Material Design implementation with AngularJS

V. Keerti Kotaru

Apress®

Material Design Implementation with AngularJS

UI Component Framework First Edition

V. Keerti Kotaru

Apress®

Material Design Implementation with AngularJS

V. Keerti Kotaru
Hyderabad, Andhra Pradesh, India

ISBN-13 (pbk): 978-1-4842-2189-1 ISBN-13 (electronic): 978-1-4842-2190-7
DOI 10.1007/978-1-4842-2190-7

Library of Congress Control Number: 2016950454

Managing Director: Welmoed Spahr
Lead Editor: Pramila Balan
Technical Reviewer: Sathish VJ
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Prachi Mehta
Copy Editor: Brendan Frost
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

I dedicate the book to my true support system, my parents, Lakshmi and Rama Rao, and my wife, Sowmya.

bczajak@comcast.net

Contents at a Glance

bczajak@comcast.net

Contents

About the Author

Keerti Kotaru has been associated with various software development projects from 2002. He has acquired knowledge and expertise designing and developing web and mobile applications. In recent times he has used AngularJS and related JavaScript technologies extensively.

He has a Masters in Software Systems degree from the University of St. Thomas, Minneapolis/St. Paul, Minnesota, USA.

Keerti Kotaru is awarded Microsoft Most Valuable Professional (MVP) in 2016. He is a regular speaker and organizer for an AngularJS Hyderabad Meetup group (meetup.com/ngHyderabad). He is involved in technology activities and events for Google Developer Groups (GDG) Hyderabad. He presented multiple sessions for this group, including the annual events DevFest 2014 and DevFest 2015.

He has also presented sessions for TechGig, AngularJS Pune, and AngularJS Chicago Meetup groups.

He blogs at `http://bit.ly/kotaru`. Learn more about him at `http://bit.ly/keertikotaru`.

bczajak@comcast.net

About the Technical Reviewer

Sathish VJ is a technologist who is passionate about science and all sorts of technologies. Among other things as a full-stack engineer, he has previously been a front-end architect developing solutions using AngularJS and Angular Material. In the area of Angular, he is currently working on Angular2 and Ionic2 alongside related technologies.

bczajak@comcast.net

Acknowledgments

My ongoing journey with software development has been overwhelming and yet thoroughly enjoyable. Along the way, I have been trained and mentored by individuals, institutions, organizations, and last but not least, software developer communities.

ngHyderabad and GDG Hyderabad (Google Developer Groups) are two developer communities in my city that have had an enormously positive influence on me. The communities include engaging discussions, sharing of ideas, and proactive volunteering efforts from each member. They created an opportunity for showcasing my knowledge. This book is a direct result of my interactions and learning with the communities.

I thank CDK Global for providing initial direction, space, and the opportunity for learning the latest programming languages and tools.

CHAPTER 1

Introduction to Angular Material

Web application development has evolved in recent years. Earlier JavaScript was primarily used for form validations; to check if user provided data in required form fields or if a phone number followed a pattern and so on. With the advent of Ajax, integration with services (on server) is possible without reloading the whole page. Due to more powerful client machines and browsers, rich front-end development is done with JavaScript, HTML, and CSS.

Multiple frameworks including AngularJS helped organize JavaScript code better. AngularJS is a superheroic JavaScript framework that enables developing an application using MV* framework (Model-View-Controller and its variants).

It is a similar story with HTML and CSS. HTML is now customizable. It is possible to create elements, attributes, or CSS classes in markup and reuse. CSS3 has been powerful with advanced styling and animation capabilities.

Scenarios

As capabilities to develop rich applications using HTML, JavaScript, and CSS got better, there arose a variety of scenarios and hence challenges. Consider the following use cases.

Multiple form factors: With the advent of mobile technologies, there are a variety of screen sizes. Content needs to fit multiple of screen sizes and still be legible.

Rich UI development: With new capabilities in HTML, CSS, and JavaScript, it is possible to develop better user interactions, controls, and user experience.

Earlier browser plug-ins were used for rich UI (User Interface). Flex, Silverlight, or a similar plug-in needed to be installed on the browser. Many times these were heavy and took time to load. Browser understands HTML, JavaScript, and CSS and the plug-ins were another layer on top of the browser.

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-2190-7_1) contains supplementary material, which is available to authorized users.

V. K. Kotaru, *Material Design Implementation with AngularJS*,
DOI 10.1007/978-1-4842-2190-7_1

Single Page Application (SPA): Many applications tend to be developed as SPAs. With this approach, the whole page does not always have to reload. Rather, it dynamically gets content for sections of the page and renders the UI. This helps improve page performance and avoid unnecessary network calls and browser activity. In addition, the user does not lose context between pages.

More Power, More Responsibility

As more logic moves to the client or browser, there is a need for better processes and quality. HTML, CSS, and JavaScript code organization needs to be better. Code reusability is more important than ever.

Design Patterns: A pattern could be identified in a problem. For quite a long time, web applications have had a variety of design patterns implemented. A common scenario with web applications has been that

1. It has multiple screens or views. They could be forms, reports, or widgets.
2. Data objects (also called model) in the application need to be associated with UI controls. As user edits data in UI controls, changes need to reflect in the object and vice versa.
3. These changes to the data need to be persisted and propagated to data store and other views.
4. Application of logic, validations, and in some cases, transformation of data to a different type of object has to happen.

A design pattern MVC (Model-View-Controller) is apt for these problem statements or requirements. MVC and its variants (MVVM [Model-View-View-Model] and MVP [Model-View-Presenter]) are widely used.

I am bringing up an opinionated scenario here. However, in my experience I have seen this happen many times.

Unit Testing: It has great influence on code quality. Teams find Test-Driven Development (TDD) very effective. Such practices need to be applied to JavaScript code as well.

Dependency Injection (DI): This is important for loose coupling among artifacts. An object not instantiating its dependencies allow caller to swap implementation. This makes object immune to changes in dependency.

Better unit testing is a major advantage of this approach. When we unit test code, we should focus only on the current object, not its dependencies. Unit tests should not invoke end-to-end calls. All dependencies should be mocked or stubbed to invoke a dummy implementation. This is made possible with DI.

Angular Material inherits goodness from these concepts as it is built using AngularJS. These concepts are not directly related to Angular Material. Hence, we do not go into an in-depth discussion on these topics. However, teams and individuals developing applications using AngularJS (including Angular Material) can take advantage of the features.

What Is Material Design?

Material Design is a visual language that aims to provide consistent experience across devices, screen, sizes, and form factors. The term Material is analogous to real-world paper and ink. Yet, it is open to capabilities of digital world.

3D aspects of Material Design are a result of studying light and shadow. They convey feeling of surface and real-world material. Motion and animations are integral to Material Design. They maintain continuity between user actions and screen changes. They also provide subtle feedback to the user.

Refer to the following URL for Material Design spec. It is a living document and expected to be updated often. http://www.google.com/design/spec/material-design/introduction.html#introduction-principles.

Why Material Design?

Google went into great detail trying to develop the visual language. They have observed material objects, light, shadows, and so on and came up with a design specification, which to an extent resembles real-world materials and interactions.

On Material Design, there is quite a bit of adoption already. Multiple products including many of Google's own applications have used Material Design concepts.

Imagine, for an application in development, that using such a system allows you to take advantage of current state. Many users are already acquainted with the design and hence intuitively relate to screen layout, transitions, positioning of elements, and so on. The concepts have evolved for some time. The new application has a great starting point instead of reinventing the wheel.

Recently talking to an UX expert, I have realized there are two approaches to UX design.

1. *Reuse existing design guidelines, UI controls, and components.* This allows one to develop fast. Users readily understand the app. More importantly, the app team has high focus on real application logic rather than user experience. Part of the puzzle is already solved.

2. *Develop something new.* Invest great energy and money developing a completely new user experience. UX is not easy. It needs experts and the subject is a science unto itself.

   Nevertheless, with this approach your application will be unique. A tech-savvy audience might like such an approach. However, there is always a learning curve. The general population might fail to understand the concept.

While both approaches have pros and cons, using Material Design, we are likely to fall into the former category.

Why Angular Material?

For implementing Material Design on AngularJS application, Angular Material is a great choice. It is a Google open source project. It provides ready-made controls and services for Material Design.

There are advantages using AngularJS. It is a comprehensive framework that provides the following:

1. Routing, which allows associating a view (and controller) to a route in a SPA
2. Karma for Unit Testing
3. DI
4. Services and factories for encapsulating functionality
5. Data binding between model objects and views

Angular Material Basics

Here is a high-level overview of Angular Material.

Theming

Theming is important for providing consistent look and feel across the application and feeling of brand to the application. One of the important aspects of theming is colors. The two colors we care about in Material Design are primary and accent colors. Primary colors best fit title bars, status bars, and so on, while accent colors aim to grab attention: they are bright. It could be a button at a corner of a page, slider control's knob, and so on. Angular Material also defines "warn" colors, which are used for warning and error messages. This means that the user needs to be careful about making a choice (in the context of the application): for example, an alert to check if user really wants to delete a record.

Angular Material theming allows definition of these colors, which convey the meaning of the brand along with consistency.

Layout

Angular Material uses adaptive layout. Content on the screens adjust to various screen sizes and resolutions. It is a classic problem while developing an application for multiple form factors and screen sizes. A table of data with eight columns might be readable on laptop screen but would make no sense on a mobile screen. The table needs to wrap around and get rid of secondary information on a mobile screen. Adaptive design addresses this problem. Angular Material uses flexbox, a layout mode in CSS3.

Typography

With Angular Material, CSS classes define text font and size (see Figure 1-1). They provide consistent look and feel across the application. Out of the box, they are confined to Material Design specification for Typography.

Figure 1-1. *Sample typography classes in Angular Material CSS. Reference: Angular Material website:* `https://material.angularjs.org`

■ **Note** Material Design spec for typography: `www.google.com/design/spec/style/typography.html`

Directives and Services

We use various directives and services for controls and functionality in Angular Material. These are at the heart of Angular Material. They provide ready-made Material Design features and functionality to the application.

■ **Note** All Material Design directives and services are prefixed with *md*.

CHAPTER 2

Getting Started

This chapter discusses multiple options for getting started with a project based on Angular Material. Along with referencing Angular Material and AngularJS JavaScript libraries, the chapter will elaborate on development environment setup, package manager options, and so on. In addition, the chapter will focus on setting up the environment for ES5 as well as ES2015 (ES6). The approaches described in the chapter should help a medium or large project setup.

Scripts

The following scripts are required for running an Angular Material application:

1. **AngularJS**: Primary dependency, AngularJS framework library.
2. **Angular Animate**: AngularJS animations library.
3. **Angular ARIA**: ARIA (Accessible Rich Internet Applications) provide state and semantic information for tools used by persons with disabilities. ngAria in AngularJS provides out-of-the-box support and improves default accessibility of the application.

Angular Material scripts:

4. **Stylesheet**: the angular-material.css provides CSS classes and styles for Angular Material.
5. **Angular Material** library.

Optional dependencies:

6. **Angular Messages** is for showing messages and errors within the HTML templates. It includes the module *ngMessages*. Typically, the module is used while performing client-side validations for a form. Include this library when directives and other artifacts that are part of the *ngMessages* module are used in the application.

V. K. Kotaru, *Material Design Implementation with AngularJS*,
DOI 10.1007/978-1-4842-2190-7_2

7. **Angular Sanitize** is to sanitize HTML by escaping tags keyed into input elements (or by other means). The library includes the module *ngSanitize*. Include the library when the service, provider, or filter that is part of the *ngSanitize* module is used. Not a mandatory dependency.

Code Editor/Integrated Development Environment (IDE)

All the concepts discussed in this book are implemented using JavaScript, HTML, and CSS. To get started, we can even code in a simple code editor like Notepad. However, a good code editor helps with better code formatting, editing features, autocomplete function signatures and elements, debugging aspects, and so on.

The following are recommended to use. They are easy to install, free for non-commercial use, and installable on both Windows and Mac.

1. Sublime Text
2. Visual Studio Code
3. Atom
4. Chrome Dev Editor

Get Started with Angular Material

Use this section to set up a sample code repository for trying out Angular Material samples on your machine. Practice various code samples demonstrated during course of this book.

To get started, create an empty directory and open it in a code editor of your choice. Create a new file and name it index.html. Next, run through the following steps.

1. Code "Hello World—Angular Material."
2. Set up a developer class web server and run the sample.

Let us look into each step in detail.

Step 1: Code "Hello World—Angular Material"

Let us get started by referencing all needed libraries for an Angular Material application. The easiest way to reference needed JavaScript libraries is by using a CDN (Content Delivery Network) URL. Refer to the preceding "Scripts" section for a list of required libraries.

Copy-paste the following code in the index.html we just created.

```
 <!DOCTYPE html>
<html>
<head>
```

```
    <!-- Reference Angular Material stylesheet -->
    <link rel="stylesheet" href="https://cdn.gitcdn.link/cdn/angular/bower-
    material/v1.0.8/angular-material.css">
        <title>Hello World</title>
</head>

<body ng-app="sampleApp" layout="column">

    <!-- Bootstrap Angular Material Application. We create a module with
    ngMaterial as a dependency. Learn more about an AngularJS module in a
    later section of the chapter -->
    <script>
        angular.module("sampleApp", ["ngMaterial"]);
    </script>

    <!-- Create a title: Here we are creating a simple toolbar. We will get
    into details of various controls in Angular Material during course of
    this book. For the moment, understand that it is an Angular Material
    element for creating a title and a toolbar-->
    <md-toolbar layout-padding>
        <div class="md-toolbar-tools">
            <h2>Welcome to Angular Material</h2>
        </div>
    </md-toolbar>

    <!-- Reference needed scripts. See scripts section above for details on
    each script-->
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/
    angular.js"></script>
    <script src="https://ajax.googleapis.com/ajax/libs/
    angularjs/1.4.8/angular-animate.min.js"></script>
    <script src="https://ajax.googleapis.com/ajax/libs/
    angularjs/1.4.8/angular-aria.min.js"></script>
    <script src="https://ajax.googleapis.com/ajax/libs/
    angularjs/1.4.8/angular-messages.min.js"></script> <!—Angular Messages
    is optional for this sample -->
    <script src="https://cdn.gitcdn.link/cdn/angular/bower-material/
    v1.0.8/angular-material.js"></script>

</body>
</html>
```

At the time of writing this book, version 1.5.6 for the first four scripts and 1.0.9 for Angular Material were the most recent. Look for a newer version while trying your samples and preferably use the latest version.

Step 2: Set up a developer class web server and run the sample

We need to run samples on a local web server. As a developer running samples on a laptop/desktop, we do not need the high-end features of a web server. The following are easy to install and lightweight options. You may choose to follow these instructions and set up on a developer machine, or you can run very well an existing web server that you are comfortable with.

Option A: Live Server

Live server autorefreshes the browser window as we make changes to the files in the editor. Often, by the time we switch to the browser for results, will see the window updated with the latest code.

Prerequisite: Make sure Node Package Manager (NPM) is already installed on the machine. Refer to Setup Node Package Manager Section, later in the chapter for details.

Install: Live Server is available as an NPM package. Install by using the following command.

```
npm install -g live-server
```

Note that -g option installs the package globally on the machine. It allows using the package from any directory without performing another local install.

On a Mac or Linux machine, you might need sudo access to install the package globally. Consider running the command as follows:

```
sudo npm install -g live-server
```

On a Windows machine, run command prompt as administrator.

Run the app

Open terminal (or command prompt) and CD (change directory) into the newly created directory with index.html.

Next run the following at the prompt.

```
live-server
```

This should run the live server on current directly and open the default browser with index.html loaded. See Figure 2-1 for the result.

> *At this point we are all set. We can use this setup to code more samples demonstrated during the course of this book and run them.*

Furthermore, notice that it defaults to port 8080. To see all available options with live server, run

```
live-server -h
```

Option B: Serve

If you are uncomfortable with live reload of web pages, consider using serve. Effectively, it is similar to live server without the live reload option.

Prerequisite: Make sure NPM is already installed on the machine. Refer to Setup Node Package Manager Section, later in the chapter for details.

Install: Serve is available as an NPM package. Install by using the following command.

```
npm install -g serve
```

Note that the -g option installs the package globally on the machine. It allows using the package from any directory without performing another local install.

On a Mac or Linux machine, you might need sudo access to install the package globally. Consider running the command as follows:

```
sudo npm install -g serve
```

On a Windows machine, run command prompt as an administrator.

Run the app

Open terminal (or command prompt) and CD into the newly created directory with index.html.

Next run the following at the prompt.

```
serve
```

This should serve current directly and its files on an HTTP URL for browsers. Notice that it defaults to port number 3000. Open the browser of your choice and run localhost:3000. See Figure 2-1 for the result.

> *At this point we are all set. We can use this setup to code more samples demonstrated during the course of this book and run them.*

Furthermore, to see all available options with serve, run

```
serve -h
```

Option C: IIS Express

If you are on a Windows machine, you might consider this option. Download and install the latest version of IIS Express from Microsoft's download page: www.microsoft.com/en-us/download. Run through the setup wizard to install IIS Express.

Typically, IIS Express will be installed on "C:\Program Files\IIS Express" or "C:\Program Files (x86)\IIS Express."

Run the app

Run the following command:

```
C:\IIS-installed-location\iisexpress /path:C:\your-sample-folder /port:3001
```

3001 is a sample port number. Run on any available port number.

Files in your sample folder now can be accessed over an HTTP URL. Open the browser of your choice and run http://localhost:3001. See Figure 2-1 for the result.

> *At this point we are all set. We can use this setup to code more samples demonstrated during the course of this book and run them.*

For more information on IIS Express run,

```
C:\IIS-installed-location\iisexpress /help
```

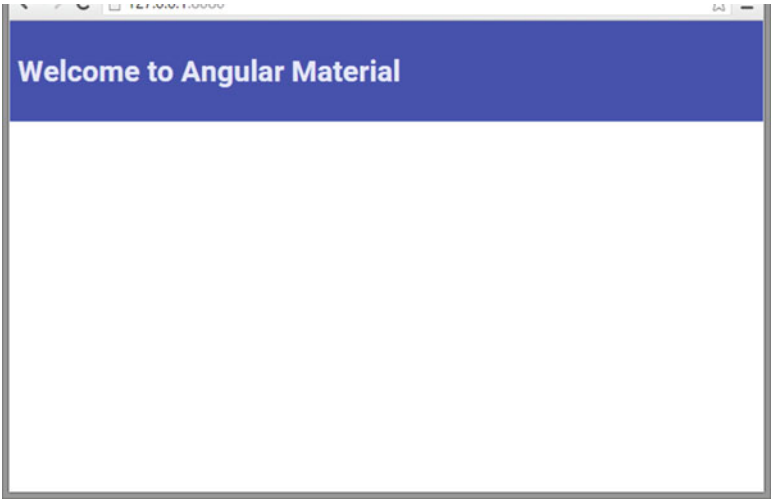

Figure 2-1. *Rendered output for "Hello World—Angular Material"*

Working with Code Samples

All code samples developed for this book are available at this GitHub URL: https://github.com/kvkirthy/Angular-Material-Samples. Clone or download the repository.

Run Samples

Use a web server of your choice. There are three options—live server, serve, and IIS express—described in the preceding section. Run the web server at root level. As we open samples in a browser, will see a layout depicted in Figure 2-2.

Figure 2-2. *Code samples*

On the left, in the navbar, notice a complete list of chapters. Links to individual samples are on the right. Click the link; the sample opens in a new window.

Folder Structure

Open the downloaded samples' code repository in an IDE of your choice.

The code samples follow a simple directory structure. You will see a directory for each chapter. Browse through code samples for each chapter, under its folder. See Figure 2-3.

For many samples, within the chapter, a folder named "app" has JavaScript code (modules, controllers, etc.).

All the libraries are included in the bower_components folder at the root level.

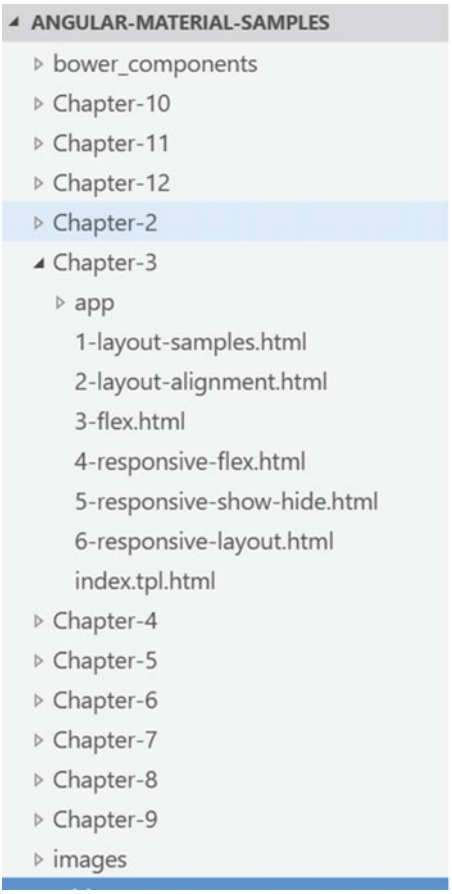

Figure 2-3. *Code samples folder structure*

AngularJS Basics

This section describes AngularJS basics. It is intended to provide context for understanding Angular Material. If you are familiar with AngularJS, you should be good to skip this section and move on to the next section.

As mentioned earlier, AngularJS is an MVW framework. That is, Model-View-Whatever. Effectively we can use AngularJS to implement Model-View-Controller or Model-View-View-Model patterns. In either case a model (or view model) is in the browser. It is a JSON object, which could be bound to a view or an HTML template. A property in the model JSON object could be bound to a control like text field, drop-down, radio button, and so on.

Data Binding

AngularJS supports two-way data binding. A change made to the model is updated in the view. If view makes changes, they are updated in the model.

Consider the following sample. It has a text field and a label. A model object *aModelObject* is bound with a text field and a label in "strong" HTML element. Curly braces are used for writing the value of an object or an expression directly in HTML. Using a directive *ng-init*, we initialize the variable *aModelObject* with an initial value. We use another directive *ng-model* for binding a value with an HTML element, in this case an input tag.

```
<div ng-app>
<span ng-init="aModelObject='Initial Value'"></span>
 <input type="text" ng-model="aModelObject">
 <div>
   <strong>{{aModelObject}}</strong>
 </div>
</div>
```

Initially, the text field and the label show "Initial Value" as they are bound to the model object (aModelObject). See Figure 2-4.

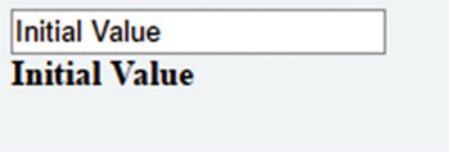

Figure 2-4. *Initial value as the controller loads*

As we change value in the text field (change triggered in the UI), the model object is updated. And hence, the label shows the new value. See Figure 2-5.

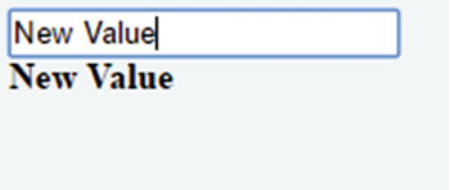

Figure 2-5. *As user edits the field, the label is udpated*

Directive

AngularJS directives allow creating custom HTML elements, attributes, comments, and CSS classes. In the preceding example, we used three built-in directives.

1. ng-app: Used for initializing the AngularJS application.
2. ng-init: Used for initializing a variable or model object.
3. ng-model: Used for binding model and view.

There are many built-in directives, and we can create custom directives specific to an application. Explore more about directives at docs.angularjs.org/guide/directive.

■ **Note** All built-in directives are prefixed with ng-.

AngularJS Module

Logical units in AngularJS app could be grouped together to create a module. Every application bootstraps with one module. All other modules would be its dependencies or dependencies of dependencies.

Create a module using the following application programming interface (API).

```
var myModule = angular.module("sampleApp",["module1", "module2"]);
```

Find out more about modules at https://docs.angularjs.org/guide/module.

DI

DI helps with loose coupling among code units, functions, and objects. AngularJS uses DI quite effectively. The following are some of the code artifacts we could create in AngularJS:

1. Service
2. Factory
3. Provider
4. Value
5. Constant
6. Controller
7. Filter

In an example, a service could be injected in a controller or another service or a factory. During this process, AnguarJS ensures that a given object is ready for use and that API or functions on the object may be invoked. For example, every controller needs *$scope* object injected. Variables and objects on *$scope* are accessible in the HTML view.

Controller

It binds view (HTML template) and model objects created in JavaScript together. In the earlier example, we initialized and used a model variable in HTML. It might work for simple flags. However, for complex data representations, which could also be obtained from a server-side API, we need to use JavaScript code.

A controller could be a JavaScript function. Create a controller using the following API.

```
myModule.controller("firstController", function($scope){
                $scope.title = "Select a City";
                $scope.cities = ["San Francisco", "New York", "Bengaluru",
"Mumbai", "Hyderabad"];
            });
```

Here the first parameter is used as name of the controller. The controller definition is the second parameter, a callback function. This function has been injected with a scope object, $scope.

Scope (*$scope*) provides context and is used to hold model objects. A JavaScript variable or an object could be set on scope. It will be available to use on view or HTML template. In this example, title could be used in associated HTML template.

Find out more about controllers at https://docs.angularjs.org/guide/controller.

View / HTML template

HTML templates (markup) renders the view or the UI. The user interacts with the elements in the view. We use various AngularJS directives and filters for providing additional power and functionality to HTML.

Consider the following code and explanation of the HTML template.

```
<div ng-app="sampleApp">
    <div ng-controller="firstController">
        <h2>{{title}}</h2>
        <select name="dpCities" id="dpCities" ng-model="selectedCity" ng-
        options="city for city in cities"></select>
    </div>
</div>
```

1. As for the earlier sample, use *ng-app* to bootstrap Angular. Here we specify the module name. Hence, it is the main or root module. All other modules are dependencies of this module or dependencies of dependencies.
2. Use a directive *ng-controller* and set the context of controller within a div tag.
3. The title is shown as an h2 element. “title” is a variable on scope bound on the UI as a label.

4. Notice the select element with *ng-options* directive. The controller's scope has list of cities. It is being set to the drop-down using the *ng-options* directive. The selected value will bind to a variable *selectedCity*.

Services

Services are reusable code artifacts in AngularJS. They are singleton objects and maintain state once the application bootstraps.

In an example, code to make HTTP API calls is encapsulated in a built-in service called $http.

■ **Note** All built-in services in AngularJS are prefixed with $

We can create our own services using the following API. Consider the following code.

```
myModule.service("sampleService", function(){
  // sample service definition.
});
```

The *service*() API creates a new service. The first parameter is the service name. The second parameter is a callback function with definition of the service.

The "sampleService" could be used anywhere in the module (and in other modules with the current module as a dependency).

Provider

A provider is very similar to service. In fact, a service is a type of provider. We create a provider in a special case, which is a function or JSON object required while bootstrapping the module.

AngularJS module has an API "config," which accepts a callback function. It is invoked only once, while bootstrapping the application (as the browser loads the app).

A provider is a specialized object which could be injected/used in a config function. A provider is expected to have a *$get* function. When a provider is injected/used in another service or a controller, only the code in *$get* is exposed. The *$get* function is a factory. It is expected to return an object (or a function) that could be used in a service.

However, while it is used in a *config* function, API (functions) and fields on "this" object could be used.

Consider the following sample. It defines a provider with a function on *this* object and *$get* factory function.

```
myModule.provider("aSample", function(){
    this.aProviderFunction = function(){ // This could be invoked directly
    in a config function
             return "Provider Function";
         };
```

```
        this.$get = function(){ // This is used in a factory, service, or
        controller.
            console.log("$get invoked. A factory function");
            return this.aProviderFunction;
        };
});
```

When the provider is injected into a controller, *$get* is invoked. In the current sample, it prints "$get invoked. A factory function."

As mentioned earlier, the $get function acts as a factory. It returns a function for use in the controller.

```
// Provider injected in the controller.
myModule.controller("sampleController", function($scope, aSample){
                console.log(aSample()); // use the function returned by $get
            })
```

When it is injected into *config* function, whole provider object and its API are accessible. Consider the following code. As we call *aProviderFunction()*, it prints the returned string, "Provider Function" on console.

```
myModule.config(function(aSampleProvider){
    console.log(aSampleProvider.aProviderFunction());
});
```

Learn more about providers here: https://docs.angularjs.org/guide/providers.

Making the Code Minification Safe

Unlike function parameters, objects may be injected in any order. The object is identified by its name. When the JS is minified, the variables are renamed and DI no longer works. The following syntax solves this problem. It injects an object using its name as a string.

Here is the syntax we used earlier.

```
myModule.controller("firstController", function($scope, aSample){
        // Controller definition
});
```

To make it minification safe, use the following array syntax.

```
myModule.controller("firstController", ["$scope","aSample",function(scope,
sample){
        // Controller definition
});
```

Notice parameters on the function that are named scope and sample. They could be named anything now. DI uses string values provided before the function. As the string values are untouched during DI, it is minification safe. DI will not break with minified code as well.

Learn more about DI here: https://docs.angularjs.org/guide/di.

Pakage Managers and JavaScript Modules

Setup Node Package Manager - NPM

NPM is a popular package manager. Many open source developer tools, JavaScript libraries, and frameworks are available as NPM packages. Hence, it has become a one-stop source for downloading such packages.

A package manager downloads and installs a given package and all of its dependencies at once. As a consumer of the package, we do not need to keep track of dependencies or separately download or install them.

NPM is part of a larger NodeJS offering. Download the node installer from nodejs.org. For the purposes of this book, we will primarily be using NPM.

You can install a package by running the following command:

```
npm install <package name>
```

It downloads the package (and its dependencies) to node_modules folder in the current directory. As an option, -g helps install a package globally on the machine. Use this option to install packages that are environment specific and are not directly related to the code base; for example, task runners, tools, web servers, and so on. This helps make project folder self-contained and not polluted with things beyond the application.

Often, we need elevated access to run the npm install with -g option. Hence use sudo on a Mac or Linux machine (that is sudo npm install -g <package name>). On a Windows machine, run command prompt as administrator.

The preceding section "Get Started with Angular Material" explains the easiest implementation for referencing required JavaScript libraries and writing the JavaScript code. The following are more sophisticated and effectively help to set up a medium- to large-scale JavaScript project.

Download Angular Material using NPM

Angular Material is available as an NPM package. To get started with NPM, create a new directory and open a command prompt or terminal at this directory. Install Angular Material with the following commands:

```
npm install angular-material

It downloads Angular Material and its dependencies to node_modules
directory.
```

■ **Note** Creating an NPM package for the sample will help maintain repository and dependencies in one place. This may not be a necessity for the unsophisticated sample we are working on. However, this process saves time and energy for a big and complex code repository.

```
npm init
(Initializes an NPM package)

npm install angular-material –save
(Saves Angular Material as dependency of current project in package.json, so
that next time just running npm install downloads all given packages)
```

Reference scripts

Create an index.html in the newly created directory (at the level of node_modules). Reference the downloaded scripts. References could be script tags (and link tags) in index.html or any module loader like RequireJS (Asynchronous Module Definition - AMD implementation), CommonJS, and so on.

Let us use the simplistic approach and include scripts in index.html

```
<link rel="stylesheet" type="text/css" href="node_modules/angular-material/
angular-material.min.css">

<script type="text/javascript" src="node_modules/angular/angular.min.js"></script>

<script type="text/javascript" src="node_modules/angular-animate/angular-
animate.min.js"></script>

<script type="text/javascript" src="node_modules/angular-aria/angular-aria.
min.js"></script>

<script type="text/javascript" src="node_modules/angular-material/angular-
material.min.js"></script>
```

Download Angular Material using Bower

Bower is a popular package manager for front-end artifacts. It is optimized for front-end libraries and scripts. It is lightweight due to its flat dependency tree. While using ES5 (current JavaScript version at the time of writing this book), bower is a good package manager to use for front-end libraries.

Install bower globally on your machine with NPM. This is a one-time activity. Subsequently, as we download more packages using bower on the same machine, running this step will not be required again.

```
npm install -g bower
```

To avoid running into access issues, consider running the preceding command with sudo (that is, sudo npm install -g bower) on a Mac or Linux machine. Run the command prompt as administrator on a Windows machine.

Then use bower to download Angular Material.

```
bower install angular-material
```

It downloads the entire Angular Material library and all of its dependencies under bower_components folder. See Figure 2-6.

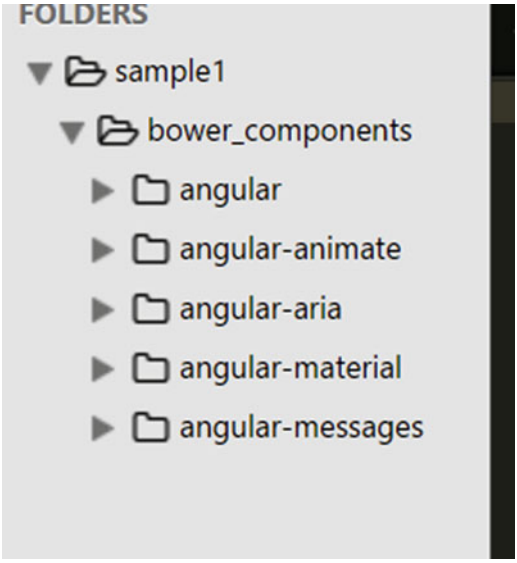

Figure 2-6. *Angular Material and its dependencies downloaded with bower*

■ **Note** Creating a bower package for the sample will help maintain repository and dependencies at one place. This may not be a necessity for the unsophisticated sample we are working on. However, this process is a must and saves time and energy for a big, complex code repository.

```
bower init
(Initializes a bower package)

bower install angular-material –save
(Saves Angular Material as dependency of current project in bower.json, so
that next time just running bower install downloads all needed packages)
```

Reference scripts

Create an index.html at root folder of the project. Reference the downloaded scripts in bower_components. References could be script tags (and link tags) in index.html or any module loader like RequireJS (AMD implementation), CommonJS, and so on.

Let us begin with the simplistic approach and include scripts in index.html.

```
<link rel="stylesheet" type="text/css" href="bower_components/angular-
material/angular-material.min.css">

<script type="text/javascript" src="bower_components/angular/angular.min.
js"></script>

<script type="text/javascript" src="bower_components/angular-animate/
angular-animate.min.js"></script>

<script type="text/javascript" src="bower_components/angular-aria/angular-
aria.min.js"></script>

<script type="text/javascript" src="bower_components/angular-material/
angular-material.min.js"></script>
```

SystemJS & JSPM (JavaScript Package Manager)

SystemJS is a good module loader until browsers support the ES2015 way of importing them. This is one step in the right direction. As browsers start supporting new format, SystemJS gets out of the way easily. SystemJS understands existing JavaScript module loaders like RequireJS or CommonJS and of course the ES6 module loader.

JSPM is a package manager for SystemJS-based system. It is a node package. Install it with the following command.

```
npm install jspm -g
```

■ **Note** This is a one-time command that installs JSPM globally on the machine (with -g option). For future package installations, this step need not run again.

However, JSPM could install locally to the project. Replace -g option with --save-dev option to save it as a Dev dependency of the project's package.

To install Angular Material and dependencies, use the following command.

```
jspm install angular-material
```

■ **Note** We could configure to use Babel, Traceur, or TypeScript transpilers. While setting up the package for the first time, JSPM will prompt to choose a transpiler.

Install CSS plug-in for loading CSS files.

```
jspm install css
```

Code "Hello World—Angular Material"

Add references to SystemJS and its configuration. *JSPM install* command downloads packages to *jspm_packages* folder. It downloads SystemJS as well, which supports multiple module formats and allows loading them.

```
<script src="./jspm_packages/system.js" type="text/javascript"></script>
<script src="./config.js" type="text/javascript"></script>
```

The following script in Index.html will load main file (root)—main.js in app folder. Every other file that loads is a dependency of this file or dependencies of its dependencies.

Import function loads main file and its dependencies asynchronously. It returns a promise, which is resolved once all files load.

```
<script type="text/javascript">
        System
        .import('app/main')
        .then(() => console.log("Angular Material Sample loaded
successfully"))
</script>
```

■ **Note** The then function called on resolving the promise uses arrow function syntax of ES2015.

Config file has reference paths to modules and script files:

```
"angular": "jspm_packages/github/angular/bower-angular@1.5.0",
"angular-animate": "jspm_packages/github/angular/bower-angular-
animate@1.5.0",
"angular-aria": "jspm_packages/github/angular/bower-angular-aria@1.5.0",
"angular-material": "jspm_packages/github/angular/bower-material@1.0.5",
```

Import these scripts in main.js. Here we are using ES6 syntax. It is transpiled to ES5 format by Babel or Traceur.

```
import 'angular';
import 'angular-animate';
import 'angular-aria';
import 'angular-material';
```

Controller: export the function and register it with AngularJS as a controller. Similar to the previous example, it has a single data element on $scope, message.

controller.js

```
export default function($scope){
     $scope.message = "Hello World";
};
```

"*export default*" is ES2015 syntax for modules. The given function is exported and available for all that import the current module.

Consider main.js or root file that acts as starting point to the application in JavaScript.

main.js

```
import 'github:angular/bower-material@1.0.5/angular-material.min.css!';
import 'angular';
import 'angular-animate';
import 'angular-aria';
import 'angular-material';
import controller from './controllers';

angular.module('es6Sample', ['ngMaterial'])
.controller('firstController', controller);
```

This file imports all of its dependencies. Each dependency might have more dependencies.

As for the ES5 sample, create a new Angular module, *es6Sample*. Add ngMaterial dependency for Angular Material services and directives. Import controller function from controllers.js and register with the module.

■ **Note** Angular Material CSS file loads with the help of CSS plug-in.

Limit Scope Using Closure

Consider the following coding practice. If you are not using any module loader in JavaScript, this is a good alternative. For the purposes of this book, it is only for reference. The code sample in the "Bower" section of Chapter 2 uses this approach.

The start point for the app is *main.js*, and *controller.js* has the first sample controller. Consider code for the following two files. In this sample, a coding style that helps to avoid creating global variables and objects is shown. A global variable could be accessed across the application, causing unforeseen behavior and hence resulting in bugs. We limit the scope by writing code in a self-executing function: self-executing because code needs to run as soon as script loads. All variables declared in it have local scope, accessible only within the function.

main.js

```
// ngMaterialSample is an Angular module used across the application. Hence
it is intentionally a global variable.
var ngMaterialSample = (function(angularRef){
        'use strict';
        return angularRef.module('ngMaterialSample', ['ngMaterial']);
})(angular);
```

Here, *angular* (comes from angular.min.js script) is passed-in as a parameter. We create a new Angular module *ngMaterialSample*. It has dependency on *ngMaterial*. This will enable using Angular Material directives and services in the application.

controller.js

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

        app.controller("sampleController", function($scope){
                $scope.title = "Welcome to Angular Material";
        });

})(ngMaterialSample); // ngMaterialSample is the module object to use for
creating a controller.
```

In controller.js (it should load after main.js), pass-in ngMaterialSample global object as a parameter. We create a controller in module/self-executing function. Controller has one data element, *title* on *Scope*.

Include the following scripts in index.html

```
<script type="text/javascript" src="app/main.js"></script>
<script type="text/javascript" src="app/controller.js"></script>
```

***Index.html - template to show title*:**

```
<body ng-app="ngMaterialSample">
        <div ng-controller="sampleController" >
                <md-toolbar>
                        <h2>{{title}}</h2>
                </md-toolbar>
        </div>
</body>
```

ng-app is set on body tag. The *ngMaterialSample* module bootstraps here. Controllers (and other artifacts) of this module could be used on any child elements of body tag. The sampleController is scoped to *div* element.

md-toolbar is an element/directive in *ngMaterial* module. ngMaterial is referenced as a dependent module for *ngMaterialSample*. The directive helps render Material Design–style toolbar on the page.

■ **Note** In AngularJS, a directive helps create custom DOM elements, attributes, CSS classes, or comments. More often, with directives we can provide custom functionality to HTML elements or attributes. There are many directives available out of the box with the framework. We can create our own custom directives as well.

Notes on ES2015 (Also Called ES6)

Everyone in the JavaScript world is excited about ES2015. It has great language features, inbuilt module support, classes, arrow functions, better variable scope management, and so on.

However, we are not fully there yet (at least at the time of writing this book). Browser support is growing, but it will be a while before all current browsers run ES2015 out of the box.

To start using ES2015 features on unsupported browsers, we could use transpilers like Traceur, Babel, or TypeScript.

Angular Material features demonstrated in this book are based on Angular 1.x version. It is still based on ES5 JavaScript. Angular2 can fully take advantage of ES2015 (ES6). Having said that, we can code in ES2015 today. Use transpilers and convert it to ES5 so that browsers can run the code.

Summary

This chapter aims to provide various options to get started with Angular Material. It not only focuses on ways to reference Angular Material and its dependencies in the project but also helps set up the project. It aims to look beyond samples demonstrated.

At the time of writing this book (2016), a JavaScript project should brace itself for migrating to ES2015 (ES6). The language features in the newer version of JavaScript are too good to ignore. With such migration in the context, SystemJS and JSPM fit the bill. Create your Angular Material project with SystemJS and JSPM the approach described in the chapter. SystemJS supports multiple module formats. AMD (RequireJS) and Node-style CommonJS are today's famous module systems. ES2015 has come up with new syntax and features for module loading. SystemJS supports these module formats.

If you are planning on a small-scale Angular Material project, it is possible that such a setup could be overwhelming. Get started with Google CDN or the bower approach. It is easy to begin and saves time up front.

References

For live server NPM package, see `https://www.npmjs.com/package/live-server`
For Serve NPM package, see `https://www.npmjs.com/package/serve`

CHAPTER 3

Layout Management

In this chapter, we will explore layout management and styling aspects. The focus of this chapter will be markup or templates. We will also explore aspects of responsive design and adjusting view and screen content based on screen resolution.

For the project setup, continue to use a sample created in the previous chapter. Of the multiple approaches detailed, it does not matter which you followed.

Flexbox

Angular Material uses CSS3 Flexbox for responsive design. In the HTML markup, combining Angular Material directives and Flexbox CSS we can build rich UI. This combination (Angular Material & Flexbox) distinguishes between layout management and styling. HTML *attributes for layouts* and *CSS for styles* helps with clear separation of concerns. The Angular Material library provides attribute directives, which are primarily used for layout management.

Layout

With Angular Material, *layout* is one of the basic attributes required for layout management. It is one of the many directives provided by the framework. It transforms to apply Flexbox CSS classes while rendering the HTML page.

We can provide two possible values to the directive (attribute): "*row*" and "*column*." Any element with a row layout applied on it will align child elements horizontally. It will behave as a single row. Similarly, the element with the layout value *column* will align child elements vertically. We can chain these layouts to get the desired layout structure.

Note In a later chapter, Angular Material–styled input elements (input text boxes, drop-downs, etc.) have been discussed. For simplicity and focus on layouts, this chapter uses basic HTML input elements.

V. K. Kotaru, *Material Design Implementation with AngularJS*,
DOI 10.1007/978-1-4842-2190-7_3

Consider the following code:

```
<div layout="column">
        <textarea rows="5" cols="50" placeholder="text field 1"></textarea>
        <textarea rows="5" cols="50" placeholder="text field 2"></textarea>
</div>
```

This will result in two text areas aligned vertically, in a column. See Figure 3-1.

Figure 3-1. *Vertically aligned when layout is set to the value "column"*

Change the layout value to *row*, will and you will see the text fields aligned next to each other—horizontally, in one row. See Figure 3-2.

Figure 3-2. *Horizontally aligned when layout is set to the value "row"*

```
<div layout="row">
        <textarea rows="5" cols="50" placeholder="text field 1"></textarea>
        <textarea rows="5" cols="50" placeholder="text field 2"></textarea>
</div>
```

Use a combination of layout options to get desired results. For example, align title on top and form fields under it. Column has two elements: title (h2) and the form (div). Form is a row, and it in turn has more elements like labels and input boxes. See Figure 3-3.

```
<div layout="column">
        <h2>Welcome to Angular Material Sample</h2>
        <div layout="row">
                <strong flex="15">Enter first name</strong>
                <input flex="20" type="text"/>
        </div>
</div>
```

Figure 3-3. *A layout arrangement*

Layout-Align

Manages horizontal and vertical alignment of child elements. It works in combination with *layout a*ttribute value. Possible values are *start*, *center*, and *end*. Provide two values. Depending on layout, the first value is either horizontal or vertical. That is, if layout is column, then the first value of layout-align is vertical, and the second value is horizontal. It flips if layout is row. See Figure 3-4 and Figure 3-5.

Figure 3-4. ***layout="column"*** *layout-align="start end"*

Figure 3-5. ***layout="row"*** *layout-align="start end"*

Consider the following sample, aligning child element to bottom-right.

```
<div layout="column" style="min-height: 500px; background-color: skyblue"
layout-align="end end" >
        <img src="/img/angular-logo.png">
</div>
```

■ **Note** Consider using *space-around* or *space-between* values for the first parameter. It is applied horizontally or vertically based on layout value.

More Layout Attributes

layout-padding	**Provides padding around the element**
layout-fill	**Fills the available space in the container**
layout-wrap/layout-nowrap	**Control wrapping content of an element to next line**
layout-margin	**Provides margin for the element**

Flex

Flex lets content adjust to the layout container element (a parent element with layout value row or column). Content can grow and shrink to fit the available space. Specify required behavior on the flex attribute. See Figure 3-6 for the result.

Consider the following code. This demonstrates how flex fills available space, based on a given container's layout.

```
<!-- Following row has two text fields. Both together fill the row -->
<div layout="row" layout-padding>
        <input flex type="text" placeholder="Text box 1 in a row"></input>
        <input flex type="text" placeholder="Text box 2 in a row"></input>
</div>

<!-- Following row has just one text field. Whole row is taken by the text
field-->
<div layout="row" layout-padding>
        <input flex type="text" placeholder="Only text box in a row">
</input>
</div>

<!-- Here container is a column. Hence second element moves to the next row
in the column-->
<div layout="column" layout-padding>
        <input flex type="text" placeholder="Text field 1 in a column">
</input>
        <input flex type="text" placeholder="Text field 2 in a column">
</input>
</div>
```

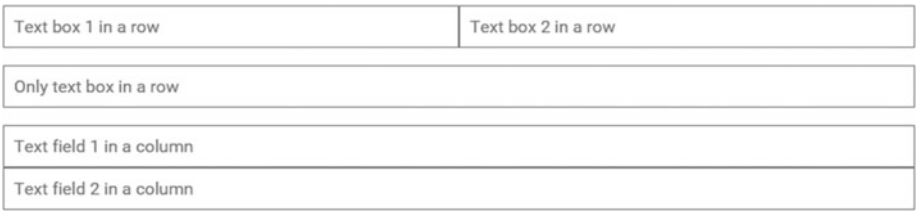

Figure 3-6. *Flex streching and skewing with available space in the layout*

To the flex attribute (directive), you can provide a numeric or a set of predefined values. The following are the possible flex values.

flex	Merely specifying flex attribute on an element allows it to grow or shrink as needed.
none	Do not grow or shrink.
nogrow	Do not grow. However, can shrink.
noshrink	Do not shrink. However, can grow.
initial	Can shrink. Set to initial height and width.
auto	Can grow and shrink. Set to initial height and width
Numeric values	Numbers between 0 and 100 (the value is considered to be a percentage). Only multiples of 5 and values 33 and 66 are allowed.

Consider the following code sample. A text field for a person's name is expected to take less space than a complete address. Use a smaller number like 33 for name. Use 66 for the address. See Figure 3-7.

Figure 3-7. *Using flex values 33 and 66*

■ **Note** On the HTML element, if you apply flex=“33”, it results in 100/3. And flex=“66” results in 200/3. Two elements with flex=“33” and flex =“66” together will take up 100% space. Sum of the two numbers is 100.

```
<div layout="row" layout-padding>
        <input flex="33" type="text" placeholder="Provide full name">
</input>
        <input flex="66" type="text" placeholder="Provide complete
address"></input>
</div>
```

■ **Note** The preceding specified percentages and relative space each control occupies are the same even when you resize the window (or view the page on a smaller resolution screen). The layout remains similar.

When you need controls to adapt and move around, so that they are usable on a smaller screen (mobile phones and tablets), CSS breakpoint alias in combination with Angular Material directives could be used. We will look at them later in the chapter.

Responsive Design

As described in previous chapter, web front-end is no more about developing for desktop or laptop screens. Applications need to render on much smaller screens like tablets and mobile phones. Of late, many devices are touch enabled. It is an important factor while presenting and interacting with the content.

Next few chapters explore the following scenarios, challenges, and solutions.

Real Estate

Imagine a big table of data, half a dozen columns and tens of rows. It provides snapshot of information on a laptop screen. It is quite useful to see the big picture and take full advantage of the available real estate. However, the same table on a mobile phone is not so useful. The user cannot read all the information at one shot. Columns and cells will not be legible.

The table need to realign and resize to show in one column. Show only the important information and hide nitty-gritty details. When user taps on the cell and navigates to details screen, provide more information.

Feedback for User Actions

On mobile devices, feedback for user actions like tap (click) on a button becomes even more important. The user is directly interacting with the content. The user is touching the content. Feedback that a button was pressed (with a tap) needs to be instantaneous and apparent.

Screen transitions between views need to be intuitive for the user. Imagine that the user tapped on a Create Document button. If the action results in navigation to a new form screen, it needs to be obvious for the user. Animation should depict that tapping on the button is resulting in transition.

■ **Note** Do not attempt to solve user feedback with ngTouch and Ionic Framework library functions in Angular Material. Version 1.x has issues integrating with those libraries. It might change in the future. Make sure to understand fully before attempting to integrate with the two libraries.

Breakpoints

Angular Material has CSS breakpoint alias defined for certain screen resolutions (primarily screen width). It helps define layout, alignment, and other CSS behavior for a given screen resolution. This is a powerful feature for responsive UI development.

Use breakpoint alias in combination with other Angular Material directives and attributes. Examples are provided later in the chapter.

The following are the default breakpoints provided by Angular Material.

Breakpoint Alias	Resolution	Description
xs	Screen width less than 600 pixels (not equal to)	Very small screen. Most mobile phone screens come under this category.
gt-xs	Any screen with width equal to or above 600px	Greater than extra-small. Use this breakpoint when we need to code for all non-mobile phone screens. Eliminates mobile phone screens. Include everything ranging from tablets, laptops, desktops, or even TVs.
sm	Screen width ranging from 600px (including 600px) to 960px (excluding)	Small screen. This breakpoint addresses most tablet screens in portrait mode. Often a tablet tilted in landscape mode is greater than small.
gt-sm	Screen width greater than or equal to 960px	Greater than small. Include all devices greater than small. This excludes tablets in portrait mode and mobile phones. It includes tables in landscape mode, low-resolution laptop/desktop screens, high-resolution laptop/desktop screens, and even high-definition TVs.
md	Screen width ranging from 960px (including) to 1280px (excluding)	Medium-size screen. Most laptops in landscape mode and low-resolution laptops/desktops come under this category.
gt-md	Screen width greater than or equal to 1280px	Greater than medium. Excludes tablets (in either landscape or portrait mode), low-resolution laptop/desktop screens. This breakpoint addresses high-resolution laptop/desktop screens and TVs.
lg	Screen width ranging from 1280 (including) to 1920px (excluding)	Large screens. This breakpoint addresses most laptop/desktop screens.

(continued)

Breakpoint Alias	Resolution	Description
gt-lg or xl	Screen width greater than or equal to 1920px	Greater than large screens or extra-large screens. This breakpoint excludes most laptop, tablet, and mobile phone screens. All super-high-resolution desktop/laptop screens and high-definition TV screens come under this category.

Use breakpoints along with any layout/alignment directives or attributes (layout, flex, etc.). Suffix breakpoint alias (sm, gt-sm, etc.) to the directive name. Most directives have implementation to support breakpoints. For building responsive UI, built-in breakpoints are quite useful.

Consider the following code. Among the three div elements, the first and last are for providing margins to the page. Margins collapse on a medium or smaller screen. That allows content to take complete available space on a smaller screen. See Figures 3-8 and 3-9 for results. Read comments for details on code.

```
<div layout="row" flex layout-padding>
        <!-- flex-gt-md=10. Flex 10% on a greater than medium screen -->
        <div flex-gt-md="10"></div>

        <div flex-gt-md="80" flex>
        <!-- flex fills available space by default. For a screen greater
than medium fill up to 80% -->
                Lorem ipsum dolor sit amet, consectetur adipiscing elit.
                <!-- Saving space. Can have paragraphs of information here.
-->
        </div>

        <!-- flex-gt-md=10. Flex 10% on a greater than medium screen -->
        <div flex-gt-md="10"></div>
</div>
```

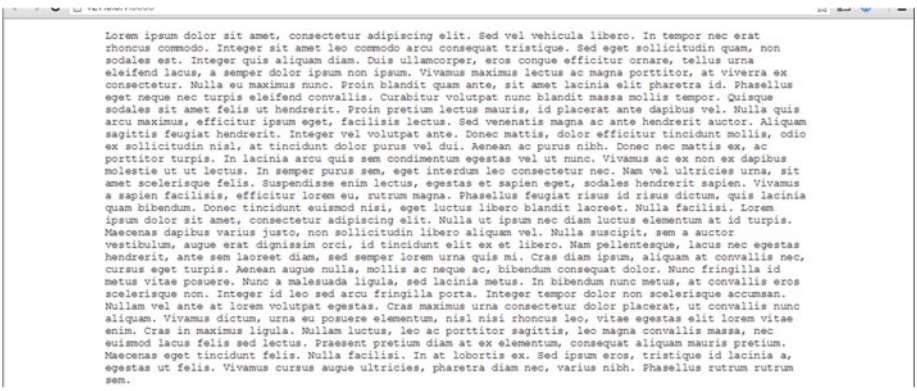

Figure 3-8. *Large screen (gt-md)*

On a small screen, emulating screen width of 414px, left and right div elements collapse. Space around the paragraph of text is with layout-padding directive. It stretches to 100% of available screen width (flex).

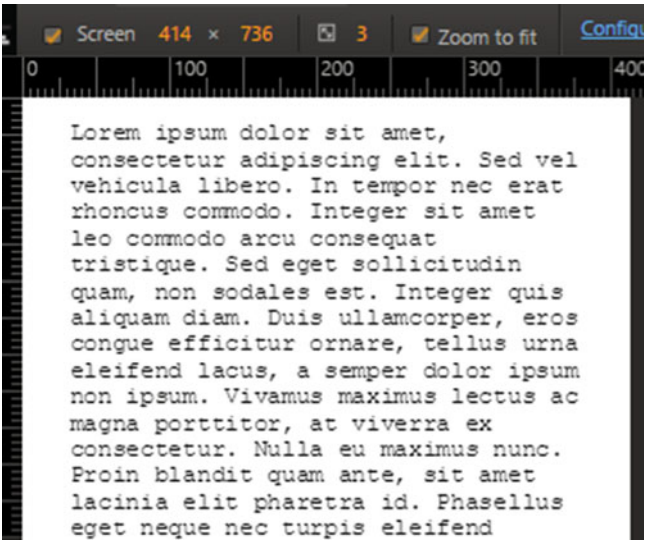

Figure 3-9. *Small screen (emulated to 414px width)*

■ **Note** At the time of writing this book, *layout-padding* and *layout-margin* do not support breakpoint alias. Expect enhancement to support alias with the directives in future releases. Here is the GitHub issue tracking progress: https://github.com/angular/material/issues/1984

Show/Hide

Use show or hide directives to display or hide elements in markup. It is similar to ng-show or ng-hide. However, show or hide could be used with breakpoint alias.

Consider the following code:

```
<div layout="row" layout-padding>
       <!-- show only on screens greater than small -->
       <div show-gt-sm hide>
              <img src="/img/angular-logo.png">
       </div>
       <div>
              <strong>AngularJS</strong>
```

```
                <div>Superheroic JavaScript framework</div>
                <div>Website: angularjs.org</div>
                <div>Twitter: @angularjs</div>
        </div>
</div>
```

The element is laid out as a row. The first cell on the left shows an image. On a smaller screen, hide the image div element. Show on all screens larger than small (>= 960px). Always show the second element. It shows textual information that fits even a small screen. See Figures 3-10 and 3-11.

Figure 3-10. *On a screen gt-sm*

Figure 3-11. *On an xs (extra-small) screen (emulated to a mobile phone)*

Responsive Layout

Another regular use case would be to fill page horizontally (grow the row) on bigger screens. On a mobile screen with smaller width, skew the page to show content in a column.

Consider the following code: it is laid out as a row on gt-sm breakpoint (greater than small, which is a screen at least 960px wide). On the other hand, it is skewed to a column on smaller screens. See Figures 3-12 and 3-13.

```
<div layout="column" layout-gt-sm="row" layout-padding>
        <div>
                <strong>AngularJS</strong>
                <div>Superheroic JavaScript framework</div>
                <div>Website: angularjs.org</div>
```

```
                <div>Twitter: @angularjs</div>
        </div>
        <div>
                <strong>Angular Material</strong>
                <div>AngularJS and Material Design</div>
                <div>material.angularjs.org</div>
                <div>Twitter: @angularjs</div>
        </div>
</div>
```

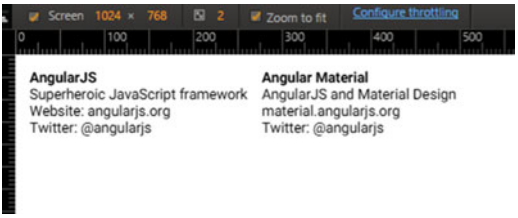

Figure 3-12. *On a screen gt-sm*

Figure 3-13. *On an emulated small screen*

Summary

This chapter detailed out basics of layout management and responsive design in Angular Material. The focus of this chapter has been HTML markup. We used various attribute directives for layout management.

Angular Material uses CSS Flexbox. It encapsulates layout-related CSS classes in attribute directives. As a developer, while managing layout, we use Angular Material API and provide values for attributes.

Angular Material achieves better separation of concerns by using attributes and values in an HTML element for layout and CSS classes for styling. Under the hood, it transforms given directive values to elements with CSS classes applied.

The next set of chapters will focus on sophisticated code samples. We explore Angular Material services and directives.

References

See Angular Material official documentation at https://material.angularjs.org/latest/layout/children

For information on Flexbox styles for layout features, refer to the following URL: https://gist.github.com/ThomasBurleson/88152ec57c9133dec57a

For issue status on layout margin and layout padding support for breakpoints, refer to the following: https://github.com/angular/material/issues/1984

For AngularJS documentation and logo (used in samples): https://angularjs.org/

CHAPTER 4

Navigation & Container Elements

This chapter will detail some more Angular Material directives. We will primarily deal with navigation and container elements. This chapter helps build the skeleton for the application.

What are directives? Reiterating an AngularJS concept, directives help create new HTML elements.

In HTML, we use various elements (or tags) for describing, formatting, and managing the behavior of the content on the web page. For example, elements like h1 and h2 define headers in the web page. A tag "strong" applies bold style on text, useful for emphasis. Use input elements for various UI controls and components like buttons, textboxes, drop-downs, and so on.

If we need to create new elements in addition to what we already have, directives are the way to go in AngularJS. This is one place to manipulate DOM. From the best-practices point of view, directives could access and manipulate DOM directly. (It is not preferred to directly deal with DOM elsewhere.)

Many times, they are packaged as UI components, which could be reused across the application.

What are services? In AngularJS, Services are reusable JavaScript objects. They can be injected into controllers and other services.

Note All Material Design directives are prefixed *md-*. Services are prefixed *$md*.

Content (md-content)

This directive is a container element for workspaces in the application. The content this directive holds could be text, images, and/or other controls user interacts with. It applies styles that allow content to scroll. Optionally add layout-padding attribute to the directive for padding a little bit of empty space around margins.

V. K. Kotaru, *Material Design Implementation with AngularJS*,
DOI 10.1007/978-1-4842-2190-7_4

Usage

```
<md-content layout-padding>
    Welcome, This is the scrollable content on the workspace.
    s...
</md-content>
```

■ **Note** This directive makes content inside scrollable. If it is nested with other scroll-enabled directives, it could create multiple scroll bars on the page (which is a bad user experience). Hence it is advised to use md-content as a sibling of other containers and not nest them.

Toolbar (md-toolbar)

Often, the toolbar presents the page title. It describes the purpose of the page or screen. It could have one or more page-level actions.

Usage

The following is a very basic usage of the directive. See Figure 4-1.

```
<md-toolbar layout-padding class="md-toolbar-tools">
    <h2>Page Title</h2>
</md-toolbar>
```

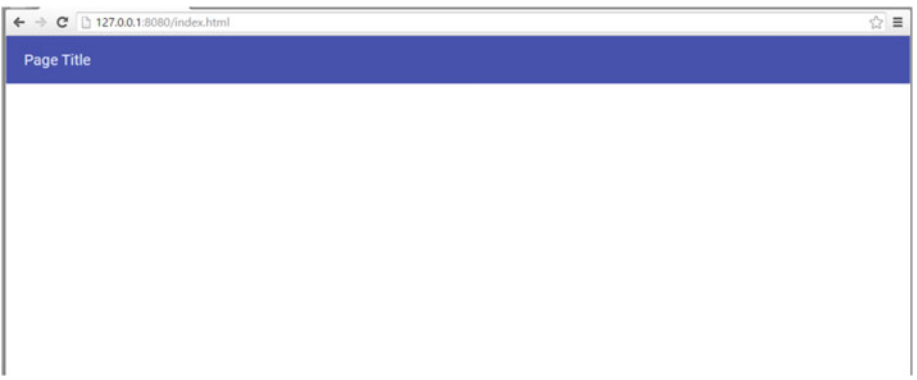

Figure 4-1. *Basic toolbar*

Let us now explore more toolbar features.

1. Apply CSS class *md-tall* for larger toolbar as the page loads. The other possible value is *md-medium-tall*.
2. Use attribute md-scroll-shrink="true". The user scrolling the page will collapse the title bar and allow more workspace to be seen.
3. Optionally, use CSS class md-toolbar-tools-bottom for better scroll experience. It vertically aligns titles and buttons to the bottom of the toolbar. As user scrolls up, blank space on the toolbar collapses, leaving the title and actions still visible.
4. We may also use CSS classes *md-warn* or *md-accent* for showing warning or accent colors for the toolbar. Chose them such that if you are alerting the user, you may use warn color.
5. Group all toolbar items under an element with CSS class "md-toolbar-tools".

Actions: Page-level actions take their place in the toolbar.

6. Use a flex element with buttons followed by it, so that they are aligned right. Use *md-button* directive for the Material Design button.
7. For icon buttons, use CSS class *md-icon-but*ton. This will adjust the button to size of the icon. It won't be wide like a text button. Use *md-ico*n element (directive) in md-button.

Here is the complete sample. See Figure 4-2.

```
<md-toolbar md-scroll-shrink="true" layout-padding class="md-tall md-
toolbar-tools-bottom">
    <span flex></span>
    <div class="md-toolbar-tools">
        <md-button class="md-icon-button">
            <md-icon md-svg-src="/img/ic_menu_black_24px.svg"></md-icon>
        </md-button>
        <h2>Page Title</h2>
        <span flex></span>
        <md-button>An Action</md-button>
    </div>
</md-toolbar>
```

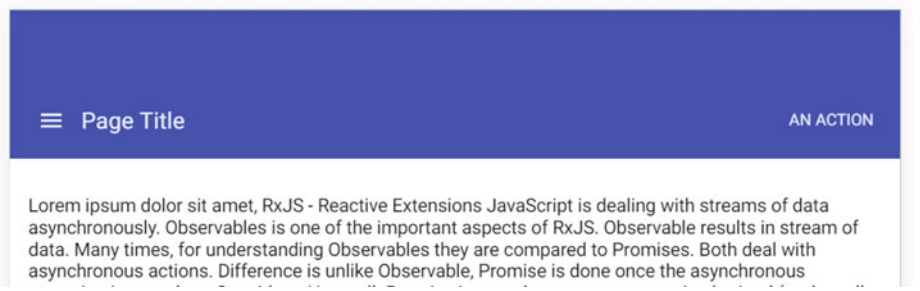

Figure 4-2. *A tall toolbar*

■ **Note** There are better ways to use the md-icon directive for using icons in the applications (be it a button or not). A later Chapter 10 in the book details icons in Angular Material.

Sidenav (md-sidenav)

Sidenav, a navigation control, is often seen in mobile and desktop web applications. A typical sidenav is used to provide a site-map. It lists actions and links to various functionalities in the application. We could expand and collapse the sidenav on a need basis.

Basic Usage

```
<md-sidenav class="md-sidenav-left">
        <!-- toolbar specific markup goes here -->
</md-sidenav>
```

Notice *md-sidenav-left* CSS class. On the page, it aligns sidenav to the left. Use the CSS class *md-sidenav-right* for it to align on the right.

By default, the navbar will not show. Use attribute *md-is-open="true"* for it appear straightaway. Attribute value could also be an expression that returns a Boolean.

```
<md-sidenav md-is-open="true"  class="md-sidenav-left">
   <md-toolbar layout-padding class="md-medium-tall">
        <h2 class="md-toolbar-tools">All Actions</h2>
   </md-toolbar>
</md-sidenav>

<md-content layout="row">
   <div layout-padding >
            Lorem ipsum
                ...
    </div>
</md-content>
```

In the samples so far, sidenav is in parallel to *md-content*. It has its own toolbar with a title. Navbar appears on top of the content (like a menu) and hides if user clicks anywhere in the workspace. Consider Figure 4-3. Click anywhere on the text; it hides.

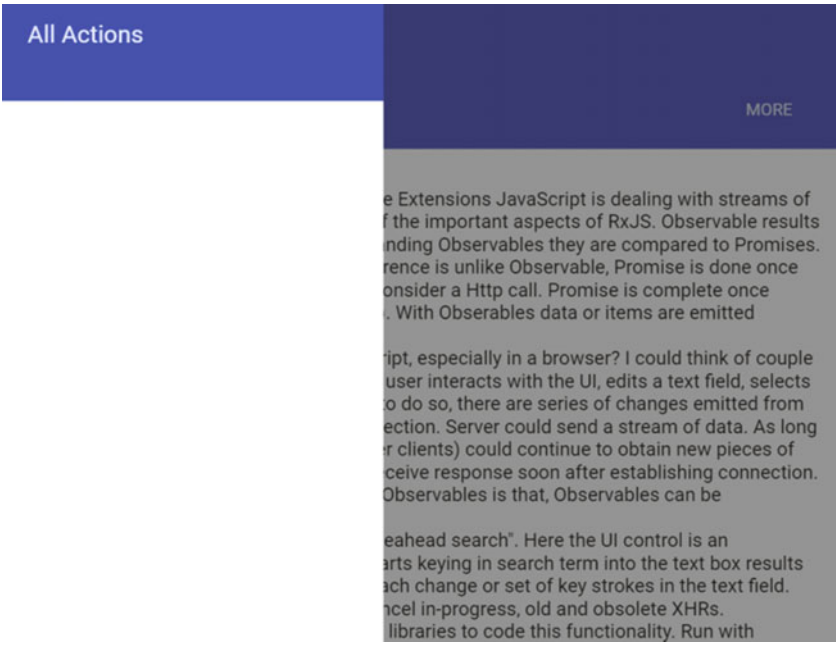

Figure 4-3. *Sidenav overlapping the page content*

Sidenav Along with the Content

Consider using *md-is-locked-open*. It takes an expression as the input. When true, clicking away (in the workspace) doesn't close the sidenav. Preferably, make sidenav part of the workspace. When locked open, it does not override the content and will show the sidenav on the side, along with the content. See Figure 4-4.

Consider the following code snippet. As it is part of the workspace now, unlike the earlier sample, it is preferred to be coded within the *md-content*. Also use row layout for sidenav to appear in the content.

```
<md-content layout="row">
   <md-sidenav md-is-locked-open="true" flex class="md-sidenav-left">
     <h4>side nav content</h4>
    </md-sidenav>
```

```
    <div layout-padding flex="80">
       Lorem ipsum
        ...
    </div>
</md-content>
```

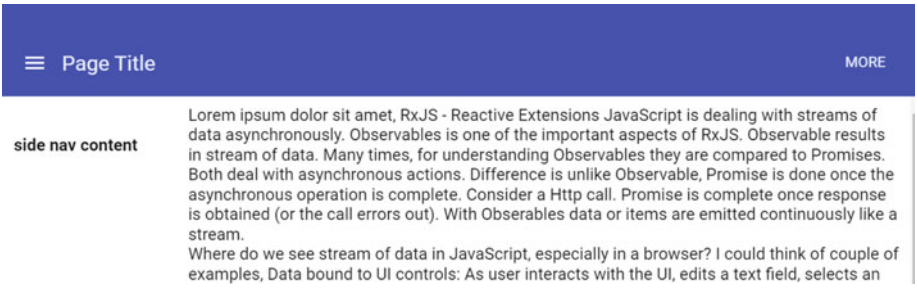

Figure 4-4. *Sidenav along with the page content*

Show/Hide Sidenav On Demand

Another common scenario is to use the menu button on the top-left, clicking which will show/hide sidenav. Consider the following code snippet.

```
<md-sidenav md-is-open="showLeftSidenav"  class="md-sidenav-left">
   <md-toolbar layout-padding class="md-medium-tall">
        <h2 class="md-toolbar-tools">All Actions</h2>
   </md-toolbar>
</md-sidenav>
```

Notice that *md-is-open* is using a variable on scope, *showLeftSidenav*. Unlike the first sample, it is dynamically showing or hiding sidenav. Consider the following controller code.

```
$scope.showLeftSidenav = false;

$scope.toggleLeftSidenav = function(){
    $scope.showLeftSidenav = !$scope.showLeftSidenav;
};
```

In the HTML template, the menu button in the toolbar calls (ng-click) the function *toggleLeftSidenav*. It flips current value. Shows if sidenav is hidden and or not. Here is the code for menu button in the toolbar.

```
<md-button class="md-icon-button" ng-click="toggleLeftSidenav()"
  <md-icon md-svg-src="/img/ic_menu_black_24px.svg"></md-icon>
</md-button>
```

Responsive—Show/Hide Sidenav Based on Screen Width

Instead of opening or closing the sidenav upon the click of a menu button in the toolbar, we may choose to dynamically decide by the screen size. If there is enough space for sidenav, show it. On a smaller screen, hide it by default.

Consider the following code. It uses a service *$mdMedia*, which evaluates media query and returns true/false for a given screen size.

```
 <md-sidenav md-is-locked-open="$mdMedia('gt-sm')" flex="20" class="md-
sidenav-left" layout-padding>
   <h4>side nav content</h4>
 </md-sidenav>
```

It shows the navbar for a screen greater than small (greater than 960px width).

$mdSidenav Service

$mdSidenav is another related service. A controller function could use API on the service. The following is the API.

1. It is possible to have multiple sidenavs on the page. In the controller select a specific sidenav using the service $mdSidenav('sidenavId1').

   Provide id in the HTML markup, <md-sidenav md-component-id='sidenavId1'>

2. $mdSidenav('sidenavId1').open(): shows sidenav with id sidenavId1.

3. $mdSidenav('sidenavId1').close(): collapses sidenav wih id sidenavId1.

■ **Note** The *open* and *close* functions return a promise. The promise is resolved once the respective action is complete. A promise allows asynchronously perform another action only after open or close operation.

---

4. $mdSidenav('sidenavId1').isOpen(): returns true if sidenav is open.

5. $mdSidenav('sidenavId1').isLockedOpen(): returns true if sidenav is locked open.

Tabs

Tabs are one way of categorizing content on a page. See Figure 4-5. They provide high-level organization of view elements and an easy way to switch to and fro among them. They create a sense of grouping. A logical unit of information, for example, text, images, video, and so on, could be in a tab.

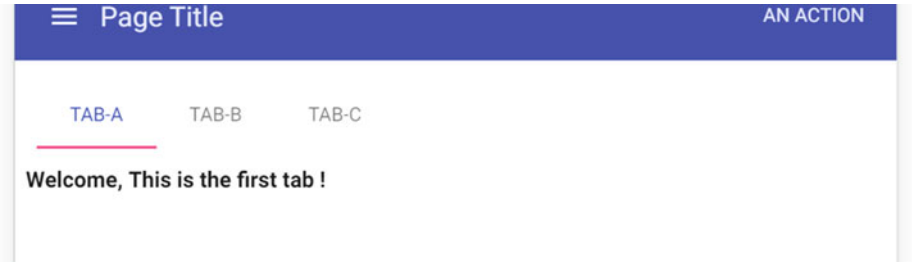

Figure 4-5. *Sample tabs*

The Angular Material component for tabs provides a subtle ripple effect as feedback on click. It is easy to identify the selected tab. It also shows animation while switching between tabs, which provides a sense of navigation.

Angular Material tabs are built with the following directives.

1. **md-tabs**: holds all tabs together.
2. **md-tab**: each tab's content and title are encapsulated in this element/directive.
3. **md-tab-label**: optional element. Useful if tab title is not simple text and needs additional markup.
4. **md-tab-body**: used for separation of title from tab's content. This element is mandatory only if *md-tab-label* is used. It allows separation of tab content from the title.

Usage

The following is basic usage of tabs. We will make it more sophisticated by using many other features of these directives.

```
<md-content layout-padding flex>
    <md-tabs>
        <md-tab label="Tab-A">
            <h4>Welcome, This is the first tab !</h4>
        </md-tab>
        <md-tab label="Tab-B" >
            <h4>There you go, I stand second</h4>
        </md-tab>
        <md-tab label="Tab-C">
            <h4>At last, you got to me.</h4>
        </md-tab>
    </md-tabs>
</md-content>
```

Now, let us explore the *md-tabs* directive.

1. **md-selected:** Use *this* attribute to select a different tab by default. Provide tab index value (0 based). In the preceding example, the following will select second tab.

   ```
   <md-tabs md-selected="1">
   ```

2. **md-stretch-tabs="always":** will let tab titles take full horizontal space. In the preceding example, the clickable area for "tab-a", "tab-b", and so on stretches horizontally to take up all available space. Other possible values are

   a. auto: stretched to take full width on a smaller screen like mobile phone and tablet in portrait mode. It does not stretch tabs on a mobile screen in landscape mode or a desktop screen (greater screen width).

   b. never: will not stretch tabs on any screen size. Their size is decided by the space taken by the label.

3. **md-center-tabs:** With horizontal space available, it aligns tabs to center. If we use auto or never for md-stretch-tabs, preferably set *md-center-tabs*.

4. **md-swipe-content:** It allows user to swipe on content of the tab right or left to move to next or previous tabs. For a better mobile/table or any other touch screen experience, set md-swipe-content.

■ **Note** A cosmetic aspect: for any attribute with Boolean value (like *md-center-tabs* or *md-swipe-content*) providing the attribute on parent directive is good enough. It will set the value to *true*. The following does the same task.

```
<md-tabs md-center-tabs>
<md-tabs md-center-tabs="true">
```

If you need to provide a value *false* to the attribute, simply do not use the attribute.

md-dynamic-height: Tabs' content area has fixed height. If content goes over it, will show a vertical scrollbar within the page. The attribute *md-dynamic-height* lets tab adjust height dynamically. If the tab content is even more than the page height, will only show one vertical scrollbar for the whole page. No scrollbar will show for the tab.

5. **md-no-pagination:** Tab titles could grow beyond the available horizontal space. Default behavior will show pagination arrow icons to move left/right. Setting *md-no-pagination* will jam tab titles to fit all tabs in the available horizontal space.

6. **md-align-tabs="bottom":** *A*ligns tabs at the bottom of the control (directive). Default value is "*top*".

7. **md-border-bottom:** Shows a separator below tab titles.

Consider the following code for enhanced tab. See Figure 4-6.

```
<md-tabs md-selected="1" md-stretch-tabs="auto" md-align-tabs="bottom" md-
dynamic-height md-border-bottom md-center-tabs md-swipe-content>
    <md-tab label="Tab-A">
        <md-tab-body layout-padding="true">
            <h4>Welcome, This is the first tab !</h4>
        </md-tab-body>
    </md-tab>
    <md-tab label="Tab-B" >
        <md-tab-body layout-padding="true" >
            <h4>There you go, I stand second</h4>
            <div>
                <!-- Removing Lorem ipsum for readability -->
                <br/>
            </div>
        </md-tab-body>
    </md-tab>
    <md-tab label="Tab-C">
        <md-tab-body layout-padding="true">
            <h4>At last, you got to me.</h4>
        </md-tab-body>
    </md-tab>
</md-tabs>
```

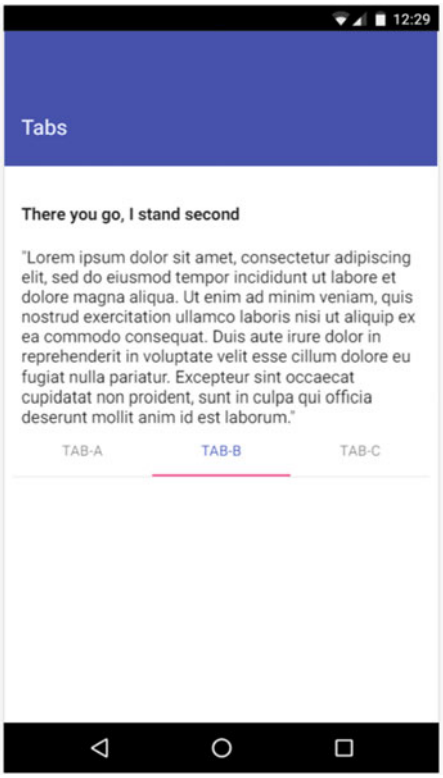

Figure 4-6. *Tabs positioned at the bottom*

md-tab

This is child or a nested element of *md-tabs*. It represents "a tab" in the group. We may specify a label attribute and provide title to the tab. Alternatively we can use *md-tab-label* element nested within the *md-tab.*

md-tab-label allows additional formatting to the tab title. Consider the following code snippet and Figure 4-7. It adds a rower icon and label to the tab label.

```
<md-tab>
  <md-tab-label>
     <md-icon md-svg-src="/img/ic_rowing_black_24px.svg"></md-icon>
     <span>Rower</span>
  </md-tab-label>
</md-tab>
```

Figure 4-7. *Tabs with icon*

■ **Note** If *md-tab-label* is used, it's mandatory to use *md-tab-body*, which otherwise is an optional element.

Attributes:

1. *md-on-select* and *md-on-deselect* evaluate a given expression or run a function on scope. The former runs when tab is selected, and the latter runs as the user navigates away by selecting a different tab. Consider the following code snippet. It calls functions on scope on tab selected and deselected.

```
<md-tab md-on-select="tabSelected()" md-on-deselect="tabDeselected()">

.controller('sampleController', function($scope){
    $scope.tabSelected = function(){
          ...
        console.log('tab selected');
    };

    $scope.tabDeselected = function(){
          ...
        console.log('tab deselected');
    };
```

2. Use *ng-disabled* to disable a tab. A variable on scope could dynamically enable or disable the tab. Consider the following code, which enables/disables tab-3 based on checkbox selection.

```
<input type="checkbox" ng-model="isTab3Enabled">
<span>Disable Tab 3</span>

<md-tab label="Tab-C" ng-disabled="isTab3Enabled">
  ...
</md-tab>
```

3. *md-active*: when set true on an *md-tab,* selects the tab by default. Only one md-tab is expected to have this attribute. This is an alternative to *md-tabs* element's *md-selected* attribute.

Cards

A card is a container. It is a logical unit of information. Material Design spec defines it as "a sheet of material that serves as an entry point to more detailed information." It may have text, images, captions, and so on. See Figure 4-8.

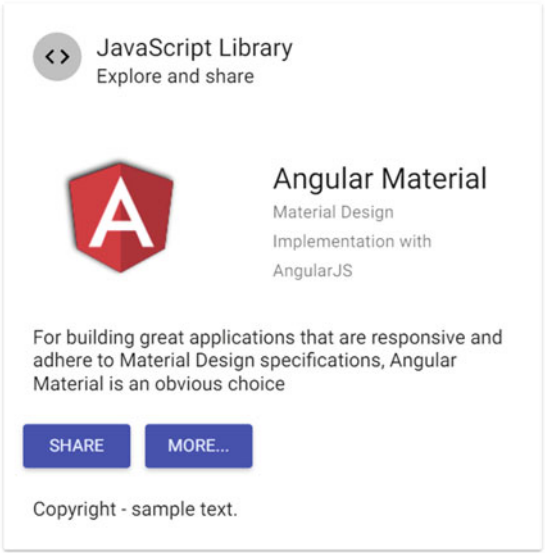

Figure 4-8. *A sample card*

Use the following elements/directives to create a card.

1. *md-card*: It is the root element encapsulating the card control.
2. *md-card-header*: It is an optional element. It is header for the whole card. Usually, card header shows category details. The specific information on the card could have another title or caption. A card header may have the following elements.

   md-card-avatar: As the term indicates, it provides a persona for the card. Figure 4-8 shows an icon as the card avatar. Consider the following code.

```
<md-card-header>
<md-card-avatar class="md-user-avatar">
          <md-icon md-svg-src="img/ic_code_black_24px.svg"></md-
          icon>
     </md-card-avatar>
  <!—Additional elements -->
  </md-card-header>
```

In the preceding sample, we may use an image element instead of md-icon. Consider using CSS class md-user-avatar for a person's photograph or image.

md-header-text: It is another element md-card-header. It can contain elements that show card title and short description. Use CSS class md-title on title element and md-subhead on short description. Consider the following code.

```
<md-card-header>
   <!—Additional elements -->
<md-header-text>
         <div class="md-title">JavaScript Library</div>
         <div class="md-subhead">Explore and share</div>
     </md-header-text>
</md-card-header>
```

3. *md-card-footer*: Optional element for card's footer. See Figure 4-8; footer was used for sample copyright text.

4. *md-card-content*: It is the workspace of the card. If you choose to show image within the content, for bigger dimensions consider using CSS class *md-media-xl.*

5. *md-card-title*: It represents the caption and description text for specific content on the card. In the preceding example, card for Angular Material and its description is coded within the card title. A card title element can have the following elements.

6. *md-card-title-text*: Use it to show the title text and the description. Use CSS classes md-headline and md-subhead for HTML elements that has the title and description. Consider following code.

```
<md-card-title layout-align="center center">
    <!-- Title caption and description -->
    <md-card-title-text flex="50">
        <span class="md-headline">Angular Material</span>
        <span class="md-subhead">Material Design Implementation
        with AngularJS</span>
    </md-card-title-text>
</md-card-title>
```

7. *md-card-title-media*: Use this element to show card title image. In Figure 4-8, it is AngularJS logo. On the image element, you may choose one of the three sizes by applying a CSS class from the following.

1. md-media-sm
2. md-media-md
3. md-media-lg

Consider the following code for the card title image.

```
<md-card-title layout-align="center center">
    <md-card-title-media flex="50">
        <img src="img/angular-logo.png" class="md-media-md"
        alt="AngularJS">
        </md-card-title-media>
        <!—More elements here. -->
    </md-card-title>
```

Refer to the complete code for the following card sample.

```
<md-card >
    <!-- Header for the card. Usually describes category -->
    <md-card-header>
        <!-- Avatar - symbolic representation of content in the card -->
        <md-card-avatar class="md-user-avatar">
            <md-icon md-svg-src="img/ic_code_black_24px.svg"></md-icon>
        </md-card-avatar>
        <!-- Header content -->
        <md-header-text>
            <div class="md-title">JavaScript Library</div>
            <div class="md-subhead">Explore and share</div>
        </md-header-text>
    </md-card-header>
    <!-- Workspace or the main content of the card -->
    <md-card-content>
        <!-- Card title content -->
        <md-card-title layout-align="center center">
            <md-card-title-media flex="50">
                <img src="img/angular-logo.png" class="md-media-md"
                alt="AngularJS">
            </md-card-title-media>
                 <!-- Title caption and description -->
                 <md-card-title-text flex="50">
                     <span class="md-headline">Angular Material</span>
                     <span class="md-subhead">Material Design
                     Implementation with AngularJS</span>
                </md-card-title-text>
            </md-card-title>
```

```
            <p>For building great applications that are responsive
            and adhers to Material Design specifications, Angular
            Material is an obvious choice</p>
        </md-card-content>
        <!-- Actions or buttons that you might need on card -->
        <md-card-actions layout-align="end end">
            <md-button class="md-raised md-primary">Share</md-
            button>
            <md-button class="md-raised md-primary">More...</md-
            button>
        </md-card-actions>
        <!-- Card footer -->
        <md-card-footer>
            <span>Copyright - sample text.</span>
        </md-card-footer>
    </md-card>
```

Summary

This chapter is one step toward building a complete application with Angular Material. It helps design the skeleton for the application. After reading this chapter, think about the high-level view of your application. Think which aspects of it communicate purpose and provide a title, how navigation should be managed, how these aspects adjust on smaller screen sizes. More importantly be consistent with the approach across the application, so that the user is not lost while working with one of many functionalities.

In this chapter, we began describing *md-content* element/directive. It often is the workspace of the page. It allows content to scroll.

Then, we delved into aspects of the toolbar. We explored the *md-toolbar* element/directive for showing the title for the page. It could also be used on other components. In this chapter, we saw an example of navbar using its own title.

Navbar provides a site-map for the application. It is an easy way to access the most-used links and actions in the application. We explored the *md-navbar* element/directive, including responsive aspects that adjust navbar according to the screen size.

Tabs are effective in segregation of UI elements and functionality. *md-tabs* and other child elements are used for creating tabs in Angular Material.

At the end, this chapter detailed cards. It is a container element/directive. We discussed using *md-cards* and other child elements to create cards in Angular Material.

References

For Material Design specification on cards refer to the following URL:
https://www.google.com/design/spec/components/cards.html

CHAPTER 5

Action Buttons

Angular Material provides variety of action buttons. The directives or elements define Material Design style, behavior, and experience. From a developer point of view, they are easy to integrate within a web application.

One of the important aspects of Material Design is the feedback the user receives while interacting with the controls in the UI. The user will clearly know that he/she tapped on a button, used a link, navigated to a different view, and so on. Animations are subtle yet effective in providing context to the user. Primarily, buttons trigger most actions on a web page. By default, Angular Material provides a ripple effect on the click of a button. This is a common feedback mechanism in Angular Material for many other controls like tabs, menus, and so on.

Angular Material integrates well with *ngAnimate*, an AngularJS module for animations. Lots of animations for the purpose of feedback on user actions come from this module. As discussed in Chapter 2, this is a mandatory script to include in an Angular Material application.

This chapter will begin by detailing a simple Angular Material button directive. We will explore variations of Angular Material buttons and the user experience value they provide. The chapter will also describe the concept of FAB (Floating Action Buttons) in Material Design. It will describe directives, which are ready-made FAB controls.

Button Directive (md-button)

For the Material Design button, use *md-button* instead of the default button element.

```
<md-button>Click Me</md-button>
```

Use *ng-href* attribute and use the button as a link.

```
<md-button ng-href="https://material.angularjs.org/" target="_blank">Use as
a link</md-button>
```

Use the CSS class *md-raised* for an elevated 3D effect on the button.

```
<md-button class="md-raised">Raised Button</md-button>
```


V. K. Kotaru, *Material Design Implementation with AngularJS*,
DOI 10.1007/978-1-4842-2190-7_5

Use the CSS class *md-icon-button* for an icon button (without a label and only an icon/image). It is styled accordingly and adjusts height and width to fit the icon. To show the icon, use the *md-icon* directive. The following code shows using an SVG file (for icon) on the button. See Figure 5-1 for sample buttons.

```
<md-button class="md-icon-button">
    <md-icon md-svg-src="/images/ic_menu_black.svg"></md-icon>
</md-button>
```

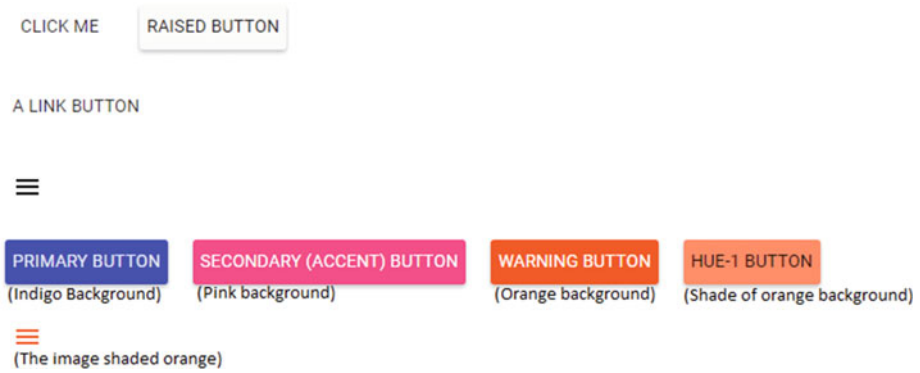

Figure 5-1. *Various Angular Material buttons*

Style and Intention

Use a button as one of the following. They depict intention.

1. Primary: It represents default or primary action in the view. Consider a form and a submit or a save button. It preferably is a primary button.

   Apply CSS class *md-primary* to make it a primary button.

2. Accent: Use accent colors on a highlighted action. In general, these are special actions. These are expected to grab user attention. In some scenarios, they are the most used actions.

   For example, on a shopping application, add to shopping cart could be an accent button.

   Apply CSS class *md-accent*.

3. Warning: As the term indicates, the user needs to use this action carefully; for example, an action that results in deleting a document or a record.

   Apply CSS class *md-warn*.

4. Hue: A primary, accent, or warning action might need to be identified distinctly. It could be done with a subtle change in the color shade. Consider a delete action/button. It is a warning button intended to delete a record. Along with it, "Delete Permanently" could be a warning action with subtle change. Hence, use a hue, along with warning style.

   Apply CSS class *md-hue-1*, *md-hue-2*, or *md-hue-3*.

Consider the following code and Figure 5-2. On a raised button, we used the style intentions mentioned previously.

```
<md-button class="md-raised md-primary">Primary Button</md-button>
<md-button class="md-raised md-accent"> Accent Button</md-button>
<md-button class="md-raised md-warn">Warning Button</md-button>
<md-button class="md-raised md-warn md-hue-1">Hue-1 Button</md-button>
```

Figure 5-2. *Styled buttons*

Consider the following code and Figure 5-3 for styles applied on a default button and an icon button.

```
<div>
     <md-button class="md-icon-button md-warn">
       <md-icon md-svg-src="/images/ic_menu.svg"></md-icon>
    </md-button>
    <strong>(Menu Button)</strong>
</div>

<div>
    <md-button class="md-warn"><strong>Default Button</strong></md-button>
</div>
```

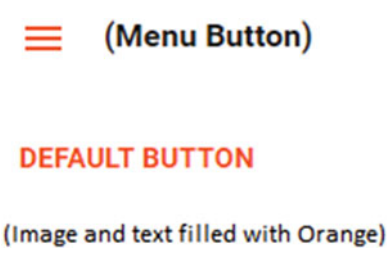

Figure 5-3. *Other styled buttons*

■ **Note** While using SVG for an icon button as in the preceding, ensure SVG does not have a fill attribute of its own. That could override warn or any other theme color from the CSS class.

SVG with fill color

```
<svg fill="#000000" height="24" viewBox="0 0 24 24" width="24"
xmlns="http://www.w3.org/2000/svg">
    <path d="M0 0h24v24H0z" fill="none"/>
    <path d="M3 18h18v-2H3v2zm0-5h18v-2H3v2zm0-7v2h18V6H3z"/>
</svg>
```

Button click/tap feedback:

1. **md-no-ink:** Angular Material buttons show ripple effect on clicking/tapping the button. For any reason if it needs to be disabled, use an attribute *md-no-ink* on *md-button.* It will disable the feedback, ripple effect.
2. **md-ripple-size:** Modify ripple effect's size. Possible values are *full*, *partial*, and *auto.*

■ **Note** Like any other control in Angular Material, we can disable a button by using *ng-disabled*. We can programmatically disable or enable by attaching a model value (on the controller).

FAB

FAB are a distinct aspect of Material Design. They are above the UI controls on the page. They highlight and promote an action. See Figure 5-4. "Compose e-mail in Google Inbox" is a familiar example. The plus button indicating an action to open the compose dialog is floating above the Inbox controls.

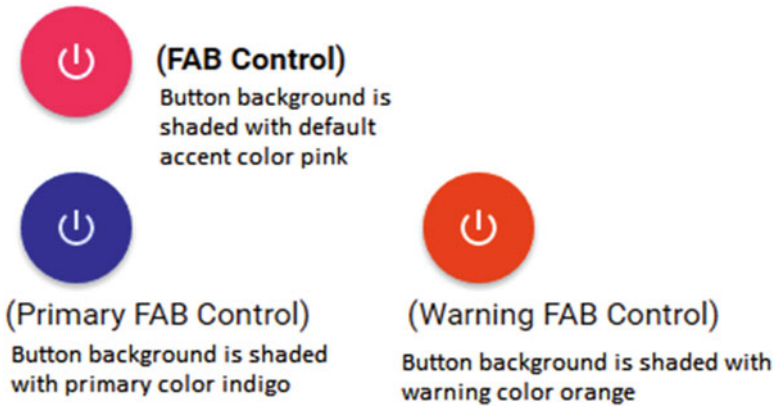

Figure 5-4. *FAB control*

To create a FAB control, use the CSS class *md-fab*. Use an icon in the button directive. Consider the following code.

```
<md-button class="md-fab">
   <md-icon md-svg-src="/images/ic_power_settings.svg"></md-icon>
</md-button>
```

Use *md-mini* CSS class for a smaller-sized FAB control.

A FAB control by default uses the accent color. We can apply CSS classes for Primary and Warning on the directive. See Figure 5-5.

```
<section>
   <md-button class="md-fab md-primary">
     <md-icon md-svg-src="/images/ic_power_settings.svg"></md-icon>
    </md-button>
    <div class="label">(Primary FAB Control)</label>
</section>

 <section>
   <md-button class="md-fab md-warn">
      <md-icon md-svg-src="/images/ic_power_settings.svg"></md-icon>
   </md-button>
   <div class="label">(Warning FAB Control)</div>
</section>
```

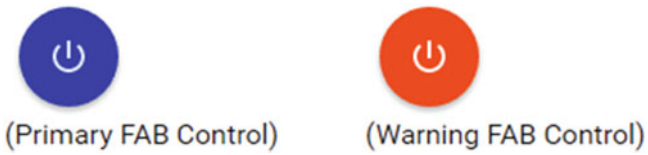

Figure 5-5. *FAB controls, styled primary and warn*

Position the FAB control on any of the four corners of the view/screen. Apply one of the following CSS classes to do so.

1. md-fab-top-left
2. md-fab-top-right
3. md-fab-bottom-left
4. md-fab-bottom-right

Speed Dial

It is a FAB control designed for frequently used actions. Tapping or clicking this button pops open a list of related actions. At most, in two taps or clicks you should be performing the desired action.

Consider Figure 5-6. It shows frequently accessed settings as speed dial. Tapping on the Settings FAB control (button) results in a list of available settings.

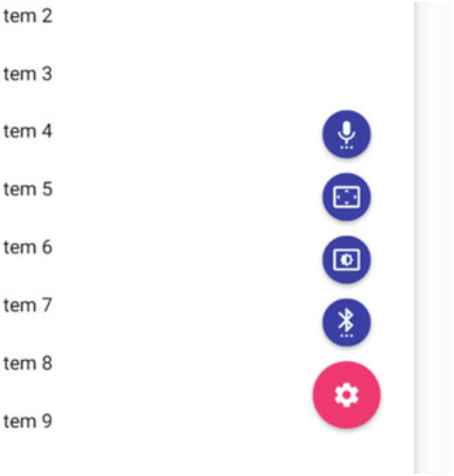

Figure 5-6. *Settings speed dial*

Material Design guidelines suggest the following.

1. All actions that speed dial expands should be related. It should be a logical grouping.
2. A minimum of three frequent actions should be included in speed dial. We are reaching desired action in two clicks/taps. For a single action, it is not useful to expand with an extra click. For two actions, decide which action is more important and use that one action (as a separate FAB, without using speed dial).

3. A maximum of six actions should be included in speed dial. More than that would make the grouping complex and will be counterintuitive.

4. Do not use speed dial as miscellaneous actions or as a replacement for the "more actions" button on the toolbar. It should highlight frequently used functionality.

md-fab-speed-dial

This is the root directive for creating a speed dial using Angular Material. The child directives are as follows:

1. Speed Dial Trigger (*md-fab-trigger*): defines the floating FAB control.

2. Actions (*md-fab-actions*): individual actions that pop out from speed dial.

   (Review Figure 5-6 for better understanding.)

The *md-fab-speed-dial* has the following attributes.

1. **md-direction**: define direction to show actions. Possible values are "*up*," *"down*," *"left*," and "*right."*

2. **md-open**: By default, speed dial opens on clicking the fab control. Use this attribute to programmatically manage opening and closing the speed dial. It takes a Boolean value.

■ **Note** Use *ngCloak* for md-open to be effective. Otherwise, when the view loads for the first time, it always shows the speed dial open. There is an open issue in Angular Material project to change this behavior. Follow the link to see details about the issue.
https://github.com/angular/material/issues/6788

Typically, ngCloak is used to hide Angular HTML template or code while it is not yet ready. Without this attribute, for a brief moment, users might see HTML template or code. However, it will ultimately render with bindings applied.

Instead of click-opening the speed dial, if it needs to open on mouse-over, consider the following approach. Use the attribute *ng-mouseenter* and set the model value for *md-open* to true. Set it back to false for when the mouse leaves the button area. The attribute for mouse leave is *ng-mouseleave.* Consider the following code.

```
<md-fab-speed-dial md-open="isOpen" md-direction="up" class="md-fab-bottom-
right" ng-mouseenter="isOpen=true" ng-mouseleave="isOpen=false">
```

■ **Note** If you need the hover to work even when mouse enters/leaves the action buttons area (not just the speed dial trigger), use a class *md-hover-full.* This opens the speed dial even when mouse hovers in the intended action buttons area. Otherwise, speed dial opens only on hovering over the trigger button.

We can also control the animation while opening action buttons. Use CSS class *md-fling* or *md-scale*. The former is the default.

Like a FAB control, position the speed dial on any of the four corners of the view/screen. Apply one of the following CSS classes to do so.

1. md-fab-top-left
2. md-fab-top-right
3. md-fab-bottom-left
4. md-fab-bottom-right

md-fab-trigger

The *md-button* acts as the FAB speed dial trigger. Clicking (or hovering on) this button expands the list of actions in speed dial.

Encapsulate *md-button* in *md-fab-trigger* directive. In the following sample, CSS class *md-fab* is applied so that the speed dial is actually shown as a FAB control. Speed dial should highlight the most-used actions. Hence, consider using *md-accent* class on the *md-button*.

```
<md-fab-trigger>
    <md-button class="md-fab">
        <md-icon md-svg-src="/images/ic_settings.svg"></md-icon>
    </md-button>
</md-fab-trigger>
```

md-fab-actions

Use this directive to encapsulate list of action buttons that pop out of speed dial trigger. Design the functionality to use no less than three and no more than six action buttons. These numbers are carefully chosen for better user experience.

Use the CSS class *md-fab* for action button to be a FAB control. Consider using another CSS class *md-mini*, which makes action buttons look smaller than the speed dial button. This gives an impression that the action buttons are derivatives of the speed dial trigger. Like the speed dial trigger, action buttons too are FAB controls. Hence, to show an icon on the button, use *md-icon* within *md-button*. Consider the following code (skeleton) for FAB actions.

```
<md-fab-actions>
  <md-button class="md-fab md-primary md-mini">
      ...
  </md-button>
  ...
</md-fab-actions>
```

Optionally, use **tooltip** to describe the button. Use the directive *md-tooltip*.

```
<md-tooltip md-direction="left">Bluetooth Settings</md-tooltip>
```

It will appear as the user hovers over the action button. Set the direction to show the tooltip depending on available space in the app.

Consider the following speed dial code. This is for building the speed dial shown in Figure 5-6.

```
<md-fab-speed-dial md-open="isOpen" md-direction="up" class="md-fling md-
fab-bottom-right" ng-mouseenter="isOpen=true" ng-mouseleave="isOpen=false">
    <md-fab-trigger>
        <md-button class="md-fab">
            <md-icon md-svg-src="/images/ic_settings.svg"></md-icon>
        </md-button>
    </md-fab-trigger>

    <md-fab-actions>
        <md-button class="md-fab md-primary md-mini">
            <md-tooltip md-direction="left">Bluetooth</md-tooltip>
            <md-icon md-svg-src="/images/ic_settings_bluetooth.svg"></md-
            icon>
        </md-button>
        <md-button class="md-fab md-primary md-mini">
            <md-tooltip md-direction="left">Brightness</md-tooltip>
            <md-icon md-svg-src="/images/ic_settings_brightness.svg"></md-
            icon>
        </md-button>
        <md-button class="md-fab md-primary md-mini">
            <md-tooltip md-direction="left">Display Overscan</md-tooltip>
            <md-icon md-svg-src="/images/ic_settings_overscan.svg"></md-
            icon>
        </md-button>
        <md-button class="md-fab md-primary md-mini">
            <md-tooltip md-direction="left">Voice</md-tooltip>
            <md-icon md-svg-src="/images/ic_settings_voice.svg"></md-icon>
        </md-button>
    </md-fab-actions>

</md-fab-speed-dial>
```

FAB Toolbar

Another variant of FAB is the FAB toolbar. It is a collapsed FAB control (a button) that expands to become a full-fledged toolbar. See Figure 5-7 and Figure 5-8. The following are some guidelines while designing FAB toolbar.

1. Always choose related items on a FAB toolbar. It should be a logical grouping.
2. Do not use FAB toolbar like a more action button on the toolbar. For miscellaneous items, use "more button" on an original toolbar.

Figure 5-7. *Settings, a FAB toolbar*

Figure 5-8. *Clicking the FAB toolbar results in opening the toolbar.*

FAB toolbar is very similar to speed dial, with subtle differences.

1. In a more obvious difference, FAB toolbar expands to show the toolbar. Hence, it is possible to expand right or left. This is unlike speed dial, which can expand up and down as well.
2. FAB toolbar hides the FAB while showing the toolbar. It will appear again as the toolbar closes.

md-fab-toolbar (Directive)

Use the Angular Material directive *md-fab-toolbar* for a FAB toolbar. Similarities between *md-fab-speed-dial* and *md-tab-toolbar* are as follows. These define the appearance and behavior of the control.

1. *md-open*: programmatically opens or closes the FAB toolbar.
2. Use one of the following CSS classes to position the FAB control. Class names are self-explanatory:
    a. md-fab-top-left
    b. md-fab-top-right
    c. md-fab-bottom-left
    d. md-fab-bottom-right

3. *md-direction*: Possible values are right and left. Preferably, use right if positioned on top-left or bottom-left, and use left if positioned on top-right or bottom-right. See Figure 5-9.

Figure 5-9. *FAB toolbar—direction*

md-fab-toolbar is a parent directive for the following.

1. **FAB Trigger**: A floating button, which expands to become a toolbar. Directive is *md-fab-trigger*. (Same directive used earlier for FAB speed dial.)
2. **The toolbar**: Part of FAB toolbar. Directive is *md-toolbar*. (Same directive used earlier for a page-level toolbar.)
3. **FAB Actions**: A directive *fab-action-buttons* for buttons on the toolbar. In the context of FAB toolbar, it is a child element under *md-toolbar*. (Same directive used earlier for FAB speed dial.)

Usage

```
<md-fab-toolbar md-open="isOpen" md-direction="right">
    <md-fab-trigger class="align-with-text">
        <md-button aria-label="Settings" class="md-fab md-primary">
            <md-icon md-svg-src="images/ic_settings.svg"></md-icon>
        </md-button>
    </md-fab-trigger>
```

```
    <md-toolbar>
        <md-fab-actions class="md-toolbar-tools">
            <md-button aria-label="Bluetooth Settings" class="md-icon-button">
                <md-icon md-svg-src="images/ic_settings_bluetooth.svg">
                </md-icon>
            </md-button>
            <md-button aria-label="Brightness Settings" class="md-icon-button">
                <md-icon md-svg-src="images/ic_settings_brightness.svg">
                </md-icon>
                </md-icon>
            </md-button>
            <md-button aria-label="Overscan Settings" class="md-icon-
            button">
                <md-icon md-svg-src="images/ic_settings_overscan.svg"></md-
                icon>
            </md-button>
        </md-fab-actions>
    </md-toolbar>
</md-fab-toolbar>
```

Notice that on action buttons, we are using class *md-icon-buttons* (opposed *to md-fab-button*). On toolbar, action buttons are icon buttons.

■ **Note** You might see warning wherever ARIA attributes are not provided. ARIA provide screen readers and tools with the ability to interpret controls and read provided text for the visually impaired. Angular Material integrates highly with ngAria, an AngularJS module for these features. A separate chapter in the book describes ngAria features in Angular Material. It is highly recommended to make use of these features.

Menu

Using a menu, multiple actions could be collapsed into a button. Click the button to expand individual options. See Figure 5-10. All page-level actions are collapsed into a button on the toolbar. It acts as a trigger. As the user clicks it, each available option is shown.

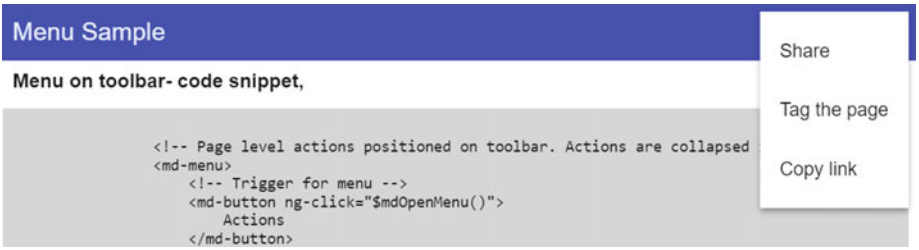

Figure 5-10. *Menu in the toolbar*

To create a menu, use the following elements/directives.

1. *md-menu*: It is the root element while creating a menu. The trigger button should be coded directly under it. Consider the following code snippet.

```
<md-menu>
    <!-- Trigger for menu -->
    <md-button ng-click="$mdOpenMenu()">
        Actions
    </md-button>
                ...
</md-menu>
```

   Use *$mdOpenMenu* function to expand the menu. It is called on trigger button's click event.

2. md-*menu-content*: Code it under *md-menu*. It encapsulates all menu options.

3. *md-menu-item*: Typically, a menu content element contains one or more menu items. Use this element to code each menu option.

   Typically, menu items are buttons. Use a controller function on scope to handle events as these buttons are selected or clicked.

Refer to the following complete code below:

```
<!-- Page level actions positioned on toolbar. Actions are collapsed into a
menu -->
        <md-menu>
            <!-- Trigger for menu -->
            <md-button ng-click="$mdOpenMenu()">
                Actions
            </md-button>
            <!-- Individual menu options and buttons-->
            <md-menu-content>
                <md-menu-item>
                    <md-button ng-click=”shareHandler()”>
                        Share
                    </md-button>
                </md-menu-item>
                <md-menu-item>
                    <md-button ng-click=”tagHandler()”>
                        Tag the page
                    </md-button>
                </md-menu-item>
                <md-menu-item>
                    <md-button ng-click=”copyHandler()”>
                        Copy link
                    </md-button>
                </md-menu-item>
            </md-menu-content>
        </md-menu>
```

Alignment

Menu options, by default, are aligned relative to the *md-menu* element. To align it to the trigger button, specify the *md-menu-origin* attribute on *md-button*. Also, specify the *md-menu-align-target* attribute on individual menu options. Consider the following code.

```
    <!-- Page level actions positioned on toolbar. Actions are collapsed
into a menu -->
        <md-menu >
            <!-- Trigger for menu -->
            <md-button md-menu-origin ng-click="$mdOpenMenu()">
                Actions
            </md-button>
```

```
            <!-- Individual menu options and buttons-->
            <md-menu-content >
                <md-menu-item>
                    <md-button md-menu-align-target >
                        Share
                    </md-button>
                </md-menu-item>
...
            </md-menu-content>
        </md-menu>
```

For subtle changes in positioning the menu relative to the trigger, use *md-offset*. Provide values for the x and y axes. Consider the following sample and Figure 5-11. We repositioned the menu on the y axis. Compare it to Figure 5-10; trigger button "actions" are visible here while the menu is expanded. Menu does not override the trigger.

```
        <md-menu md-position-mode="target-right target" md-offset="0 40">
...
```

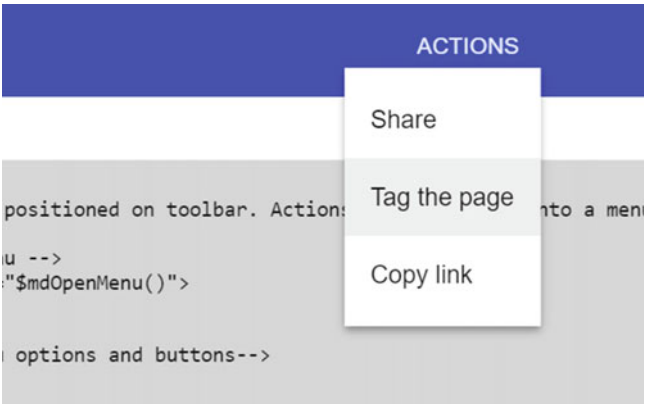

Figure 5-11. *Offset y axis by 40*

Use *md-position-mode* to change the menu origin. The menu expands out of the given x and y axis positions. However, possible values that we can provide are limited. The default value is *target* on the x and y axes. We can change the x axis value to *target-right* for flipping the default alignment. Consider the following code.

```
  <md-menu md-position-mode="target-right target" md-offset="0 40">
                    <!-- Trigger for menu -->
                ...
        <!–rest of  the menu code -->
</md-menu>
```

Wider Menu Options

Use the *width* attribute on *md-menu-content* to change the default width. See Figure 5-12.

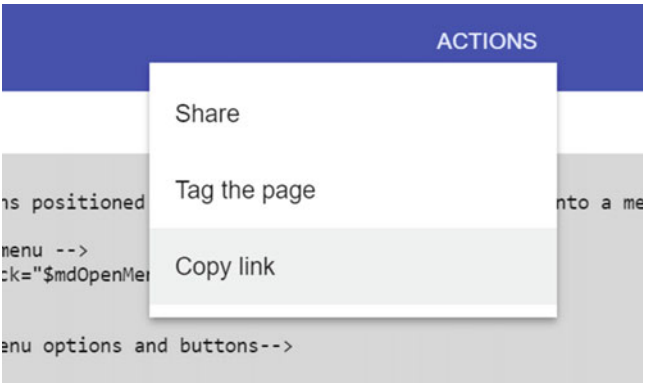

Figure 5-12. *Wider menu options*

The code snippet follows. Set width to 4 points. Possible values for this attribute are 2, 4, or 6.

```
    <md-menu-content width="4">
...
```

Separator

Consider using a separator for better grouping. Use *md-divider* element/directive. Code it as a menu item. See Figure 5-13 and the following code.

Figure 5-13. *Menu separator*

```
<md-menu>
    <md-menu-content>
<!–More menus items -->
     <md-menu-item>
            <!-- Separator for grouping -->
            <md-divider></md-divider>
        </md-menu-item>
<!–More menus items -->

        </md-menu-content>
</md-menu>
```

Menu Bar

A menu bar is a traditional approach to menus. Typically, Windows and Mac applications use menus on top of the screen or window. They list all possible actions with the window. Sometimes menus are used in web apps as well.

Angular Material provides elements/directives for creating such an interface in a web application. Consider Figure 5-14.

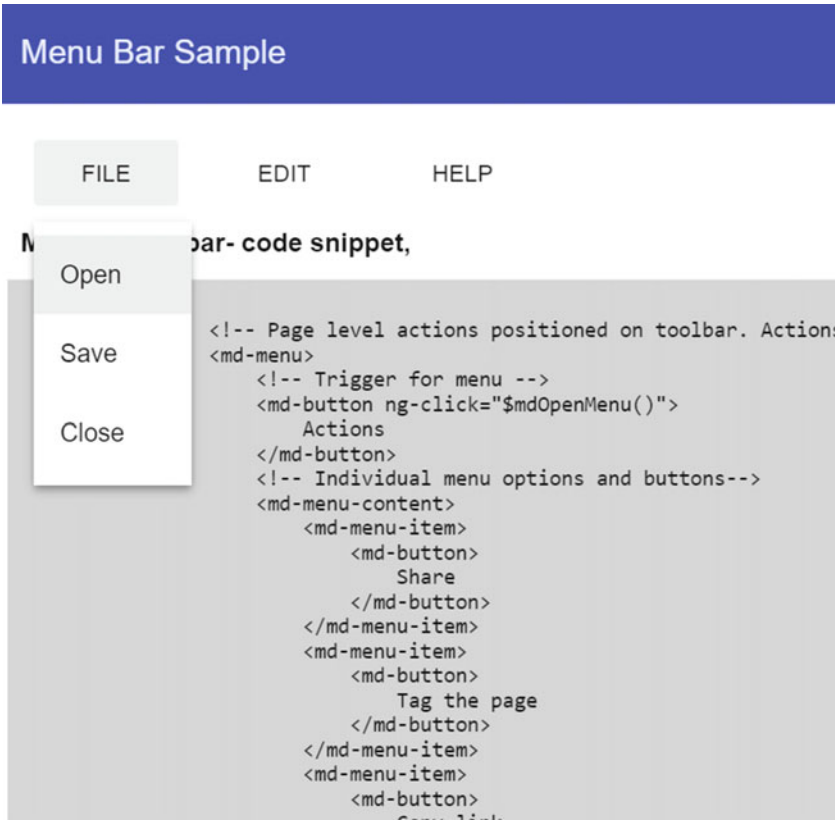

Figure 5-14. *Menu bar sample*

Use *md-menu-bar* element to create a menu bar. A menu bar contains one or more menus (*md-menu* elements). As we have seen so far, each *md-menu* encapsulates a complete menu and its options. As seen in Figure 5-14, "file" is a menu (*md-menu*) with three menu options: open, save, and close. The menu bar (*md-menu-bar*) wraps three menus—file, edit, and help—each with multiple menu options.

Refer to the following code sample for menu bar depicted in Figure 5-13.

```
<md-menu-bar>
    <md-menu md-offset="0 4">
        <button ng-click="$mdOpenMenu()">File</button>
        <md-menu-content width="2">
        <!—Menu options go here -->
        </md-menu-content>
    </md-menu>
```

```
    <md-menu md-offset="0 4">
        <button ng-click="$mdOpenMenu()">Edit</button>
        <md-menu-content width="2">
        <!—Menu options go here-->
        </md-menu-content>
    </md-menu>

    <md-menu md-offset="0 4">
        <button ng-click="$mdOpenMenu()">Help</button>
        <md-menu-content width="2">
        <!—Menu options go here-->
        </md-menu-content>
    </md-menu>
</md-menu-bar>
```

We can nest menus. See Figure 5-15.

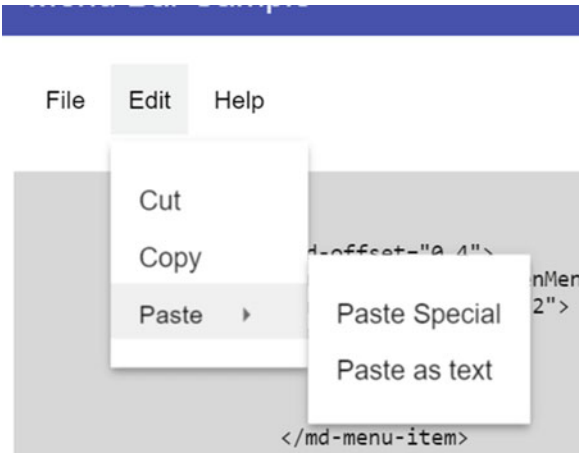

Figure 5-15. *Nested menus*

Consider the following code. In the menu item for "paste," instead of an *md-button*, an *md-menu* has been coded.

```
<md-menu md-offset="0 4">
    <button ng-click="$mdOpenMenu()">Edit</button>
    <md-menu-content width="2">
        <md-menu-item>
            <md-button>
                Cut
            </md-button>
        </md-menu-item>
```

```
        <md-menu-item>
            <md-button>
                Copy
            </md-button>
        </md-menu-item>
        <md-menu-item>
            <md-menu>
                <md-button>Paste</md-button>
                <md-menu-content width="2">
                    <md-menu-item>
                        <md-button>Paste Special</md-button>
                    </md-menu-item>
                    <md-menu-item>
                        <md-button>Paste as text</md-button>
                    </md-menu-item>
                </md-menu-content>
            </md-menu>
        </md-menu-item>
    </md-menu-content>
</md-menu>
```

Summary

For the most part, buttons drive actions on the page. The controls or directives demonstrated in this chapter help make our application use Material Design styles, visual effects, and animations, and hence create an overall Material Design experience. These directives are easy to integrate and provide high user experience value. Some of the aspects to highlight are as follows.

1. Feedback or ripple visual effect to the user on tap/click.
2. Appearance, positioning, and behavior of FAB controls.
3. Animation effects.

In this chapter, we initially explored using *md-button* directive to create a button. We explored variations in options with styling and themes.

We delved into FAB, described using FAB controls as speed dial and toolbar. We also used *md-fab-speed-dial* and *md-fab-toolbar* directives (and many other child elements) to create respective controls.

Finally, this chapter detailed using traditional menus and various options with it. We used *md-menu* directive to create a menu and *md-menu-bar* for a menu bar.

References

For specification on Material Design FAB controls, see https://www.google.com/design/spec/components/buttons-floating-action-button.html#buttons-floating-action-button-floating-action-button

For FAB control implementation in Angular Material and documentation, see https://material.angularjs.org/

CHAPTER 6

Themes

Themes provide consistency in look and feel. A theme is defined by set of colors and shades.

This chapter describes themes in Material Design and the implementation in Angular Material. Terminology is described at the beginning of the chapter. It details using themes out of the box and using the CSS classes provided by the library. For most applications, it should be sufficient.

Some might need to go in-depth and customize. Later sections of the chapter describe creating custom theme, palette, and so on. It details directives, provider, and other Angular Material API to customize themes.

Angular Material Theming

With Material Design, various color and styling aspects have already been carefully designed. By adopting Material Design, we know we are adhering to the guidelines and hence in the application are providing a consistent and beautiful user experience.

Theming in Angular Material serves this purpose. Various aspects of theming help our web application adhere to Material Design guidelines.

Palette

Color is an important aspect of theme. Color defines intention. A palette is set of shades/hues of a color. In Angular Material, we work with the following palettes.

1. Primary: As the term indicates, the control using this style is a primary element. For example, in a form, the submit button is a primary button.

   In Angular Material, the default primary color is indigo.

2. Accent: These are highlighted controls and text. Preferably, use accent colors on FAB, sliders, switches, highlighted text, and so on.

   In Angular Material, the default accent color is pink.

V. K. Kotaru, *Material Design Implementation with AngularJS*,
DOI 10.1007/978-1-4842-2190-7_6

3. Warn: Angular Material provides warn theme that indicates caution while performing an action. It could be a prompt to confirm deleting a document or a record or exiting without saving work.

   In Angular Material, the default warn color is red.

4. Background: By default, a grey palette is set for the background. White is a hue in the palette.

Consider the palette for indigo, a default primary color, in Figure 6-1. From the color palette, we can select one or more hues/shades to indicate variation in purpose.

	Color	Shade
	#3F51B5	500
	#E8EAF6	50
	#C5CAE9	100
	#9FA8DA	200
	#7986CB	300
	#5C6BC0	400
	#3F51B5	500
	#3949AB	600
	#303F9F	700
	#283593	800
	#1A237E	900
	#8C9EFF	A100
	#536DFE	A200
	#3D5AFE	A400
	#304FFE	A700

Figure 6-1. *Indigo palette*

Material Design recommends using up to three hues of the primary color. The accent color could be a hue from secondary palette.

Basic Usage

Use CSS classes to apply themes and color intentions in HTML markup. Class names are self-explanatory.

1. md-primary
2. md-accent
3. md-warn

■ **Note** By convention, CSS classes and directives provided by Angular Material are prefixed with "*md-*".

Consider the following code and Figure 6-2. It shows color intention usage on buttons.

```
<div layout="row">
  <md-button class="md-raised">Cancel</md-button>
  <md-button class="md-raised md-warn">Reset</md-button>
  <md-button class="md-raised md-primary">Save And Close</md-button>
</div>
```

Figure 6-2. *Color intention on buttons*

On a typical form,

1. Save may be the primary button. Hence, md-primary CSS class has been applied.
2. Clicking reset will lose data. Hence md-warn CSS class is applied.
3. Cancel has the default color theme of a button.

Accent color is not included in this sample, primarily because the accent color is intended for special actions, cursors, the radio button knob, and so on. From a color intention point of view, it does not fit the use case here. Multiple other samples in the chapter explain accent colors.

The default intention/color theme is different for different types of controls. Consider the following code and picture. Toolbar and switch control have different colors even without applying a CSS class. Following Material Design guidelines, these directives have their respective styles applied by default.

The toolbar has primary color intention and the switch has accent color intention. See Figure 6-3.

```
<md-toolbar >
 <div class="md-toolbar-tools ">
     Sample Header
 </div>
</md-toolbar>

<md-content layout-padding>
  <md-switch ng-model="true">Power On?</md-switch>
</md-content>
```

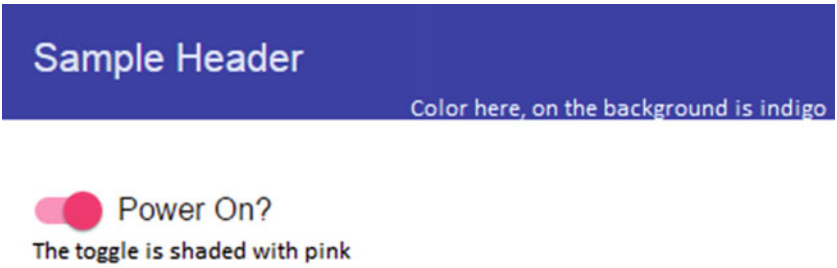

Figure 6-3. *Primary and accent colors on toolbar & switch controls, respectively*

We could change this by applying a CSS class. Consider the following sample and Figure 6-4.

```
<md-toolbar class="md-accent" >
 <div class="md-toolbar-tools ">
     Sample Header
 </div>
</md-toolbar>
<md-content layout-padding>
    <md-switch ng-model="true" class="md-primary">Power On?</md-switch>
</md-content>
```

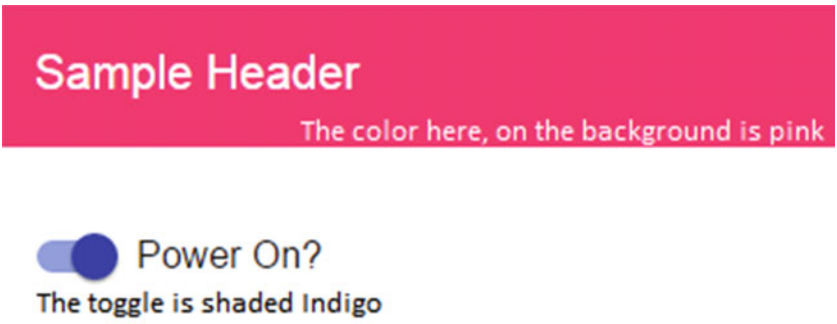

Figure 6-4. *Override default color styles*

It is not advisable to use accent colors on toolbar and primary colors on switch control. Having said that, there are special cases. For example, while we show a warning message to the user (say, confirm dialog to finally make a decision on deleting a document), we might set the toolbar intention to warn. This draws extra attention while making the critical decision.

Shade or Hue

We could show variation in selection by using hue on the color intention. The following is a sample. It is not recommended to use too much variation. It could overwhelm the user with too many colors/shades on the screen. However, in the whole app, we may choose to use up to three hues other than the default on primary color intention.

Consider the following code and Figure 6-5.

```
<md-button class="md-raised md-primary md-hue-1">Hue-1</md-button>
<md-button class="md-raised md-primary md-hue-2">Hue-2</md-button>
<md-button class="md-raised md-primary md-hue-3">Hue-3</md-button>
<md-button class="md-raised md-primary">Default</md-button>
```

Figure 6-5. *Color hue*

Customize Themes

Angular Material provides good flexibility in managing themes. The following are the available palettes in Material Design and hence in Angular Material. All have multiple hues/shades. See Figure 6-6.

Figure 6-6. *Available palettes*

■ **Note** Material Design includes explicit black and white palette as well. For Angular Material they are hues on grey palette.

By default, Material Design has indigo, pink, and red as primary, accent, and warn colors, respectively. We can modify the default to use different color palettes. Consider the following code to edit the default color theme.

```
angular.module("ngMaterialSample", ["ngMaterial"])
.config(function($mdThemingProvider){
    $mdThemingProvider
        .theme('default')
        .primaryPalette('cyan')
        .accentPalette('lime')
        .warnPalette('orange');
})
```

config is an AngularJS function that is invoked at bootstrap time, while the module is loading. Providers and constants can be injected into a config function. The preceding code configures the default theme using a provider, *$mdThemingProvider* of *ngMaterial*.

■ **Note** In AngularJS, a provider is a JavaScript object. We can create providers and expose a set of API or functions. It is one of the core artifacts in AngularJS. It could be injected at bootstrap time into config functions. It is recommended while performing application/module-wide configurations.

Provider needs a $get function defined. While injecting a provider into controllers and other services, this function works as a factory. However, in config function all methods on "this" object of provider function could be used.

The preceding code uses the default theme. It sets primary, accent, and warn palettes to the respective colors.

As we edited the default theme, these colors are applied in the UI already. No changes are required in the markup/HTML. Refer to primary and accent colors in Figure 6-7.

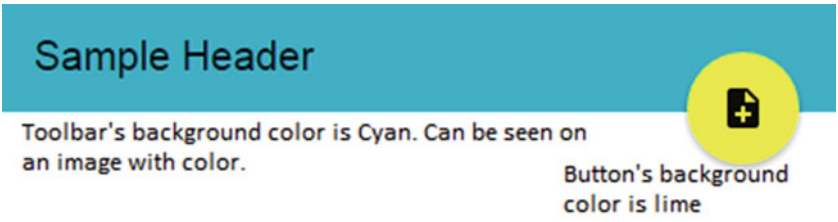

Figure 6-7. *Toolbar and FAB with edited default theme*

Here is the code to show toolbar and FAB control.

```
<md-toolbar>
    <div class="md-toolbar-tools ">
        Sample Header
    </div>
</md-toolbar>
<div>
    <md-button aria-label="Settings" class="md-fab md-fab-top-right">
        <md-icon md-svg-src="images/ic_add.svg"></md-icon>
    </md-button>
</div>
```

For each theme, there are two contrast values available: dark and light. By default, most themes including the default load light contrast. To set dark contrast, call the dark function on the theme object. Consider the following code.

```
$mdThemingProvider
          .theme('default')
          .dark();
```

Not calling the *dark* function causes light contrast to be used by default. However, if we need to be explicit to use light contrast, call *dark* function with false as a parameter.

Along with primary, accent, and warn, we could also edit background palette. Consider the following sample.

```
$mdThemingProvider
                .theme('default')
                .backgroundPalette('lime');
```

It changes background palette on default theme. See Figure 6-8.

Figure 6-8. *Background palette changed*

Define a New Theme

We can create a custom theme of our own. Consider the following code. Unlike the earlier code sample, here the theme name is a custom name.

```
angular.module("ngMaterialSample", ["ngMaterial"])
.config(function($mdThemingProvider){
    $mdThemingProvider
        .theme('aCustomTheme')
        .primaryPalette('cyan')
        .accentPalette('lime')
        .warnPalette('orange');
});
```

While using it in the markup, specify the new theme name using a directive/attribute *md-theme* on any element. The new theme is applied on the element itself and on all child elements. Using the directive allows markup to use different themes with different elements. Another section or another template file might continue to use the default theme or yet another custom theme.

```
<div md-theme="aCustomTheme">
    <md-toolbar>
    <div class="md-toolbar-tools ">
       Sample Header
    </div>
    </md-toolbar>
    <div>
       <md-button aria-label="Settings" class="md-fab md-fab-top-right">
           <md-icon md-svg-src="images/ic_add.svg"></md-icon>
           </md-button>
    </div>
</div>
```

This opens up the possibility to bind the theme value to a model on scope. However, themes are not watched dynamically. Use an explicit directive *md-theme-watch* for it to work. Consider the following code.

```
<div md-theme="{{aThemeSelectedOnTheFly}}" md-theme-watch>
   ...
</div>
```

To apply the same behavior across the application use *alwaysWatchTheme* API on *$mdThemingProvider* (in config function).

```
$mdThemingProvider.alwaysWatchTheme(true);
```

However, do realize that applying the theme dynamically will have performance implications.

■ **Note** To make a custom theme as the default theme, you could use *setDefaultTheme* API on *$mdThemingProvider*. Unlike the earlier sample where we modified palettes on default, we could create a completely new theme and set it as default on demand. However, this does not provide flexibility to use different themes in different templates.

```
$mdThemingProvider.setDefaultTheme('aCustomTheme');
```

Hue Configuration

Consider the image in Figure 6-1. Each palette has fifteen hue values. By default, in the theme:

1. For primary and warn, 500, 300, 800, and A100 are default, hue-1, hue-2, and hue-3, respectively.
2. For accent, A200, A100, A400, and A700 are default, hue-1, hue-2, and hue-3, respectively.

We configure different values on the palette as default, hue-1, hue-2, and hue-3. Consider the following code. It is the same API used to configure theme. See Figure 6-9 for the result rendered.

```
$mdThemingProvider
    .theme('aCustomTheme')
    .primaryPalette('cyan', {
        'default': '900',
        'hue-1': '50',
        'hue-2': '200',
        'hue-3': '600'});
```

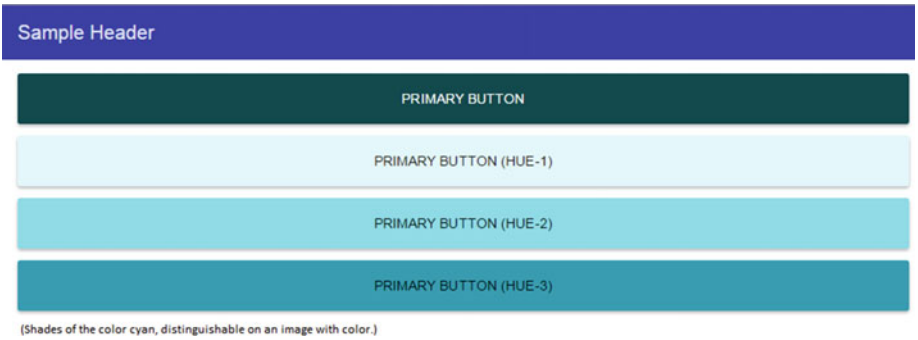

Figure 6-9. *Multiple hue values*

The JSON object as second parameter will provide hue configuration. The following markup could make use of hue values in the template.

```
<md-content md-theme="aCustomTheme" layout-padding layout="column">
     <md-button class="md-raised md-primary">
           Primary button
     </md-button>
```

```
      <md-button class="md-raised md-primary md-hue-1">
            Primary button (Hue-1)
      </md-button>
      <md-button class="md-raised md-primary md-hue-2">
            Primary button (Hue-2)
      </md-button>
      <md-button class="md-raised md-primary md-hue-3">
            Primary button (Hue-3)
      </md-button>
 </md-content>
```

The same can be done for accent and warn colors as well; that is, provide hue configuration as a parameter to *accentPalette* and *warnPalette* functions.

Create Custom Palette

Angular Material provides API to create a custom palette. If the out-of-the-box palettes available (refer to Figure 6-3) are not sufficient, that is, if we need a new variation, we could create a new palette altogether.

$*mdThemingProvider* API *definePalette* will let us create a new palette with a given configuration. We need to make sure to provide color codes for all hues that are mandatory for creating a palette.

```
$mdThemingProvider
        .definePalette('aCustomPalette', {
            '50': '#ff7b82',
            '100': '#ff626a',
            '200': '#ff4852',
            '300': '#ff2f3a',
            '400': '#ff1522',
            '500': '#fb000d',
            '600': '#e1000c',
            '700': '#c8000a',
            '800': '#ae0009',
            '900': '#950008',
            'A100': '#ff959a',
            'A200': '#ffaeb3',
            'A400': '#ffc8cb',
            'A700': '#7b0006',
            'contrastDefaultColor': 'light'

    });
```

We could use the newly created custom palette on a theme. The following code is using it as a primary palette on a custom theme.

```
$mdThemingProvider
    .theme('aCustomTheme')
    .primaryPalette('aCustomPalette');
```

In the palette object, valid values for *'contrastDefaultColor'* are *dark* and *light*. See Figure 6-10 and Figure 6-11, which depict the difference in contrast.

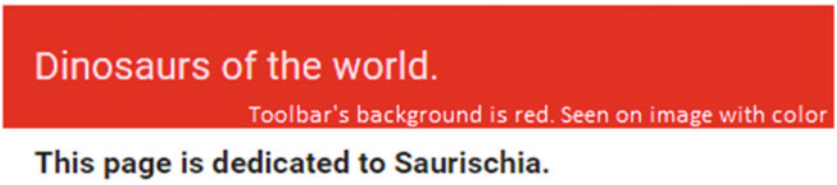

Figure 6-10. *With contrastDefaultColor value light*

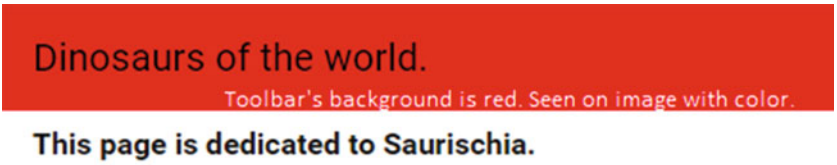

Figure 6-11. *With contrastDefaultColor value dark*

Summary

Out of the box, Angular Material provides everything needed for developing consistent UI adhering to Material Design principles. It supports good variations so that each application has its own identity and they all do not look the same. Yet for personalizing the experience further, powerful customization API are provided.

The default colors and hues are used with CSS classes *md-primary*, *md-accent*, and *md-warn*. Each depicts a purpose. Main action controls are 'primary'; Highlighted actions are 'accent' and the ones that could let user potentially lose data are 'warn'. We can customize the color combination by editing the default theme.

If we need to use multiple themes, we can create additional themes with custom theme names. *$mdThemingProvider* makes all these customizations possible.

The provider also allows configuring three hues, namely, hue-1, hue-2, and hue-3 (in addition to the default value) for each color purpose (primary, accent, and warn).

$mdThemingProvider lets the application configure a custom palette, which is a completely new set of color shades or hues.

References

For Material Design color schemes, refer to https://www.google.com/design/spec/style/color.html#color-color-schemes

For Material Design color palette information, refer to https://www.google.com/design/spec/style/color.html#color-color-palette

For Angular Material theming details, refer to https://material.angularjs.org/latest/Theming/01_introduction

CHAPTER 7

Forms

This chapter will detail various Angular Material input controls and directives. Beginning with simple input text boxes, it will delve into many elements like sliders, drop-down, radio buttons, and so on. Validations are an important aspect of working with forms in a web application. This chapter details various out-of-the-box features for checking form validity and pattern matching. It explains options in displaying and highlighting errors to the user.

Input Container Directive

md-input-container is a parent directive for input elements on a form. It groups the label and the input element (text field, text area, or a select drop-down).

Usage

```
<md-input-container>
    <label>Enter your full name</label>
    <input type="text">
</md-input-container>
```

This results in an Angular Material–style text field, which shows label similar to placeholder text. As the user clicks the text field (or gets focus by placing the cursor), it will move away with a little animation. See Figure 7-1.

Figure 7-1. *Angular Material input textbox*

V. K. Kotaru, *Material Design Implementation with AngularJS*,
DOI 10.1007/978-1-4842-2190-7_7

Another usage is code with placeholder attribute on the input field, instead of a label element.

```
<md-input-container >
    <input type="text" placeholder="Enter full name">
</md-input-container>
```

There is also an option to disable floating animation of the placeholder text. Use *md-no-float* on *md-input-container* element. The placeholder text hides as the user starts keying in values. It is the default behavior of HTML elements with placeholder.

For another useful feature, we could provide hints on the fields. Write an additional element (could be a div tag). Use a CSS class *hint* to render the element as a guide for the field. Consider the following sample and Figure 7-2.

```
<md-input-container flex >
   <input type="text" placeholder="Phone number" flex name="phone"
   ng-model="phone" required >
   <div class="hint">+xx (yyy) zzz-zzzz; x= Country code; y= Regional code;
z= Phone number </div>
</md-input-container>
```

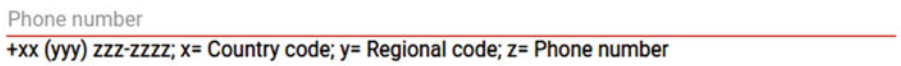

Figure 7-2. *Hint text on textbox*

Form Validations

Angular Material uses module *ngMessages* for form field validation. Make sure to include this dependency while creating our application module.

```
angular.module('sampleApp', ['ngMaterial', 'ngMessages'])
```

Validation messages are shown using two directives, *ng-messages* and *ng-message*. The latter is a child element of the former. A specific validation error message (like required filed value not provided or an incorrect e-mail format) is written using ng-message directive. The ng-messages directive groups all these for a single form field (first name field, e-mail field, or a phone number field). In most cases, there are more than one *ng-message* directive for each *ng-messages*. Code them on each field under *md-input-container*.

Validation attributes are added on the fields to be validated. Validation error/warning messages are shown on child elements of the *ng-messages* directive. Consider the following sample. A required field validation needs to happen on full name. Hence, the required attribute is specified on the input element. Next to the input element, code an element with an error/warning message for any validation failures. This element has *ng-messages* directive applied.

```
<md-input-container >
  <input type="text" placeholder="Enter full name" name="fullName" ng-
  model="fullName" required md-no-asterisk>
  <div ng-messages="userApplication.fullName.$error">
    <div ng-message="required">Fullname is mandatory.</div>
  </div>
</md-input-container>
```

A $*error* object will be added on *fullName* field for any validation errors. The object structure for accessing the error message is *formName.fieldName.$error*. In this sample it is *userApplication.fullName.$error.*

A required filed validation has been coded in this sample. There could be more validations as well. The ng-message directive (child div element) looks for a specific message to be shown in case of error. Refer to Figure 7-3. It displays a validation message as the field loses focus.

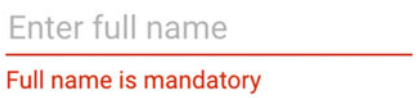

Figure 7-3. *Validation error message on textbox*

Max Length Validation

The *md-maxlength* directive validates the maximum length specified in the directive. Consider the following example and Figure 7-4. The directive shows the character count and the validation error message.

```
<md-input-container flex >
   <input type="text" placeholder="Enter full name" flex name="fullName" ng-
model="fullName" required md-maxlength="20">
   <div ng-messages="userApplication.fullName.$error">
         <div ng-message="required">Fullname is mandatory</div>
         <div ng-message="md-maxlength">This field can not accept more than
         twenty characters</div>
    </div>
</md-input-container>
```

Figure 7-4. *Max length validation error*

E-mail Address Validation

Use input element type e-mail. It performs e-mail validation. Use *ng-messages* directive to show a specific validation message. See Figure 7-5.

```
<md-input-container flex >
   <input type="email" placeholder="Primary e-mail" flex name="email" ng-
   model="email" required >
   <div ng-messages="userApplication.email.$error">
      <div ng-message="required">Email address is mandatory</div>
      <div ng-message="email">Invalid email address</div>
   </div>
</md-input-container>
```

Figure 7-5. *E-mail address validation error message*

RegEx Validation

Use *ng-pattern* attribute for RegEx validation. Consider the following example for RegEx validation for a number format, including country code.

```
+xx (yyy) zzz-zzzz; x= Country code; y= Regional code; z= Phone number
```

Specify RegEx in the *ng-pattern* directive. Use *ng-message* directive with value *pattern*. Consider the complete code,

```
<md-input-container flex >
   <input type="text" placeholder="Phone number" flex name="phone" ng-
   model="phone" required ng-pattern="/^\+[0-9]{1,2} \([0-9]{3}\) [0-9]{3}
   [0-9]{4}$/" >
   <div class="hint">+xx (yyy) zzz-zzzz; x= Country code; y= Regional code;
   z= Phone number </div>
      <div ng-messages="userApplication.phone.$error">
           <div ng-message="required">Phone number is mandatory</div>
           <div ng-message="pattern">Invalid phone number</div>
       </div>
</md-input-container>
```

Multiple Validation Messages

There are scenarios where multiple validation messages need to be shown on a field. Consider the following example. A basic e-mail validation is required and all the e-mails on this form should end with @mycompany.com. By default, the top validation message in *ng-messages* is shown. As the user fixes it, the next error message is shown (if it fails). If we need to show all validation messages that failed at a time, use *multiple* attribute on *ng-messages*. Consider the following sample and Figure 7-6.

```
<md-input-container flex>
   <input type="email" placeholder="Primary e-mail" flex name="email" ng-
   model="email" required  ng-pattern="/^[a-zO-9]*@mycompany.com$/">
   <div ng-messages="userApplication.email.$error" multiple>
         <div ng-message="required">Email address is mandatory</div>
         <div ng-message="email">Invalid email address</div>
         <div ng-message="pattern">Not a useful email. All on this form
         should end with mycompany.com</div>
     </div>
</md-input-container>
```

Figure 7-6. *Multiple validation messages*

More Form Elements

The following are useful Angular Material form or input elements:

Drop-down

Use *md-select* for Angular Material–style drop-down. Consider the following array object on $scope.

```
$scope.superHeroes = [
    "Iron Man",
    "Mowgli",
    "Spiderman",
    "Superman",
    "Chhota Bheem"
];
```

Values of the array are used as options on select drop-down. The *md-option* element/directive shows each item on the drop-down. Consider the following template code. The *md-option* iterates through the array for each option on the drop-down. It uses Angular's ng-repeat to iterate.

```
<md-select ng-model="selectedSuperHero" placeholder="Select your favorite
super hero">
    <md-option ng-repeat="item in superHeroes">{{item}}</md-option>
</md-select>
```

See Figure 7-7 for the drop-down rendered.

Figure 7-7. *A drop-down*

The default text before selection is specified on the placeholder attribute of *md-select* element. *ng-model* on *md-select* will hold the selected value.

md-select allows multiple selection. Set the multiple attribute value to true for making it a multiselect drop-down. See Figure 7-8.

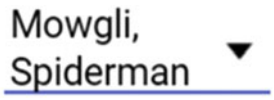

Figure 7-8. *Drop-down with multiple values selected*

Dynamically Retrieve Drop-down Options

Consider the following sample. It uses attributes *md-on-open* and *md-on-close* attributes. They expect an expression as an input. If a function (on scope) is provided, it will be invoked when the drop-down is open and when the selection is complete, respectively. The *md-on-open* could be used to load drop-down options dynamically.

```
<md-select ng-model="selectedSuperHero" md-on-open="loadSuperHeroes()" md-
on-close="superHeroSelectionComplete()" placeholder="Select your favorite
super hero">

   <md-option ng-value="item" ng-repeat="item in superHeroes">{{item.
   name}}</md-option>

</md-select>
```

Also, consider the following controller function.

```
.controller('selectSampleController', function($scope, $timeout){
    $scope.loadSuperHeroes = function(){
       $timeout(function(){
            $scope.superHeroes = [
                {id:1, name:"Iron Man"},
                {id:2, name:"Mowgli"},
                {id:3, name: "Spiderman"},
                {id:4, name:"Superman"},
                {id:5, name:"Chhota Bheem"}
               ];
            },1000);
    };

    $scope.superHeroSelectionComplete = function(){
        console.log("Selected Hero - " + $scope.selectedSuperHero.name);
    };
```

The timeout is mimicking time delay if we are retrieving options from an external API over HTTP.

Notice that in this sample we are using objects for options instead of simple strings. A new attribute, *ng-value*, is set on md-options.

```
<md-option ng-value="item" ng-repeat="item in superHeroes">{{item.name}}
</md-option>
```

With this, when the user selects an option, the complete object is set on *ng-model*. Remember, the *ng-model* variable name is *selectedSuperHero*. It holds the selected option. The *superHeroSelectionComplete* function is invoked once the user makes a selection from the drop-down options. As the *ngModel* now holds an object of superhero, it is logging *$scope.selectedSuperHero.name.*

More Options

1. While strings were used for drop-down options, a comparison with selected option is simple. However, with objects, it is a shallow equality check, which means that the selected object on model is not equal to the object in options array.

   Use an additional option, trackBy in ng-model-options, to fix this. We could use id or any other similar field on the model.

   ng-model-options="{trackBy:'$value.uniqueId'}" // uniqueId is a property on model object or the item in options array.

2. Select options could be grouped using md-optgroup directive/element. Use the label attribute to set group title. Consider the following code and Figure 7-9.

```
<md-select ng-model="selectedSuperHero"  md-on-open="loadSuperHeroes()" md-
on-close="superHeroSelectionComplete()" placeholder="Select your favorite
super hero">
    <md-optgroup label="Marvel">
        <md-option ng-value="item" ng-repeat="item in superHeroes|filter:'Ma
        rvel'">{{item.name}}</md-option>
     </md-optgroup>

     <md-optgroup label="Disney">
         <md-option ng-value="item" ng-repeat="item in superHeroes|filter:'D
         isney'">{{item.name}}</md-option>
      </md-optgroup>

     <md-optgroup label="Indian">
     <md-option ng-value="item" ng-repeat="item in superHeroes|filter:'India
     n'">{{item.name}}</md-option>
       </md-optgroup>
</md-select>
```

It creates three option groups: Marvel, Disney, and Indian. Consider the model object on scope as well.

```
$scope.superHeroes = [
        {id:1, name:"Iron Man", category: "Marvel"},
        {id:2, name:"Mowgli", category: "Disney"},
        {id:3, name: "Spiderman", category: "Marvel"},
        {id:4, name:"Superman", category: "DC Comics"},
        {id:5, name:"Chhota Bheem", category: "Indian"}
    ];
```

Figure 7-9. *Drop-down with groupings*

Autocomplete Drop-down

This directive/element allows the user to start typing to filter values in a drop-down. See Figure 7-10. We could even dynamically retrieve values from a server-side API. Consider states in the United States. Assuming the list is available in the browser (in a JSON object), as the user starts typing, the controller could filter and show a subset of values that match a given string. On the other hand, if it is a stock ticker, the number of available values is huge. Hence, the controller can initiate a server API call on each key store (or set of key strokes) and filter the results.

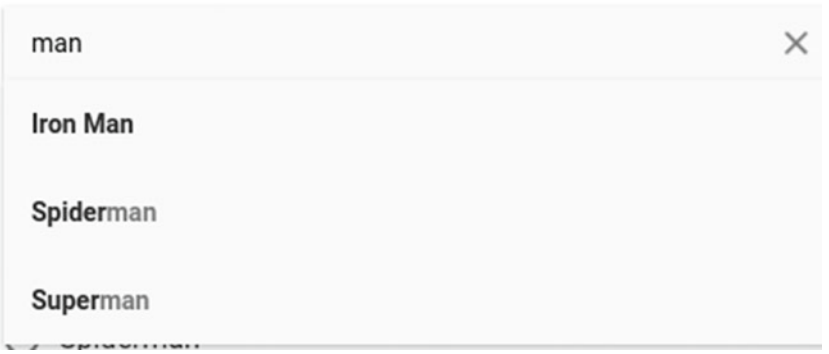

Figure 7-10. *Filter in a drop-down*

Use *md-autocomplete* element/directive. It is a parent element, which includes a template (*md-item-template*) and a placeholder message (*md-no-results*) when filter does not yield results. Use *md-item-template* element/directive for showing filtered results. When there are no results for given filter string, use *md-not-found* for placeholder message.

Consider the following code.

```
<md-autocomplete md-search-text="filterString" md-items="item in filterSuper
heroes(filterString)">
       <md-item-template>
            <span md-highlight-text="filterString">{{item}}</span>
      </md-item-template> <md-not-found>
          No results.
     </md-not-found>
</md-autocomplete>
```

The variable set on *md-search-text* property will hold keyed-in text in filter string. In the sample, filter String property on *$scope* will hold the filter text keyed in by the user. The *md-items* attribute loops through results. Each result item is represented by a variable named *"item"*.

filterSuperheroes() is a function on $*scope* with the logic to filter by a given string. This function takes filterString from *md-search-text* to apply the filter. Note that this variable is on *$scope* as well. We could also use it directly on *$scope*, instead of a parameter to *filterSuperheroes* function.

md-item-template holds an HTML template for each item in the drop-down. Optionally, use *md-menu-class* attribute and specify the CSS class to be applied on drop-down items. In the code sample, for simplicity, we are using a span for item template.

Highlight the Filter Results

The *md-highlight-text* property helps highlight given text. Here, we pass the filterString variable on *$scope*. That would result in keyed-in text on autocomplete highlighted on items in the drop-down.

We could provide additional options for highlight logic. Consider using attribute md-highlight-flags. Possible options are:

1. "i" for case-insensitive match on highlighted text.
2. "^" for highlighting only items beginning with the filter string.
3. "$" for highlighting only items ending with the filter string.

More Options

1. As user selects an option from the drop-down, use *md-selected-item* to hold the selected item. This variable could be used in other elements on the markup or even in the controller. The following sample shows the selected item in the template.

```
<strong>User selected {{selectedVariableOnScope}}</strong>
<md-autocomplete md-selected-item="selectedVariableOnScope" ... >
</md-autocomplete>
```

2. *md-no-cache* could be used to disable implicit caching of results in the directive.
3. In case of making an API call to get filtered results, it is advisable to make calls after a given number of milliseconds. Otherwise, each keystroke could result in a call to the API, which could be too many for the server to handle. Use *md-delay* to specify this delay in milliseconds.
4. Another variation useful while making API calls is *md-min-length*. Set a number representing the number of characters. Only after user keys in so many characters is filter logic applied.
5. Use *md-input-maxlength* and *md-input-minlength* to set a limit on filter string. This limit is applied for validation only.
6. Use *md-select-on-match="true"* to autoselect an item if exact filter string has been typed in.

Chips

Chips is a good UI for selecting multiple custom values. We see this commonly used for tags on a document or an article, recipient list in an e-mail, and so on. See Figure 7-11.

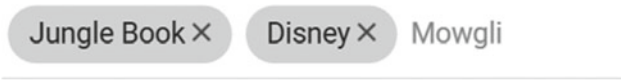

Figure 7-11. *Chips*

Create chips using *md-chips* element/directive. Set array on *ng-model* to show them as pre-existing tags. User can add new chips and remove the existing. The *ngModel* is updated accordingly.

```
<!-- Template -->
<md-chips ng-model="tagsOnMowgli">

// Variable on scope in the controller.
$scope.tagsOnMowgli = ["Jungle Book", "Disney"];
```

Use *placeholder* and *secondary-placeholder* attributes. When secondary placeholder is present, the placeholder value is shown only if the chips text field is empty. If there is at least one chip, secondary placeholder value is shown. See Figure 7-12 and Figure 7-13. The former does not have a chip selected. Hence, the placeholder value is shown. With two chips selected in Figure 7-13, value from the secondary placeholder is shown (Mowgli).

Figure 7-12. *Empty field for chips*

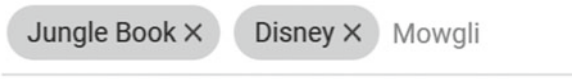

Figure 7-13. *With chips selected, placeholder text changed*

Consider the following code.

```
<md-chips ng-model="tagsOnMowgli" placeholder="AddEnter tags on Mowgli"
secondary-placeholder="Mowgli">
</md-chips>
```

Use shorter secondary placeholder. If there is a chip already, more often than not, the user will understand the purpose already. The user will not need a complete explanation (which is required when the text field is empty).

Like any other control, we could mark chips read-only with *readonly* = *true*. At this point, the user cannot add or remove chips.

Transform Chips

As the user keys in values for chips, it is a good idea to transform them to be consistent. For example, consider a state field. Users key in state names in various forms. Some might key in Minnesota, while others provide MN.

We could write a transform function to get state code and replace it with full string provided by the user. Consider the following controller function.

```
$scope.transformChip = function(chip){
  if(chip){
      for(var state in $scope.uSStateList){
          if($scope.uSStateList[state] === chip
                || state === chip){
              return state; // return state code when you find a match.
          }
      }
      return null; // return null - so the chip won't be added.
  }
};
```

The transform function returns null, indicating that if the given string is not a name in the US states list, it will be ignored and not added as a chip. However, if the given string matches one of the state names, state code will be returned.

There could be other scenarios where given value should be added as a chip even when there is no match. To handle such a case, return undefined instead of null.

Use the following code, *md-transform-chip* attribute, to invoke the transform function while adding a chip. The $*chip* represents the chip being added.

```
<md-chips name="states" ng-model="selectedStateList" md-transform-
chip="transformChip($chip)" placeholder="Enter States" secondary-
placeholder="States">
```

Use *md-on-add*, *md-on-remove*, or *md-on-select* to execute an expression or run a function when user adds, removes, or selects a chip, respectively.

Custom Templates

We could customize a chip by using a template. Use *md-chip-template* element. Consider the following code. It is adding additional text to the given chip value.

```
<md-chips name="states" ng-model="selectedStateList" md-transform-
chip="transformChip($chip)" placeholder="Enter States" secondary-
placeholder="States">
 <md-chip-template>
     Send the package to <span>{{$chip}}</span>
 </md-chip-template>
</md-chips>
```

Again, the *$chip* represents the chip being added.

Contact Chips

Chips use input field while adding/editing/removing chips. See Figure 7-14. Contact chips use an autocomplete drop-down, so that values could be selected from a list.

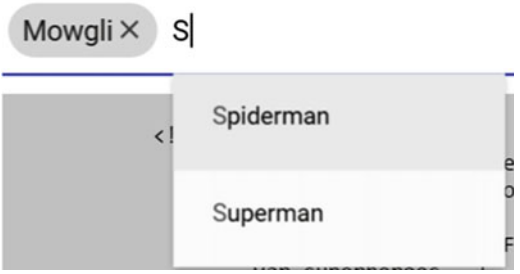

Figure 7-14. *Autocomplete in chips*

Consider the following code. Controller functions first:

```
$scope.filterSuperheroesForContactChip = function(filterString){
    var superheroes = [
            {id:1, name:"Iron Man", category: "Marvel"},
            {id:2, name:"Mowgli", category: "Disney"},
            {id:3, name: "Spiderman", category: "Marvel"},
            {id:4, name:"Superman", category: "DC Comics"},
            {id:5, name:"Chhota Bheem", category: "Indian"}
        ];
    if(filterString){
       return superheroes.filter(function(item){
            return item.name.toLowerCase().indexOf(filterString.
            toLowerCase()) >= 0;
        });
    }else{
        return superheroes;
    }
```

Out of available array of superhero objects, the filter function selects ones that match the filter query and returns them.

An attribute *md-contacts* is invoking the filter function. It could get a list of objects or a promise. If promise is returned, until it is resolved a loading indicator is shown. In the preceding sample, the template gets a list of objects. *md-contact-name* is one of the fields on the template. The value name is passed, which is a property on each item in the drop-down.

```
<md-contact-chips ng-model="contactChipSuperHeroes" md-contacts="filterSup
erheroesForContactChip($query)" md-contact-name="name" placeholder="Select
your superhero"></md-contact-chips>
```

Other possible values on the template are *md-contact-email* and *md-contact-image*. Also, consider using *ng-model* to get a list of all selected chips.

Similar to *md-chips*, *secondary-placeholder* is another attribute on the element. It could be used to set a placeholder when there is at least one chip selected.

Radio Buttons

Use *md-radio-group* and *md-radio-button* directives/elements to create Angular Material–style radio buttons. The later (radio buttons) are grouped as child elements of radio group. See Figure 7-15 for Angular Material–styled radio buttons.

Consider the following code.

```
<strong>Selected {{radioSelectedSuperHero.name}}</strong>
<md-radio-group ng-model="radioSelectedSuperHero">
    <md-radio-button ng-value="item" ng-repeat="item in
    radioSuperHeroes">{{item.name}}</md-radio-button>
</md-radio-group>
```

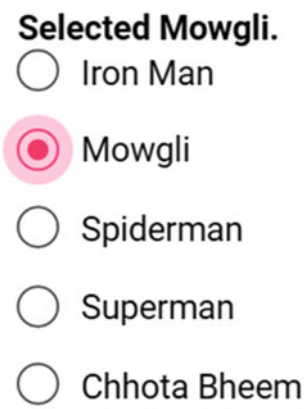

Figure 7-15. *Radio buttons*

The selected radio button/option is set to *ng-model* on radio button group. Use ng-repeat on *md-radio-button* and iterate through object array. In the preceding sample, *ng-value* on the radio button is set as the whole object. Hence, when selected whole object is set on *ng-model* (of radio group). Hence, while showing the selected value use *selecteditem.name*.

■ **Note** A radio button uses accent color by default.

Check Box

Use *md-checkbox* directive to show a check box. Consider the following code sample. It iterates through superheroes array and shows check boxes. See Figure 7-16.

```
<div ng-repeat="item in checkSuperHeroes">
   <md-checkbox ng-model="item.isChecked">{{item.name}}</md-checkbox>
</div>
```

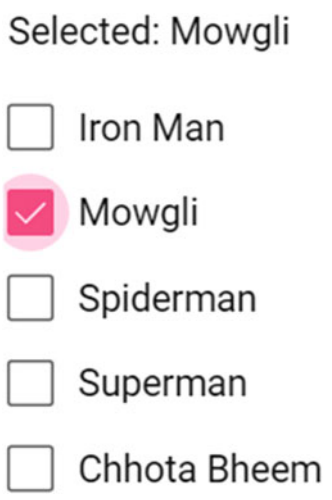

Figure 7-16. *Check boxes*

ng-model on the check box sets *true* or *false* based on selection. However, if you need to set different values than true or false, use *ng-true-value* and *ng-false-value*. For example:

```
<md-checkbox ng-model="selected" ng-true-value="MowgliSelected" ng-false-
-value="MowgliUnselected">Mowgli</md-checkbox>
```

Value of selected (on ngModel) will be MowgliSelected or MowgliUnselected based on the user action.

A check box could be in a third state (other than selected or unselected): that is indeterminate. To represent an unknown value or partially selected set, this could be used. When *ngModel* is not set or when *md-indeterminate* attribute is set to true, check box will be in this state.

Slider

Use slider to set a value on a range. Imagine feedback on a scale of one to ten. Slider is a good choice for representing such an item on the form. See Figure 7-17.

Figure 7-17. *Slider*

Use *md-slider* directive to create a slider. Consider the following code.

```
<md-slider min="1" max="10" step="1" ng-model="spidermanRating">Spiderman
</md-slider>{{spidermanRating || 'unrated'}} on a scale of 10
```

The attributes *min* and *max* specify lowest and highest values possible on the control. *ngModel* holds the input (rated) value. "*step*" represents the increment/decrement as the user drags the slider. As the value set is 1, each drag will increment the value by one. We can even use decimal values like 0.5.

Use *md-discrete* to show selected value as a bubble on the slider itself. See Figure 7-18.

Figure 7-18. *Slider*

At the time of writing this book, the *md-vertical* attribute is added in a release candidate for 1.0.10 to show a flipped slider. It could be used for controls such as a volume slider.

Date Picker

For date values in a form, date picker is effective and useful. See Figure 7-19 for Angular Material date picker.

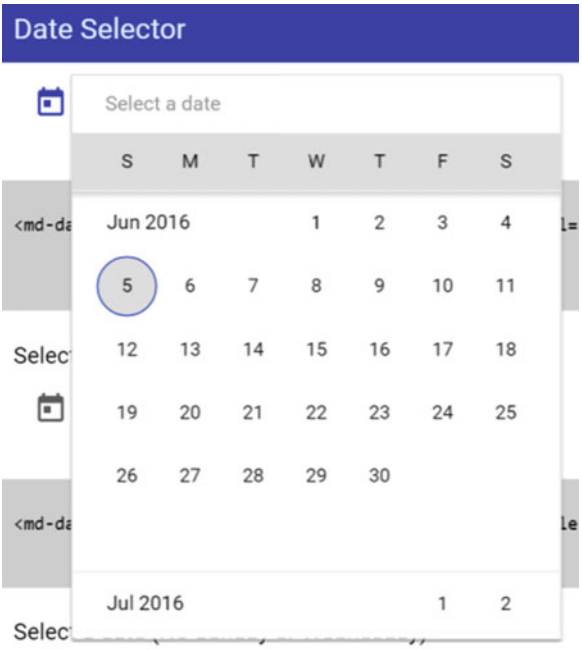

Figure 7-19. *Date picker*

Use *md-datepicker* to create a date picker. Consider the following code.

```
<md-datepicker md-placeholder="Select a date" ng-model="dateValue">
</md-datepicker>
```

Use *md-placeholder* for default text before selecting a date. *ng-model* has the selected date by the user.

We can restrict selecting within a date range. Use *md-min-date* and *md-max-date* attributes to do so. See Figure 7-20.

Figure 7-20. *Date picker with min date and max date set*

```
<md-datepicker style="width:400px" md-placeholder="Select a date" ng-
model="dateBetweenYears" md-min-date="minDate" md-max-date="maxDate">
</md-datepicker>
```

```
$scope.minDate = new Date('01/01/2016');//min date
set to 1st Jan '16

$scope.maxDate = new Date(); // Max date set to
current date.
```

As shown in Figure 7-20, dates after the current date are disabled.

We could code a custom validator. Consider the following sample. It does not let the user select a Sunday or a Wednesday. Use this option to code custom rules that are specific to an application. See Figure 7-21.

Figure 7-21. *Custom rules for date selection*

Use the attribute *md-date-filter*. In the following sample, validateDate is a function on $scope.

```
<md-datepicker md-placeholder="Select a date" ng-model="noWedSun" md-date-
filter="validateDate"></md-datepicker>
```

The validateDate function has access to the selected date, which is passed in as a parameter. The code in the function checks the day for whether a given date is a Sunday or Wednesday and returns true only if it is not one of those days.

```
$scope.validateDate = function(selectedDate){
    if(selectedDate.getDay() == 0 || selectedDate.getDay() == 3){
        return false; // return false for Sunday and Wednesday
    }else{
        return true; // return true for all other days.
    }
}
```

Summary

This chapter described many elements and directives needed to create forms on a web page. These are the most-used input controls. They conform to an Angular Material style of coding. They provide a Material Design user experience all along.

We began by using a simple input text box and wrapping it in *md-input-container* element. It allows creating Angular Material–style input elements. We explored validation controls and services. We could make use of validation across the form and provide out-of-the-box support for showing error messages, when the user keys in incorrect and inconsistent values.

Check boxes, radio buttons, and drop-downs explored in this chapter are traditional HTML controls; however, we used Angular Material directives to get Material Design styling. The multiselect feature of drop-down is highly useful in many forms we create. The following special controls provide additional flexibility and a better user experience while working a form.

1. Autocomplete drop-down makes it easy for the user to quickly select the option he/she is looking for. The filter allows quick narrow-down to the desired option.
2. Chips provide a great user experience while the user is making free-form selection of items like tags or e-mail addresses. Unlike a drop-down, any text could be provided as input. This is true in most cases, as there are ways to restrict users to provide predefined values only.
3. Slider allows quick drag approach to select a value on a range. An example would be feedback on a scale of one to ten.

Finally, we explored date picker control, the *md-datepicker* element/directive for Angular Material–style date selector.

References

For Angular Material documentation, use `https://material.angularjs.org`

CHAPTER 8

Lists and Alerts

In this chapter, we will explore some more Angular Material directives and services. We will begin with list and grid list. They are quite effective controls to represent complex data. The controls are easy to use and are highly configurable. They provide flexibility in look-and-feel and functionality.

We will then detail dialog and toast elements that alert and inform data to the user. An application will have a variety of alerting requirements. Some are information only. Certain others are critical and need user attention. The Angular Material elements/directives cater many of these requirements. Adhering to Material Design specifications, they provide efficient controls.

List

List view is a frequently used UI element. Angular Material provides directives which give the list UI and functionality out of the box. See Figure 8-1.

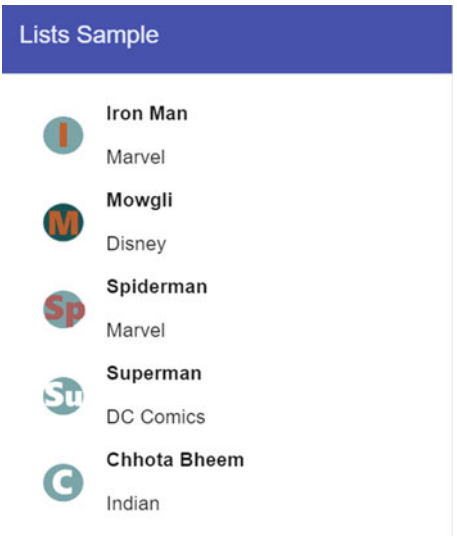

Figure 8-1. *Angular Material list*

V. K. Kotaru, *Material Design Implementation with AngularJS*,
DOI 10.1007/978-1-4842-2190-7_8

Use directives *md-list* and *md-list-item* to create a list. The directive/element *md-list* is a container of multiple list items. Use *md-list-item* to create each list item. Consider the following code sample.

```
<md-list>
        <md-list-item class="md-2-line" ng-repeat="item in superheroes"
        ng-click="null">
        <img ng-src="{{item.imageUrl}}" class="md-avatar">
        <div class="md-list-item-text" layout="column" >
            <strong>{{item.name}}</strong>
            <div>{{item.category}}</div>
        </div>
    </md-list-item>
</md-list>
```

Using *ng-repeat*, we iterate through the array of items (superheroes) on *md-list-item*. Consider using CSS classes *md-2-line* or *md-3-line*. As the name suggests, they define the number of lines of text in each list item. The CSS class adjusts the height of the list item accordingly.

Optionally, we can show an image on each list item. If we show an image, use the following CSS classes.

1. **md-avatar** for an image file. It could be a png or jpeg file.
2. **md-avatar-icon** for an icon to be used as an image on the list item.

Using the preceding CSS classes, fit the image to the list item (with respect to height and width). Review Figure 8-1, which demonstrates how images fit to the height and width of the list item.

The text part of the list item, titles, description, and so on should be wrapped in any HTML element (say div) with *md-list-item-text* CSS class applied on it.

Notice *ng-click* on each *md-list-item*. As the value provided is null, on click, no action is performed. Nevertheless, we may provide a controller function. Use it to navigate to the details screen after clicking a list item. We may choose to exclude *ng-click* altogether. If we do so, the list item will not show a highlight on hovering. Moreover, clicking does not generate an animation (like a button click).

If we need to show a paragraph of information in each list item, maybe similar to description text, use *md-long-text* CSS class. See Figure 8-2.

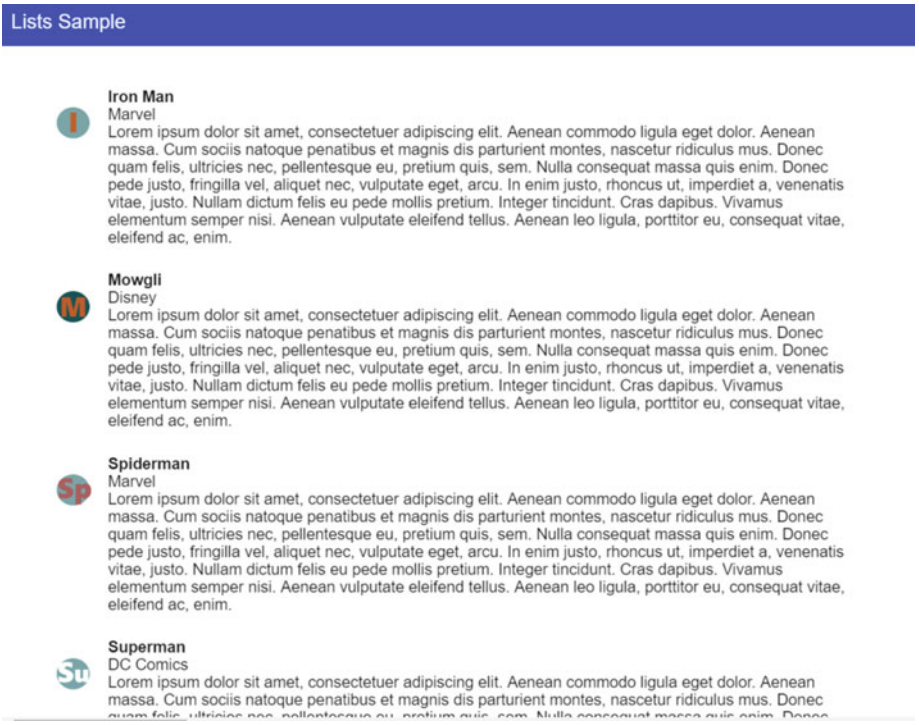

Figure 8-2. *List box with long text formatting*

The following is another sample with some of these variations.

```
        <md-list>
            <md-list-item class="md-3-line md-long-text" ng-repeat="item in
            superheroes">
<!-- no ng-click-->
                <img ng-src="{{item.imageUrl}}" class="md-avatar">
                <div class="md-list-item-text" layout="column" >
                    <strong>{{item.name}}</strong>
                    <div>{{item.category}}</div>
                    <div>Lorem ipsum ... </div>
                </div>
            </md-list-item>
        </md-list>
```

Further exploring options with lists, we could use a secondary button on a list item. It is aligned to the right end of the row. It could act similar to more action buttons. Consider the following code and Figure 8-3.

```
<md-list>
    <md-list-item class="secondary-button-padding" ng-repeat="item in
    superheroes" ng-click="buttonClickHandler()">
        <img ng-src="{{item.imageUrl}}" class="md-avatar">
        <div class="md-list-item-text" layout="column" >
            <strong>{{item.name}}</strong>
            <span>{{item.category}}</span>
        </div>
        <md-button class="md-secondary" ng-click="secondaryButtonClickHandl
        er()">
           Actions
        </md-button>
    </md-list-item>
</md-list>
```

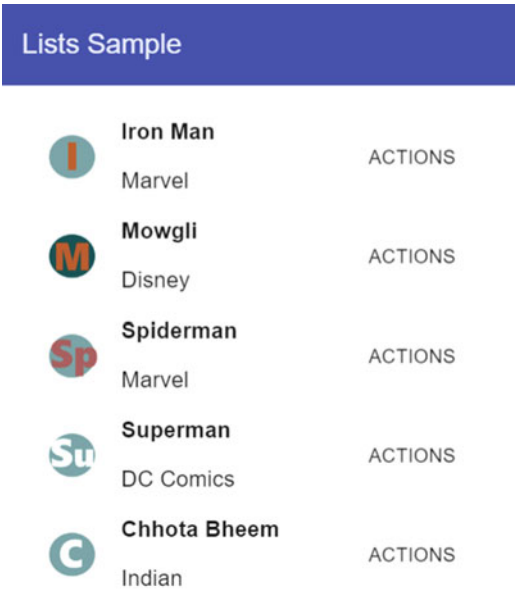

Figure 8-3. *List with secondary action buttons*

On the list item element/directive, using a CSS class *secondary-button-padding* aligns the secondary button to the right edge of the screen. Another CSS class *md-secondary* is applied on the button. You may choose to make the whole list item a primary button. If done, the secondary button action will not trigger primary action even though it is part of the row. It will trigger secondary action in isolation. In the sample, if we click anywhere on the list item, it triggers *buttonClickHandler* (this is a function defined on *$scope*). If we click the "Actions" button, it triggers *secondaryButtonClickHandler* in isolation.

Grid List

Grid list is a special representation of data compared to list view. It highlights content better and fits descriptive text and images on each item in the list.

Angular Material provides a grid component, set of directives out of the box. It takes a different approach from ng-grid, which (ngGrid) supports functionalities like sort and filter. Consider this grid as an alternative to list. It is a different and unique depiction of data. Each list item here is represented as a tile. See Figure 8-4.

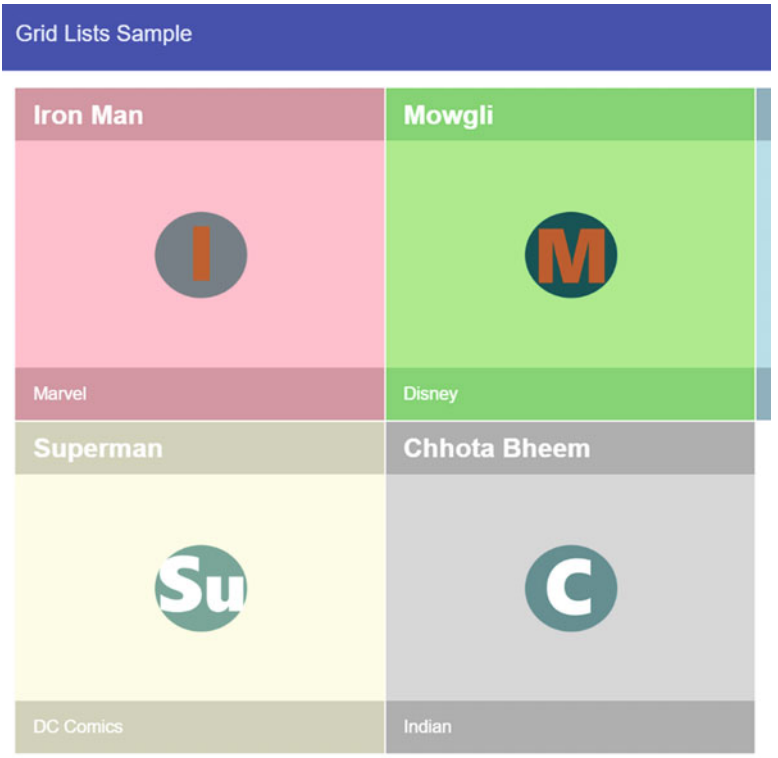

Figure 8-4. *Grid list sample*

The following is a basic sample.
Consider the following code.

```
<md-grid-list md-cols="2" md-row-height="300px">
    <md-grid-tile ng-class="item.background" ng-click="null" ng-repeat="item
    in superheroes">
        <img style="max-width:100px" ng-src="{{item.imageUrl}}" />
        <md-grid-tile-footer layout-padding>
            <span>{{item.category}}</span>
        </md-grid-tile-footer>
        <md-grid-tile-header layout-padding>
            <h2>{{item.name}}</h2>
        </md-grid-tile-header>
    </md-grid-tile>
</md-grid-list>
```

Grid List Element (*md-grid-list*)

md-grid-list directive encapsulates all the elements of a grid list.

1. Use an attribute *"md-cols"* to configure number of columns or cells in a row. In the preceding sample, we have two columns.
2. Consider making the grid responsive. Configure different numbers of columns depending on screen size. There is another example later in the chapter, which demonstrates this behavior.
3. Configure row height with an attribute *md-row-height*. In the preceding sample, we set it to 300px. We could set a value in rem or ratio of width and height. For example, 38rem and 3:4 are valid values.
4. Configure gutter size between tiles with the *md-gutter* attribute. Provide a value in px (pixels).

Grid Tile Directive (*md-grid-tile*)

Each tile/item/cell is represented by a child element/directive *md-grid-tile*. In the preceding sample, we are iterating through the *superheroes* array (on scope) to create multiple grid tile elements.

A grid tile may contain the following. These are child elements/directives for a grid tile.

1. **Header:** Use *md-tile-header* for header on the tile. In general, it could be used for showing a title on the tile.
2. **Footer:** Use *md-tile-footer* for footer on the tile. In the sample, we are showing category information on the footer.
3. **Workspace:** We could show text or images in a tile. In the sample, we show images of superheroes.

It is not necessary to use all the child elements of *md-grid-tile*. A tile may contain a header, a footer, both of them, or neither of them. All possible combinations can be used.

On *md-grid-tile,* consider using attributes md-colspan and md-rowspan. This specifies number of columns or rows a given tile should occupy. This allows defining different sizes for each tile. We may choose to show a bigger first tile. Alternatively, a dynamic grid list layout where each tile's size is defined by length of the content inside.

The following sample takes a simplistic approach. It marks the beginning of the grid list with a bigger tile. On the first tile, row span value is set to two. See Figure 8-5.

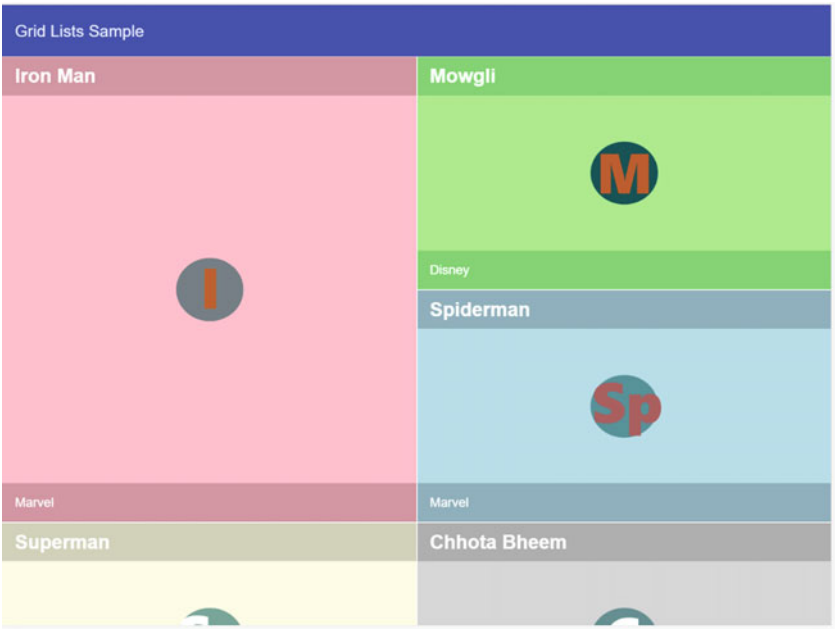

Figure 8-5. *Grid list with row span on first tile*

Consider the following code. Code in boldface uses an expression to identify the first tile and resize to occupy two rows. *ng-repeat* tracks index of an item while iterating through the list using a variable *$index*. If the value is 0, set rowspan to 2; for all other tiles set it to 1 (default size).

```
<md-grid-tile md-rowspan="{{($index===0)?2:1}}"  ng-class="item.background"
ng-click="null" ng-repeat="item in superheroes">
                <img style="max-width:100px" ng-src="{{item.imageUrl}}" />
```

■ **Note** *md-colspan* value should not be greater than md-cols value set on the parent *md-grid-list.*

Responsive Attributes

Use responsive attributes to let the grid list adapt to different screen sizes. See Figure 8-6 and Figure 8-7.

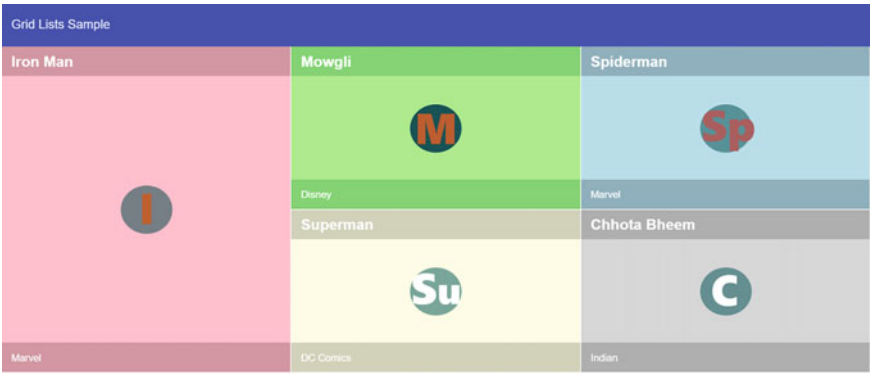

Figure 8-6. *Three columns on a large screensize (gt-md)*

Figure 8-7. *Single column and default row span for first tile on a small screen*

Consider the following code.

```
<md-grid-list md-cols-sm="1" md-cols-md="2" md-cols-gt-md="3" md-row-
height="16:9">
        <md-grid-tile md-rowspan-gt-sm="{{($index===0)?2:1}}"
        ng-class="item.background" ng-click="null" ng-repeat="item in
        superheroes">
        <img style="max-width:100px" ng-src="{{item.imageUrl}}" />

        <md-grid-tile-footer layout-padding>
            <span>{{item.category}}</span>
        </md-grid-tile-footer>
        <md-grid-tile-header layout-padding>
            <h2>{{item.name}}</h2>
        </md-grid-tile-header>
    </md-grid-tile>
</md-grid-list>
```

As the boldface code shows, *md-cols* sets the number of columns to one on a small screen, two on a medium-size screen, and three on a greater-than-medium screen. Also notice that on md-grid-tile, the rowspan logic described previously is applied to greater-than-small screens only. Thus, on a small screen a default value of one is applied for row span.

Alerts and Dialogs

Alerts and dialogs are important UI elements. To the user, they provide information or alert a critical action and allow acceptance of a final confirmation or provide a choice to select from a series of options. We could even show a pop-up window (not a browser window) within the page with any additional content. The service and elements/directives being discussed in this section will help create Material Design style dialog boxes in Angular Material. See Figure 8-8.

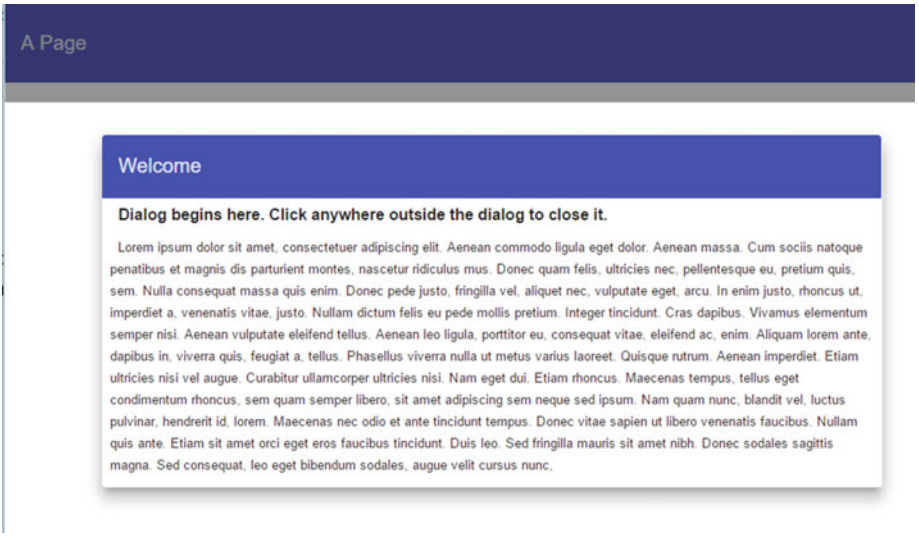

Figure 8-8. *An elaborate Angular Material dialog*

Consider the following code. This template has been used in the dialog window.

```
<script type="text/ng-template" id="dialogTemplate.html">
   <md-dialog>
     <md-toolbar>
         <div class="md-toolbar-tools">
             <h2>Welcome</h2>
         </div>
     </md-toolbar>
     <md-dialog-content class="md-dialog-content">
            <strong>
             Dialog begins here. Click anywhere outside the dialog to close it.
            </strong>
            <br/>
            <br/>
            <sup>Lorem ipsum …</sup>
     </md-dialog-content>
   </md-dialog>
</script>
```

The template is referenced as *dialogTemplate.html*.

md-dialog Element

A dialog is encapsulated in *md-dialog* element/directive. It may contain one or more child elements/directives.

1. To show dialog content, use the element/directive md-dialog-content. On it, apply a CSS class *md-dialog-content* if padding is required.
2. Group all dialog action buttons (for example, OK, Cancel buttons) in *md-dialog-actions* element/directive.
3. These two elements take care of alignment and positioning for the dialog.
4. Like any other window, for a toolbar and title use *md-toolbar* element/directive.

The preceding template could be in a separate HTML file. For the sample, it has been included in the index.html page (in a template script) and provided with id "*dialogTemplate.html*". This template or the dialog is not shown until we explicitly make a call to show.

Inject a service *$mdDialog* into the controller. Use the *show* function on the service to show the dialog.

```
$mdDialog.show({
    templateUrl: "dialogTemplate.html",
    parent: angular.element(document.body), // dialog is a child element
    of body
    clickOutsideToClose: true
});
```

Notice *clickOutsideToClose* set to true. If this value is set to false, we might have to add a button to close the dialog (in *md-dialog-actions*).

The following are certain other configurations we can take advantage of, while showing a dialog.

1. **template**: Use *template: '<!-- template -->'* to provide template as a string instead of linking a template file.
2. **onComplete**: function(){} - Use onComplete callback function to run an action after show is complete.
3. **openFrom**: "#idOfTheElement" - The animation that depicts opening dialog zooms in from this element. This element could be placed on left, right, center (center is default for the dialog anyway), or any corner of the page.
4. **closeTo**: "#idOfTheElement" - The animation that depicts closing dialog zooms out to this element. This element could be placed on left, right, center (default for dialog anyway), or any corner of the page.

Pass Values to the Dialog

Many times, the launching controller will have dynamic values to pass to the dialog. Also, using a separate controller for the dialog will isolate its logic from parent controller. It will be a better separation of concerns. Consider the following code.

```
$scope.message = "Good Morning";
$mdDialog.show({
    templateUrl: "dialogTemplate.html",
    parent: angular.element(document.body), // alert is a child element of
    body

    clickOutsideToClose: true,
    locals: {
       aTitle: $scope.message
    },
    controller: function($scope, aTitle){
        $scope.title = aTitle;
    }
});
```

The "*locals*" variable can hold one or more value providers. Here we added *aTitle* to it. And *aTitle* is injected into the controller. Use value from *aTitle* on *scope*. Refer to the following template code to see the scope variable used in the template.

```
<md-dialog-content class="md-dialog-content">
    <h2>{{title}}</h2>
    <strong>
        Dialog be... <!-- rest of the template as in earlier example -->
```

Alert Dialog

Alert is used for information purposes and will only contain an OK button to close the dialog.

Consider Figure 8-9 and the following code. This approach allows creating an alert reference and reusing it. We create an alert with the following code. It is not shown to the user yet.

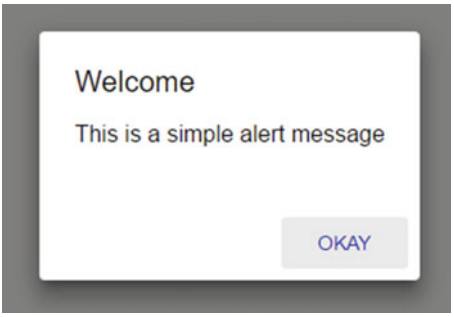

Figure 8-9. *Simple Angular Material–styled alert box sample*

```
var aSimpleAlert = $mdDialog
    .alert()
    .title("Welcome")
    .textContent(" This is a simple alert message")
    .ok("Ahaa")
;
```

Use *alert* function on $*mdDialog* to create an alert instance. Set title and content of the alert using *title* and *textContent* functions, respectively. If an OK button needs to be shown, use the API/function *ok* to set text for the button.

Show the Alert

Show the alert when needed. It could be used when the user clicks a button or on certain other events that require the alert to be shown.

```
$mdDialog.show(aSimpleAlert);
```

As mentioned, *$mdDialog.alert()* creates an alert/dialog. The sample uses more API functions (title, textContent, and ok) to configure the dialog. There are more sophisticated API available on an alert object. Consider the following.

1. **htmlContent('**<!-- **Html template** -->**'):** Unlike textContent API, we can provide HTML templates to this function. Note that ngSanitize module needs to be added as a dependency to use this function.
2. **templateUrl('templates/yourTemplate.html'):** Specify a template file for the alert. Template should have alert's content.
3. **theme('themeName'):** Provide a theme name as parameter to apply a custom theme while showing the dialog box.

4. **clickOutsideToClose(true):** Passing a value true allows the dialog to close when clicked outside the alert box. Default is false. Hence, an action button to close the dialog is needed.
5. **ariaLabel('a descriptive message'):** Sets a string as a description for accessibility requirements. Screen reader/tools use this string.
6. **openFrom("#idOfTheElement"):** The animation that depicts opening alert zooms in from this element. This element could be placed on left, right, center (default for the dialog anyway) or any corner of the page.
7. **closeTo("#idOfTheElement"):** The animation that depicts closing alert zooms out to this element. This element could be placed on left, right, center (default for the dialog anyway) or any corner of the page.

Hide the Alert

To hide an alert, use *$mdDialog.show(alertDialogReference)*. Consider the following code. It hides the dialog 5 seconds after showing it.

```
$scope.showSimpleMessage = function(event){
    $mdDialog.show(aSimpleAlert);
    $timeout(function(){
        $scope.hideAlertMessage();
     }, 5000);
};
```

Confirm Dialog

Confirm allows the user to make a choice. It could be used for warning messages, which allow the user to make a final choice. See Figure 8-10.

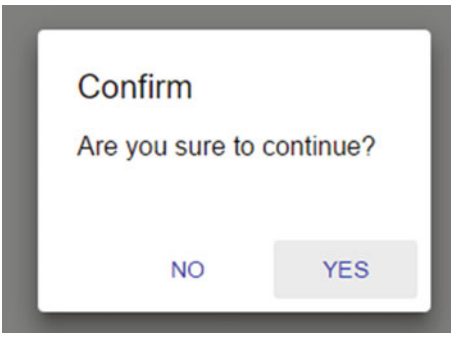

Figure 8-10. *Angular Material–styled confirm dialog*

Consider the following code. Similar to alert, we create an object of the confirm dialog. We could reuse it across the controller (whichever is the scope of the object).

```
var aConfirmDialog = $mdDialog
.confirm()
.parent(angular.element(document.body))
.title("Confirm")
.textContent(" Are you sure to continue?")
.ok("Yes")
.cancel("No")
.openFrom("#left")
;
```

Create a confirm dialog reference using *confirm*() function on *$mdDialog* service. Configure multiple aspects about the dialog.

1. Set parent using *parent*() API. The sample sets the whole body element in the HTML as the parent.
2. Similar to alert, set title and textContent on the confirm dialog.
3. Unlike alert dialog, confirm has two buttons: OK and Cancel. Set text on the button using ok() and cancel() API.
4. ariaLabel('a descriptive message'): Sets a string as a description for accessibility requirements. Screen reader/ tools use this string.
5. Use openFrom(elementReference) to open the confirm dialog from given element in the template. The open animations zoom in from this element in the template. It could be placed on left or on right or at any corner of the page.
6. Use closeTo(elementReference) to close the confirm dialog to a given element in the template. The close animations zoom out to this element in the template. It could be placed on left or on right or at any corner of the page.
7. theme('themeName'): Provide a theme name as parameter to apply a custom theme while showing the dialog box.
8. htmlContent('<!-- Html template -->'): Unlike textContent API, we can provide HTML templates to this function. Note that ngSanitize module needs to be added as a dependency to use this function.
9. templateUrl('templates/yourTemplate.html'): Specify template file for the confirm dialog.

Show the Confirm Dialog

Show the confirm dialog using *$mdDialog.show* API. However, it returns a promise. As the user makes a selection, it will call success callback or error callback if user selects OK or Cancel button, respectively. Consider the following code.

```
$mdDialog.show(aConfirmDialog).then(function(result){
    console.log(result + " - User selected yes"); // Success callback on OK
    click. Output will be "true - User selected yes".
}, function() {
    console.log("User selected no"); // Error callback on Cancel click
});
```

Notice success callback accepts a result variable as parameter. When the user selects yes, the result value will be true.

Hide the Dialog Box

To hide the dialog box, use *$mdDialog.hide* API. Consider the following code. In the sample, we close the dialog automatically after 5 seconds. This is only to demonstrate using hide.

```
$timeout(function(){
    $mdDialog.hide(aConfirmDialog);
}, 5000);
```

It resolves the promise with success callback. However, result value will not be true but rather will contain the whole confirm dialog object. This can be used to check if user closed the dialog by clicking OK or it was closed by the alternate logic to hide the dialog.

Toast

Toast is another approach to interact and provide information to the user. More often than not, toast does not obstruct the user's actions. It pops up a message at a corner and shows the required information. Toast can automatically close itself (disappear) after a few seconds. See Figure 8-11.

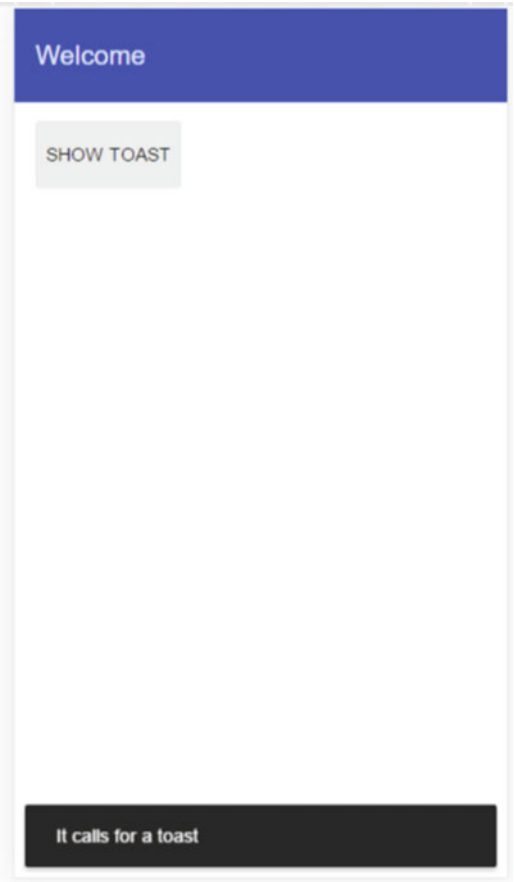

Figure 8-11. *A toast message at the bottom, on a mobile view*

It might not suit critical information messages. It is possible that the user might miss the information popped up by the toast. Systems generally will have another view or screen to show alerts shown by toast. However, toast provides a quick non-obstructive way to show the information to the user.

Angular Material service *$mdToast* encapsulates toast functionality.

To show a simple toast, consider using the following code. It uses *$mdToast* service, which is injected into the controller.

```
.controller('toastSampleController', function($scope, $mdToast){
    $scope.showBasicToast = function(){
        $mdToast
            .showSimple("It calls for a toast")
            .then(function(){
                console.log("Done with the toast");

            });
    }
```

The *showSimple* function shows a ready-made toast with the given message. The toast will appear at the bottom of the screen.

Notice that the function returns a promise. Promise is resolved once toast disappears. If an action needs to be performed after the toast message, we can write code here. In the sample, we are logging a message ("Done with the toast") to the console.

Basic Customizations

We can customize and reuse a simple toast message. *$mdToast* exposes an API simple, which returns an object *$mdToastPreset*. This object stores the configured toast. We could reuse this toast message in the application.

Consider the following code. It creates *$mdToastPreset* object.

```
var toastRef = $mdToast
    .simple() // creates a toast message reference ($mdToastPreset)
    .textContent("Welcome to a reusable toast") // set text to be shown on
    the message.
    .capsule(true); // round edges
```

These functions are chained, which helps create the whole configuration with one line of JavaScript code. Notice the capsule API. It gives round edges to the toast. See Figure 8-12.

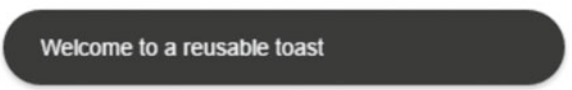

Figure 8-12. *Toast message styled with round edges*

Remember, this toast is not shown yet. Use the following line of code to show it on the screen.

```
$mdToast
    .show(toast)
    .then(function(){
        console.log("Done with the toast");
    });
```

Use *show* function for a preset toast. Similar to the *showSimple* function, show function too returns a promise, which is resolved as the toast is closed.

Additional Options with *simple()* API

The following are the additional options while creating a preset toast.

1. Action: A toast message can have an action button. See Figure 8-13. It has a button "star it" at the end of the row. The user could click or tap on it to perform related action for the toast message. For example, consider a "new message" alert by toast message; the user could star/favorite it using the action.

```
toast = $mdToast
    .simple()
    .textContent("A new message arrived")
    .action("Star it!");
```

Figure 8-13. *Toast messages with action defined on it*

As the user clicks the action button, promise resolved on the *show* function provides a value "*ok*" indicating button click.

```
$mdToast
    .show(toast)
    .then(function(result){
        console.log(result); // result will be "ok"
    });
```

2. Use highlightAction(true) API to highlight the button.
3. Use theme("themeName") to show the toast with a different theme (defined by *$mdThemingProvider*).

Advanced Customizations

Use *build* API/function to take advantage of advanced options with toast messaging. Similar to *simple*(), *build*() function also returns $*mdToastPreset* object. We could reuse the object across the controller (whichever is the scope of the object). Consider the following code.

```
var toastPreset = $mdToast
      .build()
      .template("<md-toast> Advanced Toast</md-toast>")
      .position("top right")
      .hideDelay(10000);

$mdToast.show(toastPreset);
```

Here, we are chaining the result to add additional configuration.

Unlike *simple*() API, build allows configuring a template (instead of just a text message). Use template function to provide HTML for the template. However, the *md-toast* element/directive is mandatory in the template.

Use *templateUrl*('url/to/the/template.html') to configure a template file.

Configure a Controller

Toast can have its own controller. Use *controller*() API and pass a function. It helps better isolation and separation of concerns from the parent controller.

```
$mdToast
   .build()
   .controller(function($scope){
           // Controller definition
    });
```

Hide a Toast Message

hideDelay() configures the number of milliseconds the toast should stay on. We can provide a value of 0, which will show the toast forever. In this scenario, create an action button to close the toast. The action button needs to be created in the template.

To close the toast, use *$mdToast.hide(toastPresetObject).* Add this line in a controller function, which is called upon clicking the action button.

Consider the following code.

```
 var toastPreset = $mdToast
       .build()
       .template("<md-toast> <strong flex='85'> <sup>Highly </sup>advanced
       toast </strong> <md-button ng-click='closeToast()'>Close </md-
       button></md-toast>")
        .hideDelay(0)
        .controller(function($scope, $mdToast){
             $scope.closeToast = function(){
                 $mdToast.hide(toastPreset);
             };
         });

$mdToast.show(toastPreset);
```

Position a Toast Message

Position() API allows configuring the position of the toast on the page. Possible values are *top*, *left*, *right*, *bottom*. We could use a combination of these values, that is, "*top left*", "*bottom right*", and so on. See Figure 8-14 for the toast positioned top-right.

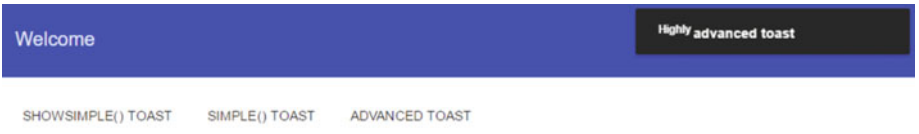

Figure 8-14. *Position a toast message*

However, on a sm screen (small) it resets the position to bottom. This fits better with small screens. See Figure 8-15.

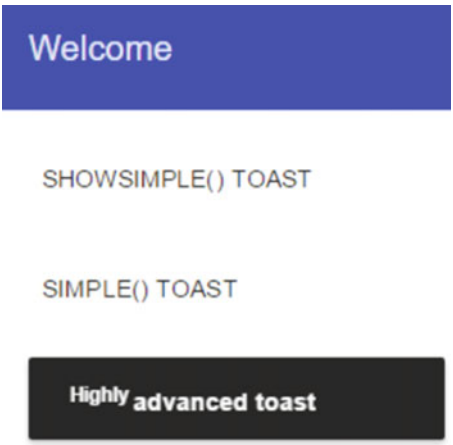

Figure 8-15. *On a mobile view, toast positioned at the bottom*

However, if you need to show it at a particular position on all screen sizes, use the following approach. Let that position be *"top right"* all the time or in the middle of the page, next to the element the user is interacting with.

Use *parent()* api/function and set a particular element on the HTML page as the parent for the toast. That enforces the toast shown at that element's position all the time.

Consider the following sample. This shows toast in the div element, which is in the middle of a page, in the workspace.

```
<div id="advancedButton">
  <md-button ng-click="showAdvCustomizedShow()">Advanced Toast</md-button>
 </div>

var toastPreset = $mdToast
         .build()
         .position("top right")
         .parent(angular.element(document.getElementById("advancedButton")))
         .template("<md-toast> <strong> <sup>Highly </sup>advanced toast
         </strong></md-toast>")
         .hideDelay(0);

$mdToast.show(toastPreset);
```

The div element has an id "advancedButton". It is selected in the JavaScript code with *document.getElementById()* function. The selected element is passed as a parameter to *parent()* (for toast) function.

Show with Options

Consider using show function without a *toastPreset*. It is best suited when toast references are not reused.

```
$mdToast.show({
    position: "top right",
    parent: angular.element(document.getElementById("advancedButton")),
    template: "<md-toast> <strong flex='85'><sup>Highly </sup>advanced toast
    </strong> <md-button ng-click='closeToast()'>Close </md-button>
    </md-toast>",
    hideDelay: 0,
    locals: {
        dynamicValue: parentVal
    },
    controller: function($scope, $mdToast, dynamicValue){
        console.log(dynamicValue);
        $scope.closeToast = function(){
            $mdToast.hide(toastPreset);
        };
    }
});
```

■ **Note** Similar to alert dialogs, a dynamic value can be passed from parent controller using locals. The key value pairs in locals are injected into the controller as a value type.

Summary

This chapter details two important data representations in any web application: list and grid list. We start by exploring directives *md-list* and *md-list-item* for creating a list view. We discuss configuring and taking advantage of variations in the list view. Depending on the data that need to be represented onscreen, we could take advantage of CSS styling to show a two-line or three-line view of the control. We go through options to show even larger description (of a record). We also learn creating action buttons on list items.

Grid list represents data that are more complex and provides a different perspective. We explore *md-grid-list* and *md-grid-tile* elements/directives. We explore nuances of creating a set number of columns in a grid, configuring size, alignment, and so on. We also look at responsive attributes and making the grid adaptive to the screen size.

After data representation, we start exploring alerting or showing dialog boxes with Angular Material elements/directives. At first, we look at creating a dialog pop-up using *md-dialog* element/directive. We explore showing alerts for warning and information messages. We use *$mdDialog* service as well. We also go through using confirm dialogs.

Finally, we look at toast messages. Toast messages are more often than not used for information-only messages. We look at using *$mdToast* service and *md-toast* element/directive in the template. We begin by making use of ready-made API for creating a simple toast message. We also look at configuring and customizing the toast messages.

References

For Angular Material documentation use `http://material.angularjs.org`

CHAPTER 9

Mobile-Friendly Elements

Some of the controls and elements provided by Angular Material are very relevant on a mobile device. Technically, there is nothing stopping these controls from being used on a bigger screen, like a laptop or a desktop. However, they are designed to look better and more usable on a mobile device. Let us have a look at some of those controls.

Bottom Sheet

This is a directive or control for showing a menu of options. Typically, this is used upon clicking a button. The menu or list of available options pops up at the bottom of the screen. See Figure 9-1.

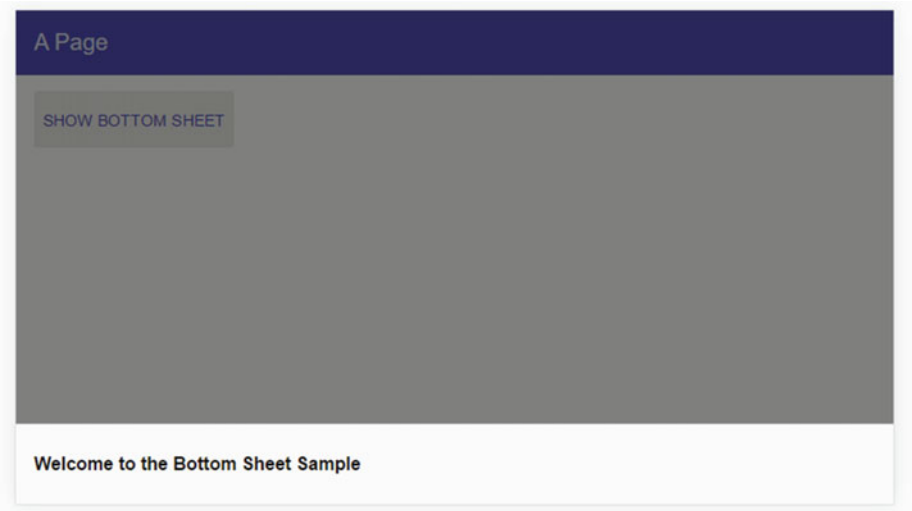

Figure 9-1. *A basic bottom sheet for demonstration*

V. K. Kotaru, *Material Design Implementation with AngularJS*,
DOI 10.1007/978-1-4842-2190-7_9

A directive *md-bottom-sheet* and a service *$mdBottomSheet* are used for this control. Consider the following code. It is basic implementation of bottom sheet. For simplicity in the first sample, the bottom sheet menu does not show any list of options. A better example is demonstrated later in the section to depict real usage.

```
angular.module("sampleApp", ["ngMaterial"])
     .controller('sampleController',function($scope, $mdBottomSheet){
         $scope.showBottomSheet = function(){
             $mdBottomSheet.show({
                 template: "<md-bottom-sheet> <strong> Welcome to the Bottom
                 Sheet Sample</strong> </md-bottom-sheet>"
             });
         };
     })
```

Notice that the *$mdBottomSheet* service is being injected in the controller. An API *show()* on this service draws the bottom sheet. We have a sample button in the HTML template, which invokes *showBottomSheet()* function on scope.

```
<md-button ng-click="showBottomSheet()" class="md-primary">Show Bottom
Sheet</md-button>
```

As the user clicks the button, the function on scope, *showBottomSheet()*, calls *show()* API on *$mdBottomSheet* service. We pass a JSON object with a template for the bottom sheet. The bottom sheet template must contain the *md-bottom-sheet* element/directive.

Let us now make it little more sophisticated. We could show buttons, a list of related actions in the bottom sheet. The component provides two default views for actions: list and grid.

Bottom Sheet—List View

Consider Figure 9-2. It shows the bottom sheet for login options.

Figure 9-2. *Bottom sheet depicting login options*

The following code is used for creating the sample.

```
angular.module("sampleApp", ["ngMaterial"])
    .controller('sampleController',function($scope, $mdBottomSheet){
       $scope.showBottomSheet = function(){
          $mdBottomSheet.show({
             templateUrl: "/bottom-sheet-template.html"
          });
       };
    })
```

Unlike the previous sample, here we are using template URL. This is a preferred option because as templates grow in size, it becomes difficult to fit it all in the template field (of the JSON object). Consider the following template.

```
<md-bottom-sheet class="md-list ">
     <md-subheader>
        Choose Login Mechanism
    </md-subheader>
    <md-list>
        <md-list-item>
            <div>
                <md-button >
                     <md-icon md-svg-src="icons/ic_verified_user_black_24px.
                     svg"></md-icon>
                     <span>UserId & Password</span>
                 </md-button>
             </div>
         </md-list-item>
         <md-list-item>
             <div>
                 <md-button >
                     <md-icon md-svg-src="icons/ic_fingerprint_black_24px.
                     svg"></md-icon>
                     <spanFingerprint</span>
                 </md-button>
              </div>
         </md-list-item>
         <md-list-item>
             <div>
                 <md-button >
                     <md-icon md-svg-src="icons/ic_camera_black_24px.svg">
                     </md-icon>
                        <span>Facial Recognition</span>
              </md-button>
          </div>
      </md-list-item>
     </md-list>
    </md-bottom-sheet>
```

Notice the CSS class *md-list* on the root element of the template, *md-bottom-sheet*. It ensures that the bottom sheet options are aligned vertically, like a list.

We are using *md-subheader* element for the header on the bottom sheet. We could use an h1, h2, or h3 element. Nevertheless, a subheader directive provides proper spacing and font size for the subheader. It is also a personal choice. We could choose to use a different element altogether.

The second element in *md-bottom-sheet* is the *md-list* itself. It has a list of options arranged by the list directive. We are using buttons with icons for descriptive information about each option.

Bottom Sheet—Grid View

The other view option with bottom sheet is to arrange actions on it like a grid. See Figure 9-3.

Figure 9-3. *Bottom sheet aligned to a grid*

Changes are straightforward. Compared to the list view, change the CSS class on *md-bottom-sheet* to *md-grid* (from *md-list*). Considering the available space on the bottom sheet, you might choose to remove textual titles below or adjust the font accordingly. The following are the snippets that are different from the preceding sample.

```
<md-bottom-sheet class="md-grid">
```

Here is the list item template in the sample.

```
<md-list-item>
   <div>
       <md-button >
            <md-icon md-svg-src="icons/ic_verified_user.svg"
            style="width:48px; height:48px; color:red;"></md-icon> <br/>
             <sup>User Id & Password</sup>
        </md-button>
    </div>
</md-list-item>
```

Handle Bottom Sheet Actions

The buttons on bottom sheet are only meaningful if an action occurs on clicking them. We need to code handlers for those events. Review *show()* function on $*mdBottomSheet* service. The JSON object passed in as parameter also accepts a controller. Here, we can write handlers for each button on the bottom sheet. Consider the following code.

```
$mdBottomSheet.show({
   templateUrl: "/bottom-sheet-template.html",
   controller: function($scope){
       $scope.defaultAuthenticationHandler = function(){
           console.log("Take user to login form");
       }
       $scope.fingerprintHandler = function(){
                        console.log("Let's authenticate user by
                        fingerprints");
      };
      $scope.facialRecognitionHandler = function(){
          console.log("Use camera and identify the user");
               };
         }
       });
```

We have three handler functions written on scope, namely, *defaultAuthenticationHandler, fingerprintHandler*, and *facialRecognitionHandler.*

Next, we bind them on the HTML template on *ng-click*. As the user clicks one of the three buttons, the appropriate handler is called. Consider the following HTML template code. For the sample, as the user clicks one of the buttons, the log statement is printed on the browser's console. In a full-fledged app, we can navigate to the appropriate login screen or invoke hardware functionality (for example, switch on camera for facial recognition).

```
<md-list >
    <md-list-item>
        <div>
            <md-button ng-click="defaultAuthenticationHandler()">
                 <md-icon md-svg-src="icons/ic_verified_user.svg"
                 style="width:48px; height:48px; color:red;">
                 </md-icon> <br/>
                <sup>User Id & Password</sup>
            </md-button>
        </div>
    </md-list-item>
    <md-list-item>
        <div>
            <md-button ng-click="fingerprintHandler()">
                      <md-icon md-svg-src="icons/ic_fingerprint.svg"
                      style="width:48px; height:48px; color:red;">
                      </md-icon><br/>
                 <sup>Fingerprint</sup>
            </md-button>
        </div>
    </md-list-item>
    <md-list-item>
        <div>
            <md-button ng-click="facialRecognitionHandler()">
              <md-icon md-svg-src="icons/ic_camera.svg" style="width:48px;
              height:48px; color:red;">
              </md-icon><br/>
                 <sup>Facial Recognition</sup>
            </md-button>
        </div>
    </md-list-item>
</md-list>
```

The *show()* API returns a promise. The promise is resolved or rejected on calling *hide()* or *cancel()* API, respectively. That is, *$mdBottomSheet.hide()* resolves the promise, whereas *$mdBottomSheet.cancel()* rejects the promise. Hence, in the handler we can invoke the appropriate API. Consider the following code.

```
$scope.fingerprintHandler = function(){
   console.log("Let's authenticate user by fingerprints");
   $mdBottomSheet.hide();
};
```

The fingerprint handler can call *hide()* API, indicating that the bottom sheet's job is done. If we have a cancel button, to recognize withdrawing the bottom sheet, use *cancel()* API. It rejects the promise. Consider the following code.

```
$mdBottomSheet.show({
   ... // show function definition
}).then(function(){
    console.log("Promise resolved"); // promise returned by show is
    resolved.
}, function(){
    console.log("Promise rejected"); // promise returned by show is
    rejected.
});
```

The following are additional options while using *$mdBottomSheet.*

1. Scope: Use it to pass scope values between parent controller and bottom sheet directive. If no value is provided, it will create an isolated scope for the bottom sheet.
2. preserveScope: true: Scope value is retained between multiple instances of show/hide of the bottom sheet.
3. clickOutsideToClose: false: Default value is true. When set to false, the bottom sheet will not be closed on losing focus or on clicking outside the control.
4. disableBackdrop: true: Default value is false. When set to true, does not fade the backdrop. See Figure 9-4.

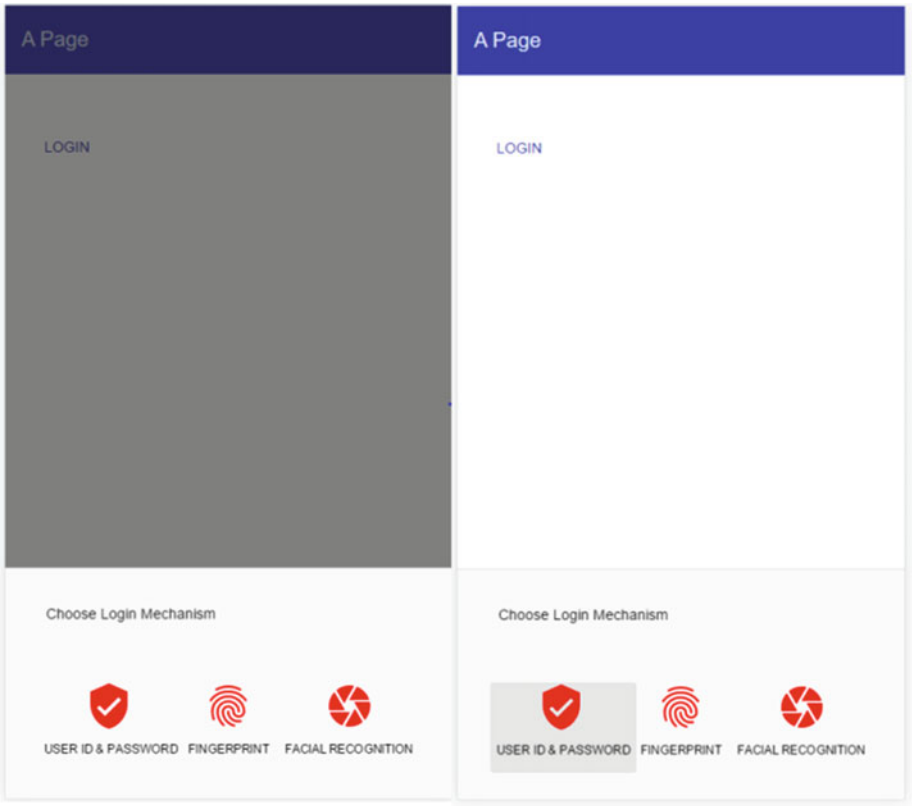

Figure 9-4. *Backdrop enabled on the left and disabled on the right*

5. escapeToClose: false: Default value is true. If set to false, does not close the bottom sheet on Esc key press on the keyboard.
6. disableParentScroll: false: Default value is true. If set to false, allows backdrop to scroll even when bottom sheet is open.

Swipe

Swipe is a common gesture with mobile devices. It in fact makes sense on any touch screen, even on a laptop with touch screen.

Angular material supports four directives that make it easy to support swipe in an application. Consider the following code.

```
<any md-swipe-left="swipeHandler()"></any>
```

It is an attribute directive. Use it on any element (for example div). It supports swipe gesture and calls the handler function attached on the *$scope*.

The following are the available directives.

1. md-swipe-left
2. md-swipe-right
3. md-swipe-up
4. md-swipe-down

Consider the following code snippet for usage.

```
<div md-swipe-right="swipeHandler()" class = "md-raised md-primary"> Swipe
Right </div>
<div md-swipe-left="swipeHandler()" > Swipe Left </div>
<div md-swipe-up="swipeHandler()" > Swipe Up </div>
<div md-swipe-down="swipeHandler()" > <sub> </sub> Swipe Down  <sub> </sub>
</div>
```

The sample is simplistic and just prints a console statement.

```
angular.module("sampleApp", ["ngMaterial"])
    .controller('swipeSampleController', function($scope){
       $scope.swipeHandler = function(){
           console.log("Swiped !");
       }
    });
```

However, in the real world, on swipe, we could perform actions like navigate to a new view/page (possibly swipe left for forward and swipe right for backward navigation), refresh view with new content (possibly swipe down), and so on. Figure 9-5 depicts swipe directions.

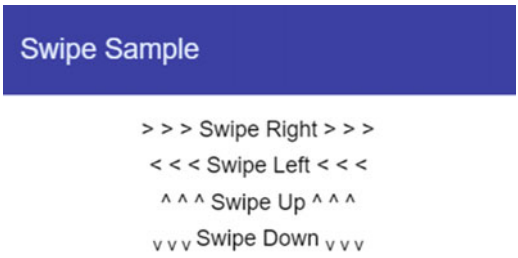

Figure 9-5. *Swipe directions*

Summary

Many Angular Material controls and features help build a responsive UI. The screen adjusts content based on the screen size. However, certain features provide mobile-specific behavior and functionality. These help to create a better mobile experience and easier use. It functionally works on bigger screens as well. However, they are designed for mobile views and touch screens.

This chapter elaborates mobile-specific services and directives initially. It details usage and features of *md-bottom-sheet* directive. We may perceive bottom sheet as a mobile-friendly menu. It has a set of actions or buttons that the user could tap on.

A bottom sheet supports two layouts: grid and list. Use CSS class *md-list or md-grid* to apply respective layout.

Then we delved into swipe functionality for touch screens. Angular Material provides four directives, which could be used as attributes on an HTML element. The directives are *md-swipe-left*, *md-swipe-right*, *md-swipe-up*, and *md-swipe-down*. On a touch screen, they have intuitive behaviour for moving forward, going back, pulling actions, and refreshing a screen, respectively. Implementing such gestures will be very effective for a touch screen.

References

For Angular Material documentation use `https://material.angularjs.org`

CHAPTER 10

Miscellaneous—Icons and ARIA

In this chapter, let us explore two important features of Angular Material. We will begin by looking at options for using icons in an application. Angular Material provides API, directives, and services to effectively download and show icons in an application. We will explore using SVGs and fonts for Icons.

Later in the chapter, we will explore ARIA, an important accessibility feature. Following ARIA standards, a web application will be easy to use for the visually impaired. Screen readers will make use of ARIA functionality in the web application for letting the differently abled use the app.

ngAria is an AngularJS module that provides accessibility features. Angular Material is highly dependent on this module and takes advantage of the features to the fullest.

Icons

Icons are widely used in web and mobile applications. Icons add visual clues on the screen and make the UI easily understandable. With an icon, users can instantly relate to the UI control. Many buttons or links show icons along with the title. Certain UI elements like tabs, icon buttons, and FAB controls may exclude text completely (possibly to save space on a smaller screen).

On a mobile screen, where the real estate needs to be used with austerity, icons become even more important. The rest of the chapter offers details on approaches to show icons in an Angular Material application. See Figure 10-1 for a set of Material Design icons shown on a page.

V. K. Kotaru, *Material Design Implementation with AngularJS*,
DOI 10.1007/978-1-4842-2190-7_10

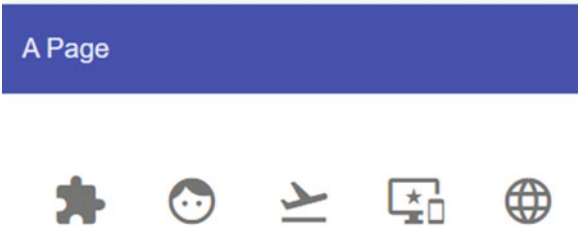

Figure 10-1. *Material icons on a web page*

Icon Fonts

It is one of the effective ways to show icons in an application. We could use font glyphs as icons in the Angular Material application. They help download fonts as a single unit, instead of downloading multiple images for each icon.

Using Material Design Icons (CDN Option)

To use Material Design icons as fonts, include the following style sheet in your project/page.

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons"
rel="stylesheet">
```

Use *md-icon* directive. Consider the following code. It uses the material-icons font set. The attribute *md-font-set* with a value *material-icons* specifies the same. Outside Angular Material, if you noticed HTML elements using font-set, it needs a special code mapped to each icon. However, with the *md-icon* directive we use a friendly name to select an icon (among many icons available in the font file).

```
<md-icon md-font-set="material-icons" style="font-size:42px" >face</md-icon>
```

Using Material Design Icons (with Files on Your Server)

As you might have also noticed, we are using CDN locations for style sheets and fonts. The preceding link element downloads the CSS, which in turn requests for font files.

If you prefer to render styles and fonts from your infrastructure rather than from the CDN, add the following styles to your project. This is copied from the previously mentioned material icons' link element, `https://fonts.googleapis.com/icon?family=Material+Icons`.

Create a style sheet with the following styles.

```
@font-face {
  font-family: 'Material Icons';
  font-style: normal;
  font-weight: 400;
  src: local('Material Icons'), local('MaterialIcons-Regular'), url(your-
  url/your-font.woff2) format('woff2');
}

.material-icons {
  font-family: 'Material Icons';
  font-weight: normal;
  font-style: normal;
  font-size: 24px;
  line-height: 1;
  letter-spacing: normal;
  text-transform: none;
  display: inline-block;
  white-space: nowrap;
  word-wrap: normal;
  direction: ltr;
  -webkit-font-feature-settings: 'liga';
  -webkit-font-smoothing: antialiased;
}
```

Notice the highlighted URL for font location. Preferably, use bower or npm to download Angular Material icons. Copy the WOFF2 file from the downloaded packages. The following is a bower install command for downloading the package.

```
bower install material-design-icons
```

Using Custom Icons

As specified earlier, for Material Design icons, we get the friendly name for an icon out of the box. If we need to use custom icons, copy your font files in a folder. Use a style sheet with class names for each icon. The following is a sample.

```
.icon-home:before {
    content: "\e900";
}
```

Use Angular Material element/directive *md-icon* and reference the CSS class as a value for *md-font-icon.*

```
<md-icon md-font-icon="icon-home" style="font-size:42px" ></md-icon>
```

■ **Note** The style sheet referenced previously and the icons downloaded are from IcoMoon app: https://icomoon.io/app/#. You could select a subset of icons and download only the ones you need. It also creates a style sheet with classes and codes for icons out of the box.

Using SVGs for Icons

SVG are vector graphics, which do not lose quality as we scale the image/icon up or down. If we prefer to use SVGs for icons, we could do that using an attribute *md-svg-src* on *md-icon* element/directive. Consider the following code.

```
 <md-icon md-svg-src="svg-icons/ic_alarm.svg" style="width:48px;
height:48px; color:red" ></md-icon>
```

Specify path to the SVG file on *md-svg-src* attribute. Consider using a custom CSS class or style to change the size and color of the icon. The sample has inline style for simplicity. However, a CSS class for all icons is preferable.

■ **Note** You could change color and size for both SVGs and font icons.

Ensure that SVG file does not have *fill* property set. If we open the SVG in a text editor, should see an XML file with various details of the image. Ensure that you remove the fill property if you choose to control color from the HTML view.

We could download Material Design icons from the following URL: https://design.google.com/icons.

Angular Material Icon Sets

We could also use icon sets to preload SVGs up front. We could load an icon set as a default icon set. That means we could reference the SVG icons by name anywhere in the HTML templates.

Consider the following code.

```
angular.module("sampleApp", ["ngMaterial", "material.svgAssetsCache"])
    .config(function($mdIconProvider){
       $mdIconProvider.defaultIconSet('img/icons/sets/core-icons.svg', 24);
    });
```

In the sample, we are using icon sets pre-created and available on CDN. The CDN location used in the sample is https://s3-us-west-2.amazonaws.com/s.cdpn.io/t-114/svg-assets-cache.js. All the icons in the sample are packaged in AngularJS module *material.svgAssetsCache*. As we create the sample module, add this module as a dependency. (*material.svgAssetsCache*: review the preceding highlighted code).

In the config function which runs as we bootstrap the module, inject *$mdIconProvider*. Using an API *defaultIconSet()*, we could set a default iconset. In the sample, we are considering *core-icons.svg* as the default icon sets. This icon set is available in svg-assets-cache.js.

Notice the second parameter, 24. It is the default view box size, in pixels for the given icon set. A bigger number shows a bigger icon by default.

Use *md-icon* element/directive and *md-svg-icon* attribute to use an icon out of the default icon set.

```
<md-icon md-svg-icon="account-balance" style="width:48px; height:48px;
color:red" ></md-icon>
```

Similar to earlier examples, we are keeping style attributes as is, except that the way SVG file loads has changed. Instead of referencing the SVG file directly, we are using a friendly name referenced in the icon set. See Figure 10-2.

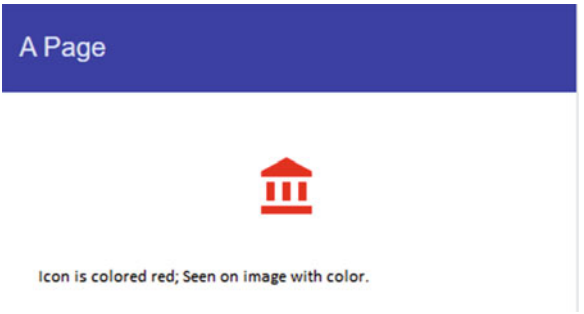

Figure 10-2. *Account balance icon with styling applied*

Additional Icon Sets

We could preload additional icon sets as well. Consider the following code.

```
angular.module("sampleApp", ["ngMaterial", "material.svgAssetsCache"])
 .config(function($mdIconProvider){
    $mdIconProvider
     .defaultIconSet('img/icons/sets/core-icons.svg', 24)
     .iconSet('communication','img/icons/sets/communication-icons.svg',24);

});
```

Here we used an API *iconSet*. It has an additional first parameter to name the icon set. In the sample, we are naming it *communication*. Consider the following HTML template.

```
<md-icon md-svg-icon="account-balance" style="width:48px; height:48px;
color:red" ></md-icon>
<md-icon md-svg-icon="communication:business" style="width:48px;
height:48px; color:red" ></md-icon>
```

Notice *iconsetName:icon-name* syntax (*md-svg-icon="communication:business"*). As we are not loading the icon from default icon set, we identify the icon set that has the given icon named business. See Figure 10-3 for a page rendered using this approach.

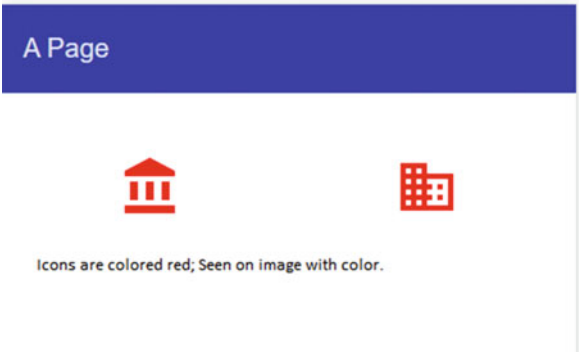

Figure 10-3. *Icon set sample*

There is an alternative syntax to show an icon in the icon set. Consider the following.

```
<md-icon md-icon-set="communication" >business</md-icon>
```

Specify the icon id as the value for *md-icon* element/directive. The icon set is provided through an attribute *md-icon-set*.

Preload Individual Icons

In the earlier section, we used *$mdIconProvider* to preload SVG icon sets. Using API provided by *$mdIconProvider*, we can load individual icons at the module bootstrap time. This helps preload all icons up front and the individual views load faster (when needed). Consider the following sample.

```
angular.module("sampleApp", ["ngMaterial", "material.svgAssetsCache"])
.config(function($mdIconProvider){
```

```
    $mdIconProvider
      .icon('account-circle','svg-icons/ic_account_circle.svg');
  });
```

Here, an icon ic_account_circle.svg is preloaded in the config function. This function is called as the sampleApp module bootstraps. The icon is available with an id *account-circle* in the application.

Consider the following code. We could now use the icon in the HTML template (with the image provided to the icon API earlier).

```
<md-icon md-svg-icon='account-circle' ></md-icon>
```

Font Sets

Similar to icon sets, we could also use font sets. By default, the font set '*angular material*' is applied on the application. We could use a different font set as the default using *$mdIconProvider* and its API *defaultFontSet()*.

Use the following.

```
$mdProvider.defaultFontSet('a-custom-font-set')
```

We could also load other font sets using *fontSet* API.

```
$mdProvider.fontSet('another-font-set', 'font-class-name')
```

While using another font set, in the HTML template specify the font set on md-icon directive.

```
<md-icon md-font-set="another-font-set" style="font-size:42px" >fontAlias
</md-icon>
```

■ **Note** Consider using *defaultViewBoxSize(aNumberValue)* API with *$mdIconProvider*. It changes icons' default view box size across the module. The value is in pixels.

md-icon directive internally uses a service *$mdIcon*. It is used for making an HTTP request for the icon and in turn cache it in the browser.

ARIA

ARIA is a W3C standard for allowing persons with disabilities access applications (especially web applications). AngularJS provides extensive support to ARIA through a module named *ngAria*. This module is a primary requirement of Angular Material. It is included as a module dependency with *ngMaterial* (Angular Material module). All directives and controls in Angular Material provide support for these features through *ngAria*.

If we get started with Angular Material using a package manager like npm, bower, or JSPM, *ngAria* is already downloaded. If we are using CDN, make sure to include the *ngAria* script. Consider the following URL. X.Y.Z are AngularJS versions.

```
//ajax.googleapis.com/ajax/libs/angularjs/X.Y.Z/angular-aria.js
```

ARIA features are usable through screen readers for persons with disabilities. The screen reader will read description and state of the control aloud. *ngAria* integrates well with *ngModel*, *ngChecked*, *ngValue*, *ngShow*, and so on. This means that when a check box is selected/checked, the screen reader through the ARIA attribute could pick this state information.

Consider the following code sample.

```
 <md-button ng-click="submitFunction()" aria-label="Submit customer
form">Submit</md-button>
```

We provide a description in *aria-label* attribute for screen readers. This should be done for all controls in the application.

For any control with label, screen reader can pick up the label text. Using *aria-label*, we can provide text that is more descriptive for screen readers. When there is no default label value, the *ngAria* API expects *aria-label* attribute on the control. If neither is on the control, it shows a warning in the browser (Figure 10-4).

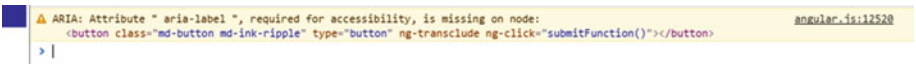

Figure 10-4. *Example browser warning*

Tab Index: ngAria sets tab index on controls automatically. This will help the accessibility feature to move focus to the next control using the keyboard.

Hidden for accessibility features: Any control not visible on the screen will also automatically be hidden from ARIA features. It is achieved by automatically applying an attribute *aria-hidden*="*true*". However, for any reason if it does not hide automatically (possibly it is hidden by changing opacity, which the ARIA logic will not pick up) or a control needs to be excluded from accessibility features, use the *aria-hidden* attribute explicitly.

Summary

This chapter explores icons in a little more detail. Icons have been used in earlier chapters as well. Nevertheless, they were basic implementations. When we need a holistic strategy for using icons in the application, review the approaches in this chapter.

We could use icon fonts, which would allow downloading icons as a unit. It also provides great ease of use. Material Design provides many frequently used icons out of the box. Of course, you could create and use custom fonts. This chapter described ways to implement the same.

If you decide to use SVG icons, with icon sets, all the needed icons could be grouped into a single file. On the other hand, the API also supports loading individual SVG icons. Unless a very small number of icons are used in your application, creating an icon set is preferred.

The chapter later describes ARIA features; *aria-label* is almost always seen on many directives and controls of the Angular Material application. Take advantage of the simplicity of *ngAria* and make your application accessible for the differently abled.

References

For Material icons, see https://design.google.com/icons
For IcoMoon, see https://icomoon.io/app/#
For accessibility and ngAria documentation, see https://docs.angularjs.org/guide/accessibility and https://docs.angularjs.org/api/ngAria
For ARIA specification and documentation, see https://www.w3.org/TR/wai-aria/

CHAPTER 11

Miscellaneous

The following are some useful miscellaneous elements and attributes in Angular Material. These are simple to use and provide value in improving the user experience.

Whiteframe

On an element, Whiteframe provides an elevated, 3D appearance with shadow effect. It is an attribute directive. It could be used on an element link div in HTML. See Figure 11-1.

Figure 11-1. *Whiteframes with various values of elevation*

Use an element/directive *md-whiteframe* to create a whiteframe. Consider the following code. For the four elements, dp values of 4 (default), 10, 16, and 24 are provided. Possible values are -1 to 24. The first value -1 will not show any elevation. It is useful if we dynamically need to remove the effect on an element.

```
<div md-whiteframe flex layout-margin style="min-height:200px">
   Default
</div>
<div md-whiteframe="10dp" flex layout-margin style="min-height:200px">
   Whiteframe - 10dp
</div>
<div md-whiteframe="16dp" flex layout-margin style="min-height:200px">
   Whiteframe - 16dp
</div>
```


V. K. Kotaru, *Material Design Implementation with AngularJS*,
DOI 10.1007/978-1-4842-2190-7_11

```
<div md-whiteframe="24dp" flex layout-margin style="min-height:200px">
   Whiteframe - 24dp
</div>
```

Tooltip

Tooltips can provide useful help on various controls in the application. In an unobtrusive way, it can provide additional information about a text field, button, and so on. For an Angular Material tooltip, see Figure 11-2.

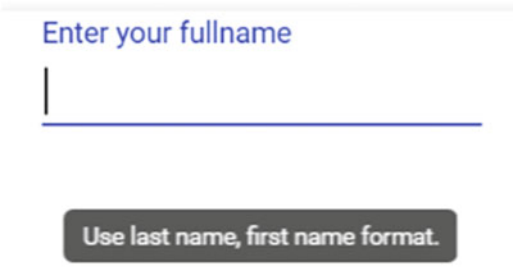

Figure 11-2. *Angular Material tooltip*

Use *md-tooltip* element/directive to create a tooltip. Refer to the following usage.

```
<md-input-container>
    <md-tooltip>Use last name, first name format.
    </md-tooltip>
    <label>Enter your fullname</label>
    <input type="text">
</md-input-container>
```

Consider using one of the following options with tooltip.

1. Use the attribute *md-direction*, with a value left, right, top, or bottom for explicitly positioning the tooltip. Default is bottom.
2. Use the attribute *md-delay* to delay showing the tooltip for a certain number of milliseconds after focus. This ensures that the tooltip is shown only when the form element really gets focus, instead of popping it while the user is in motion and quickly tabbing through the controls or moving the mouse cursor across.

3. Use the attribute *md-autohide* to let the tooltip disappear after moving the mouse out of the control's region.
4. Use the attribute *md-visible* with a value false to hide tooltip programmatically.

Subheader

Use element *md-subheader* to create a subheader. It is in addition to default HTML headers h1, h2, h3, and so on. Subheader indents a little to the right, indicating that it is a section under one of the headers.

md-subheader is simple to use. It has smaller font and emphasizes text for a subheader. See Figure 11-3.

Figure 11-3. *Headings and a subheader at the bottom*

Usage

```
< md-subheader > Sub Header </md-subheader>
```

Divider

It is a horizontal divider for sections of a view or screen. Use directive *md-divider*.

Usage

```
<div>Content Section A</div>
<md-divider></md-divider>
<div>Content Section B</div>
<md-divider md-inset></md-divider>
<div>Content Section C</div>
```

md-divider is an empty element that results in a horizontal divider as shown in Figure 11-4. Notice *md-inset* on the second divider. It results in an inset style divider. Refer to Figure 11-4.

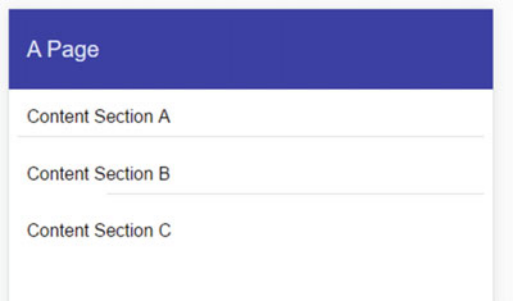

Figure 11-4. *Divider*

Progress Bar

Progress bars are an important aspect of any application. For a long-running process within the application, it provides important feedback to the user that the application is working in the background. The user will intuitively understand to wait and won't see a confusing frozen window. The long-running process could be a server-side API call, file IO, and so on.

Angular Material provides linear and circular progress bars.

Linear Progress Bar

A linear progress bar is laid out horizontally. As it comparatively has bigger screen space, will show elaborate status on the long-running process. See Figure 11-5.

Figure 11-5. *Linear progress bar*

Create a linear progress bar with a directive *md-progress-linear*. Angular material provides four modes for the progress bar.

1. *Determinate*: If we have definitive information on the percentage of long-running job complete, this mode will be effective. The progress bar fills up once the job is fully done. Consider the following code.

```
<md-progress-linear md-mode="determinate" value="{{progress}}">
</md-progress-linear>
```

The precentage of job done is determined by the value attribute. The following code mimics long-running process with a timeout service. Once the value of progress is above 100, it will quit. In the sample, it is a function defined in the controller.

```
              function updateProgress(){
                   $timeout(function(){
                       $scope.progress += 1;     // increment progress by 1.
                       if($scope.progress<=100){
// Continue to call update progress recursively till progress is 100%
                           console.log($scope.progress);
                           updateProgress();
                       }
                   }, 200);
              }
```

2. *Indeterminate*: When we are not certain about the percentage of job done, this is a better mode. Hide the progress bar once the job is complete. Till then, it keeps showing work-in-progress animation.

```
<md-progress-linear md-mode="indeterminate"></md-progress-linear>
```

3. *Buffer*: When there are two statuses to show, use this mode. For example, consider video streaming. The extent video buffered and the percentage of time that the user saw the video are two statuses. See Figure 11-6 for buffer mode depiction.

Figure 11-6. *Buffer mode*

Notice two progress statuses (light orange colored progress and dark orange colored progress). On the directive, while value represents percentage of job progress, the second buffer status is determined by *md-buffer-value* attribute. In the sample, we use another *$scope* variable buffer. Based on its value, the second status progress is shown.

```
<md-progress-linear md-mode="buffer" value="{{progress}}" class="md-
warn"  md-buffer-value="{{buffer}}"></md-progress-linear>
```

4. *Query*: Use this mode to depict pre-loading process. It demonstrates that the real job hasn't begun yet.

```
<md-progress-linear md-mode="query"></md-progress-linear>
```

Circular Progress Bar

A circular progress bar is another depiction of progress for a long-running process. See Figure 11-7 for a circular progress bar.

Figure 11-7. *Circular progress bar*

Use the directive *md-progress-circular* for creating a circular progress bar. There are two modes in which circular progress bar could be used.

1. *Determinate*: It shows the extent of long-running process complete. As the circle closes fully, the job is done 100%.

   Consider the following code. Use directive *md-progress-circular* with *md-mode* value "determinate" to create it in this mode. The extent of job completion is determined by value property.

```
<md-progress-circular md-mode="determinate" value="{{progress}}">
</md-progress-circular>
```

As for the linear progress bar, in the sample, we mimic a long-running process with a $timeout service.

```
function updateProgress(){
     $timeout(function(){
         $scope.progress += 1;     // increment progress by 1.
         if($scope.progress<=100){
// Continue to call update progress recursively till progress is 100%
              console.log($scope.progress);
              updateProgress();

         }
     }, 200);
}
```

2. *Indeterminate*: When we do not know the percentage of job done, use this mode. Hide the progress bar once the long-running process is fully done.

```
<md-progress-circular md-mode="indeterminate" ></md-progress-circular>
```

Summary

The chapter describes miscellaneous directives that are simple to use and are effective. We began by exploring attribute directive *md-whiteframe*. It provides 3D elevation and shadow effects to an element.

We explored md-tooltip for showing additional help with controls on the page. In a web application, use this directive with forms, buttons, links, tabs, and so on to show additional information about the control on mouse hover.

A divider could be useful in certain cases to show logical separation between controls and sections of the page. We use *md-divider* directive for the separator.

Finally, we explored linear and circular progress bars to show status on a long-running process. Such feedback to the user from the application is important. It avoids confusion, multiple form submits, unnecessary page refresh, and so on. We used directives *md-progress-linear* and *md-progress-circular*. Each has multiple modes to support different use cases with long-running processes.

References

For Angular Material documentation, use https://material.angularjs.org

CHAPTER 12

Responsive Design Patterns

In Angular Material, we have used multiple controls and elements that adapt Material Design principles and approaches. FAB is a popular component following Material Design principles. The speed dial among FAB controls is widely used in Material Design applications.

In this chapter, we will explore responsive patterns recommended by Material Design. They suggest an approach for adapting to multiple screen sizes, ranging from mobile phones to desktop computers. We use flexbox and Material Design features to achieve the same.

Reflow

This is a responsive design pattern recommended in Material Design. It allows controls and content to reflow or take up the available space on a screen.

The following example is useful on mobile devices in landscape and portrait modes. On mobile devices, a view or arrangement of controls that makes sense on a landscape mode might not always be relevant in portrait mode; the opposite is true too. There is a need to reflow the content that better fits landscape or portrait mode.

The example has two sections on the screen: one for selecting time and the other with a greeting message. The idea is to adapt to the layout change and reflow. A real-world example could be little more complex. However, the sample here is to get an idea on reflow pattern. See Figure 12-1 and Figure 12-2.

V. K. Kotaru, *Material Design Implementation with AngularJS*,
DOI 10.1007/978-1-4842-2190-7_12

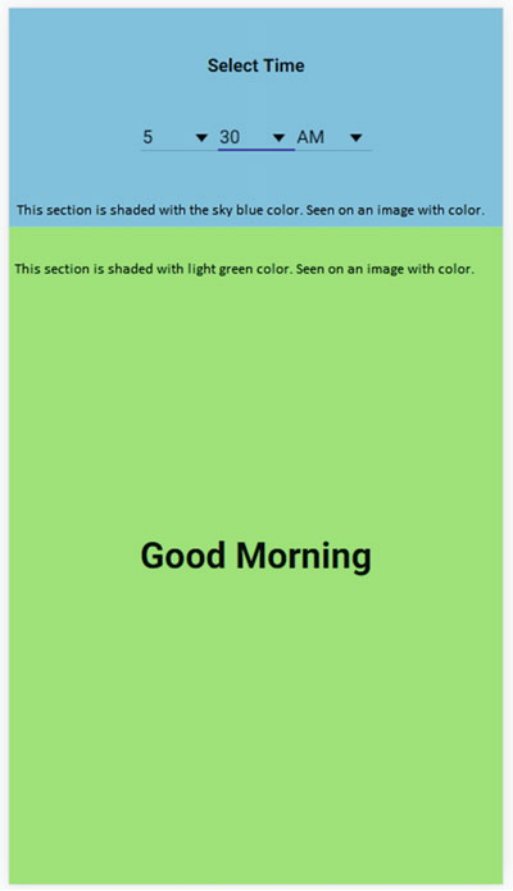

Figure 12-1. *A "reflow" sample in potrait mode*

We could achieve this layout by using layout attribute with a value "column". It has two child elements:

1. Sky blue colored time selection pane.
2. Light green colored greeting message pane.

Consider the following code.

```
<div flex layout="column" ng-controller="sampleController" >
    <div style="background-color:skyblue" flex="25" layout="column"
    layout-padding layout-align="center center">
        <strong>Select Time</strong>
        <!-- A div element that constructs time selection dropdowns.
        Removing code for better readability. Down below, complete
        snippet available. -->
   </div>

    <!-- following div element constructs the greeting message -->
    <div layout="column" layout-padding layout-align="center
    center"  flex="75" style="background-color:lightgreen">
        <h1>{{greetingMessage}}</h1>
    </div>
</div>
```

Notice the boldface text in the code. On the root element layout is set to be a column. There are two sections or cells in a column. The first section flexes to 25% and the second section flexes to 75%.

Now, let us make it reflow in a landscape mode. See Figure 12-2.

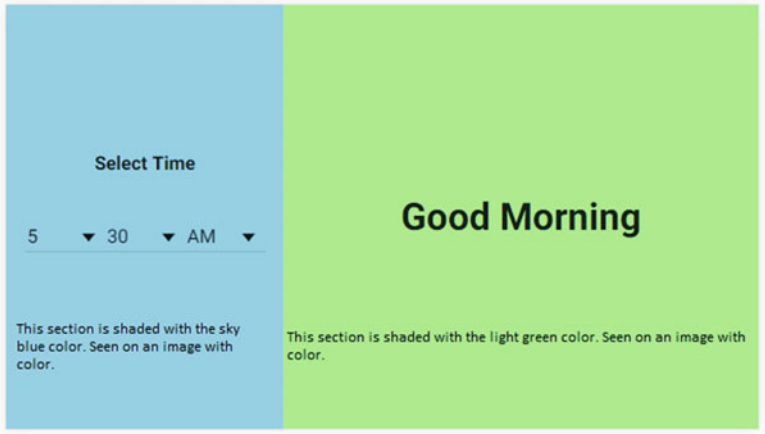

Figure 12-2. *A "reflow" sample in landscape mode*

When in landscape mode, the screen size is greater than extra-small. Now, reflow to a row. On the root element, set layout value to be row. It overrides default layout value without a breakpoint postfix (gt-xs).

```
<div layout-gt-xs="row" flex layout="column" ng-
controller="sampleController" >
```

This rearranges the two sections to become cells in a row instead of a column. Here is the complete HTML template code.

```
        <div layout-gt-xs="row" flex layout="column" ng-
        controller="sampleController" >
<!--
the rowFill CSS class forces the div to occupy full height in a row layout.
ng-class uses a variable isGreaterThanXs to check the screensize and hence
identify landscape mode
-->
            <div style="background-color:skyblue" ng-class="{rowFill:isGre
            aterThanXs}" flex="25" layout="column" layout-padding layout-
            align="center center">
                <strong>Select Time</strong>
                <div layout="row">
                    <md-select ng-model="selectedHour">
                        <md-option ng-repeat="hour in hours" ">
                            {{hour}}
                        </md-option>
                    </md-select>
                    <md-select ng-model="selectedMinute">
                        <md-option ng-repeat="minute in minutes">
                            {{minute}}
                        </md-option>
                    </md-select>
                        <md-select ng-model="selectedAMPM">
                        <md-option ng-repeat="item in AM_PM" ">
                            {{item}}
                        </md-option>
                    </md-select>
                </div>
            </div>
            <div layout="column" layout-padding layout-align="center
            center"  ng-class="{rowFill:isGreaterThanXs}" flex="75"
            style="background-color:lightgreen">
                <h1>{{greeting}}</h1>
            </div>
        </div>
```

Position

The pattern is about repositioning controls and actions to fit better with the view. Consider menu as an example. On a bigger and wider screen, menu on toolbar is effective. On a smaller mobile screen, a menu is difficult to work with, especially with a large number of actions that could result in a scrollbar. Position pattern advocates making it a more accessible control for a smaller screen. Here we make it a bottom sheet on a smaller screen.

Consider Figure 12-3 and Figure 12-4. We show menu on a larger and wider screen.

Figure 12-3. *Wider screen; show actions as menu*

We show menu options on a bottom sheet on a mobile view (Figure 12-4).

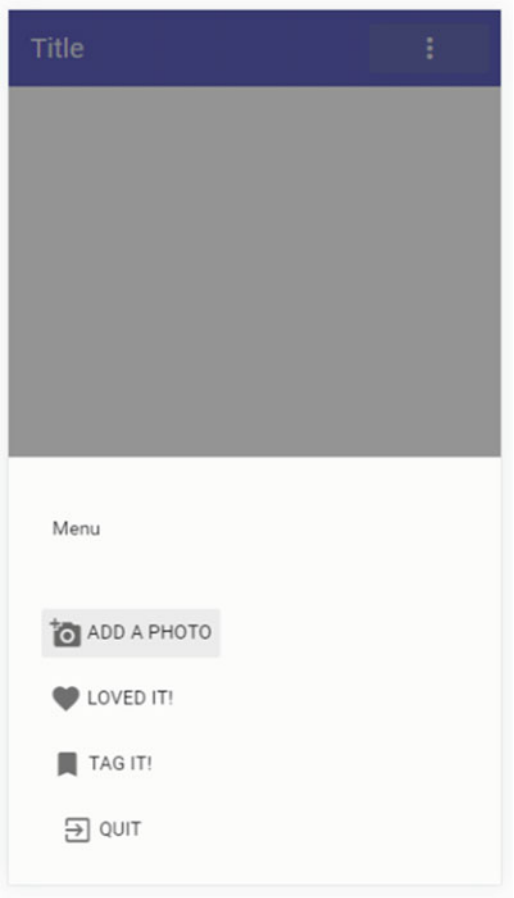

Figure 12-4. *Smaller form factor resulting in bottom sheet*

Consider the following code. Notice that "menu content" is hidden by default. It is shown on a screen greater than small (*gt-sm*), that is, medium and above.

```
<md-toolbar>
      <div class="md-toolbar-tools" layout="row">
          <h2>Title</h2>
          <span flex></span>

          <md-menu>
              <!-- Trigger element is a md-button with an icon -->
              <md-button ng-click="showActions($mdOpenMenu,$event); ">
                  <md-icon md-font-set="material-icons">more_vert
                  </md-icon>
              </md-button>
              <!-- hide details of menu for better readability -->
              <md-menu-content hide show-gt-sm>
                  <md-menu-item>
                     ...
                  </md-menu-item>
                 ...
              </md-menu-content>
          </md-menu>
      </div>
</md-toolbar>
```

Notice the *md-button* that triggers showing the menu. On clicking, we call a function on the associated controller. The controller function has further logic for position. Consider the following controller code.

```
$scope.showActions = function($mdOpenMenu, event){
       if($mdMedia('gt-sm')){
          $mdOpenMenu(event);
       }else{
           $mdBottomSheet.show({
              templateUrl: "/bottom-sheet-template.html"
          });

       }
      };
```

Notice the statement in the preceding code. On a screen size greater than small (*gt-sm*), that is, medium and larger, $*mdOpenMenu* will run. It results in showing a menu bar. Otherwise, on a screen size that is small and extra-small, we show the bottom sheet.

Review the complete code for a holistic understanding.

Template for bottom sheet.

```
<script type="text/ng-template" id="/bottom-sheet-template.html">
    <md-bottom-sheet class="md-list">
        <md-subheader>
            Menu
        </md-subheader>
        <md-list>
            <md-list-item>
                        <md-button ng-click="null" >
                            <md-icon md-font-set="material-icons">add_a_
                            photo</md-icon>
                            <span>Add a photo</span>
                        </md-button>
            </md-list-item>
            <md-list-item>
                        <md-button ng-click="null" >
                            <md-icon md-font-set="material-icons">favorite</
                            md-icon>
                            <span>Loved It!</span>
                        </md-button>

            </md-list-item>
            <md-list-item>
                        <md-button ng-click="null" >
                            <md-icon md-font-set="material-icons">bookmark</
                            md-icon>
                            <span>Tag It!</span>
                        </md-button>
            </md-list-item>
            <md-list-item>
                        <md-button ng-click="null" >
                            <md-icon md-font-set="material-icons">exit_to_
                            app</md-icon>
                            <span>Quit</span>
                        </md-button>
            </md-list-item>
        </md-list>
    </md-bottom-sheet>
</script>
```

Toolbar with menu included.

```
<md-toolbar>

    <div class="md-toolbar-tools" layout="row">
        <h2>Title</h2>
        <span flex></span>

        <md-menu>
            <!-- Trigger element is a md-button with an icon -->
            <md-button ng-click="showBottomSheet($mdOpenMenu,
            $event); ">
                <md-icon md-font-set="material-icons">more_vert </
                md-icon>
            </md-button>

            <md-menu-content hide show-gt-sm>
                <md-menu-item>
                    <md-button ng-click="null" layout="row">
                        <md-icon md-font-set="material-icons">add_a_
                        photo</md-icon>
                        Add a photo
                    </md-button>
                </md-menu-item>
                <md-menu-item>
                    <md-button ng-click="null" layout="row">
                        <md-icon md-font-set="material-
                        icons">favorite</md-icon>
                        Loved it
                    </md-button>
                </md-menu-item>
                <md-menu-item>
                    <md-button ng-click="null" layout="row">
                        <md-icon md-font-set="material-
                        icons">bookmark</md-icon>
                        Tag it
                    </md-button>
                </md-menu-item>
                <md-menu-item>
                    <md-button ng-click="null" layout="row">
                        <md-icon md-font-set="material-icons">exit_
                        to_app</md-icon>
                        Exit App
                    </md-button>
                </md-menu-item>
            </md-menu-content>
        </md-menu>
    </div>

</md-toolbar>
```

Complete Controller,

```
myModule.controller('positionSampleController', function($scope, $mdMedia,
$mdBottomSheet){
      $scope.showBottomSheet = function($mdOpenMenu, event){
             if($mdMedia('gt-sm')){
                $mdOpenMenu(event);
             }else{
                 $mdBottomSheet.show({
                    templateUrl: "/bottom-sheet-template.html"
                 });

             }
            };
});
```

Transform

Transform, another responsive design pattern, recommends realigning and rearranging elements on the page for the best view on a given screen size.

Consider an example we already discussed: a responsive grid list that adapts to screen size on a mobile phone/tablet, laptop, or desktop. (Review Chapter 8, Grid List section, for reference.) Consider the following code and images that depict a transform pattern. It renders a single-column grid list by default, a two-column grid list on a small and medium-size screen, and a three-column grid list on an even bigger screen. See Figure 12-5 and Figure 12-6.

```
<md-grid-list md-cols="1" md-cols-gt-xs="2" md-cols-gt-md="3" md-row-
height="16:9">
        <md-grid-tile md-rowspan-gt-sm="{{($index===0)?2:1}}"  ng-
class="item.background" ng-click="null" ng-repeat="item in superheroes">
        <img style="max-width:100px" ng-src="{{item.imageUrl}}" />

        <md-grid-tile-footer layout-padding>
            <span>{{item.category}}</span>
        </md-grid-tile-footer>
        <md-grid-tile-header layout-padding>
            <h2>{{item.name}}</h2>
        </md-grid-tile-header>
    </md-grid-tile>
</md-grid-list>
```

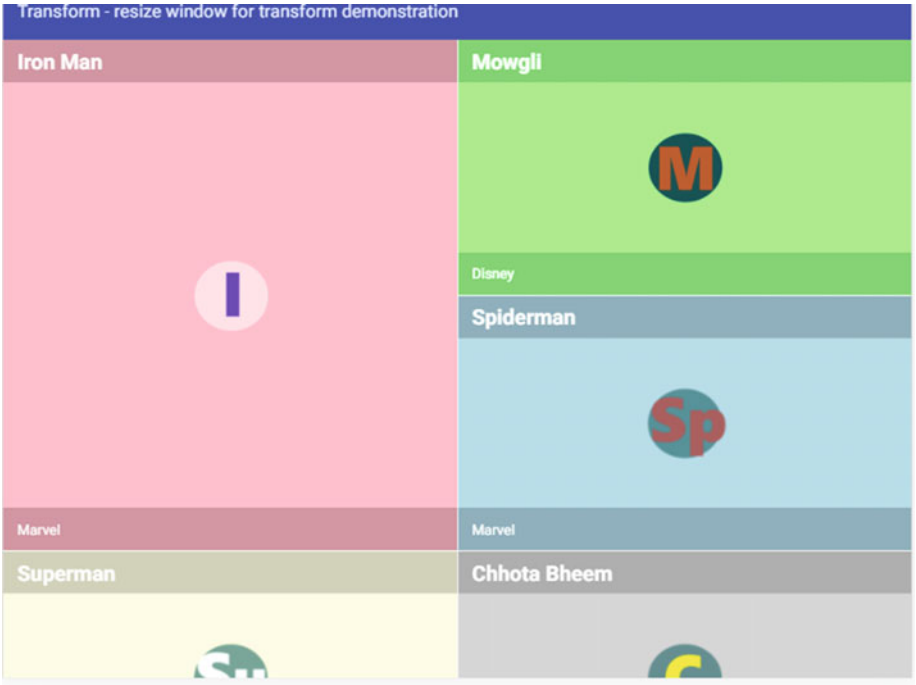

Figure 12-5. *Two columns on a medium screen size*

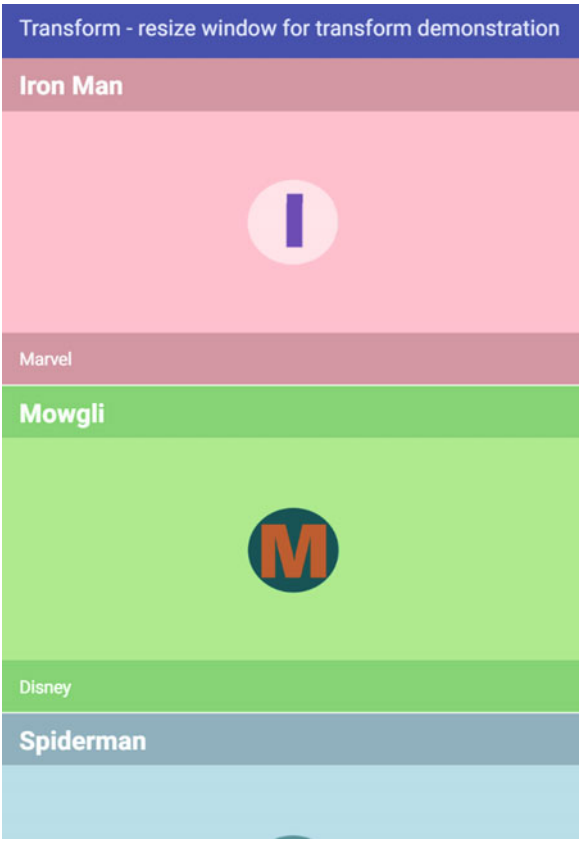

Figure 12-6. *Single column and default row span for first tile on a small screen*

Let us consider a completely new approach on top of it. Instead of a single-column grid list, we can hide the grid list on an extra-small screen and show a list view. The list view is compact and better on a smaller screen. Consider the following code. Highlighted code shows/hides the grid list. It is hidden by default. However, it will show if it is greater than an extra-small screen (i.e., small, medium, large, and extra-large).

```
<md-grid-list hide show-gt-xs md-cols="2" md-cols-gt-md="3" md-row-
height="16:9">
                <md-grid-tile md-rowspan-gt-sm="{{($index===0)?2:1}}" ng-
                class="item.background" ng-click="null" ng-repeat="item in
                superheroes">
                    <img style="max-width:100px" ng-src="{{item.imageUrl}}" />
```

```
            <md-grid-tile-footer layout-padding>
                <span>{{item.category}}</span>
            </md-grid-tile-footer>
            <md-grid-tile-header layout-padding>
                <h2>{{item.name}}</h2>
            </md-grid-tile-header>
        </md-grid-tile>
    </md-grid-list>
```

On an extra-small screen, the following list view will show (see Figure 12-7). Notice that it shows the same data and in the code has the same bindings (imageUrl, name, category, etc.). Nevertheless, it is hidden on a bigger screen. It will show when grid list does not (which is on an extra-small screen).

```
<md-list hide show-xs>
    <md-list-item class="md-2-line" ng-repeat="item in
    superheroes" ng-click="null">
        <img ng-src="{{item.imageUrl}}" class="md-avatar">
        <div class="md-list-item-text" layout="column">
            <strong>{{item.name}}</strong>
            <div>{{item.category}}</div>
        </div>
    </md-list-item>
</md-list>
```

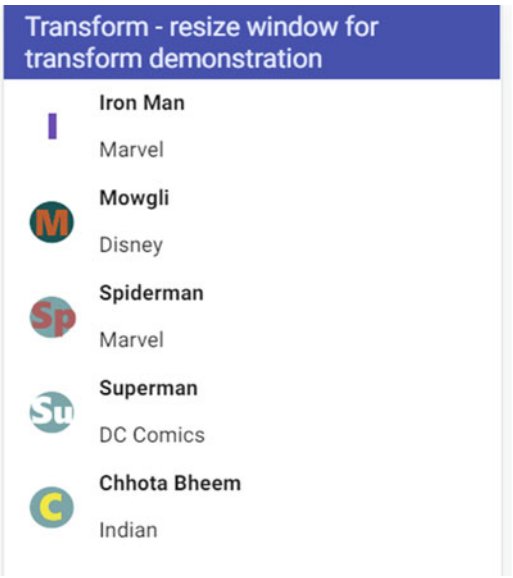

Figure 12-7. *List view on an emulated mobile phone view*

Reveal

Reveal is another responsive Material Design pattern. The UI reveals more options, content, and actions on a bigger screen. These might be hidden or collapsed on a smaller screen.

Consider the responsive approach to sidenav described in Chapter 4. On a bigger screen, sidenav could always be shown. Imagine a medium or large screen. It could be a tablet in landscape mode or a browser on desktop/laptop full screen. We have enough space to show all options. Hence, sidenav will always show. On a smaller screen, hide the sidenav to give way to workspace and the main content of the view.

Consider the following images of a browser emulated to iPad screen in portrait and landscape modes (Figure 12-8 and Figure 12-9, respectively). The former does not have enough space for sidenav, whereas the latter reveals sidenav.

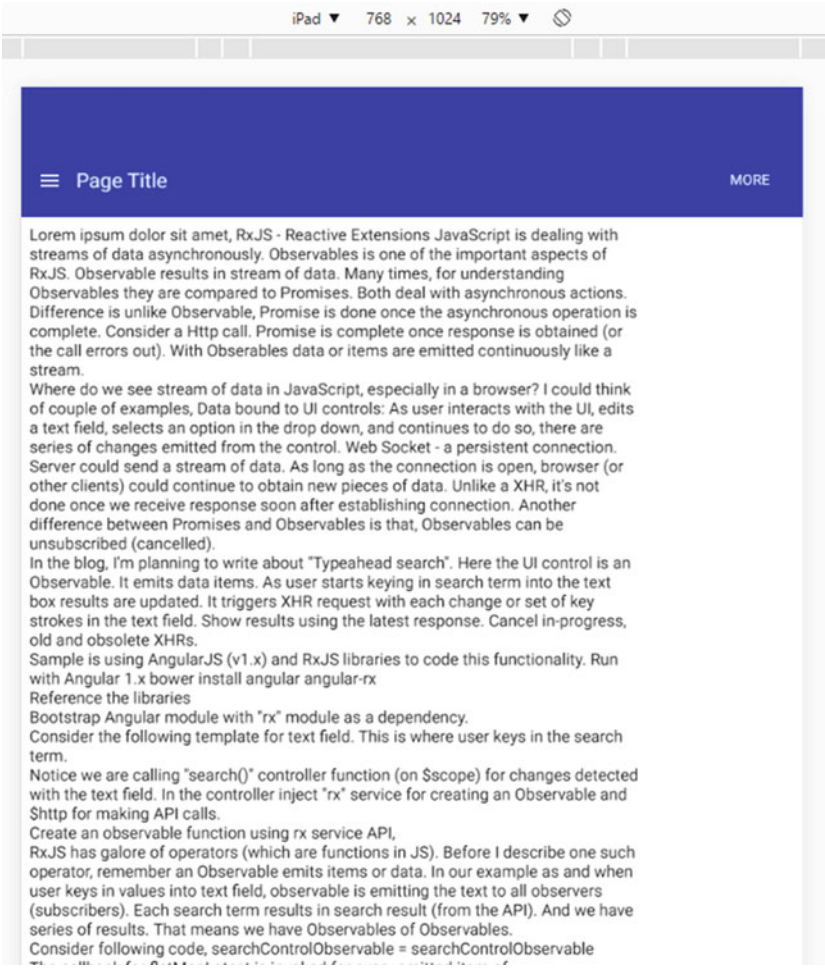

Figure 12-8. *Browser emulated to iPad in potrait dimentions. It does not show sidenav*

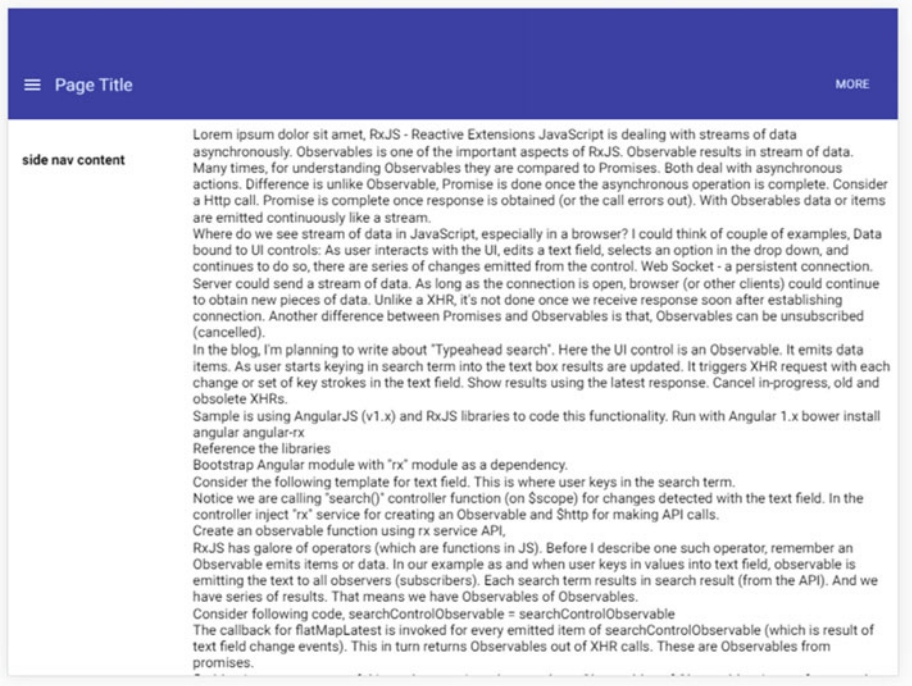

Figure 12-9. *Browser emulated to iPad in landscape dimentions. It has enough space for sidenav*

Consider the following code to achieve the same. Notice that the code has a responsive attribute to implement the reveal pattern. The sidenav is locked open on a screen greater than small (*gt-sm*), which is medium and above.

```
<md-sidenav md-component-id='content-sidenav' md-is-locked-
open="$mdMedia('gt-sm')" flex="20" class="md-sidenav-left" layout-padding>
              <h4>side nav content</h4>
 </md-sidenav>
```

Review Chapter 4 for complete code snippets on sidenav.

The sidenav, when it is hidden on a small screen, we could place a menu button on toolbar to open and use available options.

Reveal—Toolbar Actions Example

Let us explore another implementation of reveal pattern with toolbar actions. These are page-level buttons placed on the toolbar. We could use the pattern to show more options on a larger screen. On a smaller screen we could collapse to show only the most used or relevant actions.

Similar to sidenav, we may transform the remaining options into a menu. The additional options that do not fit on a smaller screen could collapse into a more options menu. This allows the screen to be clean and usable, with only a required set of actions or buttons.

Implementing a more options menu is analogous to a transform pattern, as the buttons on toolbar transform to a menu. However, adjusting available buttons on a screen is analogous to a reveal pattern.

Consider Figure 12-10. There are six toolbar actions available for a given page. As you read through the rest of the section, you will realize that these buttons gradually collapse under "more actions" buttons. A couple of them collapse initially on a small screen. More will go under the menu as the screen size reduces further. See Figure 12-10, Figure 12-11, and Figure 12-12.

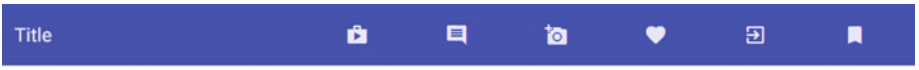

Figure 12-10. *Medium screen size; all buttons shown*

On a big enough screen, we could show all available buttons. On an extra-small screen, we show only two action buttons with a "more" menu. See Figure 12-11. The first two action buttons are shopping and comments (icons). Consider using the most used and relevant icons for a mobile screen. The more button acts like a menu and shows rest of the actions. See Figure 12-12.

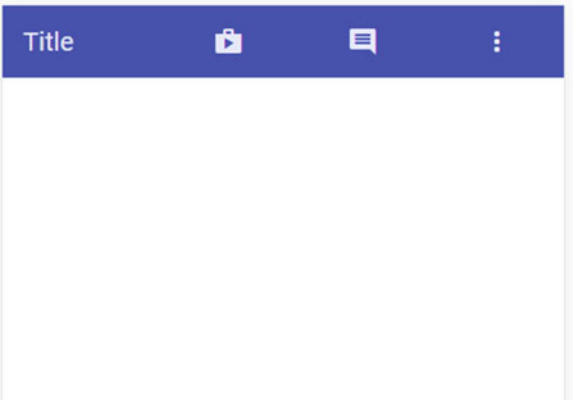

Figure 12-11. *Toolbar with actions on extra-small screen*

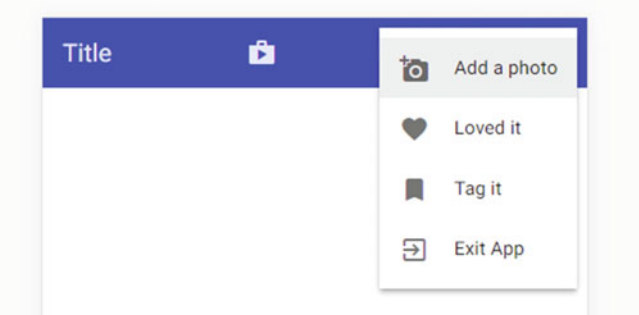

Figure 12-12. *Hidden actions revealed by clicking more button*

Hide/Show Buttons on Toolbar

At first, let us look at code to hide or show buttons on the toolbar depending on the screen size. In the next section, will detail showing those on the menu.

Consider the following code. On the toolbar, shopping and comments buttons are shown irrespective of screen size. No flexbox, responsive attributes (breakpoints) for these elements.

```
<div class="md-toolbar-tools" layout="row">
    <h2>Title</h2>
    <span flex></span>
    <md-button ng-click="null">
        <md-icon md-font-set="material-icons">shopping</md-icon>
    </md-button>
    <md-button ng-click="null">
        <md-icon md-font-set="material-icons">comment</md-icon>
    </md-button>
```

However, the following actions are shown on a screen greater than extra-small, that is, small and above. Review highlighted code for responsive breakpoints.

```
<md-button ng-click="null" hide show-gt-xs>
    <md-icon md-font-set="material-icons">add_a_photo</md-icon>
</md-button>

<md-button ng-click="null" hide show-gt-xs>
    <md-icon md-font-set="material-icons">favorite </md-icon>
</md-button>
```

As the screen size increases, we could show more actions. Two more toolbar buttons are shown on a medium and above screen size. On a smaller screen (small and below) they are hidden under the menu. Review boldface code for breakpoints. Also see Figure 12-13 and Figure 12-14.

```
<md-button ng-click="null" hide show-gt-sm>
    <md-icon md-font-set="material-icons">exit_to_app</md-icon>
</md-button>

<md-button ng-click="null" hide show-gt-sm>
    <md-icon md-font-set="material-icons">bookmark</md-icon>
</md-button>
```

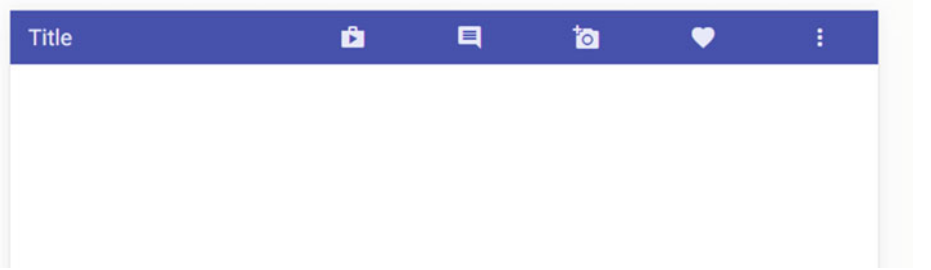

Figure 12-13. *On a small screen size (breakpoint sm), two action buttons are hidden under the "more action" button*

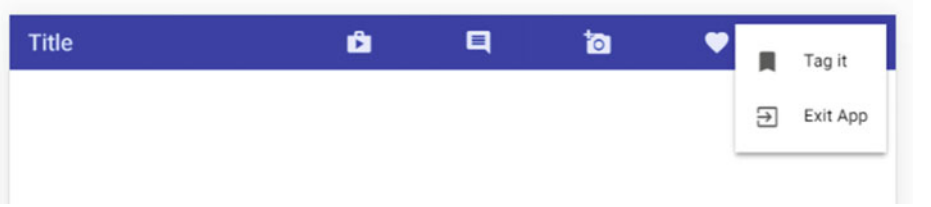

Figure 12-14. *Small screen size. The menu shows only the two hidden menu options*

On the other hand, the menu for more options is shown on small and extra-small screens only. Notice that the "more actions" menu needs to have inverse breakpoints compared to other buttons on the toolbar. That is because if a button is already shown on the toolbar, it need not be shown under the "more" menu. When it is hidden because of space constraints on the toolbar, it needs to be available on the more actions menu.

Review the following code: highlighted breakpoints for the menu. It is hidden on a screen size greater than small (medium and larger). As you might have noticed in the preceding, medium and larger screen sizes have no toolbar action buttons hidden. Hence, the more button is not necessary.

```
<md-menu hide-gt-sm>
    <!-- menu trigger, a button on the toolbar. -->
    <md-button ng-click="$mdOpenMenu($event)">
    <md-icon md-font-set="material-icons">more_vert </md-icon>
</md-button>
```

Hide/Show Buttons on the Menu

Now that we saw code to manage buttons on the toolbar, let us look at hiding and showing the same on menu. When a button is hidden on the toolbar due to space constraints, it needs to be available under the menu.

Use Angular Material menu control to implement the "more" button. The icon button is a trigger for the menu.

Consider the following code for buttons under the menu. As learnt while discussing menu control in an earlier chapter, these buttons are coded under *md-menu-content* element/directive.

```
<md-menu-content>
    <md-menu-item hide-gt-xs>
        <md-button ng-click="null" layout="row">
            <md-icon md-font-set="material-icons">add_a_photo</md-icon>
            Add a photo
       </md-button>
   </md-menu-item>
   <md-menu-item hide-gt-xs>
       <md-button ng-click="null" layout="row">
           <md-icon md-font-set="material-icons">favorite</md-icon>
           Loved it
      </md-button>
   </md-menu-item>
```

Each menu item is wrapped in *md-menu-item.* Review breakpoints on the menu, which hide buttons on a screen size greater than extra-small (hide-gt-xs). This means that screen sizes of small and larger show these buttons on the toolbar. Hence, the menu does not need to have "add photo" and "favorite" (Loved it) buttons.

The following is what remains of menu content. These are hidden on a greater than small screen size (medium and above). If the menu button is shown, these buttons are also visible.

```
 <md-menu-item hide-gt-sm>
     <md-button ng-click="null" layout="row">
    <md-icon md-font-set="material-icons">bookmark</md-icon>
          Tag it
    </md-button>
</md-menu-item>
<md-menu-item hide-gt-sm>
      <md-button ng-click="null" layout="row">
      <md-icon md-font-set="material-icons">exit_to_app</md-icon>
         Exit App
     </md-button>
 </md-menu-item>
```

Here is the complete code.

```
<md-toolbar class="md-secondary">
    <div class="md-toolbar-tools" layout="row">
        <h2>Title</h2>
        <span flex></span>
        <md-button ng-click="null">
            <md-icon md-font-set="material-icons">shopping</md-icon>
        </md-button>
        <md-button ng-click="null">
            <md-icon md-font-set="material-icons">comment</md-icon>
        </md-button>
        <md-button ng-click="null" hide show-gt-xs>
            <md-icon md-font-set="material-icons">add_a_photo
            </md-icon>
        </md-button>
        <md-button ng-click="null" hide show-gt-xs>
            <md-icon md-font-set="material-icons">favorite </md-icon>
        </md-button>
        <md-button ng-click="null" hide show-gt-sm>
            <md-icon md-font-set="material-icons">exit_to_app
            </md-icon>
        </md-button>
        <md-button ng-click="null" hide show-gt-sm>
            <md-icon md-font-set="material-icons">bookmark</md-icon>
        </md-button>
        <md-menu hide-gt-sm>
            <!-- Trigger element is a md-button with an icon -->
            <md-button ng-click="$mdOpenMenu($event)">
                <md-icon md-font-set="material-icons">more_vert
                </md-icon>
            </md-button>
            <md-menu-content>
                <md-menu-item hide-gt-xs>
                    <md-button ng-click="null" layout="row">
                        <md-icon md-font-set="material-icons">add_a_photo</
                        md-icon>
                        Add a photo
                    </md-button>
                </md-menu-item>
                <md-menu-item hide-gt-xs>
                    <md-button ng-click="null" layout="row">
                        <md-icon md-font-set="material-icons">favorite</md-
                        icon>
                        Loved it
                    </md-button>
                </md-menu-item>
```

```
                <md-menu-item hide-gt-sm>
                    <md-button ng-click="null" layout="row">
                        <md-icon md-font-set="material-icons">bookmark</md-
                        icon>
                        Tag it
                    </md-button>
                </md-menu-item>
                <md-menu-item hide-gt-sm>
                    <md-button ng-click="null" layout="row">
                        <md-icon md-font-set="material-icons">exit_to_app</
                        md-icon>
                        Exit App
                    </md-button>
                </md-menu-item>
            </md-menu-content>
        </md-menu>
    </div>

</md-toolbar>
```

Summary

Responsive design is one of the core features of Angular Material. In this chapter, we explored multiple patterns which facilitate better user experiences on all screen sizes. These patterns identify and provide solutions to commonly occurring screen layout problems.

The preceding implementations are my take on Material Design–responsive patterns. They are implemented using flexbox and Angular Material features. I acknowledge there could be multiple ways to implement the same pattern (within the Angular Material context). The preceding examples should provide some context and an initial idea.

In this chapter, we discussed the following:

1. **Reflow:** It rearranges controls and the content to fit various screen sizes. There could be variations in landscape and portrait layouts on the same device, or the app could be opened on a bigger and better screen. The reflow pattern suggests a way to rearrange content to fill available space.

2. **Position:** It suggests repositioning the control or component by screen size. A control like menu might not be suitable for a small screen. We might need a different control, of positioning at a different location on a smaller mobile screen. We may use bottom sheet instead of menu on a smaller screen and fall back to menu on a bigger screen.

3. **Transform:** It suggests approaches to rearrange and use different layouts by screen size. Transforming a grid to show different number of columns on various screen sizes is one effective approach. We could even take it to the next level and use a better and compact control like "list" on a smaller mobile screen.

4. **Reveal:** It recommends using small and important actions on smaller screens, like mobile phones. From there, reveal to show more options on a bigger screen and hence make the view or functionality more powerful on a bigger screen.

References

For Material Design–responsive patterns, see https://www.google.com/design/spec/layout/responsive-ui.html#responsive-ui-patterns

Index

A

V. K. Kotaru, *Material Design Implementation with AngularJS*,
DOI 10.1007/978-1-4842-2190-7

G

H

I

J, K

L

M

N, O

P, Q

R

S

T

U

V

W, X, Y, Z