Community Experience Distilled

Responsive Web Design with AngularJS

Leverage the core functionalities of AngularJS, to build responsive single page applications

Sandeep Kumar Patel

[PACKT] open source* PUBLISHING community experience distilled

Responsive Web Design with AngularJS

Leverage the core functionalities of AngularJS, to build responsive single page applications

Sandeep Kumar Patel

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Responsive Web Design with AngularJS

First published: December 2014

Production reference: 1131214

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

ISBN 978-1-78439-842-2

www.packtpub.com

Credits

Author
Sandeep Kumar Patel

Reviewers
Anthony Chu
Shaunak De
Jude Osborn

Commissioning Editor
Julian Ursell

Acquisition Editor
Subho Gupta

Content Development Editor
Arwa Manasawala

Technical Editor
Utkarsha S. Kadam

Copy Editor
Merilyn Pereira

Project Coordinator
Purav Motiwalla

Proofreaders
Simran Bhogal
Ameesha Green

Indexer
Rekha Nair

Graphics
Abhinash Sahu

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

About the Author

Sandeep Kumar Patel is a senior web developer and founder of `www.tutorialsavvy.com`, a widely read programming blog created in 2012. He has more than 5 years of experience in object-oriented JavaScript and JSON-based web applications development. He is GATE-2005 Information Technology (IT) qualified and has a Master's degree from VIT University, Vellore. At present, he works as a web developer at SAP Labs India. You can learn more about him from his LinkedIn profile at `http://www.linkedin.com/in/techblogger`. He has received the Dzone Most Valuable Blogger (MVB) award for technical publications related to web technologies. His articles can be viewed at `http://www.dzone.com/users/sandeepgiet`. He has also received the Java Code Geek (JCG) badge for a technical article published in JCG. His article can be viewed at `http://www.javacodegeeks.com/author/sandeep-kumar-patel/`.

He has also worked on: *Instant GSON* and *Developing Responsive Web Applications with AJAX and jQuery*, both by Packt Publishing.

I would like to thank the three most important people in my life, my parents, Dilip Kumar Patel and Sanjukta Patel, for their love and my wife, Surabhi Patel, for her support and the joy that she has brought to my life.

A special thanks to the team at Packt Publishing without whom this book wouldn't have been possible.

About the Reviewers

Anthony Chu has been developing web applications for over 15 years. As a lead developer and architect, he works with a team of talented developers building AngularJS applications on Microsoft's ASP.NET and the Azure stack. Anthony lives in Vancouver, Canada with his wife and two children. He blogs at `anthonychu.com` and his Twitter handle is `@nthonyChu`.

Shaunak De has been working with imaging, Web-technologies, cluster and cloud computing for over 8 years. He has keen interest in the developments of backends and scientific computing for the Web. A valedictorian of the University of Mumbai, he is currently pursuing his PhD at the Indian Institute of Technology in the domain of Deep Learning.

Shaunak can be found on Twitter `@shaunakde` and on his journal at `http://shaunak.ws`.

Jude Osborn is a creative developer for Google's Creative Lab in Sydney, on behalf of development agency, Potato. Originally from the US, he has travelled a lot. Jude's experience spans 20 years of software development, including the desktop, Web, and mobile. He loves to learn and play with sweet new technologies, and is currently thoroughly enjoying WebGL, AngularJS, and Chromecast.

Every day Jude walks across Sydney's Pyrmont Bridge, soaking up the sunshine and looking forward to his next technological challenge.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

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.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Welcome to *Responsive Web Design with AngularJS*. If you want to learn and understand responsive web application development using AngularJS, then this book is for you. It covers a systematic approach to build a responsive web application.

All the key features of AngularJS that can help in building a responsive application are explained with the detailed code. This book also explains how to debug and test an AngularJS-based web application during development.

What this book covers

Chapter 1, *Introduction to Responsive Single Page Application and AngularJS*, introduces you to responsive design, single page application, and the AngularJS library. This chapter also gives a kick start of the single page responsive application that we are going to build to demonstrate the AngularJS role in application development.

Chapter 2, *The AngularJS Dynamic Routing-based Approach*, explores the power of AngularJS-based routing of templates. It also explores the use of AngularJS routing for responsive web application development.

Chapter 3, *The AngularJS Directive-based Approach*, introduces the custom directive development in Angular JS. It also demonstrates the building of custom directives to address responsive web application development.

Chapter 4, *The AngularJS-based Breakpoints for Layout Manipulation*, introduces the CSS3 breakpoint concept for responsive layout development. It also provides coded examples in AngularJS to present the breakpoint concept in the context of web design.

Chapter 5, *Debugging and Testing Responsive Applications*, provides a list of debugging tools for AngularJS-based applications.

What you need for this book

The tools and libraries required for this book are as follows:

- WebStorm or Sublime Text
- The WAMP server or WebStorm default server
- AngularJS 1.3
- SASS
- COMPASS
- Ruby 64-bit

Who this book is for

If you are an AngularJS developer who wants to create responsive web applications, this is the book for you. It is also helpful for those who want to learn different approaches provided by AngularJS to develop a responsive single page web application. Finally, the book is for anyone who wants to understand AngularJS-based responsive web application development.

Conventions

In this book, you will find a number of text styles 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: "The `$window` object has many useful properties that can be used for application development."

A block of code is set as follows:

```
<div class="row">
    <div class="pic">
        <h1 class="name">Sandeep Kumar Patel</h1>
        <img class="profile-image"
          src="http://www.gravatar.com/avatar/4205261332ff1
          31e971b48db31dcb528.png"
          alt="profile image"/>
    </div>
</div>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
angular.module('profileApp.profileServices', [])
.service('ProfileServices',["$resource", function($resource){
   return{
/*Method for getting personal detail JSON file*/
       getPersonalDetail : function(){
           return $resource("data/personalDetail.json")
       },
/*Method for getting professional detail JSON file*/
       getProfessionalDetail : function(){
           return $resource("data/professionalDetail.json")
       }`
   }
}]);
```

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

```
require 'compass/import-once/activate'
#root of your project when deployed:
http_path = "/"
css_dir = "css"
sass_dir = "sass"
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "To use Batarang you need to select the **Enabled** checkbox to debug the AngularJS application."

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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in 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 at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt Publishing 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 from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. 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 could 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 to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt Publishing, 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

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Introduction to Responsive Single Page Application and AngularJS

In this chapter, you will learn about the need for responsive web design, explore the building blocks of a single page application. At the end, we will discuss some features of the AngularJS library with a quick look at their syntax and understand their role in responsive web application development.

Why responsive design?

In the current age of digital revolution, there are many devices with different screen sizes. This increases an additional layer of complexity for web development. A web application now has to present a similar experience across different devices. To solve this problem, it needs a design solution .This solution is termed as responsive design. There are also some key areas in the responsive design approach. They are as follows:

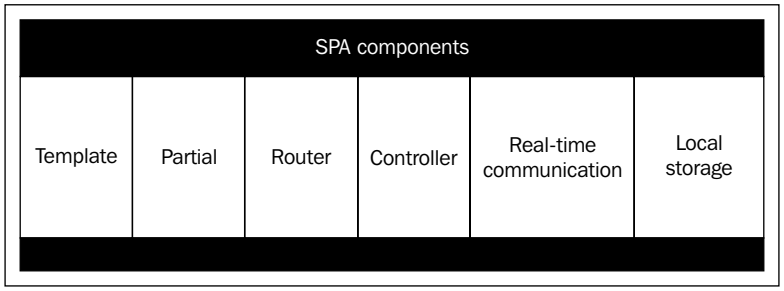
- **Increase in the use of handheld devices (mobile, tablet)**: One example is data analytics provider, Flurry Inc. It released data about mobile application usage. The published report shows the mobile application usage is increasing. You can view this report at `http://www.flurry.com`.
- **A single codebase is easier to manage**: From a developer's point of view, it is easier to manage a single code repository base for different devices. The developer has to make a change in one place for a change request in the functionality.

- **Better Engaging content**: Based on the device, the view will be optimized through responsive web development, which increases the user's interest. Hence, the engagement of the user will remain the same as the original application.
- **Conversion rate and lead generation**: Through responsive web development, an optimized view will be presented to the user which will increase engagement and thus the probability of a higher conversion rate and lead generation.
- **Better user experience**: Using responsive design, we can show better content to the user that includes appropriate images and widgets. This increases the user experience for the application.
- **Page load speed**: Using responsive design, only required content can be downloaded to the targeted device; hence, the page speed will increase.

What is single page web application?

In a **single page application** (**SPA**) architecture, the presentation logic is moved to the client side. SPA can redraw a part of the UI on a page without a complete round trip or reload of a window.

The following diagram shows the building blocks of a single page web application:



Template

Templates can be thought of as placeholders for styling and structure that will be modified by the application's code. This way the content to be displayed can be generated dynamically by the app, while the presentation remains static. In web application development, domain templating is a very well known concept and has been used by developers. In a more specific sentence, we can define a template as any document or file with a presentation format that can be created initially and does not have to be recreated for subsequent use.

To know more about the latest developments in templating, visit `http://www.html5rocks.com/en/tutorials/webcomponents/template`.

Partial

The partial components is similar to a template that can be created once and reused many times. The only difference between partial and template is that partial is the smallest unit of a reusable chunk of code.

Router

The router components is used for correctly navigation to a view based on the request. In single page application, the routing logic is present in the client side.

Controller

The controller components is the owner of a part of a page. It provides a scope to be used by the specific part of the page. In single page application, controller is used to share objects or create common behavior in terms of event callback functions.

Real-time communication

Real-time communication introduces the two-way request-response mechanism. Technologies like WebSocket, **Server-Sent Event** (**SSE**), and WebRTC made this possible. In single page application, real-time communication is used to share the load between the server and client. Instead of polling from the client side, now push updates from the server side can now be used to sync the application.

Local storage

Local storage provides a client-based storage system to cache the data in the browser. This really helps by reducing the number of HTTP requests to the server. In single page application, the client first checks the data required by a request in the local storage and if it is not present in local, then it makes a call to the server.

Understanding responsive single page application

In the previous section, we explored responsive and single page applications, two modern features of a web application. In this book, we will develop an SPA with responsive design using AngularJS, which is a perfect fit for SPA development as all the characteristics of SPA are present within the library. In the next section, we will quickly go through these features.

What is an AngularJS framework?

AngularJS is a complete client-side solution for web application development and is maintained by Google Inc. The AngularJS framework is one of the best frameworks to create single page application. AngularJS follows declarative programming similar to an HTML element declaration. This makes AngularJS simple.

The AngularJS library is a perfect fit for SPA development. AngularJS is more like a **Model View Whatever** (**MV*/MVW**) design pattern instead of **Model view Controller** (**MVC**).This means that an AngularJS application can be developed with the model and view features.

To know more about MVW, visit `https://plus.google.com/+AngularJS/posts/aZNVhj355G2`.

Exploring the features of AngularJS

In this section, we will quickly go through some of the important features provided by the AngularJS library. The goal of this section is to understand each feature with its appropriate syntax.

Some of the features that we are interested in to build responsive web applications are listed as follows:

- The AngularJS module
- The controller scope
- The AngularJS routing module
- The provider
- Data binding
- Angular expressions

- Built-in directive
- Custom directive

The AngularJS module

Using the AngularJS module feature, an independent code section can be created. A module can be easily detachable. This module helps with better code management and it helps to work in large teams. Loose coupling allows developers to create their parts of the project independently. The `angular.module()` method is used to create a module. The syntax to create an AngularJS module is as follows:

```
var aModule = angular.module("<moduleName>",
["<injectedModule1>","<injectedModule2>"]);
```

In the above code, the `angular.module()` function takes two parameters: the first is the module name and second is the array of dependent module. The parameters used to create a module are as follows:

- **moduleName**: This represents the name of the module registered to AngularJS
- **injectedModule**: In the above syntax, `injectedModule1` and `injectedModule2` are modules that are injected as input to the targeted module

To learn more about the AngularJS module, visit `https://docs.angularjs.org/guide/module`.

The controller scope

AngularJS provides the controller scope to create a new controller to manipulate HTML DOM indirectly under its scope by modifying properties present inside its scope. AngularJS uses the controller feature to perform the following operations:

- Setting up the scope object inside the controller
- Modifying the value of the scope object inside the controller

The `controller()` method is used to create a controller section. The syntax to create a controller is as follows:

```
var aModule = angular.module("<moduleName>",
    ["<injectedModule1>","<injectedModule2>"]);
aModule.controller('<controllerName>',
    ['<injector1>','<injector2>',
```

```
    function(injector1,injector2) {
    //Definition of the controller
}]);
```

Some parameters used in the preceding syntax to create a controller section are as follows:

- **controllerName**: This represents the name of the controller created using the AngularJS library
- **injector**: In the above syntax, injector1 and injector2 are individual modules that are injected to be used by the controller scope

A controller scope can be defined inside HTML DOM using the `ng-controller` directive. The following code shows the use of the controller directive in the HTML file:

```
<div ng-controller="<controllerName>">
    <! -- HTML element inside the controller scope -->
</div>
```

To learn more about the AngularJS controller, visit: `https://docs.angularjs.org/guide/controller`.

The AngularJS routing module

AngularJS provides a router module to determine which HTML template or partial will load based on the request. AngularJS provides the `ngRoute` module with `$routeProvider` to route the request to the appropriate view. The following code shows the syntax to create the route to the request:

```
var myApplication = angular.module("<applicationName>",
    ["ngRoute"]);
myApplication.config(function($routeProvider) {
   $routeProvider.when('/viewName1', {
      templateUrl:'/partial1.html'
   });
   $routeProvider.when('/viewName2', {
      templateUrl:'/partial2.html'
   });
   $routeProvider.otherwise({redirectTo:'/viewName1'});
});
```

The details of the preceding code are as follows:

- `applicationName`: This represents the module name of the application.
- `config()`: This function configures the routing module for the incoming request.
- `viewName`: This represent structure of the incoming request pattern. In the previous code, the incoming requests that are handled are `viewName1` and `viewName2`.
- `when` and `otherwise`: These are the associated clauses to redirect the request to the targeted HTML view.
- `templateUrl` and `redirectTo`: `templateUrl` contains the address of the URL for the targeted response and `redirectTo` points to default route for a request.

To learn more about the AngularJS router provider, visit `https://docs.angularjs.org/api/ngRoute/provider/$routeProvider`.

The AngularJS provider

AngularJS provider is the core feature through which you can serve an API throughout an application. Angular JS-based value, factory, services, and constants are syntactic sugar on top of core provider implementation. Let's take a look at each implementation on the core provider in the following section. A provider can be created using the `provider()` method and an object with a `$get` function that returns an instance. The angularJS provider creates only one instance of itself to be used. The syntax to create a AngularJS provider is as follows:

```
var myApplication = angular.module("<applicationName>",[]);
myApplication.provider("<theProviderName>", {
    //Code for the provider implementation
    $get: function() {
        return //object that will be used by the caller;
    }
});
```

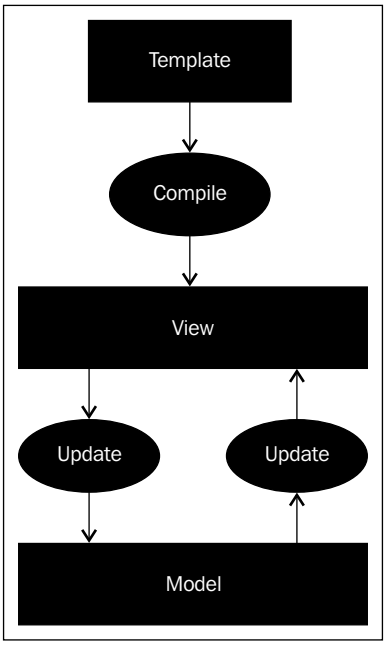
A provider method can be injected and called inside a configuration block or module. The following code shows the use of a declared provider inside the configuration block:

```
var myApplication = angular.module("<applicationName>",[]);
myApplication.config(function (theProviderNameProvider) {
//code to access theProviderNameProvider
})
```

Data binding

AngularJS provides scope-based data binding from HTML, DOM, and JSON data. Data binding happens in two different steps: compiling and linking. In the compile phase, the HTML file is converted into a JavaScript function. In the linking phase, the real data is linked to the HTML element. Angular provides two-way binding between the view and model features by synchronizing JavaScript objects and HTML elements.

The following diagram shows the process of data binding, where **template** is compiled once and **view** is generated by linking to **model**. This shows the tight coupling between the model and view features where any change in either of them is in sync.



AngularJS expressions

Expressions in AngularJS are like JavaScript statements that can be evaluated at runtime. These expressions are represented by double curly braces. They involve the JavaScript value object whose exact value arrives while linking the time of model objects. The following code shows the syntax of using AngularJS expression:

```
{{expression}}
```

An example of AngularJS expressions are as follows:.

```
<div>
    Addition of 2 and 3 are {{2+3}}
</div>
```

AngularJS also provides a one-time binding of an expression. The one-time binding expression value, once set, will not change. In simple words, if a variable is being updated on every click, normal binding would cause the display to change each time the variable changes. However, in a one-time bind expression, the display will remain set to the first valid value. A one-time expression can be defined using double colon (::).The following code shows the use of one-time expression binding:

```
<div>
    Name of the student is {{::studentName }}
</div>
```

One-time binding of the scope variable value was introduced in AngularJS version 1.3.

Built-in directive

AngularJS provides many built-in directives to help with web application development. These core directives are prefixed with `ng-*`.It should be noted that all core directives are prefixed with the `ng` keyword and should not be used while creating a custom directive. Some of the built-in directives present inside the AngularJS library are as follows:

- **ngApp**: This represents the root element of the application.
- **ngRepeat**: This directive is used to iterate an array or properties inside the object.
- **ngIf**: This directive is used to evaluate conditional expression and adds or removes the element from the DOM based on its result.
- **ngClick**: This directive is used to attach a custom callback for a click event.

- **ngInclude**: This directive is used to fetch and compile the external HTML fragment and include it to the current document.
- **ngClass**: This directive is used to manipulate CSS classes that are used for an element.
- **ngBind**: This directive is used to replace the text content with the specified HTML. The `ngBindHTML` replaces an element's inner HTML with the specified HTML.
- **ngSubmit**: This directive is used to attach a custom callback method to an onSubmit event of a form.
- **ngModel**: This directive is used to attach a form element to a scope.

To learn more about built-in directives and for the list of all core directives inside the AngularJS library, visit `https://docs.angularjs.org/api/ng#directive`.

Custom directive

AngularJS follows the directive-based approach for reusable component development. A custom directive is similar to an HTML element with its own definition and method to manipulate the DOM. AngularJS provides the `directive()` method to create a custom directive. The following code shows the syntax to create a custom directive:

```
var myApplication = angular.module("<applicationName>",[]);
myApplication.directive("<directiveName>", function() {
    return {
        restrict: "<represent the usage of directive>",
  require:"<Dependent module>"
        scope: {
           //Scope variable declaration
        },
        template:"<HTML template string>",
        templateUrl:"<URL of the HTML template>",
        replace: "<Boolean value>",
        priority:"<Number value>",
        terminal:"<Boolean value>",
  transclude: "<Boolean value>",
        controller:  function ($scope, $element, $attrs){
            //Code for scope object and behavior
        },
        link: function (scope, element, attrs) {
            //Code for link phase
```

```
            }
        }
    });
```

The details of the properties shown in the preceding code are as follows:

- `restrict`: This property controls the use of the custom directive. A directive in AngularJS can be used as Attribute (A), Element (E), Class (C), and Comment (M). It can have the value of any combination of A, E, C, and M. An example of this attribute is as follows:

  ```
  restrict : "AC"
  ```

  The preceding code designates the custom directive and can only be used as an attribute or a class. The default value of the restrict property is **A** if nothing is supplied to it.

- `require`: This property is to inject other required directives or set of directives. It takes a string or array of strings of the directive name as an input.
- `scope`: This property is very vital as it determines the current scope of the directive. It takes two types of values, either true or an unanimous JavaScript object `{}` with some attribute and values. Some uses of these properties are as follows:
  - `true`: This creates a new scope with access to the current parent scope(normal scope).
  - `{}`: This creates a new scope and separates it to the current scope without access to the parent directive scope (isolated scope). This object has the following attributes to configure: `@`, `&`, and `=`.
- `template`: This property takes its value as an HTML markup string. This mark-up will be replaced/appended to the current element. An example of this attribute are as follows:

  ```
  template:"<h2>Hello</h2>"
  ```

  The preceding code designates a template string that will replace/append the current element.

- `templateUrl`: This property is an alternative option for a template. This property accepts a URL value.

- `replace`: This property takes a Boolean (true or false) value as an input. If it is true, it replaces the current element with the given template. If it is false, it appends the template to the current element, for example, replacing `true` designates that the current element will be removed and the template will be placed in that position.
- `priority`: This property takes a number as a value. This property determines the execution order of directives. A higher value of priority implies that it will run first. The default value of priority for a directive is 0.
- `terminal`: This property takes a Boolean (true or false) as the value. When this property is set to `true`, the directive will execute.
- `transclude`: This property takes `true` or `element` as the value. Based on the input value, it compiles the current directive or elements and enables it to present a directive.
- `controller`: This property is used as an initializer and can be used to pass values among directives. It has the following property access: `$scope`, `$element`, and `$attr`.
- `link`: This property is used to write code to manipulate the DOM element inside the scope, for example, a click event listener callback can be written inside it.

To find out more about creating custom directives, visit `https://docs.angularjs.org/guide/directive`.

Role of AngularJS

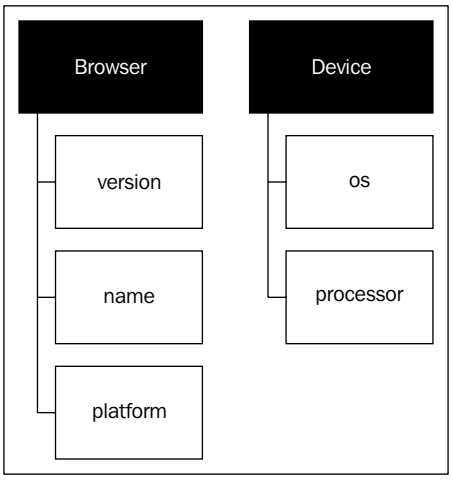
In this section, we will explore the role of AngularJS for responsive web development. Before going into AngularJS, you will learn about responsive web development in general. Responsive web development can be performed in two ways:

- Using the browser sniffing approach
- Using the CSS3 media queries approach

Using the browser sniffing approach

When we view web pages through our browser, the browser sends a user agent string to the server. This string provides information like browser and device details. Reading these details, the browser can be redirected to the appropriate view. This method of reading client details is known as browser sniffing.

The browser string has a lot of different information about the source from where the request is generated. The following diagram shows the information shared by the user string:



Details of the parameters present in the user agent string are as follows:

- **Browser name**: This represents the actual name of the browser from where the request has originated, for example, Mozilla or Opera
- **Browser version**: This represents the browser release version from the vendor, for example, Firefox has the latest version 31
- **Browser platform**: This represents the underlying engine on which the browser is running, for example, Trident or WebKit
- **Device OS**: This represents the operating system running on the device from where the request has originated, for example, Linux or Windows
- **Device processor**: This represents the processor type on which the operating system is running, for example, 32 or 64 bit

A different browser string is generated based on the combination of the device and type of browser used while accessing a web page. The following table shows examples of browser strings:

Browser	Device	User agent string
Firefox	Windows desktop	Mozilla/5.0 (Windows NT 5.1; rv:31.0) Gecko/20100101 Firefox/31.0
Chrome	OS X 10 desktop	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/29.0.1547.66 Safari/537.36
Opera	Windows desktop	Opera/9.80 (Windows NT 6.0) Presto/2.12.388 Version/12.14
Safari	OS X 10 desktop	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8) AppleWebKit/537.13+ (KHTML, like Gecko) Version/5.1.7 Safari/534.57.2
Internet Explorer	Windows desktop	Mozilla/5.0 (compatible; MSIE 10.6; Windows NT 6.1; Trident/5.0; InfoPath.2; SLCC1; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET CLR 2.0.50727) 3gpp-gba UNTRUSTED/1.0

AngularJS has features like providers or services which can be most useful for this browser user-agent sniffing and a redirection approach. An AngularJS provider can be created that can be used in the configuration in the routing module. This provider can have reusable properties and reusable methods that can be used to identify the device and route the specific request to the appropriate template view.

To discover more about user agent strings on various browser and device combinations, visit `http://www.useragentstring.com/pages/Browserlist/`.

CSS3 media queries approach

CSS3 brings a new horizon to web application development. One of the key features is **media queries** to develop a responsive web application. Media queries uses media types and features as deciding parameters to apply the style to the current web page.

Media type

CSS3 media queries provide rules for media types to have different styles applied to a web page. In the media queries specification, media types that should be supported by the implemented browser are listed. These media types are as follows:

- `all`: This is used for all media type devices
- `aural`: This is used for speech and sound synthesizers
- `braille`: This is used for braille tactile feedback devices
- `embossed`: This is used for paged braille printers
- `handheld`: This is used for small or handheld devices, for example, mobile
- `print`: This is used for printers, for example, an A4 size paper document
- `projection`: This is used for projection-based devices, such as a projector screen with a slide
- `screen`: This is used for computer screens, for example, desktop and laptop screens
- `tty`: This is used for media using a fixed-pitch character grid, such as teletypes and terminals
- `tv`: This is used for television-type devices, for example, webOS or Android-based television

A media rule can be declared using the `@media` keyword with the specific type for the targeted media. The following code shows an example of the media rule usage, where the background body color is black and text is white for the screen type media, and background body color is white and text is black for the printer media type:

```
@media screen {
  body {
    background:black;
    color:white;
  }
}

@media print{
  body {
    background:white;
    color:black;
  }
}
```

An external style sheet can be downloaded and applied to the current page based on the media type with the HTML link tag. The following code uses the link type in conjunction with media type:

```
<link rel='stylesheet' media='screen' href='<fileName.css>' />
```

To learn more about different media types,visit `https://developer.mozilla.org/en-US/docs/Web/CSS/@media#Media_types`.

Media feature

Conditional styles can be applied to a page based on different features of a device. The features that are supported by CSS3 media queries to apply styles are as follows:

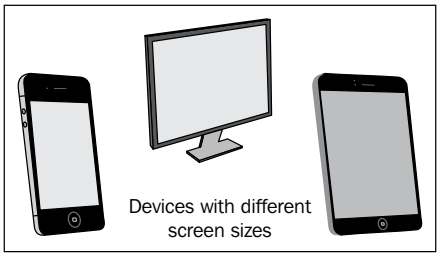
- `color`: Based on the number of bits used for a color component by the device-specific style sheet, this can be applied to a page.
- `color-index`: Based on the color look up, table styles can be applied to a page.
- `aspect-ratio`: Based on the aspect ratio, display area style sheets can be applied to a page.
- `device-aspect-ratio`: Based on the device aspect ratio, styles can be applied to a page.
- `device-height`: Based on device height, styles can be applied to a page. This includes the entire screen.
- `device-width`: Based on device width, styles can be applied to a page. This includes the entire screen.
- `grid`: Based on the device type, bitmap or grid, styles can be applied to a page.
- `height`: Based on the device rendering area height, styles can be used to a page.
- `monochrome`: Based on the monochrome type, styles can be applied. This represents the number of bits used by the device in the grey scale.
- orientation: Based on the viewport mode, landscape or portrait, styles can be applied to a page.

- `resolution`: Based on the pixel density, styles can be applied to a page.
- `scan`: Based on the scanning type used by the device for rendering, styles can be applied to a page.
- `width`: Based on the device screen width, specific styles can be applied.

The following code shows some examples of CSS3 media queries using different device features for conditional styles used:

```
//for screen devices with a minimum aspect ratio 0.5
@media screen and (min-aspect-ratio: 1/2)
{
  img
  {
    height: 70px;
    width: 70px;
  }
}
//for all device in portrait viewport
@media all and (orientation: portrait)
{
  img
  {
    height: 100px;
    width: 200px;
  }
}
//For printer devices with a minimum resolution of 300dpi pixel
density
@media print and (min-resolution: 300dpi)
{
  img
  {
    height: 600px;
    width: 400px;
  }
}
```

In this section, we will explore more about the screen width. We will explore how AngularJS can help you use CSS3 media queries for a responsive application development. There are huge number of devices with different screen sizes present on the market, so we will focus on the most commonly used devices. The following diagram shows the different screen sizes of most commonly used devices such as mobile, desktop, and tablet:



To learn more about different media features, visit `https://developer.mozilla.org/en-US/docs/Web/CSS/@media#Media_features`.

What are we building?

In the following chapters, we will build a small responsive SPA to understand the helpful features of AngularJS. The plan of action to develop this application is as follows:

- We are going to build a `My Portfolio` web application
- This application is going to have different sections such as personal details, academics, skills, and experiences
- We will maintain all the data in different JSON files and serve them with appropriate templates when the user clicks on these sections
- We will explore how AngularJS makes this application responsive using browser sniffing-redirection, media queries, and a directive-based approach

Summary

In this chapter, you learned about responsive design and the SPA architecture. You now understand the role of the AngularJS library when developing a responsive application. We quickly went through all the important features of AngularJS with the coded syntax. In the next chapter, you will set up your AngularJS application and learn to create dynamic routing-based on the devices.

2
The AngularJS Dynamic Routing-based Approach

In this chapter, you will learn about dynamic routing using user agent detection controlled by AngularJS. The key concepts of AngularJS that we will cover in this chapter are `ngRoute` and `provider` module. To learn these features you will configure a new project. This chapter deals with setting up a code base to explore these features.

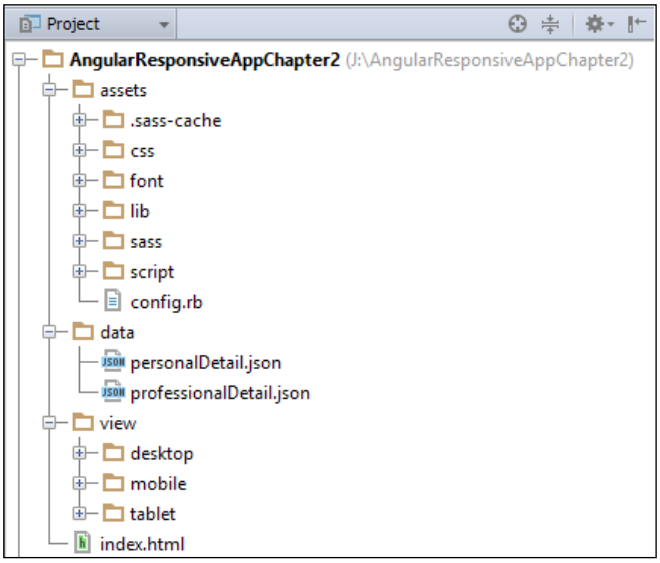
Setting up an AngularJS project

In this section, we will set up a new project and configure the development environment to develop a dynamic routing module using the AngularJS library. The following tools are required to build a development environment:

- **Code editor**: This editor is used to develop applications in WebStorm IDE from JetBrains for development purposes. However, you can use your own preferred editors such as Sublime Text, Notepad ++, and so on.
- **Web server**: A simple HTTP server such as WAMP stack can be used to host the project.

Building a project's structure

We need to create multiple directories to have related styles, fonts, markup and script files in the group. The following screenshot shows the directory structure of the project:



In the previous screenshot each directory has a specific type of files to be used by the application. The details of these directories are as follows:

- `assets`: This directory contains `sass`, `css`, `font`, `lib`, and `script` subdirectories and Ruby configuration files for SASS-based COMPASS watcher. The details of these directories are as follows:
    - `sass`: This subdirectory contains all the SASS-based SCSS files to generate styling in CSS files. These SCSS files are `desktop.scss`, `mobile.scss`, `tablet.scss`, and `icon-font.scss` files containing CSS for desktop, mobile, and tablet-specific styles. The `_icon-font.scss` file contains Mono Social Icons Font for various social media web pages.
    - `css`: This subdirectory is the target for all the CSS files generated by the SCSS file compilation. It contains the `desktop.css`, `mobile.css`, and `tablet.css` files generated from their respective SASS files.
    - `font`: This subdirectory contains the `MonoSocialIconsFont-1.10.ttf` file for social media icons.

You can find more details of these icons at `http://drinchev.github.io/monosocialiconsfont`.

- `lib`: This subdirectory contains all the library-related AngularJS frameworks. These files are `angular.min.js`, `angular-resource.min.js`, and `angular-route.min.js` containing code for AngularJS core, resource, and routing modules. This book has used AngularJS 1.3 version to demonstrate the coded examples. The AngularJS library file can be downloaded from `https://angularjs.org`.

We can also use the AngularJS library from Google's CDN. For more information about Google's CDN refer to the following link: `http://angularjs.blogspot.in/2012/07/angularjs-now-hosted-on-google-cdn.html`.

    - `script`: This subdirectory contains user defined script files related to the application. One of the important script files inside this directory is `app.js`, which contains the entry point to the AngularJS-based application. We will explore this file in the coming sections.
    - `config.rb`: This file contains the COMPASS watcher configuration for the SCSS file compilation.

- `data`: This directory contains all the related JSON data that we will use in our application development. It contains `personalDetail.json` and `professionalDetail.json` files containing profile-related data. Content of these files are as follows:
    - `personalDetail.json`: This JSON file contains personal details in key-value pairs. The following code shows the content of this file:

```
{
    "name":"Sandeep Kumar Patel",
    "spouse":"Surabhi Patel",
    "father":"Dilip Kumar Patel",
    "mother":"Sanjukta Patel",
    "bloodGroup":"O+ve",
    "height":"5.7 feet",
    "weight":"68 kg",
    "chest":"40 inch",
    "address":"House no 8 Marathalli bangalore 69"
}
```

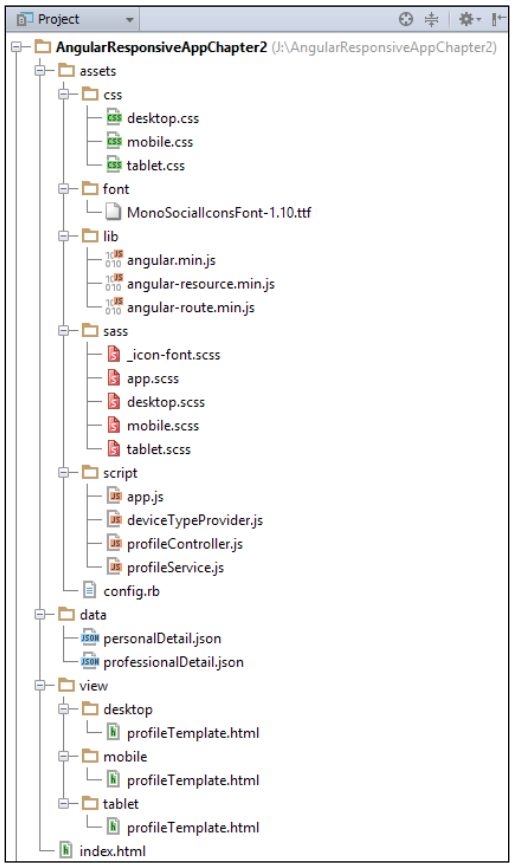
- `professionalDetail.json`: This JSON file contains professional details in key-value pair. The following code shows the content of this file:

```
{
   "aboutme":"I, Sandeep Kumar Patel, am a web developer
and blogger living in Bangalore India. I have many years
of experience in creating web applications both using
custom and popular JavaScript libraries and frameworks
such as AngularJS and YUI.",
   "years":4,
   "roles":[
      "Web Developer",
      "JavaScript Developer",
      "Front End Lead"
   ],
   "languages":[
      "JavaScript",
      "Java"
   ],
   "webdevelopment":[
      "HTML5",
      "CSS3",
      "AngularJS",
      "Jquery",
      "BootStrap",
      "YUI"
   ],
   "tools":[
      "SASS",
      "COMPASS",
      "GRUNT",
      "GIT"
   ],
   "ides":[
      "WebStorm",
      "Intelij Idea",
      "Eclipse"
   ],
   "social":{
      "linkedin":"http://www.linkedin.com/in/
techblogger",
      "facebook":"http://www.facebook.com/
SandeepTechTutorials",
      "twitter":"http://twitter.com/MySmallTutorial",
      "googleplus":"https://plus.google.com/
u/0/+SandeepKumarPatel/"
   },
   "contact":{
      "email":"sandeeppateltech@gmail.com",
```

```
            "mobile":"+91-8105469950",
            "address":"Whitefield Bangalore 560087"
        }
    }
```

- `view`: This directory is the parent of three different subdirectories containing all the HTML templates. The details of these subdirectories are as follows:
    - `desktop`: This subdirectory contains the `profileTemplate.html` file that has all the HTML code for desktop type devices
    - `tablet`: This subdirectory contains the `profileTemplate.html` file that has all the HTML code for tablet type devices
    - `mobile`: This subdirectory contains the `profileTemplate.html` file that has all the HTML code for desktop type devices

The following screenshot shows the full project structure including subdirectories and code files:



The SASS configuration

SASS (**Syntactically Awesome Style Sheets**) is a CSS authoring language. It provides more control over writing CSS. SASS provides many features such as variables, mixins, and others, which help in the organization of large code style sheets.

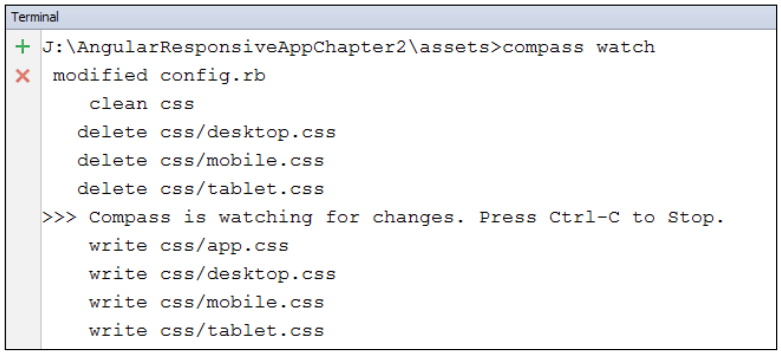
To know more about SASS framework refer to `http://sass-lang.com/documentation/file.SASS_REFERENCE.html`.

To author CSS style sheets we have used a SASS-based COMPASS file watcher to convert SCSS files to CSS files. To install COMPASS, we need to have Ruby installed in our system. A configuration file named `config.rb` needs to be added inside the `assets` folder before starting the COMPASS watcher. This configuration file provides the source and target of the SASS files from where COMPASS watcher will read the SCSS code and convert them into CSS files. The content of the `config.rb` file is as follows:

```
require 'compass/import-once/activate'
#root of your project when deployed:
http_path = "/"
css_dir = "css"
sass_dir = "sass"
```

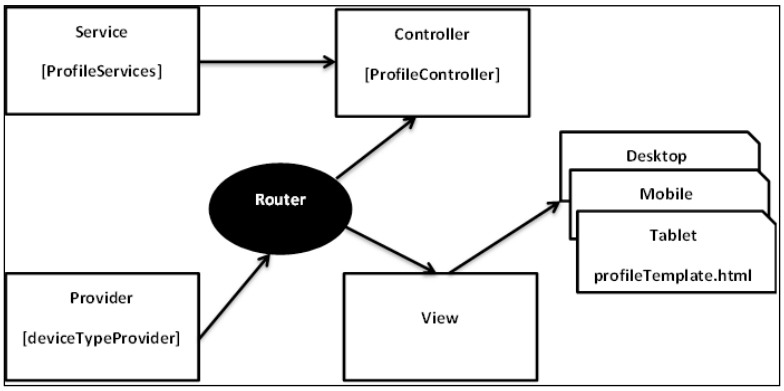
To know more about the installation of COMPASS refer to `http://compass-style.org/install`.

After a successful installation, COMPASS watcher can be used to monitor the `assets` directory. The watcher looks for any changes in SCSS files and moves the changes to their respective files inside the CSS directory. The following screenshot shows the command issued to start the COMPASS watcher:



Profile application demo

We are going to build a profile page application. This application will show two different sections of personal and professional data. The following diagram shows the architectural structure of this application:



The previous diagram shows the SPA architecture for our demo profile application. **Router** is the main module that chooses the device type using `deviceTypeProvider`. Once the device type is identified, **Router** chooses the HTML template file and gives control to `ProfileController`, which loads the JSON data. `ProfileController` links these JSON data with the HTML template and renders it in the browser.

Building data services

To build our profile application, we have maintained two different JSON files, `personalDetail.json` and `professionalDetail.json`, in the data directory. An AngularJS-based service is created using the `ngResource` module. The name of the data service is `ProfileServices` and is maintained in the `profileService.js` script file. This service is packaged inside the `profileApp.profileServices` namespace and injected into the application module. The content of `ProfileServices` is as follows:

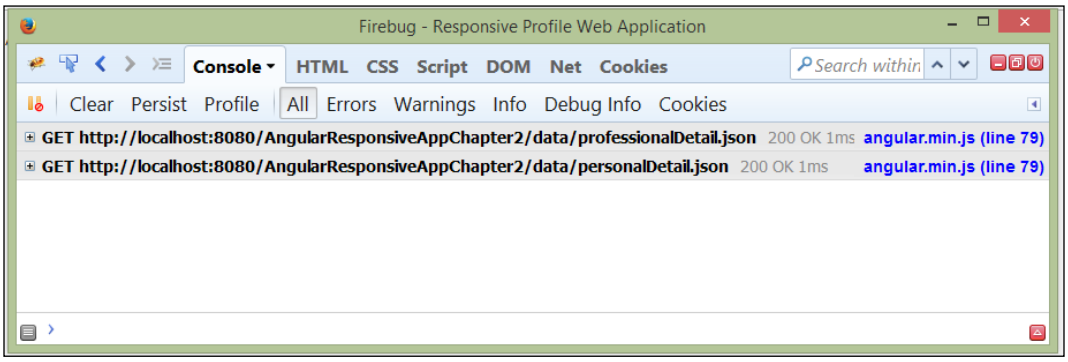
```
angular.module('profileApp.profileServices', [])
.service('ProfileServices',["$resource", function($resource){
   return{
/*Method for getting personal detail JSON file*/
       getPersonalDetail : function(){
           return $resource("data/personalDetail.json")
       },
/*Method for getting professional detail JSON file*/
       getProfessionalDetail : function(){
```

```
            return $resource("data/professionalDetail.json")
        }`
    }
}]);
```

In the previous code, `ProfileServices` has two methods to access the profile data. The details of these methods are as follows:

- `getPersonalDetail()`: This method returns personal details in the JSON format. It uses the `$resource` function with the data URL pointing to the `personalDetail.json` file.
- `getProfessionalDetail()`: This method returns professional details in the JSON format.

While the application is loading, these two services get called by the controller. The following screenshot shows the firebug console loading these resources through an AJAX call:



Device-based routing

In the device-based routing approach, the requesting devices are identified by the application and client is redirected to the appropriate HTML template. In this approach, different HTML templates are maintained by the application and served to the client. To identify a device the user agent string can be used to detect `deviceType`. In this chapter, we have used the approach present at `http://detectmobilebrowsers.com`. It maintains a set of user agent strings for various mobile devices and checks the incoming request against these strings through JavaScript's regular expression.

AngularJS routing

AngularJS provides a routing module to cater to an incoming request for an appropriate template and controller pair. The AngularJS routing module code can be found in the `angular-routr.min.js` file and named as `ngRoute` package. Like an SPA, the routing logic is present entirely in the browser's side. To create dynamic routing, an AngularJS application module needs to perform the following steps:

1. Set up an AngularJS application.
2. Configure a routing module.

Setting up an AngularJS application

An AngularJS application module can be initialized using the `ng-app` built-in directive in the HTML markup. In our demo application, we have used this attribute in the index page. The following code shows the content of `index.html` file:

```
<!DOCTYPE html>
<html  lang="en" ng-app="profileApp">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1, maximum-scale=1, user-scalable=YES">
        <title>
            Responsive Profile Web Application
        </title>
        <link href='http://fonts.googleapis.com/css?family=Robo
to:400,100,300,500,700' rel='stylesheet' type='text/css'>
        <link ng-if="styleType.length > 0"
              ng-href='assets/css/{{styleType}}.css'
              rel='stylesheet' type='text/css' >
    </head>
    <body>
        <!--AngularJS view -->
        <div ng-view="">
        </div>
        <!--external scripts start -->
        <script  src="assets/lib/angular.min.js"></script>
        <script  src="assets/lib/angular-route.min.js"></script>
        <script  src="assets/lib/angular-resource.min.js"></script>
        <script  src="assets/script/deviceTypeProvider.js"></script>
        <script  src="assets/script/profileController.js"></script>
        <script  src="assets/script/profileService.js"></script>
        <script  src="assets/script/app.js"></script>
```

```
        <!--external scripts end -->
    </body>
</html>
```

In the previous code, the AngularJS application is initialized with the module name `profileApp` using the `ng-app` directive. This directive is the root element of the AngularJS application and is normally used in the `body` or `HTML` root element. A few key points about the `ng-app` directive are as follows:

- This directive is used to autobootstrap the AngularJS application.
- This directive runs with priority 0, which means it runs with the lowest priority.
- Multiple `ng-app` directives with autobootstrap is not possible. To use multiple bootstrapping, the `angular.bootstrp()` method can be used.
- Nested `ng-app` directive declaration is not permissible.

To know more about AngularJS application bootstrapping refer to `https://docs.angularjs.org/api/ng/directive/ngApp`.

Configuring a routing module

After the application is set up, a routing module can be configured using `$routeProvider` present inside the `ngRoute` module. This `ngRoute` module can be used by including the `angular-route.min.js` file. The following code shows the content of routing logic for the profile application:

```
'use strict';
angular.module('profileApp',[
    'ngRoute',
    'ngResource',
    'profileApp.profileController',
    'profileApp.profileServices',
    'profileApp.deviceTypeProvider'
])
.config(['$routeProvider','deviceTypeProvider',
   function($routeProvider,deviceTypeProvider) {
     var deviceTypeProvider = deviceTypeProvider.$get(),
         deviceType = deviceTypeProvider.getDeviceType();
     /*Route to Desktop view*/
```

```
    $routeProvider.when('/',{
           templateUrl:   'view/'+deviceType+'/profileTemplate.html',
           controller: 'ProfileController',
           styleType:deviceType
       });
}]);
```

In the previous code, the routing module is injected with a device type provider. The routing module calls the `getDeviceType()` method to retrieve the type of device in string format. We will have a look at the detailed implementation of this provider in the following section. The `$routeProvider` module uses the `when` clause to route the request to the appropriate template view. Different values of device type and template path are listed in the following table:

Device type	Template path	Detail
Desktop	`desktop/profileTemplate.html`	When `deviceType` is detected as desktop, the profile template present inside the desktop directory is served to the client
Mobile	`mobile/profileTemplate.html`	When `deviceType` is detected as mobile, the profile template present inside the mobile directory is served to the client
Tablet	`tablet/profileTemplate.html`	When `deviceType` is detected as tablet, the profile template present inside the tablet directory is served to the client

The controller inside the routing module then sets a global scope variable named `styleType` with the device type string. This `styleType` controller is shared in the root scope of the application and is used in the `index.html` file to load the appropriate style sheet in the CSS format. Different possible style sheets `desktop.css`, `mobile.css`, and `tablet.css` can be loaded based on the device type string. The following code shows the external style sheet pointing to different CSS files:

```
<link ng-if="styleType.length > 0"
ng-href='assets/css/{{styleType}}.css'
   rel='stylesheet' type='text/css'>
```

Configuring a profile controller

Each template is under the scope of `ProfileContoller`. This controller is injected with `ProfileServices`. Using these services, it loads the personal and professional data to its scope variables `$scope.personal` and `$scope.professional`. This controller calls the `get()` method on the resource instance of personal and professional data service. The code for this controller is as follows:

```
'use strict';
angular.module("profileApp.profileController", [])
.controller('ProfileController',
function ($scope, $rootScope, $route, ProfileServices,$log) {
var professionalDetail =   ProfileServices.getProfessionalDetail(),
    personalDetail = ProfileServices.getPersonalDetail();
    $rootScope.styleType = $route.current.styleType;
    $scope.professional = {};
    $scope.personal = {};
    //Default menu button selected to true
    $scope.selected = true;
    /*Calls the Angular service to load professional JSON data*/
    professionalDetail.get(function(jsonData){
       $scope.professional = jsonData;
    });
    /*Calls the Angular service to load personal JSON data*/
    personalDetail.get(function(jsonData){
       $scope.personal = jsonData;
    });
    /*Method to change the user selection*/
    $scope.getDetail=function(event){
      $scope.selected = !$scope.selected;
    };
})
```

To hide and show the personal and professional data the `$scope.selected` variable is used. The `$scope.getDetail()` method is used to change the value of this variable by altering its true value.

Developing a device type provider

A device type detector is developed as a provider to inject to the routing configuration. The `getDeviceType()` method is used to return the type of device. This methods reads and uses the window object provided by AngularJS `$windowProvider`.

This provider reads the user agent string and runs a JavaScript pattern to match it with the list of smart device names. For our profile application, we have created a small list of smart device types such as iPhone, iPad, Android, and so on. You can add more device names to this list. Also, we have considered desktop, mobile, and tablet device types. The default device is taken as desktop. The code for the `deviceType` provider is as follows:

```
'use strict';
angular.module("profileApp.deviceTypeProvider",[])
.provider('deviceType', ['$windowProvider', function($windowProvider)
{
    var $window = $windowProvider.$get();
    this.$get = function() {
       return{
           /*Returns the device type desktop, mobile and tablet,
default device type is desktop*/
           getDeviceType:function(){
//Let, default device type
var deviceType='desktop',
userAgentString = $window['navigator']['userAgent']
||$window['navigator']['vendor'] ||$window['opera'],

width = $window['outerWidth'],isSmart = (/iPhone|iPod|iPad|Silk|Androi
d|BlackBerry|Opera Mini|IEMobile/).test(userAgentString);
                    if(isSmart&& width >= 768){
                        deviceType = "tablet";
                    }else if(isSmart&& width <= 767){
                        deviceType = "mobile";
                    }
                return deviceType;
           }
       }
    };
}])
```

The method of detecting `deviceType` is not a foolproof solution. For demonstration purposes, we have categorized the devices by considering the following points:

- **Smart device**: If the specified user agent string passes the JavaScript pattern match against the list of device names, then it is considered as a smart device. The pattern match is performed using the `test()` method provided by the JavaScript regular expression object. This method returns a Boolean value. For a successful match, it returns the `isSmart` variable as true. Again, the smart device can be either a tablet or a mobile. The following list shows the categorization among the smart devices:
    - **Tablet**: A device is identified as a tablet if it is smart and its width is more than 768 px. The screen width can be calculated using the `outerWidth` property of the `$window` object. In the previous code, `$window['outerWidth']` determines the screen width.
    - **Mobile**: A device is identified as mobile if it is smart and its width is less than 767 px.
- **Desktop**: If the device is not smart, that is, the Smart flag is false, then it is a desktop device.

Developing a desktop view

To implement the redirection approach, we need to create three different HTML files for desktop, mobile, and tablet. In all three HTML templates, the elements are all similar except for their placement and width, which will be different for desktop, mobile, and tablet devices. The HTML template for desktop type devices are divided into four rows:

- Name and profile image row
- Category selection row
- Category content
- Social buttons row

Name and profile image row

This portion of the HTML markup contains the name and image of the candidate wrapped by the `.row` class. The `h1` element is used to represent the name and an `img` element points to the Gravatar profile image URL. The following code shows the HTML markup for this section:

```
<div class="row">
    <div class="pic">
        <h1 class="name">Sandeep Kumar Patel</h1>
        <img class="profile-image"
          src="http://www.gravatar.com/avatar/4205261332ff1
          31e971b48db31dcb528.png"
          alt="profile image"/>
    </div>
</div>
```

Category selection row

This section of the HTML markup contains two buttons for category selection. For our profile application, we have personal and professional categories. Each of these buttons are attached to angular attribute directives; details of these are as follows:

- `ng-click`: This attribute is attached with a callback function and is triggered on click event.
- `ng-class`: This attribute is used for styling the selected button. A selected class is attached when the `selected` scope property is set to true.
- `ng-disabled`: This attribute is for the disabling and enabling button. This attribute also takes the Boolean value from the `selected` scope property.

The following code shows these buttons for category selection with the attribute directive attached to it:

```
<div class="row">
    <div class="about">
        <button class="btn" ng-model="professional"
                ng-click="getDetail($event)"
                ng-class="{'selected':selected}"
                ng-disabled="selected">
            Professional
        </button>
        <button class="btn" ng-model="personal"
                ng-click="getDetail($event)"
                ng-disabled="!selected"
                ng-class="{'selected':!selected}">
            Personal
        </button>
    </div>
</div>
```

Category content

This section contains the HTML markup for the personal and professional categories. The category content is wrapped around the display class element. The following code shows the HTML content for the professional category:

```
<div class="detailProfessional" ng-show="selected">
    <p class="aboutme">
        {{professional.aboutme}}
    </p>
    <div class="section">
        <div class="divider">
            <h4>No of Year Experience</h4>
            {{professional.years}}
            <h4>Job Roles</h4>
            <ol>
                <li ng-repeat="role in
                professional.roles">{{role}}</li>
            </ol>
            <h4>Languages Known</h4>
            <ol>
                <li ng-repeat="language in
                professional.languages">{{language}}</li>
            </ol>
            <h4> Tools</h4>
            <ol>
                <li ng-repeat="tool in
                professional.tools">{{tool}}</li>
            </ol>
        </div>
        <div class="divider">
            <h4>Web Technologies</h4>
            <ol>
                <li ng-repeat="webdev in
                professional.webdevelopment">{{webdev}}</li>
            </ol>
        </div>
    </div>
</div>
```

The HTML content of the personal category is rendered in the browser when the personal category button is selected by the user. The following code shows the HTML content of the personal category:

```
<div class="detailPersonal" ng-show="!selected">
    <div class="section">
        <div class="divider">
            <h4>Full Name </h4> {{personal.name}}
```

```
            <h4>Spouse Name </h4> {{personal.spouse}}
            <h4>Father Name </h4> {{personal.father}}
            <h4>Mother Name </h4> {{personal.mother}}
            <h4>Blood Group </h4> {{personal.bloodGroup}}
            <h4>Height </h4> {{personal.height}}
        </div>
        <div class="divider">
            <h4>Weight </h4> {{personal.weight}}
            <h4>Chest </h4> {{personal.chest}}
            <h4>Address </h4> {{personal.address}}
        </div>
    </div>
</div>
```

Social buttons row

This section has the HTML markup to show social links. When the user selects these links, the application navigates to the targeted social page of the mentioned link. The following code has the HTML markup for these social links:

```
<div class="footer">
    <a ng-href='{{professional.social.twitter}}' class='symbol'
title='&#xe286;'></a>
    <a ng-href='{{professional.social.facebook}}' class='symbol'
title='&#xe227;'></a>
    <a ng-href='{{professional.social.linkedin}}' class='symbol'
title='&#xe252;'></a>
    <a ng-href='{{professional.social.googleplus}}' class='symbol'
title='&#xe239;'></a>
</div>
```

The complete HTML code is present inside the `profileTemplate.html` file and can be downloaded from the Packt Publishing support website.

We have created different SCSS files for each device. For social media icons, we have used the `icon-font.scss` file that is used by all three types of devices. The code for this `icon-font.scss` file is as follows:

```
@font-face {
  font-family: 'Mono Social Icons Font';
  src: url('../font/MonoSocialIconsFont-1.10.ttf') format('truetype');
  font-weight: normal;
  font-style: normal;
}
.symbol, a.symbol:before {
  font-family: 'Mono Social Icons Font';
  -webkit-text-rendering: optimizeLegibility;
```

```
    -moz-text-rendering: optimizeLegibility;
    -ms-text-rendering: optimizeLegibility;
    -o-text-rendering: optimizeLegibility;
    text-rendering: optimizeLegibility;
    -webkit-font-smoothing: antialiased;
    -moz-font-smoothing: antialiased;
    -ms-font-smoothing: antialiased;
    -o-font-smoothing: antialiased;
    font-smoothing: antialiased;
  }
  a.symbol:before {
    content: attr(title);
    margin-right: 0.3em;
    font-size: 130%;
  }
  .symbol, a.symbol:before {
    font-family: 'Mono Social Icons Font';
    -webkit-text-rendering: optimizeLegibility;
    -moz-text-rendering: optimizeLegibility;
    -ms-text-rendering: optimizeLegibility;
    -o-text-rendering: optimizeLegibility;
    text-rendering: optimizeLegibility;
    -webkit-font-smoothing: antialiased;
    -moz-font-smoothing: antialiased;
    -ms-font-smoothing: antialiased;
    -o-font-smoothing: antialiased;
    font-smoothing: antialiased;
  }
  a.symbol:before {
    content: attr(title);
    margin-right: 0.3em;
    font-size: 130%;
  }
```

The preceding social media font file is imported using the `@import` statement inside the SCSS file. The SCSS file for the desktop device is created inside the `desktop.scss` file. The following code shows the `desktop.scss` file for desktop type devices:

```
  @import "icon-font";
  body {
    background: #eee;
    font-family: 'Roboto', sans-serif;
    font-size: 20px;
    font-weight: 300;
    .my-profile-container {
```

```
background: white;
width: 1024px;
margin: auto;
box-shadow: 2px 2px 2px 2px lightgrey;
.row {
  .pic {
    height: 300px;
    text-align: center;
    .name {
      font-weight: 100;
      text-align: center;
      background: #eee;
      color: green;
    }
    .profile-image {
      border-radius: 100%;
      margin: 70px 0px 50px;
    }
  }
  .about {
    display: flex;
    .btn {
      border: 0 none;
      color: green;
      height: 60px;
      width: 50%;
      margin: 5px 0px;
      font-weight: 100;
      font-size: 16px;
      cursor: pointer;
      &:hover {
        background: green;
        color: #eee;
      }
      &.selected {
        background: green;
        color: #eee;
        cursor: text;
      }
    }
  }
  .display {
    height: 100%;
    padding: 0px 30px;
```

```
            .detailProfessional, .detailPersonal {
              .section {
                display: flex;
                .divider {
                  width: 500px;
                  h4 {
                    font-weight: 300;
                    color: green;
                  }
                }
              }
              position: relative;
              .aboutme {
                &:first-letter {
                  font-size: 25px;
                  color: green;
                }
              }
            }
          }
          .footer {
            background: #eee;
            text-align: center;
            a {
              color: green;
              text-decoration: none;
              font-weight: 300%;
            }
          }
        }
      }
    }
```

Developing a mobile view

For mobile type devices, we have created a different version of the `profileTemplate.html` file. The HTML template code for mobile view is almost the same; the only difference is the order of these sections. The HTML template code has four rows similar to the desktop view with a given order. The order of these sections is as follows:

- Name and profile image row
- Social buttons row

- Category content
- Category selection row

You can download the complete code for the mobile view from the Packt Publishing support website.

For mobile devices the SCSS file is created under the `mobile.scss` file. The following code shows the content of the `mobile.scss` file:

```
@import "icon-font";
body {
  background: #eee;
  font-family: 'Roboto', sans-serif;
  font-size: 15px;
  font-weight: 400;
  overflow:hidden;
  .my-profile-container {
    background: white;
    width: 300px;
    box-shadow: 2px 2px 2px 2px lightgrey;
    margin: auto;
    .row {
      height: 100%;
      .pic {
        height: 120px;
        text-align: center;
        .name {
          font-weight: 300;
          text-align: center;
          background: #eee;
          color: green;
        }
        .profile-image {
          border-radius: 100%;
        }
      }
      .about {
        display: flex;
        .btn {
          border: 0 none;
          color: green;
          height: 60px;
          width: 50%;
          font-weight: 100;
```

```
      font-size: 16px;
      cursor: pointer;
      &:hover {
        background: green;
        color: #eee;
      }
      &.selected {
        background: green;
        color: #eee;
        cursor: text;
      }
    }
  }
  .display {
    height: 100%;
    padding: 0px 15px;
    .detailProfessional, .detailPersonal {
      .section {
        display: flex;
        .divider {
          width: 50%;
          .text{
            font-size: 15px;
          }
          h4 {
            font-weight: 300;
            color: green;
            margin: 10px 10px 2px 0;
          }
        }
      }
      position: relative;
      .aboutme {
        &:first-letter {
          font-size: 25px;
          color: green;
        }
      }
    }
  }
  .footer {
    background: #eee;
    text-align: center;
    a {
```

```
            color: green;
            text-decoration: none;
            font-weight: 200%;
          }
        }
      }
    }
  }
```

Developing a tablet view

For tablet type devices, a different version of the HTML template, which is a `profileTemplate.html` file, is created inside the `tablet` directory. The HTML markup for the desktop template is as follows:

```
<div class="my-profile-container">
    <div class="row">
        <div class="pic">
            <h3 class="name">Sandeep Kumar Patel</h3>
            <div class="row">
                <div class="about">
                    <button class="btn" ng-model="professional"
                            ng-click="getDetail($event)"
                            ng-class="{'selected':selected}"
                            ng-disabled="selected">
                        Professional
                    </button>
                    <button class="btn" ng-model="personal"
                            ng-click="getDetail($event)"
                            ng-disabled="!selected"
                            ng-class="{'selected':!selected}">
                        Personal
                    </button>
                </div>
            </div>
            <img class="profile-image"
                 src="http://www.gravatar.com/avatar/4205261332ff131e9
71b48db31dcb528.png" alt="profile image"/>
        </div>
    </div>
    <div class="row">
        <div class="footer">
            <a ng-href='{{professional.social.twitter}}'
class='symbol' title='&#xe286;'></a>
```

```
            <a ng-href='{{professional.social.facebook}}'
class='symbol' title='&#xe227;'></a>
            <a ng-href='{{professional.social.linkedin}}'
class='symbol' title='&#xe252;'></a>
            <a ng-href='{{professional.social.googleplus}}'
class='symbol' title='&#xe239;'></a>
        </div>
    </div>
    <div class="row">
        <div class="display">
            <div class="detailProfessional" ng-show="selected">
                <p class="aboutme text">
                    {{professional.aboutme}}
                </p>
                <div class="section">
                    <div class="divider">
                        <h4>Years of Experience</h4>
                        <span class="text">{{professional.years}}</
span>
                        <h4>Job Roles</h4>
                        <span class="text" ng-repeat="role in
professional.roles">
                            <em ng-if="!first"> </em> {{role}}
                        </span>
                        <h4>Languages Known</h4>
                        <span class="text" ng-repeat="language in
professional.languages">
                            <em ng-if="!first"> </em>{{language}}
                        </span>
                    </div>
                    <div class="divider">
                        <h4>Web Technologies</h4>
                        <span class="text" ng-repeat="webdev in
professional.webdevelopment">
                            <em ng-if="!first"> </em>{{webdev}}
                        </span>
                        <h4> Tools</h4>
                        <span class="text" ng-repeat="tool in
professional.tools">
                            <em ng-if="!first"> </em>{{tool}}
                        </span>
                    </div>
                </div>
            </div>
            <div class="detailPersonal" ng-show="!selected">
                <div class="section">
                    <div class="divider">
                        <h4>Full Name </h4>
                        <span class="text">{{personal.name}}</span>
                        <h4>Spouse Name </h4>
```

```
                        <span class="text">{{personal.spouse}}</span>
                        <h4>Father Name </h4>
                        <span class="text">{{personal.father}}</span>
                        <h4>Mother Name </h4>
                        <span class="text">{{personal.mother}}</span>
                        <h4>Blood Group </h4>
                        <span class="text">{{personal.bloodGroup}}</
span>
                        <h4>Height </h4>
                        <span class="text">{{personal.height}}</span>
                    </div>
                    <div class="divider">
                        <h4>Weight </h4>
                        <span class="text">{{personal.weight}}</span>
                        <h4>Chest </h4>
                        <span class="text">{{personal.chest}}</span>
                        <h4>Address </h4>
                        <span class="text">{{personal.address}}</span>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

For tablet type devices, the `tablet.scss` file is created. The following code shows the content of the `tablet.scss` file:

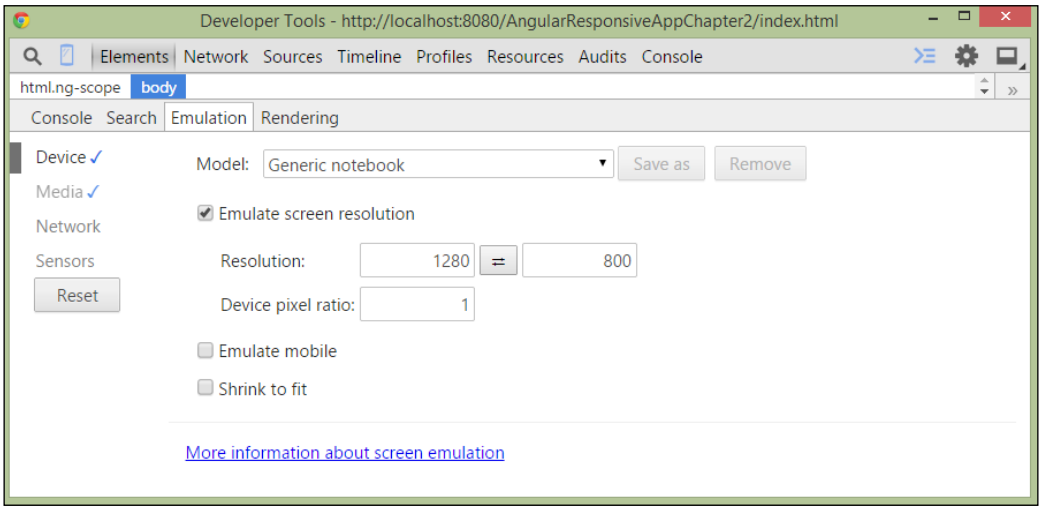
```
@import "icon-font";
body {
  background: #eee;
  font-family: 'Roboto', sans-serif;
  font-size: 15px;
  font-weight: 400;
  overflow:hidden;
  .my-profile-container {
    background: white;
    width: 768px;
    box-shadow: 2px 2px 2px 2px lightgrey;
    margin: auto;
    .row {
      height: 100%;
      .pic {
        text-align: center;
        .name {
          font-weight: 300;
          text-align: center;
          background: #eee;
          color: green;
          margin-bottom: 0px;
```

```
    }
    .profile-image {
      border-radius: 100%;
    }
  }
  .about {
    display: flex;
    .btn {
      border: 0 none;
      color: green;
      height: 60px;
      width: 50%;
      font-weight: 100;
      font-size: 16px;
      cursor: pointer;
      &:hover {
        background: green;
        color: #eee;
      }
      &.selected {
        background: green;
        color: #eee;
        cursor: text;
      }

    }
  }
  .display {
    height: 100%;
    padding: 0px 15px;
    .detailProfessional, .detailPersonal {
      .section {
        display: flex;
        .divider {
          width: 50%;
          .text{
            font-size: 15px;
          }
          h4 {
            font-weight: 300;
            color: green;
            margin: 10px 10px 2px 0;
          }
        }
      }

      position: relative;
      .aboutme {
        &:first-letter {
          font-size: 25px;
```

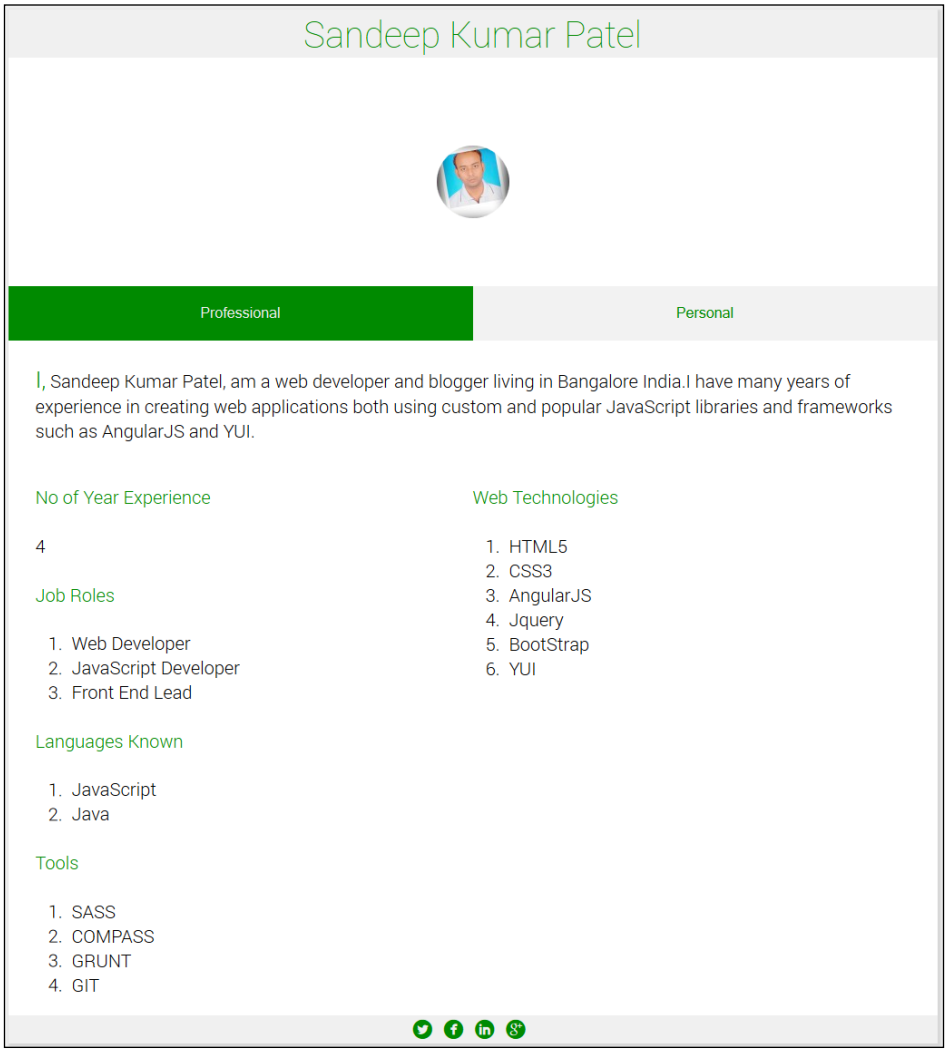
```
                color: green;
              }
            }
          }
        }
        .footer {
          background: #eee;
          text-align: center;
          a {
            color: green;
            text-decoration: none;
            font-weight: 200%;
          }
        }
      }
    }
  }
```

Verifying responsiveness

In this section, we have used the Chrome browser and the developer console to simulate different device user agent strings for desktop, mobile, and tablet devices. The following screenshot shows the Chrome console emulator for different devices:

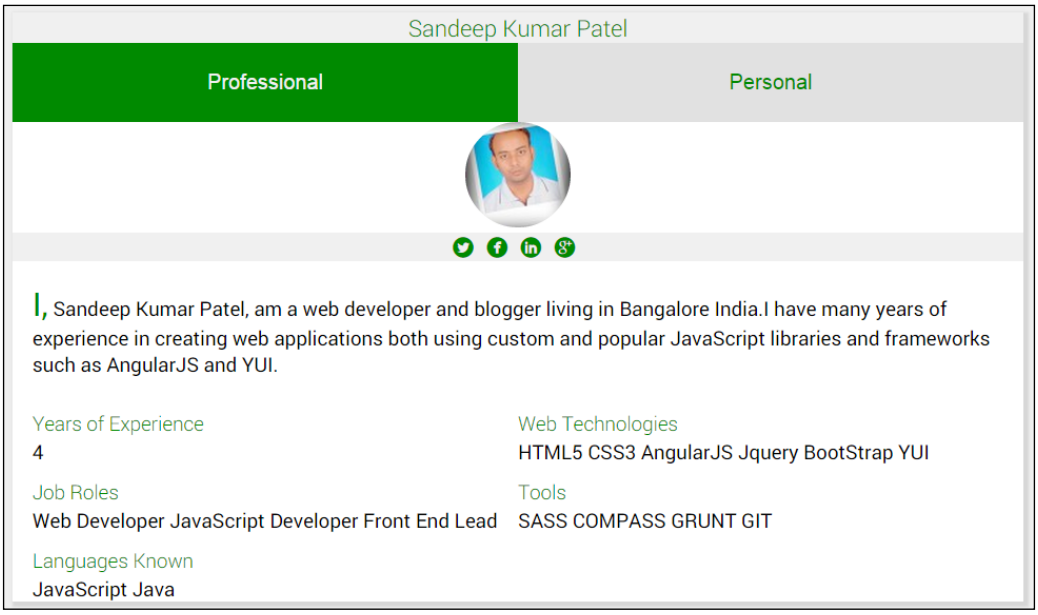


You can change the model dropdown to different devices that have been emulated such as Samsung Galaxy, Apple iPhone, and so on. The following screenshot shows the desktop view of the profile page for general notebooks:



The following screenshot shows the mobile view of the application, where the model name is chosen as Apple iPhone 5:

The following screenshot shows the tablet view of the profile; the model name is iPad mini device:



Limitations of dynamic routing

Using a dynamically based routing approach is a straightforward solution for responsive application development. However, this approach leads to various limitations and challenges for application maintenance. These limitations are as follows:

- Maintaining the user agent string list
- Maintaining the device type list
- Maintaining the template list

Summary

In this chapter, you learned how to set up an AngularJS project, a routing module, and a device detection provider followed by building three different HTML template versions for desktop, tablet, and mobile devices. Towards the end of the chapter, you learned the limitations of the dynamic routing approach. In the next chapter, you will learn about the directive-based approach for responsive web application development.

3
The AngularJS Directive-based Approach

In this chapter, you will learn about AngularJS directive concepts such as built-in and custom directives and how to develop them. We will explore how to use the power of these directives to leverage responsiveness in our application.

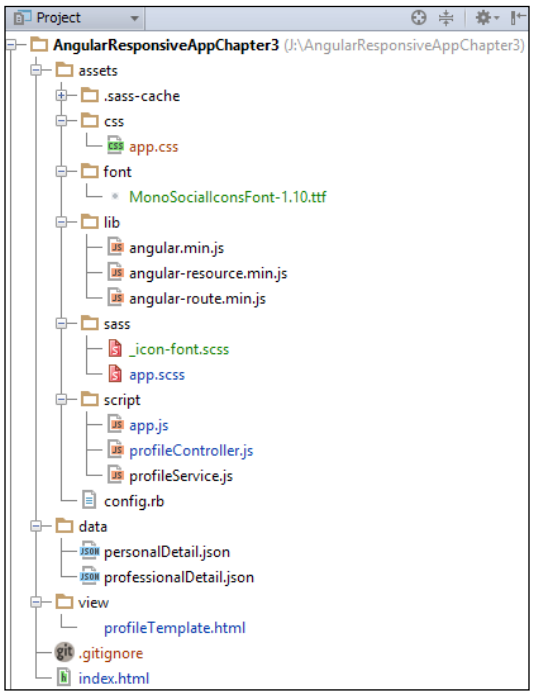
Modifying the project structure

In the previous chapter, we looked at different HTML markup for different devices. However, in this chapter, we are going to explore the directive approach for which we have to change the structure of the project. The changes are as follows:

- Changes in the directory structure
- Changes in the routing module
- Changes in the profile template

Changes in the directory structure

Most of the directory structure is the same as the previous chapter except that we don't have a different directory for HTML markup or SCSS files. We have removed the `desktop`, `tablet`, and `mobile` subdirectories from the project structure. The following screenshot shows the modified directory structure for this chapter:



Changes in the routing module

In this chapter, we will have a single routing destination for all devices. There are no more device-wise routing destinations. The following code shows the modified `app.js` file and the changed routing code:

```
'use strict';
angular.module('profileApp',[
    'ngRoute',
    'ngResource',
    'profileApp.profileController',
    'profileApp.profileServices'
])
.config(['$routeProvider',
    function($routeProvider) {
```

```
    $routeProvider.when('/',{
        templateUrl: 'view/profileTemplate.html',
        controller: 'ProfileController'
     });
}]);
```

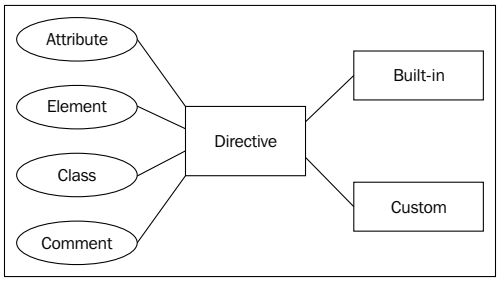
Changes in the profile template

In the previous chapter, we maintained different `profileTemplate.html` files for different devices. In this chapter, we are going to use a single HTML template file `profileTemplate.html` and SCSS file `app.scss`, which produces the `app.css` file.

Directives allow us to cut down on the complexity of multiple files and use a single template. This results in less code and better code maintenance.

Directives

Directives are the building blocks of AngularJS framework. They work as a marker and are used by web developers in different places of application code. When the AngularJS framework loads, it reads these directives and takes appropriate actions. A directive can be used as an **Attribute** (**A**), an **Element** (**E**), a **Class**(**C**), and a **Comment** (**M**) inside DOM. The following diagram shows the element and types of directives:



Before going into the implementation of directives for responsive web application, let's discuss a few helpful concepts and features of AngularJS. These new concepts are really useful in responsive application development, as follows:

- The `$window` service
- The `$watch` method
- The event binding function
- The `$log` service

The $window service

AngularJS provides the `$window` service inside the `ng` module. As you know, JavaScript also provides the `$window` object, which contains information about the browser. In the AngularJS framework, the same window object is served with a `$window` wrapper. This extra layer of wrapper by AngularJS helps to control the `$window` object inside a scope by suppressing its global nature. This helps when overloading and mocking the window object.

The `$window` object has many useful properties that can be used for application development. It is worth discussing the properties as follows:

- `outerWidth`: This property returns the width of the window, including all the interface elements such as the toolbar and scrollbar. When this property is used in conjunction with a DOM element, it returns the width, including the padding, border, and optionally, the margin. This property is read-only and does not have any default value. This property can be accessed by using the following code:

  ```
  $window.outerWidth
  ```

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. 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.

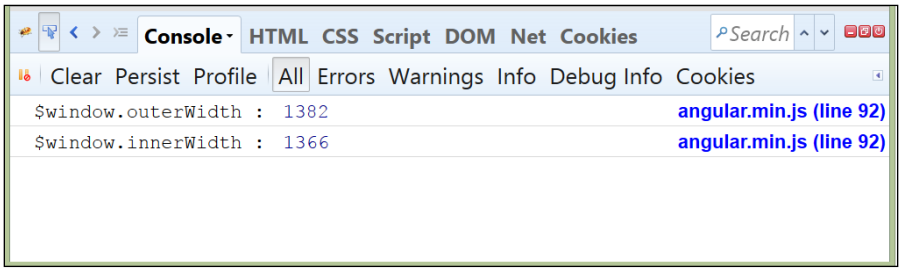
To find out more about the `outerWidth` property, check out `https://developer.mozilla.org/en-US/docs/Web/API/Window.outerWidth`.

- `innerWidth`: This property returns the width of the window excluding interface elements such as the toolbar and scrollbar. When this property is used in conjunction with any other DOM element, it returns the width including the padding and excluding the border. This property can be accessed by using the following code:

  ```
  $window.innerWidth
  ```

To learn more about the `innerWidth` property, go to `https://developer.mozilla.org/en-US/docs/Web/API/Window.innerWidth`.

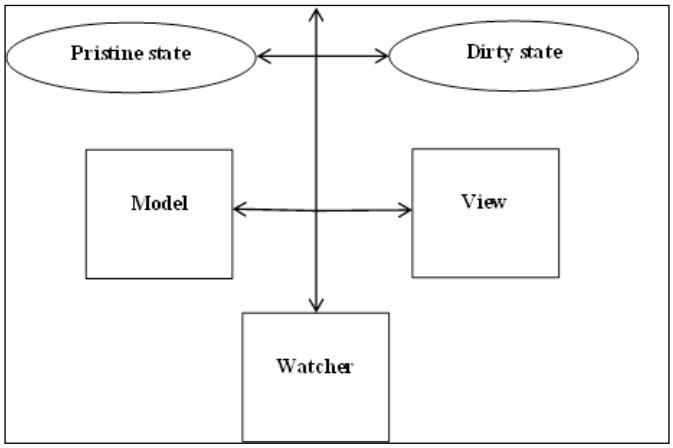
The following screenshot shows the log in the Firefox developer console for the inner and outer width value of the current window object:



In the previous screenshot, the outer and inner widths are **1382** and **1366** respectively for the window. This clearly shows that the inner width is less than the outer as it excludes the scrollbar interface element.

The $watch method

AngularJS follows a life cycle to maintain the two-way data binding between model and view features. In two-way data binding, whenever a change occurs to model, view also gets updated; similarly, a change in View modifies the value of model too. This whole process of model-view duplex communication is maintained through the digest life cycle. The following diagram shows the building blocks of a digest cycle:



AngularJS uses the digest life cycle by performing dirty bit checking. In a nut shell, AngularJS compares a value with its previous value, and if it has changed then a change event is fired for dirty bit checking. AngularJS uses two CSS classes to mark a DOM element as modified or not. These two CSS classes are as follows:

- `ng-dirty`: This class is used as a flag that the DOM is modified by some means
- `ng-pristine`: This class is used as a flag that the DOM element is clean

AngularJS also provides two corresponding methods to manipulate dirty bits involved in AngularJS. These two methods are as follows:

- `$setDirty()`: This method sets the state of the element to dirty
- `$setPristine()`: This method sets the state of the element to pristine

There is no specified time for the digest cycle run in AngularJS. However, the digest cycle has to run at least twice to sync the **Model** and **View** features using two-way binding.

To learn more about the digest cycle, check out `https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest`.

The event binding function

AngularJS provides the `$event` object that can propagate through the DOM element, or a callback method can be attached to read this object. AngularJS provides the `bind()` method to bind a callback method to an element. The syntax for the `bind()` method is provided as follows:

```
angular.element(<DOM element>).bind('<event name>',
function(<event object>){
     //code of event callback
});
```

The previous code shows an anonymous callback function attached to a DOM element, `<DOM element>`, for a given event, `<event name>`. A DOM element can be referred to using the `angular.element()` method. The `angular.element()` method wraps a DOM element as a jQuery element. AngularJS uses a lighter version of jQuery called jqLite.

To find out more about the `angular.element()` method, you can go to `https://docs.angularjs.org/api/ng/function/angular.element`.

Let's bind a callback to a window resize event. This means when the browser window is resized, the callback method is called and the code inside the method is executed. The following code shows the resize event binding for the window element:

```
angular.element($window).bind('resize',
function(event) {
  //Code for resize event callback
  console.log("event ", event);
});
```

To learn more about the `angular.bind()` method, go to `https://docs.angularjs.org/api/ng/function/angular.bind`.

The $log service

AngularJS provides logging services to debug the code. This is really helpful for production issues and is similar to the console logging mechanism. The benefit of using the `$log` service in the code is get to an independent logging mechanism that suffice a browser that does not have console debugging; however, if we use the console log mechanism and the browser does not have the logging mechanism, the code will break at runtime. AngularJS provides the `$logProvider` service to configure the logging default in an application.

The `$log` service provides many utility methods to debug the application based on the severity level of errors. These methods are as follows:

- `log()`: This method writes a log message if the console object is available in the browser
- `info()`: This method writes an information message if the console object is available in the browser
- `warn()`: This method writes a warning message if the console object is available in the browser
- `error()`: This method writes an error message if the console object is available in the browser
- `debug()`: This method write a debugging message if the console object is available in the browser

Built-in directives

The AngularJS architecture is based on the directive concept. There are many useful built-in directives through which AngularJS controls DOM elements. We can use some of them for responsive web application development. Let's explore some of the built-in directives and discuss their use in responsive web application development.

The `ng-if` directive is a conditional directive present inside the `ng` module. This directive will be really useful to manipulate DOM in runtime. For smaller devices, we can only show important sections of the application. To implement this behavior, the `ng-if` directive is very useful. The following pseudocode demonstrates the use of the `ng-if` directive:

```
<div ng-if="isMobile">
<!—Inner HTML DOM content-->
</div>
```

Similarly, there are many built-in directives that we can use for condition-based display, for example, `ng-switch`, `ng-style`, `ng-show`, and `ng-hide`.

Custom directives

AngularJS supports the development of custom directives. If an application needs a custom widget to be displayed on the page, then developers can go for the AngularJS custom directive approach. AngularJS provides a lot of control to the developer over the DOM element, by which page rendering can be manipulated based on the device during runtime. In the following section, we will create some responsive custom directives for our profile page.

Responsive directives

In this section, we will develop some custom responsive directives that can be useful for our profile page. Let's first identify the parts of a profile page that need to be responsive. Yes, there are many parts we can identify that need a responsive design, but let's consider some that really make an impact to a responsive design. The following list shows the important sections that we are going to focus on:

- Responsive images
- Responsive text
- Responsive item lists

Responsive images

In our profile application, we have used the profile image from Gravatar. A Gravatar image follows the user from site-to-site beside the username. Gravatar is a great online tool to manage your profile image universally. You can use your profile image uploaded to Gravatar using a URL. Also, Gravatar supports access to different sizes of the profile image using the URL request parameter. We are going to use the Gravatar image to demonstrate the responsive image.

To learn more about Gravatar images, go to `https://en.gravatar.com/site/implement/images`.

In our profile application, we are going to use three different sizes of profile images from Gravatar. These profile images are 250 x 250, 150 x 150, and 80 x 80 for screen widths more than 767 px, a width between 400 px and 767 px, and a width less than 400 px, respectively.

We will create a custom image directive that will take the Gravatar profile base URL and alternative text input as an attribute to it. This profile URL will be appended with the size query parameter to load different sizes of images to the browser screen. The code for the `responsiveImage` custom directive is as follows:

```
.directive("responsiveImage",
    ["$log", "$window",
    function ($log, $window) {
    return {
        restrict:'E',
        replace:true,
        scope: {
            'respalt': '@imagealt',
            'respsrc': '@imagesrc'
        },
        template: '<img class="profile-image"' +
                  'ng-src="{{modifiedsrc}}" alt="{{respalt}}"/>',
        link: function(scope, element, attribute){
            scope.width = $window.outerWidth;
            scope.$watch("width", function(newWidth, oldWidth){
                $log.log("New width of window : ",newWidth);
                if(newWidth <= 400){
                   scope.modifiedsrc = scope.respsrc +"?s=80";
                }else if(newWidth >400 && newWidth <=767){
                    scope.modifiedsrc = scope.respsrc +"?s=150";
                }else{
```

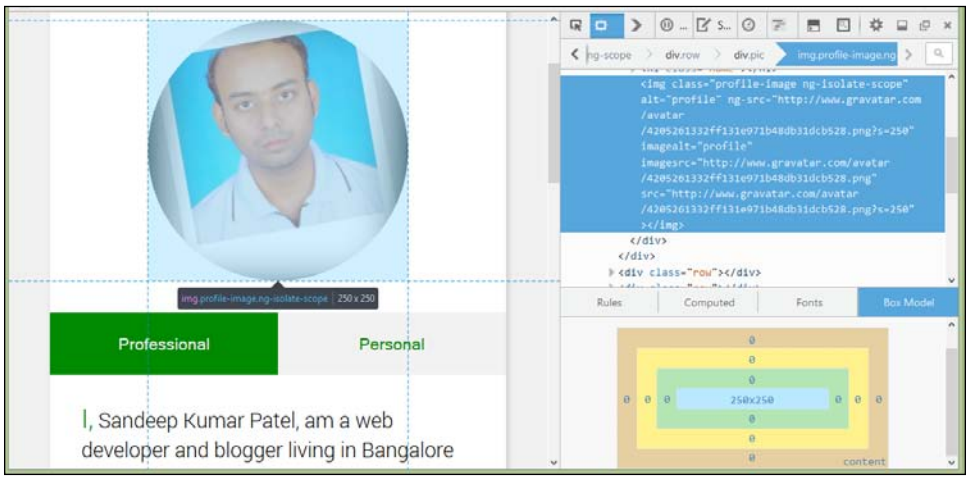
```
                    scope.modifiedsrc = scope.respsrc +"?s=250";
                }
            });
            angular.element($window).bind('resize',function(){
                //Asking AngularJS to run digest cycle
                scope.$apply(function(){
                    scope.width = $window.outerWidth;
                })
        });
    }};
}]);
```

In the previous code, we find the `scope.$apply()` method used inside the callback function of the window's `resize` event. The event callback method updates the `width` property of the scope object outside the AngularJS context that changes the state of the width property to dirty. To change the state of the width property to pristine, a digest cycle needs to be called. The `$apply()` method calls the digest cycle. For this reason, the `$apply()` method is used inside the callback function of the `resize` event.

This custom directive can be called like a normal HTML element. The following code shows the process of calling responsive image directives in our profile application:

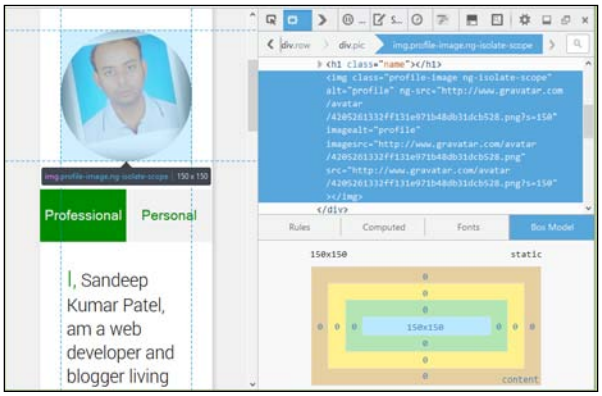
```
<responsive-image imagealt="profile"        imagesrc="http://www.
gravatar.com/avatar/4205261332ff131e971b48db31dcb528.png">
</responsive-image>
```

For desktop type devices where the screen is pretty large, the query parameter for profile image size will be `s = 250`. A COMPASS watcher is attached to the width property to detect the change in the windows width. The following screenshot shows the developer console with the box model for a screen size greater than 767 px:

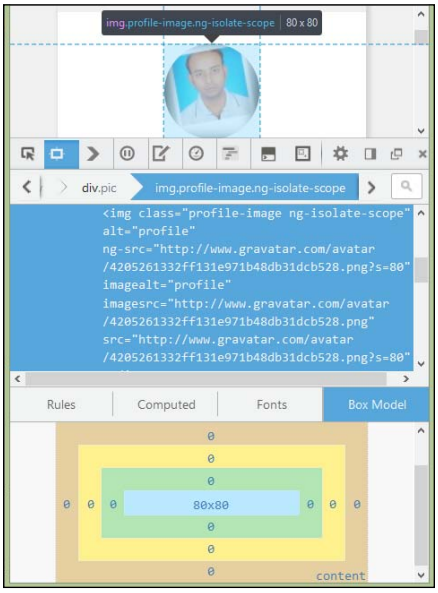


In the previous screenshot, you can see that the `responsiveImage` directive has replaced itself with the given HTML template. Also, the watcher has detected the screen size and modified the profile image URL to the size query parameter of 250 px.

For medium sized devices such as a tablet, the Gravatar URL for the profile image is called with the size query parameter `s = 150`. The following screenshot shows the developer console for a medium device screen size:



For small sized devices such as mobiles, the Gravatar URL for the profile image will have the size query parameter `s = 80`. The following screenshot shows the console for medium sized devices with the **Box Model**:



Responsive text

In the profile application, we have many sections with text content. We will categorize them as header and paragraph types. The top of the profile application page contains the name whose size should be responsive based on the device size. We will develop a custom directive that will take this header text as one of its attributes, and based on the screen size it will apply different CSS classes to change it to the responsive size of the header text. The following code shows three CSS classes based on the screen size:

```
.largeDevice{
  color: green;
  font-size: 4rem;
  background: #eee;
  font-weight: 100;
}
.mediumDevice{
  color: green;
  font-size: 2rem;
  background: #eee;
  font-weight: 100;
}
.smallDevice{
  color: green;
  font-size: 1rem;
  background: #eee;
  font-weight: 300;
}
```

These CSS classes are going to be used by our new custom directives to resize the header text present at the top of the profile page. Details of these CSS classes are as follows:

- `largeDevice`: This CSS class will be used by large devices and has the font size as `4rem` and weight as `100`. Other properties like background and color of the font are the same for all the devices.
- `mediumDevice`: This CSS class will be used by medium devices and has the font size as `2rem` and weight as `100`. The background and font color are the same as other types of devices.
- `smallDevice`: This CSS class will be used by small devices and has the font size as `1rem` and weight as `300`. The background and font color are the same as other types of devices.

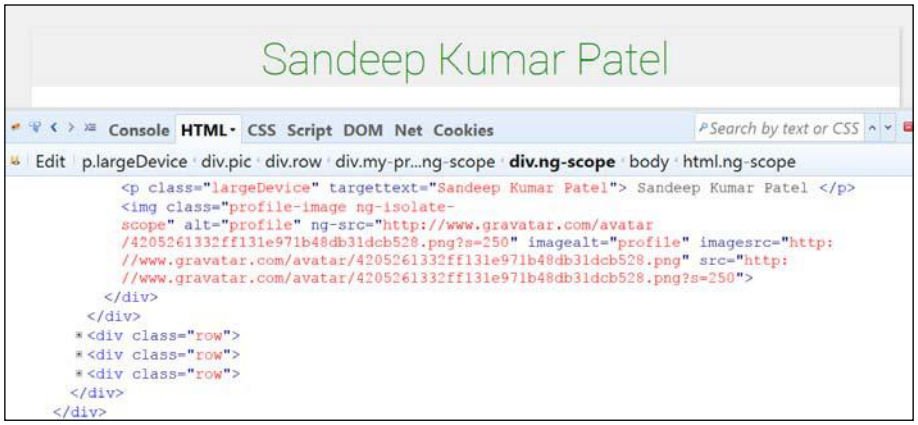
The code for the `responsiveHeader` custom directive is as follows, with the `targetText` attribute as an input to it:

```
.directive("responsiveHeader",
    ["$log","$window",
    function ($log, $window) {
       return {
         restrict:'E',
         replace:true,
         scope:{
              'respText':'@targettext'
          },
         template: "<p class='{{deviceSize}}'> {{respText}} </p>",
         link : function(scope, element, attribute){
             scope.deviceSize = "largeDevice";
             scope.width = $window.outerWidth;
             scope.$watch("width",function(newWidth,oldWidth){
                if(newWidth <= 400){
                  scope.deviceSize = "smallDevice"
                }else if(newWidth >400 && newWidth <=767){
                  scope.deviceSize = "mediumDevice";
                }else{
                  scope.deviceSize = "largeDevice";
                }
              });
              angular.element($window).bind('resize',function(){
                 scope.$apply(function(){
                      scope.width = $window.outerWidth;
                  });
               });
           }};
 }]);
```

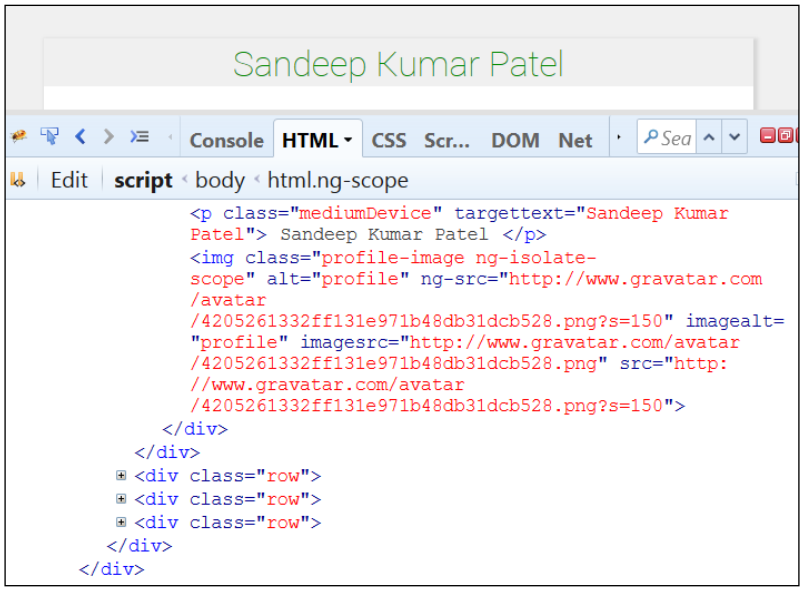
To use the previous custom directive, it should be called like a normal DOM element with the attribute that contains the targeted text to be displayed in the browser. The following code shows the call of this `responsiveHeader` directive inside the DOM element:

```
<responsive-header targetText="Sandeep Kumar Patel">
</responsive-header>
```

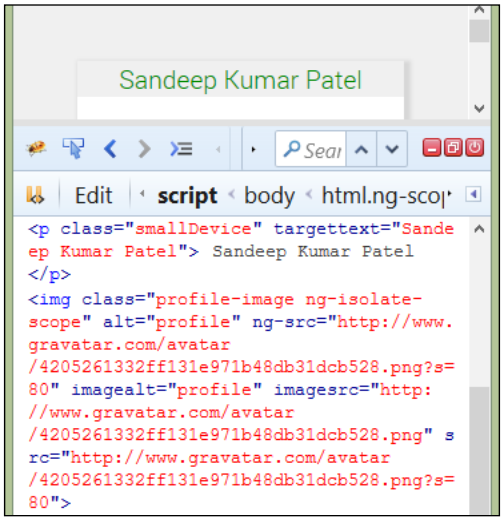
The following screenshot shows the profile header name in large devices. This screenshot also shows the DOM element of the header text of the paragraph element with the CSS class as `largeDevice`.



The following screenshot shows the profile header name in medium type devices. This screenshot has the DOM element using the `mediumDevice` CSS class to change the style of the header text. We can simulate this event by resizing the browser window by dragging one of its corners.



In mobile devices, the header will take the `smallDevice` classes where the text size will be decreased to `1rem` with an increase in font weight to bold or 300. The following screenshot shows the header text for `smallDevice`:



Similarly, we can make a paragraph responsive. In our profile application, we have an `aboutme` section that is a paragraph. This `aboutme` section must be responsive to the device size. The following code shows the SCSS style classes that can be used in the `aboutme` section for different screen sizes. These style classes are similar to the header section except that the first letter has different style and more text:

```
.aboutme {
  &:first-letter {
     color:green;
  }
  &.smallPara{
      font-size: 0.8rem;
     &:first-letter {
      font-size: 1.6rem;
     }
  }
  &.mediumPara{
       font-size: 1.5rem;
      &:first-letter {
       font-size: 1.2rem;
      }
    }
```

```
    &.largePara{
       font-size: 2rem;
      &:first-letter {
        font-size: 4rem;
      }
    }
}
```

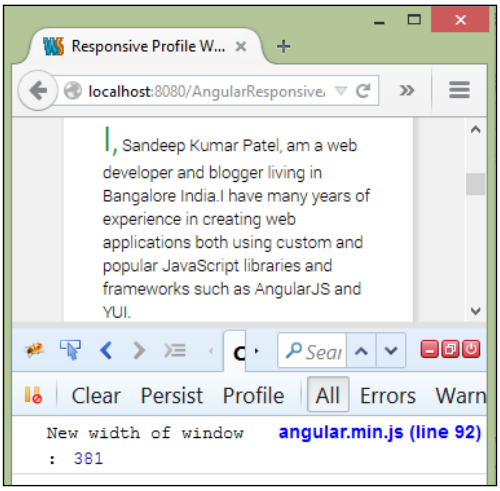
The previous style code can be used by our responsive custom directive `responsiveParagraph` to style the `aboutme` paragraph in the profile page. The code for the custom directive `responsiveParagraph` is as follows:

```
.directive("responsiveParagraph",
 ["$log","$window",
 function ($log, $window) {
   return {
      restrict:'E',
      replace:true,
      scope:{
        'respPara':'@targetpara'
      },
      template: "<p class='aboutme {{paragraphSize}}'> {{respPara}} </
p>",
      link : function(scope, element, attribute){
           scope.paragraphSize = "largePara";
           scope.width = $window.outerWidth;
           scope.$watch("width",
function(newWidth, oldWidth){
                    if(newWidth <= 400){
                      scope.paragraphSize = "smallPara"
                    }else if(newWidth >400 && newWidth <=767){
                      scope.paragraphSize = "mediumPara";
                    }else{
                            scope.paragraphSize = "largePara";
                        }
                    });
         angular.element($window).bind('resize',function(){
            scope.$apply(function(){
                scope.width = $window.outerWidth;
            });
         });
    }};
 }]);
```

The previous code shows the definition of the `responsiveParagraph` custom directive. This directive can be used inside the markup like an HTML element. The following code shows the use of this directive in our profile page for the `aboutme` paragraph:

```
<responsive-paragraph targetPara="{{professional.aboutme}}">
</responsive-paragraph>
```

For small size devices, the `aboutme` paragraph takes the `smallPara` style class and looks like this:



For medium size devices, the `aboutme` paragraph takes the `mediumPara` style class and looks like this:

For larger devices, the `aboutme` paragraph takes the `largePara` style class and looks like this:

Responsive item lists

In our profile application, we have some sections, such as job roles, languages known, tools, web technologies, and so on, as a list of items displayed vertically. Devices with less height have a problem with space. In this section, we will create a custom directive that takes a list of items to be displayed, and based on the device height, the custom directive shows the number of items in the browser and more actionable text. On clicking this actionable text, the list expands and shows all the items.

For a demonstration of the responsive list, we have changed some static profile data to add few more items to the list. The following code shows a part of the modified JSON data:

```
{
  ....
"roles":["Web Developer", "JavaScript Developer", "Front End
Lead","Java Developer"],
  "languages": ["JavaScript", "Java","Python", "C#", "Ruby"],
  "webdevelopment": ["HTML5", "CSS3", "AngularJS", "Jquery",
"BootStrap", "YUI", "Polymer"],
  "tools": ["SASS", "COMPASS", "GRUNT", "GIT", "GULP", "BOWER",
"YEOMAN"],
  "ides": ["WebStorm", "Intelij Idea", "Eclipse", "Sublime Text"],
  ...
}
```

This custom directive is going to use some style classes to control the displayed behavior list. The following code shows the part of style classes that is required for the demonstration of this responsive list:

```
.item-list-container{
  ol{
      margin-bottom: 2px;
      font-size:1rem;
      .smallText{
        font-size:0.8rem;
      }
   }
  .show-more{
     background: none;
     border: none;
          cursor: pointer;
          color:blue;
          margin-left:10px;
       }
    }
```

In our profile application, we have taken the list item limit as 2 for small height devices and 3 for medium height devices. You can change this limit based on your list length. The name of this custom directive is `responsiveList` and the code for this directive is as follows:

```
.directive("responsiveList", ["$log", "$window",
  function($log, $window) {
    return {
      restrict: 'E',
      replace: true,
      scope: {
        'itemList': '=targetlist'
      },
      template: '<div class="item-list-container">' +
        '<ol>' +
        '<li ng-class="{smallText:isMorePresent}" ng-repeat="item in
itemDisplayList">{{item}}</li>' +
        '</ol>' +
        '<button class="show-more" ng-show="isMorePresent" ' +
        'ng-click="showMore(itemDisplayList)"><span>More...</span></
button>' +
        '</div>',
      link: function(scope, element, attribute) {
        scope.isMorePresent = false;
```

```
            scope.$watch("itemList", function(newItemList, oldItemList) {
              scope.itemList = newItemList;
              scope.height = $window.outerHeight;
              scope.itemDisplayList = scope.itemList;
            }, true);
            scope.$watch("height", function(newHeight, oldHeight) {
              var listLength = angular.isDefined(scope.itemList) ?
                scope.itemList.length : 0;
              if (newHeight < 400 && listLength > 2) {
                scope.isMorePresent = true;
                scope.itemDisplayList = scope.itemList.slice(0, 2);
              } else if (newHeight >= 400 && newHeight < 700 && listLength
    > 3) {
                scope.isMorePresent = true;
                scope.itemDisplayList = scope.itemList.slice(0, 3);
              } else {
                scope.isMorePresent = false;
                scope.itemDisplayList = scope.itemList;
              }
            });
            angular.element($window).bind('resize', function() {
              scope.$apply(function() {
                scope.height = $window.outerHeight;
              });
            });
            scope.showMore = function(initalList) {
              scope.itemDisplayList = scope.itemList;
              scope.isMorePresent = false;
            }
          }
        };
    }]);
```

The details of the previous custom directive code are listed as follows:

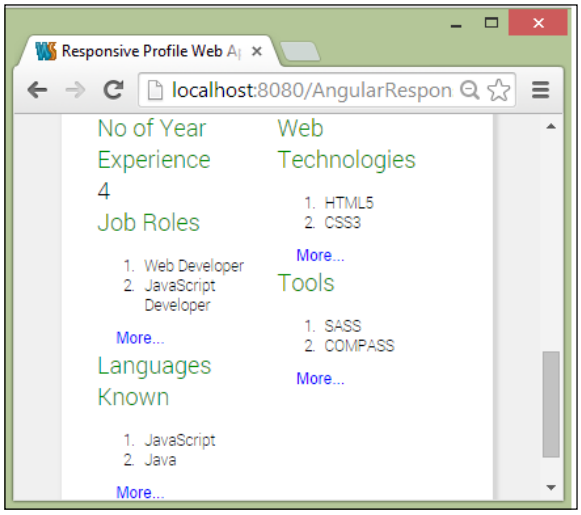
- `scope.height`: This represents the height scope variable with a watcher attached by the `scope.$watch()` method.
- `$window.outerHeight()`: This method returns the outer height of the window.

- `scope.isMorePresent`: This scope variable is a Boolean variable. This flag variable has true or false value based on the following condition:
    - `true`: This condition is used if the device has medium device height, that is, between `400` and `700` and list item length is greater than `3`. It is also used if the device height is less than `400` and item list length is greater than `2`. You can change this condition based on your requirements and the item list size that you will be using for your profile.
    - `false`: This condition is used if the device is of greater height.
- `scope.showMore()`: This method is attached to the click handler to show the full list.
- `smallText`: This is a style class used with the `li` element with the `ng-class` directive and gets activated if `scope.isMorePresent` is `true`. The purpose of this style class is to change the font size of the list item to a smaller one.

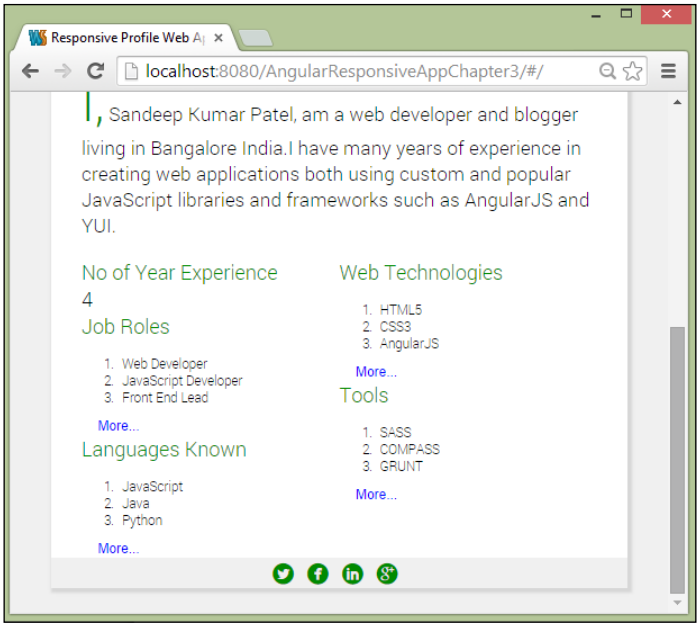
The following HTML code shows the use of this `responsiveList` custom directive in our profile application. Do note: this is just a part of the full code and snippet, and the full code can be found on the Packt Publishing code support website:

```
<div class="section">
  <div class="divider">
    <h4>No of Year Experience</h4>
    {{professional.years}}
    <h4>Job Roles</h4>
    <responsive-list targetlist="professional.roles">
    </responsive-list>
    <h4>Languages Known</h4>
    <responsive-list targetlist="professional.languages">
    </responsive-list>
  </div>
  <div class="divider">
    <h4>Web Technologies</h4>
    <responsive-list targetlist="professional.webdevelopment">
    </responsive-list>
    <h4> Tools</h4>
    <responsive-list targetlist="professional.tools">
    </responsive-list>
  </div>
</div>
```

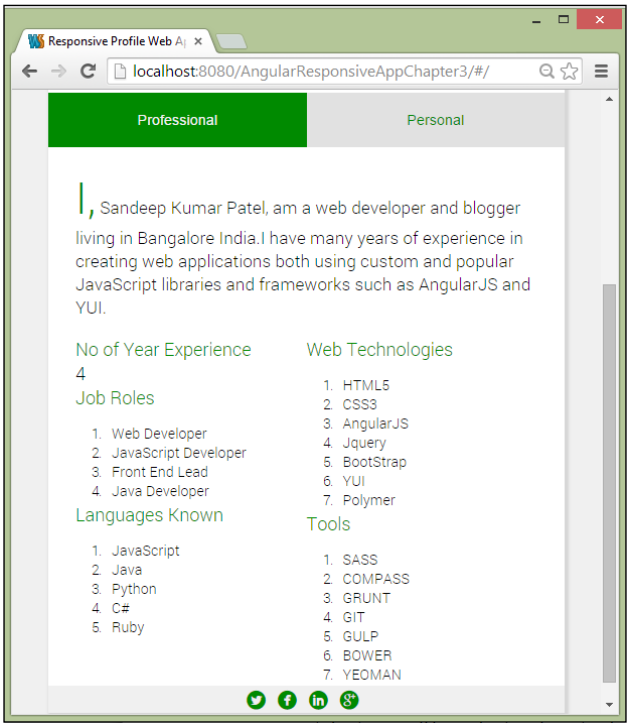
For lower height devices, the **More**... text will appear at the bottom of the list if it has more than two items. The following screenshot shows a small height device showing two items and a more text label at the bottom:



For a medium sized devices, the list displays three items and a more text label at the bottom of the list. The following screenshot shows the modified list for a medium sized device:



For taller devices, we will not restrict the item list being shown. The following screenshot shows the profile page for taller devices:



Summary

In this chapter, you learned about creating custom directives using AngularJS. Also, we explored how a custom directive can be responsive to different screen sizes. In the next chapter, you will learn about implementing breakpoints in the layout using the AngularJS framework.

4
The AngularJS-based Breakpoints for Layout Manipulation

In this chapter, you will understand the concept of breakpoints and its role in responsive web application development followed by a section on the publisher-subscriber mechanism implementation in AngularJS application. You will also learn how to use AngularJS framework to implement breakpoints in a web application.

Page layout

The main purpose of a web application is to display meaningful and relevant information to the end users in web pages. From the developer's perspective, this information is nothing but page content. These page contents are arranged in a specific format and order. The order in which this content is presented to the end user is nothing but page layout. In other words, a layout means the way the page is divided into columns and rows.

Page layout has a strong impact on user experience. Hence, the layout design phase needs more focus before proceeding to development. While designing for page layout we must consider the following points:

- **Devices**: Users can browse on different kinds of devices such as mobile, tablet, and desktop. All these devices may have a different operating system installed.
- **Browsers**: Users can browse the application on different browsers such as Chrome, Firefox, and Internet Explorer. Also, the browsers have some additional sections such as a toolbar and status bar.
- **Screen resolution**: Users can use different resolution displays such as VGA and HD.

All the previous points impact the page layout design, as content presentation has a direct relationship with it.

Layout type

There are four different types of web page layout design. These layout designs are as follows:

- **Fixed**: Web pages have a fixed width and do not change irrespective of device, browser, or screen type.
- **Fluid**: Web pages have a percentage-based width. This means each section expands and shrinks relatively.
- **Adaptive**: Web pages are developed based on CSS3 media queries with a fixed width component.
- **Responsive**: Web pages build on fluid grids and scale up and down using media queries.

Breakpoints

Breakpoints are sets of points, where we tweak our page content with a modified arrangement. In the early days, these break points were considered only for media types. Both, adaptive and responsive layout uses the breakpoints and rearranges the content inside it. However, there is a difference in their approach. In adaptive design, the page content has a fixed width for each breakpoint. In the responsive layout approach, the content is fluid in nature and rearranges the contents relatively.

Responsive and common breakpoints

Developers have web application development support for multiple targeted devices. However, a web application must be responsive enough, should not depend on the devices, and can arrange its content in the best possible way without losing user engagement. Based on the approach of responsive web application development, these breakpoints can be of two types: common and responsive breakpoints. Based on the application type, a developer can choose either of these approaches.

AngularJS publisher and subscriber

AngularJS supports the publish and subscribe mechanism to communicate among modules. It uses `$emit`, `$broadcast`, and `$on` to implement, publish and subscribe design pattern. Using `$emit` and `$broadcast`, a message can be published inside the AngularJS application. Using the `$on` method, a published message can be subscribed.

Publishing a message using $emit

When a message is published using `$emit`, it starts propagating from the current scope to the upper level scope and continues up to the application scope, which is `$rootScope`. The following code shows the syntax of the `$emit` method to publish a message:

```
$scope.$emit("<message>",<argument>);
$rootScope.$emit("<message>",<argument>);
```

In the previous code, you can see that the emit message takes two parameters; their details are as follows:

- `message`: This field represents the name of the message that will be published inside the AngularJS application
- `argument`: This field represents the argument parameter object that is attached and propagated with the published message

Also, we discussed that the `$emit` message propagates up to the application root. So, why is the second line `$rootScope.$emit()` as shown in the previous code? It seems vague, right? But it works, and the message is published and available to the subscribers present in the root scope application.

Publishing a message using $broadcast

When a message is published using `$broadcast`, it starts propagating from the current scope to the next inner child scope and continues up to the last child scope. The following code shows the syntax of the `$broadcast` method:

```
$scope.$broadcast("<message>",<argument>);
$rootScope.$broadcast("<message>",<argument>);
```

In the previous code, the broadcast message takes two parameters similar to the emit message, where the first parameter is the name of the message itself and the second parameter is the additional argument that passes as object to the subscribers.

Subscribing to a message using $on

A published message through any approach, `$emit` or `$broadcast`, can be subscribed using the `$on` method inside AngularJS application. The following code shows the syntax of `$on` while subscribing a message:

```
$scope.$on("<message>", "<callback>");
```

In the previous code, the `$on` method takes two parameters to subscribe to a published message and the details are as follows:

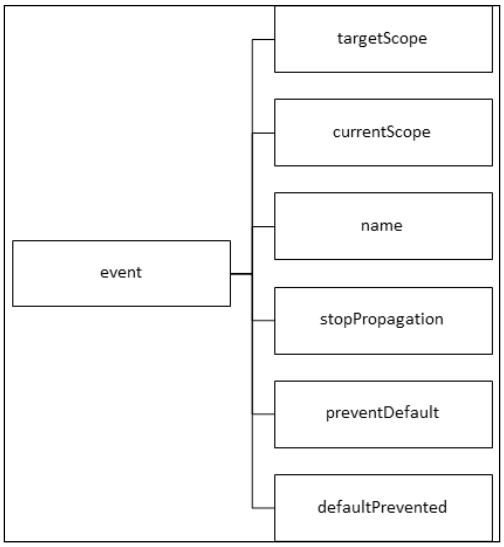
- `message`: This field represents the name of the message that is subscribed to by the module.
- `callback`: This field represents the listener function that can be executed once the message is received. The syntax for the callback function is as follows:

  ```
  function(event, arguments) {
      //Code for listener...
  }
  ```

  The callback method takes two parameters and their details are as follows:

  - `arguments`: This field represents the arguments that passed with the message while publishing the message
  - `event`: This field represents the event object created from the source while publishing the message

The callback method returns the event object, which has useful information stored as properties. The following diagram shows some of the important properties contained inside this event object:



In the previous diagram, we can identify most of the event properties resembling the jQuery event object. This is due to the AngularJS internally using a lighter version of jQuery called **jqLite**. However, we will quickly go through these properties. Details of these properties are as follows:

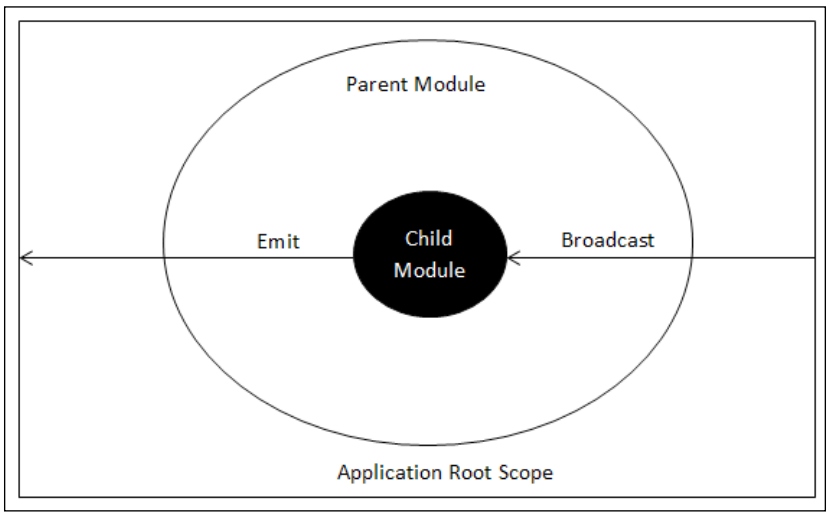
- **targetScope**: This property represents the scope of the source module from where the message has been published
- **currentScope**: This property represents the scope of current module from where the message is currently subscribed
- **name**: This property represents the name of the event
- **stopPropagation**: This property can be called to stop the propagation of the published message by the `$emit` method
- **preventDefault**: This property can be called to stop the default behavior by setting the `defaultPrevented` flag
- **defaultPrevented**: This property is the Boolean flag and is set to true when the method is `preventDefault`

AngularJS provides an exceptional handling mechanism using the `$exceptionHandler` service. Any exception occurred while subscribing to a message is passed to the AngularJS `$exceptionHandler` service module.

To learn more about the exceptional handling mechanism in AngularJS, visit `https://docs.angularjs.org/api/ng/service/$exceptionHandler`.

The difference between $emit and $broadcast

However, there is some difference between `$emit` and `$broadcast` in the way the published message propagates inside the AngularJS application. The following diagram shows the graphical representation of the published message using `$emit` and `$broadcast`:



The previous diagram clearly shows the direction of propagation of a published message. The message published using the `$emit` method can be canceled by any subscriber; however, a message published using `$broadcast` is not cancelable.

To learn more about the publish and subscribe mechanism visit `https://docs.angularjs.org/api/ng/type/$rootScope.Scope`.

An example of the publish and subscribe mechanism

In this section, we will quickly go through the publish and subscribe mechanism, with a quick, small example. This example uses the emit method to publish a message. However, you can use the broadcast method too; it is just a matter of choice for this quick example. In this example, we have created a button with a click event listener attached to it. On clicking this button, a message will be published on the subscriber's side, which is displayed in the browser screen. The HTML code for this example is present in the `pubSubExample.html` file and is as follows:

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
  …
  …
  <title>AngularJS Pub-Sub Example</title>
</head>
<body>
  <div ng-controller="PubController">
    <button name="publishButton" ng-
    click="publishMessage()">Publish</button>
  </div>
  <div ng-controller="SubController">
    <h2>{{receivedMessage}}</h2>
  </div>
</body>
</html>
```

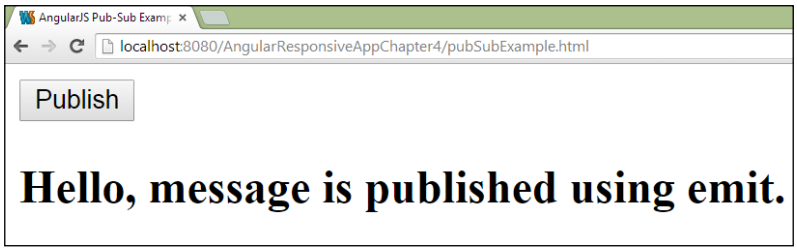
In the previous HTML code, we created an AngularJS application module named `myApp` and have added two controllers, a button, and an AngularJS expression. The details of these items are as follows:

- `PubController`: This is an AngularJS controller through which we will publish the message
- `publishButton`: This is an HTML button element and is attached to the AngularJS click event, which calls the `publishMessage()` method
- `SubController`: This is an AngularJS controller through which we will subscribe to the message
- `recievedMessage`: This is an AngularJS expression that will display the received message

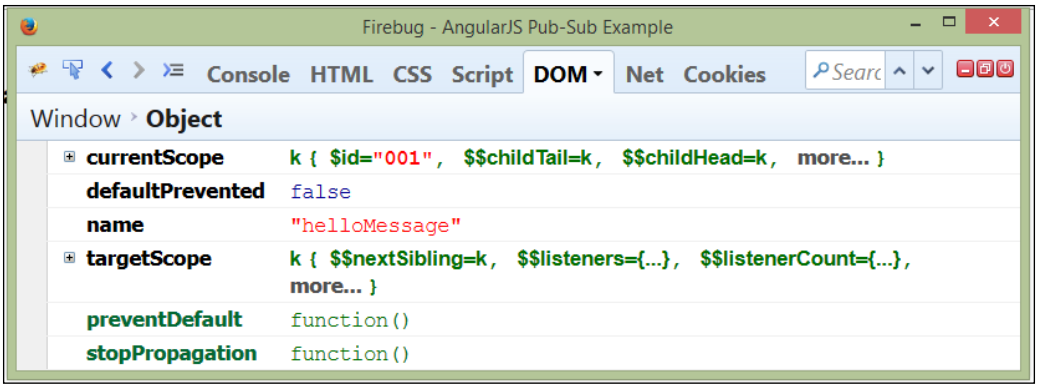
Now, let's check out the definition of each controller and member properties and methods in their scope. The code for these controllers is as follows:

```
<script>
  var myApp = angular.module('myApp', []);
  myApp.controller("PubController",
    function($scope) {
      $scope.publishMessage = function() {
        $scope.$emit("helloMessage", {
          "data": "Hello, message is published using emit."
        });
      }
    });
  myApp.controller("SubController",
    function($scope, $rootScope) {
      $rootScope.$on("helloMessage", function(event, arguments) {
        $scope.recievedMessage = arguments.data;
      });
    });
</script>
```

In the previous code, we saw that the `publishMessage()` method contains the code for publishing a message name, `helloMessage`, with a message object with data as the key and a string message as the value. In `SubController`, a subscriber is attached to the `helloMessage` event and on arrival of the message it extracts the data property from the argument object and assigns it to the `recievedMessage` scope variable, which ultimately binds to an expression in the HTML code. The output of the previous program, when the publish button is clicked on, can be seen here:



Let's debug the event object that is caught by the subscriber present inside `SubController`. To debug the event object we have added a line, `$log.log(event)`, inside the callback method. The following screenshot shows the Firebug console of the event object with all five properties discussed in the previous section of the chapter:



Custom attributes

A custom directive in AngularJS can be used as an element, attribute, and a comment. In the previous chapter, we learned about the custom directive that is used as an element inside an application. In this chapter, we will explore how the custom directive can be used as an attribute and help in responsive application development.

We will create a custom attribute named breakpoint, which can be used in the HTML element or with the body tag of the document. This custom attribute takes a serialized JSON object as the string that contains `<key, value>` pairs. A sample of this string is as follows:

```
{
   "400":"small-screen",
   "700":"medium-screen",
   "1000":"large-screen"
}
```

The details of the previous JSON string are as follows

- `key`: In the previous example, the keys are `400`,`700`, and `1000`. These keys represent the size of the window, where the content needs to be rearranged. In other words, these are the breakpoints.
- `value`: In the previous example, the values are the `small-string`, `medium-screen`, and `large-screen` strings. These are CSS classes that can be used by the subscriber directive when the breakpoint changes.

Developing a custom attribute

A custom attribute directive can be created using the AngularJS directive module and using the restrict property with the `attribute` (A) value. The code for the custom attribute directive is as follows:

```
.directive("breakpoint",
["$log", "$window", "$rootScope", "$timeout",
   function($log, $window, $rootScope, $timeout) {
     return {
       restrict: 'A',
       link: function(scope, element, attributes) {
         var breakpointString = attributes.breakpoint,
           customBreakpoints = angular.fromJson(breakpointString);
         scope.breakpoint = {
           windowSize: $window.outerWidth,
           styleClass: ''
         };
         //Method for broadcast breakpointClassChange event
         scope.broadcastBreakEvent = function() {
             $log.log("Broadcasting breakpointClassChange...", scope.
breakpoint);
             $rootScope.$broadcast('breakpointClassChange', scope.
breakpoint);
           }
           //Scope watcher for styleClass property to broadcast
breakpointClassChange event
         scope.$watch('breakpoint.styleClass', function(newStyleClass,
oldStyleClass) {
           if (newStyleClass.length > 0 && newStyleClass !=
oldStyleClass) {
             $timeout(function () {
                 scope.broadcastBreakEvent();
              });
           }
         });
         //Scope watcher for windowSize property to update the new
style class
         scope.$watch('breakpoint.windowSize', function(newSize,
oldSize) {
           var className = 'small-screen';
           for (var customPointKey in customBreakpoints) {
             var breakSize = parseInt(customPointKey, 10);
             if (breakSize < newSize) {
               className = customBreakpoints[breakSize];
```

```
            }
          }
          scope.breakpoint.styleClass = className;
        });
        //Window resize event updates the windowSize property
        angular.element($window).bind('resize', function() {
          scope.$apply(function() {
            scope.breakpoint.windowSize = $window.outerWidth;
          });
        });
        //For first time page load
        angular.element(document).ready(function() {
          $timeout(function() {
            scope.broadcastBreakEvent();
          }, 100);
        });
      }
    };
  }
])
```

The previous code represents the definition of the custom directive attribute, breakpoint. The details of this code are as follows:

- `breakpoint`: This is a scope object of the breakpoint custom directive. This breakpoint object has two properties: `windowSize` and `styleClass`. The `windowSize` property has the `$window.outerWidth` value form.
- `resize`: This event is attached with the `$window` object. Whenever the window is resized, the callback function gets executed. The callback function calculates `outerWidth` of the window and updates the value of the `windowSize` properties present inside the `breakpoint` scope object.
- `customBreakpoints`: This variable store saves all the user-supplied breakpoint JSON objects. This supplies the JSON string parsed message using the `angular.fromJSON()` method.
- `windowSize watcher`: A watcher is created using the `$watch()` function to list the `windowSize` property changes. In the code, the `resize` event callback calculates the outer width and updates the `windowSize` property, which in turn triggers this watcher. This watcher checks the value of the new width and compares it to the user-supplied `customBreakpoints` ranges. Based on the comparison, the `styleClass` property value gets updated.

- `styleClass watcher`: A watcher is created using the `$watch` function. This watcher lists any changes in the `styleClass` property. The `styleClass` value is changed by the `windowSize` watcher by comparing the user-supplied breakpoint string to the current size.
- `broadcast`: When the `styleClass` property value is changed, the `breakpointClassChange` event is broadcasted inside the AngularJS application. It also attaches the breakpoint scope object in the argument.

Implementing a custom attribute

The previous custom directive can be used in any top level element or in the body HTML element with the user-supplied breakpoint in the JSON string format. This custom directive broadcasts the `breakpointClassChange` event message with the breakpoint argument attached. The other member directives of the AngularJS application can listen to this message and apply the changed class. To demonstrate the previous attribute, we will use the `responsiveParagraph` and `responsiveImage` custom directives that we developed in the previous chapter. The breakpoint attribute is used in our responsive application in the body HTML element and its code is as follows:

```
<!DOCTYPE html>
<html lang="en" ng-app="profileApp">
<head>
...
...
</head>

<body breakpoint='{"400":"small-screen", "700":"medium-
screen","1000":"large-screen"}'>
  <!--AngularJS view -->
  <div ng-view="">
  </div>
...
...
</body>
</html>
```

The `responsiveParagraph` custom directive listens to the broadcast of `breakpointClassChange` and updates with the changed class. The modified `responsiveParagraph` custom directive code is as follows:

```
.directive("responsiveParagraph", ["$log", "$window", "$rootScope",
  function($log, $window, $rootScope) {
    return {
```

```
        restrict: 'E',
        replace: true,
        scope: {
          'respPara': '@targetpara'
        },
        template: "<p class='paragraph {{paragraphSize}}'> {{respPara}}
</p>",
        link: function(scope, element, attribute) {
          scope.$on("breakpointClassChange", function(event, argument) {
            $log.log("responsiveParagraph receiving
breakpointClassChange ", argument);
            scope.$apply(function() {
              scope.paragraphSize = argument.styleClass;
            })
          });
        }
      };
    }
])
```

The details of the modified `responsiveParagraph` code are as follows:

- `Message subscription`: The `responsiveParagraph` custom directive listens to the published message using the `$on()` method. The first argument, `breakpointClassChange`, is the message name that is subscribed to and the second argument is the callback function. The first parameter of the callback function is the event object and the other remaining parameters are the objects that are associated to the source's side. In our application, we have attached the breakpoint object containing the changed `styleClass` name and the current `windowSize`.
- `Updating css class`: The callback function retrieves the attached argument by the source and retrieves the value of the `styleClass` name and updates its DOM.

In the previous code, you saw a small change in the template string and the paragraph element has the class property binded with the AngularJS expression, `paragraphSize` . This scope variable, `paragraphSize`, gets its value by listening to the `breakpointClassChange` event. So, we need to change the style classes in our SASS file. The changes in the style are as follows:

```
p.paragraph { &:first-letter {
    color: green;
  } &.small-screen {
    font-size: 0.8 rem; &: first-letter {
      font-size: 1.6 rem;
```

```
      }
    } &.medium-screen {
      font-size: 1.2 rem; &: first-letter {
        Font-size: 3 rem;
      }
    } &.large-screen {
      font-size: 2 rem; &: first-letter {
        font-size: 4 rem;
      }
    }
}
```

The next custom directive that we will change is `responsiveImage`. This directive will listen to the `breakpointClassChange` event and based on the incoming message, it will change the profile image URL. The modified code for the `responsiveImage` directive is as follows:

```
.directive("responsiveImage", ["$log", "$window",
  function($log, $window) {
    return {
      restrict: 'E',
      replace: true,
      scope: {
        'respalt': '@imagealt',
        'respsrc': '@imagesrc'
      },
      template: '<img class="profile-image"' +
        'ng-src="{{modifiedsrc}}" alt="{{respalt}}"/>',
      link: function(scope, element, attribute) {
        scope.$on("breakpointClassChange", function(event, argument) {
          $log.log("responsiveImage receiving breakpointClassChange ",
argument);
          scope.$apply(function() {
            if (angular.equals(argument.styleClass, "large-screen")) {
              scope.modifiedsrc = scope.respsrc + "?s=250";
            } else if (angular.equals(argument.styleClass, "medium-
screen")) {
              scope.modifiedsrc = scope.respsrc + "?s=150";
            } else if (angular.equals(argument.styleClass, "small-
screen")) {
              scope.modifiedsrc = scope.respsrc + "?s=80";
            }
          })
        });
      }
    };
  }
])
```

In the previous code, the directive listened to the broadcast event. After receiving the message, it extracts the style class and compares it to the device type string using the `angular.equals()` method. Based on the comparison, it updates the `modifiedsrc` scope variable with the appropriate profile image URL. Similarly, we can develop any number of custom directives listening to the `breakpointClassChange` event and manipulate the CSS classes for an optimized responsive page. You can find some new additional directives for the text and list items in the code package of this book. These directives have similar implementation such as the `responsiveParagraph` and `responsiveImage` directive.

In the profile application, we have some text values such as **weight** is **68kg**, **chest** is **49 inch**. To make these strings responsive a new custom directive, `responsiveText`, needs to be developed. This directive will be similar to other developed directives. The code for the `responsiveText` directive is as follows:

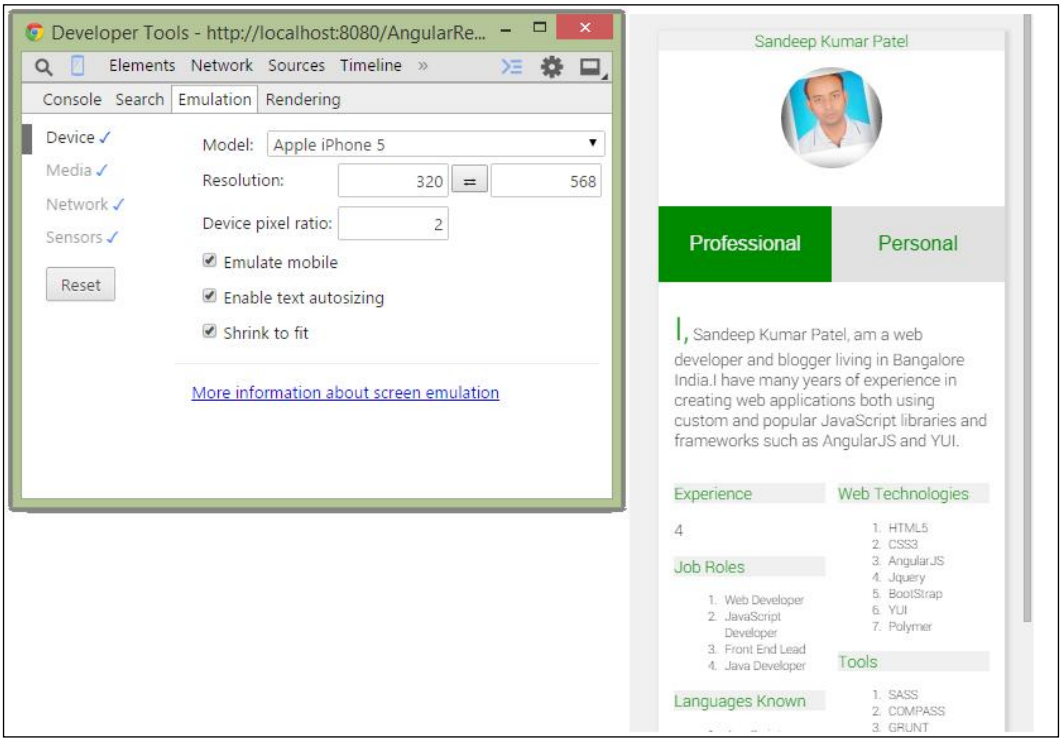
```
.directive("responsiveText", ["$log", "$window",
  function($log, $window) {
    return {
      restrict: 'E',
      replace: true,
      scope: {
        'respText': '@targettext'
      },
      template: "<p class='text {{deviceSize}}'> {{respText}} </p>",
      link: function(scope, element, attribute) {
        scope.$on("breakpointClassChange", function(event, argument) {
          $log.log("responsiveText receiving breakpointClassChange ",
argument);
          scope.$apply(function() {
            scope.deviceSize = argument.styleClass;
          })
        });
      }
    };
  }
])
```

The `responsiveText` custom directive needs CSS style classes for different service sizes, which it receives from the `breakpointClassChange` event. The style classes for this directive in SASS format are as follows:

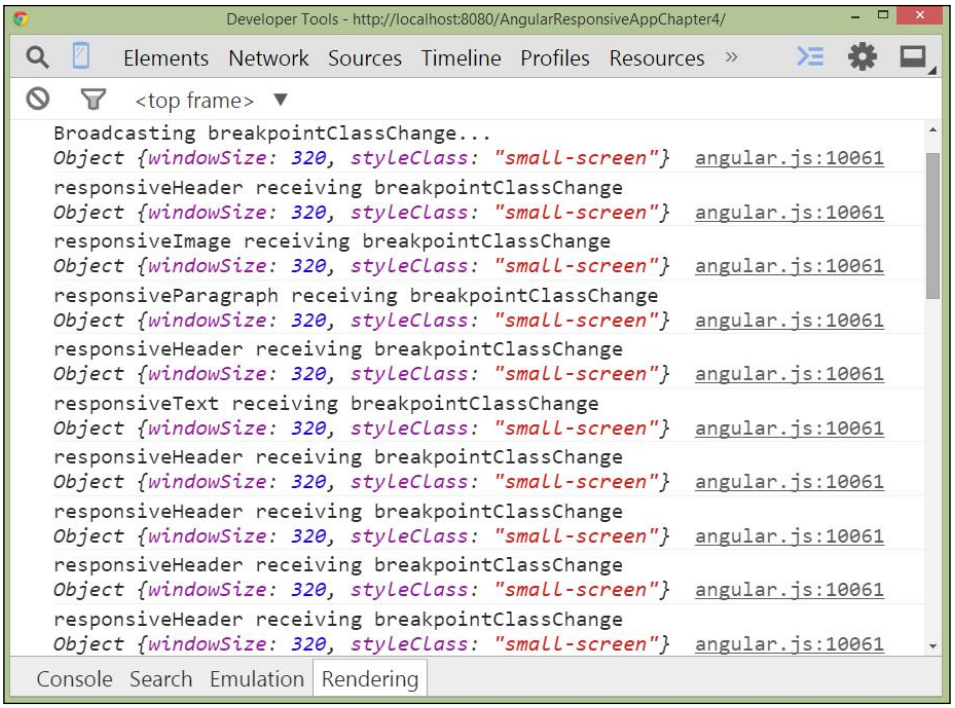
```
p.text {&.small-screen {
    font-size: 0.8 rem;
  } &.medium-screen {
    font-size: 1.2 rem;
```

```
    } &.large-screen {
      font-size: 2 rem;
    }
  }
```

After all these changes, we can test the responsiveness of the profile application. To test this on mobile devices we can use the Chrome developer console emulation tab. The following screenshot shows the profile application with the Chrome emulation set to Apple iPhone 5:



During the page load, the `breakpointClassChange` event is published by the breakpoint directive, is subscribed by all other directives, and CSS classes get applied to the content. All these directives are attached to the `$log` module to get debug information about these directives. The following screenshot shows the Chrome developer console log for the previous iPhone 5 that simulates the screen in the Chrome developer simulation option:



The complete code for the profile application can be found and downloaded from the Packt Publishing website. This code package contains all the previous directive implementations that we have discussed in this book.

Summary

In this chapter, you learned about breakpoints and the publish and subscribe mechanism in AngularJS. Also, you learned how the publish and subscribe mechanism can be used to implement breakpoints in the responsive web application.

5
Debugging and Testing Responsive Applications

In this chapter, we will explore some of the available online and offline tools to debug and test responsive web applications. The scope inside a module is the backbone of an AngularJS-based application. It's really important for an AngularJS developer to know about debugging and its scope. You will learn some of the scope debugger tools to track scope variables.

Batarang

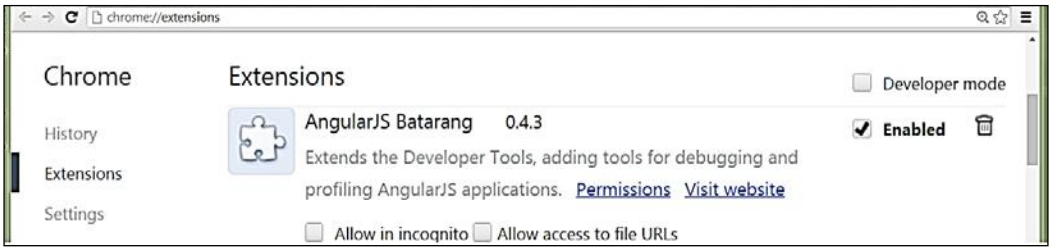
Batarang is the Chrome extension to debug the AngularJS application. This extension is developed by the AngularJS team. Using Batarang, we can debug models, bindings, dependencies, scopes, and applications. Batarang also exposes some methods as APIs to be accessed by the developer console.

Installing and configuring Batarang

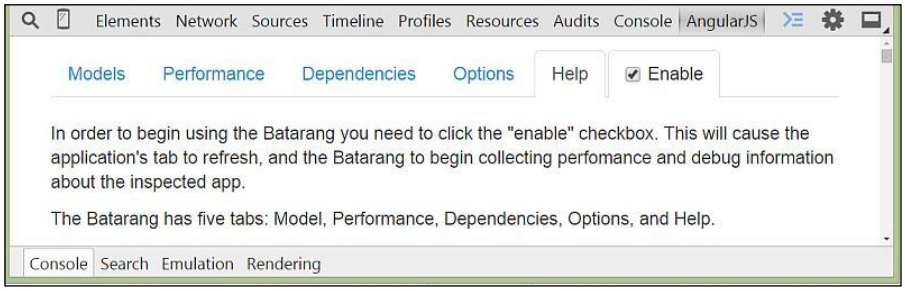
Batarang is a Chrome extension and is available in Chrome Web store. You can search for Batarang in Chrome Web store and install it.

The direct link to this extension is `https://chrome.google.com/webstore/detail/ighdmehidhipcmcojjgiloacoafjmpfk`.

You can open this link in the Chrome browser and install it. After installing of Batarang, it needs to be enabled. To enable this extension go to `chrome://extensions` in the Chrome browser. The following screenshot shows the Chrome Extensions tab with Batarang installed:



To enable this extension, we need to select the **Enabled** checkbox. Now, Batarang is enabled and can be found in the Chrome developer console. By default, Batarang is available as a tab in the developer window. The following screenshot shows the Chrome developer console with Batarang:



Using Batarang

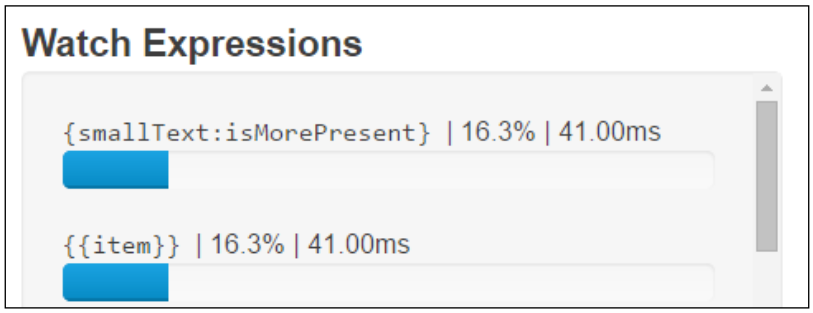
To use Batarang you need to select the **Enabled** checkbox to debug the AngularJS application. You can find this checkbox in the previous screenshot on the right side of the window.

The Batarang extension is enabled in the browser, but debugging still needs to be enabled.

Once the checkbox is selected, Batarang collects all the data relevant to debugging in a different tab. The details of these tabs for the profile application are as follows:

- **Model**: This section contains a hierarchical representation of all the models and their corresponding values in a tree structure, for example, the following screenshot shows model values for the corresponding element:

- **Performance**: This section contains details of all the performance-related information, such as the time taken to retrieve the value of a watch expression. The following screenshot shows the `smallText` and `item` scope variables' performance statistics:



- **Dependencies**: This tab shows a graphical representation of relationships among different AngularJS modules, including custom directives.
- **Options**: This tab has additional configurable options for the Batarang tool.

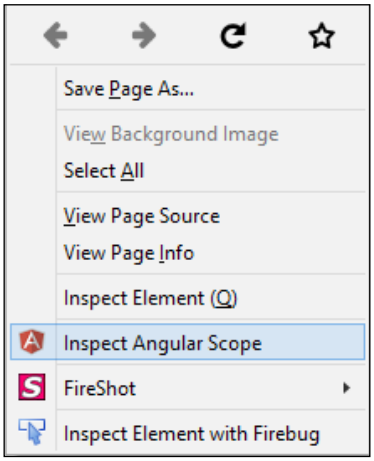
AngularJS scope inspector 0.1.2

If you are using the Firefox browser for application development, then this extension is for debugging. The scope inspector is a Firebug extension to debug scope values in an AngularJS application. To install this extension, we need to open the Firefox add-ons section. To open the Firebug add-ons section, issue the `about:add-ons` command in the Firefox browser's address box. Then, search for the AngScope extension and install it. There is also another way to install this debugger directly by downloading the `.xpi` file.

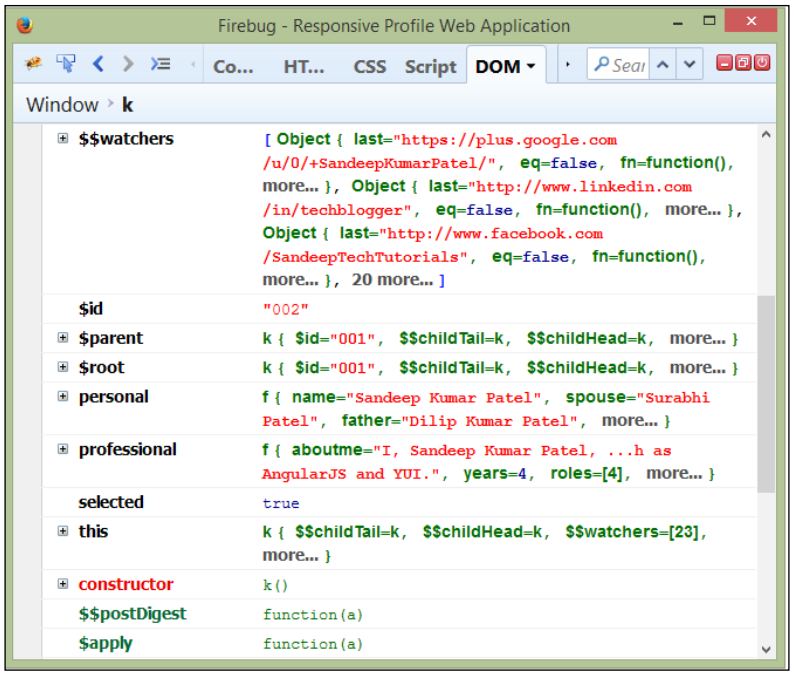
We can find the AngScope's `.xpi` file and a `build` directory at `https://github.com/kosprov/AngScope`.

The `build` directory contains the AngScope's `.xpi` file. To install this file, we need to use the **Open with Firefox** option by clicking the right mouse button.

After successful installation of AngScope, you can find the extension listed as **Inspect Angular Scope** under the drop-down menu items that you can navigate to by clicking the right mouse button. The following screenshot shows the AngScope listed in the menu:



On clicking on this **Inspect Angular Scope** menu item, all the scope variables will be listed in the Firefox developer console. The following screenshot shows the scope variable listing for the profile application that we developed in *Chapter 3*, *The AngularJS Directive-based Approach*:



In the previous screenshot, you can see that all the scope variables are listed. This really helps a debugger to track each of them while fixing an application-related issue.

Online and offline tools

In previous sections of this chapter, you learned about the debugging scope and other related parameters in an AngularJS application. In this section, we will list some of the online and offline tools to test a responsive application. These tools provide a simulated environment for different devices to help developers to test their application without spending a lot on real devices.

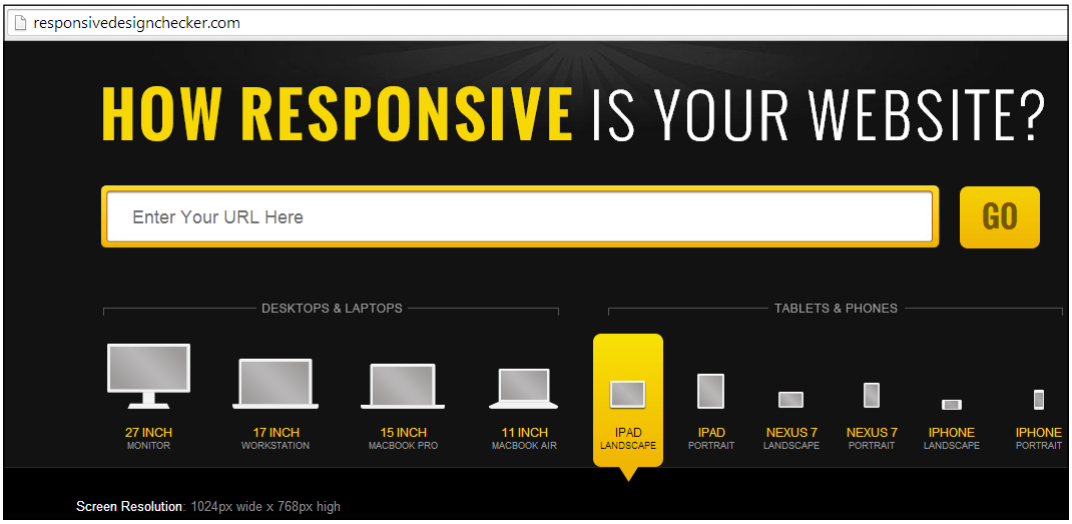
Online tools

There are many online tools available for different browsers to debug responsive applications; however, we will explore only a few of them. As we have not deployed our profile applications, we cannot test them on online tools as these tools require a real server URL; however, we will get introduced to some of them. The lists of online tools that we are going to explore are as follows:

- The responsive design checker tool
- The responsive test online tool

The responsive design checker tool

The responsive design checker tool is an online tool to test responsive designs. The sizes are present in inches. This online tool provides multiple online simulators for mobile, tablet, and desktop devices. This can be found at `http://responsivedesignchecker.com`. The following screenshot shows the home page of this tool:



The responsive test online tool

The responsive test tool is an online tool to test responsive applications. This tool provides many online device simulators to test the application. In the desktop category, you can find large, medium, and thunderbolt display sizes. It is available at `http://responsivetest.com`. The following screenshot shows the home page of this tool:

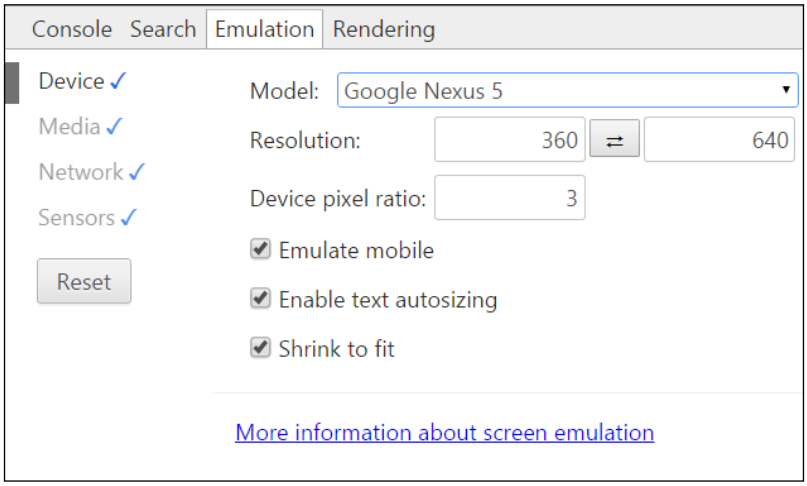
Offline tools

In the previous section, we explored some of the online tools. Let's explore some of the offline tools now. The offline tools that we will discuss are as follows:

- Chrome developer emulation
- Opera mobile emulator
- FireBreak add-ons

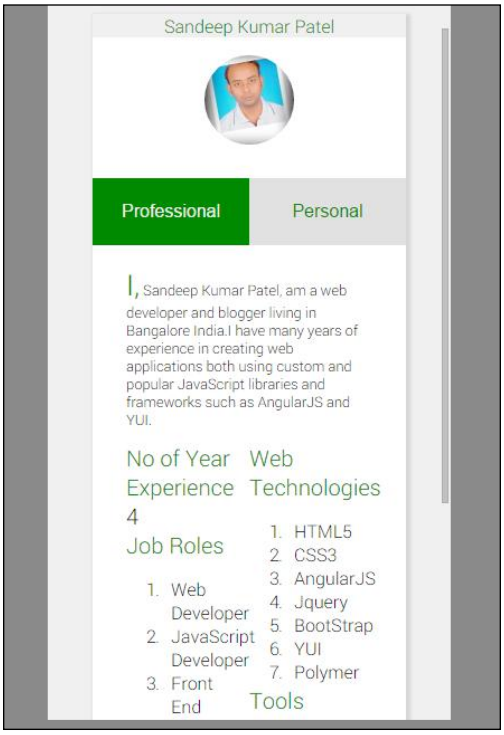
Chrome developer emulation

This tool is available as a part of the Chrome developer console and is developed by the Chrome team. The Chrome developer console can be opened using the `F12` key or by selecting the **Developer tools** option from the tools menu item. The following screenshot shows the Chrome emulation tab:



In the previous screenshot, you can identify that the Chrome emulation section provides many configurable sections, such as **Model**, **Resolution**, and **Device pixel ratio** with some additional options.

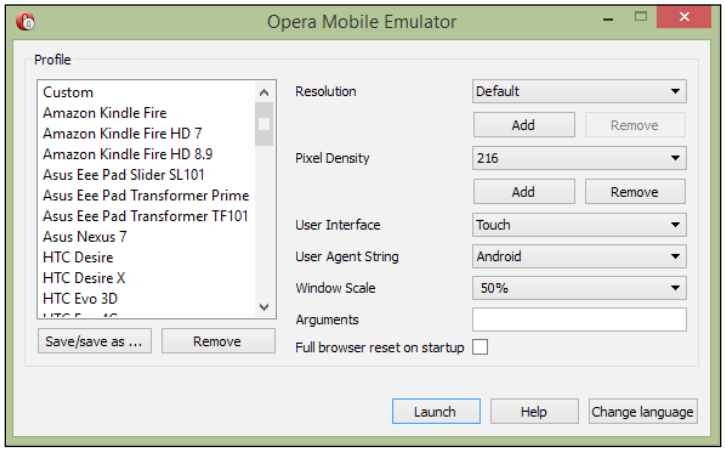
The following screenshot shows our profile application from *Chapter 3*, *The AngularJS Directive-based Approach* in the Chrome mobile emulator with the configuration model value for Nexus 5:



Opera mobile emulator

The mobile emulator is an offline tool developed by Opera Software. The tool can be found at `http://www.opera.com/developer/mobile-emulator`. It is a desktop application that supports many responsive parameters that are to be configured to simulate different device environments. It includes the device name, the resolution, and many other parameters.

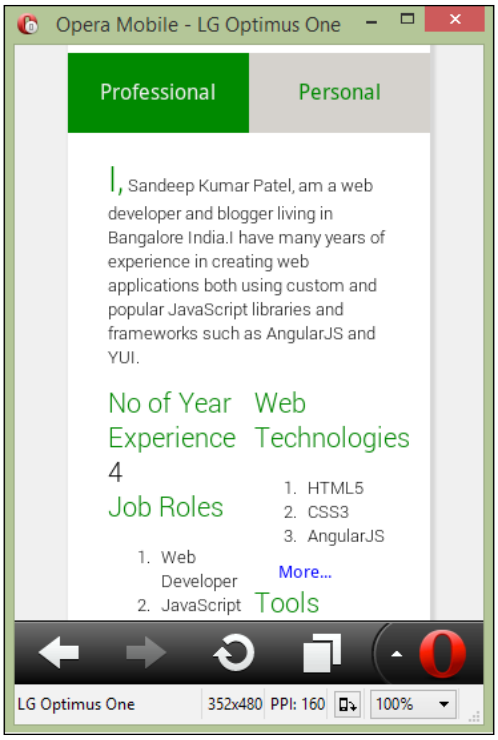
The following screenshot shows the initial windows when **Opera Mobile Emulator** starts. This window provides all the configuration parameters that can be customized to produce simulated devices:



The settings parameters include **Profile**, **Resolution**, **Pixel Density**, **User Interface**, **User Agent String**, **Window Scale**, and **Arguments**. The details of these settings parameters are as follows:

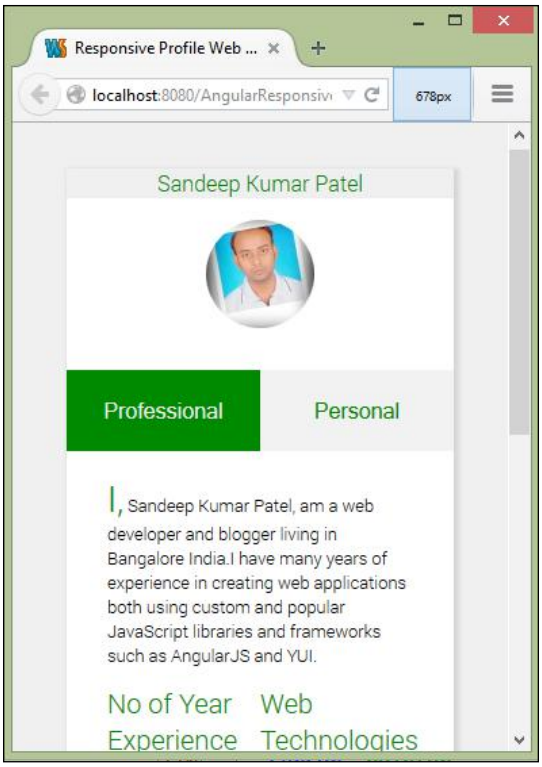
- **Profile**: This option holds a list of device names to be picked for emulation, for example, Amazon Kindle Fire and HTC desire
- **Resolution**: This option contains all the supported display resolution types in a drop-down list, for example QVGA and VGA
- **Pixel Density**: This option is used to configure pixel density for the emulator display area.
- **User Agent String**: This option is used for configure user agent strings in a drop-down list. It includes MeeGo, Desktop, Android, and default devices.
- **Window Scale**: This option has the value for window scale sizes in drop-down lists.
- **Arguments**: This option is used to pass additional parameters to the emulator before startup.

The following screenshot shows our profile application from *Chapter 3*, *The AngularJS Directive-based Approach* in the Opera mobile emulator with the profile value as LG Optimus One, resolution as HVGA, pixel density as 160 and user agent string as Android:



FireBreak add-ons

FireBreak add-ons are for the Firefox browser to debug a responsive application. You can install it from the Firefox add-ons list. You can also find it at `http://www.filipjohansson.se/firebreak`. After the installation of this add-on, you can see a small section at the top-right corner of the browser showing the current pixel size of the window. On resizing the window, the pixel value changes instantaneously showing the current size of the screen. The current version of FireBreak is 1.4 as of today. The following screenshot shows the profile application in the Firefox browser with FireBreak 1.4. You can see the FireBreak value of the screen is 678 px.



Summary

In this chapter, you learned how to debug AngularJS applications and about their scope properties. We explored some of the online and offline tools to test responsive applications. This was the last chapter of this book, and we covered all the choices that a developer has to build an AngularJS-based responsive application. Now you can start building your own responsive application.

Index

Symbols

A

B

C

D

E

F

G

I

J

M

O

P

R

S

T

W

Thank you for buying
Responsive Web Design with AngularJS

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 that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

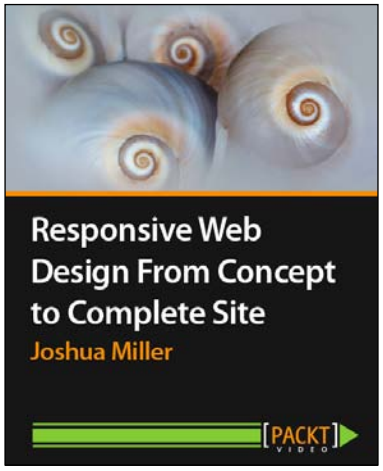
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced 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, then please 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



Responsive Web Design From Concept to Complete Site [Video]

ISBN: 978-1-78216-570-5 Duration: 02:04 hours

Easily design responsive websites that can adapt to any device regardless of screen size using HTML 5 and CSS3

1. Learn how to create fluid styles that flow to fill a browser of any size.
2. Discover the best design and coding practices in HTML5 and CSS3 for flexible layouts.
3. Contains everything you need to know to create simple-to-complex responsive sites starting from a design mockup to implementing it as a finished product.

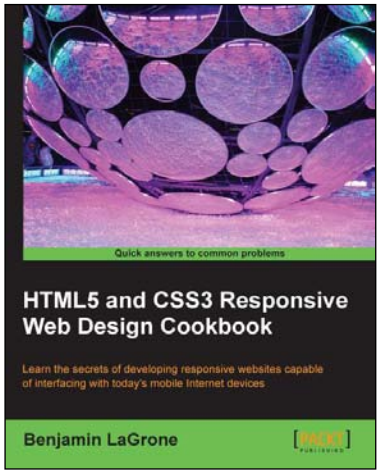
Responsive Web Design with HTML5 and CSS3

ISBN: 978-1-84969-318-9 Paperback: 324 pages

Learn responsive design using HTML5 and CSS3 to adapt websites to any browser or screen size

1. Everything needed to code websites in HTML5 and CSS3 that are responsive to every device or screen size.
2. Learn the main new features of HTML5 and use CSS3's stunning new capabilities including animations, transitions, and transformations.
3. Real-world examples show how to progressively enhance a responsive design while providing fall backs for older browsers.

[PACKT] open source
community experience distilled
PUBLISHING

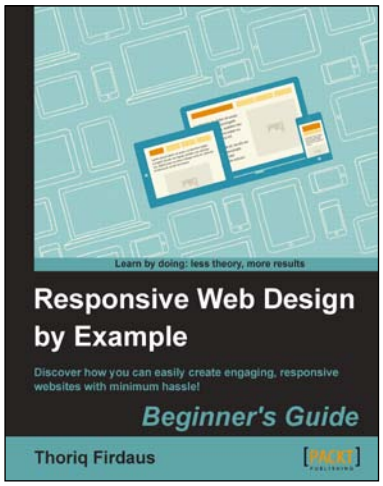


HTML5 and CSS3 Responsive Web Design Cookbook

ISBN: 978-1-84969-544-2 Paperback: 204 pages

Learn the secrets of developing responsive websites capable of interfacing with today's mobile Internet devices

1. Learn the fundamental elements of writing responsive website code for all stages of the development life cycle.
2. Create the ultimate code writer's resource using logical workflow layers.
3. Full of usable code for immediate use in your website projects.



Responsive Web Design by Example Beginner's Guide

ISBN: 978-1-84969-542-8 Paperback: 338 pages

Discover how you can easily create engaging, responsive websites with minimum hassle!

1. Rapidly develop and prototype responsive websites by utilizing powerful open source frameworks.
2. Focus less on theory and more on results, with clear step-by-step instructions, previews, and examples to help you along the way.
3. Learn how you can utilize three of the most powerful responsive frameworks available today: Bootstrap, Skeleton, and Zurb Foundation.