and ‘Services’.



Specialized Objects:

- Conform to a specific Angular framework API
- You have been introduced to a number of specialized objects already including ‘Controllers’ and ‘Directives’
- There are also Filters and Animations

Services:

- Services are objects whose API is defined by the developer writing the service
- Use services to organize and share code across your application
- Services are a huge part of Angular and provide the means to make a modular application

Injectors / Providers

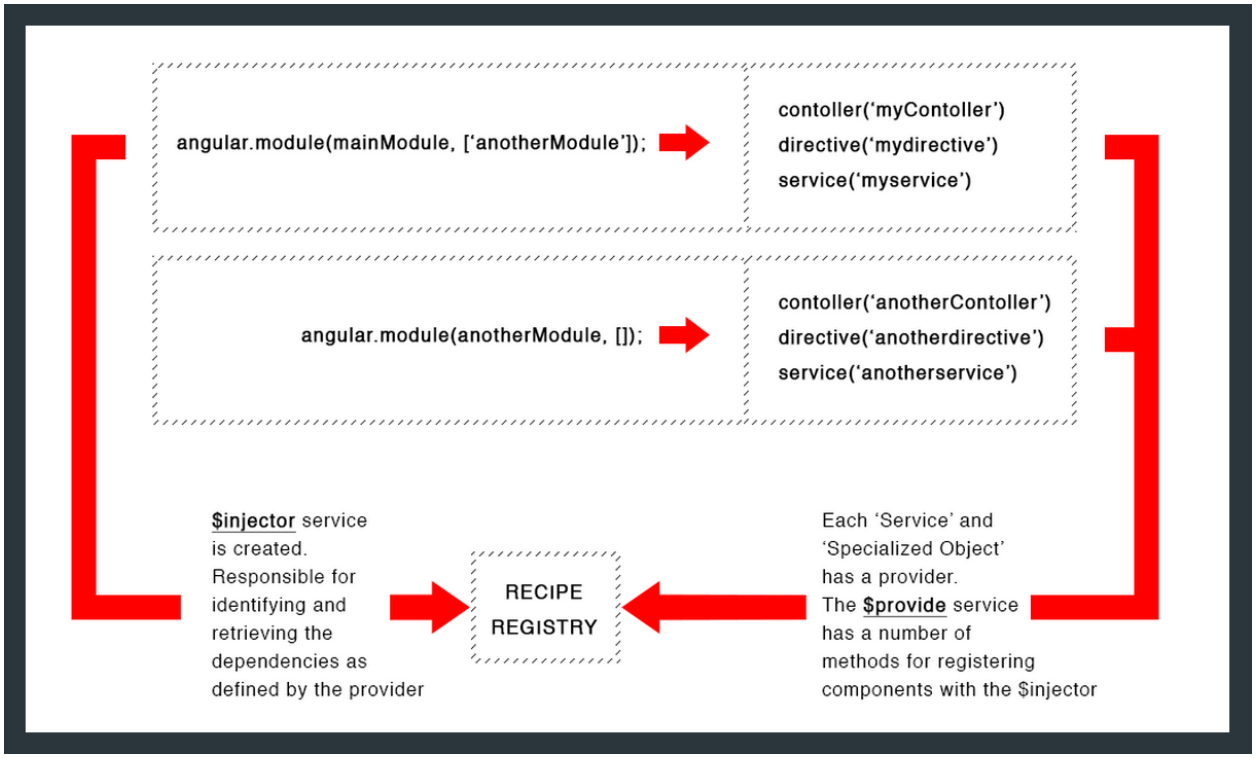
Specialized objects and services have something in common. They each have a recipe to instruct Angular how to construct them.

Each web application you build is composed of objects that collaborate to get stuff done. These objects need to be instantiated and wired together for the app to work.

In Angular apps most of these objects are instantiated and wired together automatically by the injector service. The injector creates two types of objects, services and specialized objects The injector needs to know how to create these objects.

In order for the injector to know how to create and wire together all of these objects, it needs a registry of "recipes". Each recipe has an identifier of the object and the description of how to create it.

The six types of service recipes are Providers, Values, Factories, Services, Constants and Decorators. It is easier to continue to describe these ‘service’ types as ‘recipes’ (in this document) as there is a service type called ‘service’ which would be confusing and drawn out to explain each time I reference it. In addition, a Provider creates an injectable service which may be confusing if I refer to the provider as a service.



Provider

The Provider is the core recipe type. All the other recipe types are just syntactic sugar on top of it. It is the most verbose recipe with the most abilities, but for most services it's overkill. You should use the Provider recipe only when you want to expose an API for application-wide configuration that must be made before the application starts. Providers are one of the two recipe types that can be injected into a config function.

Along with the actual provider functions which can be configured, the provider creates a service that can be injected into controllers and other services. The created service is defined by the $get.

```
myApp.provider('movie', function () {
    var movieTitle;
    return {
        setMovie: function (value) {
            movieTitle = value;
        },
        $get: function () {
            return {
                title: movieTitle
            }
        }
    }
});
```

The $get function is the service definition available to controllers and other recipes.

```
myApp.config(function (movieProvider) {
    movieProvider.setMovie('Point Break');
});
```

```
myApp.controller('ctrl', function (movie) {
    this.movie = movie.title;
});
```

```
<div ng-controller="ctrl as ctrl">
    <p>{{ctrl.movie}} is one of those classics we all know and love</p>
<<div>
```

When a provider is injected into a config function, we append “Provider” to the end of its name. e.g. In the above, the Provider ‘movie’ is injected into the config function as ‘movieProvider’.

The Provider service as defined by the $get is what is injected into the controller, so we only have access to whatever the $get gives us access to. e.g. ‘title’.

In the above controller, we do not have access to the ‘setMovie’ function as it is outside of the $get.

Value

The Value recipe is as simple as it gets. Use it to store a simple value whether it a string, number, date-time, array, object or function.

This recipe type is not configurable in the config function. We construct it by giving it a name, (e.g. publicToken) and a ‘value’ (e.g. ‘12345’). We access the value by passing in the recipe name to our controller or service.

```
myApp.value('publicToken', '12345');
```

```
myApp.controller('DemoCtrl', function (publicToken) {
    this.publicToken = publicToken;
});
```

```
<div ng-controller="DemoCtrl as demo">
    <p>Public Token: {{demo.publicToken}}</p>
</div>
```

Constant

Values and Constants are much the same in terms of how they are structured. We use constants to create default configurations for directives, because we can modify those configuration on our config functions. The only other recipe type that can be injected into a config function.

Data formats can include 'string', 'number', 'date-time', 'array', 'object'. Constants remain constant so we do not update or change them 'outside of the configuration function'.

```
myApp.constant('publicToken', '12345');
```

Service & Factory

I decided to lump Services and Factories together because there is often confusion about the difference and when to use one over the other.

First of all, you will most often see Factories and services CRUD data from Restful services, however they are extremely useful for sharing code around the application rather than hard-coding in multiple controllers.

Factory:

```
myApp.factory('userToken', function() {
    return {
        getUserToken: function(userId) {
            return "12345_" + userId;
        };
    }
});
```

Service:

```
myApp.service('userToken', function() {
    this.getUserToken = function(userId) {
        return "12345_" + userId;
    };
});
```

These examples are very basic and don't represent real life scenarios but I want to show you the differences between factories and services without over complicating the functionality.

Both of these provide the same function of 'getUserToken' which returns '12345' plus whatever is passed into the function.

You'll notice one major difference between the service recipe and the factory recipe: the use of the keyword ‘this’. The reason is that when a service recipe is injected, it will be instantiated using the ‘new’ keyword.

Simply put: A factory returns whatever you would like while a service returns a constructor function. Therefore, a factory can return objects or non-objects, while a service will only ever return an object created by the constructor function.

Using one or the other (service or factory) is about code style. But, the common way in AngularJS is to use a factory. Another common suggestion is to use a factory when you have complex logic and you don't want expose this complexity, by using private variables.

Using either a factory or a service will create an application-wide singleton. Meaning anytime you use that service/factory throughout the application, you are using the exact same instance of that class, so it is the same object you are calling.

```
myApp.controller('UserTokenCtrl', function (userToken) {
    this.userToken = userToken.getUserToken();
});
```

Factories and services are injected in just like any of the other recipe types. They are used the same way by calling the exposed functions.

Decorator

We can decorate or add functionality to a recipe without the original recipe or the constructor function where the recipe is being injected knowing that the decorating is happening.

This comes in really handy when we are using 3rd party recipes and we don’t want to make a copy or modify that recipe directly. Instead, we can just decorate it with our custom functionality.

Use the decorator method on Angulars $provide service. Pass in the name of the 3rd party recipe we want to ‘decorate’ and use the delegate service (the original recipe instance) to attach custom functionality.

You can then call both the original recipe methods and the decorated method as if they were always available in the original.

You will see the major benefits of this when there is a new release version of the 3rd party recipe and you upgrade without any issues whilst retaining the decorator methods you assigned.

```
myApp.config(function($provide) {
    $provide.decorator('thirdPartyService',
        function($delegate) {
            $delegate.greet = function() {
                return "Hi, I am a new function of 'thirdPartyService'";
            };
            return $delegate;
        });
});
```

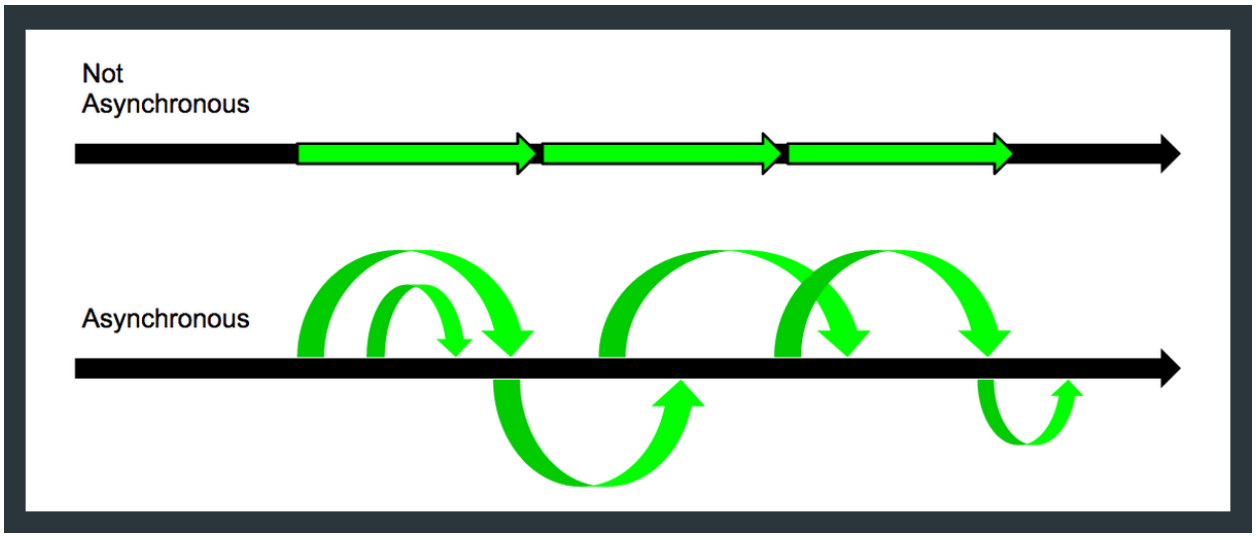
```
myApp.controller('DemoCtrl', function(thirdPartyService) {
    this.greet = thirdPartyService.greet();
});
```

Asynchronous

In computer programming, asynchronous events are those occurring independently of the main program flow.

JavaScript is a single-threaded language meaning events are placed in a queue and are processed one after the other.

Asynchronous (async) behavior means being able to do multiple things at the same time without locking the main thread. In node and JavaScript in general, this applies particularly to ajax file requests. Interaction fires off several async calls to retrieve data, and the main thread will not be locked while it renders the content.



$HTTP Service

"The $http service is a core Angular service that facilitates communication with the remote HTTP servers."

The $http service is used to generate an HTTP request and returns a promise with two $http specific methods: success and error. A promise represents the eventual result of an asynchronous operation and is called asynchronously.

The below example makes a call to '/someUrl' to GET data. If the request is successful, the 'success' function will be called. On error, the 'error function will be called.

```
$http.get('/someUrl')
    .success(function(data, status, headers, config) {
        // this callback will be called asynchronously
        // when the response is available
    })
    .error(function(data, status, headers, config) {
        // called asynchronously if an error occurs
        // or server returns response with an error status.
    });
```

Injecting the $http service into a Factory function allows us to create a reusable server request for data.

```
myApp.factory('userToken', function($http) {
    return {
        getUserToken: function(userId) {
            $http.get('/someUrl/' + userId)
                .success(function(data, status, headers, config) {
                    return data;
                })
                .error(function(data, status, headers, config) {
                    // error
                });
        };
    }
});
```

Although the above use case would work and return data, it is unlikely we would want to use this setup as our Controller where the Factory is injected would not be aware of when the asynchronous event is complete.

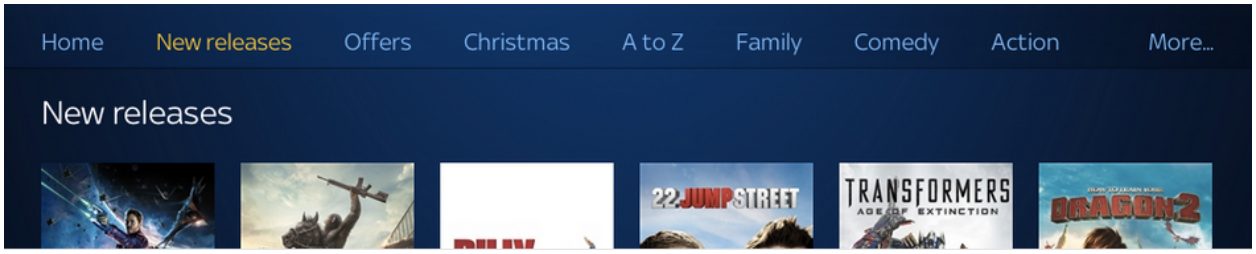
The below setup is more likely to be an approach you will take, where the Factory returns the promise. We can then listen for the success and error promises in our Controller and do something with the returned data.

```
myApp.factory('userFactory', function($http) {
    return {
        getUserToken: function(userId) {
            return $http.get('/someUrl/' + userId);
        };
    }
});
```

```
myApp.controller('UserCtrl', function(userFactory) {
    userFactory.getUserToken('007')
        .success(function(data, status, headers, config) {
            this.userToken = data;
        })
        .error(function(data, status, headers, config) {
            // error
        });
})
```

Angular Asynchronous Workflow

The below screenshot is from the 'Sky Store' website and although my example code may not be their exact implementation, I want to show you how we can navigate between movies in this single page application.



The HTML:

- Initializes a controller inline using the 'ngController' directive
    - Create a variable name for our controller using the 'controller as' syntax
- Use 'ngRepeat' to generate the navigation
- Each navigation link uses the 'ngClick' directive to call a function in the Controller, passing through the category id as an argument

- The ‘movieList’ div is a container for a movie banner (to avoid code bloat, I have not shown this code). This would be repeated based on the length of ‘mvCtrl.movieList’

```
<div ng-controller="moviesCtrl as mvCtrl">
    <div class="nav" ng-repeat="category in mvCtrl.movieCategories">
        <a ng-click="mvCtrl.getMovies(category.id)">{{category.title}}</a>
    </div>
    <div class="movieList" ng-repeat="movie in mvCtrl.movieList">
        // Create Movie Banners
    </div>
 </div>
```

The Controller:

- We assign properties and functions to the controller (not Scope)
    - By using a variable ‘vm’ for ‘this’, we avoid issues with knowing what ‘this’ is. For example, when we are inside of a promise, ‘this’ may refer to the promise itself and not the controller
- ‘vm.movieCategories’ is the array of objects used for populating the navigation
- ‘vm.getMovies’ is the function call assigned to ‘ngClick’ in our navigation. This accepts a single argument (the category id) and calls the ‘getMovies’ function of ‘movieService’
- If the promise is successful, the ‘success’ callback is triggered and sets a ‘movieList’ property on the controller. This data would be used to generate the movie banners

```
myApp.controller('moviesCtrl', function(movieService) {
    var vm = this;
    vm.movieCategories = [{title: 'New releases', id: 0},
                          {title: 'Offers', id: 1},
                          {title: 'Christmas', id: 2},
                          {...}];
    vm.getMovies = function(catId){
        movieService.getMovies(catId)
            .success(function(response){
                vm.movieList = response.data;
            })
            .error(function(err){
                // error
            });
    }
});
```

The Service:

- This service is very simplistic
- Angulars $http service is injected
- The ‘getMovies’ function returns a promise which will trigger the callbacks in the controller

```
myApp.factory('movieService', function($http) {
    return {
        getMovies: function(catId) {
            return $http.get('/movies/' + catId);
        }
    }
});
```

The Workflow:

- The user clicks on a navigation link
- The ‘getMovies’ function is called in the controller, passing through the category id
- The ‘getMovies’ function in the controller calls the ‘getMovies’ function in the ‘movieService’
- The Controller listens for the promise return and triggers either the success or error callback
- The controller ‘movieList’ property is updated if the promise is successful
- The ‘movieList’ ‘ngRepeat’ re-renders with the updated data

$Q Service

In some situations you will be using an API where you can't take advantage of the $http service with success and error callbacks. The $http API is based on the 'deferred promise' API exposed by the $q service. $q is a service that helps you run functions asynchronously.

```
myApp.factory('greetingService', function($q, $timeout) {
    return {
        getGreeting = function() {
            var deferred = $q.defer();
            $timeout(function() {
                deferred.resolve(['Hello World!']);
            }, 2000);
            return deferred.promise;
        };
    };
});
```

```
greetingService.getGreeting()
    .then(function(response){
        // do something with data
    }, function(error){
        // do something with error
    });
```

The $q service has a defer method. You can defer a variable which will only be returned when it is resolved or rejected. It can also provide updates on the status of the promise's execution.

$http is sugar on top of the $q service and exposes a success and error callback whereas $q has a 'then method. The then method has three (optional) function parameters, 'successCallback', 'errorCallback', 'notifyCallback'. Progress / notify callbacks are not currently supported via the ES6-style interface so we will focus on the success and error functions.

The onSuccess function is called whenever the asynchronous tasks ends successfully and the onFailure function is called if the tasks fails to execute. For each promise only one of the functions can ever be called. The task either succeeds or it fails.

The above example looks much the same as the previous $http example with the replacement of $q being injected instead of $http. I have also injected the $timeout service

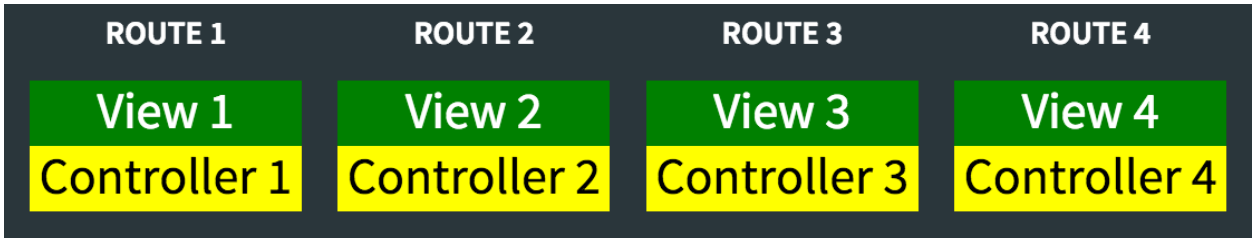
into the example to mimic time lag, e.g. server request (although, I would hope this wouldn't take 2 seconds).

We defer a variable 'deferred' and can either resolve or reject it. In this example we resolve the variable to 'Hello World!'. The deferred promise is only returned when the deferred object is resolved or rejected.

The second code block is the service call in a Controller. Instead of 'success' or 'error', we use 'then', with the first function parameter being 'success' where we can do something with the returned data. The second function parameter is the error callback, if the deferred object is rejected. In the error callback you could alert the user of the unsuccessful service call.

Routing

Routing helps you in dividing your application in logical views and bind different views to Controllers. Routing helps us build single page applications, switching content based on the current route



Routing is not included in the standard angular javascript package so you need to include the script file for the **'ngRoute'** Module (angular-route.js).

```
<body ng-app="myApp">

    <div ng-view></div>

    <script src="angular.js"></script>
    <script src="angular-route.js"></script>

</body>
```

'ngView' is a directive in the 'ngRoute' Module. This is the container for the current route view. The view contains HTML related to whichever route the user is on.

As **'ngRoute'** is a module, to use this in your app, you need to inject it in as a Module dependency.

```
angular.module("myApp", ['ngRoute']);
```

'$routeProvider' is a provider in the **'ngRoute'** module. Providers and constants are the only service recipe types that can be injected into an application config function.

The **config** method gets executed during the provider registrations and configuration phase (early stages of app bootstrapping).

```
myApp.config(function($routeProvider){
    $routeProvider
        .when('/me', {
            templateUrl: 'partials/me.html',
            controller: 'MyCtrl'
        })
        .when('/you/:userId', {
            templateUrl: 'partials/user-detail.html',
            controller: 'UserDetailCtrl'
        })
        .otherwise({ redirectTo: '/me' });
});
```

$routeProvider's API provides the ability to define routes. A Route is used for deep-linking URLs to controllers and views (HTML partials). Routes are specified by the ‘when’ method.

$routeProvider watches $location.url() and tries to map the path to an existing route definition. In the above example, when we are at url ‘myApp.com/me’, the HTML from me.html will be inserted into the ng-view div and will be using the controller ‘MyCtrl’.

when(path, route);

- path = '/me' …. (url: myApp.com/me)
- route = { templateUrl: 'partials/me.html', controller: 'MyCtrl'}

You can see this is much like a switch statement. If none of the paths are matched, it will default to the route specified by ‘otherwise’.

Parameters are defined starting with a colon. In the above example, when we navigate to ‘/you/’, Angular will extract from the URL (route path) whatever comes after the ‘you/’.

```
.when('/you/:userId', {
    templateUrl: 'partials/user-detail.html',
    controller: 'UserDetailCtrl'
});
```

```
.controller('UserDetailCtrl', function($routeParams){
    this.userId = $routeParams.userId;
});
```

The controller function can get access to route parameters via the **'$routeParams'** service, referencing the parameter name defined in the route, e.g. 'userId'.

Route Properties

Each route has access to the following properties:

- controller
- controllerAs
- template
- templateUrl
- resolve
- redirectTo
- reloadOnSearch
- caseInsensitiveMatch

UI-Router

At Front Edge Digital, we opt to use the **'ui.router'** Module made available to us by including the file **'angular-ui-router.js'**. You do not need to include the 'ngRoute' Module when using 'ui.router'.

```
<body ng-app="myApp">

    <div ui-view></div>

    <script src="angular.js"></script>
    <script src="angular-ui-router.js"></script>

</body>
```

'uiView' is a directive in the 'ui.router' Module. This is the container for the current state view. Unlike the $route service in the Angular 'ngRoute' Module, which is organized around URL routes, 'ui.router' is organized around states.

```
angular.module("myApp", ['ui.router']);
```

As **'ui.router'** is a module, to use this in your app, you need to inject it in as a Module dependency.

'$stateProvider' is a provider in the **'ui.router'** module. Providers and constants are the only service recipe types that can be injected into an application config function.

The state based approach lets you navigate states by state name in addition to url, below we have a state name 'me', which has a url of '/me'.

Just like 'ngRoute', each state has a view and a controller. Parameters are also defined starting with a colon but are accessed via the $stateParams service in the controller.

```
myApp.config(function($stateProvider){
    $stateProvider
        .state('me', {
            url: '/me',
            templateUrl: 'partials/me.html',
            controller: 'MyCtrl'
        })
        .state('you', {
            url: '/you/:userId', // or you/{userId}
            templateUrl: 'partials/user.html',
            controller: 'UserCtrl'
        });
```

```
.controller("UserCtrl", function($stateParams){
    var params = $stateParams;
});
```

Navigating Between States

Here are two examples for changing state by state name: ui-sref in the DOM and the **'$state'** service in the controller.

```
<body ng-app="myApp">

    <div ui-view></div>

    <a ui-sref="me">My State</a>
    <a ui-sref="you({userId: user})">Your State</a>

    <script src="angular.js"></script>
    <script src="angular-ui-router.js"></script>

</body>
```

```
.controller("me", function($state){
    this.goToProfile = function(user){
        $state.go('you', {userId: user});
    }
});
```

‘ui-sref’ acts similar to an ‘href’ on an anchor tag, however, it allows us to navigate to states via their name. In addition, we can pass parameters through as defined by the state.

The injected **‘$state’** service provides a means of navigation from within a Controller, again, with the option of passing parameters.

Child States

‘ui.router’ allows for child states. Child states are indicated using the dot syntax, e.g. ‘you.details’ with ‘details’ being a child state of ‘you’.

You can have grandchild states, great-grandchild states and so on. The url is concatenated from parent state to child state and so on, e.g. ‘/you/userId/details’. If using Scope, Controllers in child states have access to parent Scopes via the ‘$parent’ method on the Scope object.

```
.state('you', {
    url: '/you/:userId',
    templateUrl: 'partials/user.html',
    controller: 'UserCtrl'
})
.state('you.details', {
    url: '/details',
    templateUrl: 'partials/user-Details.html',
    controller: 'UserDetailCtrl'
})
// you.details URL will be 'myapp.com/you/jimBob/details'
```

```
.controller("indexCtrl", function($state){
    this.goToUserDetails = function(user){
        $state.go('you.details', {userId: user});
    }
}};
```

You can set and retrieve parent parameters with ui-router. In the code above, we are navigating to the ‘you.details’ state passing in a userId. The parent state specifies the parameter ‘userId’ but as the child states concatenate the url, it is available in both states.

Any state that has a child state must have a ui-view directive initialized in the HTML. In the above example, user.html must initialize a ui-view directive for the child view to be rendered.

Resolves

Resolve is a property on the state configuration, and each property on resolve can be an injectable function (meaning it can ask for service dependencies). The function should return a promise.

```
myApp.config(function($stateProvider){
    $stateProvider
        .state('me', {
            url: '/me',
            templateUrl: 'partials/me.html',
            controller: 'MyCtrl',
            resolve: {
                message: function(messageService){
                    return messageService.getMessage();
                }
            }
        });
```

```
myApp.controller("messageController", function (message) {
    this.message = message;
});
```

When the promise completes successfully, the resolve property (‘message’ in the above scenario) is available to inject into the state controller function. In other words, all a controller needs to do to grab data gathered during a resolve is to ask for the data using the same name as the resolve property (message).

This means you can wait for data to become available before showing a view, and simplify the initialization of the model inside a controller because the initial data is given to the controller instead of the controller needing to go out and fetch it.

State Properties

Each state has access to the following properties:

- controller
- controllerAs
- template
- templateUrl
- resolve

- url
- views
- abstract
- onEnter
- onExit
- reloadOnSearch
- data
- params

We have talked about a few of these and could spend a lot longer on each property individually but I would suggest researching the available functionality of ui.router independently as there is no better way to learn than to experiment on your own.

Multiple Views

Naming your ui-views can allow you to add multiple views to your page. This can be very powerful if you have a widget based interface. e.g. a grid, a tree and a form perhaps. Each widget can be a stand-alone component with multiple views providing a way to isolate while still accessing convenience options of the '$stateProvider'.

```
<body>
    <div ui-view="viewA"></div>
    <div ui-view="viewB"></div>
</body>
```

```
.state('route1', {
    url: "/route1",
    views: {
        "viewA": {
            templateUrl: "viewA.html",
            controller: "viewAController"
        },
        "viewB": {
            templateUrl: "viewB.html",
            controller: "viewBController"
        }
    }
});
```

Filters

A filter:

- formats the value of an expression for display to the user
- lets you organize data
- or displays data only if it meets certain criteria

Below, we have our data model. An array of guitars with name and price.

```
$scope.guitars = [{
    "name": "Acoustic Guitar",
    "price": 230.78
},{
    "name": "Bass Guitar",
    "price": 162.55
},{
    "name": "Electric Guitar",
    "price": 184.34
}]
```

We display the guitars in a list using the 'ngRepeat' directive.

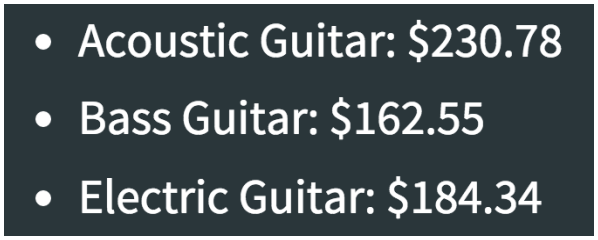
```
<li ng-repeat="guitar in guitars | orderBy:'price'">
    {{guitar.name}}: {{guitar.price | currency}}
</li>
```

To use Angular filters in our HTML, we specify the filter we want to use by inserting a pipe symbol after the content we wish to filter. In the above example we are using multiple filters:

- orderBy
- Currency

'orderBy' takes an argument detailing how you want to order the repeating content. Arguments are specified using a colon after the filter name. Multiple arguments are possible by continually using a colon and an argument name, e.g. filterName: 'argument': 'argument'.

In the above example, we are ordering by ‘price’. This will order the list with the cheapest guitar at the top and most expensive at the bottom.



As you can see, the ‘currency’ filter is adding a dollar sign in front of the number for us.

Commonly used Angular Filters

- **currency**: Formats a number as a currency (ie $1,234.56). When no currency symbol is provided, the default symbol for current locale is used.
- **number**: Formats a number as text
- **date**: Formats date to a string based on the requested format
- **json**: Allows you to convert a JavaScript object into JSON string
- **lowercase**: Converts string to lowercase
- **uppercase**: Converts string to uppercase
- **limitTo**: Creates a new array or string containing only a specified number of elements
- **orderBy**: Orders a specified array by the expression predicate
- **filter**: Selects a subset of items from array and returns it as a new array

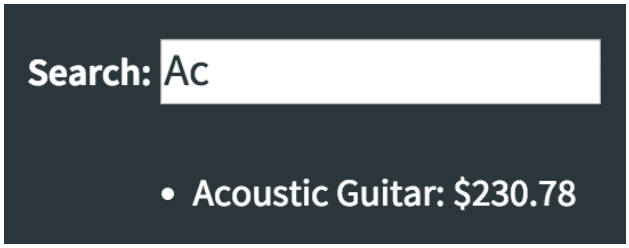
The last one in the list ‘filter’, may be slightly confusing so we have provided the below example:

```
Search: <input ng-model= "searchText" />
```

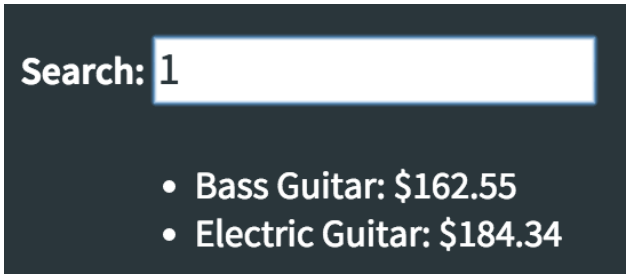
```
<ul>
    <li ng-repeat="guitar in guitars | filter:searchText">
        {{guitar.name}}: {{guitar.price | currency}}
    </li>
</ul>
```

By using the ‘filter’ filter, and passing through the Scope object ‘searchText’ as an argument which was created by the ‘ngModel’ directive on the form input, the filter returns an array of items that have the search characters in the specified order. This is not case sensitive.

Only a single guitar contained ‘Ac’ in that order, so the filter returns an array with only that item in it to be looped through with the ‘ngRepeat’.



Two guitars contained the number 1, so the filter returns an array with those two items in it to be looped through with the ‘ngRepeat’.



Custom Filters

'Filters' are another of Angulars specialized objects and just like Controllers and Directives, we can create our own. Filters take a name and then and a filter factory function.

```
.filter('reverse', function() {
    return function(input, upperCase) {
        input = input || '';
        var out = "";
        for (var i = 0; i < input.length; i++) {
            out = input.charAt(i) + out;
        }
        if (uppercase) {
            out = out.toUpperCase();
        }
        return out;
    };
});
```

The factory function always takes an input and then optional arguments. The above example takes a string as input and a boolean argument for upperCase. The filter reverses the string and potentially changes the case based on whether an 'upperCase' argument was provided and true.

Here it is in action:

```
.controller('MyController', function() {
    this.greeting = 'hello';
});
```

```
<div ng-controller="MyController as myCtrl">
    <p>{{myCtrl.greeting | reverse:true}}</p>
</div>
```

OLLEH

Angular Forms

In the past, client side form validation has always been a beast that is tough to manage and requires a lot of custom code to handle. Angular's form directive streamlines and tames this beast making it simple to handle all of your form validation and display needs!

- Angular Wrapper that provides client side form validation
- Taps into HTML5 Form Validation API
- Easily implement custom input validation
- Manage form state and error message display
- Handle disabling of form submission

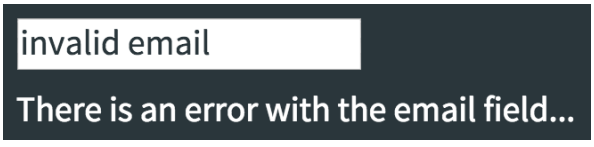
ngModel

Angular makes HTML forms very powerful with the way it handles errors and validation. By assigning a name to the form itself and then to any form elements you want to validate, we can access various methods, including $invalid.

The below will display a message if the 'myEmail' input field is invalid. We access the element through the form 'myForm.myEmail'.

```
<form name="myForm">
    <div class="field">
        <input type="email"
               name="myEmail"
               minlength="5"
               maxlength="100"
               ng-model="myEmail"
               required />
        <div ng-if="myForm.myEmail.$invalid">
            There is an error with the email field...
        </div>
    </div>
</form>
```

Invalid:



Valid:

The logic of passing values back and forth between the DOM and the scope is all managed by a well constructed controller called '**ngModelController**'.

Each time an input element is tied together with the ng-model attribute then the **'ngModel**' directive will create an instance of 'ngModelController'. All of the parsing, formatting, validating and transporting of data will be handled by this controller.

The value of a form field won't be written to scope by Angular until the element is valid. In the above case, the ngModelController will check to see that:

- The value is entered by the user (required)
- The value has a minimum length of 5 (minlength)
- Has a maximum length of 100 (maxlength)
- It is an email field (the email type attribute specifies this)

These validation attributes aren't Angular specific (HTML5), however, Angular handles the validity for the form based on the validation requirements.

When a validator fails, it registers it's error on the instance of the ngModelController on the $error object. Before we can display errors we need to first get ahold of the model within the template. As mentioned above, this can be achieved by first wrapping the model inside of a form and providing a name value to both the form and the input element.

We can now examine the validation status of the model via 'myForm.myEmail.$error'. We can also examine other properties on the model such as $pristine, $dirty, $valid, and $invalid. Having access to the model within the template allows for us to display error messages whenever any of these values change.

Property	Class	Description
$valid	ng-valid	Tells you whether model is currently valid
$invalid	ng-invalid	Tells you whether model is currently invalid
$pristine	ng-pristing	True if input has not been used yet
$dirty	ng-dirty	True if input has been used yet

HTML5 Attribute	ng Attribute	Registered Error
required="bool"	ng-required="..."	ngModel.$error.required
minlength="number"	ng-minlength="number"	ngModel.$error.minlength
maxlength="number"	ng-maxlength="number"	ngModel.$error.maxlength
min="number"	ng-min="number"	ngModel.$error.min
max="number"	ng-max="number"	ngModel.$error.max
pattern="patternValue"	ng-pattern="patternValue"	ngModel.$error.pattern

input type="..."	Registered Error
type="email"	ngModel.$error.email
type="url"	ngModel.$error.url
type="number"	ngModel.$error.number
type="date"	ngModel.$error.date
type="time"	ngModel.$error.time
type="datetime-local"	ngModel.$error.datetimelocal
type="week"	ngModel.$error.week
type="month"	ngModel.$error.month

ngSubmit

'ngSubmit' enables binding angular expressions to onsubmit events and prevents the default action of submitting the form. 'ngSubmit' is placed on the form element itself. The below example calls a controller function when the form is submitted.

```
<div ng-controller="formController as fromCtrl">
    <form ng-submit="fromCtrl.submit()">
        <input type="text" ng-model="fromCtrl.text" />
        <button type="submit">Submit</button>
    </form>
</div>
```

Angulars convenience methods for error checking and validation save us writing a huge amount of code and allows us to focus on the user experience.

There is a lot more to forms with Angular than we have covered here so we suggest experimenting with the available methods and functionality to really understand the time and code it reduces.

Translation

Localization is important when building applications. Planning for translation from the start will save considerable time and effort if and when you decide to push your application to different regions.

Pascal Precht translation module provides a clean implementation for language translations. At configuration time of your app you can set translations on the translateProvider for your supported languages. The translation object takes a language identifier and an object with key value pairs for the translations themselves.

```
angular.module("myApp", ['pascalprecht.translate'])
    .config(function ($translateProvider) {
        $translateProvider.translations('en', {
            TITLE: 'Hello',
            DESCRIPTION: 'This is a paragraph.'
        });
        $translateProvider.translations('de', {
            TITLE: 'Hallo',
            DESCRIPTION: 'Dies ist ein Paragraph.'
        });
        $translateProvider.preferredLanguage('en');
    });
```

- **Module**: pascalprecht.translate
- **Provider**: $translateProvider
- **Translation object**: (lang, translations)
    - ('en', { TITLE: 'Hello', DESCRIPTION: 'This is a paragraph.' })
- **'preferredLanguage'** method for setting language

To translate your content, add the translate filter to your translation key.

```
<h2>{{'TITLE' | translate}}</h2>
```

You can also change language at run time setting the language key with the $translate service.

```
<h2>{{'TITLE' | translate}}</h2>
<p>{{'DESCRIPTION' | translate}}</p>

<button ng-click="ctrl.changeLang('en')">
    {'BUTTON_LANG_EN' | translate}}
</button>
<button ng-click="ctrl.changeLang('de')">
    {{'BUTTON_LANG_DE' | translate}}
</button>
```

```
.controller('Ctrl', function ($scope, $translate) {
    this.changeLang = function (key) {
        $translate.use(key);
    };
});
```

English:

German:

It is likely that you do not want to hardcode your translation data in your configuration file. Angular translate gives you convenience methods to load local and server files.

```
$translateProvider.useUrlLoader('foo/bar.json');
$translateProvider.preferredLanguage('en_US');
```

```
$translateProvider.useStaticFilesLoader({
    prefix: 'locale-',
    suffix: '.json'
});
$translateProvider.preferredLanguage('en_US');
```

You can even create your own custom factories for loading the translation data. In the below example we have a custom built factory returning our translation data which we configure using the ‘useLoader’ method of the ‘$translateProvider’.

```
.config(function ($translateProvider) {
    $translateProvider.useLoader('translateLoader');
    $translateProvider.preferredLanguage('en_US');
})
```

```
.factory('translateLoader', function ($http, $q) {
    return function () {
        var deferred = $q.defer();
        $http.get('api/translations')
            .success(function(data) {
                deferred.resolve(data);
             })
            .error(function(error) {
                deferred.reject(error);
            });
        return deferred.promise;
    };
});
```

References

http://tylermcginnis.com/angularjs-factory-vs-service-vs-provider/

http://stackoverflow.com/questions/13762228/confused-about-service-vs-factory

http://www.airpair.com/angularjs#ESqt2XrvzwqkwLuI.99

http://www.airpair.com/angularjs/posts/top-10-mistakes-angularjs-developers-make#0U0fqeqBkuq6YqBy.99

https://github.com/angular-ui/ui-router

http://angular-ui.github.io/ui-router/site/#/api/ui.router.state.$state

http://blog.codepath.com/2012/11/15/asynchronous-processing-in-web-applications-part-1-a-database-is-not-a-queue/

http://www.i-programmer.info/programming/theory/6040-what-is-asynchronous-programming.html

http://chariotsolutions.com/blog/post/angularjs-corner-using-promises-q-handle-asynchronous-calls

http://toddmotto.com/digging-into-angulars-controller-as-syntax/

http://tutorials.jenkov.com/angularjs/watch-digest-apply.html

http://www.jvandemo.com/the-nitty-gritty-of-compile-and-link-functions-inside-angularjs-directives/