Community Experience Distilled

AngularJS Services

Design, build, and test services to create a solid foundation for your AngularJS applications

Jim Lavin

[PACKT] PUBLISHING open source* community experience distilled

www.allitebooks.com

AngularJS Services

Design, build, and test services to create a solid foundation for your AngularJS applications

Jim Lavin

BIRMINGHAM - MUMBAI

www.allitebooks.com

AngularJS Services

Copyright © 2014 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2014

Production reference: 1140814

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-356-8

www.packtpub.com

Cover image by Ádám Plézer (bitangkajla@gmail.com)

www.allitebooks.com

Credits

Author
Jim Lavin

Reviewers
Ruy Adorno
Mike McElroy
JD Smith

Acquisition Editor
Joanne Fitzpatrick

Content Development Editor
Anila Vincent

Technical Editors
Pankaj Kadam
Aman Preet Singh

Copy Editors
Janbal Dharmaraj
Karuna Narayanan
Alfida Paiva

Project Coordinator
Neha Bhatnagar

Proofreaders
Maria Gould
Ameesha Green

Indexer
Tejal Soni

Production Coordinators
Melwyn D'sa
Alwin Roy

Cover Work
Melwyn D'sa

www.allitebooks.com

About the Author

Jim Lavin has been involved in software development for the past 30 years. He is currently the CTO for one of the leading service providers of online ordering for restaurants where his team uses AngularJS as a core part of their service offerings. He is the coordinator of the Dallas/Fort Worth area's AngularJS meetup group, and routinely gives presentations on AngularJS to other technology groups in the Dallas/Fort Worth area.

I'd like to acknowledge all the people who have provided the inspiration and support that made this book a reality.

To my father and mother, who taught me that there are no limitations in life, and that you can do anything as long as you are willing to put in the concentration and hard work to see it to the end.

To my daughter, who would always help me get off the fence and make a decision by saying, "That sounds cool! Go for it!"

To my team at work, who embraced the notion of rewriting our application with AngularJS with such enthusiasm that we did the impossible in a matter of months.

And finally, to the AngularJS community, which has been a great part of my inspiration since I've adopted AngularJS. All their blog posts, videos, questions, and answers have helped to accelerate the adoption of AngularJS at an amazing rate, making me proud to be a part of their growing community.

---

www.allitebooks.com

About the Reviewers

Ruy Adorno is a senior front-end developer with more than 10 years of experience working in web development, application interfaces, and user experience. You can get to know more about him on his personal website `http://ruyadorno.com`.

Mike McElroy is a longtime fan, booster, and contributor to the AngularJS community. He originally met the author through the AngularJS community on Google+ while doing a series of hangouts on AngularJS. He worked with AngularJS professionally, shortly after it emerged from the beta period, and has continued to be involved in the community to this day. He currently works for DataStax, developing the UI for their OpsCenter product, and lives in Columbus, Ohio, with his wife and menagerie of animals.

JD Smith is a front-end architect with 15 years of consulting experience, ranging from small businesses to Fortune 500 companies. He enjoys working on large JavaScript applications and coming up with innovative improvements.

In addition to consulting work, he runs a boutique staffing firm, UI Pros, with the goal of matching the best developers to the best jobs. Contact him at `www.uipros.com` or send an e-mail to `jd@uipros.com`.

www.allitebooks.com

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

PACKTLiB™

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

www.allitebooks.com

Table of Contents

Preface **1**

Chapter 1: The Need for Services **7**

AngularJS best practices **8**
Responsibilities of controllers **8**
Responsibilities of directives **9**
Responsibilities of services **10**
Summary **11**

Chapter 2: Designing Services **13**

Measure twice, and cut once **13**
Defining your service's interface **13**
Focus on the developer, not yourself 14
Favor readability over brevity 14
Limit services to a single area of responsibility 14
Keep method naming consistent 15
Keep to the top usage scenarios 15
Do one thing only 15
Document your interface 15
Designing for testability **16**
Law of Demeter 16
Pass in required dependencies 17
Limiting constructors to assignments 20
Use promises sparingly 21
Services, factories, and providers **22**
Structuring your service in code **28**
Configuring your service **31**
Summary **32**

www.allitebooks.com

Chapter 3: Testing Services 33
The basics of a test scenario 33
Loading your modules in a scenario 35
Mocking data 36
Mocking services 38
Mocking services with Jasmine spies 40
Handling dependencies that return promises 42
Mocking backend communications 45
Mocking timers 47
Summary 48
Chapter 4: Handling Cross-cutting Concerns 49
Communicating with your service's consumers using patterns 49
Managing user notifications 54
Logging application analytics and errors 61
Authentication using OAuth 2.0 66
Summary 69
Chapter 5: Data Management 71
Models provide the state and business logic 71
Implementing a CRUD data service 75
Caching data to reduce network traffic 79
Transforming data in the service 82
Summary 87
Chapter 6: Mashing in External Services 89
Storing events with Google Calendar 89
Using Google Tasks to build a brewing task list 94
Tying the Google Calendar and task list together 98
Summary 105
Chapter 7: Implementing the Business Logic 107
Encapsulating business logic in models 108
Encapsulating business logic in services 111
Models or services, which one to use? 113
Controlling a view flow with a state machine 114
Validating complex data with a rules engine 120
Summary 123
Chapter 8: Putting It All Together 125
Wiring in authentication 126
Displaying notifications and errors 126
Controlling the application flow 127

www.allitebooks.com

Displaying data from external services **128**
Building and calculating the recipe **130**
Messaging is the heart of the application **132**
Summary **132**
Index **135**

www.allitebooks.com

Preface

AngularJS is a popular JavaScript framework for building single-page applications that has taken the open source community by storm since it reached the spotlight in 2013. Since then, AngularJS has seen an exponential growth in its adoption. With this adoption, new modules are released almost daily that integrate popular libraries such as Bootstrap, D3.js, and Cordova into AngularJS, helping to accelerate frontend development like never before.

AngularJS also allows frontend developers to use HTML markup to define data bindings, validation, and response handlers to interact with user actions in a declarative format that also contributes to this same acceleration. This declarative nature makes AngularJS easy for non-programmers to learn and include into their daily development workflow.

With this ease and acceleration of development comes the evolution of richer applications that tend to grow in size and complexity. Knowing how to properly architect your AngularJS applications as they grow becomes more important.

That is where this book comes in. It uses a sample application to show you how to architect and build a series of AngularJS services that address the common architectural layers for authentication, messaging, logging, data access, and business logic that's required for any moderately complex application.

This book also shows how to integrate external cloud services and third-party libraries into your application in such a way that AngularJS can take advantage of them with a little effort as you cross the boundary between AngularJS' internal workings and the external library's code.

What this book covers

Chapter 1, *The Need for Services*, discusses why services provide the core foundation in AngularJS applications and what should and shouldn't go into a service. The chapter also covers AngularJS best practices and how to properly partition an application's functionality across the various components of AngularJS.

Chapter 2, *Designing Services*, covers how to design and structure AngularJS services by leveraging best practices from both the AngularJS community and the core concepts of multi-tiered application architecture patterns.

Chapter 3, *Testing Services*, shows how to write scenarios to unit test your AngularJS services. The chapter also covers how to mock your service's dependencies by leveraging the angular.mock library and Jasmine's spy functionality.

Chapter 4, *Handling Cross-cutting Concerns*, shows how to build a core set of services to implement common functionality that can be leveraged by the controllers, directives, and other services in your application. The concept of how to wrap third-party JavaScript libraries inside an AngularJS service is also introduced by creating services to authenticate the user against an external cloud service and to log application errors to an external server.

Chapter 5, *Data Management*, discusses how to build services that create, retrieve, update, and delete application data on external servers. The chapter also discusses how to cache application state and data on the client as well as how to provide services to transform data into the various formats required by controllers and directives.

Chapter 6, *Mashing in External Services*, covers how to incorporate third-party libraries into your application by using services to wrap non-AngularJS JavaScript libraries. The chapter also covers how to build services to interact with popular cloud services.

Chapter 7, *Implementing the Business Logic*, discusses how to build services that move business logic to the client side to build a new class of serverless application, which requires no application server. A sample rules-based engine and finite state machine is also discussed to show how more complex business logic can be incorporated into your application.

Chapter 8, *Putting It All Together*, discusses how the services built in previous chapters can be combined to build a web application that allows home-brewing hobbyists to formulate beer recipes.

What you need for this book

This book is about writing AngularJS applications and as such, you'll want to have a computer to look through the sample application and run it.

Since the sample application uses Grunt.js to build, run test suites, and run the application, you'll also need Node.js installed on your computer.

I would also recommend having a good code editor such as Eclipse, NetBeans, Sublime Text, Visual Studio, or WebStorm. If price is an issue and you have limited resources, you could alternatively look at some of the free web-based IDEs such as Brackets, Cloud9, and Codio.

Several chapters use external cloud-based services such as mongolabs.com, Google+, the Google Calendar API, and Google Tasks API. To see them in action as you run the application, you'll need developer accounts for each of the services used. They are free for developers and allow limited usage without costs. The source code for the sample application includes instructions on how to register and set up your accounts for each of the services used.

Who this book is for

This book assumes that you are already familiar with JavaScript and have some experience or exposure to writing applications using the AngularJS framework. This book does not try to teach you JavaScript or AngularJS. If you are just starting out, I would suggest you look to other books published by Packt Publishing to get you up to an experience level where you will better understand this book and its concepts.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "You can then inherit that functionality by using the `$controller` service."

A block of code is set as follows:

```
var app = angular.module('angularjs-starter', []);

app.controller('ParentCtrl ', function($scope) {
  // methods to be inherited by all controllers
```

```
    // go here
  });

  app.controller('ChildCtrl', function($scope, $controller) {
    // this call to $controller adds the ParentCtrl's methods
    // and properties to the ChildCtrl's scope
    $controller('ParentCtrl', {$scope: $scope});
    // ChildCtrl's methods and properties go here
  });
```

Any command-line input or output is written as follows:

```
bower install
npm install
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Clicking on the **Next** button moves you to the next screen."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

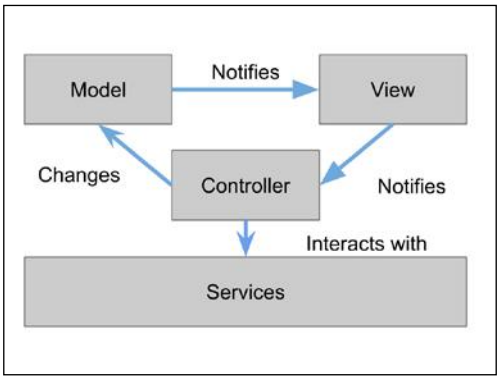
1
The Need for Services

In this chapter, we are going to cover why services are needed, how they support your application, and their responsibility in the overall application. Along the way, we will cover the responsibilities of the other AngularJS components and provide some guidelines on how to determine where your code should go according to the best practices.

I've watched presentations on AngularJS, and people always talk about the models, views, and controllers, but they often leave out the most important component, services.

Services underlie everything; they provide cross-cutting functionality, request and manipulate data, integrate with external services, and incorporate business logic, and yet they're as simple as a JSON object.

Services are like the foundation of a building. Sure, a building can be built without a foundation, but it won't last for long. And without services in your application, it will soon become so unwieldy that it too will not last for long. The following figure shows the AngularJS components and their interactions:



AngularJS best practices

One of the most important things to know when picking up a new framework is how to use it correctly, and that is where best practices come in. Best practices are guidelines on how best to use a framework. They also provide guidelines on how to structure your application to make it maintainable, testable, and reusable.

It is through best practices from the AngularJS community that you'll learn how to best architect your applications and see the true values of services to an AngularJS application.

One of the most important best practices is that each component type has a unique and specific responsibility when it comes to the overall application. This specific responsibility provides you a guide on how to partition your code across the various AngularJS component types. Separation of responsibilities also helps with the maintainability of the application by keeping all the code for a given responsibility to a particular type of component.

Let's spend some time discussing the different types of AngularJS components and their responsibilities, so you can better understand how you should structure your application.

Responsibilities of controllers

Controllers are responsible for managing the interaction between the user of the application and the model. They provide the event handlers that respond to the various user interactions that occur throughout the course of the application. Controllers also handle the program flow by invoking methods on services to update data and direct the browser where to navigate next.

They do not manipulate **Document Object Model** (**DOM**), interact with external services, store data, nor do they perform lengthy and complex business calculations. These responsibilities are for the other components in your application.

Controllers should be as thin as possible code-wise. By limiting the controller to just those methods needed to interact with the model and the user's interactions, you reduce its complexity and make the controller easier to maintain.

You will often find yourself repeating the same event handling code over and over in your controllers and may be tempted to move that code to a service, but it is sometimes better to move the code to a base controller. You can then inherit that functionality by using the `$controller` service to add the base controller's properties and functions on to the controller's scope, as shown in the following code:

```
var app = angular.module('angularjs-starter', []);

app.controller('ParentCtrl ', function($scope) {
  // methods to be inherited by all controllers
  // go here
});

app.controller('ChildCtrl', function($scope, $controller) {
  // this call to $controller adds the ParentCtrl's methods
  // and properties to the ChildCtrl's scope
  $controller('ParentCtrl', {$scope: $scope});
  // ChildCtrl's methods and properties go here
});
```

By using this pattern of a base controller and inheriting the functionality in your other controllers, you reduce the amount of code in your application and increase its testability and maintainability immensely.

This pattern also allows you to deal with the more problematic issues when refactoring code. If you have redundant event handlers that need to be unregistered when a view is destroyed, you can ensure that the code to unregister the event handlers is executed when the controller is destroyed by including the redundant code in the base controller and using the `$on` method to execute it when the `$destroy` message is broadcasted. However, if we were to use the pattern of inheriting functionality by embedding an `ng-controller` directive in another `ng-controller` directive and the code to unregister our handlers was in parent `ng-controller`, the code would not get executed until the parent controller was destroyed, which will not work in this case.

Responsibilities of directives

Directives are responsible for manipulating DOM within your application and provide a way to extend the HTML syntax with new functionality.

If you find yourself using `$('element')` in your controllers or services, best practices dictate that you move that code to a directive. This is usually because, whenever you use `$('element')`, the next code fragment that follows is one that manipulates DOM in one form or another.

www.allitebooks.com

Best practices also state that you should move repetitive HTML code to a directive to help simplify the HTML code across your application:

```
<div class="row-fluid" ng-repeat="adjunct in adjuncts">
    <div class="span1">{{adjunct.Name}}</div>
    <div class="span1">{{adjunct.Type}}</div>
    <div class="span1">{{adjunct.Use}}</div>
    <div class="span1">{{adjunct.Time}}</div>
    <div class="span1">{{adjunct.Amount}}</div>
    <div class="span1">{{adjunct.UseFor}}</div>
    <div class="span1">{{adjunct.Notes}}</div>
    <div><a href="#/editadjunct/{{adjunct._id.$oid}}">
    <i class="icon-pencil"></i></a></div>
</div>
```

Instead of repeating the preceding HTML code all over your application, you can use a directive to replace it, as shown in the following code snippet. When rendered, the directive emits the same HTML code as the preceding one.

```
<div class="row-fluid" ng-repeat="adjunct in adjuncts">
    <display-adjunct item="adjunct"></display-adjunct>
</div>
```

This provides you with a declarative way of expressing your application's data and it reduces the amount of HTML you need to use to express it.

Directives can also handle the work of validating form inputs. Validation directives help you to move validation code out of your controller into reusable components that you can leverage across all of your application's forms. Using directives to validate your form data is another AngularJS best practice that you should use to reduce the complexity of your application's controllers and keep them as thin as possible.

Responsibilities of services

Services are responsible for providing reusable code libraries to the other components in your application. Services can provide cross-cutting functionality for your application, such as logging, authentication, and messaging.

They can contain code to request and store data from external servers. They can also include functionality to manipulate, sort, filter, and even transform the data into different projections as necessary.

Services should also be used to integrate with external services that you use in your application. For example, you could create a service that wraps the Dropbox API to allow your application to write its data to a user's Dropbox account or encapsulate an analytics library to keep track of how users navigate through your application.

Since AngularJS only creates a single instance of a service during your application's lifetime, you can also use a service as a great way to communicate among components. One controller can store data in a service, navigate to a new controller, and then the new controller can pull the data from the service without having to make a server request or parse query parameters.

Finally, services can be used to provide business logic to your application. You could create a rules engine to handle complex form validation or a state machine to handle user workflow for a long-running business process. Maybe you need a complex compute engine that calculates tax and shipping charges for a shopping cart. The variations in how services can be used are endless.

Summary

We've discussed how services provide a foundation for your applications and some of AngularJS' best practices, the core of which is that each component has a specific purpose.

We covered how controllers should be as thin as possible and only contain the code necessary to interact with user events and manipulate the model. We saw how common functionality should be moved to a base controller and the `$controller` service can be used to inherit the common functionality in your application controllers.

We discussed how all code that manipulates DOM should be put in directives, which provide us with a way to extend and simplify our HTML code. We also discussed another best practice of moving validation code out of controllers and into directives where they can be reused throughout your application.

Finally, we covered how services provide a large amount of functionality for your application. Your controllers and directives leverage them and in turn, services maintain data across components, encapsulate reusable code, wrap external libraries, and reduce the amount of code in your application.

As you progress further in this book, we'll cover the various types of services and you'll see how they interact together to build a solid foundation for your application.

2
Designing Services

Before you start coding your service, it is best to spend some time thinking about what functionality it provides, how it provides that functionality, and how to structure it for testability and maintainability.

Measure twice, and cut once

There is an old saying, "Measure twice, and cut once". This saying came about as a best practice to not waste building materials. If you were going to cut a board down to size, you'd measure the board twice before cutting it to ensure you cut the board to the correct size.

In this fast-paced world of software development, everybody is into the agile way of building out code, which means that a service evolves over several iterations. There is nothing wrong with this at all, except if you don't have a roadmap in mind, sooner or later, you'll have a kitchen-sink-style service that is hard to use, hard to test, and even harder to maintain.

So, you need to follow the "Measure twice, and cut once" motto when building your services, and you need to design them in such a way that they are easy to use, easy to test, and easy to maintain.

Defining your service's interface

When I start to write a service, I use a set of best practices to guide me on its construction. I've gathered these over my years of writing libraries, frameworks, and services, and they never fail to provide the proper guidance as I write my code.

Focus on the developer, not yourself

When you start to define a service, think about the consumer of your service and what their skill levels are; are they experienced developers, are they new to the framework, or are they in-between? Does your service require knowledge of higher-level concepts to comprehend? Or, is it very basic to comprehend? These are all things you should think about, because the harder it is for the developer to consume your service, the more painful their day-to-day life will be when using it. If you make your service easy to consume, developers will thank you for making things easy for them.

Favor readability over brevity

Use verb-noun combinations for method names, don't use abbreviations or contractions, and don't use acronyms unless widely accepted. If only every service and framework developer followed this practice, our lives would be easier. I find it hard to figure out what someone was thinking when I see method names such as `repr()` or `strstr()`; they're not English words, they're not familiar acronyms, and you can't tell what they do just by looking at them. Using verb-noun combinations and favoring readability, method names such as `uploadFile`, `calculateSRMColor`, and `startMashTimer` all help the developer to understand your service's interface. The developer may not know what a **Standard Reference Method** (**SRM**) color is, but they can pretty much figure out it is an acronym that has to do with the business domain and that if they call it they are going to have the service calculate it for them.

Limit services to a single area of responsibility

When designing your service interface, you should use the single-responsibility principle and limit your service's functionality based on its intended use. If the service is designed to calculate recipe parameters, limit it to just those methods; put other methods that do not deal with calculating recipe parameters into a different service. Sure, the number of services you'll end up with will be higher; however, each service will be easier to maintain since the amount of code it contains will be smaller and all in one place. Your services will also be easier to test, since you will be able to isolate their code easily. We'll cover this in depth later in this chapter when we talk about service testability.

Keep method naming consistent

Keep consistency across all of your method names to make your service interface easier to consume. If each of your services uses different naming methods, the developer trying to consume your service is going to loath your existence for all eternity. If one method that retrieves fermentable ingredients for a recipe is called `getFermentableIngredients` and another method that retrieves miscellaneous ingredients for a recipe is called `retrieveMiscelaneousIngredients`, how is the developer to know that they both go out to the server and return data? Does the method that begins with `get` only return data from the service and not request the data from the server? Is there another method he/she needs to call first to retrieve the data from the server? By keeping method names consistent in your service definitions, you can eliminate this confusion.

Keep to the top usage scenarios

Keep the interface simple by planning for the top usage scenarios of each service. In other words, don't include service functionality that isn't going to be used. Let's say you're building a data service and there is some data that is read-only; do you need to add create, update, and delete methods for that data? If the methods will never be called, don't add them to your service. Use the agile development principle of never adding code before it's time and when there comes a time, add it to your service.

Do one thing only

Each method should do one and only one thing. Break out functionality into separate methods and avoid using `SWITCH` statements to execute functionality based on function parameters. These types of methods only lead to testing nightmares, they will be the source of bugs and should be avoided at all costs. It hurts nothing to add a method for each specialized case and it'll make your service much easier to test. If there is a lot of redundant code once you're done, look at refactoring your code or using libraries such as `underscore` or `lodash` to leverage functional programming paradigms to implement the functionality.

Document your interface

Document your service interface using JSDoc, YUIDoc, Ext Doc, or some other tool to provide consumers a productivity boost. Many popular **Integrated Development Environments** (**IDEs**) can parse JSDoc-formatted comments in your code and provide IntelliSense prompts to the developer as they are editing their code.

These prompts help developers to understand how to call your service, what parameters need to be passed into a method, and what type the parameters need to be. These prompts make consuming a service much easier and help developers avoid having to keep looking at documentation while they are coding.

Now that we've covered some core guidelines to use when designing your service, let's take a look at how you should design your service for testability.

Designing for testability

AngularJS was built with testability in mind and that is evident in how dependency injection plays a major role in the various constructor methods that are part of the module component.

Law of Demeter

Dependency injection allows you to write code that is loosely coupled to the services it's dependent on. This allows you to write code that follows the Law of Demeter, which is a best practice when writing testable code. In its general form, the Law of Demeter says that each unit of code should have only limited knowledge about other units of code.

In other words, don't call chained objects in your methods. Instead, pass in the objects you need to interact with. If you call a chained object in your methods, you increase the complexity of your code and make it harder to test. For example, the following method calls chained objects:

```
$scope.getBrewerName = function (authenticate) {
  if (authenticate && authenticate.currentBrewer) {
    return authenticate.currentBrewer.firstName + ' ' +
       authenticate.currentBrewer.lastName;
  }
  return '';
},
```

Notice how the code accepts an object called `authenticate` and how it checks to ensure that both the `authenticate` object and the `currentBrewer` object are valid before accessing the `currentBrewer` object's properties to return the brewer's name.

To test the preceding code with a mock object is hard. Not only do you have to mock all of the methods called on the `authenticate` object, but you also need to mock all of the methods called on the `currentBrewer` object, making your testing code much longer than it needs to be.

However, if you follow the Law of Demeter, you can simplify your code, as follows:

```
$scope.getBrewerName = function (brewer) {
    if (brewer) {
        return brewer.firstName + ' ' + brewer.lastName;
    }
    return '';
},
```

Now, the code is only validating the object it invokes methods on, not on the wrapper object that may encapsulate it. Your test code will also be simpler, since you only have to mock out the methods called on the `brewer` object.

By following the Law of Demeter, you build services that are coupled to interfaces instead of object hierarchies and your code is less likely to break when you modify the code of those objects you pass to your methods.

With the loose coupling that the Law of Demeter provides, you can also swap out the objects your code is dependent on with mock objects that return known responses to help you in testing your code. We'll talk about this more in *Chapter 3*, *Testing Services*.

Pass in required dependencies

Another testability principle is to use dependency injection as it was intended: pass all the required dependencies of your service in the creation function, and don't use the `$injector` service to retrieve dependencies unless there is no other way to inject a required service. Although you can get an instance of a service using the `$injector` service, you cannot test that code. This is because you cannot ensure you've mocked out all the services that your code will invoke.

By passing in all the required dependencies your service will interact with, you have a road map of those services you will need to mock out for testing and you will also be able to easily determine the methods being called on each service.

Which of the following service definitions seems easier to understand? The first one takes the `$injector` service as an argument in the constructor method and then uses the `$injector.get()` method to get the various service instances the service will use. Not only does it hide the dependencies of the service, but it also makes the code more verbose and harder to understand.

```
angular.module('brew-everywhere').factory('authenticate',
  function ($injector) {

    var authenticate = {
      rootScope: $injector.get('$rootScope'),
```

```
    window: $injector.get('$window'),
    shaService: $injector.get('sha1'),
    userResource: $injector.get('userResource'),
    currentBrewer: null,
    // methods
    login: function (username, password) {
      authenticate.userResource.query({UserName: username})
        .then(function (brewers) {
          if (brewers.length === 0) {
            authenticate.window.alert("Invalid user name");
            return;
          }

          var brewer = brewers[0];

          var passwordHash =
            authenticate.shaService.hash(password +
            Date.parse(brewer.DateJoined).valueOf().toString());

          if (passwordHash !== brewer.Password) {
            authenticate.window.alert("Invalid password.")
            return;
          }

          authenticate.currentBrewer = brewer;
          authenticate.rootScope.$broadcast('USER_UPDATED');
        });
    },

    isBrewerLoggedIn: function () {
      if ((authenticate.currentBrewer !== null) &&
        (authenticate.currentBrewer !== undefined)) {
        return true;
      }

      return false;
    },

    logout: function () {
      authenticate.currentBrewer = null;
      authenticate.rootScope.$broadcast('USER_UPDATED');
    }
```

```
    };

    return authenticate;
  });
```

The following second service definition passes the service's dependencies in the factory method, which makes it much clearer to understand. Also, since the dependencies are passed in the factory method, the code is less verbose since you do not have to prefix each of the service instances when using them.

```
angular.module('brew-everywhere').factory('authenticate',
  function ($rootScope, $window, sha, userResource) {

    var authenticate = {
      // data members
      currentBrewer: null,

      // methods
      login: function (username, password) {
        userResource.query({UserName: username})
          .then(function (brewers) {
            if (brewers.length === 0) {
              $window.alert("Invalid user name");
              return;
            }

            var brewer = brewers[0];

            var passwordHash = sha.hash(password +
              Date.parse(brewer.DateJoined).valueOf().toString());

            if (passwordHash !== brewer.Password) {
              $window.alert("Invalid password.")
              return;
            }

            authenticate.currentBrewer = brewer;
            $rootScope.$broadcast('USER_UPDATED');
          });
      },

      isBrewerLoggedIn: function () {
        if ((authenticate.currentBrewer !== null)
          && (authenticate.currentBrewer !== undefined)) {
          return true;
        }
```


www.allitebooks.com

```
        return false;
      },

      logout: function () {
        authenticate.currentBrewer = null;
        $rootScope.$broadcast('USER_UPDATED');
      }

    };

    return authenticate;
  });
```

Again, leave using the `$injector` service to those occasions when you cannot pass in the dependencies to the factory method, which for services is limited to special cases where you need to instantiate services dynamically based on environmental or configuration constraints. Also, you are not guaranteed that the `$injector` service is going to return a valid service instance when you call it, which means you may also have to add defensive code to check each service instance before using it. However, you can always guarantee that when you pass dependencies in the constructor method, they will be valid. This allows you to eliminate that extra code, making your service easier to maintain.

The only other time you will ever need to use the `$injector` service is when you are trying to access a service in the `config` method of a module or during unit testing, so leave it till then.

Limiting constructors to assignments

Another good point to keep in mind when writing code for testability is that you should limit your service's constructor methods to only field assignments and method definitions. Don't call external services to get initialization data. Add this code in a startup or configuration method that should be called after your service instance has been created.

When you include calls to external services in your service's constructor method, you make it very hard to test and debug. If the external service call fails during your service's initialization, the call might throw an exception, causing your service to not be created. Not only can this cause your entire application to not start up, but it could cause your application to fail later on when the service is instantiated for the first time. I like to create an `init` method for my services that need to retrieve external data; this way, I can initialize the service in a module's `run` method, which executes after the application starts up and before the first view is displayed.

Also, by encapsulating calls to external services in a method on your service, you can test that piece of code easily and with as little code as possible. Let's face it, we programmers are lazy when it comes to writing code and the less we have to write, the easier our lives seem. Also, when it comes down to writing unit tests for a service, which would you rather do; write a simple mock service and a few lines of code to ensure the request was made, or write a complex test with many more lines of code, just to catch the service request when the service was instantiated? I will opt for the lazy coder's route every chance I get. It's simpler, it's easier to understand, and easier to test.

Use promises sparingly

The last thing I'm going to cover on designing your service for testability is a bit controversial. I recommend that you avoid returning promises from methods, and instead choose an eventing or publish/subscribe communication mechanism over promises when communicating the end of a long-running request or calculation. Now, I know that promises are a great way to eliminate the callback quagmire in your applications and they allow you to chain method calls together once a promise is resolved, but there are situations where promises don't work in complex applications.

Promises are only good for a single call. You can't reset a promise and resolve it a second time. If you have a complex application where there are many consumers of a service that need to know when the service's data is updated, promises won't work.

A good example is in the sample application. As you build a recipe and add ingredients to the recipe, a rules engine gets invoked to match the current recipe with beer styles that best match the recipe's style parameters. If you were to have the rules engine return a promise from each call, the other views displayed on the page would not get notified when the rules engine finished and they would have to invoke the rules engine themselves in order to get the same data. However, if you use an eventing system, each of the views can subscribe to the rules engine's calculations complete event and be notified whenever the rules engine is run. This cuts down the number of times the rules engine has to calculate and it simplifies the code in the application.

Don't get me wrong, promises are great and they serve their purpose when you are dealing with complex calls across several services in parallel, or when you are using them in route-resolving methods to retrieve data for the controller before it is displayed, but they are not necessary for all your long-running service calls. We'll talk more about this later in *Chapter 4*, *Handling Cross-cutting Concerns*, when we talk about how your service should communicate with the outside world.

Now that we've covered tips on how to design your service interfaces and how to design your service for testability, let's talk about the various ways you can define your service.

Services, factories, and providers

As we have discussed so far, a service is a single instance of an object, function, or value that you can leverage across the various components of your application. When you inject a service into an application, the `$inject` service first looks to check if an instance of the service already exists. If it does, the `$inject` service returns the existing instance. If it does not, the `$inject` service creates a new instance of the service and returns it.

With this in mind, we can use one of five different module-definition methods to create our service. The first two are best for static values, configuration values, and models and the rest are best for defining services based on how they are constructed or used.

The first method is the `constant` method, which is best used to define a primitive value or object that will never change and needs to be made available for use by a module's `config` method. The following is an example of using the `constant` method to define messages used for a logging service:

```
angular.module('logging').constant('logging_config', {
    traceLevel: {
        _LOG_TRACE_: '_LOG_TRACE_',
        _LOG_DEBUG_: '_LOG_DEBUG_',
        _LOG_INFO_: '_LOG_INFO_',
        _LOG_WARN_: '_LOG_WARN_',
        _LOG_ERROR_: '_LOG_ERROR_',
        _LOG_FATAL_: '_LOG_FATAL_',
    }
});
```

The preceding code illustrates how messages can be defined for use by other modules to indicate the trace level of the log message sent to a logging service.

The next definition type is the `value` method, which also allows you to define a primitive value or object that can be used by your application components. The difference is that the primitive values and objects created using the `value` method can be changed. Another difference is that the singletons created using the `value` method cannot be used by a module's `config` method. The following is the same example, this time defined as a value. Notice how it is very similar to the `constant` definition, with the only difference being the word `value` instead of `constant`:

```
angular.module('logging').value('logging_config', {
    traceLevel: {
        _LOG_TRACE_: '_LOG_TRACE_',
        _LOG_DEBUG_: '_LOG_DEBUG_',
```

```
            _LOG_INFO_: '_LOG_INFO_',
            _LOG_WARN_: '_LOG_WARN_',
            _LOG_ERROR_: '_LOG_ERROR_',
            _LOG_FATAL_: '_LOG_FATAL_',
        }
    });
```

Why would you use a `constant` method over a `value` method? The main consideration is whether you need to allow the values to be modified or overwritten by a consumer of your service. If so, use the `value` method; this way, consumers of your service can override the values using an AngularJS decorator. If you do not intend the values to be modified, then use the `constant` definition. Also, if you need to use the value service inside a module's `config` method, use the `constant` method.

Another good use for the `value` method is to define model objects that will be used by your components. This pattern provides a nice way to keep the model definition and the code that operates on the model in the same module, helping you to keep repetitive code out of your components and in a single file where it resides best. This code pattern can also help to keep your controller's code as thin as possible when it comes to operating on the model.

The following code example shows how you can define a value that contains both the model and code used to interrogate the model to derive calculated values from the model. In this case, the brewer model is defined, and two methods are defined on the prototype of the `brewer` object that can be used to retrieve the full name of the brewer and check to see if the brewer has an item in their inventory:

```
(function () {
  'use strict';

  var Brewer = function () {
    var self = this;
    self.userName = '';
    self.firstName = '';
    self.lastName = '';
    self.email = '';
    self.location = '';
    self.bio = '';
    self.webSite = '';
    self.avatar = '';
    self.photo = '';
    self.dateJoined = '';
    self.inventory = [];
  };
```

```
    Brewer.prototype = {
      fullName: function(){
        return this.firstName + ' ' + this.lastName;
      },
      hasItemInInventory: function(value){
        var result = false;
        if (this.inventory && this.inventory.length > 0){
          angular.forEach(this.inventory, function(item){
            if(item.name === value){
              result = true;
            }
          });
        }
        return result;
      }
    }
    angular.module('brew-everywhere').value('brewer', Brewer);
  })();
```

Now, when you inject this into a controller or directive for the first time, the `$inject` service will provide the constructor of the model that you can use to create new instances of the model in your code.

One thing to keep in mind when deciding to use the `constant` or `value` methods is that if you need consumers of your service to provide configuration data for your service, such as an API key, service URL, or other parameter, it is best not to use either of the previous definition methods, but to use the `provider` method. We will cover the `provider` method later in this section and provide an actual service configuration method the application can call to provide the required configuration data.

Now that we have covered how to define the different value services, let's discuss the other methods we can use to define more complex services.

The first method we will cover is the `service` method, which provides a shorthand method for registering a constructor function that will be instantiated by calling the object's new method. This is similar to the `value` method, but is best used if you define your services using the popular type/class methodology as discussed in the various books for object-oriented programming with JavaScript.

In the following code example, we've defined a logging service, which wraps the `log4javascript` library using the `service` method. First, we define a constructor function called `Logging`, which defines the data members of the service and then we define the service's functionality by adding the various methods to the `Logging` object's prototype. Finally, we define the service by providing the constructor method to the AngularJS module's `service` method.

```
(function () {
  'use strict';

  var Logging = function () {
    this.log = null;
  };

  Logging.prototype = {
    init: function () {
      // get the logger object
      this.log = log4javascript.getLogger("main");
      // set the log level
      this.log.setLevel(log4javascript.Level.ALL);
      // create and add an appender to the logger
      var appender = new log4javascript.PopUpAppender();
      this.log.addAppender(appender);
    },
    trace: function (message) {
      this.log.trace(message);
    },
    debug: function (message) {
      this.log.debug(message);
    },
    info: function (message) {
      this.log.info(message);
    },
    warn: function (message) {
      this.log.warn(message);
    },
    error: function (message) {
      this.log.error(message);
    },
    fatal: function (message) {
      this.log.fatal(message);
    }
  };

  angular.module('brew-everywhere').service('logging', Logging);
})();
```

We can also define a service using the AngularJS module's `factory` method. You should use this method if you are not using a type/class definition to define your service and you do not need to configure your service inside a module's `config` method.

The `factory` method wraps the constructor function with a default provider, which in turn is used to return the service instance when the provider's `$get` method is called. In the following code, we've defined the same logging service, but this time, we've used the `factory` method to define an object instance and return it at the end of the function:

```
(function () {
  'use strict';

  angular.module('brew-everywhere').factory('logging', function () {
    var logging = {
      log: null,
      init: function () {
        // get the main logger
        logging.log = log4javascript.getLogger("main");
        // set the logging level
        logging.log.setLevel(log4javascript.Level.ALL);
        // create and add an appender to the logger
        var appender = new log4javascript.PopUpAppender();
        logging.log.addAppender(appender);
      },
      trace: function (message) {
        logging.log.trace(message);
      },
      debug: function (message) {
        logging.log.debug(message);
      },
      info: function (message) {
        logging.log.info(message);
      },
      warn: function (message) {
        logging.log.warn(message);
      },
      error: function (message) {
        logging.log.error(message);
      },
      fatal: function (message) {
        logging.log.fatal(message);
      }
    };

    return logging;
  });
})();
```

The final way we can define a service is to use the module's `provider` method. The `provider` method registers a service provider that has a `$get` method that instantiates the service. The definition of the service provider can also have additional methods that can be called when you reference the provider itself and not the instance. If you need to configure your service before instantiating it, using the `provider` method is the best way to define your service.

In the following code, we've defined the same logging service, but using a `provider` method. We've moved the `init` method out of the service's definition to the provider's definition and created two other methods to allow the consumer to set the log level for the service and add a logging appender to the service, making the service more configurable overall.

```
(function () {
  'use strict';

  angular.module('brew-everywhere').provider('logging', function ()
    {
    var logging = {
      log: null,
      init: function () {
        logging.log = log4javascript.getLogger("main");
      },
      setLogLevel: function (logLevel) {
        logging.log.setLevel(logLevel);
      },
      setAppender: function (appender) {
        logging.log.addAppender(appender);
      },
      $get: function () {
        function trace(message) {
          logging.log.trace(message);
        }
        function debug(message) {
          logging.log.debug(message);
        }
        function info(message) {
          logging.log.info(message);
        }
        function warn(message) {
          logging.log.warn(message);
        }
        function error(message) {
          logging.log.error(message);
```

```
        }
        function fatal(message) {
          logging.log.fatal(message);
        }

        return {
          trace: trace,
          debug: debug,
          info: info,
          warn: warn,
          error: error,
          fatal: fatal
        };
      }
    };

    return logging;
  });
})();
```

To review, use the `constant` method if you need to define values that will not change over the course of your application. Use the `value` method if you need to define values or models that will change over the course of your application. The `service` method should be used if you define your services as a class and need to invoke the definition's constructor function. Use the `factory` method if you define your service as an object instance and do not need to invoke a constructor. Finally, if you need to configure your service in a module's `config` method, use the `provider` method.

Now that we've covered the various ways you can define your services, let's take a look at how we should structure our services in the actual code.

Structuring your service in code

When you start to code your service, there are a few things you need to think about as you go; for example, not polluting the global namespace and only exposing the methods and properties you intend consumers to use when they interact with your service.

The reason you do not want to pollute the global namespace is because you want to protect your code from naming collisions that may occur with the other scripts that are loaded on the page and to keep your code from stepping on some other script's code.

The easiest way to prevent polluting the global namespace is to create an **Immediately-Invoked Function Expression** (**IIFE**) that you then use to define your module and service:

```
(function () {
  'use strict';

  // module definition goes here

  // service definition goes here

})();
```

Not only can this pattern be used for your services, but you can also use it for all of your controllers and directives. The only time I would suggest not using this pattern is if you are using a build tool that concatenates your code into a single file and wraps it into a closure for you, then it is not necessary to write the extra code.

The next thing to think about when writing your service is only exposing the methods and properties you intend the users of your service to access. The easiest way to do this is to use Christian Heilmann's Revealing Module Pattern to define your service.

The advantage of using the Revealing Module Pattern is you don't have to use the name of the main object to call methods or access variables and you have the ability to expose only those methods and variables to the outside world that you feel need to be accessible.

The following example defines a publish/subscribe messaging service using the Revealing Module Pattern. The `messaging_service` function defines a closure where we define the various methods of the service and any internal variables we need to store the service's state. At the bottom of the function definition, we return an object that includes only those methods we want to make visible to consumers of the service.

```
(function () {
  'use strict';

  // Define the factory on the module.
  // Inject the dependencies.
  // Point to the factory definition function.
  angular.module('brew-everywhere').factory('messaging_service',
                                            [messaging_service]);

  function messaging_service() {
```


www.allitebooks.com

```
    //#region Internal Properties
    var cache = {};

    //#endregion

    //#region Internal Methods
    function publish(topic, args) {
      cache[topic] && angular.forEach(cache[topic],
         function (callback) {
        callback.apply(null, args || []);
      });
    }

    function subscribe(topic, callback) {
      if (!cache[topic]) {
        cache[topic] = [];
      }
      cache[topic].push(callback);
      return [topic, callback];
    }

    function unsubscribe(handle) {
      var t = handle[0];
      cache[t] && angular.forEach(cache[t], function (idx) {
        if (this == handle[1]) {
          cache[t].splice(idx, 1);
        }
      });
    }

    //#endregion

    // Define the functions and properties to reveal.
    var service = {
      publish: publish,
      subscribe: subscribe,
      unsubscribe: unsubscribe
    };

    return service;
  }
})();
```

You'll see the preceding coding pattern in the majority of the services we cover in the book. However, the code examples for the various chapters do not include the IIFE since the Grunt script for each project wraps all of the code in an IIFE as part of the build process.

Configuring your service

Earlier in this chapter, we discussed using the module `provider` method to create a service with configuration methods that consumers of your service can invoke in a module's `config` method.

By using the `provider` method to define the example logging service, consumers can now configure the logging service inside a module's `config` method in a rather straightforward way:

```
angular.module('MyApp', ['logging']).config(function(loggingProvider){
  // init the logging service
  loggingProvider.init();
  // set the log level
  loggingProvider.setLogLevel(log4javascript.Level.ALL);
  // create and add an appender to the logger
  var appender = new log4javascript.PopUpAppender();
  loggingProvider.setAppender(appender);
});
```

In the preceding code, we initialize our service, set the logging level, and set the logging appender to be used when messages are logged by the service.

However, you don't always have to use the `provider` method to allow your services to be configured. You can use both the `service` and `factory` methods to define your service and provide configuration methods that can be called in a module's `run` method, which executes after your application has been bootstrapped. You just need to ensure your service follows the design practices we discussed earlier and doesn't try to call any external services as it is instantiated.

No matter how you create your services, by providing functionality to configure the services you create, you make your services more flexible, easier to use, and reduce the complexity required to include them in an application.

Summary

In this chapter, we discussed several best practices you can use when designing your AngularJS services, along with several best practices that you can follow to ensure your services are written with testability in mind. We then covered the various ways you can define your service. We discussed using the `constant` and `value` methods to define static values and models. We saw how the `service` and `factory` methods should be used based on your coding style. We also discussed a way consumers of your service can provide configuration data to your service at runtime by using the `provider` method.

We'll see examples of all of these principles as we cover more service examples later in this book. Next, we'll look at how we can best write tests for our services, so we can validate our services work as expected.

3
Testing Services

Whether you are a test-first or a test-last coder, it doesn't really matter. A unit test suite should be mandatory for every unit of code you write. Why? Because it not only shows that your code meets specifications, but it also helps provide documentation for how your code works and a guide to ensure that your code continues to work as you add new functionality and fix bugs.

Unit tests are the trademark of a software craftsman. A thorough unit test suite exercises every piece of written code. It tests the boundary cases along with well-known scenarios provided as part of the specification.

This chapter will take you through what it takes to write good unit test scenarios for your services, and it provides examples of the more common test scenarios you will need to test the majority of your services.

The basics of a test scenario

As the AngularJS project uses Jasmine for a testing framework and you'll see it in many of the examples you come across on the Internet, we're also going to use the Jasmine framework for the unit tests in this book.

The Jasmine framework has a well-defined structure for a test file and is patterned after the popular practice of **Behavioral Driven Development** (**BDD**). Each Jasmine test file consists of a scenario, which consists of one or more specifications that define the acceptance criteria for the scenario. Normally, a test file contains one scenario, but if you have a lot of small scenarios, you can have more than one scenario in a file. The examples in this book will only have one scenario per file.

A scenario file begins with a call to the global Jasmine function, `describe`, with two parameters: a string and a function. The string is a name or title for the scenario, usually what is under test. The function is a block of code that implements the scenario:

```
describe('authenticate service', function(){
    …
    specifications go here
    …
});
```

Specifications are defined by calling the global Jasmine function, `it`, which, like `describe`, takes a string and a function. The string is a title for this specification under test, and the function is the actual test:

```
describe('authenticate service', function(){
    it('should publish _GPLUS_AUTHENTICATE_ when
        loginWithGoogle is called', function(){
        …
        test code goes here
        …
    });
});
```

By including one or more expectations in each specification, we assert that our code is correct. Expectations are built with the `expect` function, which takes a value called `actual` that is evaluated to see if it meets the expected outcome of our code under test.

```
describe('authenticate service', function(){
    it('should publish _GPLUS_AUTHENTICATE_ when
        loginWithGoogle is called', function(){
        spyOn(messaging, "publish");
        authenticate.loginWithGoogle();
        expect(messaging.publish).
        toHaveBeenCalledWith(events.message._GPLUS_AUTHENTICATE_);
    });
});
```

As the `describe` and `it` blocks are functions, they can contain any executable code necessary to implement the test. JavaScript scoping rules apply, so variables declared in a `describe` block are available to any `it` block inside the scenario.

If there are steps you need to execute prior to running the tests in a test suite, you can use the `beforeEach` method to execute them before each test declared inside an `it` function is executed. If you need to execute steps to clean up after each test, you can use the `afterEach` method to execute them after each test is executed.

To test your AngularJS services using the Jasmine framework, there are two files you need to include in your Jasmine test-runner page or your Karma configuration. The first is the core AngularJS framework in angular.js. This file brings in the definitions that your code requires to run properly. The second is the AngularJS Mocks framework in angular-mocks.js. This file provides mocks of several services that your tests can use to isolate the code under test. If your service depends on other JavaScript libraries, you'll need to include them as well.

If you look at the `karma-unit.conf.js` file in this book's sample code, the `files` property defines an array of all of the files that need to be included for the tests to run properly. This array not only includes angular.js and angular-mocks.js, but it also includes the source code for the services and their unit test scenario files.

As you can find ample documentation on the Web on how to set up a Jasmine test runner or Karma test runner, we will not cover this in the book. Some of the websites to refer to are:

- `http://jasmine.github.io/`
- `http://angularjs.org`
- `http://karma-runner.github.io/`

Loading your modules in a scenario

There are a few actions you will need to take to ensure that Jasmine loads your service module, so it will be available for your test scenarios. The AngularJS Mocks library provides several methods to help load your modules and inject services into your scenarios as needed.

You can use the `module` method to load the module you want to test. Most of the time, you wrap the call to load your module in a `beforeEach` method. The following is an example of loading the `userService` module into a test scenario:

```
describe('User Service', function () {
  beforeEach(module('userService'));
  it('should return a valid user if the credentials are correct',
  function () {
  });
});
```

Once your module has been loaded into the test scenario, you'll need to get an instance of your service so that you can test the various methods of your service. To do this, you can use the `inject` method provided by the AngularJS Mocks library. The `inject` method allows you to define the dependencies you want to instantiate for use by the scenario. The `inject` method takes a function with a list of the services you want injected. The most common way of using the `inject` method is to define the services as variables in the scenario and then use a `beforeEach` function to inject them into the scenario and assign each service to a variable accessible to each specification in the scenario. The following is an example of using the `inject` function to get an instance of `userService` that can be used by each specification in the scenario by referencing the `service` variable:

```
describe('User Service', function () {
  var service = null;
  beforeEach(module('userService'));
  beforeEach(inject(function (userService) {
   service = userService;
  }));
  it('should return a valid user if the credentials are correct',
 function () {
   var actual = service.getCurrentBrewer();
   expect(actual).not.toBeFalsy();
  });
});
```

Mocking data

Mocking the data that your service interacts with is important when it comes to unit testing. Mocked data allows you to provide the conditions required to execute the various execution paths of your service's methods.

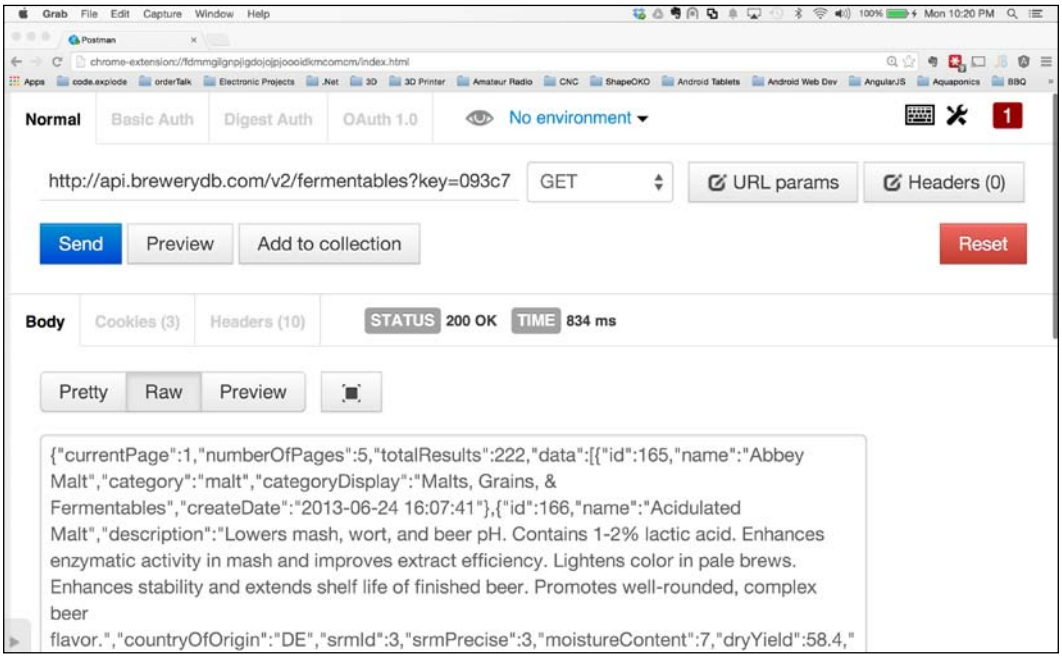
You can easily mock data objects that you pass into your services in your test files; however, if your tests require large amounts of data, it's probably easier for you to define your mock data in a separate JavaScript file. This helps keep your test files uncluttered and saves you from having to retype the code when you reuse the mock objects in your other tests.

Sometimes, when you create mock data for your tests, you will unknowingly create test data that will always meet the conditions of the code under test. To ensure that your code handles all the various types of data, it helps to use a library that will randomly generate the data for you. If your code is written properly, it should handle any data that comes its way. There are several libraries that can help you create random data; Faker.js and Chance.js are two such libraries. You just need to add the source files to the list of files that should be included with your unit tests, and then, you can start using them. The following is an example of a mock data object that uses Faker.js to create random data for the various data members of the brewer object:

```
var brewer = {
  userName: Faker.Internet.userName(),
  firstName: Faker.Name.firstName(),
  lastName: Faker.Name.lastName,
  email: Faker.Internet.email(),
  location: Faker.Location.city() + ', ' +
            Faker.Location.usState(),
  bio: Faker.Lorem.paragraph(),
  webSite: 'http://' + Faker.Internet.domainName(),
  avatar: Faker.Image.avatar(),
  photo: Faker.Image.people(),
  password: '',
  dateJoined: Faker.Date.past(),
  inventory: null,
  brewerIsAdmin: false
};
```

If your services go against existing REST-based API services, you can also use a tool such as Postman or SoapUI to capture data from a session with the services to use as mock data as well. As you make requests, you can save the responses to your test data file for use as responses to mocked `$http` requests. If the REST services you plan to call don't exist yet, you will have to mock up the data manually, and once they exist, you can then replace your mocked data with real responses.

The following is a screenshot of using the Chrome extension, Postman, to capture data from the `brewerydb.com` API:



The last thing to cover when talking about mocking data is how to use data from a database to mock your data. This becomes really easy if you execute your tests using a test runner such as Karma using Node.js. As you are already running the tests using Node.js, you can use any of the various Node.js database libraries to pull your mock data from your database of choice. Then, you can use the database library to access your test data and use it for your testing.

However, when you mock data for your unit tests, make sure that your data exercises the code under test and reflects the data that your application will actually experience when it runs.

Mocking services

If your services are of different complexity levels, they are more than likely to be dependent on other services. To isolate your service when unit testing, you need to mock out each of these services. Thankfully, you don't need to mock any of the services provided by AngularJS, as the AngularJS Mocks library provides mocks for many of the services you will interact with. However, you need to mock the other services that are part of your application.

One of the things that makes mocking services easy is AngularJS' dependency injection functionality. If we need to mock out services in our application, we can define a mock service in our unit test; we can register this service with AngularJS to replace the other service.

The AngularJS module loader uses a last-in-wins methodology when it comes to defining services. So, when Jasmine loads up your application modules, it adds an entry in the dependency injection table as each module function is executed. You can overwrite your application services by registering your mock services after your application modules. This causes `$injector` to return your mock service instead of the application service when your service under test is created.

In the following code example, the `authenticate service` is dependent on the `log_service`, and we need to mock it out to isolate the `authenticate service` during the unit tests. To do this, we define a new module named `mock.log` in our test definition, and we define a service with the same name, `log_service`. Before each test, we make sure that we include the module after our application modules, so it will overwrite the existing `log_service`. Now, when the `authenticate service` is created, it will be given an instance of our mock `log_service`, allowing us to isolate the `authenticate service` during the unit test.

```
describe("authentication service", function(){
  var log;
  var service;

  angular.module('mock.log', []).factory('log_service',
    function () {
    log = {
      init: function () {
      },
      trace: function (message) {
      },
      debug: function (message) {
      },
      info: function (message) {
      },
      warn: function (message) {
      },
      error: function (message) {
      },
      fatal: function (message) {
      }
    };

    return log;
```


www.allitebooks.com

```
    });

    beforeEach(module('application.services'));
    beforeEach(module('mock.log'));

    beforeEach(inject(function (authenticate, log_service) {
      service = authenticate;
    }));

    // ...
    // test code
    // ...
  });
```

If you have a lot of services that need mocking, you can move the module definitions to a separate JavaScript file, but just make sure that you put the call to load the mock module after the call to load your application's service modules. Also, make sure that you don't mock out your service under test; it is best to have each mock service in a separate module called `mock.<servicename>` and then only load the mock services you need. If you don't take this precaution, your service will be replaced with the mock instance, and all of your unit tests will fail.

Mocking services with Jasmine spies

If you notice, the preceding mock `log_service` doesn't do anything. This is because we can take advantage of Jasmine's spies to provide the services' behavior during the unit test.

Jasmine's spies are pretty powerful. Not only can you replace the behavior of a service method, but you can also check to see if a service method was called, how many times it was called, and with what arguments it was called. Another good thing about spies is that they only exist inside the `describe` or `it` functions where they were defined, allowing you to provide different behavior for the same method in each different unit test if needed.

In the following unit test, we have defined a spy for the `trace` method of the `log_service` using the `spyOn` method, and passing in the instance of the service and the name of the method to watch. You can then test to see if the method was called using the `toHaveBeenCalled` expectation method provided by Jasmine.

```
    it("should call the log trace method when a method is called",
      function(){
      spyOn(log, "trace");
```

```
    service.logOut();

    expect(log.trace).toHaveBeenCalled();
  });
```

If you wanted to evaluate the parameters passed in to a method, you can use the `toHaveBeenCalledWith` expectation method as shown in the following code:

```
  it("should call log trace with the method name", function(){
    spyOn(log, "trace");

    service.logOut();

    expect(log.trace).toHaveBeenCalledWith("authenticate:logOut");
  });
```

If your mock service needs to return data, you can use the spy's `and.returnValue` method to return the required data to the calling service. The following unit test creates a spy for the `checkPassword` method of the `user` service instance and returns `true` whenever it is called during this scenario. This, in turn, should cause the service under test to return `true` for the `login` method being tested. Also, notice that the code checks whether the method was called and whether it was called with the same data passed into the `login` method.

```
  it("should call the return true for successful login",
    function(){
    spyOn(user, "checkPassword").and.returnValue(true);

    var result = service.logIn("test@nomail.com", "myPassword");

    expect(result).toBeTruthy();
    expect(user.checkPassword).toHaveBeenCalled();
    expect(user.checkPassword)
          .toHaveBeenCalledWith("test@nomail.com");
    expect(user.checkPassword).toHaveBeenCalledWith("myPassword");
  });
```

There are several other methods such as `callThrough`, `callFake`, and `throwError` that can be of use when mocking out services with Jasmine spies. To learn more about these methods and other different methods, check out the Jasmine documentation at `http://jasmine.github.io/`.

Handling dependencies that return promises

If you've spent some time writing services in AngularJS that interact with Web APIs, you have probably used the `$http` service. One of the prevalent code patterns that tends to develop involves calling the `then` method on the promise that is returned from the various `$http` methods. This, in effect, waits to execute the code until the asynchronous request returns and the promise is resolved.

The following is an example of such a pattern. The `retrieveUser` function makes a request from the server using the `$http` service's `get` method and then handles the response using the `then` method of the returned promise.

```
angular.module('brew-anywhere').factory('data_service', ['$http',
function ($http) {
  var data = {
    retrieveUser: function(userId){
      var url = '/users/' + userId;

      $http.get(url).then(function(response){
        return response.data.user;
      });
    }
  };

  return data;
});
```

Testing this code can be a bit problematic for newcomers. The following is a simple way to mock your services to provide similar functionality, so you can unit test the services or controllers that contain these code patterns. The `authenticate` service is dependent on `data_service` and the `checkUser` method call of `it`. The `retrieveUser` method of `data_service` returns a promise that is later resolved once the web request returns from the service.

```
describe("authentication service", function(){
  var data;
  var service;
  var q;
  var deferred;
  var rootScope;

  angular.module('test.data', []).factory('data_service',
    function (q) {
```

```
    data = {
      retrieveUser: function (email) {
        deferred = q.defer();
        return deferred.promise;
      }
    };

    return data;
  });

  beforeEach(module('application.services'));
  beforeEach(module('test.data'));

  beforeEach(inject(function (authenticate data_service,
                     $rootScope, $q) {
    rootScope = $rootScope;
    service = authenticate;
    q = $q;
  }));

  it("should call the retrieveUser", function(){
    spyOn(data, "retrieveUser").and.callThrough();;

    var email = 'test.user@nomail.com';
    service.checkUser(email);

    deferred.resolve({userName: email,
        firstName: Faker.Name.firstName(),
        lastName: Faker.Name.lastName()});

    rootScope.digest();

    expect(data.retrieveUser).toHaveBeenCalled();
  });
});
```

Although the unit test may seem to be a bit complicated, it is rather straightforward.

Initially, we create variables to hold a reference to `rootScope`, the mock service, the `$q` service, and the deferred object created when we call `q.defer()` in our mock service.

We need to hold on to a reference to the `$q` service from AngularJS so that we can call it whenever the `retrieveUser` method of our mock service is called.

We'll also need to hold on to the deferred object created by the call to `q.defer()` so that we can resolve the promise, which in turn will execute the `then()` clause in our service's code.

Next, we define our mock service in a `beforeEach()` call so that each time a test is executed, we have a clean version of our mock service.

Our mock service is pretty simple. First, we make sure that each of the properties and methods that the service will access is defined. In this case, it would be the `retrieveUser` method. We then mock out the `retrieveUser` method by creating a deferred object, storing the deferred object, and returning the promise.

Next, we make another call to `beforeEach()` that injects the services we need and store the services so that our mock service and test can call them when needed. Now, we can go about creating our unit test scenarios.

In the example scenario, we check to see if the `checkUser` method of the service calls the `retrieveUser` method of the `data_service`. To do this, we use Jasmine's `spyOn()` method, which intercepts the call to the `retrieveUser` method and tracks how many times the method is called. We also need to tell Jasmine what to do after it intercepts the call. We can either have it return a value, call a different function, or call through to the original implementation. In our case, we'll just have it call through to the original implementation of our mock service.

Next, we call the `checkUser` method on the service. This should invoke our mock service that, in turn, creates the deferred object, stores the deferred object, and returns the promise to the service, which then waits for the promise to be resolved.

We then resolve the promise by calling `deferred.resolve()`, and as we are outside the AngularJS world, we have to call the `digest()` method on `rootScope` to have AngularJS invoke the `then()` clause in our service. Once this happens, the code in the `then()` clause of our service will be executed, and we can execute our assertions to make sure that the code performed as we expected it to. In this case, we check to see if the `retrieveUser` method of the service was called.

You could use the same scenario to handle reject processing as well by calling `deferred.reject()` instead of `deferred.resolve()`.

As you've seen, writing unit tests for service methods that have code that handles promises is a bit more complicated, as you have to manage creating the promise and resolving it, but it becomes straightforward after a few tests. You will also see this pattern in your controller unit tests that call service methods that return promises, and you can use the same pattern for these unit tests as well. It is important that you remember to call `resolve` or `reject` on the deferred object and then call the `digest()` method of `rootScope` to ensure that the `then` method is triggered in your scenario as expected.

Mocking backend communications

One of the toughest things to do when testing a service is to mock the calls to external services. However, you're lucky; the AngularJS Mocks library provides a mock service that makes this a breeze.

The `$httpBackend` service provides an HTTP backend implementation that can be used in your service unit tests. The service can be used to respond with static or dynamic responses, using its `expect` or `when` methods.

The `expect` methods provide a way to make assertions about requests made by the service and define responses for these requests. The test will fail if the expected requests are not made or if they are made in the wrong order.

The `when` methods provide a fake backend for your service, which doesn't assert if a particular request was made or not; it just returns a canned response if a request is made. The test will pass whether or not the request is made during testing.

Both sets of methods take two mandatory parameters: the method being used in the request and the URL being called. They also take two optional parameters: the HTTP request body and the HTTP headers. If the parameters do not match when the call is made, an exception will be thrown, thus failing the test.

When any of the methods are called, they return an object with a `respond` method that can be used to return the status code, data, headers, and an optional status string for the call, thus allowing you to return whatever data and status you need for your unit test.

Both sets of methods also provide shortcuts for the `GET`, `HEAD`, `DELETE`, `POST`, `PUT`, and `JSONP` methods that allow you to reduce the amount of code when writing your tests.

Once the call to either the `expect` or `when` method is made, you need to call the `flush` method to complete the request and respond with the data defined in the respond method. The `flush` method can also take a count of requests to flush if you do not want to flush all the requests at once.

The `$httpBackend` mock also has two methods, `verifyNoOutstandingExpectation` and `verifyNoOutstandingRequest`, that can be used to ensure that the expected calls were made. The `verifyNoOutstandingExpectation` method ensures that all calls defined with the `expect` methods were made during the unit test. The `verifyNoOutstandingRequest` method ensures that there are no requests that need to be flushed. You will normally call these methods in an `afterEach` function so that they are checked after each unit test is executed.

The following unit test provides an example of how to use the `$httpBackend` mock service to set up a default handler using the `whenGet` method to handle any calls to `url '/users'` and respond with a list of users. The unit test sets up an expectation with a call to the `expectGet` method that expects a call to `url '/users/1'` and responds with a single user. If the call is not made, the call to `verifyNoOutstandingExpectation` will throw an exception.

```
describe("data service", function(){
  var httpBackend;
  var service;

  beforeEach(module('application.services'));

  beforeEach(inject(function (data_service, $httpBackend) {
    service = data_service;
    httpBackend = $httpBackend;
  }));

  beforeEach(function(){
    httpBackend.whenGet('/users').respond(200, [
      {id: 1, email: 'test1@nomail.com'},
      {id: 2, email: 'test2@nomail.com'},
      {id: 3, email: 'test3@nomail.com'}]);
  });

  it("should call the retrieveUser", function(){
    httpBackend.expectGet('/users/1')
     .respond(200, {id: 1, email: 'test1@nomail.com'});

    var users = [];
    service.retrieveUsers().then(function(response){
      users = response;

      service.retrieveUser(users[0].id);
      httpBackend.flush();
    });

    httpBackend.flush();
  });

  afterEach(function() {
    $httpBackend.verifyNoOutstandingExpectation();
    $httpBackend.verifyNoOutstandingRequest();
  });

});
```

Using the `$httpBackend` mock service, your unit tests can be written to ensure that your service code handles the various combinations of data returned by the external services it interacts with. You have total control of the status and data returned by the call, thus allowing you to exercise all the various paths in your code.

Mocking timers

One last thing we'll talk about is how to handle unit testing when your service uses either the `$timer` or `$interval` service. Calls to these services can cause a delay in executing your unit tests, and one of the things we want when executing unit tests is for them to execute as quickly as possible. This way, we get feedback as quickly as possible as we modify our code.

The AngularJS Mock library provides mock services for both of these services, allowing us to skip ahead in time so that the timeout interval passes quickly, thus allowing the test to execute as fast as possible. Both mock services provide a `flush` method that allows you to specify the number of milliseconds to skip ahead so that your timeout interval is triggered:

```
describe("polling service", function(){
  var interval;
  var service;
  var httpBackend

  beforeEach(module('application.services'));

  beforeEach(inject(function (poll_service, $interval,
    $httpBackend) {
    service = data_service;
    interval = $interval;
    httpBackend = $httpBackend;
  }));

  it("should poll the status method after 30 seconds", function(){
    httpBackend.expectGet('/status').respond(200, [
      {id: 1, message: 'test message'},
      {id: 2, message: 'test message'}]);

    var messages = [];
    service.init().then(function(response){
      messages = response;
      expect(messages.length).toBe(2);
    });

    interval.flush(30000);

    httpBackend.flush();
```

```
    });

    afterEach(function() {
      $httpBackend.verifyNoOutstandingExpectation();
      $httpBackend.verifyNoOutstandingRequest();
    });
  });
```

The preceding unit test provides an example of how to use the `$interval` mock service to skip 30 seconds ahead during our unit test, thus causing the service under test to make an HTTP request to `url '/status'`. As we did earlier, we add variables to the scenario to hold the `$httpBackend` and `$interval` mock services that are injected during the call to `beforeEach`. Then, during the scenario, we make a call to the `flush` method on the `$interval` service with a value of 30,000 milliseconds to skip 30 seconds ahead so that the timeout interval is triggered, and the HTTP request is made immediately.

Summary

In this chapter, we started out discussing how a Jasmine test scenario is structured. The `describe` method defines a scenario, and each specification is defined using the `it` method. We also touched upon setting things up for our unit tests, using the `beforeEach` method, as well as how to clean up after each unit test using the `afterEach` method.

We then discussed how to mock data and touched upon a couple of libraries that can be used to provide random data that is formatted based on common data patterns.

We then discussed how to use the AngularJS Mocks library's `module` method to replace the services we are dependent on with mock services, along with how to use Jasmine's spies to provide functionality for our mock service and ensure that the service methods are called as intended.

We then covered how to use the AngularJS Mocks library to handle Ajax calls placed by our services and how to skip ahead in time so that the services that use the `$timeout` and `$interval` services can execute as fast as possible in our unit tests.

As we cover the various services throughout this book, the source code will include unit tests for each service covered that will provide you with plenty of examples of how to unit test pretty much any type of service you might build.

In the next chapter, we are going to look at services that handle cross-cutting concerns that will be used to build our lowest layer of the sample application's architecture and will become critical to the rest of the services we build.

4
Handling Cross-cutting Concerns

A good application design uses layers to separate areas of responsibility. If done right, each layer has a single responsibility and it interconnects with the other layers using a well-defined interface.

The most popular layers you'll see included in an application are data, business logic, and the user interface. However, there are services that cut across all of the other layers; those that handle cross-cutting concerns such as messaging, logging, data validation, caching, internationalization, and security.

Communicating with your service's consumers using patterns

Communicating with your service's consumers becomes very important when you have methods that are long running or frontend asynchronous AJAX calls to a server. You do not want to block execution of your application while your service performs its calculations or waits for an AJAX call to return. It is best to handle all such methods in an asynchronous manner.

There are several ways in which you can handle such methods; you can request the consumer to provide a callback function that your service method executes upon completion, you could return a promise that your service method resolves upon completion, or you could use a messaging design pattern to notify the consumer once your service method completes.

www.allitebooks.com

Although callback methods and promises work really well for notifying a single consumer once your service method has completed, they fall way short when you have multiple consumers that need to know when your service's methods complete or when data managed by your service is updated.

This especially occurs when you have multiple views visible at the same time and each one needs to be notified when your service's data changes. A good example is the shopping cart in an e-commerce application. As the user adds items to the cart, the cart needs to update its display of the contents along with other services that might display shipping costs, order totals, and so on.

To handle such situations, you need to use a publish/subscribe design pattern to both handle notifying consumers when long-running methods complete and to notify all the views in your application when your service's data changes.

Callback methods and promises also do not provide the flexibility and loose coupling that is needed for large, complex applications. If a service needs to be replaced during the course of development, the code required to make the change ripples throughout the entire application. However, if we use a messaging pattern, the code changes are minimal since the new service is the only thing that needs to be changed, all of the consumers of the service can stay as they are without changes.

The publish/subscribe pattern is a messaging pattern that allows multiple consumers to subscribe and publish messages to other components without knowing the implementation details about those components. This lack of knowledge about the other components promotes loose coupling between the components and increases the maintainability of the application immensely.

Subscribers to messages get notified when an event occurs related to the subscribed message. For long-running processes or AJAX requests, three messages are usually involved: one to invoke the process or request, another to indicate when the process or request completes, and a third to notify consumers of a failure. Consumers will normally publish the start message to invoke the process or request and then subscribe to the complete and failure messages to be notified once the process or request has completed or failed. The services that implement the functionality or long-running process will subscribe to the message to start the process and publish the success and failure messages.

Many of the publish/subscribe implementations for AngularJS leverage the `$broadcast` and `$on` methods of the `rootScope` object, but there are performance issues that can occur when using these methods. Since the `$on` method is evaluated every time the `digest` loop of `rootScope` is executed, you can unknowingly impact the `digest` loop when your application has a lot of subscribers waiting for the message to be published. This happens because every call to the `$on` method of `rootScope` needs to be evaluated during every execution of the `digest` loop.

It is better to use a simple `publish/subscribe` messaging service that leverages callbacks. This reduces the strain on the `digest` loop of `rootScope` and keeps the number of items that need to be evaluated during the `digest` loop to a minimum.

One thing to keep in mind when leveraging callbacks for messaging is that you can slow down the performance of the application if your callback handlers publish other messages as part of their processing. So, it is best to use the `$timeout` service to wrap the code that publishes messages so the `callback` handler can return as quickly as possible and not affect the performance. You will see examples of this as we discuss some of the various notification services in this chapter.

The following is the code for the `messaging` service used by the sample application:

```
angular.module('brew-everywhere')
      .factory('messaging', function () {

    var cache = {};

    var subscribe = function (topic, callback) {
        if (!cache[topic]) {
            cache[topic] = [];
        }
        cache[topic].push(callback);
        return [topic, callback];
    };

    var publish = function (topic, args) {
        cache[topic] && angular.forEach(cache[topic],
          function (callback) {
            callback.apply(null, args || []);
        });
    };

    var unsubscribe = function (handle) {
       var t = handle[0];
       if (cache[t]) {
         for(var x = 0; x < cache[t].length; x++)
         {
           if (cache[t][x] === handle[1]) {
             cache[t].splice(x, 1);
           }
         }
       }
    };
```

```
        var service = {
            publish: publish,
            subscribe: subscribe,
            unsubscribe: unsubscribe
        };

        return service;
    });
```

You can find the code and unit tests in the samples for this book under the `service` folder.

This implementation of the publish/subscribe pattern is rather straightforward. Initially, we define an internal variable called `cache` that will be used to hold all of the subscribers for each of the messages and their callback methods.

The `subscribe` method takes a `topic` string and a `callback` function to invoke when the topic is published. It then checks to see whether the `cache` contains a property with the name of the topic; if it does not exist, a new property is added to the object and an array is created and assigned to the property. Then the callback function is inserted into the array. Finally, a handle is returned that the subscriber can use to unsubscribe as needed.

The `publish` method takes a `topic` string and an array of arguments that can be passed to each callback that subscribes to the topic. The method checks to see whether the cache object contains a property with the name of the topic and if so, it iterates through each item in the array and invokes the callback with the arguments passed into the function.

The `unsubscribe` method takes the handle returned by the `subscribe` method and removes the callback function associated with the topic. The method first checks to see whether the cache object contains a property with the name of the topic, and if so, it iterates through each item in the array until it finds a callback that matches the callback contained in the handle and then removes the callback from the array.

Finally, the service definition returns an object with just the methods we want to make public to consumers of the service, following the Revealing Module Pattern.

To use the service, a consumer needs to include the `messaging` service as a dependency and then subscribe to one or more topics that it's interested in being notified about when the event occurs. The following is a sample controller that uses the `messaging` service to authenticate a user:

```
angular.module('brew-everywhere').controller('LoginCtrl',
  function($scope, messaging, events){
  //#region Internal models
  $scope.username = '';
  $scope.password = '';
  $scope.currentUser = {};
  $scope.userAuthenticatedHandle = null;
  //#endregion Internal models

  //#region message handlers
  $scope.authenticateUserCompletedHandler = function(user) {
    $scope.currentUser = user;
  }

  $scope.userAuthenticatedHandle = messaging.subscribe(
    events.message._AUTHENTICATE_USER_COMPLETE_,
    $scope.authenticateUserCompletedHandler);
  //#endregion message handlers

  //#region view handlers
  $scope.$on('$destroy', function(){
    messaging.unsubscribe($scope.userAuthenticatedHandle);
  });

  $scope.onLogin = function() {
    messaging.publish(event.message._AUTHENTICATE_USER_,
      [$scope.username, $scope.password]);
  }
  //#endregion view handlers

});
```

The `LoginCtrl` view controller takes the `messaging` service and the `events` service as dependencies. The `messaging` service is the `publish/subscribe` service implementation that we covered earlier and the `events` service is a set of constants that defines the various messages that are used by the application.

The controller then defines a message handler called `authenticateUserCompletedHandler` that is registered with the `messaging` service with the topic `_AUTHENTICATE_USER_COMPLETE_`. It then defines the `onLogin` method that is used as an `ng-click` handler to log the user in by publishing the message `_AUTHENTICATE_USER_` with `username` and `password` as arguments for the topic subscribers.

When the `ng-click` handler is executed, it publishes the message and then the controller waits for the service, which handles the message, to respond with `_AUTHENTICATE_USER_COMPLETE_` message, which then takes the `user` object passed into the callback and assigns it the scope value `currentUser`.

The handling of errors is done outside of the controller by other services and controllers. If the `_AUTHENTICATE_USER_COMPLETE_` message is never received, the controller does nothing.

The controller also uses the `$scope.$on` method to watch for the `$destroy` message that is sent when the controller is being destroyed, so the controller can unsubscribe from any messages that it has subscribed to. This is important so the `messaging` service doesn't try and call a callback function that no longer exists.

This coding pattern keeps the controller as thin as possible and keeps the handling of errors where they belong—outside of the controller.

Managing user notifications

Now that we have the messaging layer of our application in place, we can build a couple more layers on it that handle cross-cutting concerns around user notifications.

The first thing we'll build is a simple `waitSpinner` directive that leverages the `messaging` service to display and hide an animated GIF indicating that our program is busy doing something:

```
angular.module('brew-everywhere').directive('waitSpinner',
function(messaging, events) {
    return {
    restrict: 'E',
    template: '<div class="row"><img src="images/ajax-loader.gif"
       alt="loading, please wait" /></div>',
    link: function(scope, element) {
      element.hide();

      var startRequestHandler = function () {
        // got the request start notification, show the element
        element.show();
```

```
        };

        var endRequestHandler = function() {
          // got the request start notification, show the element
          element.hide();
        };

        scope.startHandle = messaging.subscribe(
          events.message._SERVER_REQUEST_STARTED_,
          startRequestHandler);

        scope.endHandle = messaging.subscribe(
          events.message._SERVER_REQUEST_ENDED_,
          endRequestHandler);

        scope.$on('$destroy', function() {
          messaging.unsubscribe(scope.startHandle);
          messaging.unsubscribe(scope.endHandle);
        });

      }
    };
  });
```

The `waitSpinner` directive is pretty simple; it basically shows and hides the element it is attached to, based on one of two messages: `_SERVER_REQUEST_STARTED_` and `_SERVER_REQUEST_ENDED_`. The template for the directive is nothing more than a `div` that wraps an image tag, which has an animated image that is used to indicate the application is busy.

Now whenever we want to indicate the application is busy, we can publish the `_SERVER_REQUEST_STARTED_` message to display the wait spinner, and when we are done, we can publish the `_SERVER_REQUEST_ENDED_` message to hide it.

The next service we are going to look at is similar in nature, in that it shows and hides a dialog to get a response from the user. However, we use a service to intercept the message and then send out a different message based on the type of message to display. Another directive is waiting for the new messages and then shows either a pop-up dialog or confirmation message based on the message that is received. The following is the source code for the dialog service:

```
angular.module('brew-everywhere')
.factory('dialog',function($timeout, messaging, events) {
  var messageText = '';
  var displayType = 'popup';
```

```
  var displayDialogHandler = function(message, type){
    messageText = message;
    displayType = type;

    $timeout(function(){
      switch(displayType){
        case 'popup':
          messaging.publish(events.message._DISPLAY_POPUP_,
            [messageText]);
          break;
        case 'confirmation':
          messaging.publish(events.message._DISPLAY_CONFIRMATION_,
            [messageText]);
          break;
        default:
          messaging.publish(events.message._DISPLAY_POPUP_,
            [messageText]);
          break;
      }
    }, 0);
  };

  messaging.subscribe(events.message._DISPLAY_DIALOG_,
     displayDialogHandler);

  var init = function() {
    messageText = '';
    displayType = 'popup';
  };

  var dialog = {
    init: init
  };

  return dialog;
})
.run(function(dialog){
    dialog.init();
  });
```

The `dialog` service again has the `messaging` and `events` services as dependencies along with the `$timeout` service. The service consists of one message handler `displayDialogHandler`, which takes the message and type of dialog to display, and translates that into one of these two messages, `_DISPLAY_POPUP_` and `_DISPLAY_CONFIRMATION_`. The `$timeout` service is used to allow the service to return immediately from the message handler, keeping the application performance impact low as explained earlier.

The other strange thing you might notice is that we are using a `run` method on the module to invoke the `dialog` service's `init` method. This is because, since there is no direct interface for consumers to call, we have to invoke a method on the service to get it to load up when the application begins.

This is one of the shortcomings of the service-messaging pattern; the services are so loosely coupled that we have to add a method to get AngularJS to load the service so it can be used. However, this loose coupling provided by the messaging pattern allows us to replace the `dialog` service with any other notification mechanism that we might want, without having to change how we code our components. If we decide to use the native alert mechanisms to display the dialogs to the end user, we only need to write a new service that implements the same message handling as the `dialog` service and none of our other code in the application needs to change.

The `dialog` directive is the other part of the notification solution. It works much like the `waitSpinner` directive in that, it shows and hides the element it's attached to, but it also uses an `ng-switch` handler in the template to either display a simple pop up with an **OK** button or a confirmation dialog with a **YES** and **NO** button.

There are also `ng-click` handlers defined that are used to hide the notification dialog and send a message indicating what button the user clicked on:

```
angular.module('brew-everywhere')
  .directive('dialog', function(messaging, events) {
  return {
    restrict: 'E',
    replace: true,
    templateUrl: 'directive/dialog/dialog.html',
    link: function(scope, element) {
      element.hide();

      scope.modalType = 'popup';
      scope.message = '';

      var showPopupHandler = function (messageText) {
        // got the request start notification, show the element
        scope.message = messageText;
```

```
          scope.modalType = 'popup';
          element.show();
        };

        var showConfirmationHandler = function(messageText) {
          // got the request start notification, show the element
          scope.message = messageText;
          scope.modalType = 'confirmation';
          element.show();
        };

        scope.showPopupHandle = messaging.subscribe(
         events.message._DISPLAY_POPUP_,
         showPopupHandler);
        scope.showConfirmationHandle = messaging.subscribe(
         events.message._DISPLAY_CONFIRMATION_,
         showConfirmationHandler);

        scope.$on('$destroy', function() {
          messaging.unsubscribe(scope.showPopupHandle);
          messaging.unsubscribe(scope.showConfirmationHandle);
        });

        scope.answeredOk = function(){
          element.hide();
          messaging.publish(events.message._USER_RESPONDED_,
              ["OK"]);
        };

        scope.answeredYes = function(){
          element.hide();
          messaging.publish(events.message._USER_RESPONDED_,
              ["YES"]);
        };

        scope.answeredNo = function(){
          element.hide();
          messaging.publish(events.message._USER_RESPONDED_,
              ["NO"]);
        };
      }
    };
  });
```

The last user notification service we are going to cover is a simple notification list directive that works with two services to display errors and alerts to the user.

The error and notification services are similar, except each one handles a different message. This split is on purpose because of the single responsibility principle and it allows you to change how certain messages are handled. Suppose you need error messages to be displayed in a pop-up dialog instead of a notification list; since the error messages are handled by a separate service, you can easily change what message the error service publishes whenever it receives a new message.

Since the error and notification service are so similar, we'll just cover the error service:

```
angular.module('brew-everywhere')
  .factory('errors',function($timeout, messaging, events) {
  var errorMessages = [];

  var addErrorMessageHandler = function(message, type){
    if(!errorMessages){
      errorMessages = [];
    }

    errorMessages.push({type: type, message: message});

    $timeout(function() {
      messaging.publish(events.message._ERROR_MESSAGES_UPDATED_,
        errorMessages);
    }, 0);
  };

  messaging.subscribe(events.message._ADD_ERROR_MESSAGE_,
    addErrorMessageHandler);

  var clearErrorMessagesHandler = function() {
    errorMessages = [];
  };

  messaging.subscribe(events.message._CLEAR_ERROR_MESSAGES_,
    clearErrorMessagesHandler);

  var init = function(){
    errorMessages = [];
  };

  var errors = {
    init: init
```


www.allitebooks.com

```
  };

    return errors;
})
.run(function(errors){
    errors.init();
  });
```

You can review the notification service's code in the code samples for the book.

The service has an internal array that is used to hold the error messages as they are received by the service. The `addErrorMessageHandler` message handler uses the `$timeout` service to allow the method to return immediately and then the message handler publishes the `_ERROR_MESSAGES_UPDATED_` message with the contents of the `errorMessages` array.

The service also has a message handler to allow consumers to tell the service to clear the messages it contains, indicating the service consumer has handled them. Again this service provides an `init` method so that AngularJS can load it by using the module's `run` method.

The `notificationList` directive handles the `_ERROR_MESSAGES_UPDATED_` and `_USER_MESSAGES_UPDATED_` messages and then adds the received messages to an internal array and then publishes the comparable clear messages event.

The directive also has an `ng-click` handler that removes the clicked notification from the internal array. The directive also watches for the `$destroy` message so that it can unsubscribe from the messaging service:

```
angular.module('brew-everywhere')
  .directive('notificationList', function(messaging, events) {
  return {
    restrict: 'E',
    replace: true,
    templateUrl:
      'directive/notificationList/notificationList.html',
    link: function(scope) {
      scope.notifications = [];

      scope.onErrorMessagesUpdatedHandler =
       function (errorMessages) {
        if(!scope.notifications){
          scope.notifications = [];
        }
        scope.notifications.push(errorMessages);
```

```
          messaging.publish(events.message._CLEAR_ERROR_MESSAGES_);
        };

        messaging.subscribe(events.message._ERROR_MESSAGES_UPDATED_,
          scope.onErrorMessagesUpdatedHandler);

        scope.onUserMessagesUpdatedHandler =
         function (userMessages)  {
          if(!scope.notifications){
            scope.notifications = [];
          }
          scope.notifications.push(userMessages);
          messaging.publish(events.message._CLEAR_USER_MESSAGES_);
        };

        messaging.subscribe(events.message._USER_MESSAGES_UPDATED_,
          scope.onUserMessagesUpdatedHandler);

        scope.$on('$destroy', function() {
          messaging.unsubscribe(scope.errorMessageUpdateHandle);
          messaging.unsubscribe(scope.userMessagesUpdatedHandle);
        });

        scope.acknowledgeAlert = function(index){
          scope.notifications.splice(index, 1);
        };
      }
    };
  });
```

Logging application analytics and errors

We covered how to alert the user when things happen. Now, let's look at how you can build a `logging` service that you can use to alert you when things happen in your application.

Logging and tracing is another layer that cuts across all of the other layers in your application. The `logging` service we are going to cover uses a common JavaScript library called `log4javascript`. This library is very versatile and allows you to log either locally or remotely using the various appenders and layout formatters in the library. The following sample uses the basic `BrowserConsoleAppender`, which sends all output to the console window in your browser:

```
angular.module('brew-everywhere').factory('logging',
 function (messaging, events) {
  var log = null;
```

```
var init = function (logName) {
  log = log4javascript.getLogger(logName);
};

var setLogLevel = function (level) {
  log.setLevel(level);
};

var setLogAppender = function (appender) {
  log.addAppender(appender);
};

var trace = function (message) {
  log.trace(message);
};

messaging.subscribe(events.message._LOG_TRACE_, trace);

var debug = function (message) {
  log.debug(message);
};

messaging.subscribe(events.message._LOG_DEBUG_, debug);

var info = function (message) {
  log.info(message);
};

messaging.subscribe(events.message._LOG_INFO_, info);

var warn = function (message) {
  log.warn(message);
};

messaging.subscribe(events.message._LOG_WARNING_, warn);

var error = function (message) {
  log.error(message);
};

messaging.subscribe(events.message._LOG_ERROR_, error);

var fatal = function (message) {
  log.fatal(message);
```

```
    };

    messaging.subscribe(events.message._LOG_FATAL_, fatal);

    var service = {
      init: init,
      setLogLevel: setLogLevel,
      setLogAppender: setLogAppender
    };

    return service;

  })
    .run(['logging', function (logging) {
      logging.init('main');
      logging.setLogLevel(log4javascript.Level.ALL);
      logging.setLogAppender(new log4javascript.
  BrowserConsoleAppender());
    }]);
```

To learn more about log4javascript, visit `http://log4javascript.org`.

The `logging` service wraps the `log4javascript` library inside an AngularJS service by wrapping the various logging methods in service methods that are invoked as messages are received by the service.

There are three methods that we put in the service interface that allows you to configure the log level and log appender. In the preceding example, each one is called in a module's `run` method to ensure the service is loaded by AngularJS and to configure the logging instance with some default settings.

The following is an example of a controller that acts as a base controller for other controllers in the application. It provides methods on the scope to handle the various logging methods and the common messaging methods. It also handles cleaning up the message handler subscriptions when the controller is destroyed:

```
angular.module('brew-everywhere')
  .controller('BaseCtrl',function($scope, messaging, events){
  //#region logging methods
  $scope.trace = function(message){
    messaging.publish(events.message._LOG_TRACE_, [message]);
  };
```

```
    $scope.traceDebug = function(message){
      messaging.publish(events.message._LOG_DEBUG_, [message]);
    };

    $scope.traceInfo = function(message){
      messaging.publish(events.message._LOG_INFO_, [message]);
    };

    $scope.traceWarning = function(message){
      messaging.publish(events.message._LOG_WARNING_, [message]);
    };

    $scope.traceError = function(message){
      messaging.publish(events.message._LOG_ERROR_, [message]);
    };

    $scope.traceFatal = function(message){
      messaging.publish(events.message._LOG_FATAL_, [message]);
    };
    //#endregion logging methods

    //#region messaging vars and methods
    $scope.messagingHandles = [];

    $scope.subscribe = function(topic, callback){
      var handle = messaging.subscribe(topic, callback);

      if(handle)
      {
        $scope.messagingHandles.push(handle);
      }
    };

    $scope.publish = function(topic, data){
      messaging.publish(topic, data);
    };

    $scope.$on('$destroy', function(){
      angular.foreach($scope.messagingHandles, function(handle){
        messaging.unsubscribe(handle);
      });
    });
    //#endregion messaging vars and methods
  });
```

With this new base controller, we can change the login controller and remove some of the code clutter in the controller. You can also see where we are calling the `traceInfo` method on the scope to log events as our program executes:

```
angular.module('brew-anywhere').controller('LoginCtrl',
  function($scope, $controller, events){
  // this call to $controller adds the base controller's methods
  // and properties to the controller's scope
  $controller('BaseCtrl', {$scope: $scope});

  //#region Internal models
  $scope.username = '';
  $scope.password = '';
  $scope.currentUser = {};
  $scope.userAuthenticatedHandle = null;
  //#endregion Internal models

  //#region message handlers
  $scope.authenticateUserCompletedHandler = function(user) {
    $scope.currentUser = user;
    $scope.traceInfo("authenticateUserCompletedHandler received: "
       + angular.toJson(user));
  };

  $scope.subscribe(events.message._AUTHENTICATE_USER_COMPLETE_,
     $scope.authenticateUserCompletedHandler);
  //#endregion message handlers

  $scope.onLogin = function() {
    $scope.traceInfo("onLogin
       authenticating user: " + $scope.username);
    $scope.publish(events.message._AUTHENTICATE_USER_,
       [$scope.username, $scope.password]);
  };
});
```

A couple of things to note about the changes we have made to the controller; we no longer need to include the messaging service in the login controller's dependency list since the base controller has the messaging service in its dependency list and all of the calls to the messaging service are wrapped in methods that are created as part of the base controller's scope. Second, we now have a dependency on the `$controller` service that is used to bind the base controller's functionality onto the child controller's scope.

Authentication using OAuth 2.0

The final cross-cutting service layer we are going to talk about is the authentication layer. The authentication layer is responsible for ensuring that the people interacting with our application are the people they say they are. There are several ways that someone can authenticate himself or herself with our application. They can use a basic username and password, they could use a certificate, or they can use an external authentication server that supports protocols such as OAuth and OpenID or they can use a two-factor authentication service such as Single Sign-On.

We are going to look at a service that frontends both a basic authentication service and OAuth 2.0 using the Google+ API. Since we haven't covered data services yet, the basic authentication part of the service will always return success with dummy data. We will complete this service in *Chapter 5, Data Management*.

The following `authenticate` service acts as a frontend for the various authentication schemes our application will provide. It leverages the messaging service to receive the requests to use the various authentication services and then sends messages to invoke the login methods of the requested support services.

```
angular.module('brew-everywhere').factory('authenticate',
 function (messaging, events) {
    var currentUser = {};

    var loginWithGooglePlus = function(){
      messaging.publish(events.message._SERVER_REQUEST_STARTED_);
      messaging.publish(events.message._GPLUS_AUTHENTICATE_);
    };

    messaging.subscribe(events.message._AUTHENTICATE_USER_GPLUS_,
      loginWithGooglePlus);

    var googlePlusAuthenticatedHandler = function(user) {
      currentUser = user;
      messaging.publish(
        events.message._AUTHENTICATE_USER_COMPLETE_,
        [currentUser]);
      messaging.publish(events.message._SERVER_REQUEST_ENDED_);
    };

    messaging.subscribe(events.message._GPLUS_AUTHENTICATED_,
      googlePlusAuthenticatedHandler);

    var authenticationFailureHandler = function() {
      messaging.
        publish(events.message._AUTHENTICATE_USER_FAILED_);
      messaging.publish(events.message._SERVER_REQUEST_ENDED_);
      messaging.publish(events.message._ADD_ERROR_MESSAGE_,
```

```
        ['Log In Failed.', 'alert-warning']);
    };

    messaging.subscribe(events.message._GPLUS_FAILED_,
      authenticationFailureHandler);

    var login = function(username, password){
      currentUser = {name: {givenName: "Test", surname: "User"}};
      messaging.publish(
        events.message._AUTHENTICATE_USER_COMPLETE_,
        [currentUser]);
    };

    messaging.subscribe(events.message._AUTHENTICATE_USER_,
      login);

    var init = function(){
      currentUser = {};
    };

    var authenticate = {
      init: init
    };

    return authenticate;
  })
  .run(function(authenticate){
    authenticate.init();
  });
```

The service responds to two messages `_AUTHENTICATE_USER_GPLUS_` and `_AUTHENTICATE_USER_`.

The `_AUTHENTICATE_USER_` message invokes a stubbed-out method, which ignores the passed-in data and creates a user object with some dummy data in it and then responds with the `_AUTHENTICATE_USER_COMPLETE_` message indicating the user has been authenticated.

The `_AUTHENTICATE_USER_GPLUS_` message invokes the `googlePlusAuthentication` service, which wraps the Google+ Sign-On API library that handles redirecting the user to Google+, allowing them to give the application permissions to access the user's Google+ account information as a means of authenticated identity.

The `googlePlusAuthentication` service handles the `_GPLUS_AUTHENTICATE_` message, which kicks the Google+ authentication process off with a call to the `gapi.auth.authorize` method passing in the client ID of your Google+ application. Once the library returns, the callback handler parses the response and then proceeds to request the user's Google+ profile, which is returned in the `_GPLUS_AUTHENTICATED_` message.

If the user fails to authenticate with Google+ or decides not to give permissions to our application, then the `gapi.auth.authorize` method returns a failure response that causes the service to publish the `_GPLUS_FAILED_` message. The source code for the `googlePlusAuthentication` service is rather long so only a few of the methods are given in the following code:

```
var login = function () {
  var parameters = {
    client_id: defaults.clientId,
    immediate: false,
    response_type: defaults.responseType,
    scope: defaults.scope
  };
  gapi.auth.authorize(parameters, handleResponse);
};

var handleResponse = function (oauthToken) {
  parseToken(oauthToken);
  if (accessResults.access_token) {
    requestUserInfo();
  } else {
    userProfileErrorHandler();
  }
};

var userProfileSuccessHandler = function (response) {
  if (response && response.data) {
    userInfo = response.data;
    messaging.publish(events.message._GPLUS_AUTHENTICATED_,
      [userInfo]);
  }
};

var userProfileErrorHandler = function () {
  messaging.publish(events.message._GPLUS_FAILED_);
};
```

```
var requestUserInfo = function () {
  var url = buildUserProfileUrl();

  $http.get(url).then(userProfileSuccessHandler,
    userProfileErrorHandler);
};

var onAuthenticateHandler = function() {
  login();
};

messaging.subscribe(events.message._GPLUS_AUTHENTICATE_,
  onAuthenticateHandler);
```

You can review the complete source in the source code for this book.

To learn more about using the Google+ API and how to set up a Google+ application that can be used with the Google+ API, check out the Google+ JavaScript API at `https://developers.google.com/+/web/api/javascript`. The documentation provides you with full coverage of using the API and provides links on how to set up a Google+ developer account and a Google+ application, which is required to use the API.

Summary

In this chapter, we covered some of the various cross-cutting services that can be used to build out a well-structured AngularJS application. We discussed how messaging plays an important part in the communications between the various components in our application. We then built out various services using the publish/subscribe messaging pattern to handle notifications, logging, and authentication.

As I mentioned in *Chapter 2*, *Designing Services*, by sticking to the single responsibility principle when we design our services, we would end up with a large number of services in our application and this chapter was a great example of this principle. Each service covered one area of responsibility messaging, errors, notifications, dialogs, logging, authentication, and external authentication.

We also saw that by using the publish/subscribe messaging pattern, our services were so loosely coupled that we had to provide the `init` methods to ensure AngularJS would load them into our application. We also discussed that because of that loose coupling, we are able to replace a service without major code changes. This is something that becomes very important the more complex your application becomes.

As you review the unit tests for these services, you'll see that keeping to a single area of responsibility and the loose coupling provided by the publish/subscribe messaging pattern makes our services very easy to test. As we cover more application layers and discuss more services in the following chapters, you'll see this same pattern repeat itself over and over.

5
Data Management

Every application is dependent on data; no matter what type of application you write, be it a weather app, a stock quote app, or a line-of-business app, they all use data in one form or another.

Managing the data in your application is one of the most basic things you'll have to do. If you are pulling data directly from a REST-based service or retrieving it as a response to an API, you are going to need some sort of a data management layer. Services provide a great way of organizing the models and code that will comprise your data management layer.

In this chapter, we're going to cover a couple of strategies on how to implement the usual create, read, update, and delete functionalities common to a typical data management layer, along with some ways on how to cache and transform data if needed.

Models provide the state and business logic

A best practice when dealing with data in an object-oriented way is to build classes that both maintain the state of an object and provides business logic to manipulate the class' state. Although JavaScript was not designed with many features of an object-oriented language, many people have written books such as *Object-Oriented JavaScript*, *Stoyan Stefanov*, *Packt Publishing* and *Principles of Object-Oriented Programming in JavaScript*, *Nicholas C. Zakas*, *No Starch Press*, which show how to apply object-oriented principles to JavaScript in order to allow us to benefit from those best practices.

K. Scott Allen, author of *Professional ASP.NET MVC 4*, *Wrox*, and numerous pluralsight courses, wrote a great article, *Building Better Models for AngularJS* at `http://odetocode.com`, which describes a good way to do this using AngularJS `values` to define both your model and the associated code to encapsulate everything into an easy-to-use service that your other AngularJS components can use. We discussed the basic concept for this in *Chapter 2*, *Designing Services*, and will touch on it again here since we can use it to help build our data management layer services.

You define the models used in your application by defining the model using a function closure. Then, you use prototypal inheritance to define the business logic code associated with the model, and finally, you use the AngularJS `value` method of the module component to make the model injectable into your components.

The following is the definition of the `Fermentable` model used in the sample application. As you can see, the model properties are defined in the enclosure, and two methods, `getGravityPerPound` and `ingredientGravity`, are added to the model's prototype:

```
var Fermentable = function() {
  var self = this;
  self.name = '';
  self.type = ''; // can be "Grain", "Sugar", "Extract", "Dry
                  // Extract" or "Adjunct"
  self.amount = 0.0;
  self.yield = 0.0; // percent
  self.color = 0.0; // Lovibond units (SRM)
  self.addAfterBoil = false;
  self.origin = '';
  self.supplier = '';
  self.notes = '';
  self.coarseFineDiff = 0.0; // percent
  self.moisture = 0.0; // percent
  self.diastaticPower = 0.0; // in lintner units
  self.protein = 0.0; // percent
  self.maxInBatch = 0.0; // percent
  self.recommendMash = false;
  self.ibuGalPerPound = 0.0;
  self.displayAmount = '';
  self.potential = 0.0;
  self.displayColor = 0.0;
  self.extractSubstitute = '';
};

Fermentable.prototype = {
  /**
```

```
     * calculates the gravity per pound for the fermentable
     * @param brewHouseEfficiency - the estimated brew house
     * efficiency
     * @returns {number} - potential extract per pound for the
     * fermentable
     */
    gravityPerPound: function(brewHouseEfficiency){
      return ((this.potential - 1) * 1000) * brewHouseEfficiency;
    },
    /**
     * calculates the gravity for the ingredient
     * @param brewHouseEfficiency - the estimated brew house
     * efficiency
     * @returns {number} - returns the total potential extract for
     * the fermentable
     */
    ingredientGravity: function(brewHouseEfficiency){
      return this.amount *
         this.gravityPerPound(brewHouseEfficiency);
    }
  }

  angular.module('brew-everywhere').value('Fermentable',
  Fermentable);
```

To use the model in your controllers, directives, or services, you can include the name of the model in your component definition function. The following is an example of injecting the preceding `Fermentable` model into the application's `fermentableDataService`:

```
angular.module('brew-anywhere').factory('fermentableDataService',
  function(messaging, events, mongolab, modelTransformer,
           Fermentable) {
…
});
```

If you need to create a new instance of the model, you can use JavaScript's `new` operator as follows:

```
$scope.fermentable = new Fermentable();
```

K. Scott Allen also provides a generic translation service that allows you to transform the plain JSON objects received from external data services into your model object. The `modelTransformer` services takes the JSON object and a model value and then uses the AngularJS extend method to apply the model to the JSON object:

```
angular.module('brew-everywhere')
  .factory('modelTransformer',function() {

  var transformObject = function(jsonResult, constructor) {
    var model = new constructor();
    angular.extend(model, jsonResult);
    return model;
  };

  var transformResult = function(jsonResult, constructor) {
    if (angular.isArray(jsonResult)) {
      var models = [];
      angular.forEach(jsonResult, function(object) {
        models.push(transformObject(object, constructor));
      });
      return models;
    } else {
      return transformObject(jsonResult, constructor);
    }
  };

  var modelTransformer = {
      transform: transformResult
  };

  return modelTransformer;
});
```

We can use this in our data services to transform the received JSON data into our model object simply by using the following code:

```
var model = modelTransformer.transform(fermentable, Fermentable)
```

With our models defined as `values` and a transformation service, we can turn our attention to the various data services that we'll need in our application.

Implementing a CRUD data service

Our application stores its data in a `mongodb` database hosted by `https://mongolab.com`. There are 10 different collections that we have to interact with, and these include adjuncts, brewers, equipment, fermentables, hops, mashprofiles, recipes, styles, waterprofiles, and yeast.

We need to create a data service for each one of these collections, but instead of duplicating the code to interact with the `MongoLab` REST API, it's better to create a single generic service that interacts with `MongoLab` and then create a data service for each collection that calls the `MongoLab` API with the specific details for each collection.

The `MongoLab` REST API consists of the following calls:

```
List All – Get /databases/{database}/collections/{collection}
Get One - Get /databases/{database}/collections/{collection}/{_id}
Create – Post /databases/{database}/collections/{collection/
Update - Put /databases/{database}/collections/{collection}/{_id}
Delete - Del /databases/{database}/collections/{collection}/{_id}
```

As you can see, the REST API follows the standard format as expected with a few dynamic parameters needed to build the URL:

- `{database}`: This is the name of the database that contains the collection
- `{collection}`: This is the name of the collection that contains the object
- `{_id}`: This is the unique identifier of the object to operate on

The API also requires that an API key be provided as part of the URL by appending `?apiKey={apikey}` to each URL, where `apikey` is the API key associated with the `MongoLab` database.

The following is the code for the `mongolab.com` data service:

```
angular.module('brew-everywhere')
  .factory('mongolab', function ($http) {
  var apiKey = '';
  var baseUrl = 'https://api.mongolab.com/api/1/databases';

  var setApiKey = function (apikey) {
    apiKey = apikey;
  };
  var getApiKey = function () {
    return apiKey;
  };
```

```
var setBaseUrl = function (uri) {
  baseUrl = uri;
};
var getBaseUrl = function () {
  return baseUrl;
};
var query = function (database, collection, parameters) {
  parameters = parameters || {};
  parameters['apiKey'] = apiKey;
  var uri = baseUrl + '/' + database + '/collections/' +
            collection;
  return $http({method: "GET", url: uri, params: parameters,
              cache: false});
};
var queryById = function (database, collection, id, parameters)
{
  parameters = parameters || {};
  parameters['apiKey'] = apiKey;
  var uri = baseUrl + '/' + database + '/collections/' +
            collection + '/' + id;
  return $http({method: "GET", url: uri, params: parameters,
              cache: false});
};
var createObject = function (database, collection, object) {
  var uri = baseUrl + '/' + database + '/collections/' +
            collection + '?apiKey=' + apiKey;
  return $http({method: "POST", url: uri, data:
              JSON.stringify(object), cache: false});
};
var updateObject = function (database, collection, object) {
  var uri = baseUrl + '/' + database + '/collections/' +
            collection + '/' + object._id.$oid + '?apiKey=' +
            apiKey;
  return $http({method: "PUT", url: uri, data:
              JSON.stringify(object), cache: false});
};
var deleteObject = function (database, collection, object) {
  var uri = baseUrl + '/' + database + '/collections/' +
            collection + '/' + object._id.$oid + '?apiKey=' +
            apiKey;
  return $http({method: "DELETE", url: uri, cache: false});
};

var mongolab = {
```

```
      setApiKey: setApiKey,
      getApiKey: getApiKey,
      setBaseUrl: setBaseUrl,
      getBaseUrl: getBaseUrl,
      query: query,
      queryById: queryById,
      create: createObject,
      update: updateObject,
      delete: deleteObject
    };

    return mongolab;
  });
```

The `mongolab` service is pretty straightforward; each CRUD method essentially builds the URL for the REST API and then returns a promise from the `$http` service call to which the required parameters for the specific call are passed. The service also includes methods to allow the service to be configured with the appropriate API key and base URL.

One thing you might notice about this service is that the methods return a promise instead of using the messaging service we built out in *Chapter 4*, *Handling Cross-cutting Concerns*. This is because of the generic nature of the service. Since we cannot tell what type of object is being returned from the service, we need to be sure that the service that makes the call to the `mongolab` service method handles the response from the `$http` service call, so we can transform it into the appropriate data model. If we try to use the messaging service, we would have every one of the services reacting to the completion messages. Then, we would have to add extra code to the data services, so they could figure out if the model can be transformed to that service's model type, which complicates things way too much.

Another thing to note is that we use the `JSON.stringify` call instead of AngularJS' `toJSON` method, based on how `MongoLab` formats the returned objects. The `MongoLab` API appends the unique identifier for each object using the `_id` property, which contains an object that has a `$oid` property that stores the object's unique identifier. This `$oid` property gets stripped out of the JSON string by the AngularJS' `toJSON` method due to the fact that AngularJS sees any property prefixed with a dollar (`$`) sign as an internally-added property, and strips it from the object's JSON representation. If the `$oid` property is stripped out of the object's JSON representation, `MongoLab` will refuse to do anything with the provided object.

Now that we have wrapped the `MongoLab` REST API with our `mongolab` service, we can turn our sights to building out each of the data services for each type of data that we'll use in our application.

Each service consists of a core set of methods: `get{type}`, `get{type}ById`, `create{type}`, `update{type}`, and `delete{type}`, where `{type}` is one of the collection object types mentioned earlier. The service also uses the `messaging` service to react to events to execute the associated service calls and respond once the `mongolab` service call completes.

The following is the code for one of the methods of the `adjunctDataService`. Each core method follows a similar pattern; a method is provided to handle the execute event. Two `$http` response handler methods are created, one for success and one for an error. Finally, a call to the `messaging` service to subscribe to the execute event is made.

Each success handler method uses the `modelTransformer` service to convert the returned object from the raw JSON response to the appropriate model type and publishes a request complete message.

Each error-handler method publishes a request failed message along with a message to the error service to add a message so that the user is notified that the request has failed. This message could be expanded further by taking the error response as a parameter to the method and sending the error message to the error service instead of a canned message:

```
angular.module('brew-everywhere')
  .factory('adjunctDataService', function (messaging, events,
     mongolab, modelTransformer, Adjunct) {
  var adjuncts = [];

  var getAdjuncts = function () {
    mongolab.query('breweverywhere_backup', 'adjuncts', [])
      .then(getAdjunctSuccessHandler, getAdjunctErrorHandler);
  };

  var getAdjunctSuccessHandler = function (response) {
    if (response.length > 0) {
      var result = [];
      angular.forEach(response, function (adjunct) {
        result.push(modelTransformer.transform(adjunct, Adjunct));
      });
      messaging.publish(events.message._GET_ADJUNCTS_COMPLETE_,
        [result]);
    } else {
      getAdjunctErrorHandler();
    }
  };
```

```
    var getAdjunctErrorHandler = function(){
      messaging.publish(events.message._GET_ADJUNCTS_FAILED_);
      messaging.publish(events.message._ADD_ERROR_MESSAGE_,
        ['Unable to get adjuncts from server', 'alert.error']);
    };

    messaging.subscribe(events.message._GET_ADJUNCTS_, getAdjuncts);
    …
    …
    var init = function(){
      adjuncts = [];
    };

    var adjunctDataService = {
      init: init,
      getAdjuncts: getAdjuncts,
      getAdjunctById: getAdjunctById,
      createAdjunct: createAdjunct,
      updateAdjunct: updateAdjunct,
      deleteAdjunct: deleteAdjunct
    };

    return adjunctDataService;
  });
```

All of the other data services follow this same pattern; you can check them out in the source code for the book under the `service` folder. There is also a `ReadMe.md` file that provides information on how to create an account on `https://mongolab.com/`, create the database, and how to populate each of the collections using the files in the `data` directory.

Caching data to reduce network traffic

One of the big issues with using external data services is finding ways to cache data on the client to reduce the number of network calls made during the lifetime of the application. AngularJS actually provides a service that allows you to cache objects on the client. The `$cacheFactory` service provides you with the ability to set up a named cache on the client that allows you to save data on the client using named value pairs.

You can provide some simple caching to your data services using the `$cacheFactory` service. What you'll need to do in your service is get a reference to the cache instance and then check it prior to making a call to the `mongolab` service. When you create a new object in the collection, you can insert the returned item into the collection that is stored in the cache so that it contains the current representation of the collection on the server. You can also do the same thing for all updates and deletes made against the collection as well.

The following is the `getAdjuncts` method of the `adjunctDataService` that was changed to implement simple caching using the `$cacheFactory` service. Notice how the success handler is changed to update the cache as needed based on the call made. Also, notice how the `getAdjuncts` method is modified to check the cache prior to making a call to the `mongolab` service:

```
angular.module('brew-everywhere')
  .factory('adjunctDataService', function ($cacheFactory,
     messaging, events, mongolab, modelTransformer, Adjunct) {
  var cache = $cacheFactory('brew-everywhere');

  var getAdjuncts = function () {
    var adjunctList = cache.get('adjuncts');

    if(adjunctList && adjunctList.length > 0){
      messaging.publish(events.message._GET_ADJUNCTS_COMPLETE_,
        [adjunctList]);
      return;
    }

    mongolab.query('breweverywhere_backup', 'adjuncts', [])
      .then(getAdjunctSuccesHandler, getAdjunctErrorHandler);
  };

  var getAdjunctSuccesHandler = function (response) {
    if (response.length > 0) {
      var result = [];
      angular.forEach(response, function (adjunct) {
        result.push(modelTransformer.transform(adjunct, Adjunct));
      });
      cache.put('adjuncts', result);
      messaging.publish(events.message._GET_ADJUNCTS_COMPLETE_,
        [result]);
    } else {
      getAdjunctErrorHandler();
    }
  };
```

Now, the `$cacheFactory` service works great for caching data in your AngularJS application while the application is running, but it does nothing for caching data between instances of your AngularJS application. To cache data across instances of your AngularJS application, you need to look at taking advantage of the browser's local storage mechanism. Luckily, there is an open source project in the AngularJS community called `angular-local-storage`, which provides you with easy access to the browser's local storage in a similar manner as we did with the `$cacheFactory` service. You can find the service on GitHub at `https://github.com/grevory/angular-local-storage`.

To use the service, you need to include the module's identifier, `'LocalStorageModule'`, in your application's dependency list and then add the service `'localStorageService'` to your component that is going to use the service. The following is the `adjunctDataService` modified to use `angular-local-storage`:

```
angular.module('brew-everywhere')
  .factory('adjunctDataService', function (localStorageService,
     messaging, events, mongolab, modelTransformer, Adjunct) {

  var getAdjuncts = function () {
    var adjunctList = localStorageService.get('adjuncts');

    if(adjunctList && adjunctList.length > 0){
      messaging.publish(events.message._GET_ADJUNCTS_COMPLETE_,
        [adjunctList]);
      return;
    }

    mongolab.query('breweverywhere_backup', 'adjuncts', [])
      .then(getAdjunctSuccessHandler, getAdjunctErrorHandler);
  };

  var getAdjunctSuccessHandler = function (response) {
    if (response.length > 0) {
      var result = [];
      angular.forEach(response, function (adjunct) {
        result.push(modelTransformer.transform(adjunct, Adjunct));
      });
      localStorageService.set('adjuncts', result);
      messaging.publish(events.message._GET_ADJUNCTS_COMPLETE_,
        [result]);
    } else {
      getAdjunctErrorHandler();
    }
  };
```

Notice how the `getAdjuncts` method now pulls the adjunct list from the `localStorageService`, and if the list exists, it returns the list without calling the `mongolab` service. The `getAdjunctSuccessHandler` has also been updated to store the resulting adjunct list in local storage using the set method of `localStorageService`.

The previous solutions to cache data on the client are two examples of ways you can implement caching. However, there is a lot of logic that needs to be addressed as new objects are created, updated, and deleted from the collection, which in turn means a lot of additional code that is outside the scope of this discussion.

If you need a robust data-caching solution, the BreezeJS library, created by Idea Blade, addresses all of this logic in a nice, easy-to-use library with a bunch of additional features. BreezeJS not only provides client-side caching and storage, but also handles dirty checking of client-side data as well as online and offline checking, which comes in handy for applications targeted at mobile devices. Since BreezeJS is a bit more complicated to implement, we are not going to cover it in this book, but you can find more information, tutorials, and examples on how to integrate BreezeJS in your AngularJS application at `http://breezejs.com`.

Transforming data in the service

Sometimes, you need to return a subset of your data for a directive or controller, or you need to translate your data into another format for use by an external service. This can be handled in several different ways; you can use AngularJS filters or you could use an external library such as underscore or lodash.

How often you need to do such transformations will help you decide on which route you take. If you are going to transform data just a few times, it isn't necessary to add another library to your application; however, if you are going to do it often, using a library such as underscore or lodash will be a big help.

We are going to limit our discussion to using AngularJS filters to handle transforming our data. Filters are an often-overlooked component in the AngularJS arsenal. Often, developers will end up writing a lot of methods in a controller or service to filter an array of objects that are iterated over in an `ngRepeat` directive, when a simple filter could have easily been written and applied to the `ngRepeat` directive and removed the excess code from the service or controller.

First, let's look at creating a filter that will reduce your data based on a property on the object, which is one of the simplest filters to create. This filter is designed to be used as an option to the `ngRepeat` directive to limit the number of items displayed by the directive.

The following `fermentableType` filter expects an array of fermentable objects as the input parameter and a `type` value to filter as the `arg` parameter. If the fermentable's `type` value matches the `arg` parameter passed into the filter, it is pushed onto the resultant array, which will in turn cause the object to be included in the set provided to the `ngRepeat` directive.

```
angular.module('brew-everywhere')
  .filter('fermentableType', function () {
  return function (input, arg) {
    var result = [];

    angular.forEach(input, function(item){
      if(item.type === arg){
        result.push(item);
      }
    })

    return result;
  };
});
```

To use the filter, you include it in your partial in an `ngRepeat` directive as follows:

```
<table class="table table-bordered">
  <thead>
    <tr>
      <th>Name</th>
      <th>Type</th>
      <th>Potential</th>
      <th>SRM</th>
      <th>Amount</th>
      <th> </th>
    </tr>
  </thead>
  <tbody>
    <tr ng-repeat="fermentable in fermentables |
        fermentableType:'Grain'">
      <td class="col-xs-4">{{fermentable.name}}</td>
      <td class="col-xs-2">{{fermentable.type}}</td>
      <td class="col-xs-2">{{fermentable.potential}}</td>
      <td class="col-xs-2">{{fermentable.color}}</td>
    </tr>
  </tbody>
</table>
```

The result of calling `fermentableType` with the value, `Grain` is only going to display those fermentable objects that have a `type` property with a value of `Grain`.

Using filters to reduce an array of objects can be as simple or complex as you like. The next filter we are going to look at is one that uses an object to reduce the fermentable object array based on properties in the passed-in object.

The following `filterFermentable` filter expects an array of fermentable objects as an input and an object that defines the various properties and their required values that are needed to return a matching object. To build the resulting array of objects, you walk through each object and compare each property with those of the object passed in as the `arg` parameter. If all the properties match, the object is added to the array and it is returned.

```
angular.module('brew-everywhere')
.filter('filterFermentable', function () {
  return function (input, arg) {
    var result = [];

    angular.forEach(input, function (item) {
      var add = true
      for (var key in arg) {
        if (item.hasOwnProperty(key)) {
          if (item[key] !== arg[key]) {
            add = false;
          }
        }
      }
      if (add) {
        result.push(item);
      }
    });

    return result;
  };
});
```

To use the filter, you include it in your partial in an `ngRepeat` directive as follows:

```
<table class="table table-bordered">
    <thead>
    <tr>
        <th>Name</th>
        <th>Type</th>
        <th>Potential</th>
```

```
          <th>SRM</th>
          <th>Amount</th>
          <th> </th>
      </tr>
      </thead>
      <tbody>
        <tr ng-repeat="fermentable in fermentables |
          filterFermentable:{type:'Grain', color: 3}">
          <td class="col-xs-4">{{fermentable.Name}}</td>
          <td class="col-xs-2">{{fermentable.Type}}</td>
          <td class="col-xs-2">{{fermentable.Potential}}</td>
          <td class="col-xs-2">{{fermentable.Color}}</td>
        </tr>
      </tbody>
  </table>
```

The preceding use of the `filterFermentable` filter will cause the array of fermentables to only include those with a `type` of `Grain` and `color` of `3`.

These are great examples of how you can build filters to reduce your dataset, and you can also use these same filters in your services. First, you should include the filter in your dependency list for the service, and then, you can invoke the filter in your service method to return a reduced dataset, as shown in the following sample method:

```
angular.module('brew-everywhere')
  .factory('fermentableDataService',
  function(messaging, events, mongolab, modelTransformer,
           Fermentable, fermentableType) {
  var fermentables = [];

  var filterByType = function(type){
    return fermentableType(fermentables, type);
  }
  }
```

In the preceding example, the array being operated on is populated by the data service's other method, `getFermentables`, and when the `filterByType` method is called, the service uses the `fermentableType` filter to reduce the array to the desired result.

Another thing you can do is translate an array of objects into a different type or reduced object. This works the same way as the two preceding filters, except that you create a new object and push it onto the resultant array. The following is an example of a filter that translates the array provided in the input parameter and returns a set of new objects that only returns the fermentable's `name` and `type`:

```
angular.module('brew-everywhere')
  .filter('translateFermentableToDisplay', function () {

    return function (input) {
      var result = [];

      angular.forEach(input, function (item) {
        result.push({name: item.name, type: item.type});
      });

      return result;
    };
  });
```

You can now use this in your service, shown as follows:

```
angular.module('brew-everywhere')
  .factory('fermentableDataService',
  function(messaging, events, mongolab, modelTransformer,
           Fermentable, translateFermentableToDisplay) {
  var fermentables = [];

  var forDisplay = function(){
    return translateFermentableToDisplay(fermentables);
  }
}
```

In the preceding example, the `forDisplay` method invokes the `translateFermentableToDisplay` filter, passing in the `fermentables` array, and returns the result that contains the new array of objects that only contains the name and type of each fermentable.

Using AngularJS filters to reduce and translate your application's data is another best practice you can leverage to encapsulate the business logic associated with your application's data. You will find that a well-planned filter will address the majority of your needs when it comes to reducing and translating your data. If you find that an AngularJS filter cannot meet your needs, I would suggest that you look at one of the two more advanced functional libraries such as underscore or lodash. These libraries provide over 80 tools to sort, search, and reduce arrays of objects using standard, functional programming paradigms that are powerful and will help to keep your code compact and easy to maintain.

Summary

In this chapter, we've discussed the subject of data management for your AngularJS application. We looked at a generic service that wraps the `MongoLab` REST API to retrieve and store data, and then discussed data services that provide a specialized interface to our database for each type of data the application uses. We then discussed how we can use the `$cacheFactory` service or third-party libraries such as `angular-local-storage` or BreezeJS to provide client-side caching of our data. Finally, we discussed how we can use AngularJS filters to reduce and translate our data and how our services can use filters to provide the same capabilities we get when we use filters in our HTML partials.

In the next chapter, we are going to talk about how we can integrate external services and JavaScript API libraries into our AngularJS applications to create mashups.

6
Mashing in External Services

As the Internet has grown, more and more innovative web-based applications have appeared that leverage the public APIs of cloud-based services. Nowadays, travel sites provide flight listings from airline booking systems, hotel listings from various hotel reservation systems, car rentals from rental companies, and tickets to popular tourist attractions from various entertainment venues all in a single, easy-to-use web-based application tailored to provide a totally new experience to the end user.

Being able to leverage cloud-based services in your application can present users with new advantages by providing functionality at a reduced cost both in terms of development time and hosting services. Cloud-based services and their APIs allow you to integrate functionality into your application quickly without having to build out the server side functionality to manage the application logic and data. In this chapter, we'll look at how we can incorporate calendar and task functionality into our sample application with a few services that handle all the interaction with the cloud-based APIs.

Storing events with Google Calendar

One of the major features of any serious brewing application is that of the brewing calendar. It allows the brewer to track the various batches currently fermenting and conditioning as well as plan for upcoming competitions. The brewing calendar can also serve as an inspiration to brewers by showing when to brew different styles of beer to keep up with the traditions of old.

Instead of building an entire calendar system, it's easier to leverage an existing cloud-based service such as Google's Calendar API; this way we can leverage an existing system that is incorporated in thousands of existing applications and mobile devices with little effort. Once we add an event to the user's Google Calendar, it appears everywhere making it easier for them to track their batches without always having to come back to our web application for an update.

We can also take advantage of various features such as notifications to alert the user when to perform an action on a batch of beer without having to build notification services and native applications; instead, we can configure events to use the existing Google Calendar features built into mobile devices and other web services to handle such notifications greatly reducing our development efforts.

In *Chapter 4*, *Handling Cross-cutting Concerns*, we discussed how to use external authentication services such as Google+ to provide authentication for our application, and we can now build on that functionality to allow our application to access the user's calendar by adding a single authorization scope to those we would normally use for authentication. The following code shows the new authorization scope we need to add to our authentication service so we can get access to the user's calendar:

```
.run(function (googlePlusAuthentication) {
  googlePlusAuthentication.configure({
    'clientId': '<YOUR_CLIENT_ID>',
    'scope': ['https://www.googleapis.com/auth/plus.login',
      'https://www.googleapis.com/auth/userinfo.email',
      'https://www.googleapis.com/auth/calendar']
  });
})
```

With this change, we now have read-write access to the user's Google Calendar, allowing us to add new brewing events to the user's existing calendar or to create a calendar specifically for our application.

Before we get into how to create calendars and events, we first need to spend some time on how to load the Google Calendar API so that our service can interact with the API. You can follow along with the full source for the service by checking out the `googleCalendar.js` file in the book's source project under the `service` folder.

The service we are going to build provides a simple generic interface that allows us to request a calendar by name, instead of providing a list of all of the calendars a user might have. We will also use this method to load the JavaScript for the Calendar API; once loaded, we will then pull a list of the user's calendars and then load the calendar with the given name. Finally, should the calendar not exist, our service provides the ability to create it. The service can be created with the following code:

```
var getCalendarByName = function(name, create){
  calendarName = name;
  createIfDoesNotExist = create;
  gapi.client.load('calendar', 'v3',
    handleCalendarClientLoaded);
};
```

```
messaging.subscribe(events.message._GET_CALENDAR_,
  getCalendarByName);

var handleCalendarClientLoaded = function() {
  messaging.publish(events.message._SERVER_REQUEST_STARTED_);
  var request = gapi.client.calendar.calendarList.list();
  request.execute(handleGetCalendarList);
};

var handleGetCalendarList = function(resp) {
  if(resp.items.length > 0){
    angular.forEach(resp.items, function(item){
      if(item.summary === calendarName){
        calendarId = item.id;
      }
    });
  }

  if (calendarId){
    getGoogleCalendarById(calendarId);
  }
  else {
    if(createIfDoesNotExist){
      createCalendar();
    }
    else {
      getCalendarFailed();
    }
  }
};

var getGoogleCalendarById = function(id){
  var request = gapi.client.calendar.events.list({'calendarId':
    id});
  request.execute(handleGetGoogleCalendarById);
};

var handleGetGoogleCalendarById = function(resp){
  if(resp.items.length > 0){
    messaging.publish(events.message._SERVER_REQUEST_ENDED_);
    messaging.publish(events.message._GET_CALENDAR_COMPLETE_,
      [resp]);
  }
  else {
```

```
      getCalendarFailed();
    }
  };

  var getCalendarFailed = function(){
    messaging.publish(events.message._SERVER_REQUEST_ENDED_);
    messaging.publish(events.message._GET_CALENDAR_FAILED_);
  };
```

To initialize the Google Calendar API library, we use the `gapi.client.load` method of the Google JavaScript client library to load the library asynchronously.

Once the library loads, the callback `handleCalendarClientLoaded` is invoked and it in turn calls the `calendar.calendarList.list` method to retrieve a list of the user's calendars.

This in turn invokes the callback handler, `handleGetCalendarList`, which iterates the list of calendars looking for the one with the name of the calendar requested and if it exists, we then call the method `getGoogleCalendarById`.

Once the calendar is returned, the callback handler, `handleGetGoogleCalendarById`, is invoked and if the response contains the desired calendar, the function returns it. If the request fails, the callback handler returns a failure notification.

If the calendar does not exist and the caller sent `true` for the `create` parameter, the service then calls the function `createCalendar`, which calls the `calendar.calendars.insert` method creating a new calendar for the user with the name passed into the `getCalendarByName` method that started the entire sequence:

```
  var createCalendar = function(){
    var request = gapi.client.calendar.calendars.insert(
      {"resource" :{'summary': calendarName}});
    request.execute(handleCreateCalendarRequest);
  };

  var handleCreateCalendarRequest = function(resp){
    if(resp && resp.summary === calendarName){
      calendarId = resp.id;
    }
    else {
      getCalendarFailed();
    }
  };
```

The rest of the service's methods use the basic CRUD pattern to create, update, and delete events for a given calendar. Each has an associated callback handler, which calls the method `getGoogleCalendarById` to reload the calendar after it has been updated. We use this pattern to take advantage of the messaging service by refreshing the calendar and then publishing a message that the calendar has been retrieved, in turn causing any external consumers that have subscribed to the `_GET_CALENDAR_COMPLETE_` message to get a new copy of the calendar. The following code shows how the pattern is implemented in the `updateCalendarEvent` method:

```
var updateCalendarEvent = function(calendarId, event){
  messaging.publish(events.message._SERVER_REQUEST_STARTED_);
  var request = gapi.client.calendar.events.update(
    {'calendarId': calendarId, 'eventId': event.id,
    'resource': event});
  request.execute(handleUpdateEventRequest);
};

messaging.subscribe(events.message._UPDATE_EVENT_,
  updateCalendarEvent);

var handleUpdateEventRequest = function(resp){
  if(resp){
    getGoogleCalendarById(calendarId);
  }
  else {
    messaging.publish(events.message._UPDATE_EVENT_FAILED_);
  }
};
```

Each of the methods accepts the unique ID of the calendar to operate on along with the event to create, update, or delete. The corresponding method then invokes the appropriate `calendar.event.x` method to insert, update, or delete the given event.

Event resources have an extensive amount of information that can be entered; however, we are only going to need a few pieces of information for our application. To create a new event, the minimum you need is a title for the event, the start date, and the end date. We are also going to provide a description that will be the same as the title and an entry to specify how reminders should be handled. The following code is an example of the data we'll be using to create a new event:

```
var addBrewDayToCalendar = function(currentRecipe,
  scheduledDate) {
  var formattedBrewDate = moment(scheduledDate)
    .format('YYYY-MM-DD');;
```

```
    var event = {'summary': 'Brew: ' + currentRecipe.Name,
      'description': 'Brew: ' + currentRecipe.Name,
      'start': {'date': formattedBrewDate},
      'end': {'date': formattedBrewDate},
      'reminders': {'useDefault': true}};

    messaging.publish(events.message._CREATE_EVENT_,
      [brewingCalendarId, event]);
  };
```

First, we have to format the date we'll use for start and end dates into a `yyyy-mm-dd` format. If we were going to provide a date and time, we'd need to ensure the date is formatted according to RFC 3339, which is the standard ISO 8601 date format. We also need to provide an object to the reminders parameter for the event; in our case, we will just specify that we want to use the defaults the user has configured; if we wanted to, we could provide overrides for the reminder defaults.

We've covered all of the functionality we'll be using in our application; however, there is more that you can do with the Google Calendar API. You can find out more about the Google Calendar API, the Event resource, and its parameters at the Google Calendar API web page at `https://developers.google.com/google-apps/calendar/`.

Using Google Tasks to build a brewing task list

Once we've scheduled our brew day, it's always good to have a brew day task list that we can use to make sure our brewing session is a successful one. To do that, we can take advantage of the Google Tasks API to create a task list that can guide us through our brew day. The Google Tasks API is very similar to the Google Calendar API, which makes our service very similar to the one we created to interact with the Google Calendar API.

As with the Google Calendar API, we need to add another authorization scope to our Google authentication configuration so that we can gain access to the user's tasks. The following code shows the new authorization scope we need to add to our authentication service so that we can get access to the user's tasks:

```
.run(function (googlePlusAuthentication) {
  googlePlusAuthentication.configure({
    'clientId': '<YOUR_CLIENT_ID>',
    'scope': ['https://www.googleapis.com/auth/plus.login',
      'https://www.googleapis.com/auth/userinfo.email',
```

```
        'https://www.googleapis.com/auth/calendar',
        'https://www.googleapis.com/auth/tasks']
    });
  })
```

With this change, we now have read-write access to the user's Google Tasks, allowing us to add new brewing tasks to the user's existing task list or one we create specifically for our application.

Our service will start out with a method that will retrieve a task list by a name that will also ensure the Google Tasks API JavaScript library is loaded, and once loaded, it will invoke another method on the API to retrieve a list of task lists for the user and then finally either retrieve the task list or create a new task list if requested.

```
var getTaskListByName = function(name, create){
  taskListName = name;
  createIfDoesNotExist = create;
  gapi.client.load('tasks', 'v1', handleTasksClientLoaded);
};

messaging.subscribe(events.message._GET_TASK_LIST_,
  getTaskListByName);

var handleTasksClientLoaded = function() {
  messaging.publish(events.message._SERVER_REQUEST_STARTED_);
  var request = gapi.client.tasks.tasklists.list();
  request.execute(handleGetTasksList);
};

var handleGetTasksList = function(resp) {
  if(resp.items.length > 0){
    angular.forEach(resp.items, function(item){
      if(item.title === taskListName){
        taskListId = item.id;
      }
    });
  }

  if (taskListId){
    getGoogleTaskListById(taskListId);
  }
  else {
    if(createIfDoesNotExist){
      createTaskList();
    }
```

```
      else {
        getTaskListFailed();
      }
    }
  };

  var getTaskListFailed = function(){
    messaging.publish(events.message._SERVER_REQUEST_ENDED_);
    messaging.publish(events.message._GET_TASK_LIST_FAILED_);
  };

  var getGoogleTaskListById = function(id){
    var request = gapi.client.tasks.tasks.list({'tasklist': id});
    request.execute(handleGetGoogleTaskListById);
  };

  var handleGetGoogleTaskListById = function(resp){
    if(resp.items.length > 0){
      messaging.publish(events.message._SERVER_REQUEST_ENDED_);
      messaging.publish(events.message._GET_TASK_LIST_COMPLETE_,
        [resp]);
    }
    else {
      getTaskListFailed();
    }
  };

  var createTaskList = function(){
    var request = gapi.client.tasks.tasklists.insert({"resource"
      :{'title': taskListName}});
    request.execute(handleCreateTaskListRequest);
  };

  var handleCreateTaskListRequest = function(resp){
    if(resp && resp.title === taskListName){
      taskListId = resp.id;
      getGoogleTaskListById(taskListId);
    }
    else {
      getTaskListFailed();
    }
  };
```

Again, we start out with the `gapi.client.load` method requesting the tasks API and pass in the callback handler, `handleTasksClientLoaded`, which will be called once the JavaScript library is loaded.

Once the library is loaded, we make a call to the `tasks.tasklist.list` method to retrieve a list of the user's task lists and pass in the callback handler. The callback handler, `handleGetTaskList`, will then iterate through the list looking for a matching title and then request the list by id calling `getGoogleTaskListById`.

The method `getGoogleTaskListById` makes an API call to `tasks.tasks.list`, passing in the `id` of the task list to return and passes in the callback handler, `handleGetGoogleTaskListById`, which in turn publishes the message `_GET_TASK_LIST_COMPLETE_` with the returned task list or calls the method `getTaskListFailed` that publishes `_GET_TASK_LIST_FAILED_`.

If the task list does not exist and the caller has passed in `true` for the `create` parameter, the service will call the method `createTaskList`, which will call the `tasks.tasklist.insert` method to create the task list with the name passed into the call `getTaskListByName`.

The rest of the service provides a similar set of CRUD methods to allow the creation, update, and deletion of tasks associated with a given task list. Each one requires the unique id of the task list and an event that will be acted upon. The following is the code for the `deleteTask` method:

```
var deleteTask = function(taskListId, task){
  messaging.publish(events.message._SERVER_REQUEST_STARTED_);
  var request = gapi.client.tasks.tasks.delete(
    {'tasklist': taskListId, 'task': task.id});
  request.execute(handleDeleteTaskRequest);
};

messaging.subscribe(events.message._DELETE_TASK_, deleteTask);

var handleDeleteTaskRequest = function(resp){
  if(resp){
    getGoogleTaskListById(taskListId);
  }
  else {
    messaging.publish(events.message._DELETE_TASK_FAILED_);
  }
};
```

Again based on the method called, a corresponding call is made to the `tasks.tasks.x` method to insert, update, and delete a task. A callback method is provided to handle the response from the API call and if successful, a call is made to `getGoogleTaskListById`, which will in turn retrieve the updated list and publish the `_GET_TASK_LIST_COMPLETE_` message causing all subscribers to update their copy of the task list with the one sent with the notification.

Tasks have several properties that we will use when we create our brew day tasks. The following is an example of creating a new task for our brew day task list:

```
var addPrepareStarterTask = function(currentRecipe,
  scheduledDate){
  var dueDate = moment(scheduledDate).subtract('days', 1);

  var task = {'title': 'Prepare Starter: ' + currentRecipe.Name,
    'status': 'needsAction',
    'due': dueDate.format('YYYY-MM-DD')
  };

  messaging.publish(events.message._CREATE_TASK_,
    [brewingTaskListId, task]);
};
```

This method starts off by calculating the due date for the task by subtracting a day from the schedule date, since the task needs to occur prior to our brew day. We then create our task object providing a title, a status, and the due date formatted in the ISO 8601 Date Format using the `moment.js` library. Finally, the method publishes the `_CREATE_TASK_` event, passing in the unique ID of the Brewing task list and the new task to insert into the task list.

We've covered all of the functionality we'll be using in our application; however, there is more that you can do with the Google Tasks API. You can find out more about the Google Tasks API, the Task resource, and its parameters on the Google Tasks API web page at `https://developers.google.com/google-apps/tasks/`.

Tying the Google Calendar and task list together

We now have services that can be used to create entries into a user's Google Calendar or task list; however, having that code all over our application adds unnecessary code and bloat to the components that need to interact with the services. It would be better to use a facade pattern to encapsulate the business logic around managing our brewing calendar and task list. This way, the components that want to interact with

the user's calendar or task list can do it via this new service in a more refined way.

The `brewCalendar` service wraps the business logic needed to manage both the brewing calendar and brewing task list. It also handles the logic to create all of the calendar events and tasks for a brew day:

```
angular.module('brew-everywhere').factory('brewCalendar', function
(messaging, events) {

  var brewingCalendar = {};
  var brewingCalendarId = '';
  var brewingCalendarName = 'My Brewing Calendar';
  var brewingTaskList = {};
  var brewingTaskListId = '';
  var brewingTaksListName = 'My Brewing Tasks';
  var reduceBrewingCalendar = function(calendar){
  var formattedDate = moment().format('YYYY-MM-DD');
  // filter calendar item for those still in the future
  var result = _.filter(calendar.items, function(event){
    if(event.end.date > formattedDate) {
      return event;
    }
  });
  // sort by start date
  result = _.sortBy(result, function(event){
    return event.start.date;});
  return result;
};

var getBrewingCalendar = function(){
  messaging.publish(events.message._GET_CALENDAR_,
    [brewingCalendarName, true]);
};

messaging.subscribe(events.message._GET_BREWING_CALENDAR_,
  getBrewingCalender);

var onHandleGetCalendarComplete = function(calendar){
  brewingCalendar = calendar;
  brewingCalendarId = calendar.id;

  if(calendar.items.length > 0){
    var result = reduceBrewingCalendar(brewingCalendar);
    messaging.publish
      (events.message._GET_BREWING_CALENDAR_COMPLETE_,
      [brewingCalendar]);
```

```
      getBrewingTaskList();
    }
    else {
      createDefaultCalendarEntries(brewingCalendarId);
    }
  };

  messaging.subscribe(events.message._GET_CALENDAR_COMPLETE_,
     onHandleGetCalendarComplete);

  var getBrewingTaskList = function(){
    messaging.publish(events.message._GET_TASK_LIST_,
      [brewingTaksListName, true])
  };

  messaging.subscribe(events.message._GET_BREWING_TASKS_,
    getBrewingCalender);

  var onHandleGetTaskListComplete = function(taskList){
    brewingTaskList = taskList;
    brewingTaskListId = taskList.id;

  messaging.publish(events.message._GET_BREWING_TASKS_COMPLETE_,
    [brewingTaskList]);
  };

  messaging.subscribe(events.message._GET_TASK_LIST_COMPLETE_,
    onHandleGetTaskListComplete);

  var createDefaultCalendarEntries = function(id){
    var event = defaultEvents.shift();
    if(event){
      messaging.publish(events.message._CREATE_EVENT_,
        [id, event]);
    }
    else {
      getBrewingCalender();
    }
  };

  var handleCreateDefaultEventRequest = function(){
    createDefaultCalendarEntries(brewingCalendarId);
  };

  messaging.subscribe(events.message._CREATE_EVENT_COMPLETE_,
    handleCreateDefaultEventRequest);
```

You can view the full source code in the sample files for the book.

Whenever a consumer publishes the `_GET_BREWING_CALENDAR_` event, the service makes a call to the `googleCalendar` service to get the calendar with the name `"My Brewing Calendar"` and `true` for the `create` parameter. When the `googleCalendar` service returns the calendar, the `onHandleGetCalendarComplete` callback handler is called and it then checks to see if the calendar has any events; if so, it uses the `reduceBrewingCalendar` method to reduce the returned events to only those that have an end date in the future and then sorts them by start date. Once the `reduceBrewingCalendar` method returns the events, the callback handler stores off the calendar, publishes the `_GET_BREWING_CALENDAR_COMPLETE_` event, and then calls the `getBrewingTaskList` method.

Should the brewing calendar have no events, we know that the calendar is newly created so the service calls the `createDefaultCalendarEvents` method, which goes through the array of `defaultEvents` one by one and adds them to the brewing calendar. This populates the brewing calendar with a series of events showing what beer styles to brew and when, helping to provide some guidance as to when to brew the different beer styles.

The `getBrewingTaskList` method makes a call to the `googleTasks` service to get the task list with the name `"My Brewing Tasks"` that has the value `true` for the `create` parameter. When the `googleTasks` service returns the task list, the `onHandleGetTaskListComplete` callback handler is called; it then stores off the task list and then publishes the `_GET_BREWING_TASKS_COMPLETE_` event. Since there are no default brewing tasks, the service does not try to create any default brewing tasks.

The other large piece of business logic the `brewCalendar` service contains is the scheduling of brew days and their related tasks. Whenever a home brewer prepares for a brew day, there are certain tasks and dates that are relevant to the batch of beer brewed. The brewer has to make sure they have all of the ingredients to brew the beer and that they've prepared the yeast prior to the brew day. They also need to know how long the beer needs to ferment, when to bottle it, and how long it should condition prior to enjoying it.

Creating these events and tasks can be quite extensive; the start and end dates need to be calculated for each event and the ingredient list needs to be compiled. The service breaks each of these pieces of business logic into one of several methods and a single method calls each of these sub-methods to make sure all of the events and tasks are created. The `createBrewDayEvents` method handles all of this. The following is the code for `createBrewDayEvents` along with a method that creates a task and one that creates a calendar event:

```
var createBrewDayEvents = function(recipe, scheduleDate,
  addBrewDayTasks, addFermentationTasks,
  addConditioningTasks){
  if(addBrewDayTasks){
    addGetIngredientsTask(recipe, scheduleDate);
    addPrepareStarterTask(recipe, scheduleDate);
  }

  addBrewDayToCalendar(recipe, scheduleDate);

  if(addFermentationTasks){
    addPrimaryToCalendar(recipe, scheduleDate);
    addSecondaryToCalendar(recipe, scheduleDate);
    addTertiaryToCalendar(recipe, scheduleDate);
  }

  if(addConditioningTasks){
    addBottlingToCalendar(recipe, scheduleDate);
    addConditioningToCalendar(recipe, scheduleDate);
  }
};

messaging.subscribe(events.message._CREATE_BREW_DAY_EVENTS_,
  createBrewDayEvents);
```

The `createBrewDayEvents` methods takes a recipe, the date of the brew day, and several flags indicating what calendar events and tasks to add to the user's calendar and task list. It then calls the various methods based on the flags. The `createBrewDayEvents` method will always add the brew day to the user's calendar.

Another method that is rather involved is the `addIngredientsTask` method. It iterates through the various ingredients in the recipe and builds a list that will be added to the task so that the user has a shopping list for everything they need to brew the given recipe. The code for the `addIngredientsTask` method is as follows:

```
var addGetIngredientsTask = function(currentRecipe,
  scheduledDate){
```

```
    var dueDate = moment(scheduledDate).subtract('days', 2);
    var ingredientList = 'Gather the following ingredients:\n\n';

    angular.forEach(currentRecipe.Fermentables, function(item){
      ingredientList += item.Amount + ' - ' + item.Name + ' ' +
      item.Type + '\n';
    });

    angular.forEach(currentRecipe.Adjuncts, function(item){
      ingredientList += item.Amount + ' - ' + item.Name + ' ' +
        item.Type + '\n';
    });

    angular.forEach(currentRecipe.Hops, function(item){
      ingredientList += item.Amount + ' - ' + item.Name + ' ' +
        item.Form + '\n';
    });

    angular.forEach(currentRecipe.Yeast, function(item){
      ingredientList += item.Amount + ' - ' + item.Name + ' ' +
        item.Form + '\n';
    });

    var task = {'title': 'Gather Ingredients: ' +
      currentRecipe.Name,
      'status': 'needsAction',
      'due': dueDate.format('YYYY-MM-DD'),
      'notes': ingredientList
    };

    messaging.publish(events.message._CREATE_TASK_,
      [brewingTaskListId, task]);
  };
```

Once the `addGetIngredientsTask` method finishes compiling the shopping list, it creates a new task and assigns the shopping list to the `notes` section of the task. The method also calculates a due date by subtracting two days from the scheduled brew date to ensure that the brewer is alerted early enough to have time to gather all the required ingredients. Finally, the method publishes a `_CREATE_TASK_` event, passing in the `id` of the user's brewing task list and the newly created task.

The `addPrimaryToCalendar` method, given in the following code, shows some of the date calculations that go into adding a calendar event that spans a given period of time. First, the method checks the recipe to make sure that the brewer has added a primary fermentation step. Then, it starts calculating the end date by adding the number of days of primary fermentation to `scheduleDate` stored in `startDate`. The method then needs to format the `start` and `end` dates into the `yyyy-mm-dd` format so that the new calendar event displays properly:

```
var addPrimaryToCalendar = function (currentRecipe,
  scheduledDate) {
  if (currentRecipe.PrimaryAge > 0 &&
      currentRecipe.PrimaryTemp > 0) {
    var startDate = moment(scheduledDate);
    var endDate = startDate.add('days',
      parseInt(currentRecipe.PrimaryAge));
    var formattedStartDate = startDate.format('YYYY-MM-DD');
    var formattedEndDate = endDate.format('YYYY-MM-DD');

    var event = {'summary': 'Primary Fermentation: ' +
      currentRecipe.Name,
      'start': {'date': formattedStartDate},
      "end": {"date": formattedEndDate},
      'description': 'Primary Fermentation at ' +
      currentRecipe.PrimaryTemp + ' degrees.',
      'reminders': {'useDefault': true}};

      messaging.publish(events.message._CREATE_EVENT_,
      [brewingCalendarId, event]);
  }
};
```

Once the dates have been formatted, a new calendar event is created and sent to the `googleCalendar` service using the `_CREATE_EVENT_` event. This time it passes in the unique id of the user's brewing calendar and the newly created event.

One of the important things about the `brewCalendar` service is that it allows us to hide away the implementation details of interacting with Google's Calendar and Tasks API. You can see how the other methods are implemented by checking out the `brewCalendar.js` file in the `service` directory of the source code for this book.

Summary

Having the ability to integrate cloud-based services into our application allows us to add additional functionality to our AngularJS applications without having to code all of the core logic required to implement such functionality.

In this chapter, we covered how to integrate Google's Calendar and Tasks APIs into our application by creating two generic services that allows us to interact with any calendar or task list belonging to the user.

We then discussed how we could encapsulate our application's business logic around creating and managing a brewing calendar and brew day task list by creating another service specific to the task.

In the next chapter, we'll discuss how to further use these services to implement our application's business logic.

7
Implementing the Business Logic

Over the past several years, the knowledge base around JavaScript has matured tremendously. Web applications appeared on the scene with more and more functionality as JavaScript libraries helped to accelerate the move from server-based applications to large client-based applications.

This trend of moving functionality from the server to client was due to many things, but mostly because of the advancements in technology that provided the same speed and processing power as our desktops to mobile devices. This new power beckoned developers, urging them to move more and more away from the server to the client. We have seen JavaScript at the core of this movement. Developers have created cross compilers and emulators that allow game engines and virtual machines to run in the browser, providing the same, if not better performance, than their original platforms.

This same shift has seen the appearance of unhosted web applications, applications that no longer rely on servers to process logic, but leverage cloud-based services to perform authentication and data access, and implement business logic as part of the web application. These unhosted web applications can run on any platform, allowing you to choose where your data lives and help keep your data private and away from app providers. They are also cheaper to host, easier to mirror, and provide near infinite scalability.

This chapter looks at the various ways we can implement our business logic in the services we create. It also explains some of the services used in our application to implement complex rule-based calculations and handle user process flow based on a finite state machine.

Encapsulating business logic in models

In *Chapter 2*, *Designing Services*, and *Chapter 5*, *Data Management*, we looked at how to create model services in our application using the `value` provider and a standard JavaScript constructor definition. This allowed us to inject the model into our controllers, directives, and services the same way we would in any other AngularJS component. We also discussed how to use prototypal inheritance to define the business logic code associated with the model. We also covered a generic translation service to allow us to transform the plain JSON objects received from external data services into our model objects.

This concept of keeping the business logic that operates on an object with the object is a standard object-oriented programming principle that many object-oriented languages provide as part of the class definition. This is accomplished in JavaScript by redefining the object's prototype as follows:

```
var Fermentable = function() {
  var self = this;
  self.Name = '';
  self.Type = ''; // can be "Grain", "Sugar", "Extract",
                  // "Dry Extract" or "Adjunct"
  self.Amount = 0.0;
  self.Yeild = 0.0; // percent
  self.Color = 0.0; // Lovibond units (SRM)
  self.AddAfterBoil = false;
  self.Origin = '';
  self.Supplier = '';
  self.Notes = '';
  self.CoarseFineDiff = 0.0; // percent
  self.Moisture = 0.0; // percent
  self.DiastaticPower = 0.0; // in lintner units
  self.Protein = 0.0; // percent
  self.MaxInBatch = 0.0; // percent
  self.RecommendMash = false;
  self.IBUGalPerPound = 0.0;
  self.DisplayAmount = '';
  self.Potential = 0.0;
  self.DisplayColor = 0.0;
  self.ExtractSubstitute = '';
};

Fermentable.prototype = {
  /**
   * calculates the gravity per pound for the fermentable
```

```
 * @param brewHouseEfficiency - the estimated brew house
 * efficiency
 * @returns {number} - potential extract per pound for the
 * fermentable
 */
gravityPerPound: function(batchType, brewHouseEfficiency){
  var result = ((this.Potential - 1) * 1000.0);

  switch(batchType)
  {
    case "All Grain":
      result = result * brewHouseEfficiency;
      break;
    case "Partial Mash":
      result = result * 0.33;
      break;
    case "Extract":
      break;
  }

  return result;
},
/**
 * calculates the gravity for the ingredient
 * @param brewHouseEfficiency - the estimated brew house
 * efficiency
 * @returns {number} - returns the total potential extract for
 * the fermentable
 */
ingredientGravity: function(batchType, brewHouseEfficiency){
  return this.Amount * (this.gravityPerPound(batchType,
           brewHouseEfficiency) / 1000);
},

/**
 * calculates the gravity points for a given recipe
 * @param batchVolume - total pre boil volume
 * @param brewHouseEfficiency - the estimated brew house
 * efficiency
 * @returns {number} - returns the total gravity points for the
 * fermentable
 */
gravityPoints: function(batchType, batchVolume,
  brewHouseEfficiency){
```

```
      return (this.ingredientGravity(batchType, brewHouseEfficiency)
               / batchVolume);
    },

    /**
     * calculates the color the fermentable will contribute to a
     * batch of beer based on the amount of the feremntable
     * fermentable used in the recipe
     * @returns {number} - the srm color value for the fermentable
     */
    srm: function(){
      return parseFloat(this.Amount) * parseFloat(this.Color);
    },

    /**
     * calculates the color the fermentable will contribute to a
     * batch of beer based on the pre-boil volume of the recipe
     * @param batchVolume - total pre-boil volume
     * @returns {number} - the srm color value for the fermentable
     */
    colorPoints: function(batchVolume){
      return this.srm() / batchVolume;
    }
  };
```

This definition allows us to create a new instance of the object and invoke the object methods as follows:

```
var grain = new Fermentable();
grain.Amount = 5;
grain.Color = 3;
grain.Potential = 1.039;

var gravityPoints = grain.gravityPoints('All Grain', 10, 0.65);
```

The advantage of using prototypal inheritance is that it allows us to define the business logic as part of the model object. It also allows logic to be easily invoked by calling the function associated with the object's prototype. This makes the code much easier to read and reduces the need to create additional functions that operate on the object. The main disadvantage is that the business logic for an application can end up being spread across your entire application. You might have some business logic in the models and other business logic in services or libraries; this can make it tough to maintain as the application gets larger.

Encapsulating business logic in services

Another common way to incorporate your business logic is to create a service that exposes various functions which encapsulate the business logic and take arrays of objects as parameters to operate as the business logic requires. We could just as easily take the same code and wrap it into a service that provides the same functionality as follows:

```
angular.module('brew-everywhere')
.factory('fermentableHelper',function() {

  /**
   * calculates the gravity per pound for the fermentable
   * @param fermentable – the fermentable to calculate the
   * value for
   * @param brewHouseEfficiency - the estimated brew house
   * efficiency
   * @returns {number} - potential extract per pound for the
   * fermentable
   */
  var gravityPerPound = function(fermentable, batchType,
      brewHouseEfficiency){
    var result = ((fermentable.Potential - 1) * 1000.0);

    switch(batchType)
    {
      case "All Grain":
        result = result * brewHouseEfficiency;
        break;
      case "Partial Mash":
        result = result * 0.33;
        break;
      case "Extract":
        break;
    }

    return result;
  },
  /**
   * calculates the gravity for the ingredient
   * @param fermentable – the fermentable to calculate the
   * value for
   * @param brewHouseEfficiency - the estimated brew house
   * efficiency
```

```
 * @returns {number} - returns the total potential extract for
 * the fermentable
 */
var ingredientGravity = function(fermentable, batchType,
    brewHouseEfficiency){
  return this.Amount * (gravityPerPound(fermentable, batchType,
           brewHouseEfficiency) / 1000);
},

/**
 * calculates the gravity points for a given recipe
 * @param fermentable – the fermentable to calculate the
 * value for
 * @param batchVolume - total pre boil volume
 * @param brewHouseEfficiency - the estimated brew house
 * efficiency
 * @returns {number} - returns the total gravity points for the
 * fermentable
 */
var gravityPoints = function(fermentable, batchType,
  batchVolume, brewHouseEfficiency){
  return (ingredientGravity(fermentable, batchType,
     brewHouseEfficiency) / batchVolume);
},

/**
 * calculates the color the fermentable will contribute to a
 * batch of beer based on the amount of the fermentable used in
 * the recipe
 * @param fermentable – the fermentable to calculate the
 * value for
 * @returns {number} - the srm color value for the fermentable
 */
var srm = function(fermentable){
  return parseFloat(fermentable.Amount) *
     parseFloat(fermentable.Color);
},

/**
 * calculates the color the fermentable will contribute to a
 * batch of beer based on the pre-boil volume of the recipe
 * @param fermentable – the fermentable to calculate the
 * value for
 * @param batchVolume - total pre-boil volume
```

```
     * @returns {number} - the srm color value for the fermentable
     */
    var colorPoints = function(fermentable, batchVolume){
      return this.srm(fermentable) / batchVolume;
    }

     var helper = {
      gravityPerPound: gravityPerPound,
      ingredientGravity: ingredientGravity,
      gravityPoints: gravityPoints,
      srm: srm,
      colorPoints: colorPoints
    };

    return helper;
  });
```

To use the service, we need to inject it into our controller and then invoke it as follows:

```
var grain = new Fermentable();
grain.Amount = 5;
grain.Color = 3;
grain.Potential = 1.039;

var gravityPoints = fermentableHelper.gravityPoints(grain,
                        'All Grain', 10, 0.65);
```

The advantage of having your business logic code in a service is that it is all in one place and not spread across your application. It also makes it easier when it comes to testing, as you do not need a separate unit test for each model object.

The disadvantage of having your business logic code in a service is that it requires additional code to handle iterating over arrays of model objects.

Models or services, which one to use?

There is no right or wrong way to implement your business logic in your AngularJS application. It really comes down to what works best for you and your team.

If you are more familiar with the object-oriented programming paradigm, defining the business logic as part of your model will feel more comfortable.

If you are more used to using libraries, defining the business logic as a service will better suit you. I found that a mix of both types works best in large applications.

I usually add small calculations that represent calculated values to the model object. This helps reduce the number of services I need to inject into a controller and is handy when I need to display the calculated values in a view.

I create services to encapsulate large amounts of complex business logic in my application. This allows me to keep the business logic in a few places, helping to keep the code maintainable and easy to test. I also find myself reusing these services across other applications, and only having to copy a few files from project to project is a lot easier than bringing an entire application's source.

Now that we've talked about the various strategies, we can use one to implement our business logic in our application. Let's look at a couple of services that we can use to implement complex business logic in our application.

Controlling a view flow with a state machine

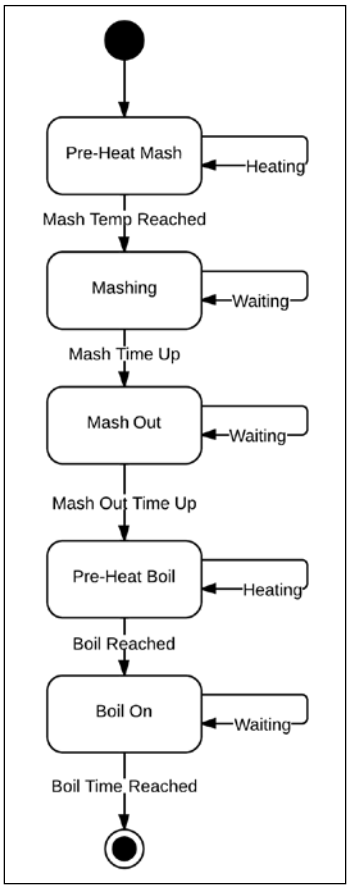
State machines represent a mathematical model of computational behavior. The model can be in any one of a number of finite states at any given time. The model changes from state to state when initiated by a triggering event or condition.

State machines are very good at handling complex user interface flow through an application. Each state can represent a view shown to the user. User interactions serve as the triggering events that can be used to transition from one state to another.

We can use a state machine to control the user's flow through our program, instead of coding the flow in our controllers. We can also use state machines to manage long-running processes between sessions.

The state machine that we're going to cover is so generic that we can use it to manage routing or long-running processes. It consists of the state machine engine and a view controller that is used to route between the different views of our application. The view controller also has the ability to invoke functions on a process controller object based on the transitions between states and whenever an event triggers a transition.

Our sample application will use the state machine to manage the brewing of a batch of beer during the brew day. It will provide the user with a brew timer that changes as the brewer progresses through the brewing of a batch of beer. The following is the state diagram for our brewing process:



As the brewer brews a batch of beer, there are several phases. The process starts with the preheating of water, the mashing in of grains, and the steeping of the grains to convert the starch into sugar. It ends with the boiling of the sugar and water mixture, which is called wort, to reach the desired ratio of sugar in the wort, prior to fermenting it to create alcohol.

Our application will help the brewer manage this process by displaying a different view to the brewer, and then, it will transition to a different view either when a predetermined time has been reached or when the brewer indicates that a stage is complete.

The state machine we'll be using requires that we provide an object that represents the various states, the transitions between each state, and a list of commands or trigger events that can be used to transition from state to state.

The following is the definition for part of our process; you can review the complete version of the file in the source of this book, in the `model` directory in the file called `brewingProcessStateMachine.js`:

```
{
  id: 'BrewProcess',
  processController: 'BrewProcessController',
  title: 'Brew Process',
  description: 'UI Process flow for Brewing a Batch of Beer',
  transitions: [
    {id: 'Begin', targetState: 'PreHeatMash'},
    {id: 'PreHeatMashHeating', targetState: 'PreHeatMash'},
    {id: 'MashTempReached', targetState: 'Mash'},
    {id: 'MashTimeWaiting', targetState: 'Mash'},
    {id: 'MashTimeReached', targetState: 'MashOut'},
    {id: 'MashOutTimeWaiting', targetState: 'MashOut'},
    {id: 'MashOutTimeReached', targetState: 'PreHeatBoil'},
    {id: 'PreHeatBoilHeating', targetState: 'PreHeatBoil'},
    {id: 'BoilReached', targetState: 'Boil'},
    {id: 'BoilTimeWaiting', targetState: 'Boil'},
    {id: 'BoilTimeReached', targetState: 'Exit'},
    {id: 'exit', targetState: 'Exit'}
  ],
  states: [
    { id: 'Exit',
      activityId:'BrewProcess',
      url: '/',
      title: 'Exit',
      description: 'Return to Home Screen',
      commands: []
    },
    { id: 'Error',
      activityId:'BrewProcess',
      url: '/Error',
      title: 'Error',
      description: 'An error has occurred while performing the
        last action.',
      commands: [
        { id: 'Previous', transition: 'exit'},
        { id: 'Next', transition: 'exit'},
        { id: 'Cancel', transition: 'exit'}
      ]},
    { id: 'PreHeatMash',
      activityId:'BrewProcess',
      url: '/PreHeatMash',
      title: 'Pre-Heat Mash',
      description: 'Pre-heat the mash water to reach the desired
```

```
          mash temperature',
        commands: [
          { id: 'Waiting', transition: 'PreHeatMashHeating'},
          { id: 'TemperatureReached', transition:
            'MashTempReached'},
          { id: 'Exit', transition: 'exit'},
          { id: 'DisplayErrorMessage', transition:
            'displayErrorMessage'}
        ]},
…
source removed for brevity
…
      { id: 'Boil',
        activityId:'BrewProcess',
        url: '/Boil',
        title: 'Boil',
        description: 'Boil the wort for the desired time',
        commands: [
          { id: 'Waiting', transition: 'BoilTimeWaiting'},
          { id: 'TimeReached', transition: 'BoilTimeReached'},
          { id: 'Exit', transition: 'exit'},
          { id: 'DisplayErrorMessage', transition:
            'displayErrorMessage'}
        ]}
    ]
}
```

The state definition includes an array of transitions that indicates which state should be set as the current state and an array of states each with an array of commands that can be used to trigger the transition from state to state.

The state definition also defines a process controller that can be used by the view controller to execute code when it transitions to a new state, when it transitions from the current state, and when any of the commands in a state are triggered. The process controller needs to follow a standard naming method for each of the methods. If you want to provide a function that will be executed when we transition to a new state, the function must be named `pre<state.id>`. To provide a function that will be executed when we transition from a state, the function must be named `post<state.id>`. You need to replace `state.id` with the ID of the state you want to provide the functions for. For example, in our previous definition, if we wanted to provide pretransition and posttransition functions for the `Mash` state, our process controller would need to have functions named `preMash` and `postMash`.

The naming convention for the functions that will be executed when a command is triggered is similar. The command needs to be named `<state.id><command.id>`, where `state.id` is the ID of the state and `command.id` is the ID of the command. So, if we wanted to provide a function for the `Waiting` command in the `Boil` state, we would need to add a function named `BoilWaiting` to our process controller.

The state machine service is meant to be used either standalone or as part of another service such as the view controller mentioned earlier. Its interface is very basic, and allows the consumer to add state definitions to the state machine and retrieve a state for a given activity, transition, or command. To simplify the state machine, it leverages the underscore library's `_.findWhere` method to retrieve a state based on the various search methods.

The following is one of the methods from the state machine that returns the target state for the transition associated with a given command on a state object:

```
/**
 * @description
 *
 * Returns the target state for the transition associated with
 * the command of the given state object
 *
 * @param state {object} the state object instance to use
 * @param commandId {string} the command to follow
 * @returns {*} {object} the target state for the transition
 * associated with the given command or null if the state
 * is not found
 */
getTargetStateForCommand: function (state, commandId) {
  var outState = null;
  var command = _.findWhere(state.commands, { id:
                     commandId});
  if (command) {
    outState = stateMachine.getTargetStateForActivity(
      state.activityId, command.transition);
  }

  return outState;
}
```

This preceding method first finds the command object in the state object's command array. It then calls another method in the state machine service to return the state associated with the transition that is associated with the command.

As I mentioned earlier, the state machine is designed to be used in conjunction with other services such as the view controller service; it doesn't do much more than maintain the current state and find the state associated with a transition or command. To navigate to the different pages in our application, we need to switch to the URL defined in each state. This means that the view controller needs to do the work to navigate to the URL once we transition to a new state. As the view controller was designed to work with AngularJS' `ngRoute` library, we also need to ensure that each view we are going to navigate to has been set up in our application using the `routeProvider` component. The following is some of the code in the view controller service that is used to navigate to another view based on a command:

```
/**
 * @description
 *
 * Invokes the navigation process based on the given command
 * identifier, which will result in a redirection to
 * a new view. If the resulting state is not set, that means
 * we redirect to the home view.
 *
 * @param commandId {string} identifier of the command of
 * the current state to use invoke the state navigation
 */
navigateToStateUsingCommand: function (commandId) {
  // if we have a current state execute the post processing
  // for the state
  if (controller.currentState) {
    controller.executePostProcessMethodForActivity(
      controller.currentState.id,
      controller.currentState.activityId);
  }
  // We are navigating within an activity, which means we
  // have a current state, and a command to execute within
  // that state anything else is incorrect usage
  var command = _.findWhere(
     controller.currentState.commands, { id: commandId });
  if (command) {
    // execute the process method for the command
    controller.executeCommandProcessMethod(commandId,
      controller.currentState.id,
      controller.currentState.activityId);
    // navigate to the new state based on the command
    controller.currentState = stateMachine
        .getTargetStateForCommand(controller.currentState,
           commandId);
```

```
        }
        // check to make sure we have a currentState
        // and return success if we do
        if (controller.currentState) {
          // execute and pre processing for the new state
          controller
            .executePreProcessMethodForActivity(
               controller.currentState.id,
               controller.currentState.activityId);
          // change the view based on the new state
          $location.path(controller.currentState.Url);
        }
      },
```

The method first checks to see whether we have a current state loaded in the controller, and if so, it executes the post-transition function associated with the state. Once this function returns, it retrieves the command from the current state and then invokes the function associated with the command on the process controller. Then, it calls the `getTargetStateForCommand` function of the state machine to get the new current state. If the current state returned is a valid object, the pretransition function on the process controller is executed. Finally, the function uses the `$location` service to change the path to the `URL` associated with the new current state.

To see more code associated with the state machine and view controller, check out the `statemachine.js` source file in the `services` folder of the sample project.

We discussed how we can use a state machine to encode the business rules in our application that covers how the user should navigate through our application. Next, let's look at how we can use a rules engine to handle complex validations within our application.

Validating complex data with a rules engine

Business rules within an application can sometimes become very complex, so complex that writing them in code can become a maintenance nightmare. This is where rules engines can help. Rules engines are computational services that can use dynamic rules to evaluate data and act on it. However, instead of using fixed code to implement the rules, rules engines can use different sets of rules to evaluate data in different ways as the conditions of the application dictate.

The business rules engine we are going to look at uses a forward-chaining algorithm to walk through a set of rules and execute them by evaluating the facts passed into the execute method of the service.

For those not familiar with rules engines, facts are the data we need to pass into the rules engine; the rules expect to evaluate these facts to determine if a rule is met. Forward-chaining means that the rules engine walks the rules in a sequence, evaluating the facts and modifying them based on the rules, until the last rule is reached. If there is no solution, the rules engine starts back at the beginning of the rule sequence with the modified facts and continues to execute through the rules until a solution is found or the maximum number of iterations are met.

Rules consist of conditions and consequences that are executed if the conditions evaluate to true. Rules also contain a name and description for documentation purposes as well as a priority and indicator that suggest whether a rule should be executed or not. This allows you to change the priority of a rule or turn a rule on or off based on the conditions in different rules, allowing rules execution to be as dynamic as possible during execution.

The facts passed into the rules engine's execute method should be an object and can have as many properties as needed. The rules engine also expects a callback function to be provided that takes an object as a parameter, which will be the result of the rules execution when it is completed.

For our application, I wanted to use the rules engine to evaluate our recipe as we formulate it to return a list of beer styles that match our recipe. As this is mainly a matching exercise, it'll be perfect for a rules engine and will require much less code than if we were to code it.

The following are some of the rules for our style-matching rule set; to see the entire rule set, check the `styleMatchingRules.js` source file in the `model` folder in the source code of the book:

```
[{
    "name": "match og style parameters",
    "description": "matches all original gravity parameters",
    "priority": 5,
    "on": 1,
    "condition": function (fact, cb) {
      fact.passOneMatches = [];

      angular.forEach(fact.styles, function(style){
        if(fact.recipe.estimatedOriginalGravity() >= style.OGMin
         && fact.recipe.estimatedOriginalGravity() <= style.OGMax)
        {
```

```
            fact.passOneMatches.push(style);
          }
        });
        cb(false);
      },
      "consequence": function (cb) {
          cb();
      }
    },
    …
    source removed for brevity
    …
    {
      "name": "match srm style parameters",
      "description": "matches all srm style parameters",
      "priority": 1,
      "on": 1,
      "condition": function (fact, cb) {
        fact.result = [];

        angular.forEach(fact.passFourMatches, function(style){
          if(fact.recipe.estimatedSRM() >= style.ColorMin &&
              fact.recipe.estimatedSRM() <= style.ColorMax) {
            fact.result.push(style);
          }
        });
        cb(true);
      },
      "consequence": function (cb) {
        cb();
      }
    }]
  });
```

The rules expect the facts that are passed into the rules engine to contain a recipe and an array of styles. The first rule iterates through each style and compares the original gravity values of the recipe with each of the styles and then pushes each style that matches onto a results array. The second rule iterates through the first rule's result array, compares the final gravity values of styles against the recipe's final gravity value, and pushes those that match onto a new result array. This continues for each of the rules until the final rule pushes the matching styles on to the results array and executes the callback by passing in `true` to exit out of the rules.

To call the rules engine from our controller, we just need to inject the `rulesEngine` service and the `rules` constants into our controller, and call it as follows:

```
rulesEngine.init(rules.styleRules);
var fact = { recipe: $scope.currentRecipe,
    styles: $scope.currentStyles };

rulesEngine.execute(fact, function(result){
      $scope.matchingStyles = result.result;
    });
```

If we composed our recipe correctly, it should match at least one of the various beer styles, and our controller should get a list of styles that match our recipe, which can be displayed to the user.

If we want to use the rules engine to execute other business rules in our application, all we need to do is repeat the preceding code, and provide a different set of rules and an object that contains the facts that the new rule set expects.

Using a rules engine can help us evaluate complex business rules with very little code. The majority of the work goes into formulating a rule set that evaluates as expected, and this can easily be tested using unit tests with canned code to exercise each rule. The unit test associated with the `rulesEngine` service provides a great example of how to use a unit test to test a rules set.

Summary

In this chapter, we discussed several ways in which we can implement business logic in our AngularJS applications. We discussed how to implement business logic as part of our models and how to encapsulate business logic as a service.

We then looked at a finite state machine that is used in our sample application to control the flow through the application as we are brewing a batch of beer.

We also looked at a rules engine we used in our application to find all of the beer styles that match our recipe as we formulate it. We also discussed how rules engines can be repurposed in our application by changing the rule set prior to having the rules engine evaluate the rules. This allows us to reduce the code we write to implement complex business logic.

In our next chapter, we will look at how all of the services we have discussed so far are used in our sample application to provide a web application that can be used to create beer recipes, schedule brew days, and help the brewer as they brew a batch of beer.

8
Putting It All Together

To tie everything together, we are going to look at a sample web application built upon the services we've covered throughout the book. The sample web application is a tool for home brewers to help with the formulating and brewing of beer recipes.

Brew Everywhere is an AngularJS application that provides home brewers with the tools to enjoy their hobby no matter what device they have or where they are located. Their recipes are stored in the cloud so that they can access them from all their devices with no need to copy files; just sign in to the application and a user can jump right into their latest recipe.

Brew Everywhere allows a home brewer to sign in to the application using either an internal data store or external authentication service. Once authenticated, brewers can view their recipes, create new recipes, schedule a brew day event, or brew a recipe.

It also integrates with Google's Calendar and Tasks so that a home brewer can create a brew day event or brewing task and have it alert them without having to be logged in to the application. This allows users of the application to schedule a brew day and set reminders to pick up brewing ingredients, prepare for the brew day, when to move a beer to a secondary fermentation, and when to bottle the beer. When these events reminders arrive, they'll notify the brewer via e-mail and post a notification to their Android devices, all without having to access the web application.

As the brewer formulates a recipe, it will constantly calculate the recipe's style parameters and provide a list of beer styles that best match the recipe, allowing them to tailor the recipe to meet a certain beer style. This comes in really handy whenever a brewer is formulating a recipe for a competition that requires the beer to match a specific style or group of styles.

Now that we've discussed what the sample application does, let's discuss how it uses the services we've created throughout this book to make these features work.

Wiring in authentication

First, we'll start with how the application handles authenticating a brewer. Back in *Chapter 4*, *Handling Cross-cutting Concerns*, we discussed the authenticate service that allows us to authenticate the user against either a data store or external authentication service such as Google+ or Facebook.

The authenticate service acts as a facade for three different services: the MonagoDB-based authentication service, the Google+ authentication service, and the Facebook authentication service. To invoke the three different methods of authentication, the login view publishes one of three different messages to start the process. Each one invokes a different function in the authenticate service: `login`, `loginWithGooglePlus`, and `loginWithFacebook` respectively.

As each of the services complete the authentication process, they send a message back to the authenticate service indicating the success or failure of the authentication. Once the authentication is complete, the authenticate service checks the user collection to find a matching username. If one exists, the authenticate service completes the login process by publishing the `_AUTHENTICATE_USER_COMPLETE_` message and passing the user object in the published message.

If a user does not exist in the users collection and the user authenticated with either Google+ or Facebook, the authenticate service creates a new user based on the external site's user data and inserts it into the user's collection and then publishes the authentication complete message.

Should the authentication fail, the authenticate service publishes the `_AUTHENTICATE_USER_FAILED_` message and publishes a message to the `error` service, which prompts the user that the login failed.

Displaying notifications and errors

The task of displaying messages to the user is carried out by several different services. The `error` service is responsible for managing the errors and the `notification` service is responsible for managing the user messages. These two services interact with the `notificationList` directive, which listens for notification events and displays the messages to the user on the web page.

Once the `notificationList` directive has displayed the messages to the user, it notifies the respective service to clear its cache of messages.

The second mechanism to display notifications to the user is the `dialog` directive. This directive uses the `messaging` service to listen for two events: `_DISPLAY_POPUP_` and `_DISPLAY_CONFIRMATION_`. Each message causes the dialog directive to display an HTML-based modal dialog to the user with the message text that was sent with the event. When the user responds to the modal dialog, the dialog directive publishes the `_USER_RESPONDED_` event with one of three responses: `OK`, `YES`, or `NO`, depending on the button clicked.

The final mechanism used to display notifications is the `waitSpinner` directive. This directive uses the messaging service to listen for two events: `_SERVER_REQUEST_STARTED_` and `_SERVER_REQUEST_ENDED_`. When the `_SERVER_REQUEST_STARTED_` message is received, the directive shows the DOM element it is bound to and when the `_SERVER_REQUEST_ENDED_` message is received, the directive hides the DOM element. Any component in the system can indicate that it is busy by publishing the `_SERVER_REQUEST_STARTED_` message and when it has finished, it can then publish the `_SERVER_REQUEST_ENDED_` message to indicate that it is no longer busy.

With these three groups of components, the application can keep the user up to date with the status of its processing, when it has issues, and when it needs the user to acknowledge something important.

Controlling the application flow

The application flow is controlled by two different mechanisms. The first is the standard AngularJS `routeProvider` service and the second is the `viewController` service that uses the `stateMachine` service.

The core of the application's navigation is managed by the `routeProvider` service. As the user clicks on links, the various controllers use the `$location` service to navigate to the various pages. This is the normal application flow as used by most AngularJS applications. You can see the routes that have been configured in the file `app.js` in the sample application's `source` folder.

When the user wants to brew one of his or her recipes, the application uses the `viewController` and `stateMachine` services to manage the application's flow. The `startBatch` controller loads the `stateMachine` service with the `brewingProcessStateMachine` state data and instantiates an instance of the `BrewProcessController` and adds it to the `viewController`. It then invokes the `navigateToBeginStateForActivity` function on the `viewController` service, which in turn asks the `stateMachine` controller to return the beginning state for the brewing process and then uses the `$location` service to load the appropriate view for the state.

As the user moves through the brewing process, the various controllers for each view that is displayed to the user will invoke the `getTargetStateForCommand` method on the `viewController` service to move to the state associated with the command. The `viewController` service also invokes the pre, post, and command functions associated with each state that exists on the `BrewProcessController`. This allows us to associate code with the transitions between each state of the state machine.

This continues until the end state of `brewProcess` and the `viewController` service navigates back to the home page of the application.

One of the things that is important is that the `routeProvider` service must have an entry for each URL of each state in the state data that points to the appropriate partial. This is because the `viewController` service leverages the `$location` service to navigate from view to view. By leveraging the `$location` service, we are able to reduce the amount of code needed by our application.

Displaying data from external services

The application utilizes data from three different sources: `mongolab.com`, Google Calendar, and Google Tasks.

The core application data, users, recipes, and ingredients are stored in a `mongolab.com` instance of MongoDB as collections.

The data-access strategy used by the application is to use data-specific services for each different type of data that act as facades for the `mongolab` service, which is a generic data-access service that interacts directly with the mongolab.com REST interface for each collection.

As the application requests data for a specific model, it publishes an event that is handled by the data-specific service, which in turn invokes the `mongolab` service that then sends the request to the `mongolab.com` REST interface and returns a promise to the calling data specific service. When the promise completes, the data-specific service publishes an event indicating the completion of the data request.

The reason the data-specific services use messages to communicate with the application is to allow other controllers and services in the application to receive notifications when data they are interested in is updated. This allows us to keep round trips to the servers to a minimum and provides the ability for the application to cache data between calls.

The `mongolab` service is the only service in the application that does not use the `messaging` service to communicate with its consumers. This is because each call to the `mongolab` service needs to be handled directly by the consumer that invoked it. If we had used the `messaging` service, we could not have made the service generic enough to query the various collections in the `MongoLab` database that an application might use. Also, there is a possibility that data returned by the `mongolab` service might not be processed by the service that requested it if the `mongolab` service published a generic event.

If the user uses Google+ to authenticate with the application, two additional features are turned on: the brewing calendar and the brewing tasklist. The reason these features are only available based on the authentication type is because we don't want to have the user authenticating with a different service each time they access a feature in the application.

The `brewingCalendar` service leverages Google Calendar to store events in the user's Google Calendar, so they can access them at any time and on any device that can connect to Google Calendar. The `brewingCalendar` service also leverages Google Tasks to create brewing tasklist items.

The same data-access strategy is used by using a data-specific service to act as a facade for a more generic service that interacts with the external Google Calendar and Google Tasks cloud services.

We covered both the `googleCalendar` and `googleTasks` services in *Chapter 6*, *Mashing in External Services*, so we won't go into detail about them here; however, we will cover how the application controllers interact with the services to display their data.

The home controller displays both the brewing calendar events and brewing tasks whenever the user logs in using Google+ authentication. Once the home controller receives an event that a user has logged in to the application, it publishes events to get both the brewing calendar and brewing tasklist from the `brewingCalendar` service.

These events in turn cause the `brewingCalendar` service to publish events to the `googleCalendar` and `googleTask` services to request the user's brewing calendar and brewing tasklist.

When the `googleCalendar` service finishes retrieving the user's brewing calendar, it publishes the `_GET_CALENDAR_COMPLETE_` event, which causes the `brewingCalendar` service to process the returned data and publish the `_GET_BREWING_CALENDAR_COMPLETE_` event that causes the home controller to update its internal array of brewing events with the result being displayed in the view.

Also, when the `googleTask` service finishes retrieving the brewing tasklist, it publishes the `_GET_TASK_LIST_COMPLETE_` event, which causes the `brewingCalendar` service to process the returned data and publish the `_GET_BREWING_CALENDAR_COMPLETE_` event that allows the home controller to update its internal array of brewing tasks, in turn updating the data displayed in the view.

The home controller also publishes a `_GET_RECIPES_` event that causes the `recipeDataService` to call the query method on the `mongolab` service to retrieve the recipes in the recipe collection. The `mongolab` service in turn sends an `$http GET` request to the `MongoLab` REST service and returns the promise to `recipeDataService`. Once the call to the REST service returns, the `recipeDataService` handles the response and transforms the returned data into instances of the internal recipe model and then it pushes each recipe onto an internal array. Then, it publishes a `_GET_RECIPES_COMPLETE_` event and returns the resulting array as part of the event.

When the home controller receives the `_GET_RECIPES_COMPLETE_` event, it stores the result to its internal recipe array that in turn causes the view to display the list of recipes.

The preceding sequence of events occurs each time a controller or service publishes an event to retrieve data from one of the data-specific services. You can see similar examples in the recipe controller source file `recipe.js` in the sample project's source folder.

You can also see another example of how controllers interact with the `googleCalendar` and `googleTasks` services by checking out the `scheduleBrewDay` controller. Once the user selects a recipe, a date, and the event options, the `scheduleBrewDay` controller creates new calendar events and tasks by publishing a `_CREATE_BREW_DAY_EVENTS_` event and passing in the data needed to create the calendar events and tasks.

The `brewingCalendar` service listens for the `_CREATE_BREW_DAY_EVENTS_` event and when it receives the event, it executes a series of internal functions that call the `googleCalendar` service to create each calendar event and call the `googleTasks` service to create each task.

Building and calculating the recipe

As the user creates a new recipe, several different services are used along with the data models we've discussed throughout the book. The recipe controller in the `recipe.js` file under the `partial/recipe` folder in the sample application starts off by publishing events to request the following collections: fermentables, adjuncts, equipment, hops, mash profiles, styles, water profiles, and yeast. Once these collections are retrieved from the `MongoLab` REST service, the recipe controller

stores them off to internal arrays that it then uses to display tables of the various ingredients as the user navigates the various tabs of the recipe view.

When the user adds an ingredient to the recipe, the recipe model iterates through each of the recipe's arrays of ingredient models to calculate the following beer style parameters; original gravity, estimated final gravity, alcohol by volume, bitterness in **International Bittering Units** (**IBUs**), and color. This recalculation happens as a side effect of the AngularJS view binding. Each time the recipe is modified by adding or removing an ingredient on a tab, it causes the AngularJS watch digest to go through and re-evalute each of the view's bindings, some of which are bound to the recipe model's functions that cause the recipe to recalculate the values that are bound to the view.

The add buttons on the various tabs also invoke the `rulesEngine` service, which runs a rule set against the recipe that finds the various beer styles that match the beer's style parameters. This gives the brewer an idea of which beer style best matches their recipe. This dynamic calculation of the recipe model's style parameters and execution of the style matching rule set allows the brewer to play around with their recipe to fine-tune it to match the style they want.

Once the user is happy with their recipe, they can save it back to the recipe collection by publishing a `_CREATE_RECIPE_` event that causes the `recipeDataService` to invoke the `mongolab` service to `POST` the newly created recipe to the recipe collection on `mongolab.com`. If the user is editing an existing recipe, the recipe controller instead publishes an `_UPDATE_RECIPE_` event that results in a recipe being posted to the collection, updating the existing record in the collection.

One of the interesting things you'll notice is that the save recipe function in the recipe controller strips off the object IDs of each of the recipe's ingredients. This has to be done because the AngualrJS JSON serializer fails if it comes across anything that has a `$` sign deep down in an object's structure. So, to get around this, it is easier to just strip off the object IDs and the recipe doesn't really need to keep the IDs, as they are just an artifact of MongoDB's automatic keying of data.

One last thing to note in the recipe controller is the paging used on each of the tabs of ingredients. You will notice almost half of the code in the controller deals with managing the paging through each of the arrays of ingredients that are displayed on each of the four ingredient tabs. This code is required to use the Angular UI pagination directive that is used at the bottom of each table of ingredients. One method is used to store off the current page and the second method calculates the starting index of the page that is used by the `startFrom` filter to return back the subset of ingredients to display in the paginated table.

Messaging is the heart of the application

One of the services that we've touched on throughout this chapter but never really mentioned is the `messaging` service. It is the workhorse of this application and is responsible for delivering events between the various services, controllers, and directives used in the application.

All of the controllers, directives, and services interact with each other via events. Throughout the source code, you'll see the various components either call `messaging.subscribe` or `messaging.publish` to subscribe to and publish events respectively.

The only time a promise is used in the application is where we need to ensure that the results of the `mongolab` service functions are handled by the consumer that made the call to the service. This ensures that the returned data is handled properly by the appropriate data-specific service.

The advantage of using the publish/subscribe pattern allows us to broadcast an event and have more than one consumer act on the event, which comes in handy when you have multiple controllers displayed on the page that needs to be notified when the data they are displaying changes in relation to an event that occurs in another controller or service, something that can't be done easily with promises or callbacks.

Another architectural aspect the `messaging` service provides is loosely coupled components. Because each component communicates with each other using messages, we can easily replace a component with little additional coding. All the new component needs to do is to publish and subscribe to the same events as the previous component and we should not have to make any other code changes to our application, effectively making our code change a drop in replacement.

This loose coupling also makes our application much easier to test. Since each component publishes and subscribes to messages, there are fewer dependencies that we need to mock out in our unit tests.

Summary

Hopefully, walking through how the Brew Everywhere application is put together has helped you to see how we can use different services to build a first rate application.

We followed AngularJS best practices to partition our code across controllers, directives, models, and services. We kept our controllers as thin as possible by using controller inheritance to encapsulate common code across the controllers into a base controller that each controller inherits from by using the `$controller` service.

We used directives to encapsulate code that manipulates the DOM and validate data. This again follows best practices by limiting DOM manipulation to the correct component and helps to keep our controller as thin as possible by moving validation code out of the controller to directives where it works best.

Next, we used value providers to define the models used within our application to provide both state and function to our models. We saw how this comes in handy when we need to iterate through arrays of model objects to calculate beer style parameters.

Finally, we used services throughout the application to manage data, control application flow, and interact with external services to reduce the amount of code we needed to build out the functionality of our application. In fact, if you notice, the Brew Everywhere application does everything without a dedicated server running in the background. Data used by the application is stored in the cloud via `mongolab.com`; calendar events and tasks are stored and managed by Google's Calendar and Tasks services. The application even utilizes Google+ and Facebook to authenticate users of the application.

Our core business logic is also encapsulated in specialized services such as finite state machines and rules engines, allowing us to change the business logic or rules set on the fly based on where in the application the user happens to be.

We also followed the best practice of single responsibility per component especially in our data-access services. We built services that specialized in managing each of the data models used in our application and then leveraged generic services to interact with the external services used to store the various data.

Hopefully this book and the services discussed will inspire you to look at your AngularJS applications differently and think about how you can leverage services to make your application more robust, better architected, and easier to maintain. Just remember, with a good library of services at your hands, you have a toolset that you can use to build just about any type of application you need.

Index

Symbols

$cacheFactory service 79-81
$inject service 22

A

addGetIngredientsTask method 103
addPrimaryToCalendar method 104
afterEach method 35
AngularJS
best practices 8
components 7
application analytics
logging 61-65
application flow
controlling 127, 128
assignments
constructors, limiting to 20
authenticate service 126
authentication
performing, OAuth 2.0 used 66-69
wiring in 126

B

backend communications
mocking 45-47
basics, test scenario 33-35
beforeEach function 35, 36
Behavioral Driven Development (BDD) 33
best practices, AngularJS 8
BreezeJS
URL 82
brewCalendar service 99
brewingCalendar service 129
brewing task list
building, Google Tasks used 94-98
business logic
encapsulating, in models 108-110
encapsulating, in services 111-113

C

callback function 52
code
service, structuring in 28-31
complex data
validating, with rules engine 120-123
configuration, service 31
constant method 22
constructors
limiting, to assignments 20
controllers
about 8
functionalities 8, 9
createBrewDayEvents method 102
CRUD data service
implementing 75-79

D

data
caching, for network traffic reduction 79-82
displaying, from external services 128-130
mocking 36-38
transforming, in service 82-87
dependencies
handling, that return promises 42-44
dependency injection 17-20
digest() method 44

directives
about 9
functionalities 9, 10
Document Object Model (DOM) 8

E

errors
displaying 126, 127
logging 61-65
events
storing, with Google Calendar 89-94
external services
data, displaying from 128-130

F

Facebook 126
factories
overview 22-28
factory method 26
Fermentable model 72
fermentableType filter 83

G

getAdjuncts method 80-82
getBrewingTaskList method 101
getTargetStateForCommand function 120
GitHub
URL, for $cacheFactory service 81
Google+ 90, 126
Google+ API 66
Google Calendar
events, storing with 89-94
Google Calendar, and task list
combining 98-104
Google Calendar API
about 90
URL 94
Google+ JavaScript API
URL 69
Google Tasks
used, for building brewing task list 94-98
Google Tasks API
URL 98

I

Immediately-Invoked Function Expression (IIFE) 29
inject method 36
Integrated Development Environments (IDEs) 15
International Bittering Units (IBUs) 131

J

Jasmine
URL, for documentation 41
Jasmine framework 33
Jasmine spies
services, mocking with 40, 41

L

Law of Demeter 16, 17
log4javascript
about 61
URL 63

M

messaging service 132
models
business logic, encapsulating in 108-110
business logic, providing 71-74
selecting 113, 114
state of object, maintaining 71-74
modules
loading, in scenario 35, 36
MongoLab
URL 75
MongoLab REST API 75
mongolab service 77

N

network traffic reduction
data, caching for 79-82
notifications
displaying 126, 127

O

OAuth 66
OAuth 2.0
used, for authentication 66-69
OpenID 66

P

patterns
used, for communicating with service's consumers 49-54
promises
using, sparingly 21
provider method 24, 27, 31
providers
overview 22-28
publish method 52
publish/subscribe pattern
about 50
advantages 132

R

recipe
building 130, 131
calculating 130, 131
retrieveUser method 42
Revealing Module Pattern 29
rules engine
complex data, validating with 120-123

S

scenario
modules, loading in 35, 36
scenario file 34, 35
service interface, defining
about 13
consistency, maintaining of method names 15
focus on developer 14
functionality, executing based on function parameters 15
interface, documenting 15
readability, favoring over brevity 14
services, limiting to single area of responsibility 14
top usage scenarios 15
service method 24
services
about 7, 10
business logic, encapsulating in 111-113
configuring 31
data, transforming in 82-87
functionalities 10, 11
mocking 38
mocking, with Jasmine spies 40, 41
overview 22-28
selecting 113, 114
structuring, in code 28-31
service's consumers
patterns, used for communicating with 49-54
services, designing for testability
about 16
constructors, limiting to assignments 20
Law of Demeter 16, 17
promises, using sparingly 21
required dependencies, passing 17-20
Single Sign-On 66
Standard Reference Method (SRM) 14
state machine
view flow, controlling with 114-120
subscribe method 52

T

test scenario
basics 33-35
timers
mocking 47, 48

U

unsubscribe method 52
user notifications
managing 54-60

V

value method 22
view flow
controlling, with state machine 114-120

W

waitSpinner directive 54, 55

Thank you for buying

AngularJS Services

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

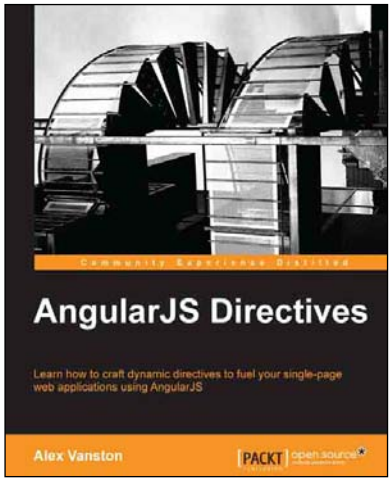
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

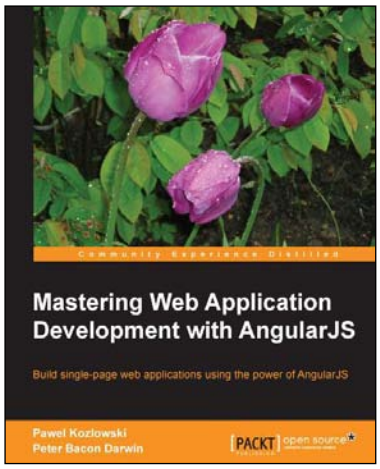


AngularJS Directives

ISBN: 978-1-78328-033-9 Paperback: 110 pages

Learn how to craft dynamic directives to fuel your single-page web applications using AngularJS

1. Learn how to build an AngularJS directive.
2. Create extendable modules for plug-and-play usability.
3. Build apps that react in real time to changes in your data model.



Mastering Web Application Development with AngularJS

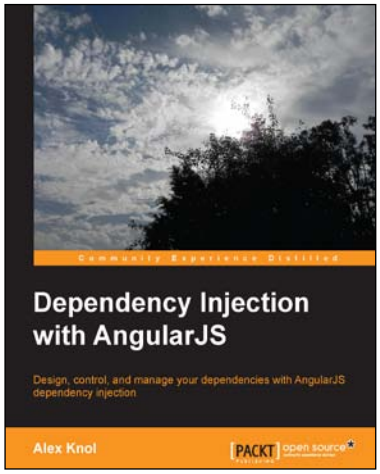
ISBN: 978-1-78216-182-0 Paperback: 372 pages

Build single-page web applications using the power of AngularJS

1. Make the most out of AngularJS by understanding the AngularJS philosophy and applying it to real-life development tasks.
2. Effectively structure, write, test, and finally deploy your application.
3. Add security and optimization features to your AngularJS applications.

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
community experience distilled
PUBLISHING

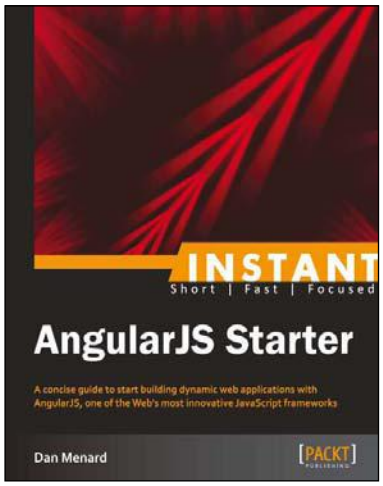


Dependency Injection with AngularJS

ISBN: 978-1-78216-656-6 Paperback: 78 pages

Design, control, and manage your dependencies with AngularJS dependency injection

1. Understand the concept of dependency injection.
2. Isolate units of code during testing JavaScript using Jasmine.
3. Create reusable components in AngularJS.



Instant AngularJS Starter

ISBN: 978-1-78216-676-4 Paperback: 66 pages

A concise guide to start building dynamic web applications with AngularJS, one of the Web's most innovative JavaScript frameworks

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Take a broad look at the capabilities of AngularJS, with in-depth analysis of its key features.
3. See how to build a structured MVC-style application that will scale gracefully in real-world applications.

Please check **www.PacktPub.com** for information on our titles