Beginners Guide to AngularJS

Beginners Guide to AngularJS

Learner's Guide

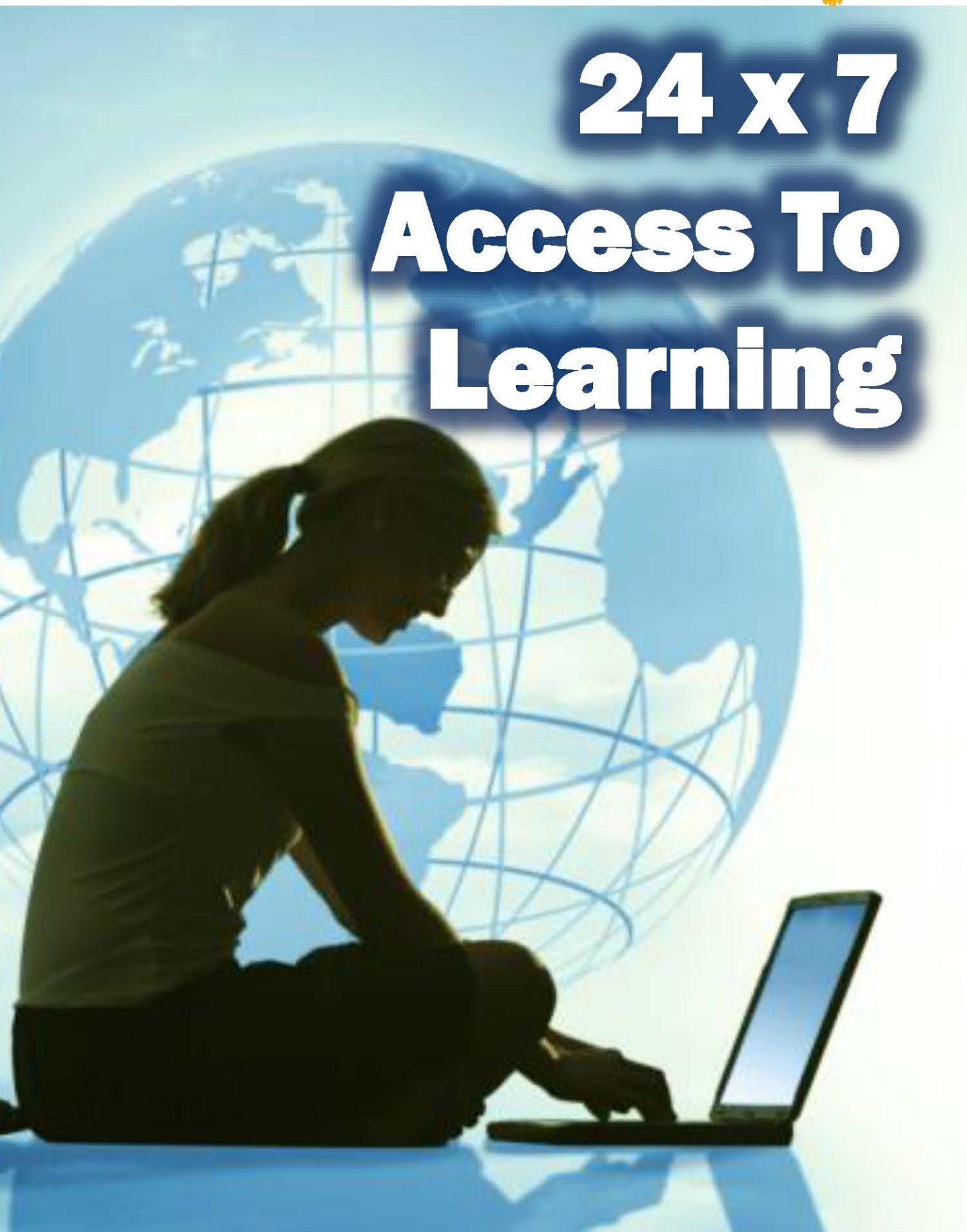
APTECH LIMITED
Contact E-mail: ov-support@onlinevarsity.com

Edition 1 - 2020

Aptech®
Unleash your potential

Aptech
COMPUTER EDUCATION
Unleash your potential

Onlinevarsity



Preface

This book, **Beginners Guide to AngularJS**, introduces you to the basic concepts of AngularJS framework and how to use it for developing Web applications. In a short time, AngularJS has become one of the most in-demand JavaScript frameworks. Basic concepts, components of AngularJS framework, Angular 9 features, developing Angular 9 applications, and Single Page Application development are covered in simple language in this book. Th book also covers AngularJS Material.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team

Onlinevarsity

Onlinevarsity App

for Android devices

Download from Google Play Store

Table of Contents

Onlinevarsity

INDUSTRY BEST PRACTICES

SYNC WITH THE INDUSTRY

Session 1

Starting Out with AngularJS and Angular

In this session, students will learn to:

- Describe what are AngularJS and Angular
- List features of AngularJS
- Analyze MVC architecture concepts in AngularJS
- Describe where AngularJS and Angular are used

This session introduces basic concepts of AngularJS and Angular frameworks.

1.1 What is AngularJS?

AngularJS is a JavaScript framework that is used to build browser based dynamic Web applications. In particular, it is very useful in building Single Page Applications (SPA). Single page applications are the latest trend where Web applications use a single page to load varying content dynamically based on user actions. User experience is similar to that of a desktop application. Another definition of SPA is that it is a Web application that fits on a single page and hence, the name.

It facilitates:

- Easier and faster coding
- Easier data binding

AngularJS is written in JavaScript with a reduced jQuery library called jQuery lite. It is completely open source and has the backing of Google and a dedicated community of followers. It builds on HTML, JavaScript, and CSS which Web developers may be familiar with already.

AngularJS extends HTML to give it Model-View-Controller (MVC) capabilities. MVC is a technology framework to organize Web application code by functions they perform. In formal terms, this organizing or separating is also called separation of concerns. The basic idea of MVC is that each part of your application code has a specific purpose. The Model determines what can be created and can hold raw materials (data). The View defines how and what the end user sees. The Controller acts as a connection between Model and View and is like a brain because it controls everything. Refer to Figure 1.1.

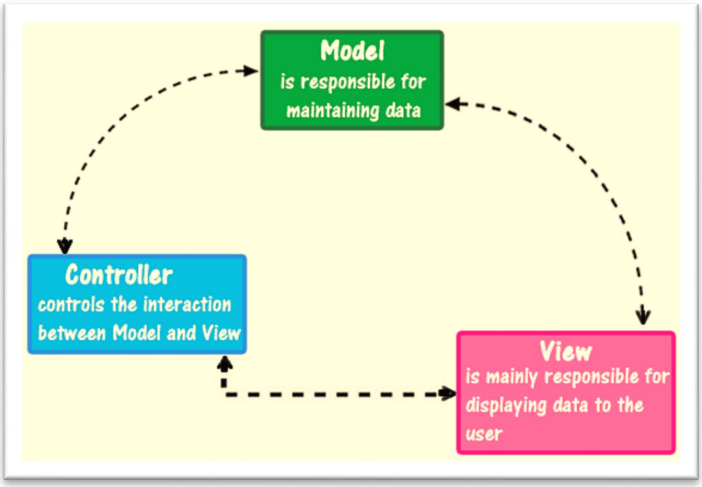


Figure 1.1: MVC Pattern

Another benefit of AngularJS is that it enables the developer to encapsulate a portion of a page as one application, rather than forcing entire page to be an AngularJS application. This is particularly advantageous if our project already includes another framework. It is also useful if a portion of a page is

to be made dynamic, while rest of the page operates as a static page. This is also the idea on which Single Page Applications (SPA) are based.

In a nutshell, the objective of AngularJS is to provide a framework that makes it easy to implement well-designed and well-structured Web pages and applications using a MVC framework. AngularJS makes Web application and the code very simple to write, test, and maintain.

1.1.1 Brief Version History

Misko Hevery and Adam Abrons are two developers who developed AngularJS in 2009 while they were in Google. It was first publicly released in October 2010. Last stable release of AngularJS is 1.7.9.

A complete of AngularJS called as Angular 2 that was released in September 2016. Subsequently, versions 4, 5, 6, and so on were released. Table 1.1 summarizes these releases along with their key features in brief.

Remember:
Angular is the term used for any Angular version above 2. Angular 2 and higher are called Angular, for short.**AngularJS** refers to the versions prior to version 2.

Version	Year of Release	Features
Angular 2	September 2016	Written entirely in TypeScript (an open-source programming language which is a superset of JavaScript and builds on it by adding syntax for type declarations) Is mobile oriented Provides more choice for languages such as ES5, ES6, TypeScript, or Dart to write Angular 2 code
Angular 3	Skipped	-
Angular 4	13 December 2016	Is backward compatible with 2.x.x for most applications No major change in Angular 4 from Angular 2 Is not a complete rewrite of Angular 2 Faster Compilation Better bug fixes alert Introduced HttpClient for making HTTP Requests. HttpClient is a smaller, easier to use, and more powerful library
Angular 5	November 1, 2017	Support for progressive Web apps, a build optimizer, compiler improvements, and improvements related to Material Design @angular/http is deprecated in Angular 5 and has been replaced with @angular/common/http library. Angular Command-Line Interface (CLI) v1.5 will generate v5 projects by default
Angular 6	May 4, 2018	Was released with Angular CLI 6 and Material 6. Angular Material is a UI component library for Angular developers.
Angular 7	October 18, 2018	Was released along with Angular CLI 7 and Angular Material 7
Angular 8	May 28, 2019	Was released with Angular CLI 8 and Angular Material 8.
Angular 9	February 6, 2020	Moves all applications to use Ivy compiler and runtime by default. Angular has been updated to work with TypeScript 3.6 and 3.7.

Table 1.1: Angular Version History

There are several differences between AngularJS and Angular, some of which are outlined in Table 1.2.

AngularJS	Angular
It was released in the year 2010. Last stable release of AngularJS is 1.7.9.	Released in September 2016 as Angular and then, other versions were released in later years. Last stable release of Angular is 9.
AngularJS has MVC based architecture.	Angular 2 onwards, the architecture is based on

AngularJS	Angular
	services and other elements.
AngularJS is not mobile-friendly.	Angular 2 onwards is mobile-friendly framework.
One does not need to know TypeScript to write AngularJS code. It is sufficient to learn HTML5 and JavaScript.	It would be beneficial to be familiar with TypeScript, ES6, and ES5 to write the code of Angular 2.
It focuses on controllers.	Angular 2 onwards is based on components.
Constants, values, providers, services, and factory are used for services.	You can only use a class to define Angular 2 (and higher) services.
You can run AngularJS on the client side only.	You can run Angular 2 (and higher) on server and client side.

Table 1.2: AngularJS Versus Angular

1.2 Why Choose AngularJS?

Everything that AngularJS provides, we can also implement ourselves by using JavaScript and jQuery or by using another MVC JavaScript framework. Even though it is possible to do it, this will need a lot of coding to build the structure requiring ample time and effort. Against this hard work, AngularJS offers us a lot of functionality out of the box. Some of the reasons to choose AngularJS are as follows:

- AngularJS framework makes us implement MVC and makes it easy for us to correctly implement MVC. The declarative style of AngularJS HTML templates makes construction of the HTML obvious.
- The model components of AngularJS are nothing but basic JavaScript objects. So, it becomes easy to access and handle our data.
- AngularJS allows us to have our custom defined tags in HTML. This makes it possible for us to name the tags in a self-describing manner. This declarative approach extends the functionality of the tags and results in a direct link between the HTML tags and the JavaScript functionality described by them.
- Another advantage of AngularJS is that it provides simple and flexible filters that help us to easily format data as it passes from the model to the view.
- AngularJS applications uses less code than traditional JavaScript applications. This is because we focus only on the logic and not all the finer details, such as data binding. They are made available to us by the AngularJS framework.
- AngularJS applications need less Document Object Model (DOM) manipulation than conventional methods. It is easier to design applications based on presenting data than on DOM manipulation.
- AngularJS supplies several built-in services and helps us to implement our own services in a structured and reusable way.
- It is easy to test applications and develop them using a test-driven approach.

1.3 MVC Concepts

Let us understand MVC.

1.3.1 MVC Architecture Pattern

We saw that AngularJS supports MVC architecture in software development. MVC pattern is a way of design and development. It is a general paradigm used in many frameworks used for software development. AngularJS also follows this pattern of development and makes the developer adhere to this

structure. In the MVC application development pattern, different parts of the application are broken into components to do distinct tasks. The Model holds data and logic, the View holds visual layout and presentation, while the Controller coordinates the Model and View.

In summary, remember that MVC is a design pattern that we can follow while we develop our applications. It is used in many other frameworks also other than AngularJS.

1.3.2 MVC Pattern in AngularJS

One can think of MVC as shown in Figure 1.2.

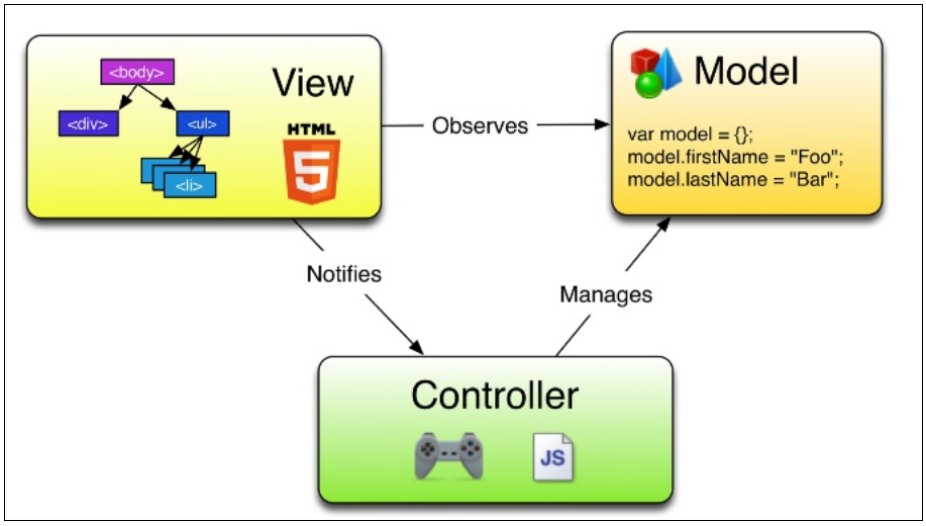


Figure 1.2: MVC Example in AngularJS

Model holds application data for the application. It has the data that we want to display to our users or accept from the user through our application. Therefore, model in an AngularJS project is mostly pure data and models are represented using JSON objects. The model is only aware of itself.

View is what is visible to users of our application. They will not get to see the model or controller of the application they are using. Generally, view knows the model, as it has to show data contained in the model and calling actions (methods) on the model as and when required.

Controller is the coding logic that would act as the link between the model and view. It is the responsibility of the controller to create and populate the model and give it to the view. Controller has the actual business logic and the code that fetches the data. It makes decisions about which part of the model to show to the user and how to handle validation. In other words, it has the core logic of the application.

For easy understanding, you can think of model and view as two persons standing on two sides of a bridge and the controller as the bridge that connects these two persons. Refer to Figure 1.3.

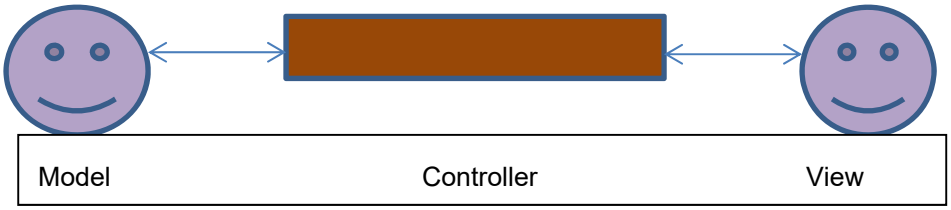


Figure 1.3: Relationships among Components

1.3.3 MVC Architecture Benefits in AngularJS

We can start with a standard HTML-based set of views and then later add a new set of views supporting a different format. According to seasoned Web developers, MVC-style architecture offers us a huge reduction in the effort required for the overall application development cycle.

We gain this benefit because, through MVC, we apply the principle of 'Separation of Concerns'. The view is in no way exclusively tied to the model, so it is far easier to handle it as a distinct component that we can exchange out for another.

MVC helps in the testing phase also. It leads to applications that are much easier to test unit wise and the total application as a whole. The test process is continued as the application grows in complexity. MVC is a tried and tested way to build robust applications.

1.4 Angular Application Lifecycle

Now, let us go through the execution cycle of a very simple AngularJS application to get a better understanding.
Once we understand how an AngularJS application is built and how it is executed, many of the implementation details of the framework will make sense. The reason for this is that the startup process provides a picture of how the components of the framework are linked together.

1.4.1 Code for a Simple AngularJS Application

Figure 1.4 shows the full code of the AngularJS application. This application can be used to compute the total cost if we provide the quantity of items we purchase and cost of each item.

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Demo </title>
    <script src="angular.min.js"></script>
</head>
<body>
    <div ng-app="costApp" ng-controller="CostController">
  <b>Bill:</b>
  <div>   Quantity: <input type="number" min="0" ng-model="qty" required ></div>
  <div> Cost: <input type="number" min="0" ng-model="cost" required > </div>
  <div>
    <b>Total:</b>
    <span >
      {{total | currency:' $ '}}
    </span>
    <button class="btn" ng-click="pay()">Pay</button>
  </div>
  <script>
    var app = angular.module('costApp', []);
    app.controller('CostController', function ($scope) {
        $scope.qty = 0;
        $scope.cost = 0;
        $scope.total = 0;
        $scope.pay = function(){
            this.total = this.qty * this.cost
        };
    });
  </script>
</body>
</html>
```

Figure 1.4: Code for the AngularJS Application

In step-1, we make the AngularJS framework available to the application in line-6 of this code. In this case, we have specified the name of the file, **angular.min.js**. We can download this file from AngularJS Website (https://code.angularjs.org/) to our computer and make it available OR we can link to any Content Delivery Network (CDN) such as Google. Here, we have downloaded the file to our computer and mentioned it on line-6 of the code.

In step-2, we inform the application, in line-9, that the `div` element is the area where the application begins operating under the control of AngularJS and the name of the controller is 'CostController'.

In step-3, in line-20, through the `script` tag, we define the model data and the controller function.

1.4.2 Working of a Simple AngularJS Application

Figure 1.5 is the initial view of the AngularJS application when we run this application from a server. After we provide some values in the text boxes, and click the Pay button, a calculated amount will be displayed.

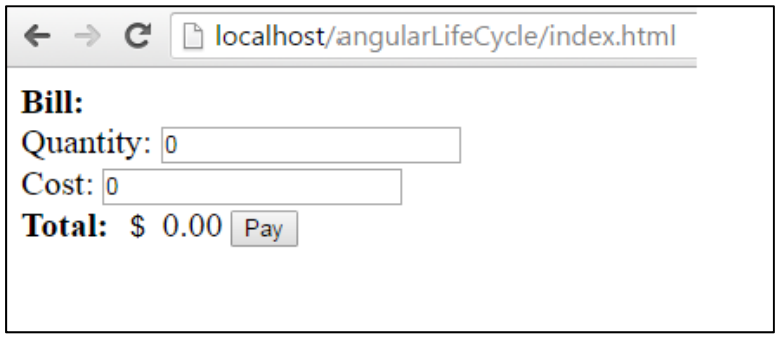


Figure 1.5: Initial View of the AngularJS Application

. The code looks like any other HTML code except for the addition of the AngularJS framework and some new attributes of AngularJS, such as `ng-app`, `ng-controller`, and `ng-click`. These are the directives that come with AngularJS framework.

The page is loaded in the browser. When the browser comes across the included *angular.min.js* file, it is loaded, and the browser becomes aware of the AngularJS application.

When AngularJS starts our application, it goes through and processes this code using the compiler. The outcome of this process is the rendered DOM, which is sometimes referred to as the view.

Directives in AngularJS give special behavior to elements of HTML. Here, in this example we use the `ng-app` directive. This directive initializes our application. The `ng-model` directive is used for storing or updating the value of the input field into/from a variable.

Another new markup shown here is the double curly braces `{{expression | filter}}`. When the compiler comes across this markup, it will substitute the code with the calculated value of the markup. An expression in a template is a code snippet, similar to a JavaScript that allows AngularJS to read and write variables. In our example, the markup tells AngularJS to 'take the data we received from the input elements and multiply them'.

This example also contains a filter. A role of a filter is to format the value of an expression for display to the user. In this example, the filter formats a number into an output that looks such as money.

Now, let us add some new values to use the application. Figure 1.6 shows the view after providing new values followed by clicking of the 'Pay' button.

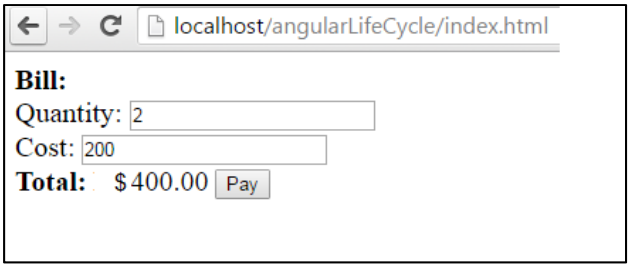


Figure 1.6: View of the AngularJS Application after new Inputs and Calculation

The interesting thing in this example is that AngularJS provides live bindings: Whenever the input values change, the values of the expressions are automatically recalculated and the model is updated with their values. Thus, we are achieving two-way data binding, where binding occurs both sides, between the input and the output. Figure 1.7 depicts this in a way that is easier to understand. Changes made to the user interface are reflected in the data variables and changes made to the data variables are reflected in the user interface.

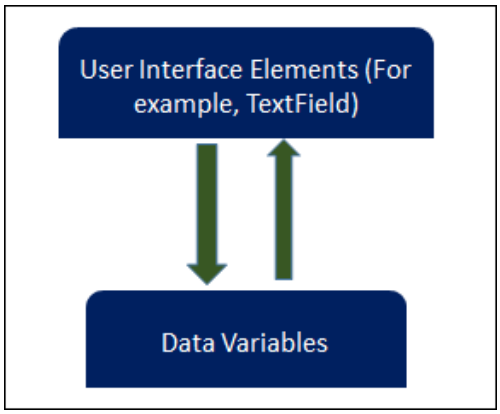
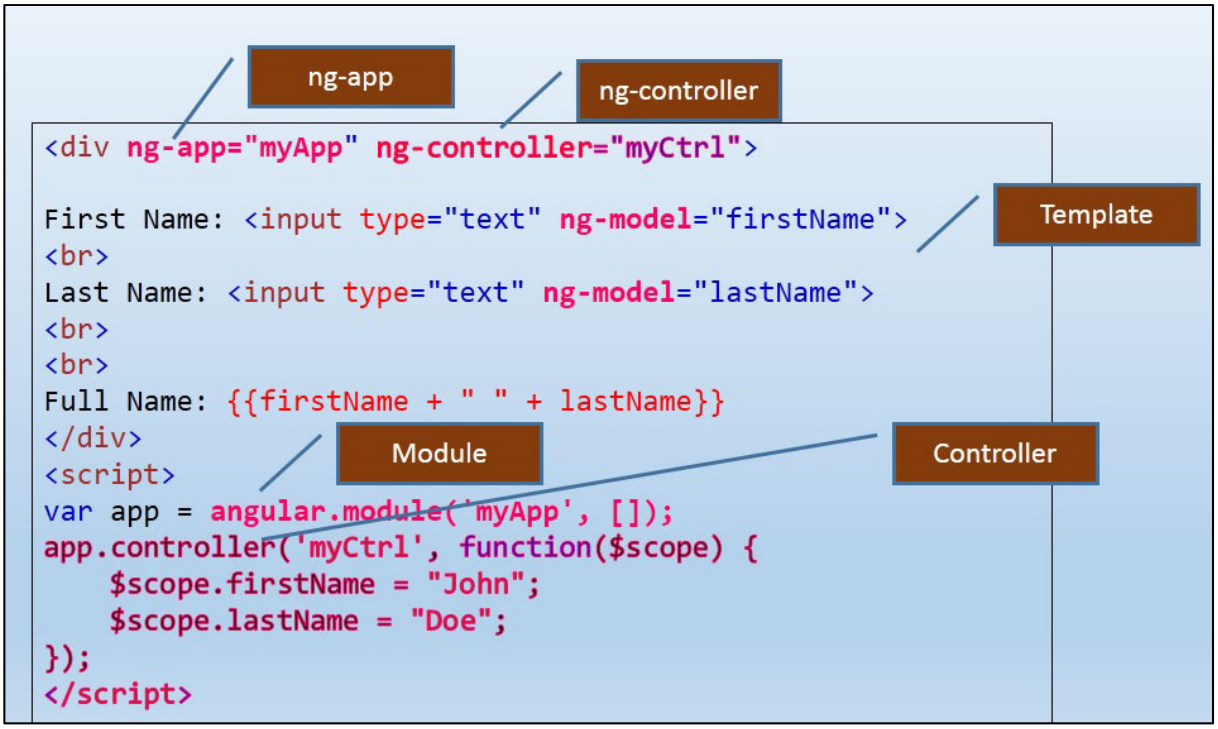


Figure 1.7: Two-Way Data Binding

To summarize, the key sections of an AngularJS application are as follows:



1.5 Where is AngularJS Used?

AngularJS provides a great platform for building dynamic Websites and Web applications. It is not surprising to see AngularJS is widely used. Figure 1.8 shows different categories of business using AngularJS.

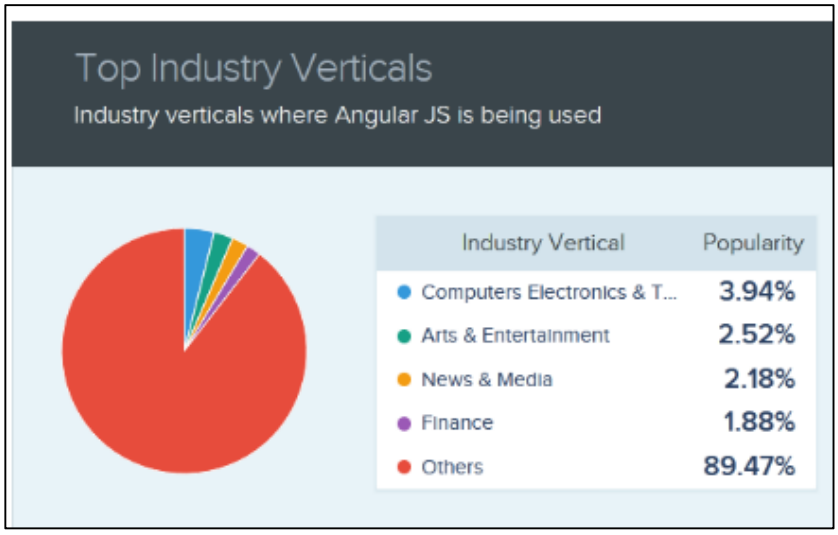
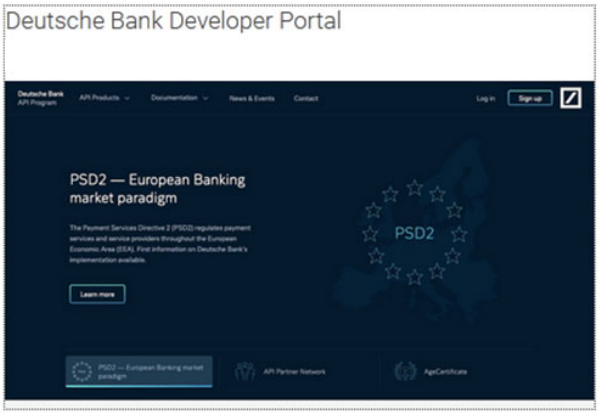


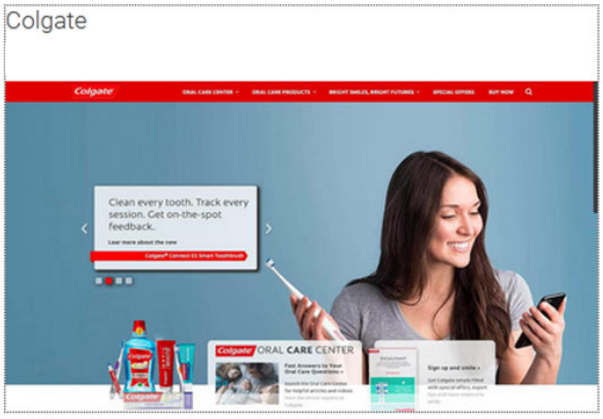
Figure 1.8: AngularJS Popular Website Verticals

Image courtesy: https://www.similartech.com/technologies/angular-js

As of today, over 219,828 companies in the world are using AngularJS.

Let us see some popular Websites and apps built with AngularJS:





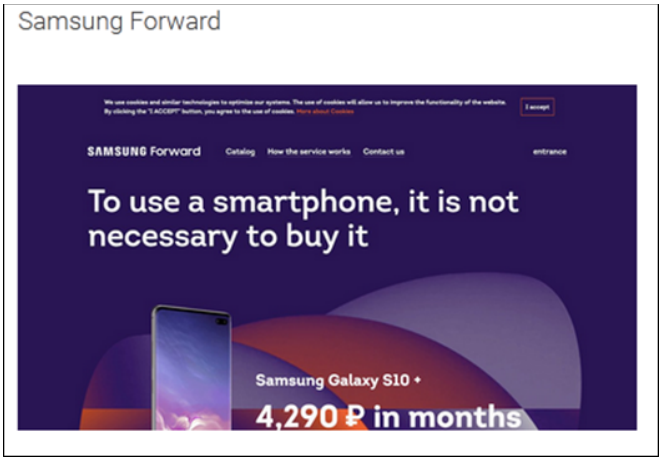


Image courtesy: https://www.madewithangular.com/

1.6 Writing AngularJS and Angular Applications

Following are some popular editor tools that can be used to create AngularJS and Angular applications:

- Sublime Text is quite popular with developers, though there can be a bit of a learning curve to use its many features.
- Notepad – built-in on Windows PCs, this tool is easy to use, but there is no coding support such as line numbering, syntax suggestions, and so on.
- Notepad++ - A free and popular text and source code editor.
- Visual Studio Code - on Windows PCs, many developers are already familiar with it.

Quick Test 1.1

1. AngularJS is a jQuery library.
   a. True
   b. False

2. AngularJS makes developing Single Page Application (SPA) easy.
   a. True
   b. False

1.7 Summary

- AngularJS is a JavaScript framework that we use to build Web applications and dynamic Websites.
- Angular 2 is a complete rewrite of AngularJS.
- Angular is the term used to refer to any version beyond 2.
- Latest version of AngularJS is 1.7.9 and that of Angular is 9.
- Model-View-Controller (MVC) is a software design pattern that we follow while we develop our applications.
- AngularJS helps us to implement well-designed and well-structured Web applications using a MVC framework.
- Model has the application data. View is what is visible to the users of our application. Controller acts as the link between the model and view.
- In MVC pattern, the view is defined in HTML, while the model and controllers are implemented in JavaScript.
- The important components of an AngularJS application are directives, expressions, and filters.
- AngularJS and Angular have attained widespread acceptance and many popular Websites and Web applications are based on them.

1.8 Exercise

1. AngularJS is written in which language?

   a) C++
   b) Java
   c) JavaScript
   d) PHP

2. Normally in an AngularJS application, which component holds the data?

   a) Model
   b) View
   c) Controller
   d) Templates

3. Which directive initializes an AngularJS application?

   a) `ng-init`
   b) `ng-app`
   c) `ng-model`
   d) `ng-controller`

4. A role of a filter is to _____ the value of an expression for display to the user.

   a) Format
   b) Delete
   c) Evaluate
   d) Change

5. _____ holds the visual layout and presentation.

   a) Model
   b) Controller
   c) View
   d) Directive

1.9 Do It Yourself

1. Download and install Coffee Cup HTML Editor. Explore its features.
2. Download and install Notepad++ Editor. Explore its features.
3. Paste the following code into a new file, save it as FirstDemo.html, and observe the output. Try it with both editors and check the editor differences.

```
<!DOCTYPE html>
<html lang="en" ng-app="myApp">
<head>
      <meta charset-"UTF-8">
      <title>Demo for a Simple AngularJS Application</title>
      <script src =
      "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.mi
n.js">
      </script>
</head>
<body ng-controller="myController">
      <h1>First AngularJS Application</h1>
</body>
</html>
```

Answers to Exercise

1. JavaScript
2. Model
3. Ng-app
4. Format
5. View

Answers to Quick Test

Quick Test 1.1

1. False
2. True

Session 2

Controllers, Expressions, Sharing Data, and Two Way Data Binding

In this session, students will learn to:

- Describe the process to create and run a simple AngularJS application
- List core components of AngularJS
- Describe how data is shared between model and view
- Identify data binding approaches

We will build a simple AngularJS application in this session. By building and running this, you will get to see the building blocks of an AngularJS application such as Modules, Controllers, and Views. It will also throw light on how data is shared between models and views.

2.1 A Simple AngularJS Application

Let us write and run a simple Web application using AngularJS. For simplicity's sake, let us call this application as Hello Application.

Step-1:

First, let us begin by creating a normal HTML page as shown in Code Snippet 1 with html, head, and body tags. Open any Web editor such as Notepad++ or Coffee Cup HTML editor and write this code in it. Do omit the line numbers while typing the code, they are only given for reference. Save the file as *Example 2.html*. Note that this is not the final application as yet, but only a beginning.

Code Snippet 1: Basic HTML Template.

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4.    <meta charset-"UTF-8">
5.    <title>Demo for a Simple AngularJS Application</title>
6. </head>
7. <body>
8. </body>
9. </html>
```

Step-2:

Next, we need to add the AngularJS capabilities to this HTML page. As this is done via a JavaScript file, *angular.min.js*, we can add this similar to any other external JavaScript file. We can download this file from AngularJS Website (https://code.angularjs.org/) to our computer and make it available OR we can link to any Content Delivery Network (CDN) such as Google. In our current example, let us link to Google CDN for our AngularJS file link. You can see the added link to *angular.min.js* in line number 6 of Code Snippet 2.

Code Snippet 2: Basic HTML Template with AngularJS link.

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4.    <meta charset="UTF-8">
5.    <title>Demo for a Simple AngularJS Application</title>
```

```
6.  <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
7. </head>
8. <body>
9. </body>
10.</html>
```

Thus, the Google CDN link for Angular library https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js has been specified in the script tag. By pointing to this Web-based file, you are spared the effort to download the file or even maintain a local copy. This becomes a quick and easy way to wok with AngularJS.

Step-3:

Next, we need to inform AngularJS, which section of the HTML will be controlled by it through the `ng-app` directive. When `ng-app` directive is added in an HTML page, it tells Angular that this HTML page is an AngularJS application. It also sets the root element for the application and initializes the AngularJS framework automatically.

It is possible to add `ng-app` directive to `<html>` tag or any element present in an HTML page. Here, we are adding `ng-app` to the root HTML element, which is the `<html>` tag. Our application is named as 'myApp'. Refer to line-2 of Code Snippet 3.

Code Snippet 3: HTML with `ng-app` Directive.

```
1. <!DOCTYPE html>
2. <html lang="en" ng-app="myApp">
3. <head>
4.   <meta charset="UTF-8">
5.   <title>Demo for a Simple AngularJS Application</title>
6.   <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
7. </head>
8. <body>
9. </body>
10.</html>
```

Step-4:

Next, we need to define our application modules and controllers. This can be done through an external JavaScript file or inline in the HTML file. In larger applications, it is common to store controllers in external files. In the current case, we are doing inline in the HTML document for the sake of simplicity. Refer to line-8 to line-10 of Code Snippet 4.

Code Snippet 4:

```
1. <!DOCTYPE html>
2. <html lang="en" ng-app="myApp">
3. <head>
4.   <meta charset="UTF-8">
5.   <title>Demo for a Simple AngularJS Application</title>
6.   <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
</head>
7. <body>
```

```
8.    <script>
9.          var app = angular.module("myApp",[]);
10.   </script>
11. </body>
12. </html>
```

Note that inside the script tag, in line-9, we define our application and give it the name 'myApp'. The square bracket following it will hold any other dependency of our application. In this case, we do not have any dependency and we leave it as an empty array.

Step-5:

Then, we add the controller of our application with name 'myController'. In AngularJS, a **scope** object is used for holding model values, which can be presented by the view. Scope is the glue between application controller and view.

We have seen that every AngularJS application has to have an `ng-app` directive compulsorily. When we attach `ng-app` directive to an element, AngularJS application creates a scope object called `$rootScope`.

Next, when we attach `ng-controller` to any element, it creates a new child scope, which inherits from the `$rootScope`. Thus, `$scope` comes in-built with AngularJS and offers a way to define the application variables and functions as its property. In our example, we define only a single variable `$scope.name` and give it a value of empty string.

Code Snippet 5: HTML after Addition of 'myController'

```
1. <!DOCTYPE html>
2. <html lang="en" ng-app="myApp">
3. <head>
4.    <meta charset="UTF-8">
5.    <title>Demo for a Simple AngularJS Application<title>
6.    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
7. </head>
8. <body ng-controller="myController">
9.    <script>
10.   var app = angular.module("myApp",[]);
11.   app.controller("myController",function($scope){
12.         $scope.name = "";
13.   });
14.   </script>
15. </body>
16. </html>
```

Step-6:

Then, we add elements that we want in our HTML page as shown in Code Snippet 6.

Note that we have added `ng-controller` to the `body` element with a value of 'myController' in line-8. With this, the view and controller get to know each other. Here, view is the `body` element of the HTML page.

The key elements of the application are the input elements with the attribute `ng-model` and the value of '`name`'. This is the same variable we have defined in the controller as `$scope.name`. This enables the value submitted by the user in the input element to be automatically assigned to model variable '`name`'.

Thus, we achieve two-way data binding. We shall learn more about data binding later. Whenever the model changes the value of an element in the controller, the modified value can be visible in the application view on the browser. For example, when the application is initially launched, the model and view both have empty string as the value for `name`. As and when the user changes the value by giving a new value in input box, it is simultaneously updated in the model and shown in HTML view also. All this is achieved without much coding on our part.

Code Snippet 6: Hello Application with the View Added to It

```
1. <!DOCTYPE html>
2. <html lang="en" ng-app="myApp">
3. <head>
4.    <meta charset-"UTF-8">
5.    <title>Demo for a Simple AngularJS Application</title>
6.    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
7. </head>
8. <body ng-controller="myController">
9.    <div>
10.       <h3>Please enter your name here.</h3>
11.       Name: <input type="text" ng-model="name" placeholder "enter your
name">
12.       <p>Hello! <strong>{{name}}</strong></p>
13.       <p>Welcome to AngularJS.</p>
14.   </div>
15.   <script>
16.       var app = angular.module('myApp',[]);
17.       app.controller('myController',function($scope){
18.            $scope.name = " ";
19.       });
20.   </script>
21. </body>
22. </html>
```

Step-7:

With this, our application is complete. Now, run the application in a modern browser such as a recent version of Google Chrome or Mozilla Firefox. The initial page looks similar to Figure 2.1.

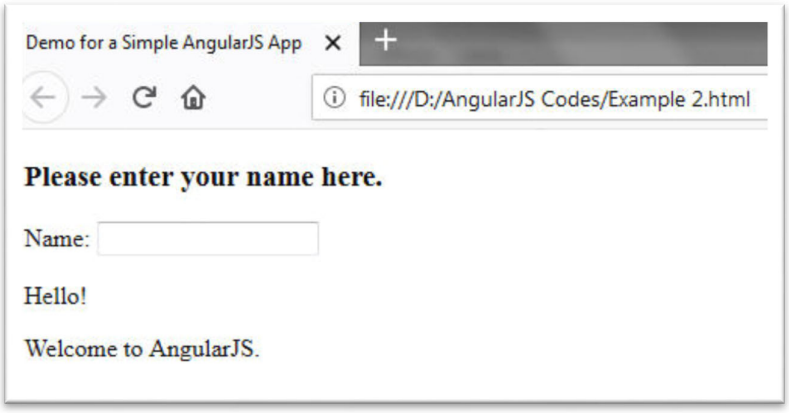


Figure 2.1: Hello Application When Loaded in a Browser

Now, as the page says, start entering your name in the input text box.

Let us input ‘George’ as the name.

As we enter the name in the input box, it is reflected beside 'Hello!', the place where we kept {{ name }} expression in our code. Refer to Figure 2.2.

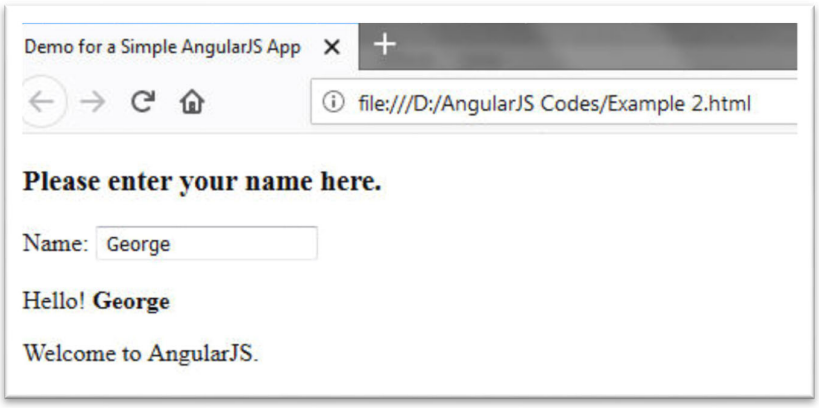


Figure 2.2: Hello Application with User Input

Congratulations! You have successfully written and run your first AngularJS application. Now, this application has been created and executed on the local computer. Since AngularJS is a framework for Web applications, it is recommended that you host applications on a Web server and then run them. Also, it is preferred that it is served from a server, since, some of the JavaScript in AngularJS applications will work only from a server.

In the current scenario, a local Web server would be ideal. Cross platform Apache, MySQL, Perl and PHP (XAMPP) is a free and open source solution that provides a local Web server option. X in the product name stands for Cross Platform. Following are the steps to install and start the local Web server:

1. Download and install XAMPP from this link: *https://www.apachefriends.org/download.html*.
2. Ensure that Apache Web Server is selected during installation.
3. Start XAMPP Control Panel.
4. Launch or start Apache. This starts your local Web server.
5. Go to the htdocs folder present under XAMPP and save your code in this folder path. Refer to Figure 2.3.

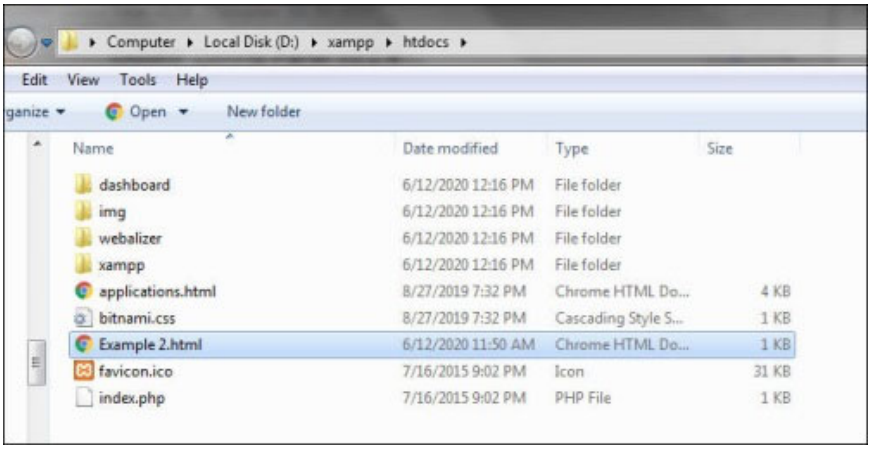


Figure 2.3: Saving Under htdocs Folder

6. Launch your AngularJS application with *http://localhost/<xyz>* where xyz is the application name. For example, *http://localhost/Example 2.html* will be the URL for our Hello Application.

Note: In this example, the URL given in the Address bar includes %20 before 2. This is because our application file name contains a space. When the URL is rendered in the browser, the %20 may be interpreted as a space, hence to be safe, the %20 is mentioned.

Refer to Figure 2.4 to view the output of the application.

Figure 2.4: Hello Application Hosted on LocalHost

2.2 Modules

In AngularJS, modules are containers used to hold other parts of an application. It is the holder for controllers, services, filters, directives, and so on. In AngularJS, we combine the functionalities in module. A module has the option to define its own controllers, services, filter, directives, and so on, which will be reachable throughout the module. A module is used by AngularJS to start an application.

Refer to line-2 of Code Snippet 3.

```
<html lang="en" ng-app="myApp">
```

By providing the module name as value to `ng-app` directive, we tell AngularJS to load this module as the main entry point for the application.

In AngularJS, we define or call a module using `angular.module` function.

```
var app = angular.module("myApp",[]);
```

Refer to line-9 of Code Snippet 4, where we define our application and give it the name 'myApp'.

All modules used in an application must be registered like this before they can be used. A module can depend on other modules. The square bracket in the code will hold any other dependency of our application. In this case, there is no dependency and it is left as an empty array.

2.3 Controllers

AngularJS applications depend on controllers to control the flow of data in the application.

Refer to line-11 of Code Snippet 5, where we add 'myController' to the application using the `app.controller()` method.

```
app.controller("myController",function($scope){
$scope.name = "";
});
```

Controllers are JavaScript objects that have properties and functions. We give to each controller `$scope` as a parameter while we define them. `$scope` in turn refers to the module that controller is to control. This is how AngularJS is designed.

We use a controller in an application by using `ng-controller` directive.

Refer to line-8 of Code Snippet 6.

```
<body ng-controller="myController">
```

AngularJS Controllers are an important part of an AngularJS application. These are JavaScript functions/objects that perform greater part of UI related work. They may act as the driver for the changes that can happen in model and view. They are the link between the model (the data in our application), and the view (whatever users see on screen and interacts with).

Their main responsibilities include:

- Making available Data to UI.
- Managing presentation, such as which elements to show/hide, what style to apply to them, and so on
- Handling user inputs. For example, how to respond when a user clicks something or how to validate a given text input
- Processing data coming from user input and then sending to server

2.4 Views

In general, we can say that views are what the user gets to see in an application.

The HTML code that is rendered by the browser and is seen by the user is also sometimes referred to as view. That is the 'User Interface' that we design and show to the user.

In our example, the view is as shown in Figure 2.5. This is same as Figure 2.4 seen earlier, it is shown again for your easy reference.

Figure 2.5: View of the Hello Application When Loaded in a Browser Initially

We can have different views in an application and show the relevant view based on User interaction.

2.5 Expressions

AngularJS expressions are similar to JavaScript code snippets. Inside the HTML code, we embed them in double braces such as '{{expression}}'.

These are used to inject values that were defined inside of the controller.

Expressions can work with primitive types such as numbers and strings.

For example,

```
{{5+6}}
{{"Hello Aptech Student!"}}
```

Expressions can also be used to work with other types of values such as JavaScript objects and arrays.

For example,

```
{{user.name}}
{{ items[index] }}
```

We can also write AngularJS expressions inside a directive, for example, `ng-bind="expression"`.

AngularJS works out the given expression and returns the result exactly in the place where the expression is written.

AngularJS expressions look similar to JavaScript expressions, however, they have a few limitations as compared to the latter. For example, they do not support regular expressions, functions, loops, or conditional statements.

In short, expressions in AngularJS:

- Bind data to HTML
- Are written inside double braces
- Can be given as `ng-bind="expression"`
- Return the result exactly where the expression is written
- Are quite similar to JavaScript expressions

Code Snippet 7 demonstrates how two literal values are added in an expression.

Code Snippet 7:

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.
min.js"></script>
<body>
<div ng-app="">
  <p>My first expression: {{ 515 + 515 }}</p>
</div>
</body>
</html>
```

The outcome of this would be:

My first expression: 1030

2.6 Data Binding

Data binding in AngularJS is the bringing together of data between the model and the view.

AngularJS applications usually have a data model that represents the system. The data model is the collection of data available for the application.

In the example shown in Code Snippet 8, we have given a `course` variable to represent the course. We defined this through `$scope.course` in the controller. Refer to line-18.

Code Snippet 8: Data Binding with `$scope`

```
1. <!DOCTYPE html>
2. <html lang="en" ng-app="myApp">
3. <head>
4. <meta charset-"UTF-8">
5. <title>Demo for a Simple AngularJS Application</title>
6. <script
7. src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min
   .js">
8. </script>
9. </head>
10.  <body ng-controller="myController">
11.  <div>
12.  <p>Hello!</p>
13.  <p>Welcome to {{course}} </p>
14.  </div>
15.  <script>
16.  var app = angular.module('myApp',[]);
```

```
17.  app.controller('myController',function($scope){
18.  $scope.course = "AngularJS";
19.  });
20.  </script>
21.  </body>
22.  </html>
```

Now, after this definition of `course` variable in the script tag, we can use it in our view with AngularJS expressions.

Only the data on the scope can be accessed in HTML. Whenever scope variable changes, the view is also updated in real time. This is one-way data binding.

2.7 Two-Way Data Binding

In many technologies/frameworks that are template-based, data binding happens only in one direction. This means that templates and model components are merged together into a view at the time of displaying in the browser. However, since they are not bound to one another, any changes made to model or related sections of the view will not be automatically mirrored in the view. Not only that, changes made to the view at runtime are also not reflected in the model. So, for example, if the user enters data in a text box (which is part of the UI, hence view), that data will not be visible in the model automatically.

In such scenarios, to enforce the binding between the model and the view, the developer has to put in hard work.

In AngularJS, two-way data binding can also be achieved with appropriate directives. Two-way data binding is a process wherein binding occurs on both sides, between input and output. Changes made to the user interface are reflected in the data variables and changes made to the data variables are reflected in the user interface. In AngularJS two-way data binding, the model data can be changed either from the model or in the view based on user inputs. The basic principle is this: when any change happens in the model, the view reflects the change, and vice versa. Refer to Figure 2.6.

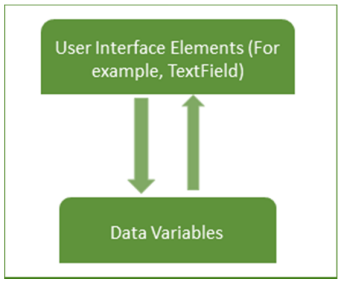


Figure 2.6: Two-Way Data Binding

Two-way binding is limited almost exclusively to basic input elements that use `ng-model`, such as text input, textarea input, or select inputs.

In our Hello application example shown in Code Snippet 6 earlier, the ‘name’ variable as defined in the model was initially an empty string. With `ng-model` directive mapped to text input element, we are modifying ‘name’ with the given user input. The output will be as shown in Figure 2.2 earlier.

Let us see another example for two-way data binding to understand it better. Refer to Code Snippet 9.

Code Snippet 9:

```
1. <!DOCTYPE html>
2. <html lang="en" ng-app="myApp">
3. <head>
4. <meta charset-"UTF-8">
```

```
5. <title>Demo for a Simple AngularJS Application</title>
6. <script
   src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.mi
   n.js"></script>
7. </head>
8. <body ng-controller="myController">
9. <div>
10.  Course <input type="text" ng-model="course" placeholder "enter
   course">
11.  <p>Hello!</p>
12.  <p>Welcome to {{course}}</p>
13.  </div>
14.  <script>
15.  var app = angular.module('myApp',[]);
16.  app.controller('myController',function($scope){
17.  $scope.course = "AngularJS";
18.  });
19.  </script>
20.  </body>
21.  </html>
```

The output of this is shown in Figures 2.7 and 2.8 respectively.

Figure 2.7: Initial Output

Figure 2.8: Output After Text Entry

2.8 Event Handling

When we create more advanced AngularJS applications such as user interacting with a form, it will become necessary to handle DOM events such as mouse moves, mouse clicks, keyboard presses, or events similar to these. AngularJS has a simple way to add event listeners to the HTML of our views.

Following list shows some common AngularJS event listener directives for DOM events:

- ng-click
- ng-dblclick
- ng-keydown
- ng-keypress
- ng-keyup
- ng-mousedown
- ng-mouseenter
- ng-mouseleave
- ng-change

The events they listen to are made obvious by their names.

For example, `<button ng-click="count ()">Click Me.</button>`

Code Snippet 10 shows an example that uses event listener directives as well as two-way data binding.

Code Snippet 10:

```
1. <!DOCTYPE html>
2. <html>
3. <head>
4. <meta charset="UTF-8">
5. <title>Event Registration</title>
6. </head>
7. <body ng-app="EventApp">
8.  <script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js
    ">
9. </script>
10.   <h1> Event</h1>
11.   <div ng-controller="eventcontroller">
12.   New Members: <input type="text" ng-model="quantity" id="quantity"
13.   ng-keypress="total=''">
14.   <br>
15.   <br>
16.   Registration Fees: <input type="text" ng-model="fees" id="fees" >
17.   <input type="button" value="Calculate" ng-click="Calc()" />
18.   <br>
19.   <br>
20.   <b>Total Amount<b>: {{total}}
21.   </div>
22.   <script>
23.   var app = angular.module('EventApp', []);
24.   app.controller('eventcontroller', function($scope) {
25.   $scope.quantity = 0;
26.   $scope.fees = 0;
27.   $scope.total = "";
28.   $scope.Calc = function () {
29.   if ($scope.quantity >0 && $scope.fees >0)
30.   {
31.   $scope.total=$scope.quantity*$scope.fees;
32.   }
33.   else
34.   {
35.   $scope.total="Invalid Input";
36.   }
```

```
37.    }
38.    });
39.    </script>
40.    </body>
41.    </html>
```

The purpose of this application is to accept number of members (as quantity) for an event and the registration fees for the event and then calculate the total amount to be paid. To achieve this, we declare two input elements and a button as part of the user interface. When first loaded, the application should display 0 as the default values for members and fees. Refer to Figure 2.9.

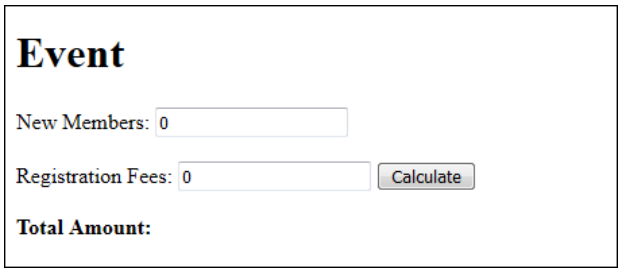


Figure 2.9: Membership Application - Initial View

When user enters values in the text boxes, we need to be able to access them in the controller in order to calculate total. Hence, we use `ng-model` directive to bind the input from the text box to the scope variables. Using this directive will also enable us to initialize these to 0 at the beginning. Refer to the following lines of code from Code Snippet 10:

```
$scope.quantity = 0;
$scope.fees = 0;
$scope.total = "";
```

Through these lines, we initialize the elements present in the DOM. Thus, we are updating the view from the model. When the user enters values in quantity and fees boxes respectively, we calculate total using the following statement:

```
$scope.total = $scope.quantity*$scope.fees;
```

Here, we are retrieving the values of the user interface elements from the view into the model. Then, we update the total in the view.

If, however the user has not chosen to edit default values, the total amount should not be calculated and instead a message saying “Invalid Input” should be displayed beside the Total Amount heading. To do this, we use the following statement:

```
$scope.total = "Invalid Input";
```

Once this is done, even if the user enters values for quantity and fees, the error message would continue to remain there until we press **Calculate** button. To fix this issue, we use the `ng-keypress` event as shown here:

```
New Members: <input type="text" ng-model="quantity" id="quantity"
ng-keypress="total=''">
```

What this will achieve is to reset the total variable to an empty string as soon as the user types a key inside the quantity text box.

When the **Calculate** button is clicked, we want to pass the control to a function in the controller which will calculate the total amount. To do this, we use the `ng-click` event and write the following code:

```
<input type="button" value="Calculate" ng-click="Calc()" />
```

This event will be handled by its listener and will call the `Calc()` function in the controller.

Quick Test 2.1

1. We can have an AngularJS application without any ng-app directive.
   a. True
   b. False

2. It is compulsory to have a value for ng-app directive in an AngularJS application.
   a. True
   b. False

3. It is compulsory to have an ng-controller directive in an AngularJS application.
   a. True
   b. False

2.9 Summary

- AngularJS applications have to link to AngularJS file. It can be to a CDN or a locally available file.
- We need to inform AngularJS which section of the HTML will be controlled by it through the '`ng-app`' directive.
- In AngularJS, modules are containers used to hold other parts of an application. It is the holder for controllers, services, filters, directives, and so on.
- AngularJS applications depend on controllers to control the flow of data in the application.
- Views are what the user gets to see in an application.
- AngularJS expressions are similar to JavaScript code snippets. Inside the HTML code, we embed them in double braces.
- In two-way data binding, the model data can be changed either from the model or in the view.

2.10 Exercise

1. We use _____ directive to inform which part of the HTML will be controlled by AngularJS.

   a) `ng-controller`
   b) `ng-app`
   c) `ng-init`
   d) `ng-model`

2. Which of the following inputs are used for two-way data binding?
   a) `Text`, `textarea`, and `select`
   b) `Label`, `legend`, and `fieldset`
   c) `Datalist`, `output`, and `option`
   d) All of these

3. _____ is the glue between application controller and view.
   a) Scope
   b) Data
   c) Model
   d) $array

4. In AngularJS, _____ are containers used to hold other parts of an application.
   a) Models
   b) Modules
   c) Controllers
   d) Directives

5. AngularJS applications depend on _____ to control the flow of data in the application.
   a) Models
   b) Modules
   c) Controllers
   d) Directives

2.11 Do It Yourself

1. Create an AngularJS application that accepts Product Price and Discount percentage as input. Then, the application should display the Amount payable after discount is applied. For example, if 250 is entered as Product Price and 8 is entered as Discount percentage, the Amount payable should be 230. Use expressions and two-way data binding to achieve this.

2. Create an AngularJS application that accepts name, date of birth, and mobile number and displays them.

3. Create an AngularJS application that accepts weight of two items and number of units of each item. Then it should calculate and display average weight of the combined items.

   For example,

   ```
   Weight of first item: 15
   No. of units of first item: 5
   Weight of second item: 25
   No. of units of second item: 4
   Average Weight = 19.444444
   ```

   Hint: Use this formula: result = ((wi1 * qty1) + (wi2 * qty2)) / (qty1 + qty2)

Answers to Exercise

1. `ng-app`
2. Text, textarea, and select
3. Scope
4. Modules
5. Controllers

Answers to Quick Test

Quick Test 2.1

1. False
2. False
3. False

Session 3

Directives, Filters, and Routes

In this session, students will learn to:

- Describe the directives in AngularJS
- List and explain the important built-in filters
- Describe routes in AngularJS applications

3.1 Introduction to Directives

AngularJS is a framework that can be used to build interactive Web applications. Since it is a framework, it has built-in support by means of various components to make our development work easier. One of these is directives. Directives are one of the most important components of AngularJS.

They help us to extend basic HTML elements and create reusable and testable code.

Refer to Code Snippet 1, which shows sample code for an AngularJS application.

Code Snippet 1:

```
<!DOCTYPE html>
<html>
<head>
<title>AngularJS Directives</title>
<script src =
"https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
</head>

<body>
<h1>Sample AngularJS Application</h1>

<div ng-app = "" ng-init = "countries = [{locale:'en-US',name:'United
States'},
{locale:'en-GB',name:'United Kingdom'}, {locale:'en-FR',name:'France'}]">
<p>Enter Your Name: <input type = "text" ng-model = "name"></p>
<p>Hello <span ng-bind = "name"></span>!</p>
<p>List of Countries with locale code:</p>

<ol>
<li ng-repeat = "country in countries">
{{ 'Country Name: ' + country.name + ', Locale Code: ' + country.locale }}
</li>
</ol>
</div>
</body>
</html>
```

Notice Code Snippet 1 has an attribute, `ng-app`' added to the div. It is not a standard attribute of div element. It is relevant to AngularJS and gives instructions to the browser that from this element onwards and including all its child elements, AngularJS will have control over the behavior of the application.

In a similar way, a new attribute `ng-init` is used. This directive initializes AngularJS application data. It defines initial values for an AngularJS application.

The `ng-model` attribute instructs and guide behaviors of the application.

The starting point of any AngularJS application is any element where `ng-app` directive is provided. All AngularJS applications must have an `ng-app` directive.

All of these attributes described so far are AngularJS directives. AngularJS comes with a wide range of directives. They become available to the application when `angular.min.js` file is linked in our HTML code. Observe the Code Snippet 1. Built-in directives are special attributes starting with `ng-prefix` in AngularJS, which offers functionality to the applications. Prefix **ng** stands for 'Angular'. In addition to this, a user can create his/her custom directives for any application.

Figure 3.1 shows the output for Code Snippet 1.

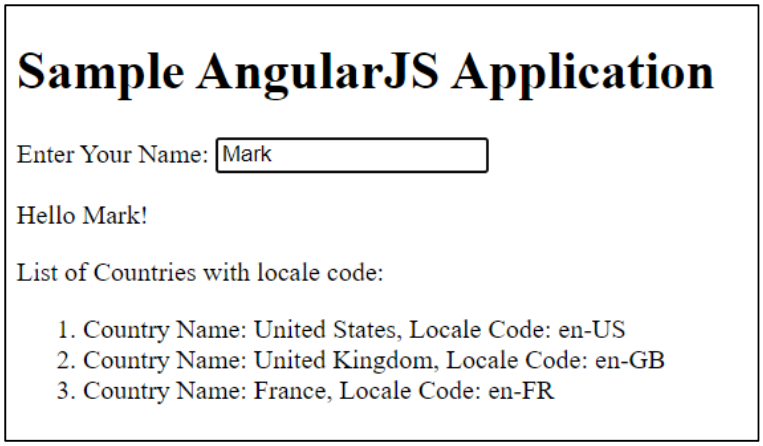


Figure 3.1: Output of Code Snippet 1

3.2 Common Directives

Some of the common and important directives of AngularJS are as follows:

- `ng-app`
- `ng-init`
- `ng-model`
- `ng-repeat`
- `ng-hide`
- `ng-show`
- `ng-include`

`ng-app`:

The `ng-app` directive is used to start an AngularJS application. It defines the root element of an AngularJS application. It is mandatory for all AngularJS applications to have a root element. By design, `ng-app` directive initializes the application when the Web page containing AngularJS application is loaded in the browser. It also loads various AngularJS modules in the application.

In Code Snippet 2, the application is defined using `ng-app` attribute for a div element. AngularJS framework will only process the DOM elements and its child elements where the `ng-app` directive is applied.

Code Snippet 2:

```
<!DOCTYPE html>
<html>
<head>
 <title>ng-app Directive Test</title>
 <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
</head>
<body >
 <div>
 {{4/2}}
 </div>
 <div id="myDivTag" ng-app>
```

```
 {{10/7}}
 <div>
 {{10/2}}
 </div>
 </div>
 <div>
 {{6/2}}
 </div>
</body>
</html>
```

In Code Snippet 2, `ng-app` directive is placed in the div element whose id is `myDivTag`. Therefore, AngularJS will only compile `myDivTag` and its child elements. It will not compile the parent or sibling elements of `myDivTag`. The output will be as shown in Figure 3.2.

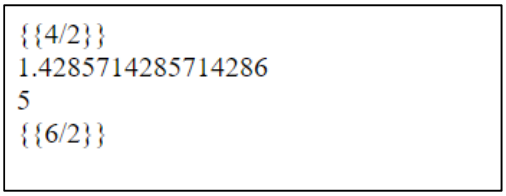


Figure 3.2: `ng-app` in div Tag

Note: Multiple `ng-app` directives are NOT allowed in a single HTML document.

The user can decide whether to give a value for `ng-app` or not. If a name is provided, such as `myAngularApp`, then AngularJS will look for it in the JavaScript code. The module name mapping for `myAngularApp` will be defined in JavaScript, as shown in Code Snippet 3.

Code Snippet 3:

```
<!DOCTYPE html>
<html>
<head>
 <title>ng-app Directive Test</title>
 <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
</head>
<body ng-app="myAngularApp">
 <div>
 {{4/2}}
 </div>
 <div>
 {{10/7}}
 <div>
 {{10/2}}
 </div>
 </div>
 <script>
 var app = angular.module('myAngularApp', []);
 </script>
</body>
</html>
```

In some apps, users can just give an empty string as a value to `ng-app`.

For example,

```
<div ng-app = "">
 ...
</div>
```

Another example is `ng-app` without any value.

For example,

```
<div ng-app>
 ...
</div>
```

These indicate that the module does not have any name.

`ng-init`:

The `ng-init` directive is used for initializing AngularJS application data in HTML. It is used to set values for variables that are used in the application. Code Snippet 4 shows how to initialize values in an array using `ng-init` directive.

Code Snippet 4:

```
<!DOCTYPE html>
<html >
<head>
 <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
</head>
<body >
 <div ng-app ng-init="greet='Hello '; amount= 100; myArr = [200, 300];
person = { firstName:'Bill', lastName :'Gates'}">
 {{greet}}
 {{person.firstName}} <br/>
            {{amount}}
 <br />
 {{myArr[1]}}
 </div>
</body>
</html>
```

In Code Snippet 4, variables for string, number, array, and object are initialized. These variables can be used anywhere in the DOM element hierarchy where they are declared.

`ng-model`:

The `ng-model` directive is used for two-way data binding in AngularJS. It binds `<input>`, `<select>`, or `<textarea>` elements to a specified property on the `$scope` object. Property value that is set via ng-model can be accessed in a controller using `$scope` object.

The `ng-model` directive can also provide additional functionalities such as:

- Provide type validation for application data (number, email, and required).
- Provide status for application data (invalid, dirty, touched, and error).
- Provide CSS classes for HTML elements.
- Bind HTML elements to HTML forms.

Let us use ‘ng-model’ in a simple example. Refer to the Code Snippet 5.

Code Snippet 5:

```
<!DOCTYPE html>
<html >
<head>
 <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
</head>
<body ng-app>
 Enter Your Name <input type="text" ng-model="name" />
 <div>
 Hello {{name}}
 </div>
</body>
</html>
```

When this code is executed in the browser, output is displayed as shown in Figure 3.3.

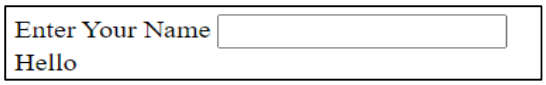


Figure 3.3: ng-model Example – Initial Web Page

If we enter the name, say 'Mark', the page changes as shown in Figure 3.4.

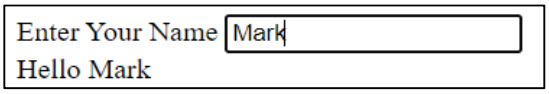


Figure 3.4: ng-model Example –After Input

`ng-repeat`:

The `ng-repeat` directive is used to repeat html elements for each item in a collection, which could be an array or an object. It is a powerful directive that can iterate over thousands of data with a single line of code.

Code Snippet 6 shows an example for `ng-repeat` directive:

Code Snippet 6:

```
<!DOCTYPE html>
<html>
<head>
 <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
 <style>
 div {
 border: 1px solid green;
 width: 100%;
 height: 50px;
 display: block;
 margin-bottom: 10px;
 text-align:center;
 background-color:yellow;
 }
 </style>
</head>
<body ng-app="" ng-init="owners=['Bill','Steve','Jack']">
 <ol>
 <li ng-repeat="name in owners">
```

```
 {{name}}
 </li>
 </ol>
 <div ng-repeat="name in owners">
 {{name}}
 </div>
</body>
</html>
```

In Code Snippet 6, `ng-repeat` is used with the 'owners' array. It creates an `<li>` element for each item in the 'owners' array and in a similar way, it repeats the `<div>` element.

This application when run in a browser produces output as shown in Figure 3.5.

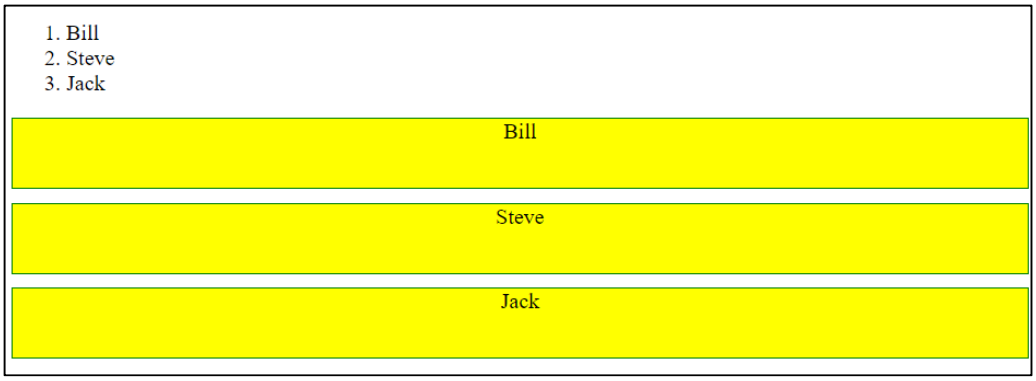


Figure 3.5: ng-repeat Directive

`ng-hide`:

The `ng-hide` directive is used to hide an element depending on some condition. Single page applications will most likely have many parts that will come and go as the state of the application changes.

This directive requires minimal code for usage. With this directive, developers need not write lengthy CSS or JavaScript code for hiding or showing data. AngularJS manages all the hiding and showing of data when this directive is applied.

The `ng-hide` directive hides specified HTML element if the expression evaluates to true.

Code Snippet 7 shows an application with `ng-hide` directive.

Code Snippet 7:

```
<!DOCTYPE html>
 <html lang="en">
 <head>
      <meta charset="UTF-8">
      <title>Demo for ng-show and ng-hide</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
 </head>
 <body ng-app="">
      <p>Do you want to hide the list of cities?</p>
      <input type="checkbox" ng-model="myVar">
      <ul ng-hide = "myVar">
<li>Nairobi</li>
<li>London</li>
<li>Paris</li></ul>
 </body>
</html>
```

Figure 3.6 shows the initial screen when this application is executed.

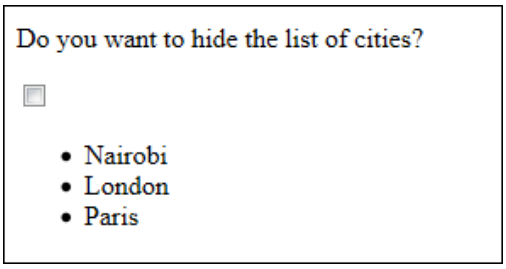


Figure 3.6: ng-hide Directive – Initial Screen

Figure 3.7 shows the output when the checkbox is clicked by the user. It hides all the cities.

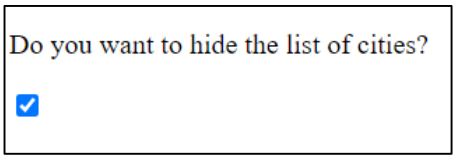


Figure 3.7: ng-hide Directive – Cities Hidden

`ng-show`:

The `ng-show` directive shows the HTML elements if the expression evaluates to true.

Code Snippet 8 shows an application with `ng-show` directive.

Code Snippet 8:

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
<body ng-app="">
Show HTML element: <input type="checkbox" ng-model="myVar">
<div ng-show="myVar">
<h1>Welcome to ng-show directive example</h1>
<p>A solution of ng-show directive.</p>
</div>
</body>
</html>
```

Figure 3.8 shows the initial screen when we run this application.

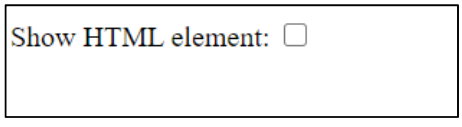


Figure 3.8: ng-show Directive– Initial Screen

Figure 3.9 shows the output when the checkbox is clicked. On clicking the checkbox, the expression given to `ng-show` becomes true and the menu is displayed.

Figure 3.9: ng-show Directive – Elements

ng-include:

HTML code by itself does not have the option to embed another HTML page in itself. It is possible using AngularJS to embed another HTML file directly within an application. This is useful in an application, when there is a large amount of content to be rendered via markup.

The `ng-include` directive is used to embed another HTML page in our application's HTML page. Generally, the embedded HTML page is from the same server.

Code Snippet 9 shows an application with `ng-include` directive pointing to the pages `mainContent.html` and `sideBar.html` in the same directory.

Code Snippet 9:

```
<!DOCTYPE html>
<html>
 <head>
 <script
src="http://ajax.googleapis.com/ajax/libs/angularjs/1.4.1/angular.min.js">
</script>
 </head>

 <body ng-app="spApp">
 <h1>ng-include directive</h1>
 <div ng-controller="SPController" class="container">
 <div ng-include="'mainContent.html'"></div>
 <div ng-include="'sideBar.html'"></div>
 </div>
 </body>
 <script>
 var MyApp = angular.module('spApp', []);
 MyApp.controller("SPController", function($scope) {
 $scope.quantity = 1;
 $scope.price = 9.99;
 });
 </script>
</html>
```

Code Snippet 9 is used for embedding the `mainContent.html` and `sidebar.html` pages respectively in our application.

Code Snippets 10 and 11 show the content of the `mainContent.html` and `sidebar.html` page respectively.

Code Snippet 10:

```
<div style="background-color:lightblue">
 <h3>Lessons</h3>
 <p>Lessons.com was established in Nov 2018. Lessons was started with a
motive to present programming codes and to help the developers all around
the globe with different queries in different Programming Languages.</p>
</div>
```

Code Snippet 11:

```
<div class="sidebar" style="background-color:lightyellow">
 <h3>Tutorials list(sidebar)</h3>
 <li>
 A<br>
 B<br>
 C<br>
 D<br>
 </li>
</div>
```

When this application is executed in a browser, output is displayed as shown in Figure 3.10. Ensure that it is run via the local server (localhost) otherwise the embedded HTML files will not load.

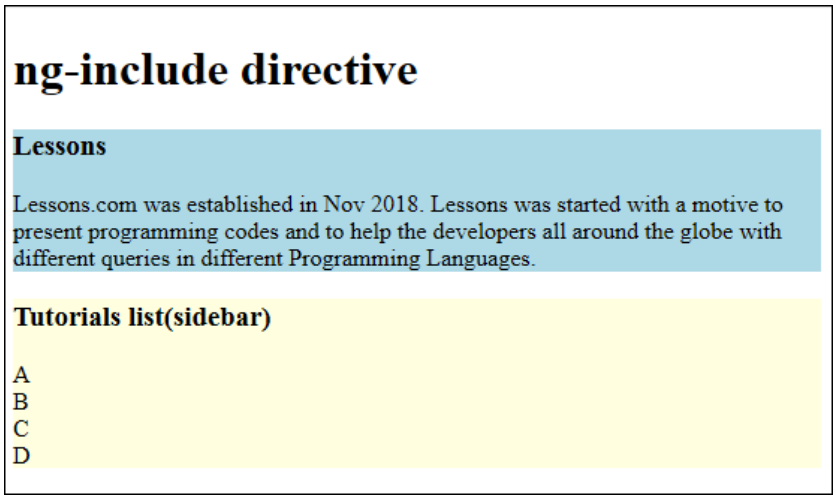


Figure 3.10: ng-include Directive

3.3 Filters

AngularJS filters allow formatting of data for visual purposes without changing the original format. Depending on the application requirement, one can display data to users from the model. Data available from the model may not be in formatted form in which it should be displayed to the user. Data available in our model/database will be unformatted in most of the cases.

For example, numbers may have too many digits after the decimal point, names of clients may be entirely in uppercase, or dates could be formatted in the wrong way.

When data is presented to a user, it must be formatted in suitable format as per user or application requirements. This is where filters come into picture.

Filters format value of an expression in order to display it to end users. Filters can be used with an expression or directives using pipe | sign.

For example, `{{expression | filter}}`

Filters can be used in view templates as well as in controllers. Filters can also be chained. That means more than one filter can be used in an expression.

For example, `{expression | filter | filter}}`

In this case, the second filter will be applied to the output of the first filter. Another point to remember is that filters do not change the underlying data in the database. They only perform the formatting for display purposes.

AngularJS comes with a collection of built-in filters. In addition, it also provides methods to define your own filters when you require something beyond standard built-in filters functionalities.

3.3.1 Built-in Filters

Following list shows some common AngularJS built-in filters:

- number
- currency
- date
- lowercase
- uppercase
- search

Number:

Number filter formats numeric data as text with comma and to the specified fraction size. Following is the syntax for Number filter:

```
{{ number_expression | number:fractionSize}}
```

If a specified expression does not return a valid number, then the number filter displays an empty string. Code Snippet 12 demonstrates how to use the number filter with number expression or a model property.

Code Snippet 12:

```
<!DOCTYPE html>
<html>
<head>
 <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
</head>
<body ng-app >
<h1>AngularJS Number Filter</h1>
 Enter the amount to format: <input type="number" ng-model="amount" /> <br
/>

 100000 | number = {{100000 | number}} <br />
 amount | number = {{amount | number}} <br />
 amount | number:2 = {{amount | number:2}} <br />
 amount | number:4 = {{amount | number:4}} <br />
 amount | number = <span ng-bind="amount | number"></span>
</body>
</html>
```

Output of Code Snippet 12 is displayed in Figure 3.11.

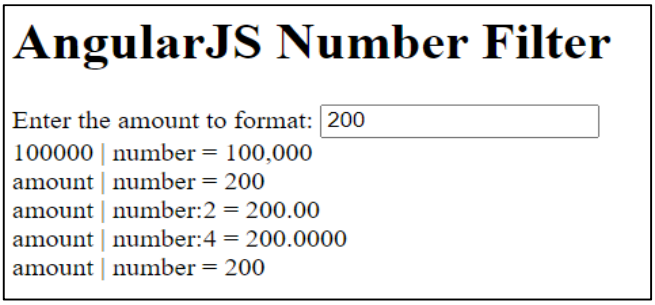


Figure 3.11: Number Filter

The `ng-bind` directive in AngularJS is used to bind text content of any particular HTML element with value that is entered in the given expression. Value of specified HTML content updates whenever the value of expression changes in `ng-bind` directive.

In this example, the `ng-bind` directive is used to bind value entered by user to the `<span>` element.

Currency:

Currency filter formats a number value as a currency. When no currency symbol is provided, the default currency symbol for the current locale is used.

```
{{ expression | currency : 'currency_symbol' : 'fraction'}}
```

In Code Snippet 13, currency filter is applied to `person.salary`, which is a numeric property. It can be displayed with different currency symbols and fractions.

Code Snippet 13:

```
<!DOCTYPE html>
<html>
<head>
 <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
</head>
<body ng-app="myApp" >
<h1>AngularJS Currency Filter</h1>
 <div ng-controller="myController">
 Default Currency: {{person.salary | currency}} <br />
 Custom Currency Identifier: {{person.salary | currency:'Euro.'}} <br />
 Without Fraction: {{person.salary | currency:'Euro':0}} <br />
 Two Decimal Fraction: <span ng-bind="person.salary|
currency:'GBP':2"></span>
 </div>
 <script>
 var myApp = angular.module('myApp', []);

 myApp.controller("myController", function ($scope) {
 $scope.person = { salary: 100000}
 });
 </script>
</body>
</html>
```

Output of Code Snippet 13 is shown in Figure 3.12.

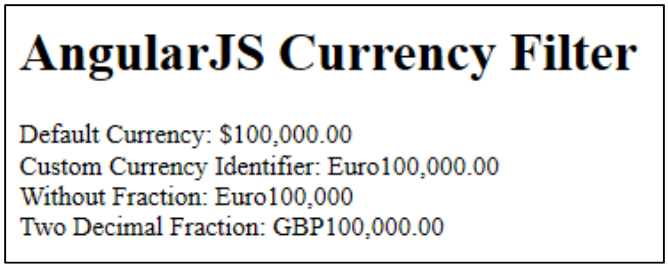


Figure 3.12: Currency Filter

Date:

This filter formats date to string based on the specified format.

```
{{ date_expression | date : 'format'}}
```

Common values used in format:

- 'YYYY' – define year
- 'YY' – define year
- 'Y' – define year
- 'MMMM' – define month
- 'MMM' – define month
- 'MM' – define month
- 'dd' – define day
- 'd' – define day
- 'hh' – define hour in AM/PM
- 'h' – define hour in AM/PM
- 'mm' – define minute
- 'm' – define minute
- 'ss' – define second
- 's' – define second

Predefined values for format:
“short” – equivalent to “M/d/yy h:mm a”
“medium” – equivalent to “MMM d, y h:mm:ss a”
“shortDate” – equivalent to “M/d/yy” (5/7/19)
“mediumDate” – equivalent to “MMM d, y” (May 7, 2019)
“longDate” – equivalent to “MMMM d, y” (May 7, 2019)
“fullDate” – equivalent to “EEEE, MMMM d, y” (Tuesday, May 7, 2019)
“shortTime” – equivalent to “h:mm a” (2:35 AM)
“mediumTime” – equivalent to “h:mm:ss a” (2:35:05 AM)

Code Snippet 14 shows the code to display date and time in different formats for today’s date.

Code Snippet 14:

```
<!DOCTYPE html>
<html>
 <head>
 <title>Date Filter</title>
  <script src =
"https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js" >
 </script>
 </head>
  <body>
 <h1>AngularJS Date Filter</h1>
 <div ng-app="gfgApp" ng-controller="dateCntrl">
 Default Date: {{ today| date }}<br/>
 Short: {{ today| date:'short' }}<br/>
 Medium: {{ today| date:'medium' }}<br/>
 Short Date: {{ today | date:'shortDate' }}<br/>
 Long Date: {{ today| date:'longDate'}}<br/>
 Medium Date: {{ today| date:'mediumDate'}}<br/>
 Full Date: {{ today| date:'fullDate'}}<br/>
 Year: {{today | date:'yyyy'}} <br/>
 Short Time: {{ today| date:'shortTime'}}<br/>
 Medium Time: {{ today| date:'mediumTime'}}<br/>
  </div>
  <script>
 var app = angular.module('gfgApp', []);
 app.controller('dateCntrl', function($scope) {
 $scope.today = new Date();
 });
 </script>
 </body>
</html>
```

Output of Code Snippet 14 is shown in Figure 3.13.

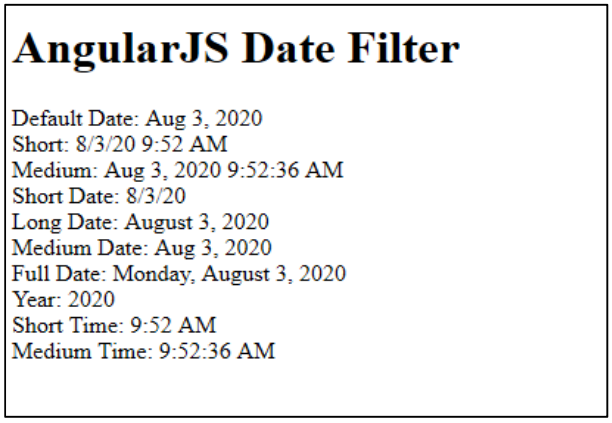


Figure 3.13: AngularJS Date Filter

Uppercase and Lowercase

Uppercase filter converts the string to uppercase and lowercase filter converts the string to lowercase. Following are the syntaxes for uppercase and lowercase filters:

```
{{ firstName | uppercase }}
{{firstName | lowercase }}
```

Code Snippet 15, shows sample code for uppercase and lowercase filters.

Code Snippet 15:

```
<!DOCTYPE html>
<html>
<head>
 <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
</head>
<body ng-app >
<h1>AngularJS Uppercase and Lowercase Filter</h1>
 <div ng-init="person.firstName='James';person.lastName='Bond'">
 Lower Case: {{person.firstName + ' ' + person.lastName | lowercase}} <br
/>
 Upper Case: {{person.firstName + ' ' + person.lastName | uppercase}}
 </div>
</body>
</html>
```

Output of Code Snippet 15 is displayed in Figure 3.14.

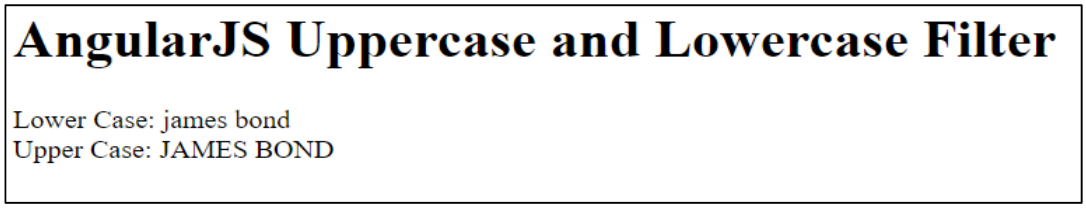


Figure 3.14: Uppercase and Lowercase Filter

Filter:

Filter selects items from an array based on the specified criteria and returns a new array.

```
{{ expression | filter : filter_criteria }}
```

Code Snippet 16 shows the code to filter items of an array `myArr` using `filter:searchCriteria` expression. This expression will be created using text entered in the input box.

Code Snippet 16:

```
<!DOCTYPE html>
<html>
<head>
 <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
</head>
<body ng-app="myApp" >
<h1>AngularJS Filter</h1>
 <div ng-controller="myController">
 Enter name to search in array: <input type="text" ng-
model="searchCriteria" /> <br />
```

```
 Result: {{myArr | filter:searchCriteria}}
 </div>
 <script>
 var myApp = angular.module('myApp', []);

 myApp.controller("myController", function ($scope) {
 $scope.myArr = ['Steve', 'Bill', 'James', 'Rob', 'Ram', 'Moin']
 });
 </script>
</body>
</html>
```

Initial output for Code Snippet 16 is shown in Figure 3.15.

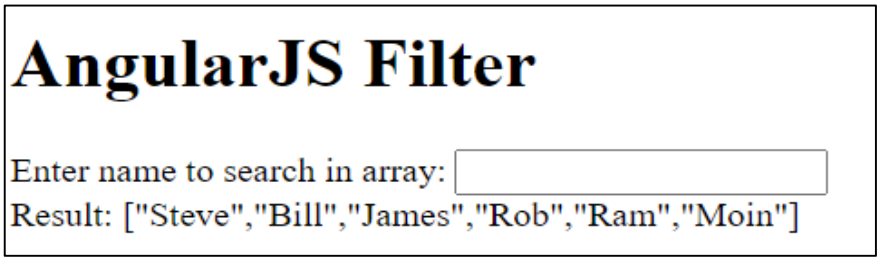


Figure 3.15: AngularJS Filter – Initial Screen

Output shown in Figure 3.16 is displayed, when 'Rob' is entered as an input field as the search string.

You can try this with different inputs and see how the application responds.

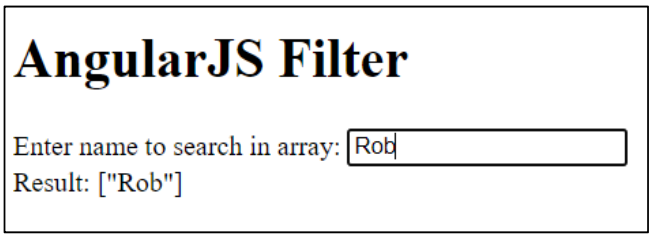


Figure 3.16: AngularJS Filter - After Input

3.3.2 Custom Filters

Built-in filters are useful for many common use cases such as formatting dates, currencies, cases or limiting the number of items displayed and so on. For some of the requirements, users may require some specific output that built-in filters are unable to provide, in those cases we go in for constructing our custom filters.

AngularJS gives us an API for creating a filter.

Controller can be defined as follows:
`app.controller('myCtrl', function(){});`

Also, a filter can be created by registering a new filter factory function with a module as follows:
`app.filter('filterName', function(){});`

Code Snippet 17, shows a custom filter defined as `myFilter`. This `myFilter` filter will format every other character to uppercase.

Code Snippet 17:

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
```

```
</script>
<body>
<h1>Custom Filter</h1>
<ul ng-app="myApp" ng-controller="namesCtrl">
<li ng-repeat="x in names">
 {{x | myFilter}}
</li>
</ul>
<script>
var app = angular.module('myApp', []);
app.filter('myFilter', function() {
 return function(x) {
 var i, c, txt = "";
 for (i = 0; i < x.length; i++) {
 c = x[i];
 if (i % 2 == 0) {
 c = c.toUpperCase();
 }
 txt += c;
 }
 return txt;
 };
});
app.controller('namesCtrl', function($scope) {
 $scope.names = [
 'Jani',
 'Carl',
 'Margareth',
 'Hege',
 'Joe',
 'Gustav',
 'Birgit',
 'Mary',
 'Kai'
 ];
});
</script>
<p> This filter, called 'myFilter', will capitalize every other character.
</p>
</body>
</html>
```

Notice the `myFilter` filter object defined in the code snippet. This returns an inline defined function, which performs the custom filter task and returns filtered output for elements in array.

In this example, custom filter takes a string as an input from an array one by one. For each string, it finds positions divisible by two, converts the character at that position to uppercase, and then moves to the next position. This process will continue until the end of the string and the end of the array.

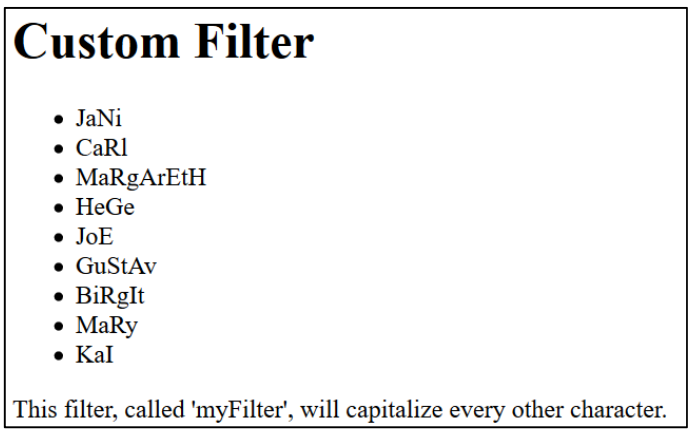


Figure 3.17: Custom Filter

Even though this is a simple and fun custom filter that we made, the mechanism will be the same even for bigger and practical applications.

3.4 Routes

AngularJS provides us a great way to build single page applications.

In conventional Web applications, the browser starts the communication with the server by requesting a page. The server then processes the request and gives the HTML, CSS, and other resources of the page to the client. In subsequent exchanges with the page, a new request is sent to the server and the flow starts again: the server processes the request and sends a new page to the browser in response to the new action requested by the client.

In SPAs, the entire page is loaded in the browser after the initial request, but the interactions will take place through AJAX requests. AJAX which stands for Asynchronous JavaScript and XML, is a non-blocking technique for a client to communicate with a Web server without reloading the page. This means that the browser has to update only the portion of the page that has changed; there is no need to reload the entire page.

The SPA approach reduces the response time to the user actions and the user gets a more fluid experience.

Let us look at Figure 3.18, which shows a wireframe of a Web page.

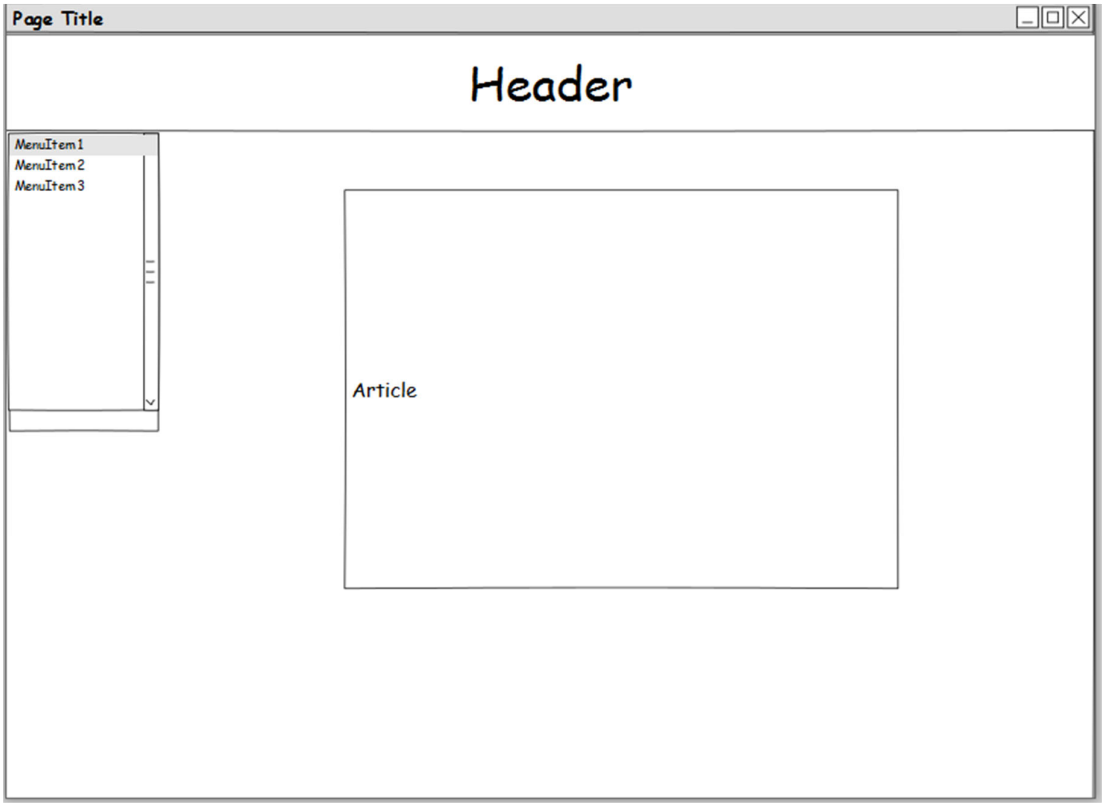


Figure 3.18: Web Page Template

After the page is loaded, suppose the user asks for menu Item 2. Instead of the server responding with a completely new page, it can send only the new content of the article for menu Item 2. Then, that element alone is updated in the SPA.

When creating SPAs, routing is very important. It provides navigation that feels similar to a normal application, that too without refreshing the content of site.

The `ngRoute` module helps our application to become an SPA.

The `ngRoute` module routes our application to different pages without reloading the entire application.

3.4.1 Routes Demo

Let us now see an SPA that uses routing.

To make our applications ready for routing, it is necessary to include the AngularJS Route module, in addition to `angular.min.js`. This module is available through the angular-route.js file.
`<scriptsrc="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular-route.js"></script>`

For this, it requires `ngRoute` as a dependency in the application module:

```
var app = angular.module("myApp", ["ngRoute"]);
```

Now, our application has access to the route module, which provides the `$routeProvider` method.

The `$routeProvider` is used to configure different routes in our application.
Code Snippet 18 contains code for a simple United Kingdom (UK) Tourism application, which has pages displaying top visited places in the UK.

Code Snippet 18:

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular-
route.js">
</script>
<body ng-app="myApp" bgcolor="lightyellow">
<p><a href="#/!">UK Tourism</a></p>
<a href="#!Dorset">Dorset</a>
<a href="#!Leeds">Leeds</a>
<a href="#!Cardiff">Cardiff</a>
<p>Click links to read about Dorset, Cardiff, and Leeds.</p>
<div ng-view style="background-color: lightblue"></div>
<script>
var app = angular.module("myApp", ["ngRoute"]);
app.config(function($routeProvider) {
$routeProvider
.when("/",{
templateUrl : "UKTourism.html"
})
.when("/Dorset", {
templateUrl : "dorset.html"
})
.when("/Leeds", {
templateUrl : "leeds.html"
})
.when("/Cardiff", {
templateUrl : "cardiff.html"
});
});
</script>
</body>
</html>
```

This application requires a container to put the content provided by the routing.

This container is the `ng-view` directive. Observe the ng-view directive defined in Code Snippet 18.

Applications can have only one `ng-view` directive, and this acts as the placeholder for all views provided by the route.

With the `$routeProvider`, you can define what page to display when a user clicks any of the links provided such as, Dorset, Cardiff, and Leeds.

Figures 3.19, 3.20, 3.21, and 3.22 displays outputs for Code Snippet 18 based on the inputs given.

Figure 3.19: Routing Home Page

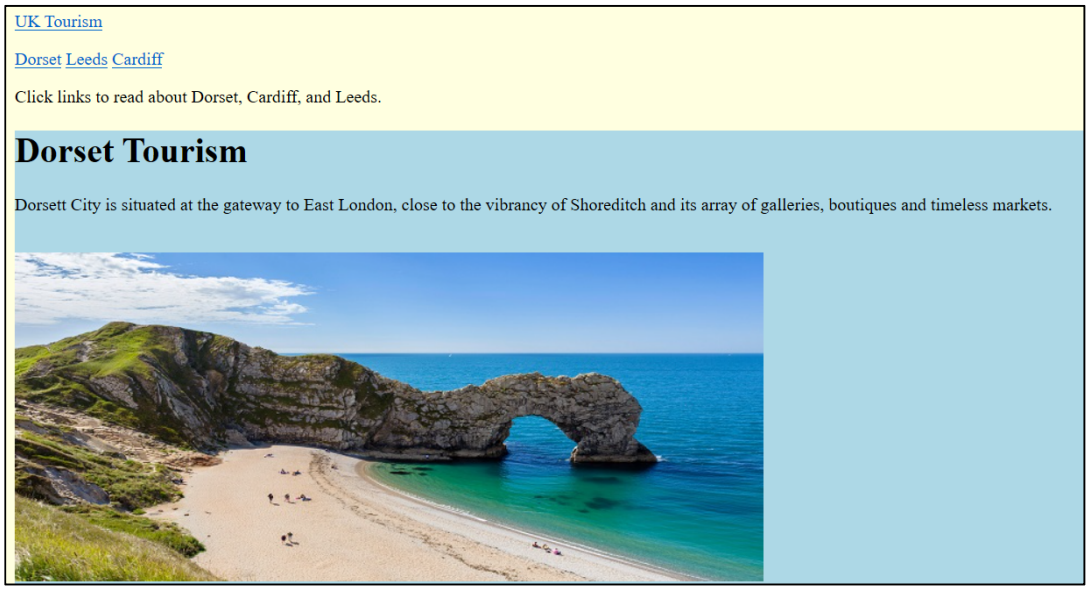


Figure 3.20: Dorset

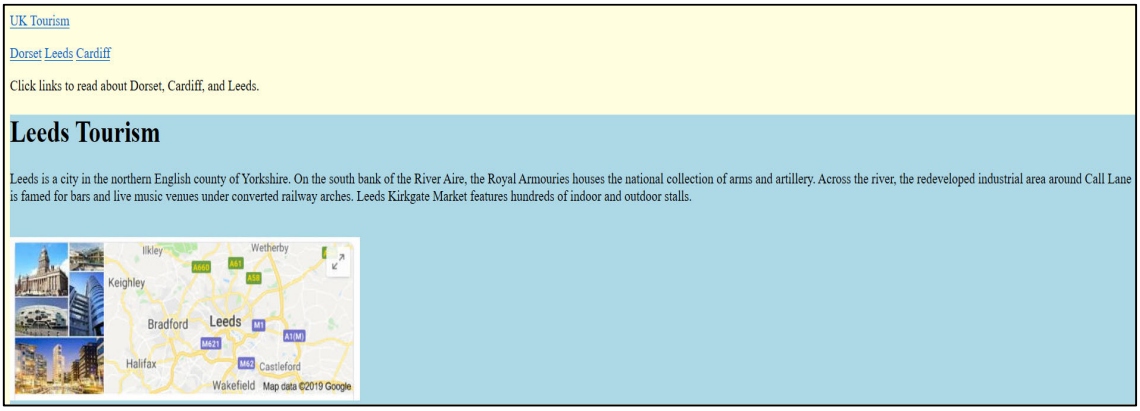


Figure 3.21: Leeds

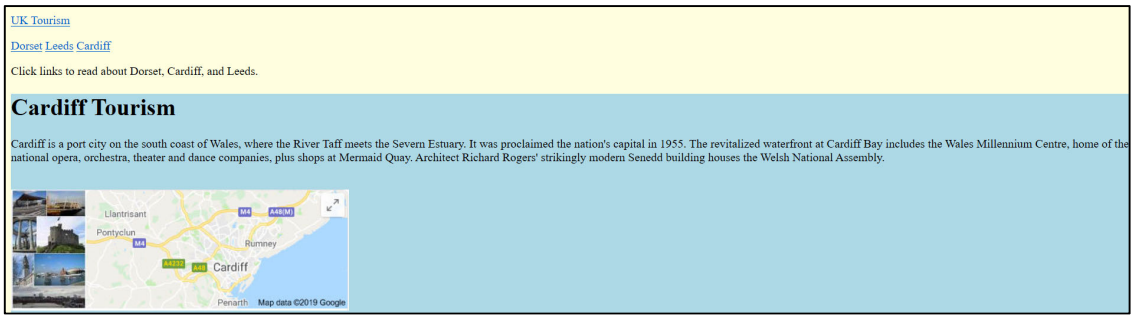


Figure 3.22: Cardiff

In all these cases, the change takes place only in the light-blue colored div element and not on the whole page.

Quick Test 3.1

1. An HTML page can be embedded into another HTML page directly in an AngularJS app.
   a. True
   b. False

2. AngularJS app can contain multiple ng-views.
   a. True
   b. False

3. Which directive can bind the input elements data to model data?
   a. `ng-init`
   b. `ng-model`

3.5 Summary

- Directives are one of the most important components of AngularJS that help us extend basic HTML elements.
- The starting point of any AngularJS application is the element where `ng-app` directive is provided.
- The `ng-model` directive is used to bind the value obtained from the user HTML inputs to model data.
- The `ng-repeat` is a powerful directive that can iterate over thousands of data with a single line of code.
- The `ng-hide` directive hides the specified HTML element if the given expression evaluates to true.
- The `ng-show` directive shows the specified HTML element if the given expression evaluates to true.
- The `ng-include` directive is used to embed one HTML page into another HTML page.
- Filters do not change the underlying data in the database. They only do the formatting for display and `$routeProvider` is used to configure different routes in our application.

3.6 Exercise

1. Which component of AngularJS helps to extend HTML elements?

   a) Directives
   b) Services
   c) Routes
   d) Filters

2. Which directive is used for initializing AngularJS application data in HTML?
   a) `ng-start`
   b) `ng-init`
   c) `ng-show`
   d) `ng-begin`

3. Which directive is used for binding the value obtained from the user from HTML inputs to application data?

   a) `ng-src`
   b) `ng-app`
   c) `ng-controller`
   d) `ng-model`

4. Filters are added to expressions by using which character?
   a) Colon :
   b) Semicolon ;
   c) Pipe |
   d) Hyphen –

5. The _____ module routes our application to different pages without reloading the entire application.

   a) `ngRoute`
   b) `ngGo`
   c) `ngPath`
   d) `ngForm`

3.7 Do It Yourself

1. Create an AngularJS application that will accept basic salary and bonus percentage for employee. Then it should calculate the total salary as sum of bonus amount and basic salary. Bonus amount will be calculated as bonus percentage of the basic salary.

   For example, if basic salary is 5000 and bonus percentage is 5%, bonus amount will be 250. Then, display the total salary as 5250. Accept the values of salary from input and then, calculate and display the total salary in dollars.

   Also, display today's date and current time. Make use of as many directives and filters as you need.

2. Create an AngularJS application that will display names of weekdays in title case.

Answers to Exercise

1. Directives
2. ng-init
3. ng-model
4. Pipe |
5. ngRoute

Answers to Quick Test

Quick Test 3.1

1. True
2. False
3. ng-model

Session 4

Custom Directives, Scope, and Services

In this session, students will learn to:

- Outline the process to create and use a custom directive
- Describe the concept of scope in AngularJS
- Explain services in AngularJS

4.1 Creating a Custom Directive

AngularJS provides a wide range of built-in directives that can take care of most of the requirements of an application.

Along with these, it also gives a way to create our own application specific, custom directives, in case the built-in directives do not work the way we require.

Sometimes, it could also be a case where we want to create self-contained functionality that we can reuse in other applications.

Let us go through the basic process that we have to follow to create and use a custom directive.

We create new directives using the `.directive` method; arguments we provide to this method are the name of the new directive and a function that creates the directive.

Following fragment of code shows creation of `myCustomDirective`:

```
var app = angular.module('myApp', []);
app.directive('myCustomDirective', function() {
  return {
        restrict: 'AE',
        template: '<h3>Hello AngularJS!!</h3>
        <p>I was made inside a Custom directive<p>'
  };
});
```

The purpose of this directive is to greet students with a custom message. The first argument that we pass to the directive method sets name of the new directive to `myCustomDirective`. We have to use the camel case convention for the name, meaning that the first word is lowercase and subsequent words' first letters are capitalized.

Notice that the function in the created directive returns a JavaScript object with some properties. The returned object is also known as 'Directive Definition Object' or DDO in short.

The `restrict` property defines how this directive needs to be used or what is the type of this directive. The `template` tells what the directive will show in the HTML view. We can use `templateUrl` to point to an HTML document, instead of writing the markup. This option is useful if the content to be displayed is long and we wish to shorten the code.

For example, instead of giving the entire HTML markup for *app.component.html* with `template`, we can write:

```
templateUrl: './app.component.html'
```

This makes the code compact.

4.2 Invoking a Custom Directive

To call a custom directive in HTML, we can simply use it as an element.

For example,

```
<body ng-app= "myApp">
      <my-custom-directive></my-custom-directive>
</body>
```

On the view, the directive is used by separating the camel case name by using a hyphen/dash, to separate the words. Observe the yellow highlighted line. We need to follow this convention strictly.

In this example, we used the custom directive as an element in HTML. This can also be used as an attribute in HTML.

For example,

```
<body ng-app= "myApp">
      <div my-custom-directive></div>
</body>
```

The purpose of using `restrict` property while defining the directive is to guide its usage in HTML. It restricts the use as an Element or as an Attribute.

The allowed `restrict` values are as follows:

- E for Element name
- A for Attribute
- C for Class
- M for Comment

By default, the value is `EA`, meaning that both element names and attribute names can invoke the directive. E and A are the more frequently used types. C and M are used very rarely.

Completed code for the custom directive example is shown in Code Snippet 1.

Code Snippet 1:

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
<body ng-app="myApp">
<w3-custom-directive></w3-custom-directive>
<div w3-custom-directive></div>
<script>
  var app = angular.module("myApp", []);
  app.directive('w3CustomDirective', function() {
    return {
           restrict: 'A',
           template: '<h3>Hello AngularJS!!</h3><p>I was made inside a
Custom directive
                      </p>'
            };
      });
</script>
</body>
</html>
```

Save the file as ***CustomDirectiveDemo.html*** and copy it to ***htdocs*** folder under XAMPP. Refer to Figure 4.1.

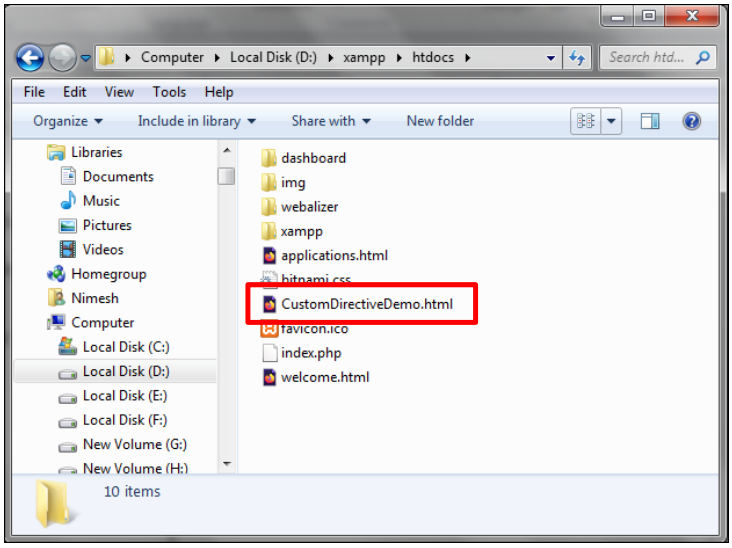


Figure 4.1: Copying the File to htdocs Folder

Then, open the XAMPP application, start the Apache server, and launch the application via the browser as shown in Figure 4.2.

The output of this application is shown in Figure 4.2.

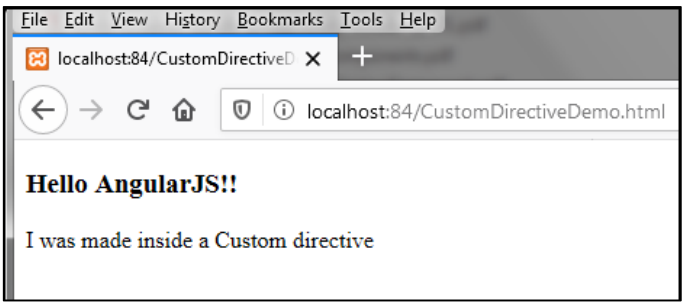


Figure 4.2: Custom Directive Demo - Output

4.3 Scopes

Consider that an AngularJS application consists of:

- View, which is the HTML markup
- Model, which is the data available for current view
- Controller, which is the JavaScript function that makes/changes/removes/controls the data

Here, **scope** is related to the **Model**.

In simple applications, the `$scope` object itself is treated as the model.

The scope is a JavaScript object with properties and methods, which are available for both view and controller. Scope is the bond between application controller and view.

Refer to Figure 4.3.

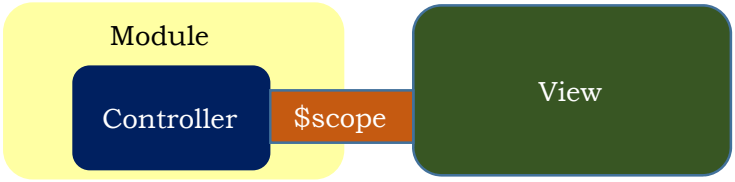


Figure 4.3: Scope

Scope gives the execution context for expressions used in the application. This is similar to which variables can be accessed in which block in general programming.

Scopes are arranged in hierarchical structure that mimics DOM arrangement of the application. They watch expressions and broadcast events.

Scopes enter the scene when we pass data to custom directives. There are three types of scopes:

1. Shared scope
2. Inherited scope
3. Isolated scope

4.3.1 Scope Hierarchies

Each application has only one root scope, but may have many child scopes. Therefore, it is important for us to know which scope we are dealing with, at any time. In simple applications, knowing our scope is not a necessity, but for larger applications, there can be sections in HTML DOM that can only access certain scopes. For example, let us consider `ng-repeat` directive. Here, each repetition will have access to the current repetition object only.

All applications have a `$rootScope` which is the scope created on the HTML element that contains the `ng-app` directive. The `$rootScope` is available in the entire application.

When a variable has the same name in both current scope and in `$rootScope`, the application gives priority to the variable in the current scope.

Let us see this with an example. The code is shown in Code Snippet 2. Save this code as ***rootScope.html***.

Code Snippet 2: `$rootScope` and `$scope` Example

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Scope Demo</title>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
</head>
<body ng-app="myApp">
<p>The rootScope's favorite color:</p>
<h1>{{color}}</h1>
<div ng-controller="myCtrl">
<p>The scope of the controller's favorite color:</p>
<h1>{{color}}</h1>
</div>
<p>The rootScope's favorite color is still:</p>
<h1>{{color}}</h1>
<script>
var app = angular.module('myApp', []);
app.run(function($rootScope) {
                  $rootScope.color = 'blue';
            });
app.controller('myCtrl', function($scope) {
            $scope.color = "red";
            });
</script>
<p>Notice that controller's color variable does not overwrite the
rootScope's color value.</p>
</body>
</html>
```

Similar to the earlier example, save this file, copy it to the ***htdocs*** folder, start the server and then, launch the application in the browser. When we run this application, we get the output as shown in Figure 4.4.

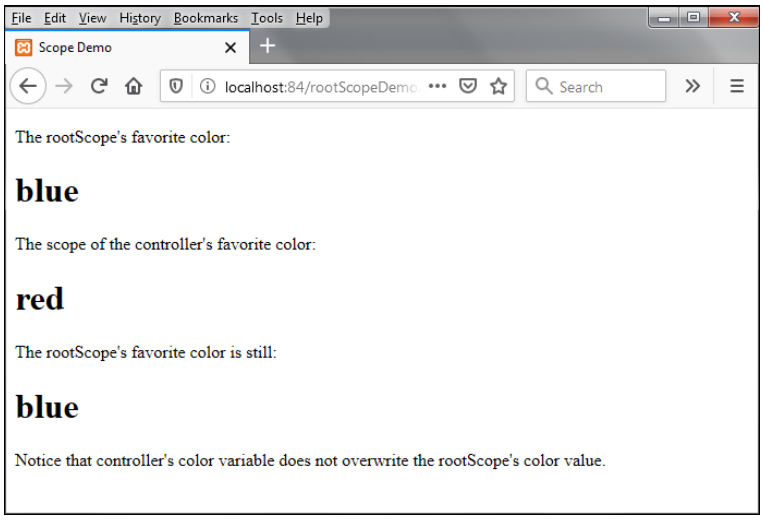


Figure 4.4: $rootScope and $scope Example - Output

Notice that we have used a variable named `color` in both the controller's scope and in the `$rootScope`. Inside the controller, the `color` variable has the value red since `$scope` operates here. Once we go out of the controller scope, `$rootScope` is the only available scope and so the variable `color` is evaluated to blue. Another point to note is that `$rootScope` is available throughout the application by inheritance.

4.4 Nested Controllers and Scopes

We have to be careful when working with nested controllers. We may land up with unexpected results while defining variables directly on `$scope` objects managed by controllers if we are not aware of how AngularJS works. Let us analyze this with an example. The HTML code for this example is shown in Code Snippet 3. Save the HTML code given in Code Snippet 3 as ***NestedControllerDemo.html*** and the JavaScript code in Code Snippet 4 as ***script.js***.

Code Snippet 3: Nested Controllers and Scopes Example - HTML Code

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Nested Scope Demo</title>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js"
>
</script>
<script src="script.js"></script>
</head>
<body ng-app="myApp">
<div>
<h3>Nested controllers with model variables defined directly on the
scopes</h3>
    (typing on an input field, with a data binding to the model, overrides
the same variable of a parent scope)
</div>
<div ng-controller="firstControllerScope">
<h3>First controller</h3>
<strong>First name:</strong> {{firstName}}<br />
<br />
<label>Set the first name: <input type="text" ng-
```

```
model="firstName"/></label><br />
<br />

<div ng-controller="secondControllerScope">
<h3>Second controller (inside First)</h3>
<strong>First name (from First):</strong> {{firstName}}<br />
<strong>Last name (new variable):</strong> {{lastName}}<br />
<strong>Full name:</strong> {{getFullName()}}<br />
<br />
<label>Set the first name: <input type="text" ng-
model="firstName"/></label><br />
<label>Set the last name: <input type="text" ng-
model="lastName"/></label><br />
<br />
<div ng-controller="thirdControllerScope">
<h3>Third controller (inside Second and First)</h3>
<strong>First name (from First):</strong> {{firstName}}<br />
<strong>Middle name (new variable):</strong> {{middleName}}<br />
<strong>Last name (from Second):</strong> {{$parentlastName}}<br />
<strong>Last name (redefined in Third):</strong> {{lastName}}<br />
<strong>Full name (redefined in Third):</strong> {{getFullName()}}<br />
<br />
<label>Set the first name: <input type="text" ng-
model="firstName"/></label><br />
<label>Set the middle name: <input type="text" ng-
model="middleName"/></label><br />
<label>Set the last name: <input type="text" ng-model="lastName"/></label>
</div>
</div>
</div>
</body>
</html>
```

The JavaScript code, ***script.js***, mapped to this HTML file is shown in Code Snippet 4.

Code Snippet 4: Nested Controllers and Scopes Example - JavaScript Code

```
var app = angular.module('myApp', [ ]);
app.controller('firstControllerScope', function($scope){
  // Initialize the model variables
  $scope.firstName = "Chris";
});
app.controller('secondControllerScope', function($scope){
  // Initialize the model variables
  $scope.lastName = "Hemsworth";
  // Define utility functions
  $scope.getFullName = function ()
  {
     return $scope.firstName + " " + $scope.lastName;
  };
});
app.controller('thirdControllerScope', function($scope){
  // Initialize the model variables
  $scope.middleName = "Whitelaw";
  $scope.lastName = "Pine";
  // Define utility functions
  $scope.getFullName = function () {
    return $scope.firstName + " " + $scope.middleName + " " +
$scope.lastName;
  };
});
```

The output of this application is shown in Figure 4.5.

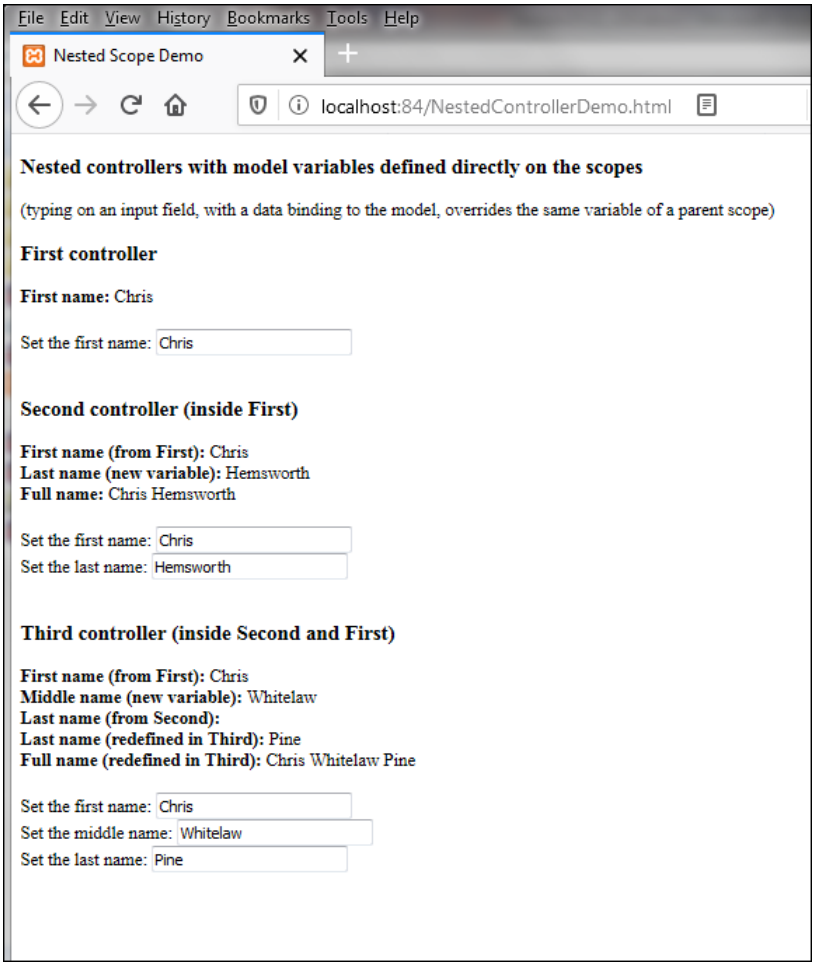


Figure 4.5: Nested Scopes and Controllers Example - Output

Here, in this example, we defined `firstControllerScope`, `secondControllerScope`, and `thirdControllerScope`. All these three controllers have variables defined directly on their respective `$scope` objects.

The `firstControllerScope` has the `firstName` variable and binds it to an input element in the HTML template. The `firstName` value changes when the user writes something in the input element.

The `secondControllerScope` is nested or placed inside `firstControllerScope`. The `secondControllerScope` has a new `lastName` variable in its own `$scope` object and the `getFullName` function here uses both a `firstName` and a `lastName` variable to return a result.

In the HTML, both `firstName` and `lastName` are added to some elements in the `div` where `secondControllerScope` is defined. When the page is first loaded, the value of `firstName` in the DOM tree belonging to `secondControllerScope` is taken from the scope of the immediately previous ancestor controller that has the variable defined in its `$scope`, in this case from `firstControllerScope`, because that variable is not defined by `secondControllerScope`.

When user enters the input element inside the `firstControllerScope` DOM tree, the value also changes in elements bound to `firstName` variable in the children controllers. However, when user types in the input element belonging to DOM tree of `secondControllerScope` and bound to `firstName`, a new variable `firstName` is created in `$scope` object. This `$scope` object is managed by `secondControllerScope`. From then onwards, there are two different instance of `firstName` belonging to scopes of `firstControllerScope` and `secondControllerScope`. On every occasion, the user types on the input of `firstControllerScope`, only its own copy of the variable changes.

A similar behavior is also observed in `thirdControllerScope`; here, we access the value of the `lastName` variable in the parent scope simply by using `$parent` notation.

We can use a way out to avoid likely conflicts and it is shown in `firstControllerObj`, `secondControllerObj`, and `thirdControllerObj`. We must define the variables inside distinct objects and define the objects on the scopes of the three controllers. By following this approach, there is no overriding of variables whenever user types in any of the input elements and we can avoid unexpected behaviors.

Code Snippet 5: Nested Scopes and Controllers - Another Example - HTML Code

Save this as ***NestedControllerDemo2.html***.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Nested Scope Demo</title>

<script src =
"https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>

<script src="script2.js"></script>
</head>
<body ng-app="myApp">

<div>
<h3>Nested controllers with model variables defined inside objects</h3>
    (typing on an input field, with a data binding to the model, acts on a
specific object without overriding variables)

<div ng-controller="firstControllerObj">
<h3>First controller</h3>
<strong>First name:</strong> {{firstModelObj.firstName}}<br />
<br />
<label>Set the first name: <input type="text" ng-model =
"firstModelObj.firstName"/>
</label>
<br />
<br />

<div ng-controller="secondControllerObj">
<h3>Second controller (inside First)</h3>
<strong>First name (from First):</strong> {{firstModelObj.firstName}}<br />
<strong>Last name (from Second):</strong> {{secondModelObj.lastName}}<br />
<strong>Full name:</strong> {{getFullName()}}<br />
<br />
<label>Set the first name: <input type="text" ng-model=
"firstModelObj.firstName"/>
</label>
<br />
<label>Set the last name: <input type="text" ng-
model="secondModelObj.lastName"/> </label><br />
<br />

<div ng-controller="thirdControllerObj">
<h3>Third controller (inside Second and First)</h3>
<strong>First name (from First):</strong> {{firstModelObj.firstName}}<br />
<strong>Middle name (from Third):</strong> {{thirdModelObj.middleName}}<br
/>
<strong>Last name (from Second):</strong> {{secondModelObj.lastName}}<br />
<strong>Last name (from Third):</strong> {{thirdModelObj.lastName}}<br />
<strong>Full name (redefined in Third):</strong> {{getFullName()}}<br />
<br />
```

```
<label>Set the first name: <input type="text" ng-model =
"firstModelObj.firstName"/> </label><br />
<label>Set the middle name: <input type="text" ng-model =
"thirdModelObj.middleName"/> </label><br />
<label>Set the last name: <input type="text" ng-model =
"thirdModelObj.lastName"/> </label>

</div>
</div>
</div>
```

The corresponding JavaScript code for this HTML is saved in ***script2.js***, shown in Code Snippet 6.

Code Snippet 6: Nested Scopes and Controllers - Another Example - JavaScript Code

```
var app = angular.module('myApp', [ ]);
app.controller('firstControllerObj', function($scope){
  // Initialize the model object
  $scope.firstModelObj = {
                          firstName: "Chris"
                          };
});
app.controller('secondControllerObj', function($scope){
  // Initialize the model object
  $scope.secondModelObj = {
                            lastName: "Hemsworth"
                           };
  // Define utility functions
  $scope.getFullName = function ()  {
                                      return $scope.firstModelObj.firstName
                                      + " " +
                                      $scope.secondModelObj.lastName;
                                    };
});
app.controller('thirdControllerObj', function($scope){
  // Initialize the model object
  $scope.thirdModelObj = {
                            middleName: "Whitelaw",
                            lastName: "Pine"
                          };
  // Define utility functions
  $scope.getFullName = function ()  {
                              return $scope.firstModelObj.firstName + " " +
                              $scope.thirdModelObj.middleName + " " +
                              $scope.thirdModelObj.lastName;
  };
});
```

The output of this is seen in Figure 4.5.

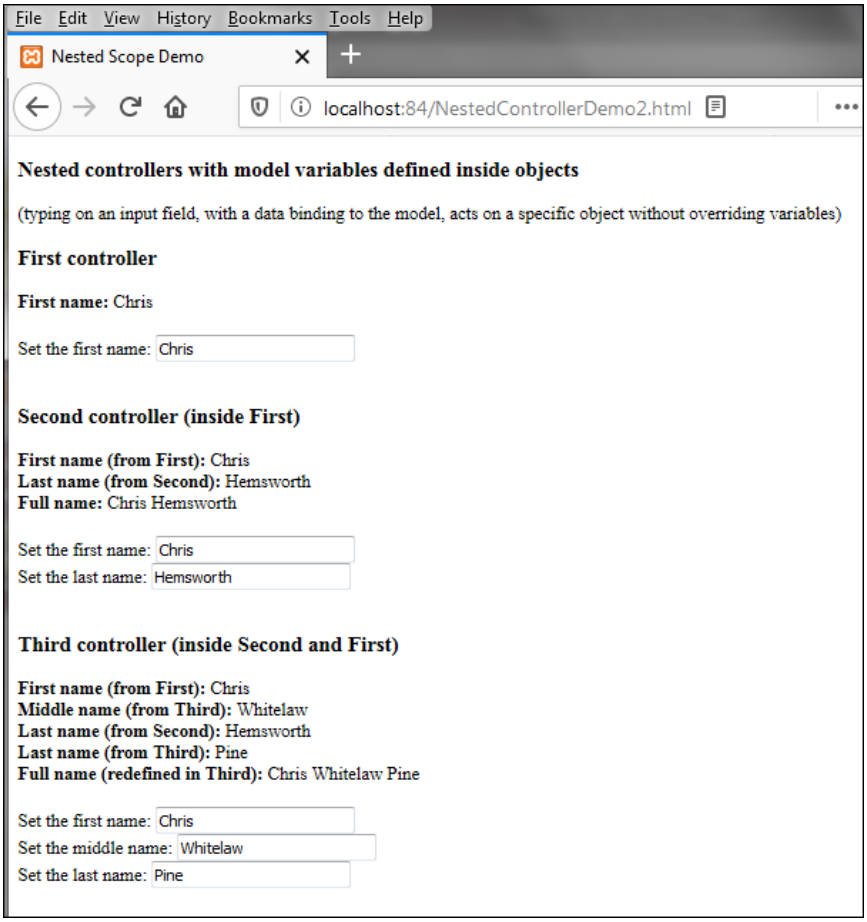


Figure 4.5: Nested Scopes and Controllers Example – Output

4.5 Services

In AngularJS, 'services' refer to simple objects that perform some sort of work. Services are JavaScript functions and are responsible to do a specific task only. This makes them an individual entity that is easily maintainable and testable. Controllers and filters can call them as on requirement basis. Services are injected using the dependency injection mechanism of AngularJS.

AngularJS provides many built-in services such as, `$http`, `$route`, `$window`, `$location`, and so on. Each service is responsible for a specific task. For example, `$http` is used to make Ajax call to get the server data. `$route` is used to define the routing information.

There are around 30 built-in services.

The built-in services provided by AngularJS are always prefixed with $.

- **`$http`:**

We use `$http` service for reading data from remote servers. It is a frequently used service. Let us see this through an example. The code is shown in Code Snippet 7. Save it as ***HTTPDemo.html***.

Code Snippet 7:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>$http service demo</title>
   <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
```

```
   </script>
</head>
<body>
      <div ng-app="myApp" ng-controller="myCtrl">
            <p>Today's welcome message is:</p>
            <h1>{{myWelcome}}</h1>
      </div>
      <p>The $http service requests a page on the server, and the response
is set as
         the value of the "myWelcome" variable.</p>
      <script>
            var app = angular.module('myApp', []);
            app.controller('myCtrl', function($scope, $http) {
                  $http.get("welcome.html")
                  .then(function(response) {
                        $scope.myWelcome = response.data;
                  });
            });
      </script>
</body>
</html>
```

Code snippet 7 uses `$http` service to get a file named ***welcome.html*** from the server using get method of `$http`. Before using `$http`, we passed this service as a dependency in our controller definition. This part of the code is highlighted in yellow for your reference. After successful receipt from the server, the response is assigned to `myWelcome` variable. After that, it becomes accessible from the view and is displayed. The output of this application is shown in Figure 4.7. Notice that the application is run via the localhost. This is because we are using HTTP and applications run via the local computer will not support the HTTP request. It is assumed that a file named ***welcome.html*** is present under htdocs along with the HTTPDemo application.

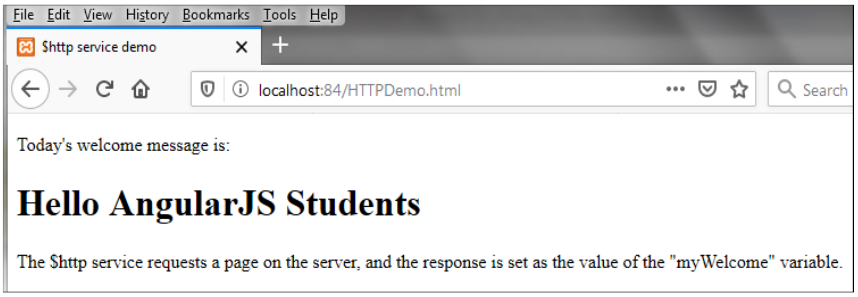


Figure 4.7: $http service – Example – Output

Here, we have used the `.get()` method of the `$http` service to get data from server. It also provides several other methods such as `.post()`, `.put()`, and `.delete()`.

- **$location:**

The `$location` service has methods which return information about the location of the current Web page. It also keeps itself and the URL in synchronization. Any modification made to `$location` is passed to the URL and whenever the URL changes (such as when a new route is loaded) the `$location` service updates itself. Another use of `$location` is to update the browser's URL to navigate to a different route or to watch for changes in `$location` and take appropriate actions in your application. Code Snippet 8 demonstrates usage of this service with an example. Save this code as ***LocationDemo.html***.

Code Snippet 8:

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
    <h4>$location Service Example</h4>
```

```
    <div ng-app="app" ng-controller="LocController">
        <div>
            Current absolute URL: {{currentURL}}
        </div>
        <br />
    </div>
<script>
          var app = angular.module("app", []);
    app.controller("LocController", function ($scope, $location) {
        $scope.currentURL = $location.absUrl();
    });
      </script>
</body>
</html>
```

In this code, the current location of the application is passed along with the scope to the controller function. This function then, retrieves the URL from the location service using `absUrl()` method and assigns it to the `currentURL` variable of scope object. The output will be as shown in Figure 4.8.

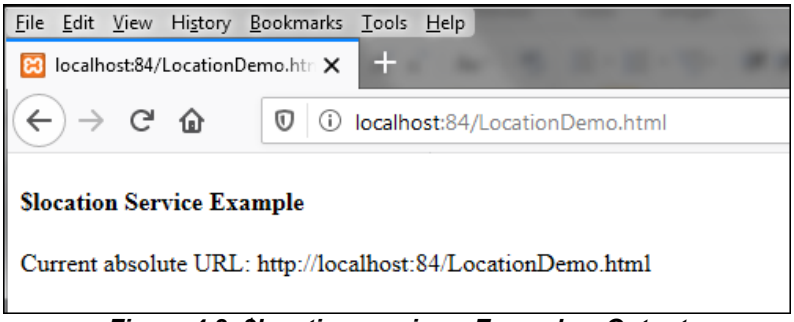


Figure 4.8: $location service – Example – Output

Quick Test 4.1

1. We must use the exact name in view as given to it in the controller definition.
    a. True
    b. False

2. If a variable has the same name in both current scope and in `$rootScope`, which variable will be used by the application?
    a. Current scope
    b. Root scope

4.6 Summary

- Developers can create new directives using `.directive` method.
- Allowed restrict values for a custom directive are as follows:
    - E for Element name
    - A for Attribute
    - C for Class
    - M for Comment
- The custom directive is used in the view by separating camel case name with a hyphen/dash.
- The $rootScope is available in the entire application.
- In a scenario where a variable has the same name in both current scope and in $rootScope, the application uses the variable defined in the current scope.
- Services are JavaScript functions and are responsible to perform only a specific task.
- Services are injected using the dependency injection mechanism of AngularJS.

4.7 Exercise

1. Which type of case do we use for naming a custom directive?
   a) Pascal case
   b) Uppercase
   c) Sentence case
   d) Camel case

2. What are the allowed values for 'restrict' property, while we define a custom directive?
   a) E, A, C, M
   b) A, B, C, D
   c) W, X, Y, Z
   d) F, R, O, G

3. What are the default values if we do not give any value to restrict?
   a) FG
   b) AB
   c) EA
   d) WX

4. _____ is the bond between the application controller and the view.

   a) Scope
   b) Model
   c) Connect
   d) Route

5. Which symbol is used to prefix a built-in service?

   a) #
   b) $
   c) |
   d) %

4.8 Do It Yourself

1. Assume that you are a budding entrepreneur, creating a new application similar to LinkedIn.

   Users of your application have said that they would like to be able to choose a profile picture just by entering a URL.

   Create an AngularJS application that accepts user name and allows users to type in a profile picture URL and have that picture appear immediately in the browser, next to their name.

   Hint: Create a field ProfilePhoto and bind it to a model with something such as "`profilepicUrl`".
   Then, use the `profilepicUrl` as the `src` attribute of an `img` tag:
   `<img src="" />`

   Create an AngularJS application that uses $`http` service to get the contents of a file named ***demo.txt*** from the server and display it on the screen. Assume that demo.txt is present in the htdocs path.

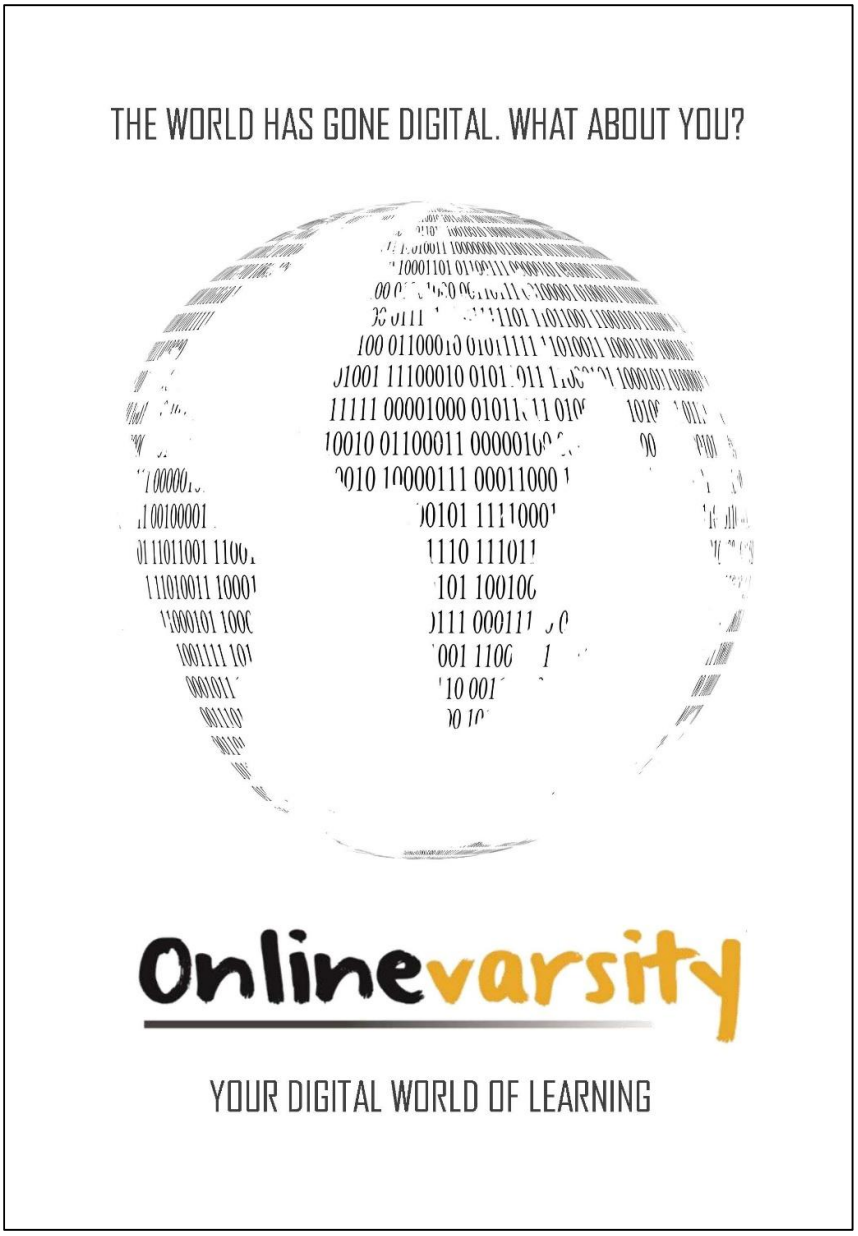
Answers to Exercise

1. Camel case
2. E, A, C, M
3. EA
4. Scope
5. $

Answers to Quick Test

Quick Test 4.1

1. False
2. Current scope



Session 5

Form Validation and AngularJS Animations

In this session, students will learn to:

- Explain form validation with AngularJS
- Identify the different Form states and Input states
- Describe how to work with AngularJS Animations
- Identify the use of CSS classes for AngularJS Animations

5.1 Forms Validation

Forms are a major way in which users communicate with applications we develop. They are an important mechanism of modern Websites and applications. Through forms, users tell us who they are, sign up for services, and purchase products.

We often need to gather some or the other information about users or their choices at some point and most of the time this is done through a form. We would generally want to validate the data that users enter into our forms.

While it is true that we perform validation at the server end too, we still have to perform first-line-of-defense validation in the Web browser.

In general, we will do a lot of the validation up front, before even considering sending it to the server. If the data does not pass validation, it will display an error message to the user.

Through this, we can reduce the load placed on our Web servers, conserve bandwidth, and provide better user experience. AngularJS continuously oversees the status of the form and input fields such as `input`, `textarea`, or `select` and help us advise the user about the current state. AngularJS also carries information about whether the fields have been entered, or modified, or not.

Additionally, AngularJS input elements within forms enable us to achieve two-way data binding as we have seen already with `ng-model` directive.

Let us now see some of the features available to us in AngularJS regarding form validation.

Required:

We use the HTML5 attribute `required` to specify that the input field must be filled out. Code Snippet 1 uses the `required` attribute.

Code Snippet 1:

```
<!DOCTYPE html>
<html>
<title>Forms Validation</title>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js">
</script>
<body ng-app="">
    <p>Try writing in the input field:</p>
    <form name="myForm">
       <input type="text" name="myInput" ng-model="myInput" required>
    </form>
    <p>The input's valid state is:</p>
    <h1>{{myForm.myInput.$valid}}</h1>
</body>
</html>
```

When we run this code, we get output as shown in Figure 5.1.

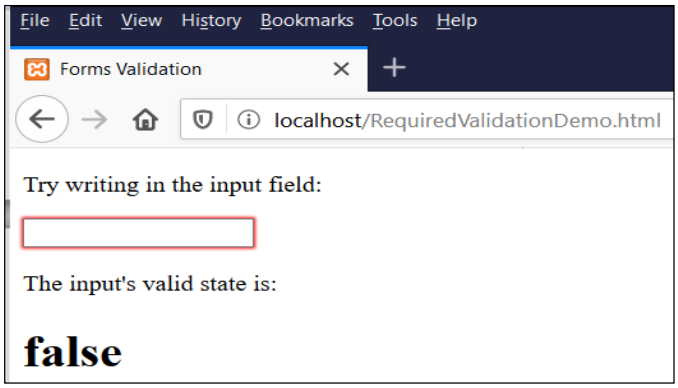


Figure 5.1: Required Example Initial Output

Since no input is given in the input field, the state is declared as 'false'.

We collect the status of the input field with the expression `{{myForm.myInput.$valid}}`.

As soon as we enter some value in the input field, status changes to 'true' as shown in Figure 5.2.

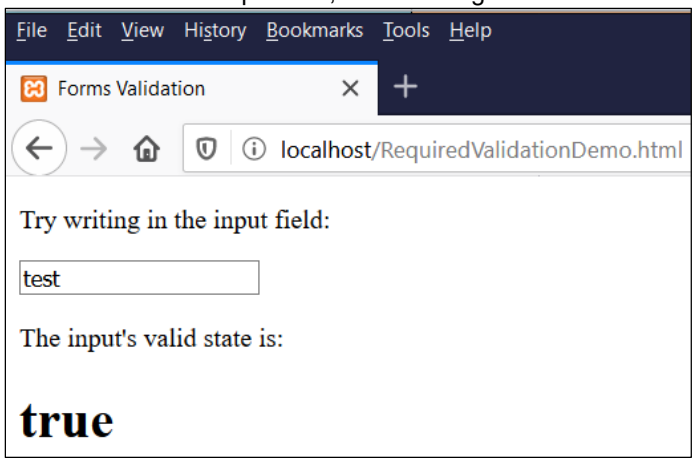


Figure 5.2: Required Example After Adding Input

E-mail:

We use the HTML5 type e-mail to specify that the value must be an e-mail. In this case, it must also be a value that is a properly formatted e-mail address.

An example code that validates an e-mail input is shown in Code Snippet 2.

Code Snippet 2:

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
<body ng-app="">
<p>Write an E-mail address in the input field:</p>
<form name="myForm" >
  Email: <input type="email" name="input" ng-model="emailid" ngMinlength=5
required>
  <br/>
  <br/>
  <span class="error" ng-show="myForm.input.$error.required">
    Required!</span><br/>
  <span class="error" ng-show="myForm.input.$error.email">
    Not valid email!</span>
  <br>
  Email Id = {{emailid}}<br/><br/>
  myForm.input.$valid = {{myForm.input.$valid}}<br/><br/>
```

```
  myForm.input.$error = {{myForm.input.$error}}<br/><br/>
  myForm.$error.email = {{!!myForm.$error.email}}<br/><br/>
</form>
</body>
</html>
```

When we run this application, the initial screen is shown in Figure 5.3.

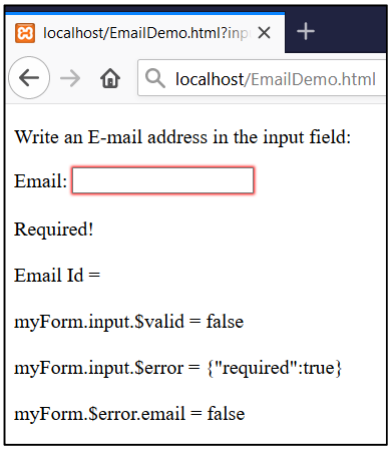


Figure 5.3: E-mail Example Initial Screen

As we start entering e-mail, it checks for standard @ symbol and string after it. Since they are missing, **Not valid email!** is displayed as shown in Figure 5.4.

Figure 5.4: E-mail Example After Partial Input

In Figure 5.5, a helpful prompt is displayed asking user to enter an e-mail address.

Figure 5.5: E-mail Example After @ Input

Finally, when we give a correct input, it is validated as true as shown in Figure 5.6.

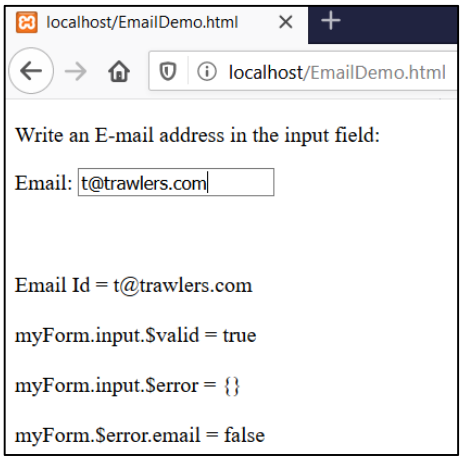


Figure 5.6: E-mail Example After Validated Input

Note: E-mail validation is done as per AngularJS built-in validation mechanism. Hence, it may not be foolproof. If you want to restrict and validate in a better way, you can choose to use regular expressions with ng-pattern.

Form states:

Form is simply a group of controls. We know that a control is a way for users to input data to an application, which allows the user to control the application. Examples of individual controls include `button`, `input`, `textarea`, `select`, and so on.

Therefore, form is just a collection of related individual controls grouped together. For example, login form, having a group of controls which allows our users to input their credential to enter the application.

Let us see the difference between plain HTML Form and AngularJS Form. AngularJS Form has some additional capabilities than plain HTML forms. It gives developers more control on how to communicate with the form. Now, developers can know each form's state since AngularJS gives state to each of them, such as pristine, dirty, valid, and invalid.

When we create a form, AngularJS creates an instance of `FormController`.

Following are the `FormController` methods and properties:

Form properties represent its states. The properties are `$pristine`, `$dirty`, `$valid`, `$invalid`, and `$submitted`.

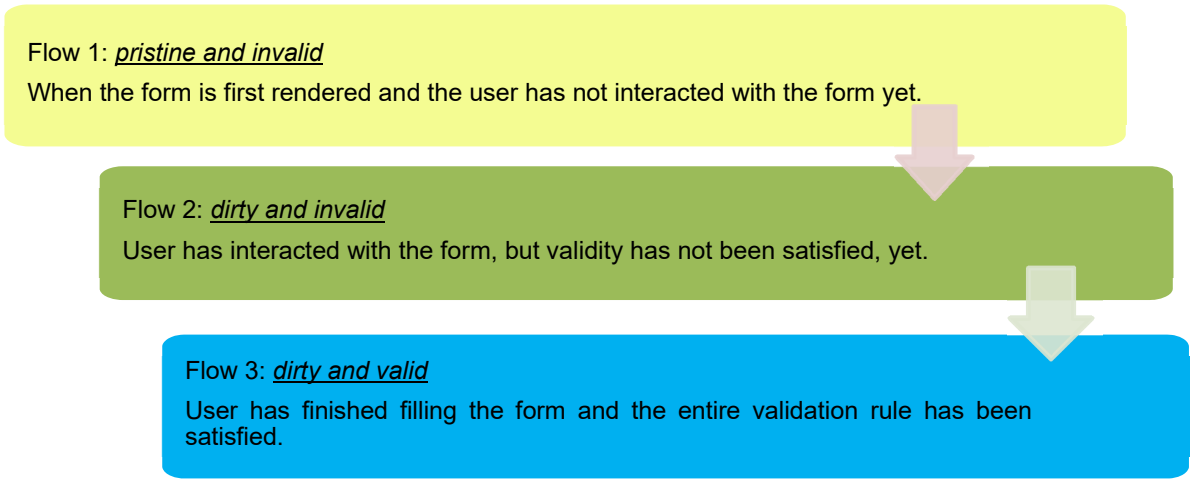
- `$pristine`: No fields have been modified yet
- `$dirty`: One or more have been modified
- `$invalid`: The form content is not valid
- `$valid`: The form content is valid
- `$submitted`: The form is submitted

For an AngularJS form, these properties will be either true or false depending on their states.

Flow of the form states:

We need to know the flow of the form states in order to use AngularJS form properly.

Following flow gives us a picture of the form state from the very first time the form is rendered until the user has finished filling the form:



Input states:

Just as the form has different states, the individual controls also have similar states.

- `$untouched`: The field has not been touched yet
- `$touched`: The field has been touched
- `$pristine`: The field has not been modified yet
- `$dirty`: The field has been modified
- `$invalid`: The field content is not valid
- `$valid`: The field content is valid

We use these states to show meaningful messages to the user. For example, if a field is required, and the user leaves it blank, we give the user a warning about this.

CSS Classes:

AngularJS gives us in-built CSS classes to allow styling of form and input controls based on the state of forms and their fields. Table 5.1 lists these fields.

CSS Class	Description
`ng-valid`	Is set if the input field is valid without errors
`ng-invalid`	Is set if the input does not pass validations
`ng-pristine`	Is set if a user has not interacted with the control yet
`ng-dirty`	Is set if the value of the form field has been changed
`ng-touched`	Is set if a user tabbed out from the input control
`ng-untouched`	Is set if a user has not tabbed out from the input control
`ng-submitted`	Is set if the form has been submitted

Table 5.1: CSS Class and Description

However, note that we have to provide implementation of these CSS classes and include in our CSS file. AngularJS automatically includes these classes based on the current state of input controls.

Code Snippet 3 demonstrates `ng-pristine`, `ng-touched`, `ng-valid`, and `ng-invalid` classes to display validity of each form control.

Code Snippet 3:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-">
<title>Forms Validation CSS class</title>
```

```
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js"
></script>
<style>
input.ng-pristine { background-color:yellow;}
input.ng-touched.ng-invalid { background-color:red;}
input.ng-touched.ng-valid {background-color:green;}
</style>
</head>
<body ng-app>
<form name="studentForm" novalidate class="student-form">
<label for="firstName">First Name: </label><br />
<input type="text" name="firstName" ng-model="firstName" ng-required
="true" />
<br /><br />
<label for="lastName">Last Name</label><br />
<input type="text" name="lastName" ng-model="lastName" ng-required="true"
/>
<br /><br />
<label for="dob">E-mail</label><br />
<input type="email" id="email" ng-model="email" name="email" ng-required
="true"/>
<br /><br />
<input type="submit" value="Submit" /><br />
<p>Initially the input fields are yellow They become green when we give
valid data in it. They turn red if data is invalid, say an empty string
</p>
</form>
</body>
</html>
```

When we run this application, we get the initial screen as shown in Figure 5.7.

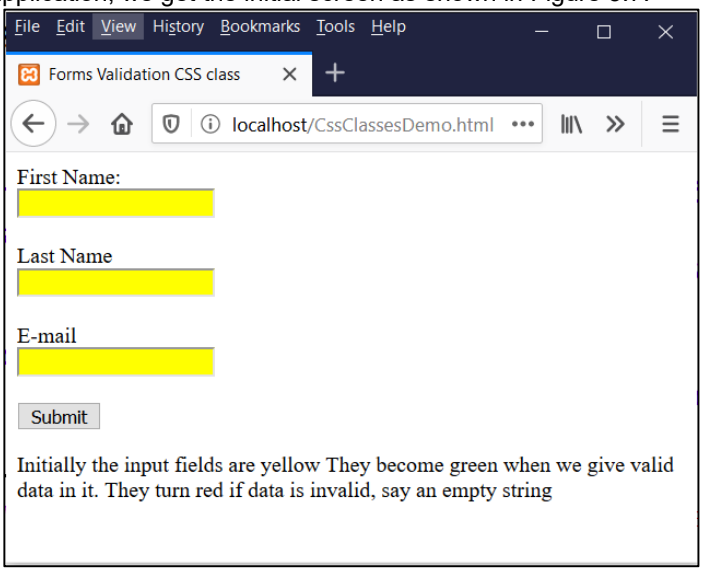


Figure 5.7: States and CSS Classes - Initial Screen

The input fields are yellow due to `ng-pristine` class definition in our code. The input fields are in pristine state, so they get this class applied.

When we give valid inputs, they turn green, as we have defined in `ng-valid` class, as shown Figure 5.8.

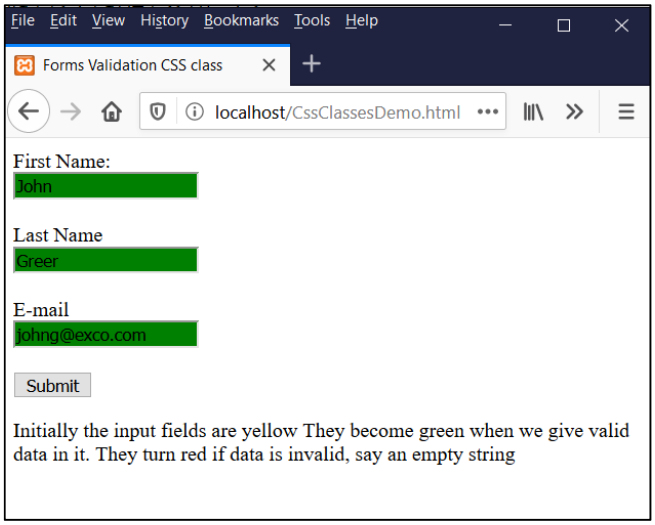


Figure 5.8: States and CSS Classes- with Valid Inputs

If we give invalid inputs, they turn red as we have defined in `ng-invalid` class, as shown Figure 5.9.

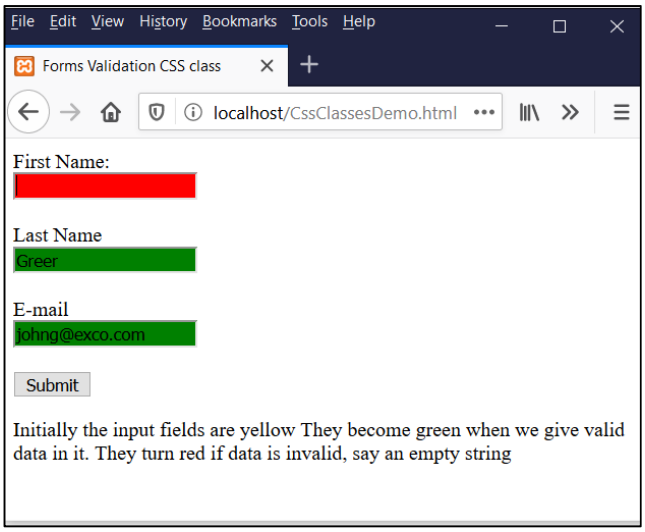


Figure 5.9: States and CSS Classes - with Invalid Inputs

You can also run this application with various inputs and check its behavior with different states.

5.2 AngularJS Animations

An animation is the transformation of an HTML element that gives us an illusion of motion. Animating elements in our application or pages add to the fun and increases the user experience. They enhance the user interface by making it smoother and more attractive. However, animations should be done in a subtle way to prop up our applications. Slowly fading in messages or views against showing them abruptly gives a pleasant user experience.

Let us look at how to achieve such things.

`$ngAnimate:`

To implement animations in AngularJS, we need to add the ***angular-animate.js*** library after adding ***angular.min.js*** library. Then, we inject the `ngAnimate` dependency into our application.

We can either download ***angular-animate.js*** library from AngularJS site and use a local copy or link it with a CDN service similar to ***angular.min.js*** library.

The `ngAnimate` module adds and removes classes.

Let us look at an example that uses AngularJS animations with `ng-show` directive.

The code for this example is shown in Code Snippet 4.

Code Snippet 4:

```
<!DOCTYPE html>
<html lang="en">
<head>
      <meta charset="UTF-8">
      <title>AngularJS Animations</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
    </script>
      <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular-
    animate.js">
    </script>
      <style>
            div {
            transition: all linear 1s;
            background-color: cyan;
            height: 100px;
            width: 100%;
            position: relative;
            top: 0;
            left: 0;
            }
            .ng-hide {
            height: 0;
            width: 0;
            background-color: transparent;
            top:-200px;
            left: 200px;
            }
      </style>
</head>
<body ng-app="ngAnimate" >
<h1>Hide the DIV: <input type="checkbox" ng-model="myCheck"></h1>
<div ng-hide="myCheck"></div>
</body>
</html>
```

We have included a local copy of ***angular-animate.js*** library after ***angular.min.js*** library inclusion. We have added the `ngAnimate` to the `body` element that enables the animations.

We have a `div` element which has an attribute `ng-hide` which is set to the expression `myCheck`.

`myCheck` is a model variable that gets its value from the check box. When we run this application, the initial screen looks as shown in Figure 5.10.

Figure 5.10: ngAnimate Example Initial Screen

Whenever we check the check box, `myCheck` evaluates to 'true' and the animation takes place. The `div` fades away over the duration of one minute. The screen after completion of animation is shown in Figure 5.11. If we uncheck the check box, the value becomes false and the `div` element reappears.

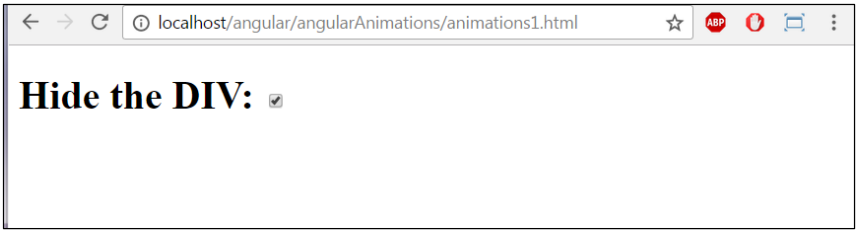


Figure 5.11: ngAnimate Example - After Checking and Completion of Animation

The `ngAnimate` module does not animate our HTML elements directly. When `ngAnimate` notices some events, such as hide or show of an HTML element, the element gets some pre-defined classes that are used to make animations.

Directives in AngularJS that add/remove classes include:

- `ng-show`
- `ng-hide`
- `ng-class`
- `ng-view`
- `ng-include`
- `ng-repeat`
- `ng-if`
- `ng-switch`

CSS Animations:

We can also use CSS transitions or CSS animations to animate HTML elements.

A CSS transition allows us to change CSS property values smoothly, from one value to another, over a given duration. For example, we can animate the height of an element as shown in Code Snippet 5.

Code Snippet 5:

```
<!DOCTYPE html>
<html lang="en">
<head>
      <meta charset="UTF-8">
      <title>AngularJS Animations</title>
      <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
    </script>
      <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular-
    animate.js">
    </script>
      <style>
            div { transition: all linear 1s;
                  background-color: cyan;
                  height: 100px;
            }
            .ng-hide {
                  height: 0px;
            }
      </style>
</head>
<body ng-app="myApp">
      <h1>Animate the DIV: <input type="checkbox" ng-model="myCheck"></h1>
      <div ng-hide="myCheck"></div>
      <script>
```

```
    var app = angular.module('myApp', ['ngAnimate']);
     </script>
</body>
</html>
```

In this example, we have linked to CDN service for angular.min.js and angular-animate.js libraries. We have also added `ngAnimate` as a dependency to our `myApp` module in the script. When this application is run, the initial screen looks as shown in Figure 5.12.

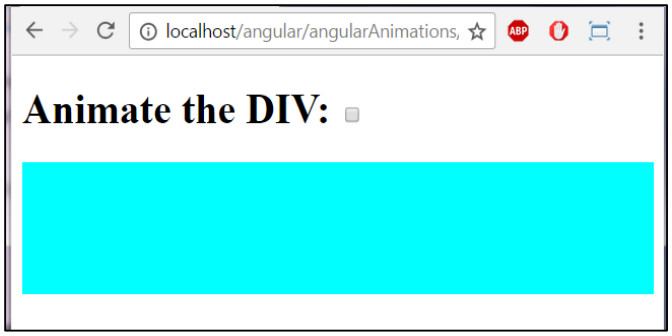


Figure 5.12: CSS Animations Initial Screen

When we interact with this application by checking the check box, the height of the `div` element reduces to 0 px, animatedly over a period of ten seconds.

You can run this code and experiment with it. The final screen after we check and complete the animation is shown in Figure 5.13.

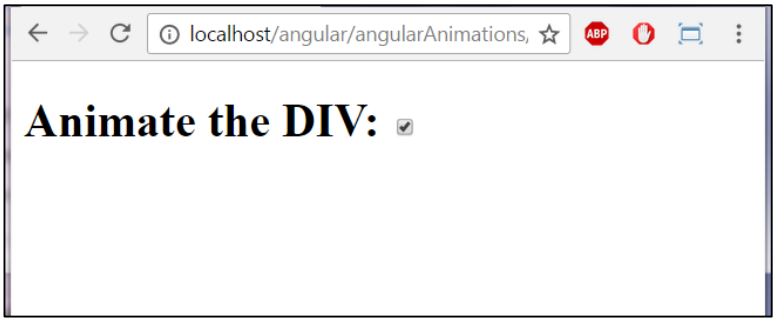


Figure 5.13: CSS Animations Final Screen

Quick Test 5.1

1. The ngAnimate module comes as part of angular.js library.
   a. True
   b. False

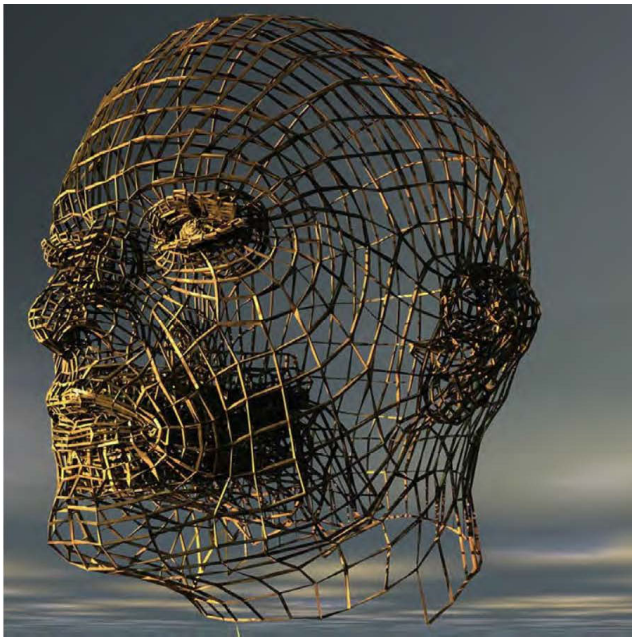
2. Form validation makes the server side validation not necessary.
   a. True
   b. False

5.3 Summary

- Forms are the major way users communicate with the applications we develop.
- First-line-of-defense validation is typically performed in the Web browser.
- Form validations reduce the load on our Web servers, conserve bandwidth, and provide better user experience.
- AngularJS monitors the state of the form and input fields and lets us notify the user about the current state.
- Animating the elements in our application adds to the fun and increases the user experience.
- The angular-animate.js library must be added in addition to the core angular.js library, to implement animations in AngularJS.
- The `ngAnimate` module adds and removes classes.

Onlinevarsity

TECHNO-WISE
THE TECHNOLOGIES OF TOMORROW



5.4 Exercise

1. What type of validation do we perform in the Web browser?
   a) Back-up defense
   b) Final defense
   c) First-line-of-defense
   d) Comprehensive defense

2. AngularJS continuously monitors the state of the form and input fields and gives state to each of them, such as _____ _____ _____ and _____.
   a) Pristine, dirty, valid, and invalid
   b) Virgin, unclean, clean, and passed
   c) Pure, partial, dirty, and rejected
   d) Incomplete, semi complete, complete, and sent

3. A form is a collection of related individual _____ grouped together.
   a) Variables
   b) Items
   c) Controls
   d) Values

4. Which library do we include to implement animations in AngularJS?
   a) angular-animate.js
   b) angular-transform.js
   c) angular-translate.js
   d) angular-vibrate.js

5. Which dependency do we need to inject for making animations?
   a) `ngRoute`
   b) `ngTransform`
   c) `ngMotion`
   d) `ngAnimate`

5.5 Do It Yourself

1. Peter is setting up a company JobSeekers for aspiring job seekers. You have to help him by creating a page that contains a Registration form. You do this by creating an AngularJS application. It should accept the user's name, age, qualifications, and e-mail. Use appropriate validation where required. Display the entered data with the user's name in uppercase.

2. Jenny has started her own cupcake baking business. She has sought out your aid in creating an Order application for her. Create an AngularJS application that accepts orders via form the following:
    - Customer Name
    - Customer Address
    - Order Date
    - Email Id
    - Number of Cupcakes

    Assume that the price of each cupcake has been fixed as five dollars. Delivery costs are extra - 5% of the total bill amount.

    Display the input in proper formatted manner and calculate and display Payable amount which will be bill amount (number of cupcakes * 5) plus delivery costs.

Answers to Exercise

1. First-line-of-defense
2. Pristine, dirty, valid, and invalid
3. Controls
4. angular-animate.js
5. ngAnimate

Answers to Quick Test

Quick Test 5.1

1. False
2. False

Session 6

Services and Communication in AngularJS

In this session, students will learn to:

- Define and explain Representational State Transfer (REST) Application Programming Interface (API)
- Describe RESTful services in AngularJS
- Explain client-server communication
- List the steps to access server using various server resources
- Explain error handling in AngularJS client-server communication

This session introduces the basic concepts of REST along with its principles and services in AngularJS. The process of communication between a client and a server for accessing different resources is explained in simple terms. The session also explains how to handle errors and exceptions that occur during client-server communication.

6.1 Introduction to REST API

REST refers to a stateless architecture that is used for creating networked applications. It was introduced in the year 2000 by Roy Fielding, it is sometimes spelled as 'REST'. It uses a cacheable client-server protocol for communication.

Almost in all cases, this architecture uses Hypertext Transfer Protocol (HTTP) to facilitate communication between devices.

The underlying notion behind the introduction of REST is to prevent the use of a complex communication mechanism such as Simple Object Access Protocol (SOAP), Remote Procedure Call (RPC), or Common Object Request Broker Architecture (CORBA). Hence, REST is a lightweight alternative to Web services. From different perspectives, the HTTP-based World Wide Web (WWW) itself is considered as a REST-based architecture. REST is fully optimized for the Internet due to which it is gaining more popularity over HTTP.

REST relies on HTTP for four operations namely, Create, Read, Update, and Delete (CRUD). RESTful applications (applications that are based on REST) send HTTP requests to read, post, or delete data.

While there are REST-based programming frameworks, using REST is so simple that you can customize it with standard library features and in languages, such as C#, Java, and Perl.

Yet, although simple, REST architecture is comprehensive. There is virtually nothing that a developer cannot do with it, which otherwise is done using Web services.

REST has no World Wide Web Consortium (W3C) recommendation, as it is not a standard. Figure 6.1 depicts REST.

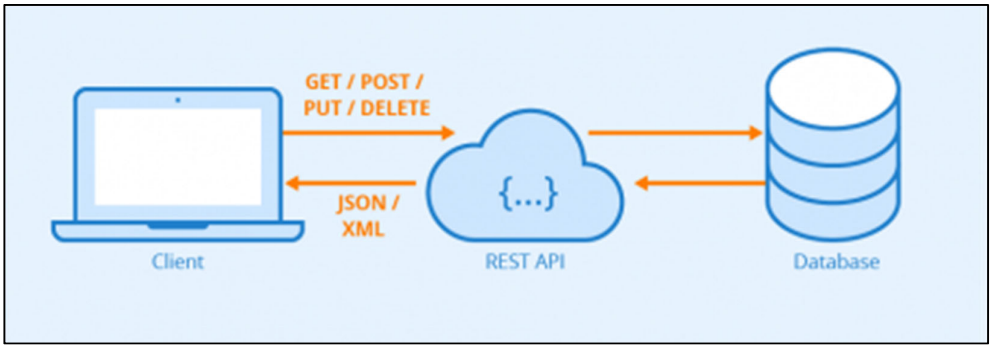


Figure 6.1: REST

Image Courtesy: https://www.seobility.net/en/wiki/REST_API

6.1.1 REST Principles

REST is referred to as an architectural style in which a set of constraints is applied to create a desired architecture for common uses. These constraints are its principles or characteristics, which differentiate REST from other architectures.

There are six such principles or constraints, which are as follows:

Client-server Communication	It is based on the principle of separation of concerns, which enables components of an application to be developed independently of one another. Thus, a client and a server are two separate components. A RESTful application should ensure such a communication by acting as a server.
Statelessness	No client-based information should be stored on the server. In other words, each client request should include all necessary information for processing the transaction successfully on the targeted server. This allows the server to understand client request without storing client's or session's state and ensures better scalability. All session state data should return to client at the end of each request. Statelessness imposes substantial restrictions on type of communication between servers and their clients for accomplishing its design goals.
Cacheability	A client is free to cache responses that can be classified as cacheable or not-cacheable. Any cacheable date is reused as a response to the same successive request later.
Uniform Interface	A single uniform interface must facilitate interaction among all components. This makes it simple to interact with various services. It also ensures that the changes during implementation are made without affecting the interface or component interaction. This also means that it is impossible to modify the interface, even if it is the only way to optimize a specific service.
Layered System	An intermediary, such as a proxy, can function between a client and a server to interrupt the communication between the two for a specific purpose such as security, load balancing, or caching. The client, at any time, is unaware of whether it is communicating with a server or an intermediary.
Code on Demand	This is a discretionary constraint in which a server momentarily extends the client's functionality by allowing it to download programs. These programs then, execute on the client. For example, a client can run a JavaScript code to interact with another service that runs on it.

6.1.2 REST versus SOAP

Many developers wonder why REST is preferable over SOAP. This is where they need to know the differences. Firstly, it is impractical to compare SOAP and REST directly. This is because while SOAP is a protocol, REST is a style of architecture. So, it is not necessary that an HTTP API that does not use SOAP is by default REST.

However, there are certain points on which they can be evaluated against one another. Table 6.1 elaborates on these points.

Point of Distinction	SOAP	REST
Degree of Coupling between Client and Server Implementations	Is tightly coupled to the server, just as a custom desktop application. A rigid contract exists between a client and a server. If any of them changes, everything breaks.	Is loosely coupled, just as a browser. A REST client is a generic client that uses standardized methods and a protocol to which an application adapts accordingly. No additional methods are defined to violate protocol standards. Changes are handled more nicely.
Orientation	Is object-oriented.	Is resource-oriented.
Size	Is heavy due to additional XML.	Is lightweight.
State	Is stateful.	Is stateless.
Standard	Is standard specific.	Is not standard specific.
Speed	Is slow due to strict specifications and requirements for more bandwidth and resources.	Is fast, as there are no strict specifications and that it consumes less resources and bandwidth.
Protocol Dependency	Is independent of transport protocol in use. It can use any transport protocol.	Is dependent on HTTP, although is not coupled to it. However, it can use any protocol that has a standard Uniform Resource Identifier (URI) scheme.
Communication	Uses only eXtensible Markup Language (XML).	Uses self-descriptive messages that control interaction and represented via media types such as XML, plain text, and JavaScript Object Notation (JSON).
Implementation	Is not so easier.	Is easier.
Client	Needs full knowledge on what it will be using, prior to the interaction.	Has no knowledge of the API, except for the entry point and media type (data format).

Table 6.1: Differences between SOAP and REST

6.1.3 Resources in RESTful Web Services

A RESTful Web service is a Web service that implements a REST API. It uses HTTP methods to implement the architecture. It also defines a URI, which refers to a service that offers HTTP methods as well as a format for representing a resource.

In a RESTful API, the basic concept is of the resource. It handles every content as a resource. It can be a document, an image, a file, a Web page, or a collection of users. A resource is of a specific type, contains data, and has a set of methods for operating on it and relationships with other resources.

A resource in REST is equivalent to an object in an object-oriented programming language. However, the main difference exists in terms of number of methods. A resource has only a few standard methods, which corresponds to standard HTTP methods; while an object usually contains several methods. Following are the commonly used HTTP methods in the REST architecture:

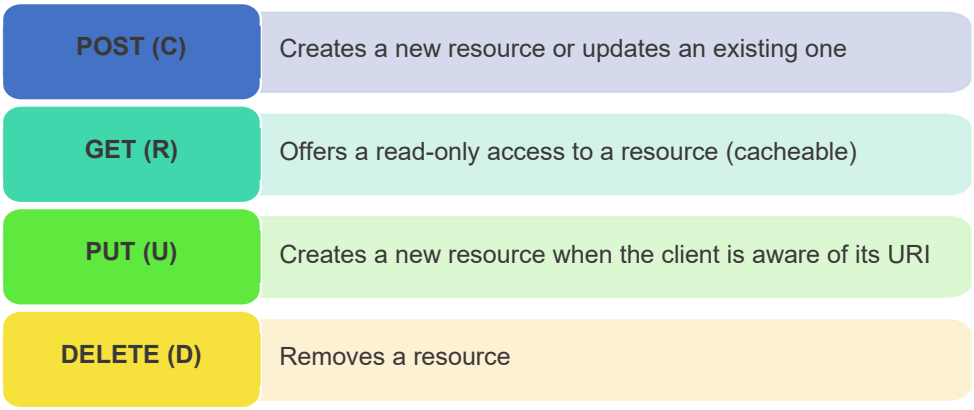


Figure 6.2 shows the functionality of HTTP methods for the **tasks** resource.

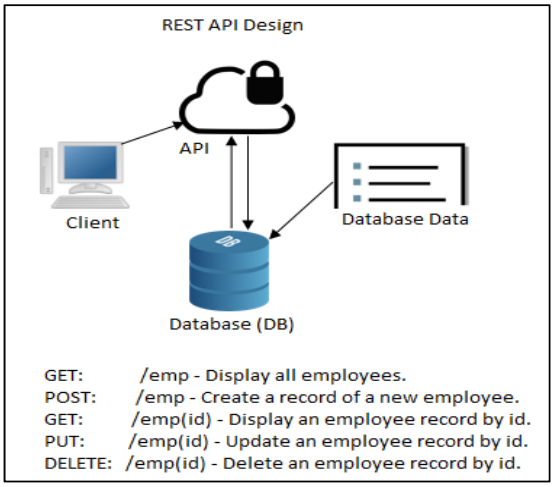


Figure 6.2: HTTP Methods

It is possible to group resources into collections. Each collection is also a resource. A collection is unordered and uniform, as it possesses only a single type of resource. It is not necessary for a resource to be a part of some collection. Such resources persisting outside a collection are known as singleton resources.

Collections may be present globally, at the top API level. However, they can be within a single resource in which they are known as sub-collections, as shown in Figure 6.3. A sub-collection specifies a kind of 'contained in' relationship.

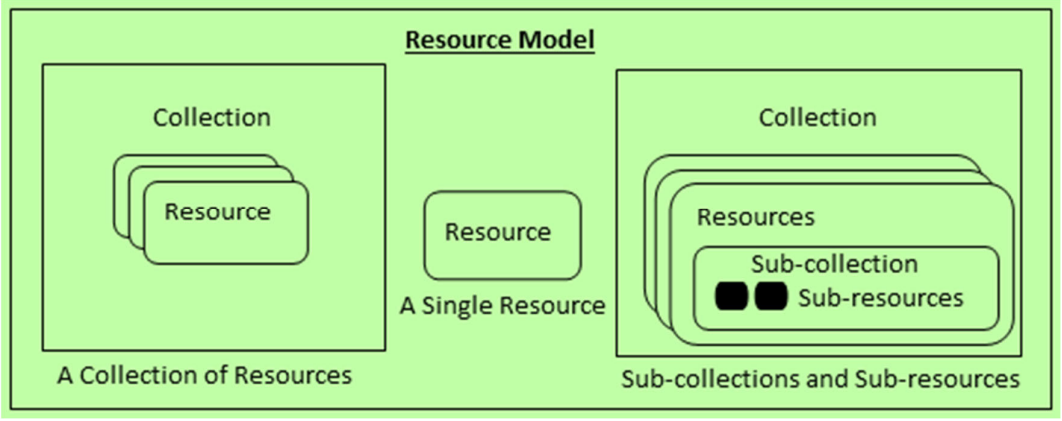


Figure 6.3: REST Resource Model

A REST server gives access to resources to a REST client that also represents them. Each resource has a unique identifier in the form of URI or a Global ID, which is useful when several components communicate with each other. For example, a Task resource can have the identifier as *http://www.users.com/tasks/501*.

To represent a resource, REST supports different media types or formats such as XML, JSON, and plain text. Currently, JSON is the popular media type used in Web services. A component triggers an action on a resource by invoking a method or operation. This method is what the component's uniform interface offers. A resource is typically represented via its intended or current state.

6.1.4 HTTP Messages

A RESTful Web service relies on HTTP for initiating communication or messaging between a client and a server. In the messaging process, the client sends an HTTP Request to the server, which then responds by sending an HTTP Response. The request and response messages contain message data along with the information about that data known as metadata.

Figure 6.4 shows the standard structure of HTTP Request.

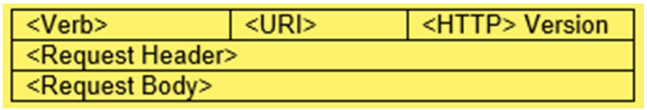


Figure 6.4: HTTP Request Structure

As shown in Figure 6.4, the structure of HTTP Request message has five parts namely:

- **Verb**: Represents an HTTP method such as GET, DELETE, PUT, or POST.
- **URI**: Indicates the identifier for identifying the desired resource on the target server.
- **HTTP Version**: Denotes the version of HTTP, for instance, HTTP v1.1.
- **Request Header**: Has metadata of the message in the form of key-value pairs. For example, the metadata includes the client (browser) type, cache settings, and message body format.
- **Request Body**: Contains the message data in the appropriate format.

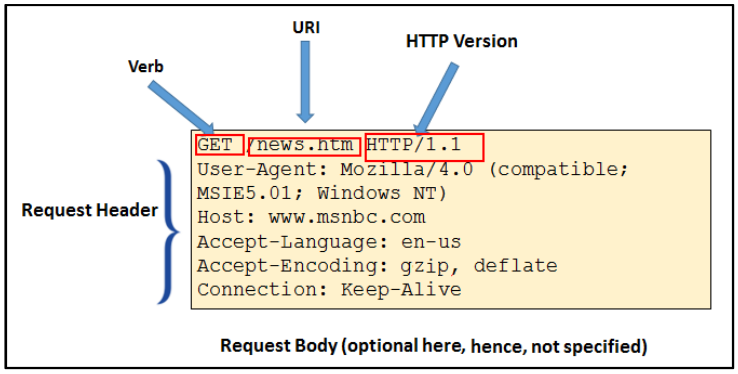


Figure 6.5: Using `$http` Service

Figure 6.6 shows the standard structure of HTTP Response.

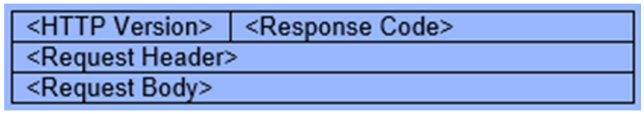


Figure 6.6: HTTP Response Structure

As shown in Figure 6.6, the structure of HTTP Response message has four parts namely:

- **HTTP Version**: Denotes the version of HTTP, for instance, HTTP v1.1.
- **Response Code:** Shows the three digit number indicating the status for the resource requested. For example, code 404 indicates **resource not found**, while 200 means the response is **OK**.
- **Response Header**: Has metadata of the response message in the form of key-value pairs. For example, the metadata includes the content length, response date, server type, and content type.
- **Response Body**: Contains the message data in the appropriate format.

Table 6.2 describes the different HTTP response codes.

HTTP Response Code	Message	Description
200	OK	Indicates success.
201	Created	Indicates that the desired resource is successfully created via PUT or POST request and returns a link to it through the location header.
204	No Content	Shows up when the response body is blank. For example, the code is displayed when a DELETE request is successfully processed.
304	Not Modified	Is used for decreasing bandwidth usage if there are conditional GET requests. The response body is blank and the header has metadata such as location and date.
400	Bad Request	Indicates that an invalid input is given in the request.

HTTP Response Code	Message	Description
		For example, it can be missing data or inappropriate parameter.
401	Unauthorized	Indicates that the client is using an unacceptable or wrong token of authentication.
403	Forbidden	Indicates that the client has no access to the requested method. For example, this code is displayed when a user, other than the administrator, makes a DELETE request.
404	Not Found	Indicates that the method is unavailable.
409	Conflict	Indicates a conflict that is triggered while running the requested method. For example, this code is displayed if a client tries to add an entry, which already exists.
500	Internal Server Error	Indicates that the server has raised an exception while executing the requested method.

Table 6.2: HTTP Response Codes

6.2 RESTful Services in AngularJS

There are three options for accessing services from a RESTful API in AngularJS, which are namely, `$http`, `$resource`, and `Restangular`.

6.2.1 $http

The `$http` service in AngularJS sends an Asynchronous JavaScript and XML (AJAX) request to a remote Web service and retrieves data from it via JSON. In the asynchronous mode, JavaScript sends the request to the server and executes it without waiting for the reply. Once a reply is obtained, a browser event is triggered, which in turn, enables the script to execute the associated actions. HTTP methods such as GET, POST, PUT, and DELETE are also available with the `$http` service.

This service is a function with only one input parameter, which is known as the request configuration object. Following syntax shows how to use the `$http` service in AngularJS:

Syntax:

```
$http({ method: 'GET', url: '/user1' })
  success(function (data, status, headers, config) {
    // ...
  })
  error(function (data, status, headers, config) {
    // ...
  });
```

In the syntax, the `$http` service takes a single parameter, which is a request configuration object. The object states the HTTP method, the URL, action to take upon success, and action to take upon failure. It also generates an HTTP request and returns a response as an object having the following properties:

- **`data`**: Is the response from the server in form of a string or an object.
- **`status`**: Is the HTTP status code.
- **`headers`**: Refer to a function that retrieves header information.
- **`config`**: Refers to the configuration object that created the request.

The `$http.get()` method returns a **promise** object with a standard `then()` method. The promise object comprises two major callback methods namely, *successCallback* and *errorCallback,* based on whether the request was resolved successfully or failed. A promise is a concept for implementing asynchronous operations and denotes the ultimate outcome of an action. It conveys what to do when a specific action succeeds or fails. In technical terms, it is an object holding a reference to two callback methods, of which a suitable one is invoked if an action succeeds or fails. In short, these methods will help to handle the server response. The *successCallback* method is invoked when the response is successfully available, while *errorCallback* is called when the response contains an error status code. The latter function obtains a single object with some properties such as `data` and `statusText` properties. Using these properties helps in finding the reasons for failure.

Thus, the `.then()` method is called on `$http.get()` and will return a promise object.

Note: In older versions of AngularJS (1.3.x and earlier) **`$http`** service has two additional methods `success()` and `error()` that are similar to `then()`.

Code Snippet 1 shows the complete code for an example that uses `get()` method of `$http`. In this example, data is retrieved from the Ergast Developer API which is a free experimental Web service providing historical record of motor racing data. Using this API, one can retrieve data for Formula One series championships, from as early as 1950 to the present.

Code Snippet 1: (f1getdemo.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
  <title>HTTP Get Demo</title>
  <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
</head>
<body ng-app="myApp" ng-controller="FormulaOneCtrl">
   {{text}}
   <script>
     var app = angular.module('myApp', []);
     app.controller('FormulaOneCtrl', function($scope, $http) {
        var                              promise                              =
$http.get("http://ergast.com/api/f1/2019/circuits.json");
        promise.then(function() {
               $scope.text ="Request succeeded.";
        }),
        (function(response, status) {
           $scope.text ="Request failed.";
        });
     });
   </script>
</body>
</html>
```

In Code Snippet 1, the `get()` method of `$http` makes a request to the given URL to retrieve motor racing circuit information for F1 races conducted during 2019. If the request is successful, a message "`Request succeeded.`" is displayed in the view through the `text` variable. If the request fails, "`Request failed.`" is displayed in the view through the `text` variable.

In the snippet, it is the job of the `promise` object to convey what to do when a request fails or succeeds. The `function` parameter is passed to the success and error callback methods. Note that the return value of `$http.get()` could have been named anything, not necessarily `promise`. This name has been used here only to make it easier to understand how a promise works.

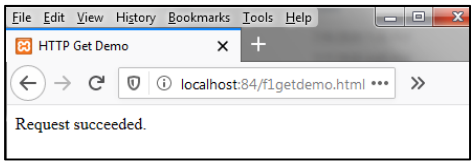


Figure 6.7: Output of Using `$http` Service

Code Snippet 2 builds on the same example to actually display the information that is retrieved using `get()` method of `$http`.

Code Snippet 2: (getdemo.html)

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <title>HTTP Get Demo</title>
    <script src =
      https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js>
  </script>
</head>
<body>
   <div ng-app="myApp" ng-controller="FormulaOneCtrl">
   <pre>
   {{text}}
   </pre>
   </div>
   <script>
   var app = angular.module('myApp', []);
   app.controller('FormulaOneCtrl', function($scope, $http) {
      $http.get("http://ergast.com/api/f1/2019/circuits.json")
       .then(function(response) {
                 var myObj = response.data;
                 $scope.text = JSON.stringify(myObj, undefined, 2);
        });
   });
   </script>
</body>
</html>
```

In Code Snippet 2, the `get()` method of `$http` fetches the response from the given URL. In the code, motor racing circuit information is being retrieved for the F1 races conducted during the year 2019. This response is then displayed in a readable format using `JSON.stringify()` method.

Figure 6.8 shows the output of the code.

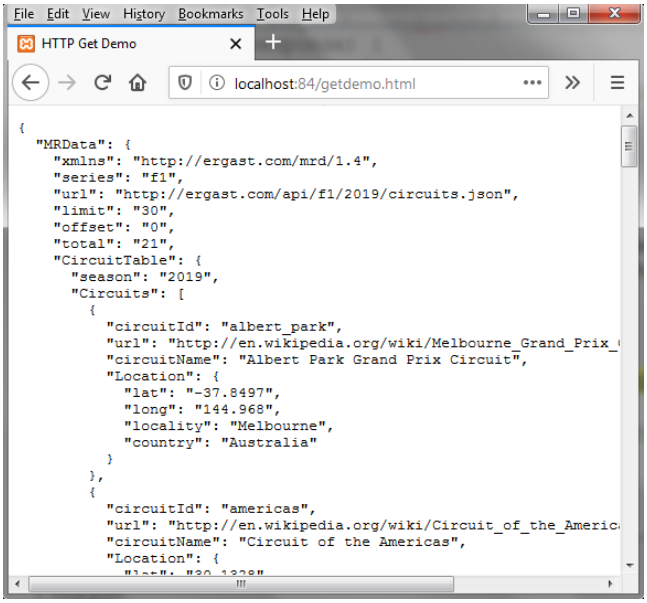


Figure 6.8: Displaying Response Data

Let us now see an example of how `$http` invokes a success or error callback. Code Snippet 3 shows the code that invokes these methods.

Code Snippet 3: (successanderror.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>HTTP Success and Error Demo</title>
    <script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
    </script>
</head>
<body>
  <div ng-app="myApp" ng-controller="FormulaOneCtrl">
  {{text}}
  </div>
  <script>
  var app = angular.module('myApp', [ ]);
  app.controller('FormulaOneCtrl', function($scope, $http) {
  $http.get("http://ergast.com/api/f1/2019/1/res.json")
        .then(
      /* success */
      function(response) {
         $scope.text=response.data;
      },
      /* failure */
      function(error) {
         $scope.text="There was an error processing this URL";
    });
  });
  </script>
</body>
</html>
```

In the code, an HTTP request is made to retrieve data from `http://ergast.com/api/f1/2019/1/res.json` using `$http` service. Then the code defines two methods `success` and `error` to map for the two possibilities that may result because of the request. If the request succeeds, an appropriate message is displayed on the console and the view is also updated with the retrieved data. If the request fails, a message is displayed on the console and the view displays a message stating about the error. Observe that both these methods are defined using functions after the `.then` clause and they are separated by a comma.

When the application is executed, the `error()` method of the `promise` object is invoked, as the request failed to access the file resource. This is because the URL here referred to a non-existent file.

Figure 6.9 shows the output.

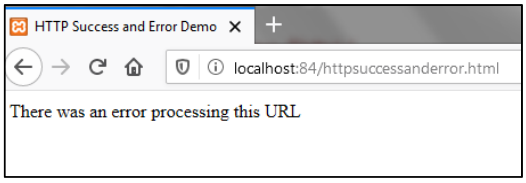


Figure 6.9: Success and Error Demo with HTTP

6.2.2 $resource

The `$resource` service in AngularJS also gives access to REST services. `$resource` is meant to retrieve data from an endpoint, manipulate it, and send it back. It is useful for implementing the CRUD operations. Built on the top of `$http`, the `$resource` service allows communicating with the RESTful backend.

6.2.3 Difference between $http and $resource in Accessing REST APIs

The `$http` service handles general-purpose AJAX calls. This is mostly what the developers will be using in most cases. Through this service, developers manually make GET, POST, or DELETE calls and process the objects they return.

In a RESTful Web scenario, `$resource` covers `$http`. Thus, for accessing a RESTful Web service, `$resource` is the easiest option. For accessing something that is not a RESTful Web service, `$http` is the common option. Although you can use it to access a RESTful Web service, it ends up being more cumbersome.

The purpose of `$resource` is to enable passing a string containing placeholders and the parameters values. The service then, replaces the placeholders with the passed parameter values. This is useful while communicating with a RESTful data source, as it works on similar principles for defining the URLs.

6.2.4 Restangular

One more AngularJS service to access REST services is Restangular, which is a third party open source framework service. It simplifies HTTP requests by having a minimum client code for them. It is ideal for a Web application that uses data from a RESTful API. Restangular is an AngularJS service that simplifies common GET, POST, DELETE, and UPDATE requests with a minimum of client code. It is a perfect fit for any Web application that consumes data from a RESTful API.

Unlike `$resource`, Restangular:

- Supports all HTTP methods.
- Eliminates the need to recall or write an URL. Just specifying the resource name fetches all the required details. In case of `$resource`, it is necessary to specify the URL.
- Allows creating custom methods.

There are three ways for downloading the service, which are as follows:

- **Through Bower Web Manager Application**: By running **$ bower install restangular** on its command prompt. Bower is a Web package manager for managing front-end elements such as HTML, JavaScript, and CSS.
- **Through Node Package Manager (NPM)**: By running **$ npm install restangular**. NPM is the default manager written in JavaScript for handling Node.js. This file is written for executing the runtime environment of JavaScript. Installing this file automatically includes the package manager. NPM is a command line utility on the client-side, which communicates with a remote registry to distribute JavaScript modules from that registry.
- **Through a Content Delivery Network (CDN)**: By including the reference in the `script` tag. The CDN option is for those who do not intend to host the service by themselves. It refers to a system of distributed servers for delivering the requested content as per the geographic location of server and the user. The closer the server, the faster is the content sent to the user. This provides the benefit of faster loading time.

  Code Snippet 4 shows the code to include in the `script` tags for referring to the service through a CDN. Note that we are referring to Restangular 1.6.1 here. Also, an additional library, lodash.js, is mentioned here. Restangular has a dependency on this library, hence, it has been included here. It is a JavaScript based library providing utility functions for common programming tasks

  Code Snippet 4: (fragment of restangulardemo.html)

```
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular-
resource.js">
</script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.4/lodash.j
s">
</script>
 <script
src="http://cdnjs.cloudflare.com/ajax/libs/restangular/1.6.1/restangu
lar.js">
```

```
</script>
```

Working with Restangular

To use Restangular, it is essential to configure its settings in the app module definition file. The service has defaults for its properties, which a Web developer can change for setting the desired configuration. To set these configurations and change the global configuration, a Web developer should use the `RestangularProvider` object of Restangular.

Code Snippet 5 shows an example of retrieving data from a public API using Restangular with AngularJS. Studio Ghibli is a well-known animation film studio based in Japan whose animated feature films have captivated audiences worldwide. Studio Ghibli has provided a free public API using which developers can retrieve information about films the studio has made.

Code Snippet 5: (complete code for restangulardemo.html)

```
<!DOCTYPE html>
<html ng-app="app">
<head>
  <meta charset="UTF-8">
  <script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.j
s">
  </script>
  <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular-
  resource.js">
  </script>
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.4/lodash.js">
  </script>
   <script src =
    "http://cdnjs.cloudflare.com/ajax/libs/restangular/1.6.1/restangular.js"
    >
  </script>
</head>
<body>
    <div ng-controller="IndexCtrl" >
      Film Titles from Studio Ghibli:<br>
        <ul>
            <li ng-repeat="title in movies">{{title.title}}<br></li>
        </ul>
    </div>
    <script>
      var app = angular.module('app', ['restangular'])
            .config(function(RestangularProvider) {

RestangularProvider.setBaseUrl('https://ghibliapi.herokuapp.com');
                });

      app.controller('IndexCtrl', function($scope, Restangular) {
            var data =
Restangular.all('/films').getList().then(function(result) {
            $scope.movies = result;
            });
        });
    </script>
</body>
</html>
```

In Code Snippet 5, the `restangular` service is injected into the AngularJS application and using the `Restangular.all()` method, information about films is retrieved as a list. `RestangularProvider.setBaseUrl` is used to set the beginning URL in the API. The code also

uses a `.then` to check whether the request is succeeded or failed. The retrieved list is then passed to the view where each film title from the list is displayed as a bulleted list item.

The statements in yellow are the major ones in the application.

Figure 6.10 shows the outcome of the code.

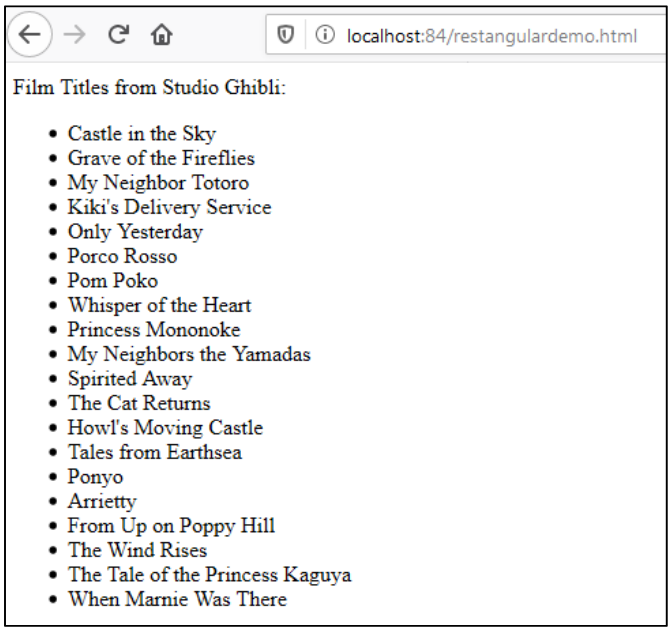


Figure 6.10: Retrieving Film Titles Using Restangular

6.3 Server Communication

AngularJS is a popular framework, as it enables extending HTML for defining templates and implementing two-way data binding. In addition to templates, real world applications are reliant on the backend servers for processing the core business logic. For communication with such servers, AngularJS offers an integral support. It offers a low-level mechanism acting as a building block for communication and built-in wrappers for interacting with RESTful services.

An AngularJS application is also a single page application due to which it offers support for AJAX communication. Regarded as the core of single page applications, AJAX facilitates interaction with the servers without refreshing the page. This ability has changed the way of developing Web applications. Now, a standard Web application separately fetches the layout (HTML) and data, which differs from the way of former applications fetching the data and layout together.

6.4 Accessing Server Resources with $http

AngularJS has various built-in ways of interacting with a Web server. One of these ways is through the `$http` service, which offers an easy way to communicate asynchronously via HTTP. It acts as the fundamental building block for server communication. Practically, this core service allows communicating with a remote HTTP server through one of the two ways namely, JSONP or `XMLHttpRequest` object of the browser.

The service is for generic AJAX calls that can interact with the server's RESTful or Non-RESTful API. It is somewhat similar to `jQuery.ajax`. The `$http` API is established on the standardized way of handling asynchronous calls in JavaScript. Following are the basic functionalities of the `$http` service:

- GET
- POST
- HEAD
- PUT
- DELETE
- JSONP

For communicating with a server, the AngularJS API utilizes the `Promise` interface that ensures non-blocking/asynchronous calls. This means that the server's response is returned in near future. The interface ensures that the appropriate callback method or the handler has handled the response.

Figure 6.11 shows how `$http` service retrieves the server resource, `assessment`. In the code, `$http` service is injected directly into the controller and the assessment resource is fetched using `config` object. The custom service, `assessmentService`, uses the built-in `$http` service. In line-4, this service takes two parameters namely, the `GET` HTTP request method and `url` to retrieve assessment. These parameters are a part of the `config` object. Hence, the `config` object is the request parameter to the `$http` function. It represents the HTTP request and includes its various properties that can be set as a part of the request. The service returns the data or response in the JSON format.

```
1  angular.module('myApp.services',[]).service('assessmentService',
2      function($http) {
3          this.getCourse = function() {
4          return $http.get('http://localhost:8000/assessment/');
5      }
6  });
7
```

Figure 6.11: Code for Custom Service Retrieving Details from a Resource

Figure 6.12 shows how a custom service, `assessmentService`, retrieves the assessment data along with the status through the callback methods.

```
1  angular.module('demoApp.controllers', []).
2    controller('DemoController', function($scope, assessmentService) {
3        $scope.getAssessment = function() {
4            assessmentService.getAssessment().then(function(response) {
5            $scope.retValue = response.data;
6        }, function(response) {
7            $scope.retValue = response.status;
8        });
9        }
10 });
```

Figure 6.12: Code for Retrieving Status and Details from the Scope of a Custom Service

In Figure 6.12, the code defines the `getAssessment()` method to retrieve the assessment details and request status. When the server completes processing the request using `$http`, it invokes either the success or error callback method. This is based on whether the request is processed successfully or not. These actions take place asynchronously. The `response` parameter passed to these callback functions is an object that contains the following properties:

- `data`: Holds a response from the server.
- `status`: Shows an HTTP status code.

Following are the steps to access a server resource using `$http`:

1. Define a custom service in which you can directly inject the `$http` service.
2. Use the `$http` service, which uses a request method.
3. Pass the parameters as a part of the `config` object, which shall return a promise object.
4. Fetch the response once the request has been processed.

The Promise API has the following flow:

1. Each asynchronous request returns a promise object. This object has a `then` function with two arguments, a success and an error callback.
2. Once the asynchronous task is over, the server invokes any of these callback handlers only once, which may return a value.
3. The `then` function now returns a promise.

6.5 Handling Errors in Client-Server Communication

Handling and logging an exception is an important aspect of application development. Many developers tend to skip this aspect. However, there is no guarantee that a developer or tester shall receive a notification when an application component fails to function. Similarly, there is no assurance that AngularJS will detect and handle every exception in any situation. Even if a developer assumes that AngularJS will detect all exceptions, it is worth thinking whether the default message shall have sufficient information to aid in debugging. Analyzing exception messages that are difficult to understand is likely to consume more time than logging and handling the corresponding exceptions. Thus, it is essential to handle and log errors via the application code and provide helpful information to the user regarding the error generated.

Developers can use `try`, `catch`, and `finally` JavaScript blocks in AngularJS modules for handling exceptions. Apart from that, AngularJS also offers the `$exceptionHandler` service for catching an uncaught exception. This service passes the exception, when caught, to `$log.error`, which is responsible for logging the exception into the browser's console. The `$exceptionHandler` service cannot deal with syntax errors, but it is possible to override it as per requirements.

Code Snippet 6 shows how to handle an uncaught exception through the `$exceptionHandler` service.

Code Snippet 6: (exceptiondemo.html)

```
<!DOCTYPE html>
<html ng-app="studentApp">
<head>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
</head>
<body class="container" ng-controller="studentController">
    Status: {{status}} <br />
    Data: {{data}} <br />
        <input type="button" value="Get Data" ng-click="getStudent()" />
    <script>
        var app = angular.module('studentApp', []);

        app.config(function ($provide) {
            $provide.decorator('$exceptionHandler', function ($delegate) {
                return function (exception, cause) {
                    $delegate(exception, cause);
                    alert('Error occurred! Please contact admin.');
                };
            });
        });
        app.controller("studentController", function ($scope) {
            var onSuccess = function (response) {
                $scope.status = response.status;
                $scope.data = response.data;
            };
            var onError = function (response) {
                $scope.status = response.status;
                $scope.data = response.data;
            }
            $scope.getStudent = function () {
                $http.get("/getdata").then(onSuccess, onError);
            };
        });
    </script>
</body>
</html>
```

In Code Snippet 6, we override default behavior of `$provide` service by implementing `$provide.decorator()` method in `app.config()`. As the `decorator()` method allows overriding the service, it contains code for logging a raised exception and showing a custom message

in the console.

Figure 6.13 shows the output in which an exception is raised after clicking the button and a custom message is displayed to the user.

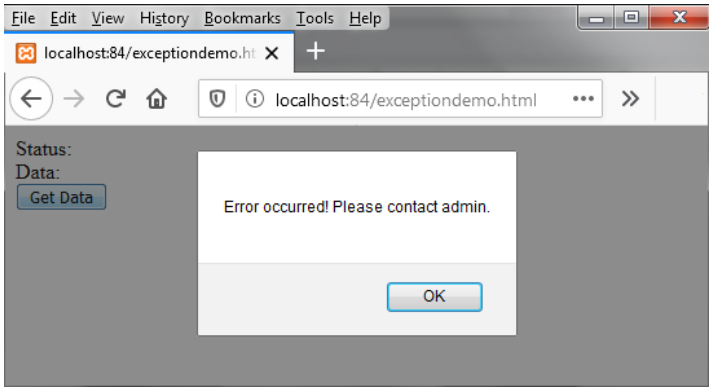


Figure 6.13: Exception Handling Output

6.6 Server Communication Using `$resource`

Developers can perform server-side operations using the `$resource` service and also access server-side resources.
CRUD operations become easier through this service.

The `$resource` service is not readily available with the main AngularJS script. Thus, a developer should download a separate file for it and add it to the index.html file. Code Snippet 7 shows how to include the file in **index.html**.

Code Snippet 7: (fragment of resourcedemo.html)

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular-
resource.min.js"></script>
```

The next step is to declare a dependency on `$resource` for using it within a controller. Then, the developer should invoke the `$resource()` function with the desired REST endpoint address. Code Snippet 8 shows these steps.

Code Snippet 8: (fragment of resourcedemo.html)

```
var app = angular.module('resourceApp', ['ngResource']);
app.controller('MainCtrl', function($scope, $resource) {
    var Customers = $resource('Customer.json').query(function(){
        . . .
```

The function `$resource('Customer.json')` returns an object of the `$resource` class, which helps in communicating with a REST backend server.

This object, by default, has five methods, which are as follows:

- `get()`
- `save()`
- `query()`
- `delete()`
- `remove()`

These methods map against HTTP actions as follows:

```
get: {method:'GET'},
save: {method:'POST'},
query: {method:'GET', isArray:true},
delete: {method:'DELETE'},
```

`remove: {method:'DELETE'}`

All these methods are available as 'class level' actions on the resource object. This means that developers can directly call them on the resource object. For example, if customer is a resource object, a developer can write `Customer.get(params)` or `Customer.query()`, with the required parameters.

Calling these methods will in turn call or invoke `$http` with the specified HTTP method, destination, and parameters.

Code Snippet 9 demonstrates the usage of `query()` and `save()` methods.

Code Snippet 9: (resourcedemo.html)

```
<!DOCTYPE html>
<html lang="en" ng-app="resourceApp">
<head>
  <meta charset="UTF-8">
  <title>Resource Demo</title>
  <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
  </script>
  <script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular-
    resource.min.js">
  </script>
</head>
<body ng-controller="MainCtrl">
     Customer Names:
    <select ng-options="Customer.Name for Customer in Customers" ng-model =
     "CurrentCustomer"></select>
      <br>
      <br>
      New Name: <input type = "text" name = "CustName" ng-
model="CustName"><br>
       <input type = "submit" value = "Submit" ng-click="Add()" >
       {{text}}
</body>
<script>
var app = angular.module('resourceApp', ['ngResource']);
app.controller('MainCtrl', function($scope, $resource) {
    var Customers = $resource('Customer.json').query(function(){
        $scope.Customers = Customers;
        $scope.CurrentCustomer = Customers[0];
        $scope.Add = function () {
            $scope.CurrentCustomer.Name = $scope.CustName;
            Customers.save($scope.CurrentCustomer, function() {
                   $scope.text="Added";
                      });
                };
       });
});
</script>
</html>
```

In Code Snippet 9, a data-bound list box is created and then populated with customer data retrieved from a JSON file named **Customer.json**. The initial item in the list is set by retrieving the first entry from the JSON file. Then, a new customer name is accepted using a text box and this name is stored into the local resource object (in this case, the variable `Customers`) and also displayed on the browser. To perform these operations of retrieving and saving, the `query()` and `save()` methods of `$resource` are used.

Customer.json should be created with following data:

```
[{
      "Name": "James March"
}, {
      "Name": "Alex Knight"
}, {
      "Name": "Polly Parker"
}]
```

Figures 6.14 and 6.15 depict the output of the code before and after submitting the new name respectively.

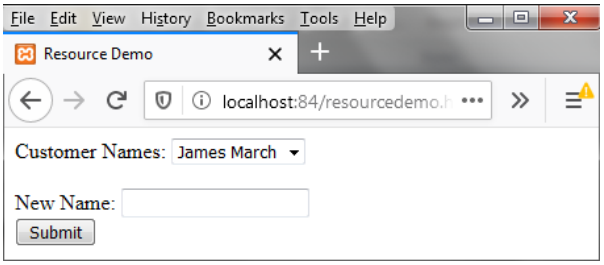


Figure 6.14: Using $resource

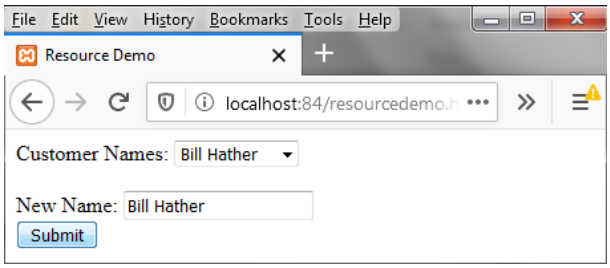


Figure 6.15: Using Save with $resource

Note that in Code Snippet 9, the code is rather simple and hence, the modified data is only displayed in the view, but is not saved to the actual file on the disk. To do that, one will need more advanced code.

Quick Test 6.1

1. REST API is a protocol.
    a. True
    b. False

2. The client and server communication in REST is asynchronous.
    a. True
    b. False

6.7 Summary

- REST is a stateless architecture that aids in making networking applications by implementing HTTP.
- RESTful applications work on six principles or constraints namely client-server communication, statelessness, cacheability, uniform interface, layered system, and code on demand.
- An HTTP response returns a status code that indicates the status of the requested resource.
- The `$http` service communicates asynchronously with a remote Web service and returns a `Promise` object.
- The `$resource` service is easier than `$http` to access a RESTful Web service and a RESTful backend.
- The `$exceptionHandler` service catches an uncaught exception and passes it to `$log.error` for displaying the exception in the browser's console.

6.8 Exercise

1. Which of the following HTTP request components indicates an HTTP method?
   a) URI
   b) Verb
   c) HTTP version
   d) Header

2. Which of the following protocols is used by a RESTful Web application?

   a) SOAP
   b) FTP
   c) HTTP
   d) RPC

3. Which of the following status codes indicates successful creation of a resource?

   a) 200
   b) 201
   c) 304
   d) 204

4. The config object returns the _____ object.

   a) `HTTP Response`
   b) `$http`
   c) `$resource`
   d) `Promise`

5. The _____ service logs the exception into the browser's console logs errors.

   a) `$log.error`
   b) `try and catch`
   c) `$exceptionHandler`
   d) `$exception`

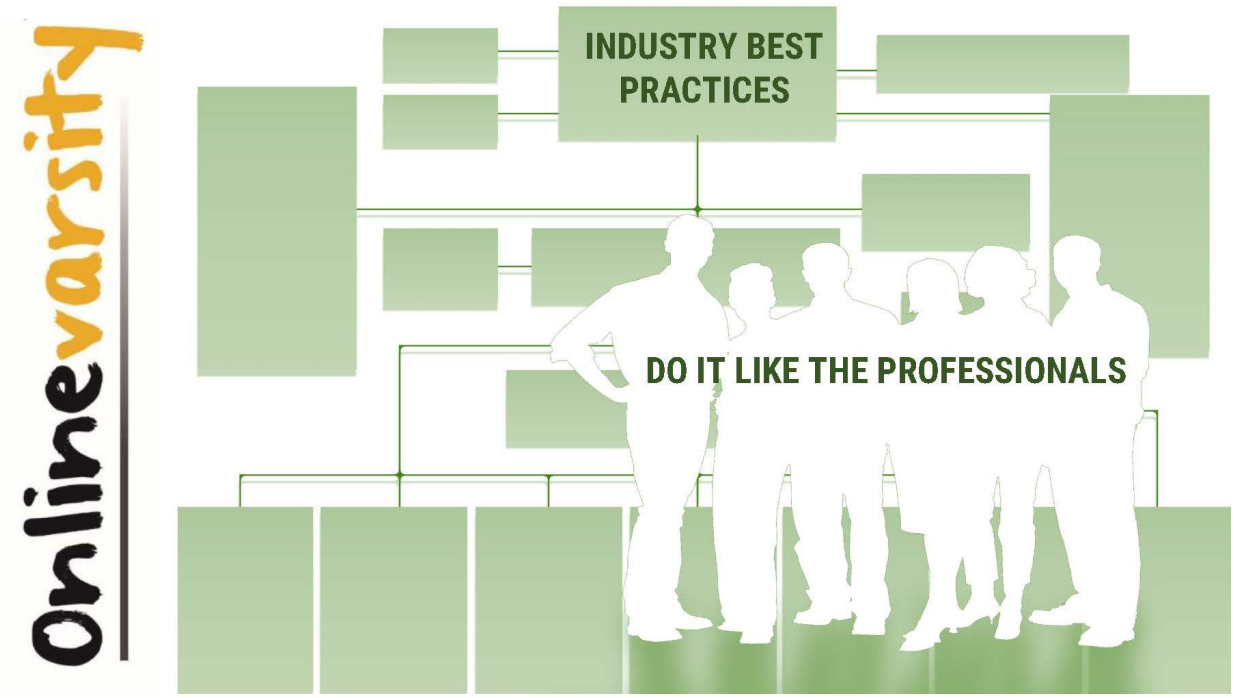
6.9 Do It Yourself

1. Create an AngularJS application that uses the public API for TV shows at TVMaze.com and retrieves a complete list of seasons for a given show. Seasons are returned in ascending order and contain the full information known about them.

   Hint: Refer to Code Snippet 2 and use the URL http://api.tvmaze.com/shows/1/seasons to retrieve the data.

2. Create an AngularJS application similar to Code Snippet 9, but which uses a data file products.json having following data and then adds a given product to the displayed list of products in the browser:

```
[{
        "Product": "Cereal"
}, {
        "Product": "Flour"
}, {
        "Product": "Sugar"
},{
        "Product": "Salt"
},{
        "Product": "Chilli"
}]
```



Answers to Exercise

1. Verb
2. HTTP
3. 201
4. Promise
5. `$log.error`

Answers to Quick Test

Quick Test 6.1

1. False
2. True

Session 7

Building Single Page Applications (SPAs) in AngularJS

In this session, students will learn to:

- Explain dependency injection and its working in AngularJS
- Describe factory and service in AngularJS
- Outline the differences between factory and service and their uses
- Explain the usage of SPAs in AngularJS

This session introduces the concept of Dependency Injection (DI) along with its core components and objects that are injectable into each other in AngularJS. Differences between the factory and service components are covered in a simplified manner. A basic knowledge of HTML, JavaScript, and JQuery will be useful in comprehending how an SPA works.

7.1 AngularJS as an MVC Framework

AngularJS is a JavaScript MVC framework developed for creating well-structured and efficiently maintainable Web applications. It focuses more on the concepts of Model, View, and Controller rather than handling DOM manipulation directly from the application logic layer. For example, there is no need to use `document.getelementbyid('..')`.

Apart from rational implementation of MVC, AngularJS ensures efficient two-way data binding process. Through this framework, it is easy to define custom HTML custom tags and attributes. To make Web applications more maintainable, AngularJS offers an integrated Dependency Injection (DI) mechanism.

7.2 DI Mechanism

DI is one of the best features available across the AngularJS framework. It refers to a technique or a software design pattern that allows passing an object as a dependency. It ensures that a component does not refer to other components directly, but obtains references to them.

Technique injects a dependent functionality into a module at the time of execution without coding for it. So, it aids in eliminating the need of hard-coded dependencies. In simple words, it is now possible to request for dependencies instead of creating or managing them through coding. Therefore, AngularJS provides an object with its dependencies.

A developer can divide an application into multiple components, which are injectable into each other. Modularizing any AngularJS Web application in this way makes it easier to configure, reuse, change, and test its components. The built-in injector sub-system is responsible for making components, sorting out their dependencies, and offering them to other components whenever requested.

Figure 7.1 illustrates the DI mechanism.

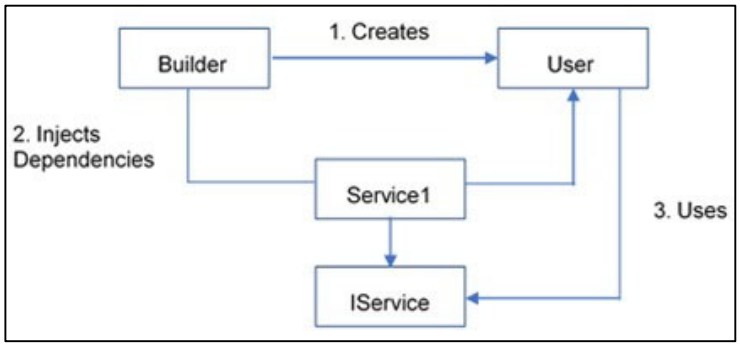


Figure 7.1: Working of DI Mechanism

In figure 7.1, the DI pattern uses a `Builder` object for initializing a `User` object and offering the required dependencies to the `User` object. The injection occurs by passing `Service1` that implements the interface, `IService`. This is how AngularJS allows injecting a dependency. It allows passing dependencies as arguments of a function or an array.

As mentioned earlier, using dependency injection makes the code more re-usable. For example, if there is functionality that shall be used across several application modules, it is ideal to define a common service with that functionality and inject it as a dependency in those modules.

Uses of DI in AngularJS include:

- Separating the process of creating and using dependencies
- Creating independently functioning dependencies
- Changing dependencies easily whenever requested
- Injecting a mock object as a dependency for more effective testing

Following are components and objects that AngularJS injects into each other as dependencies:

7.1.1 Value

A value is a JavaScript object, string, or a number that AngularJS allows injecting into a controller during the `config` phase (bootstrapping). It is usually injected into a service, controller, or a factory. Code Snippet 1 shows how to add values to a module.

Code Snippet 1:

```
var newMod = angular.module("newMod", []);
newMod.value("number", 10);
newMod.value("string", "employee");
newMod.value("object", { value1 : 50, value2 : "manager"} );
```

In Code Snippet 1, the `value()` function defines different values in the module, where the first parameter is the value name and the second parameter is the value. To inject a value into a controller function, a developer just needs to add a parameter whose name is same as value name.

Code Snippet 2 shows how to inject a value into a controller.

Code Snippet 2:

```
<html ng-app="app">
 <head>
 <script src =
"http://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
 <script>
    var app = angular.module("app", [ ]);
    app.value('empId', '101');
    app.controller("MyController", ['$scope', "empId", function
    MyController($scope, empId){
          $scope.empId = empId;
        }
     ]);
</script>
</head>
<body ng-controller='MyController'>
<p>Hello Employee, <b>{{empId}}</b>! Welcome.</p>
</body>
</html>
```

In Code Snippet 2, the controller accepts two parameters. First parameter is the controller name and second is an array.

The first two elements in the array are object names and the last one is an anonymous function. Inside this function, the parameter names are same as the object names in the array. Here, the second parameter has the same name as the value. This means that the controller has a dependency on `empId`. Its value, which is `101`, is assigned to a variable named `empId` in the `$scope` object. This passes the value from the controller to the view.

Defining an array that holds a list of strings, which are the names of dependencies, and the function itself is the most preferred way of annotating a controller function. It is known as inline array annotation. Annotating will convey about the services that AngularJS needs to inject into the controller function.

Figure 7.2 shows the output.

Figure 7.2: Output of Injecting a Value into a Controller

7.1.2 Provider

AngularJS internally uses a provider to create a service or factory during the `config` phase. A provider is a distinct factory function with the `$get()` method and returns a value, factory, or a service. It tells AngularJS how to create a new injectable service. Hence, a provider defines a service, which is created by using `$provide`.

AngularJS allows defining a provider through the `provider()` method, which in turn, invokes the `$provide` service. Code Snippet 3 shows an example.

Code Snippet 3:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Demo</title>
      <script src =
    "http://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
      </script>
  </head>
      <body>
            <h2>Provider Demonstration</h2>
            <hr/>
            <script>
              var app = angular.module('app', []);
              app.provider("myProvider", function(){
                  this.$get = function(){
                   return{
                     Display: function(){
                                 return "Displaying Hello From the
                                 Service";
                                    },
                        };
                     };
                  });
                  //myProvider
                  app.controller("myController", function ($scope,
                 myProvider){
                        //provider function call
                        $scope.ProviderOutput = "Provider output";
                        $scope.DisplayProvider =  function(){
```

```
                                    $scope.flag=true;
                                    $scope.ProviderOutput =
                                        myProvider.Display();
                                };
                    });
</script>
<div ng-app="app" ng-controller="myController">
<h3>Using Provider</h3>
<button ng-model="button" ng-click="DisplayProvider()" ng-disabled = "flag"
> Display </button>
<div ng-bind="ProviderOutput"/>
</div>
</body>
</html>
```

In Code Snippet 3, the `$provide` service is responsible for telling AngularJS how to create new injectable items called services. The code defines a new provider for a service called `myProvider` and injects it into the controller, `myController`. `$scope` sets the `ProviderOutput` property of the `DisplayProvider` object. Using the `click` property of the button, the code retrieves the string, `Displaying Hello From the Service`, which is set as the value of the `ProviderOutput` property.

Figures 7.3 and 7.4 show the output that is displayed when a user clicks the `Display` button.

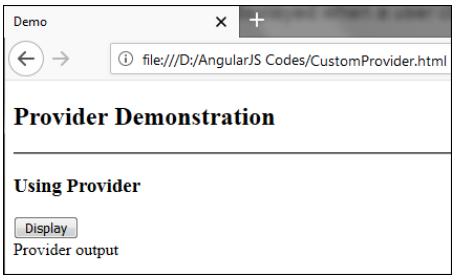


Figure 7.3: Initial Output of CustomProvider

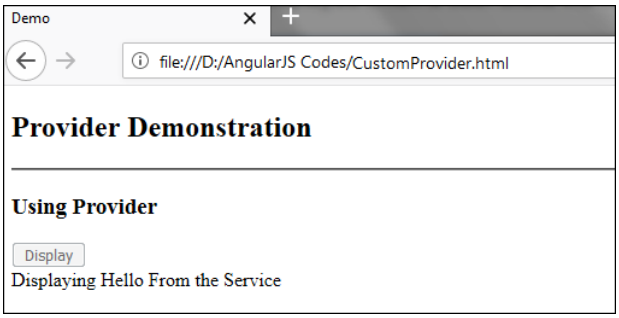


Figure 7.4: Output through CustomProvider

Though this example was not practical for real world purposes, it demonstrates how a provider can be injected for creating a service. In advanced AngularJS applications, the provider and service can be utilized to perform complex tasks.

A developer can also specify functions using `config()` and `run()`, which then run at configuration and run time, respectively. These functions are injectable with dependencies.

Code Snippet 4 shows how to inject functions using the two functions.

Code Snippet 4:

```
var application = angular.module('newapp', []);
application.config(['provider', function (provider){
    //Code to do something
}])
application.run(['service', function (service) {
    //Code to do something
}]);
```

Injection is possible either while defining components or while defining the `run` and `config` blocks. The function parameter of `config()` is injectable with the constant and provider components as dependencies. It does not accept a value or service components. Similarly, the function parameter of `run()` is injectable with three components namely, a service, value, or a constant. These components act as dependencies. It does not accept a provider.

7.1.3 Constant

In AngularJS, `config()` cannot take an instance of a service or a value as a parameter. Hence, it is impossible to inject a value or an instance into it. However, it is possible to inject a constant just as a provider for a service during the `config` phase. This means that a constant helps in passing a value during this phase, which is also available at runtime.

AngularJS does not allow changing a constant, once it is defined. This is how it differs from a value. Constants facilitate injecting vendor libraries that are globals. This ensures better testability by revealing the dependencies of your components more easily and mocking these dependencies.

Code Snippet 5 shows how to inject a constant into a controller.

Code Snippet 5:

```
<html>
   <head>
      <title>Angular JS Services</title>
      <script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
     </script>
   </head>
   <body>
     <div ng-app = "mainApp" ng-controller = "myController">
        <p>Enter radius of circle: <input type = "number" ng-model =
        "number" /></p>
         <button id="btnCal" ng-click="area()">Area</sup></button>
          <p>{{pi}}</p>
             <p>Result: {{result}}</p>
      </div>
      <script>
       var mainApp = angular.module("mainApp", []);
       mainApp.constant("pi", "3.14");
       mainApp.controller('myController', function($scope, pi) {
          $scope.area=function(a) {
          $scope.result = pi*($scope.number * $scope.number);
          };
       });
      </script>
   </body>
</html>
```

In Code Snippet 5, the controller calculates and retrieves Area value based on the given radius. Since value of pi remains constant, here it is injected as a constant into the application and then from the application, it is assigned to the scope. Figure 7.5 shows the output.

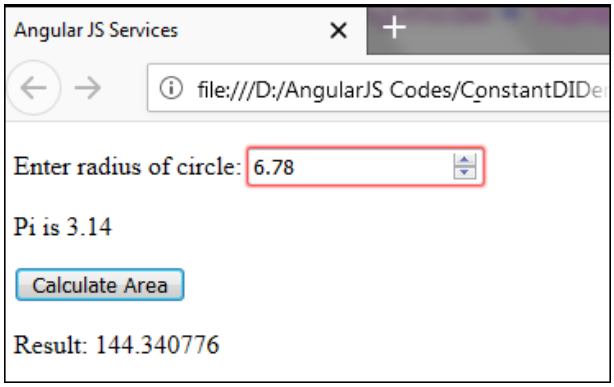


Figure 7.5: Output of Injecting a Constant into a Controller

Code Snippet 6 shows how to inject a constant into a provider.

Code Snippet 6:

```
var app = angular.module("demoapp", []);
app.constant('empId', '5001');
//Injecting a constant into a provider
app.config(['testProvider', 'empId', function (testProvider, empId) {
}]);
```

It is ideal to use constants that are not part of another service and for values that are not going to change. While using constant only for a module that many applications are likely to reuse, it is recommended placing them in a file per module having the same name as the module.

7.2 Factory and Service in AngularJS

AngularJS implements the Separation of Concerns concept through its architecture of services. AngularJS provides two functions for creating a service, which are as follows:

- `factory()`
- `service()`

7.2.1 The Factory Component

The factory component is a function that returns a value, which is reusable for performing other tasks, such as validating a business rule. It generates the desired value when a controller or service needs it from a factory. This created value is reusable for controllers and services that need it.

The component implements the `factory()` function for creating and returning a value. Therefore, unlike the value component for dependency injection, a factory uses a function and is injectable with values.

Code Snippet 7 shows how to create a service using the `factory()` function and inject it into a controller.

Code Snippet 7:

```
<html>
 <head>
   <title>Angular JS Services</title>
   <script src =
  "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js"
>
   </script>
 </head>
 <body>
     <div ng-app = "mainApp" ng-controller = "DemoController">
     <p>Enter a number: <input type = "number" ng-model = "number" /></p>
         <button ng-click = "square()">Square</button>
```

```
         <p>Result: {{result}}</p>
      </div>
      <script>
         var mainApp = angular.module("mainApp", []);
         mainApp.factory('MathService', function() {
                  // Define a factory
                   var factory = {};
                  //Assign a function to it
                   factory.square = function(a) {
                             return a * a;
                   }
                  return factory;
                  });
       //Inject the service created using factory into the controller
            mainApp.controller('DemoController', function($scope,
            MathService) {
                        $scope.square = function() {
                           $scope.result =
                              MathService.square($scope.number);
                        }
                     });
      </script>
   </body>
</html>
```

In Code Snippet 7, the code declares a new factory function, `MathService` in which `square` is set to a function returning the square of a number. The controller, `DemoController`, accepts `$scope` as a parameter, that indicates the application or module that the controller should control. `$scope` represents the application that shall use the `DemoController` object, which in turn, uses the `MathService` function as the argument. The `number` property of `$scope.MathService` object returns the square of the given number when the function is invoked once the `Square` button is clicked.

Figure 7.6 shows the output.

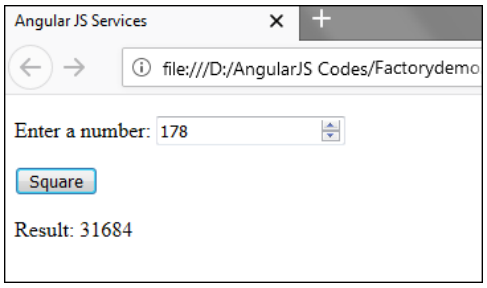


Figure 7.6: Output of Using factory()

7.2.2 Service Component

In AngularJS, a service refers to a singleton JavaScript object holding a collection of functions that are injectable in a controller. These functions have the logic to enable the service to perform its work. For example, `$http` is injected in a controller to invoke its HTTP functions.

To create a service in a module, AngularJS offers the `service()` function. In this function, we define a service and assign the desired functions to it. A service itself is a constructor function, which uses the `new` keyword for creating an object and adding functions and properties to it. However, unlike factory, it returns nothing. A service is injectable into other services, filters, controllers, and directives.

Code Snippet 8 shows how to invoke an already existing function, `MyService`, by using `service()`.

Code Snippet 8:

```
<html>
   <head>
      <title>Angular JS Services</title>
      <script src =
       "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.
js">
      </script>
   </head>
   <body>
        <div ng-app = "mainApp" ng-controller = "CalcController">
         <p>Enter a number: <input type = "number" ng-model = "number"
/></p>
         <button ng-click = "square()">Square</sup></button>
         <p>Result: {{result}}</p>
      </div>
                  <script>
         var mainApp = angular.module("mainApp", []);
         mainApp.service('MyService', function() {
            this.square = function(a) {
               return a * a;
            }
         });
             //Inject the created service into the controller
         mainApp.controller('CalcController', function($scope, MyService) {
            $scope.square = function() {
               $scope.result = MyService.square($scope.number);
            }
         });
      </script>
   </body>
</html>
```

In Code Snippet 8, `MyService` is a new service, while `square` is a function within it. This means that when the application instantiates the new service inside the controller, it would access the new function. The new function accepts only one parameter, `a`. Then, it invokes a function that calculates the square and returns it. The controller references the new service. When AngularJS executes the controller, it instantiates the object of `MyService` type.

An AngularJS service, factory, and a provider are used for defining a utility function that a Web developer can use throughout the page via an injectable object. This means that they all cater to a common purpose. However, the way they are defined and used tends to differ. Let us see one more example that creates a factory and a service.

Code Snippet 9 shows an application that defines a factory as well as a service, both of which return the `cube` of a given number.

Code Snippet 9:

```
<html>
<head>
 <title>Angular JS Services</title>
 <script src =
"https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js">
 </script>
</head>
   <body>
      <div ng-app = "mainApp" ng-controller = "DemoController">
         <p>Enter a number: <input type = "number" ng-model = "number"
/></p>
         <button ng-click = "cube1()">X<sup>3</sup></button>
         <p>Result (Using Factory): {{result1}}</p>
         <button ng-click = "cube2()">X<sup>3</sup></button>
         <p>Result (Using Service): {{result2}}</p>
```

```
      </div>
      <script>
         var mainApp = angular.module("mainApp", []);
         mainApp.factory('Math', function() {
            var factoryObj = {};
            factoryObj.multiply = function(a) {
               return a * a* a;
            }
            return factoryObj;
         });

         mainApp.service('CalcService', function() {
            this.cube = function(a) {
               return a*a*a;
            }
         });
         mainApp.controller('DemoController', function($scope, CalcService,
Math) {
            // Using Service
            $scope.cube1 = function() {
                  $scope.result1 = CalcService.cube($scope.number);
            }
            // Using Factory
              $scope.cube2 = function() {
                  $scope.result2 = Math.multiply($scope.number);
            }
         });
     </script>
   </body>
</html>
```

In Code Snippet 9, the code first creates a module named `mainApp`. Then, it defines a service using `service()` function and a factory using the `factory()` function of the module. In service, the `cube` function is defined using `this` keyword but in factory, the `multiply()` function is created using the new object, `factoryObj`. Both aim to return the cube of the given number and are injected into the `DemoController` controller but their declarations tend to differ. `$scope` refers to the application using the `DemoController` object, which in turn, uses `CalcService` and `Math` functions as arguments. Figure 7.7 shows the output.

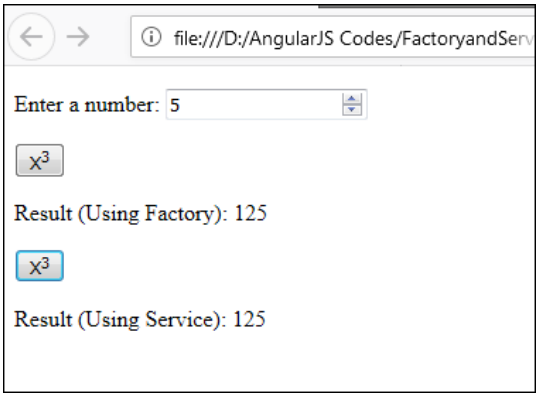


Figure 7.7: Output of Using service() and factory()

In short, even factory and service components are providers. A factory is a special provider useful for having a `$get()` function and reducing the lines of code. On the other hand, a service is a special factory for creating and sharing an instance of a new object across the application by writing less code. For example, a service is useful for sharing user authentication details. Consider using a provider for offering module-based configuration for a service object, as it is the only service that is injectable to the `config()` function.

7.4 Differences between Factory and Service

Factories and services allow creating an object that is reusable anywhere in the application. Both are singletons, facilitate creating custom services, and are not injectable in the `config()` function. However, there are differences between the two, which are similar to the differences between an object and a function.

Table 7.1 distinguishes between a factory and a service.

Point of Distinction	Factory	Service
Function Type	Is a function that returns an object or a value.	Is a constructor function that uses the `new` keyword to declare an object. It is instantiated only when a component depends on it.
Use	Is used for non-configurable services. It can also be used as a service for replacing complex logic. Go for it if you are using an object.	Is used for inserting simple logic. Go for it if you are using a class.
Properties	Are defined without `this` keyword.	Are defined with `this` keyword.
Friendly Injections	Are not supported.	Are supported.
Primitives	Are created.	Are not created.
Preferable Choice	Is more preferable due to its class-like definition.	Is preferred only for defining utility services or using ES6 classes.

Table 7.1: Differences between Factory and Service

7.5 AngularJS Dynamic Templates

There are situations when common Web development patterns cannot fulfill some specific page requirements efficiently in terms of view. Let us assume that an SPA has a route to list all cloud services for which a logged-in user is authorized. Each of these services has its own functionality such as unauthorize and change settings. In this scenario, there is a need to properly implement this view. Thus, the question arises: How can this be done? The developer might decide to design the template, as shown in Code Snippet 10.

Code Snippet 10:

```
<div ng-controller="serviceControl">
    <section ng-show="Service1">
        <div ng-click="changeSettingsToServiceOne">Service 1 Content</div>
    </section>
    <section ng-show="Service2">
        <div ng-click="changeSettingsToServiceTwo">Service 2 Content</div>
    </section>
    <section ng-show="Service3">
        <div ng-click="changeSettingsToServiceThree">Service 3 Content
        </div>
</section>
</div>
```

Using this template, the developer can hide or show the desired sections depending on whether the user is authorized or not. However, this approach has two downsides. First, the controller must implement the functionality for all services even if the user is unauthorized for all of them. Second, the template does not fulfill the requirement of showing the registered services dynamically or in a desired order.

Hence, in this scenario, it is essential to create a ***dynamic template***. To do so, the developer should use custom directives per service and then, add it dynamically to the template only in case of a registered user. This pattern allows adding each service in a dynamic order, if these details are from an API. Further, each directive wraps its own functionality and that too, in the corresponding controller and template files.

In AngularJS, custom directives help in extending the HTML functionality. They tend to replace the

elements for which they are initiated. At the time of loading, an AngularJS application finds the linked elements, invokes the `compile()` function of the directive once, and processes the elements via the directive's `link()` function as per its scope.

An AngularJS developer should define these directives using the `directive` function. AngularJS allows defining custom directives for elements, attributes, CSS styles, and comments.

Code Snippets 11 and 12 demonstrate how to define a custom directive in an AngularJS application.

Code Snippet 11: index.html

```
<!DOCTYPE html>
<html ng-app="dynamictemp">
  <head>
   <meta charset="utf-8" />
   <title>AngularJS Dynamic Template</title>
   <script>document.write('<base href="' + document.location + '" />');
   </script>
    <script src =
   "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
    </script>
    <script src="app.js"></script>
  </head>
  <body ng-controller="MainCtrl">
    <select ng-model="model.loanType">
      <option value="">Select Loan Type</option>
      <option value="1">Personal Loan</option>
      <option value="2">Housing Loan</option>
    </select>
      <loan-detail-form loan="model"/>
      <br/>
      <br/>
    <script type="text/ng-template" id="template1">
      <form>
     <br/>
     <br/>
        <fieldset>
          <legend>Personal Loan</legend>
          <label>Document needed:</label>
          <input type="text" ng-model="loan.attributeA">
        </fieldset>
     </form>
    </script>
 <!--Design the look as an enclosing box with elements inside-->
    <script type="text/ng-template" id="template2">
      <form>
      <br/>
      <br/>
        <fieldset>
          <legend>Housing Loan</legend>
          <label>Document needed:</label>
          <input type="text" ng-model="loan.attributeB" readonly><br/>
          <label>Document needed:</label>
          <input type="text" ng-model="loan.attributeC" readonly>
        </fieldset>
      </form>
    </script>
  </body>
</html>
```

Code Snippet 12: app.js

```
var app = angular.module('dynamictemp', []);
app.directive('loanDetailForm', function($templateCache, $compile) {
  var getTemplate = function(loanType) {
```

```
    var t;
    switch (loanType) {
      case "1":
        t = $templateCache.get("template1");
        break;
      case "2":
        t = $templateCache.get("template2");
        break;
      default:
        t = "<div>No loan type selected</div>";
    }
    return t;
  }
  return {
    restrict: 'E',
    scope: {
      loan: "="
    },
    link: function(scope, elem, attrs) {
      scope.$watch('loan.loanType', function(newVal){
        elem.html(getTemplate(newVal));
        $compile(elem.contents())(scope);
      });
    }
  }
})
app.controller('MainCtrl', function($scope) {
  $scope.model = {
    loanType: "1",
    attributeA: "Photo ID Documents",
    attributeB: "Address Proof",
    attributeC: "Salary Slip"
  }
});
```

Through the code in Code Snippets 11 and 12, a dynamic template is defined to display different kinds of documents required based on a given loan type. The end user chooses a loan type from a drop-down and then, based on it, the template is dynamically rendered. The key lines of importance in the code are highlighted in yellow. `ng-template` is used to render the template.

You can think of directives as indicators on DOM elements (such as an attribute, element name, and so on) that will tell AngularJS's **HTML compiler** (represented as `$compile`) to attach a specified behavior to that DOM element or transform the DOM element and its child elements. In AngularJS, 'compilation' means attaching directives to the HTML markup for the purpose of making it interactive.

The term 'compile' is used because it resembles the process of compiling source code in several programming languages.

In general, the `$compile` service can match directives based on element names (E), attributes (A), class names (C), and comments (M). Here, in our code in ***app.js***, `$compile` is used to match directives based on element names (E).

The output of this application is shown in Figure 7.8. Here, it is assumed that the end user has chosen Housing Loan from the drop-down.

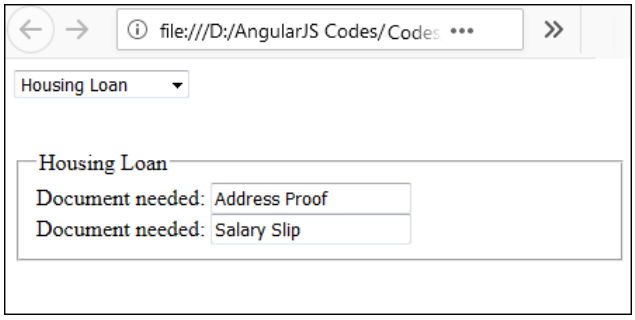


Figure 7.8: Using Custom Directives

7.6 Building SPAs in AngularJS

An SPA refers to an application that adjusts to fit within a single Web page. A single page load tends to retrieve all its code, including CSS and JavaScript. Following are the pros of using an SPA in AngularJS:

- **No Full-page Refresh**: A user navigates between pages without refreshing the whole page. Only the required section of the page that needs a change is re-loaded. AngularJS enables pre-loading and caching all pages due to which there is no need of additional requests to download them.
- **Offline Functioning**: An SPA allows navigating through its sections even if the Internet connection is suddenly lost. This is because AngularJS allows pre-loading all relevant pages.
- **Improved User Experience**: An SPA gives a feel of a desktop application, which is fast and responsive.

While there are appealing pros, SPAs also come with some limitations. Following are some limitations:

- **Complex Development**: A developer needs to do lot of JavaScript coding to manage different entities and functionalities such as permissions and shared state across pages.
- **Additional Search Engine Optimization (SEO) Care**: It is essential for a search engine crawler to execute JavaScript for indexing the built SPA application. Recently, Bing and Google began executing JavaScript for indexing Asynchronous JavaScript and XML (AJAX) pages. This means that you would need static HTML views for search engines that have not started executing JavaScript.
- **Slow Initial Load**: At the time of opening or loading for the first time, an SPA downloads more resources. This makes the initial loading quite slow.
- **Client-side JavaScript Disabled Possibility**: It is necessary for a client to enable JavaScript in its browser. This is because SPAs need it. If the client-side browser has JavaScript disabled then, the SPA will not work. Fortunately, all modern browsers have JavaScript enabled.

7.6.1 Building Single Page Applications

Each AngularJS application is begun by designing a module, which is a container holding different components of an application such as controllers and services. Let us begin with a simple SPA that displays a home page, an about page, and a tutorial page based on respective clicks. Create a folder named Simple SPA under htdocs folder of xampp path. Under this folder, create codes as given in Code Snippets 13 to 17 with the file names that are specified.

Code Snippet 13: index.html

```
<!doctype html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Single Page Application Using AngularJS</title>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
</script>
```

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular-
route.js"></script>
<script src="script.js"></script>
</head>
<body ng-app="single-page-app">
  <div ng-controller="SPAController">
  <div>
         <ul>
        <li ><a href="#/">Home<span> (current page)</span></a></li>
        <li><a href="#!about">About us</a></li>
        <li><a href="#!tutorial">Tutorial</a></li>
      </ul>
</div>
 <div style="background-color: lightgreen" ng-view></div>
<br/>
<br/>
<br/>
<hr/>
<div style="padding-left:50px;padding-right:100px;">
{{message}}
</div>
  </div>
</body>
</html>
```

Code Snippet 14: about.html

```
<div style="padding-left:50px;padding-right:100px;">
<h1>About us Page</h1>
 <p>This is simple AngularJS ng-view and ngRoute tutorial to demonstrate
single page Web application development.</p>
 </div>
```

Code Snippet 15: home.html

```
<div style="padding-left:50px;padding-right:100px;">
   <h1>Home Page</h1>
   This is a Simple SPA Demo
 </div>
```

Code Snippet 16: tutorial.html

```
<div style="padding-left:50px;padding-right:100px;">
<h1>Tutorial Page</h1>
 <p>This is basic simple SPA tutorial</p>
 </div>
```

Code Snippet 17: script.js

```
var app=angular.module('single-page-app',['ngRoute']);
app.config(function($routeProvider){
      $routeProvider
          .when('/',{
                templateUrl: 'home.html'
          })
          .when('/about',{
                templateUrl: 'about.html'
          })
              .when('/tutorial',{
                templateUrl: 'tutorial.html'
          });
});
app.controller('SPAController',function($scope){
      $scope.message="Single Page Application"; });
```

Through these code snippets, we utilized the routing capabilities of AngularJS to make an SPA. We do so by using the built-in `ngRoute` module that not only offers routing, but also directives and deep-linking services.

Following are the steps to use this module:

1. Include **angular-route.min.js** file after **angular.min.js**, which is the main AngularJS library.
2. State that the newly defined module is reliant on the `ngRoute` module for linking and routing.
3. Separate the common HTML code for each page that acts as the site's layout.
4. Specify where HTML code of each page shall be added in the layout by using the `ng-view` directive. This AngularJS directive includes the current route's template in the main file of layout. For example, the current route can be /about or /tutorial. It injects the stated file for current route into the layout where its code exists.

When the browser is launched with address *http://localhost:84/Simple SPA*, the output will be as shown in Figure 7.9.

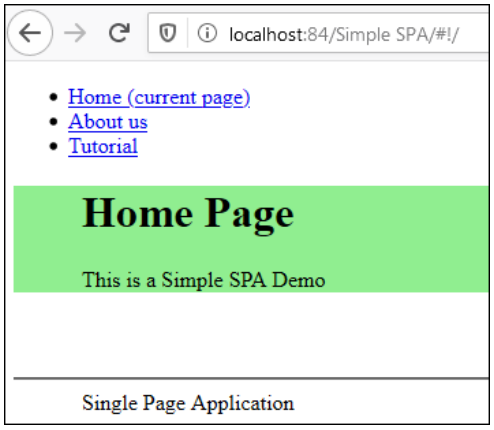


Figure 7.9: Launching SPA

When the 'About us' hyperlink is clicked, the output will be as shown in Figure 7.10.

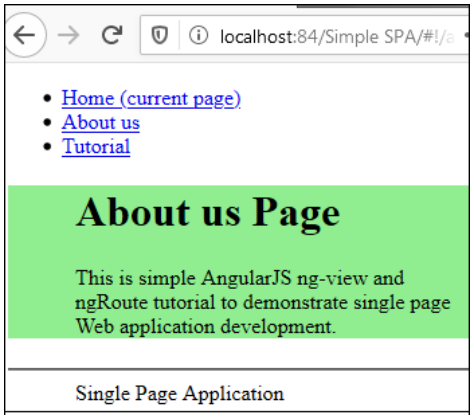


Figure 7.10: About Us Page

As you can observe, the initial part and bottom part of the page showing the hyperlink remains unchanged and only the area in green representing the About us section is dynamically rendered. Thus, we are avoiding the reload of the entire page by means of SPA logic. This has happened because of the following lines of code in the markup and the script:

Markup:

```
. . .
<li><a href="#!about">About us</a></li>
<li><a href="#!tutorial">Tutorial</a></li>
. . .
<div ng-view></div>
```

Script:

```
.when('/about',{
          templateURl: 'about.html'
})
```

In these lines of code, the route is specified for the 'About Us' hyperlink and the `ng-view` directive will render the view using the content of *about.html*. The `app.config(function($routeProvider)` statement will map the hyperlink against each `templateUrl`.

Similar logic is used to display the content of the tutorial page. Refer to Figure 7.11.

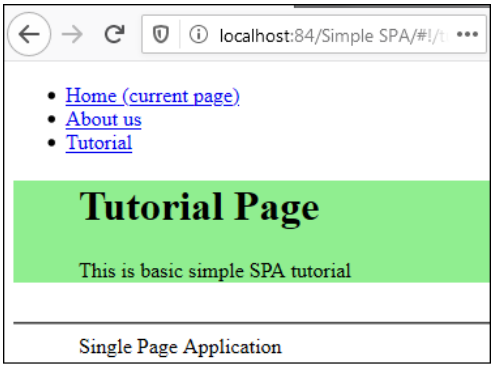
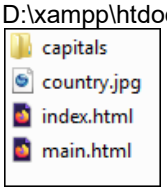


Figure 7.11: Tutorial Page

Consider another example of an SPA where we wish to display the capitals of three countries along with respective images. In this case, we shall define separate controllers for each of the three pages. The application structure is shown in Figure 7.12.

Folder: D:\xampp\htdocs\SPA



Folder: D:\xampp\htdocs\SPA\capitals

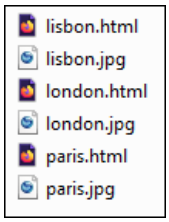


Figure 7.12: Folder Structure for the SPA

Code Snippet 18: index.html

```
<!DOCTYPE html>
<html>
<script src =
"https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
</script>
<script src =
  "https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular-
route.js">
</script>
<body ng-app="myApp">
<a href="#!london">London</a>
```

```
<a href="#!paris">Paris</a>
<a href="#!lisbon">Lisbon</a>
<p>Click each link for a description.</p>
<div ng-view>
</div>
<script>
var app = angular.module("myApp", ["ngRoute"]);
app.config(function($routeProvider) {
    $routeProvider
    .when("/", {
        templateUrl : "main.html",
    })
    .when("/london", {
        templateUrl : "capitals/london.html",
        controller : "londonCtrl"
    })
    .when("/paris", {
        templateUrl : "capitals/paris.html",
        controller : "parisCtrl"
    })
      .when("/lisbon", {
        templateUrl : "capitals/lisbon.html",
        controller : "lisbonCtrl"
    })
      .otherwise({redirectTO:'/'});
});
app.controller("londonCtrl", function ($scope) {
    $scope.msg = "Did you know? Heart London is a regional radio station of
                  London";
});
app.controller("parisCtrl", function ($scope) {
    $scope.msg = "Eiffel Tower in Paris is one of the most popular
                 landmarks of love";
});
app.controller("lisbonCtrl", function ($scope) {
    $scope.msg = "Lisbon is one of the oldest cities in the world";
});
</script>
</body>
</html>
```

In Code Snippet 18, we included *angular.min.js* and *app.js* files and then, defined the name of module as `myApp` through the `ng-app` directive. In this Code Snippet, the script code is included inline instead of a separate file.

We then use `ngRoute`, specify the `templateUrl` and `controller` values for each route, and create the controllers specified for each route. The `otherwise` function handles a request to the route that does not exist. In that case, the code redirects the user to the `"/"` route.

Finally, it is time to create the routing pages. Refer to Code Snippets 19 to 22.

Code Snippet 19: main.html

```
<html>
<h1>
Countries and their Capitals<br>
<img src="country.jpg"></img>
</h1>
</html>
```

Code Snippet 20: london.html

```
<h1>London</h1>
<h3>London is the capital city of England.</h3>
<p>It is the most populous city in the United Kingdom, with a metropolitan
area of over 13 million inhabitants.</p>
<img src="capitals/london.jpg"></img>
<p>{{msg}}</p>
```

Code Snippet 21: paris.html

```
<h1>Paris</h1>
<h3>Paris is the capital city of France.</h3>
<p>The Paris area is one of the largest population centers in Europe, with
more than 12 million inhabitants.</p>
<img src="capitals/paris.jpg"></img>
<p>{{msg}}</p>
```

Code Snippet 22: lisbon.html

```
<h1>Lisbon</h1>
<h3>Lisbon is the capital city of Portugal.</h3>
<p>Lisbon is the capital and the largest city of Portugal, with an
estimated population of 505,526.</p>
<img src="capitals/lisbon.jpg"></img>
<p>{{msg}}</p>
```

Figure 7.13 shows the output of the application.

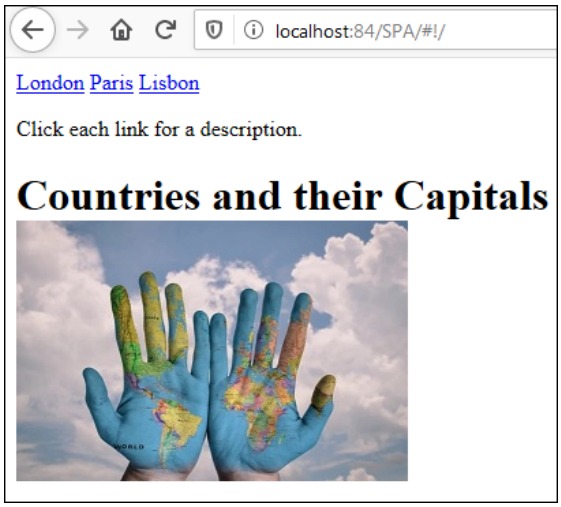


Figure 7.13: Output of SPA

Clicking each of the hyperlinks will display the corresponding page. For example, Figure 7.14 shows the page for London.

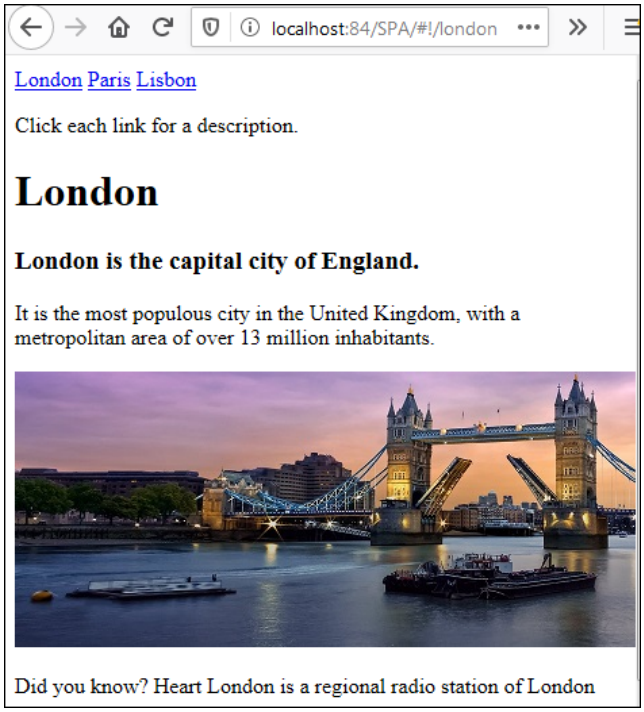


Figure 7.14: Displaying Content for London

Quick Test 7.1

1. Dependency injection makes an AngularJS application code re-usable.
   a. True
   b. False

2. The `factory()` function initializes a service object.
   a. True
   b. False

7.7 Summary

- Dependency injection is a pattern or technique for adding a dependent functionality into a module at the time of execution without coding for it.
- The benefits of dependency injection include no hard-coded dependencies, modularized applications, easy configurations and code changes, reusable modules, and mock testing of applications.
- AngularJS allow injecting values, providers, constants, factories, and services into each other as dependencies.
- Values are injected into a factory, controller, or a service.
- All the different ways of creating a service in AngularJS ultimately use `$provide`.
- The `config()` function accepts only a provider of a service or a constant as a parameter.
- A factory uses a function that returns a value, while a service is a constructor function, which uses the `new` keyword for creating an object and adding functions and properties to it.
- Factories and services are providers.
- Creating a dynamic template involves using custom directives per service such that the services are added in a random order.

7.8 Exercise

1. Which of the following components can be injected as a dependency in AngularJS?
   a) Module
   b) Integer
   c) Function
   d) Object

2. Which of the following methods does a provider internally invoke?

   a) POST
   b) GET
   c) PUT
   d) CREATE

3. Which of the following components does the `config()` function not accept as a parameter?

   a) Constant
   b) Service
   c) Provider
   d) None of these

4. A service uses the _____ keyword to define properties.

   a) `then`
   b) `new`
   c) `this`
   d) `when`

5. Which of the following is a limitation of an SPA?

   a) No working offline
   b) Full page reloading
   c) Reduced speed of initial load
   d) Not so responsive initially

7.9 Do It Yourself

1. Create an AngularJS application that defines a factory to calculate and return sum of two given numbers and a service that returns a message “Happy New Year” and invoke the factory and service.

2. Build an AngularJS SPA for an Automobile Rental Agency. The landing page should contain a welcome message. A horizontal menu should display following options:
   - About Us
   - Cars for Rent
   - Contact Us

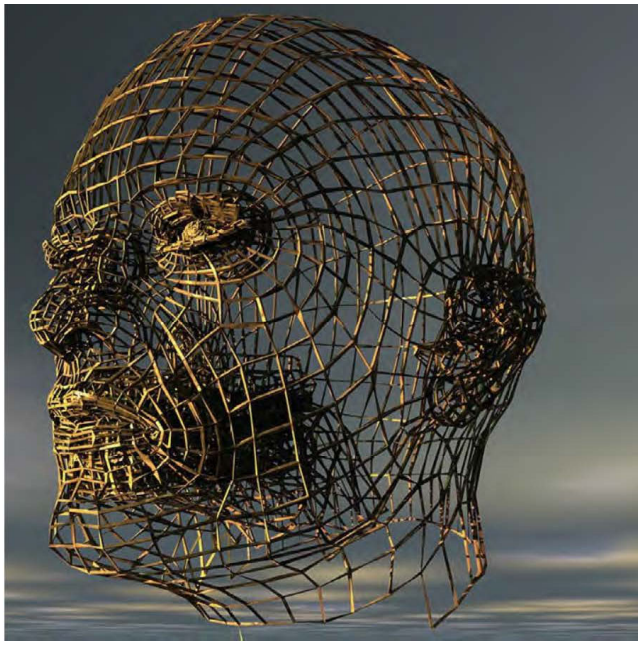
   On clicking:

   - About Us, a two-line paragraph should be displayed containing information about the agency.
   - Cars for Rent, some car images should be displayed with pricing (use fictitious data).
   - Contact Us, a phone number and email id should be displayed (use fictitious data).

Onlinevarsity

TECHNO-WISE

THE TECHNOLOGIES OF TOMORROW



Answers to Exercise

1. Module
2. GET
3. Service
4. `this`
5. Reduced speed of initial load

Answers to Quick Test

Quick Test 7.1

1. True
2. False

Session 8

Getting to Know Angular 9

In this session, students will learn to:

- List features and enhancements in Angular 9
- Explain Angular Architecture
- Outline steps to create an Angular 9 application
- Identify the use of various types of files in an Angular application
- Define Pipes and learn their usage

8.1 Introduction to Angular 9

Angular is a complete rewrite of the AngularJS Web framework. It is TypeScript-based and open-source. Angular 9 was released on February 6, 2020. In this version, all applications must use Angular's next-generation compiler and runtime by default. This compiler has been code-named Ivy. Angular has been updated to work with TypeScript 3.6 and 3.7. Figure 8.1 depicts major features and enhancements in Angular 9.

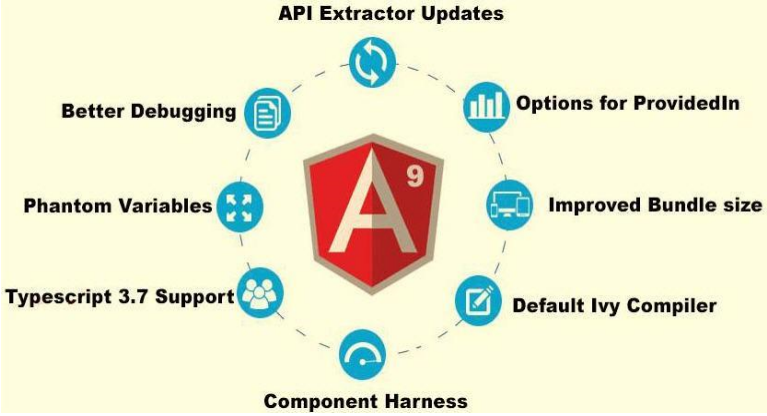


Figure 8.1: Major Features and Enhancements of Angular 9

1. **Default Ivy Compiler**

   In Angular, whenever you write a component, we write it in TypeScript and its corresponding template in HTML, which is then supplemented by Angular template syntax (such as `ngIf`). It is the compiler in Angular that turns the template into code for the Angular runtime. In other words, the HTML will be compiled by Angular into JavaScript instructions, to create or update the relevant DOM when the component appears on the page or its state changes.

   In earlier versions of Angular, View Engine was the default compiler. It had drawbacks such as large bundle sizes, slow performance, lack of type checking, and so on. The Angular team keeps changing the compiler and renderer. During their fourth rewrite of the engine, they code-named it Ivy. In Angular 9, Ivy is enabled by default. All applications created with Angular 9 are intended to use Ivy renderer as the default compiler. Apps built with Ivy are more efficient.

   Generally, in Angular applications, compilers can work in Just in Time (JiT) mode wherein it is delivered along with the application and compiles at runtime. Ahead of Time (AoT) compilation on the other hand compiles everything at build time making applications quicker. It does not require the compiler to be shipped with the application. Angular 9 enables compilation to be faster because of Ivy. In Angular 9, AoT compilation happens by default.

Following are key points to note regarding Ivy:

Tree shaking and Locality are two important terminologies with respect to Ivy.

Tree shaking means removing unused blocks of code during the bundling process. The analogy of 'tree shaking' refers to the process of physical shaking of a tree to cause dead leaves to fall off a tree. This leaves the tree clean. Tree shaking is used in Angular (in which applications and their dependencies too have a tree-like structure) to ensure clean and lean code. By using tree shaking, our application will only include that code required for the application to run. Note that it is not specific to Angular alone.

In the earlier View Engine, to compile a component, Angular will need information about all its declarable dependencies, in turn their dependencies, and so on. In Ivy, however, each component is independent. Locality refers to the process of compiling each component with only information about the component itself, except for the name and package name of its declarable dependencies. This helps to rebuild faster by compiling partial changes and not the entire project files. Ultimately, increasing the speed of your build process.

2. **Improves the bundle sizes**

As of today, the number of smartphone users worldwide is over three billion and is expected to grow by several hundred million in the next few years. Refer to Figure 8.2.

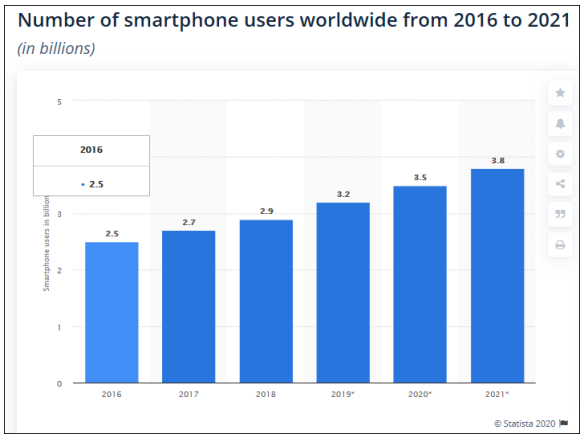


Figure 8.2: Smartphone User Statistics

Mobile Internet has grown 504% in daily media consumption since 2011. Many of the users reside in places where there are slow Internet connections. Hence, it is important for developers to redesign and decrease the size of downloadable files, for a better mobile user experience. Angular 9 supports this by enabling developers to reduce bundle sizes by 25-40 percent based on the app size.

Small apps benefit from the tree shaking feature of Ivy and Angular 9 support, as they have to generate a lesser amount of code for Angular components. Smaller bundle means better performance and speed.

3. **Better Debugging**
   Angular 9 and Ivy compiler help in better debugging by providing access to instances of the components and directives, triggering change detection with `applyChanges`, and more. They enable developers to manually call methods and update state.

4. **Phantom Template Variable Menace**
   Variables that are never referenced or defined in a template's associated component are called phantom template variables. Creating phantom variables can present danger to your applications in the long run. They should be avoided. In Angular 9, you will now get a compiler error when you create such phantom template variables.

5. **API Extractor Updates**
   There are several services and libraries that Angular requires in order to work properly. These are often difficult to maintain and update. Also, these APIs (libraries and services) may evolve and grow separately, which may be impossible for Angular users to track all the time. In Angular 9, the required libraries are tracked and updated using Bazel. Bazel is Google's open-source part of its internal build tool called Blaze. Bazel supports building automation and software testing. Bazel in turn references API Extractor, which is a tool invoked at build time by Angular. It leverages TypeScript compiler engine to detect a project's exported API surface, produces reports, finds out missing updates or new features, and documents them in a way that they are communicated easily.

6. **New options for 'providedIn'**
   A provider refers to an instruction to the Dependency Injection system regarding how to obtain a value for a dependency, which is usually a service that a developer has created. The `@Injectable()` decorator in Angular, by default, has a `providedIn` property, which creates a provider for the service. From Angular 6 onwards, it is recommended to create a singleton service by setting `providedIn` property to `root` on the service's `@Injectable()` decorator. Angular is thus instructed to provide the service in the application root.

   In Angular 9, the `providedIn` property has some additional options as follows:

`platform`	`any`
Makes the service available in a special singleton platform injector that is shared by all applications on the page. `providedIn`: 'platform' can be used for sharing services over application boundaries, such as Angular Elements. An Angular Element is an Angular component packaged as a custom element.	Provides a unique instance in every module that introduces the token. `providedIn`: 'any' can be used to make sure a service is a singleton within module boundaries.

7. **Component Harness**
   A component harness is a class that allows unit tests to interact with components through supported APIs. The API of each harness engages with a component in the same way that a user would. Through the harness API, a test protects itself against updates to the internals of a component, which would have affected it adversely. You can think of it like how a safety harness protects a construction worker on a scaffold. Angular 9 is making harnesses available to any component author as part of the Component Dev Kit (CDK).

8. **TypeScript 3.7 Support**
   TypeScript is an open-source programming language designed for building large applications and transcompiles to JavaScript. It is a superset of JavaScript and builds on it by adding syntax for type declarations. This syntax is used by TypeScript compiler to type-check code, and then, produce readable JavaScript which can run on different runtimes. Angular 9 is updated to work with TypeScript 3.6 and 3.7.

With increasing number of developers and projects using Angular, over 300 bugs have been reported and fixed, making it robust. Angular is used internally at Google for projects such as Google Domains and Firebase.

8.2 Angular 9 Architecture

Figure 8.3 depicts the architecture of Angular 9.

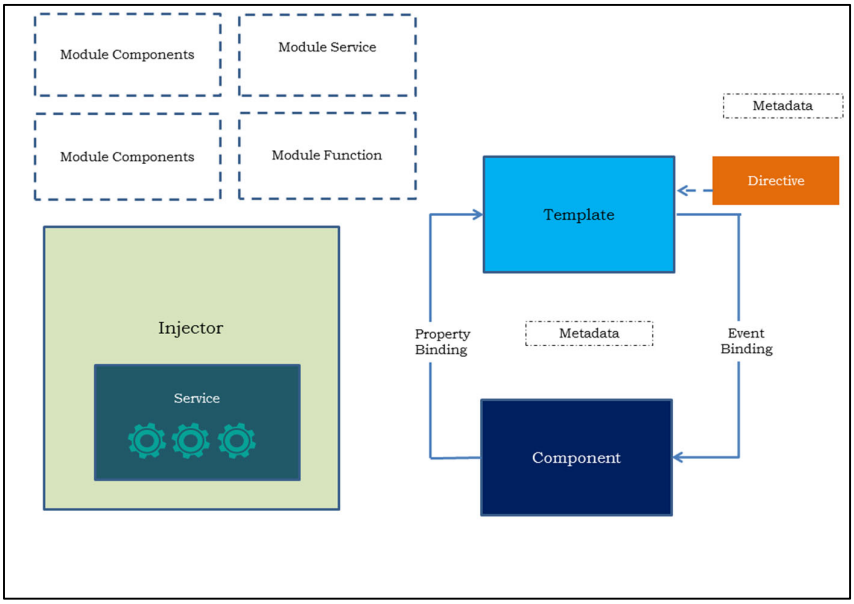


Figure 8.3: Angular 9 Architecture

Key components of the architecture are as follows:

Modules

In Angular 9, apps are modular. Angular has its own modularity system called `NgModules`. NgModules are containers for a consolidated block of code and can contain components, service providers, and other code files whose scope is defined by the enclosing `NgModule`.

Angular 9 apps comprise a root module called `AppModule` that provides bootstrap mechanism to launch an application. In addition to this, an application can contain several functional modules.

Components

In Angular 9, component-based architecture allows to use components to compose applications. Each component defines a class containing feature application data and logic. Additionally, it is associated with an HTML template defining a view. Angular applications have at least one root component (typically called `AppComponent`) that maps a page hierarchy with page DOM. A component controls a part of the screen of your application.

Template, Directives, and Data Binding

Similar to AngularJS, templates in Angular 9 combine HTML with Angular markup and modify HTML elements before displaying them. Template directives are used for program logic. Data binding markup elements connect application data and DOM. Event binding and Property binding are the two types. The former binds events to your app and responds to user input by updating application data. The latter is used to pass data from component class and facilitates interpolating values that are computed from application data into the HTML.

Services and Dependency Injection

In simple terms, services are used to organize and share code across apps. Dependency injection is a technique to pass a dependent object into another object to make all functionality of former available to latter. Using DI, you can inject a service into a component, giving the component access to that service class.

8.3 Creating Angular 9 Applications

Let us now explore how to create an Angular 9 application. First, though, you need to understand some terminologies.

- **Angular CLI**

  Angular Command-Line Interface (CLI) is the command-based interface that helps you build Angular applications, and is considered the official tool to work with Angular projects. Using CLI, the challenges and overhead of complex configurations and need for build tools such as TypeScript, Webpack, and so on is eliminated.

- **Node.js**

  Similar to many other modern front-end tools available nowadays, Angular CLI is built on top of Node.js, which is a server technology. Node.js is open-source, cross-platform, and provides a runtime environment to run JavaScript on the server and build server-side Web applications.

- **npm**

  npm stands for Node Package Manager and is a package manager program for JavaScript. It is the default package manager for Node.js. It comprises:
  - command line client, also called npm
  - npm registry, an online database of packages

Angular being a front-end technology, you only need to install Node.js on your local system for running the CLI and for installing packages from npm.

- **Step 1 - Install Node and npm**

Visit the link https://nodejs.org/en/download/ to download and install Node.js. Follow the setup wizard instructions.

- **Step 2 - Install Angular CLI 9**

Open Command Prompt and give the following command to install Angular 9 CLI tool:

```
npm install -g @angular/cli
```

Refer to Figure 8.4.

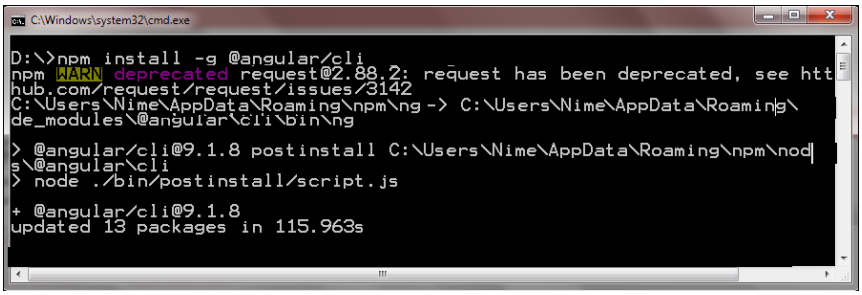


Figure 8.4: Installing Angular CLI

- **Step 3 - Verify installation**

Once it is installed successfully, verify the version with the following command:

```
ng --version
```

It will display the details of the version installed. Refer to Figure 8.5.

Figure 8.5: Verifying Angular CLI Version

- **Step 4 - Initialize a new Angular 9 Project**

```
ng new first-ang9-project
```

You will be prompted whether you want to use routing. At this stage, you do not require to add routing since this is your first Angular 9 application. Hence, you type n (or N). Refer to Figure 8.6.

Figure 8.6: Creating a New Project

Next, you will be prompted to select the type of stylesheet you want. For the time being, select CSS by pressing Enter key on the first option. Refer to Figure 8.7.

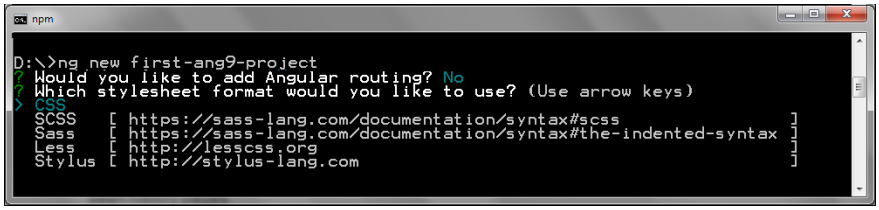


Figure 8.7: Selecting Stylesheet

Once this is done, the project creation will begin. Relevant node modules and libraries for a default project will be installed in a folder that will bear the same name that you specify with the `ng new` command. All the files required to make it a ready- to-execute application will be installed. Refer to Figure 8.8.

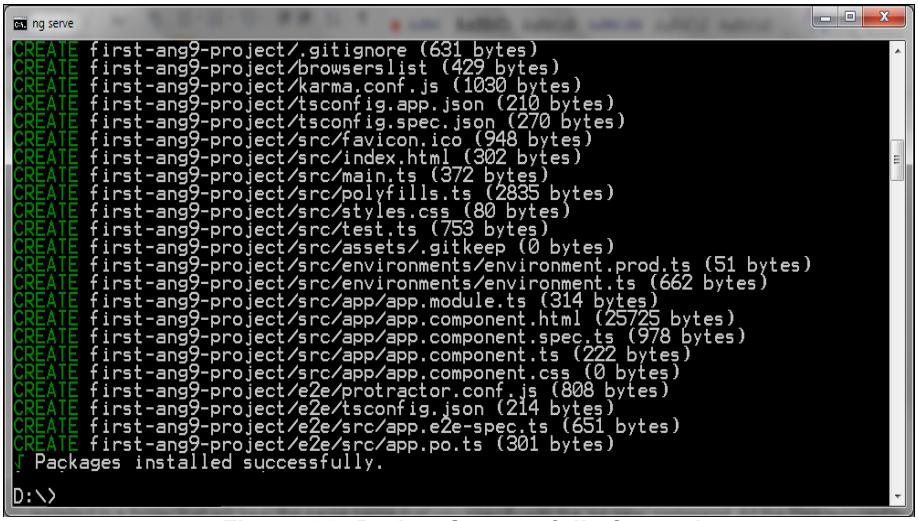


Figure 8.8: Project Successfully Created

- **Step 5 - Serve the Project**

After the project has been created, change the current working directory to the project directory and then, type the following command:

```
ng serve
```

Though there is a command `ng build`, it only builds your app and deploys it. `ng serve` on the other hand builds, deploys, serves, and continually watches your code for changes. When any changes are found in code, it builds and serves the code automatically.

Let us now serve the **first-ang9-project** application. Refer to Figure 8.9. Note that this command must be given from within the project directory, hence, a `cd` command is used first.

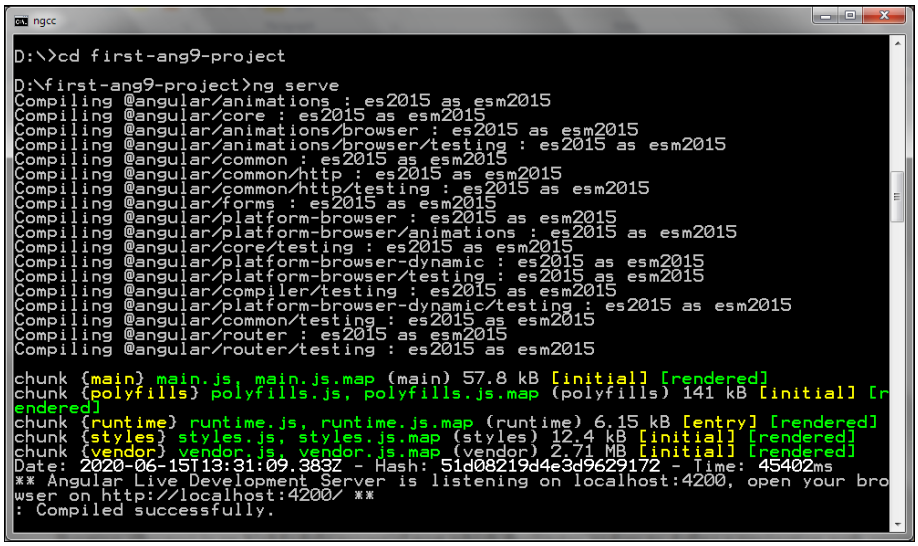


Figure 8.9: Serving the Project as a Web Application

Now, open a browser window and type http://localhost:4200.

The newly created project will be hosted on the local server. Refer to Figure 8.10.

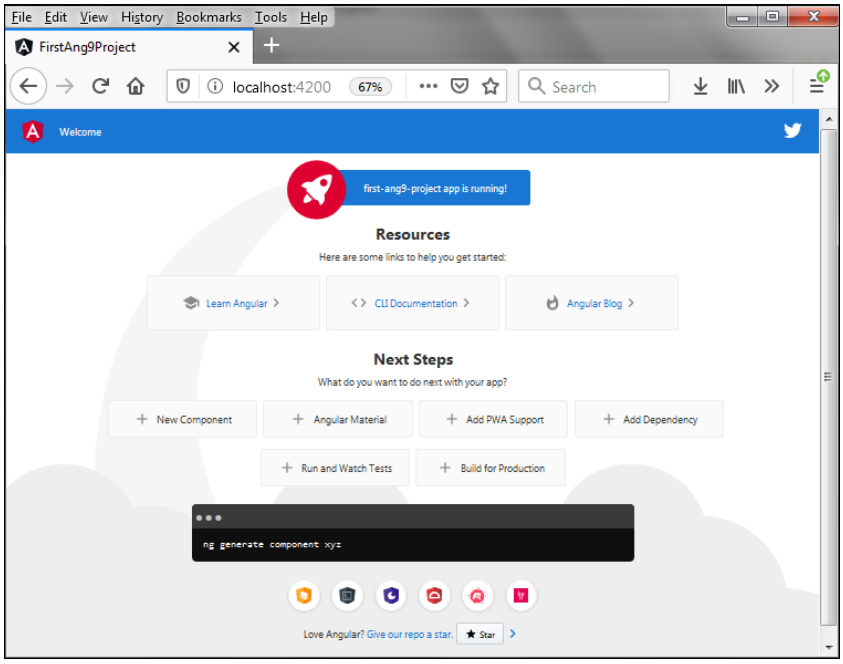


Figure 8.10: Viewing the Web Application in Browser

Notice how little effort we had to put in to create this Web application. Angular 9 and Angular CLI took care of all the advanced behind-the-scenes tasks. Open the project folders and observe the files generated. Refer to Figures 8.11 and 8.12 respectively.

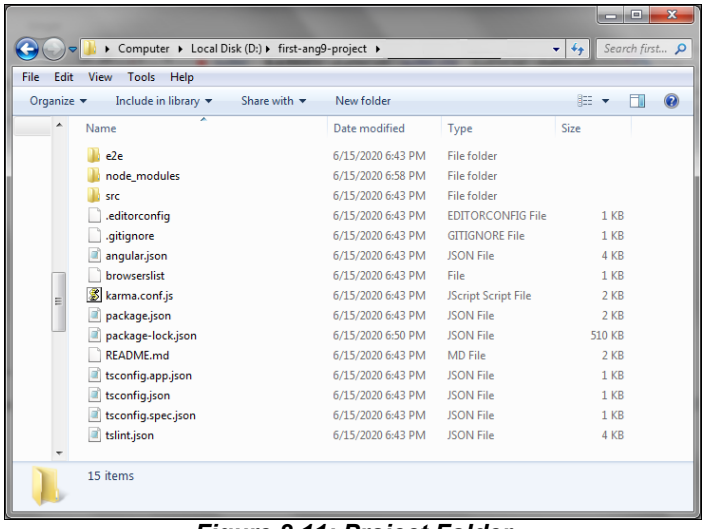


Figure 8.11: Project Folder

Navigate to the src\app sub folder to view the source files generated. You will see that there is one HTML file and few TypeScript files.

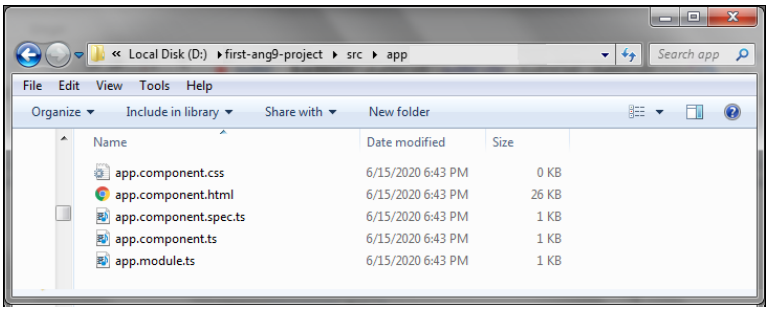


Figure 8.12: Src Project Sub-Folder

Table 8.1 outlines the purpose of these auto-generated files.

File/Folder Name	Purpose
/e2e/	Is a folder containing end-to-end tests of the Website
/node_modules/	Is a folder containing all third party libraries installations through `npm install`
/src/	Is a folder containing the source code of the application. Most work will be done here
/app/	Is a folder containing modules and components
/assets/	Is a folder containing static assets such as images, icons, and styles
/environments/	Is a folder containing environment (production and development) specific configuration files
browserslist	Is a file required by autoprefixer for CSS support
favicon.ico	Is a file that represents the favicon
index.html	Is the main HTML file
karma.conf.js	Is the configuration file for Karma (a testing tool)
main.ts	Is the main starting file from where the *AppModule* is bootstrapped
styles.css	Is the global stylesheet file for the project
test.ts	Is a configuration file for Karma
tsconfig.*.json	Is the configuration file for TypeScript
angular.json	Contains the configuration for CLI
package.json	Contains the basic information of the project (name, description, and dependencies)
README.md	Is a markdown file that contains a description of the project
tsconfig.json	Is the configuration file for TypeScript
tslint.json	Is the configuration file for TSlint (a static analysis tool)

Table 8.1: Different Files in an Angular 9 Application

Now, check the auto-generated code in some of the key files under src and its sub-folders.

index.html under src folder

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>FirstAng9Project</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

app.module.ts under src\app

This is the file that contains module definition.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

app.components.ts under src\app

This is the file that contains components definitions.

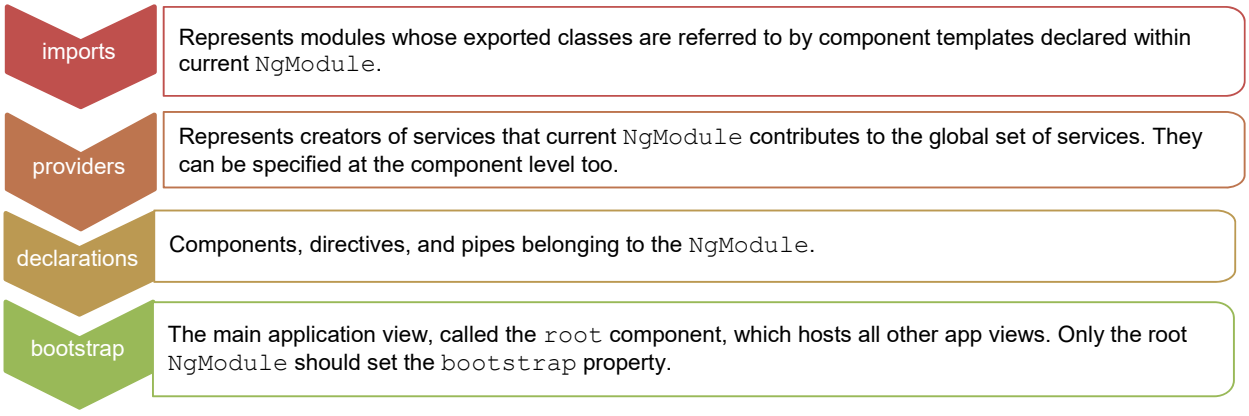
```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'first-ang9-project';
}
```

Let us now understand different parts that constitute an Angular 9 application.

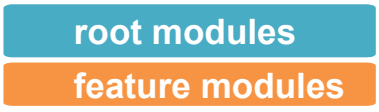
A module is a means by which one can group components, directives, services, and so on that together form an application. Each of these artifacts behaves like a part of a jigsaw puzzle, each has its own significance and role to play. The resulting application is like a completed puzzle.

To define a module, one should apply the `@NgModule()` decorator to a class. The decorator is a function that takes a single metadata object, with properties describing the module.

Some of the key properties of `NgModule` are as follows:



There are two types of modules:



In an application, you can have one root module and zero or many feature modules. Further, in a module, you can have one root component and many possible secondary components. You must inform Angular which module in your application is the root module, in order to bootstrap your application. In an existing code, one can observe the imports property of `NgModule` decorator in the application to learn which one is the root module.

Code Snippet 1 shows the code to define a component and then, Code Snippet 2 shows the code to define a basic module containing only that component and a root `NgModule` definition.

Code Snippet 1: ***app/app.component.ts***

```
1. import { Component } from '@angular/core';

2. @Component({
3. selector: 'app-root',
4. template: '<h3>Hello World, This is an Angular App</h3>'
5. })

6. export class AppComponent {}
```

This code creates a single component that will then be used in a module, as shown in Code Snippet 2. A component and its template together define a view. A template tells Angular how to render the component. Views are often structured in a hierarchical manner, so that entire UI sections or pages can be shown or hidden as a unit. Application logic for a component and thereby the view, is defined within a class, which then interacts with the view through properties and methods. Throughout the lifecycle of the application, Angular will continue to create, update, and destroy components.

In Code Snippet 1, the most basic view is defined wherein we just display a heading, Hello World, This is an Angular App. Line 2 shows how a component is defined with the `@Component` decorator. Lines 3 and 4 also specify metadata for the component and inform Angular that this is the root component. The template for the view is given on Line 4.

Code Snippet 2 utilizes this component.

Code Snippet 2: ***app/app.module.ts***

```
1. import { NgModule } from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { AppComponent } from './app.component';

4. @NgModule({
5. imports: [BrowserModule],
6. declarations: [AppComponent],
7. bootstrap: [AppComponent]
8. })
9. export class AppModule { }
```

In Line 2 of Code Snippet 2, the module is importing `BrowserModule` as an explicit dependency. `BrowserModule` is a built-in module that exports basic directives, services, and so on.

`AppComponent` is the root component in our module, hence, we have to list it in the `bootstrap` array as seen in Line 6.

In Line 7, we define `AppComponent` again in the declarations property because here, we are supposed to define all components that constitute our application.

8.4 Generating Angular Items

Using `ng generate` command of Angular CLI, developers can generate basic Angular items such as modules, components, directives, and services. For example,

```
ng generate component ang9-component
```

Here, **ang9-component** is the name of the component. Refer to Figure 8.13.

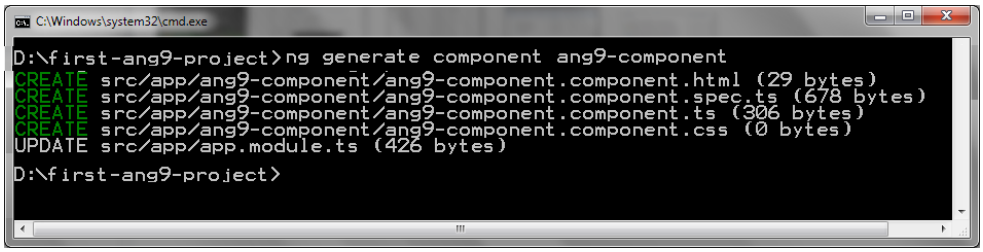


Figure 8.13: Generating a New Component

Angular CLI will create the additional component and automatically add a reference to components, directives, and so on in the ***src/app.module.ts*** file.

Refer to Figure 8.14.

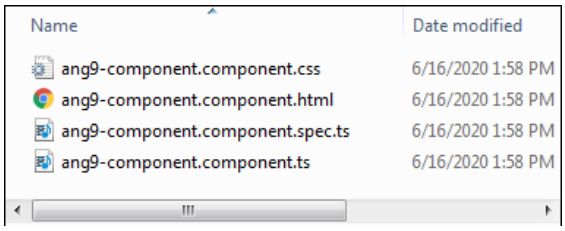


Figure 8.14: Component Files

Auto-generated code present inside *ang9-component.component.ts*:

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-ang9-component',
  templateUrl: './ang9-component.component.html',
  styleUrls: ['./ang9-component.component.css']
})
export class Ang9ComponentComponent implements OnInit {
  constructor() { }
  ngOnInit(): void {
  }
}
```

The `@Component` decorator has properties specifying metadata for the component, such as what template it will use, what stylesheet, and so on. Let us launch the newly added component. To do this, first go to app.component.html under src\app folder and then delete all the markup and add this line:

```
<app-ang9-component></app-ang9-component>
```

Then, in ***ang9-component.component.html***, add only one line:

```
<p>ang9-component works!</p>
```

When you serve the application in the browser, you will see the output similar to Figure 8.15. Note that if your application had already been running till now, you do not even need to reload it - the browser will refresh on its own and reflect the latest changes. That is the power of `ng serve` command.

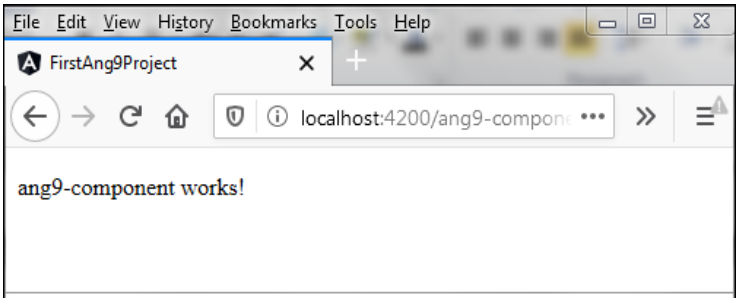


Figure 8.15: Displaying Newly Added Component

These examples are quite basic. In reality, you will create applications and pages with more functionality and rich UI elements.

Note:
If you want to add your component, directive, or so on to another module (other than the main application module, app.module.ts), you can simply prefix the name of the component with the module name and a slash:

```
ng g component module1/ang9-component
```

Here, `module1` is the name of an existing module.

8.5 Angular Pipes

Angular pipes present, a way to transform a value for display before it is rendered on the browser. An Angular pipe takes in data as input and transforms it to a desired output. If you have used filters in earlier versions of AngularJS or Angular, you already know how pipes work. Pipes are similar to filters.

Angular provides several built-in pipes such as `DatePipe`, `DecimalPipe`, `TitleCasePipe`, `UpperCasePipe`, `LowerCasePipe`, `CurrencyPipe`, and `PercentPipe`. They are all available for use in any template.

Pipe Parameters

You can pass any number of optional parameters to a pipe to tweak its output. A colon (:) along with the parameter value (such as currency:'EUR') is added to a pipe name. Multiple parameters can be given by separating values with colons (such as slice:1:5)

Code Snippet 3 shows an example that uses `CurrencyPipe`. This code should be added in *app.component.html*.

Code Snippet 3:

```
<div style = "width:100%;">
   <div style = "width:40%;float:left;border:solid 1px black;">
        <h1>Angular Currency Pipe Demo</h1>
      <b>{{7489.23 | currency:"USD"}}</b><br/>
   </div>
</div>
```

Figure 8.16 depicts the output of this code. The code has used USD parameter with the currency pipe to display $ with the currency amount and format the amount as currency.

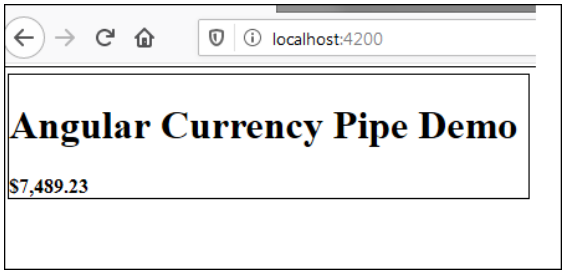


Figure 8.16: Displaying Newly Added Component

Code Snippet 4 demonstrates use of Date pipe.

Code Snippet 4:

```
import { Component } from '@angular/core';

@Component({
 selector: 'app-root',
 template: '<div><p>The annual review meeting has been rescheduled to
{{today | date}} </p></div>'
 })
export class AppComponent {
  title = 'pipeExample';
   today: number = Date.now();
}
```

Figure 8.17 depicts the output of this code. The default format for date is medium date.

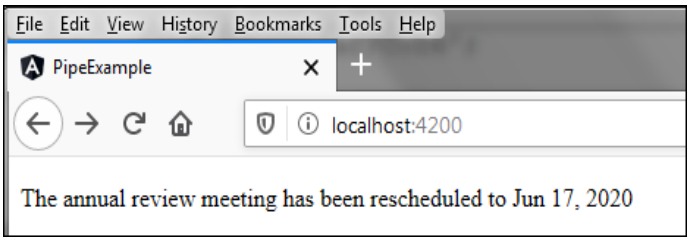


Figure 8.17: Displaying Newly Added Component

You can use a variety of options to customize and tweak the pipes that are provided by Angular.

Quick Test 8.1

1. In Angular 9, Ivy is enabled by default.
    a. True
    b. False
2. Angular CLI is the command-based interface that helps you build Angular applications.
    a. True
    b. False
3. To define a module, one should apply the @Module() decorator to a class.
    a. True
    b. False

8.6 Summary

- Angular is a complete rewrite of the AngularJS Web framework and is TypeScript-based and open-source.
- Angular 9 makes the next-generation compiler and runtime, Ivy, as default for all applications.
- Angular 9 has several new features and enhancements, including support for smaller bundle sizes, better debugging, phantom template variable menace, API extractor updates, and so on.
- Key components of the architecture include Modules, Components, Template, Directives, Data Binding, Services, and Dependency Injection.
- Angular Command-Line Interface (CLI) is the command-based interface and official tool that helps you build Angular applications.
- A module is a means by which one can group components, directives, services, and so on. To define a module, one should apply `@NgModule()` decorator to a class.
- Angular pipes are similar to filters in earlier versions of AngularJS and transform a value for display before it is rendered on the browser.

8.7 Exercise

1. In earlier versions of Angular, _____ was the default compiler.
   a) jQuery compiler
   b) JavaScript compiler
   c) View Engine
   d) Ivy

2. _____ is the tool that references API Extractor in Angular 9.
   a) Bazel
   b) Blaze
   c) Ivy
   d) Lint

3. Which of these Angular 9 artifacts are similar to filters in earlier Angular versions?
   a) Components
   b) Services
   c) Modules
   d) Pipes

4. Identify the component that is usually the root component in modules.
   a) AppComponent
   b) RootComponent
   c) Component
   d) None of these

5. Identify which of these files contains components definitions in Angular 9.
   a) app.components.ts
   b) components.ts
   c) module.components.ts
   d) app.components.js

8.8 Do It Yourself

1. Create and test an Angular 9 application that displays the sum of two numbers. For the purpose of the exercise, you can hard-code the numbers.

2. Create and test an Angular 9 application that displays current value of yen in dollars. Use currency pipe and date pipe to display the output.

Answers to Exercise

1. View Engine
2. Bazel
3. Pipes
4. AppComponent
5. app.components.ts

Answers to Quick Test

Quick Test 8.1

1. True
2. True
3. False

Session 9

Working with Forms, HTTP, REST, and Animation APIs

In this session, students will learn to:

- Describe an overview of forms in Angular 9
- Explain how to work with Angular 9 forms
- Explain how to create and consume a REST API with HTTP in Angular application
- Outline the steps to create an Animation based application in Angular 9

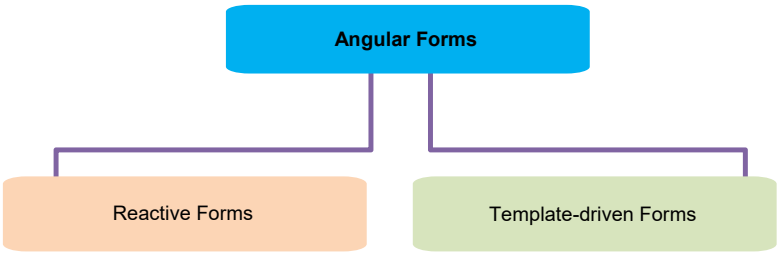
9.1 Forms in Angular 9

In an earlier session, you explored forms in AngularJS 1.7.9 and how to create them. The purpose of using forms remains the same in Angular 9 as well – to accept, validate, and process user input, however, the syntax and usage is different.

9.1.1 Types of Forms

In large practical-oriented applications, it is likely that you will use one or more forms on your pages.

Angular 9 provides two types of approaches for handling user input through forms:



Differences between the two can be outlined as follows:

Reactive Forms	Template-driven Forms
• Scalable, reusable, and testable • More Robust • Structured Data Model • Form Validation through functions	• Simple forms with logic manageable in template • Not suitable for much scaling • Unstructured data model • Form Validation through directives

Thus, each of the two offers different advantages.

Features that are common in both include same essential building blocks, which are as follows:

FormControl tracks the value and validation status of an individual form control.

FormGroup tracks the same values and status for a collection of form controls.

FormArray tracks the same values and status for an array of form controls.

ControlValueAccessor creates a bridge between Angular FormControl instances and native DOM elements.

The syntax and procedure for using forms in your code in Angular 9 is different from that in AngularJS. Let us begin with creating an application that uses forms. This example will use template-driven forms.

At the command prompt, type the following command:

```
ng new formdemo
```

This command creates an Angular 9 application and installs all necessary files and libraries in the application folder, which in this case will be **formdemo**. The **src** subfolder will contain application specific code.

Figure 9.1 shows this command in action.

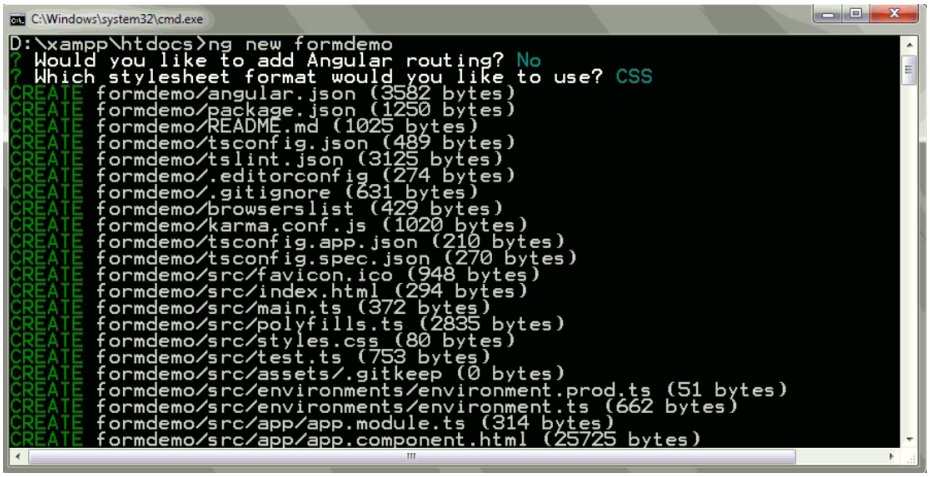


Figure 9.1: Creating the FormDemo Application

When the application is created, navigate to *src/app/app.component.html* and replace the existing code with code given in Code Snippet 1.

Code Snippet 1: *src/app/app.component.html*

```
<form #userlogin = "ngForm" (ngSubmit) = "onClickSubmit(userlogin.value)" >
   <input type = "text" name = "emailid" placeholder = "Enter emailid here" ngModel>
   <br/>
   <input type = "password" name = "password" placeholder = "Enter password here"
  ngModel>
   <br/>
   <input type = "submit" value = "Submit">
</form>
```

What has been done here is simple. A form has been created and the `ngForm` directive has been exported into a local template variable using `ngForm` as the key (`#userlogin = "ngForm"`). Usually, when we import `FormsModule` in our application, this directive becomes active by default on all `<form>` tags. However, one can still explicitly export it into a local template variable using `ngForm` as we have done here. By performing this action, you get a handy reference to the form that you can use it anywhere, such as, in the `onClickSubmit` function.

Email id and password are accepted from the user using standard HTML elements. To make them accessible in the Angular code if necessary, the `ngModel` directive has been added to both of these elements.

The `ngSubmit` event is notified when user triggers a form submission by clicking **Submit** button. Therefore, in the first line, a listener has been added for `ngSubmit`. This hijacks the default form submission process and calls the function `onClickSubmit()` on our component and passes the data from the form. `onClickSubmit()` will be a function we define later in ***app.component.ts***.

Now, replace the code in **src/app/app.module.ts** with code given in Code Snippet 2.

Code Snippet 2: *src/app/app.module.ts*

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {FormsModule} from '@angular/forms'
import {AppComponent} from './app.component';
@NgModule({
  declarations: [
    AppComponent
      ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Here, we define the form module by importing the `FormsModule` library. Template-driven forms exist in their own module, hence, it is necessary to import `FormsModule`. Once that is done, we add the `FormsModule` to the list of `imports` defined in the `@NgModule` decorator. This will give application access to all of the template-driven forms features, including `ngModel`.

Next, replace the code in *src/app/app.component.ts* with code given in Code Snippet 3.

Code Snippet 3: *src/app/app.component.ts*

```
import { Component } from '@angular/core';
@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.css']
})
export class AppComponent {
    onClickSubmit(data) {
      alert("Entered Email id : " + data.emailid);
   }
}
```

Here, we have created a function `onClickSubmit()` which will accept one parameter and will act as an event handler for the click event of **Submit** button. In this function, there is just one simple action - displaying the email id that was entered by the user.

Deploy the application using `ng serve` as shown in Figure 9.2.

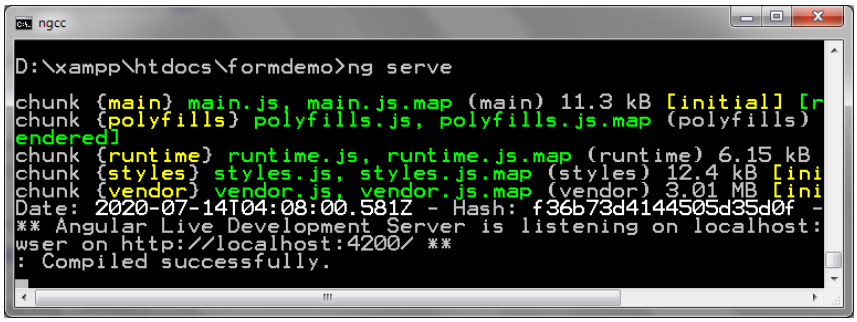


Figure 9.2: Deploying FormDemo Application

Launch the application in the browser using http://localhost:4200/.

Initial output is shown in Figure 9.3.

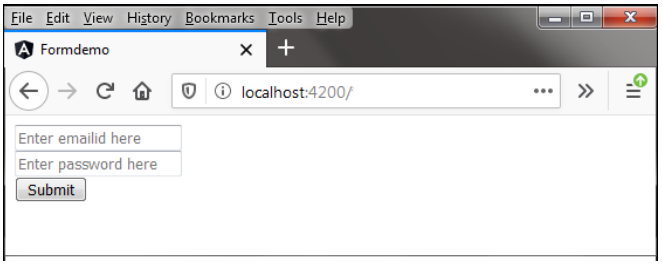


Figure 9.3: FormDemo Application – Initial View

The output after specifying email id is shown in Figure 9.4.

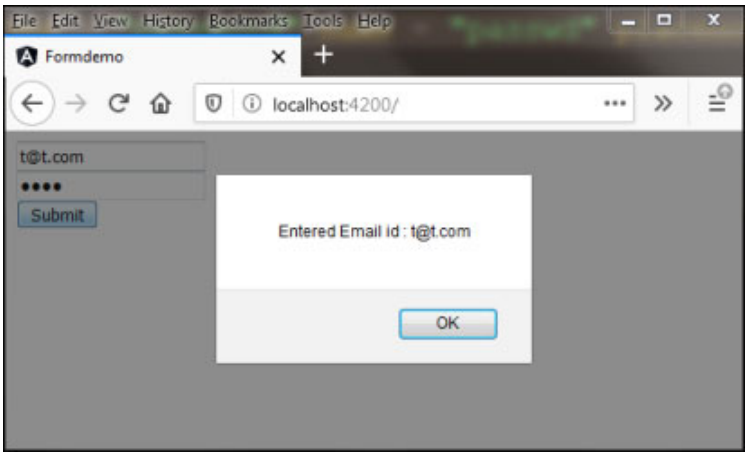


Figure 9.4: FormDemo Application After Email Id Supplied

The drawback of this code though is that there is no validation done. The application will display the alert even if the first text box (for email id) contained an invalid value. To fix this, we need to add validation to the code. Add the codes shown in Code Snippets 4, 5, and 6 to the respective files as indicated.

Code Snippet 4: *app.component.html*

```
<div>
  <h5>Angular 9 Template-Driven Form Validation</h5>
  <form #userlogin="ngForm" name="form" (ngSubmit)="userlogin.form.valid && onSubmit()>
       <div>
          <label>Email: </label>
        <input type="text" name="email" placeholder = "Enter emailid here"
            [(ngModel)]="model.email" #email="ngModel" [ngClass]="{ 'is-invalid':
            userlogin.submitted && email.invalid }" required email>
        <div *ngIf=" userlogin.submitted && email.invalid" >
        <div *ngIf="email.errors.required">Email is required</div>
        <div *ngIf="email.errors.email" style="color:red">Email must be a valid
                          email address</div>
         </div>
         </div>
         <label>Password: </label><input type = "password" name = "password"
            placeholder = "Enter password here" ngModel>
      <br>
      <br>
       <button>OK</button>
       <button type="reset">Cancel</button>
 </form>
</div>
```

Code Snippet 5: *app.module.ts*

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule} from '@angular/forms'
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Code Snippet 6: *app.component.ts*

```
import { Component } from '@angular/core';
@Component({ selector: 'app-root', templateUrl: 'app.component.html' })
export class AppComponent {
  model: any = {};
  onSubmit() {
    alert('Valid email\n' + this.model.email);
  }
}
```

The initial output after adding validation is shown in Figure 9.5 and then, after an invalid id is provided, the output shown in Figure 9.6 will be seen.

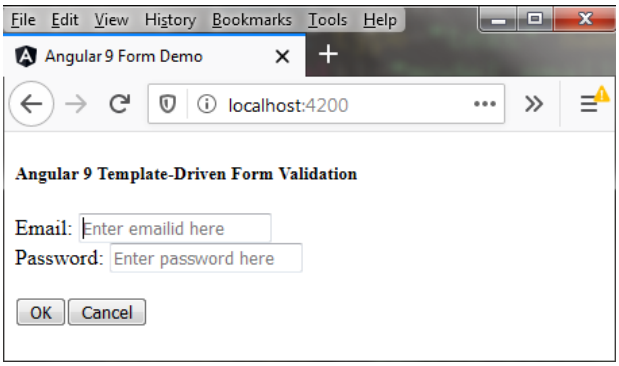


Figure 9.5: Initial Output After Adding Validation

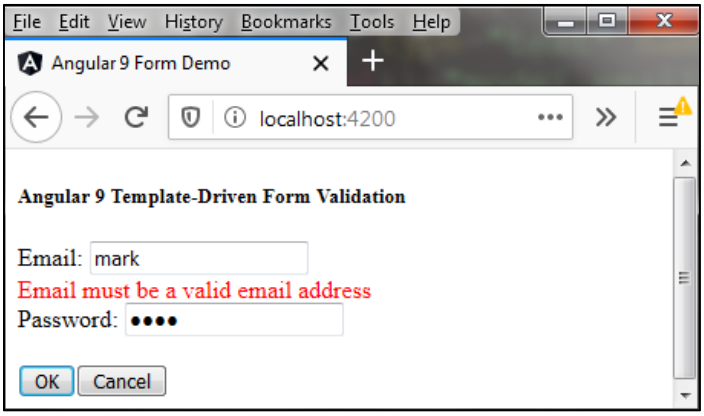


Figure 9.6: Output Showing Validation in Action

So far, these applications were basic ones without any additional components. What if the requirement demanded an application using forms and additional components? The next example discussed here makes use of both. It is an application designed to convert a given value in Miles Per Hour (MPH) to Yards Per Hour (YPH).

Let us call this application as **ConverterApp**.

As usual, begin with creating the application using `ng new`.

```
ng new ConverterApp
```

Once the application is created, change to the application directory as follows:

```
cd ConverterApp
```

Then, create a new component in the application named converter:

```
ng generate component converter
```

Open *app.module.ts* under src/app and add the code shown in Code Snippet 7.

Code Snippet 7: *src/app/app.module.ts*

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule} from '@angular/forms'
import { AppComponent } from './app.component';
import { ConverterComponent } from './converter/converter.component';
@NgModule({
  declarations: [
    AppComponent,
    ConverterComponent
      ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Here, in addition to the usual code, we include an import statement to get the component. There is also a reference to the component in the `NgModule` block that indicates that this component is now a part of the module.

Add the code shown in Code Snippet 8 to *app.component.html*:

Code Snippet 8: *src/app/app.component.html*

```
<app-converter></app-converter>
```

Observe in Code Snippet 9 how this time the *app.component* file has just one line and all the long code has been moved to *converter\converter.component.html*.

Code Snippet 9: *src/app/app.component.ts*

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'ConverterApp';
}
```

Add the code given in Code Snippet 10 to *app.component.ts*.

Code Snippet 10: *converter\converter.component.html*

```
<div >
<h2>Converter</h2>
<p>Type a value in the Miles Per Hour (MPH) field to convert value to Yards Per
Hour:</p>
<form #speed = "ngForm" (ngSubmit) = "onClickSubmit(speed.value)" >
   Enter MPH: <input type = "number" name = "inputMPH" ngModel>
   <br/>
   <br/>
   <input type = "submit" value = "Convert">
</form>
<br/><br/>
Converted Value in Yards Per Hour: <b>{{outputYPH}}</b>
</div>
```

This defines the view for the application, wherein the value of miles per hour is accepted from the user via a text box on a form and the converted value in yards per hour is displayed. The form is given an alias #speed and has an `ngSubmit` directive mapping to the user-defined method `onClickSubmit()`.
Add the code given in Code Snippet 11 to *converter\converter.component.ts*.

Code Snippet 11: *converter.component.ts*

```
import { Component, OnInit } from '@angular/core';
import {FormsModule} from '@angular/forms'
import { NgForm } from '@angular/forms';
@Component({
  selector: 'app-converter',
  templateUrl: './converter.component.html',
  styleUrls: ['./converter.component.css']
})
export class ConverterComponent implements OnInit {
outputYPH;
  constructor() { }
  ngOnInit() {  }
  onClickSubmit(data) {
      console.log(data.inputMPH*1760);
      this.outputYPH = (data.inputMPH*1760);
   }
}
```

In this code, the `onClickSubmit` function has a formula within an expression. This formula calculates the yards per hour value by multiplying the given miles with 1760. Note that the local variable `outputYPH` has to be declared before it is used in the formula. Upon deploying the application (through **ng serve**) and launching it in the browser, the output will be as shown in Figure 9.7.

Figure 9.7: Output of ConverterApp

9.2 HTTP Communication in Angular 9

Angular provides the `HttpClient` service as an injectable class for communicating with a remote server over HTTP. Using this class, you can fetch data in your Angular application from a REST API server.

The primary purpose of `HttpClient` is to facilitate HTTP requests. This class provides various methods, using which developers can perform different kinds of requests.

Let us look at an example that will implement an Angular service and fetch data from the REST API server. In a practical scenario, one would have a backend REST API that would take a lot of effort to create. For this example, however, we will use a tool, *json-server*, to simulate a fully working backend API with fake data.

The aim of this example application will be to retrieve a series of records representing address information stored in JSON format and display it in a human readable format. The example application will make use of a JavaScript library to generate fictitious address information comprising city, street name, street address, zip code, and country details in JSON format. The application will use `HttpClient` service to perform HTTP requests to the (fake) REST API server, retrieve this address information, and display in it a formatted table on the browser page.

Step 1: Create a new application named **addressbook** as follows:

```
ng new addressbook
```
Make sure that you press the Y key when asked about routing. This will add routing capabilities to our application that we will need later.

Step 2: Import `HttpClient` in the Angular project

`HttpClient` is a built-in Angular service, but is not included in the project by default. Hence, you need to import it.

This can be easily done by importing `HttpClientModule` and adding it to the module where you want to use it. At this step, we have only one module – the application module, so import `HttpClientModule` in that module as shown in Code Snippet 12.

Code Snippet 12: *src/app/app.module.ts*

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HttpClientModule } from '@angular/common/http';
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 3: Generate a new Home Component
```
ng generate component Home
```

Step 4: Generate a new About Component
```
ng generate component About
```

After generation of components, the code in module.ts would be as shown in Code Snippet 13.

Code Snippet 13: *src/app/app.module.ts*

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
import { HttpClientModule } from '@angular/common/http';
@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    AboutComponent
  ],
  imports: [
   BrowserModule,
   AppRoutingModule,
   HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 5: Set up Routing

In this step, you will discover how to add routing to the example application. Routing will allow us to map or associate **home** and **about** components to specific URL paths, such as, */home* and */about*. Using this approach, you can create an app with multiple views. At the time of creating the project (in Step 1), routing was already set up by Angular CLI. The auto-generated file *src/app/app-routing.module.ts* contains the routing module with an empty `routes` array.

A route is a JavaScript object containing properties such as `path`, `pathMatch`, `redirectTo` and `component` that are described as follows:

`path`	contains the path segment of the URL of the route
`pathMatch`	specifies how the router should match the path
`redirectTo`	specifies the path to redirect to
`component`	specifies the component to associate with a specific path

In the current example, **home** path is associated with the `HomeComponent` and **about** with `AboutComponent`. We added a redirection route that redirects the empty path to `home` path so when users first land in our app via the main URL, they will be redirected to the home view where they can find something useful. Refer to Code Snippet 14.

Code Snippet 14: *app-routing.module.ts*

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full'},
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
```

```
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Step 6: Simulate a Fully-Working REST API Using JSON-Server

As mentioned earlier, we will be using a tool, *json-server*, to simulate a fully working backend REST API with fake data.

To do this, open a new command prompt in the current path and run the following command to install `json-server` using `npm`:

```
npm install --save json-server
```

Next, we will use Faker.js (https://github.com/marak/Faker.js/) to generate data. Faker.js is a readymade JavaScript library that generates massive amounts of realistic fake data in Node.js and the browser.

Navigate to the following link:

https://rawgit.com/Marak/faker.js/master/examples/browser/index.html

You will find that Faker has following databases available with readymade data:

- address
- commerce
- company
- database
- date
- finance
- hacker
- helpers
- image
- internet
- lorem
- name
- phone
- random
- system

For our application, we shall use the **address** database. Next, create a directory named `backend` in your Angular project:

To install Faker, go to the command prompt and type the following command:
`npm install faker`

Next, create an *init.js* file in the backend directory. Refer to Code Snippet 15.

Code Snippet 15: *init.js*

```
var faker = require('faker');
var database = { addresses: []};
for (var i = 1; i<= 15; i++) {
  database.addresses.push({

    city:faker.address.city(),
    streetName: faker.address.streetName(),
    streetAddress: faker.address.streetAddress(),
     zipCode: faker.address.zipCode(),
     country:faker.address.country()
  });
}
```

We first imported faker and next we defined an object with one empty array for addresses. Next, we entered a `for` loop to create 15 fake entries using `faker` methods such as `faker.address.streetName()` and `faker.address.city()` for generating address details.

When this file is run, the data that is generated in the JSON file will look something like this:


```
{"addresses":[{"city":"Port Breana","streetName":"Shawna Wells","streetAddress":"2542
Klein Courts", "zipCode":"07205", "country":"Mozambique"},
{"city":"Niafort","streetName":"Elena Key","streetAddress": "976 Carroll Roads",
"zipCode":"90217-7189","country":"Yemen"},{"city":"East Maziehaven","streetName":"Tad
Corners","streetAddress":"7525 Nader Stravenue",
"zipCode":"16001","country":"Turkey"},{"city":"Millerfurt","streetName":"Prohaska
Isle","streetAddress":"5261 Wolff Island","zipCode":"67762-
0129","country":"Myanmar"},{"city":"Jessicaton","streetName":"Streich
Street","streetAddress":"7044 Minerva Branch","zipCode":"34401-
0199","country":"Gibraltar"},{"city":"Weissnatport","streetName":"Lang
Loaf","streetAddress":"712 Abe Bridge","zipCode":"37486-8488","country":"Antarctica
(the territory South of 60 deg S)"},{"city":"North Cristalport","streetName":"Buck
Well","streetAddress":"8428 Herman Pines","zipCode":"70777-
8065","country":"Austria"},{"city":"Langoshshire","streetName":"Joanne
Inlet","streetAddress":"249 Reba Squares","zipCode":"76706-5012","country":"Turks and
Caicos Islands"},{"city":"South Trinityshire","streetName":"Ferry
Camp","streetAddress":"714 Dayne Park","zipCode":"91526-
6068","country":"China"},{"city":"Carrollchester","streetName":"Abernathy
Route","streetAddress":"45665 Adelle Parkways","zipCode":"34034-
1112","country":"Albania"},{"city":"Arashire","streetName":"Kendra
Pike","streetAddress":"0322 Bridgette
Parks","zipCode":"56938","country":"Eritrea"},{"city":"South
Julietport","streetName":"McKenzie Club","streetAddress":"3836 Xzavier
Vista","zipCode":"26258","country":"Bahrain"},{"city":"Lake
Maybell","streetName":"Giuseppe Flat","streetAddress":"483 Verner
Street","zipCode":"46753","country":"Norfolk Island"},{"city":"Port
Sophia","streetName":"Melvin Divide","streetAddress":"174 Lebsack
Divide","zipCode":"07778-
7003","country":"Guadeloupe"},{"city":"Smithshire","streetName":"Maximilian
Stravenue","streetAddress":"9629 Nader Bypass","zipCode":"74482","country":"Nepal"}]}
```


Next, add the `init-api` and `serve-api` scripts to the *package.json* file as given in Code Snippet 16.

Code Snippet 16: *package.json*

```
{
  "name": "httpdemo",
  "version": "0.0.0",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e",
  "init-api": "node ./backend/init.js > ./backend/data.json",
  "serve-api": "json-server --watch ./backend/data.json"
  },
```

What we have done here is to ensure that when `init-api` is called, it will in turn call `init.js` present in the backend folder and generate the fake data in `data.json`. Likewise, when `serve-api` is called, it will turn execute the command `json-server --watch ./backend/data.json`.

Ensure the code in *app.component.ts* is as given in Code Snippet 17.

Code Snippet 17: *app.component.ts*

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'httpdemo';
}
```

Step 7: Populate Data

Next, go to your command-line interface and run the `init-api` script as follows:

```
npm run init-api
```

This will populate data from the Faker database into our *data,json* file in the **backend** folder.

Step 8: Run the REST API server

Next, run the REST API server in a new command window:

```
npm run serve-api
```

Refer to Figure 9.8. As we can see, this command in turn calls the `json-server --watch ./backend/data.json` command.

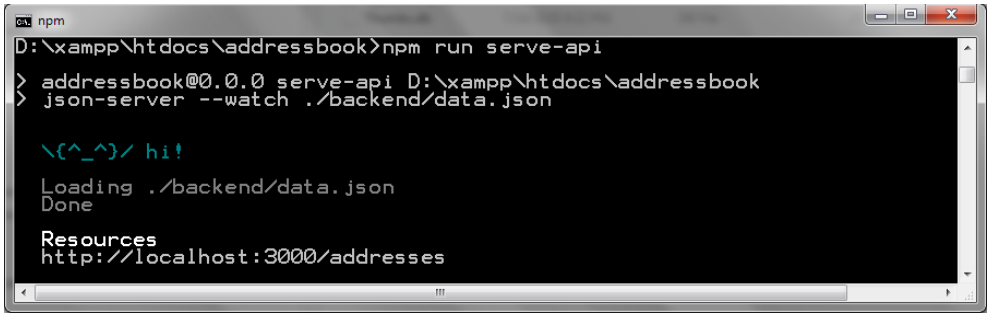


Figure 9.8: Running the REST Server

You can now consume the REST API in your application. It will be available from the *http://localhost:3000/addresses*.

API endpoints that can be used via the REST API server are as follows:

- `GET /addresses` for getting the address
- `GET /addresses/<id>` for getting a single address by id
- `POST /addresses` for creating a new address
- `PUT /address/<id>` for updating a address by id
- `PATCH /address/<id>` for partially updating address by id
- `DELETE /address/<id>` for deleting address by id

Step 9: Send HTTP Requests with Angular HttpClient

In this step, we send HTTP `GET` requests to fetch data from our REST API server using Angular 9, `HttpClient`, and a service.

Create an Angular service that allows us to consume data from our REST API server. To do this, at the command-line interface, type the following command:

```
ng generate service api
```

Next, head to the *src/app/api.service.ts* file, and add code as shown in Code Snippet 18.

Code Snippet 18: *api.service.ts*

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
@Injectable({
  providedIn: 'root'
})
export class ApiService {
  private SERVER_URL = "http://localhost:3000";
  constructor(private httpClient: HttpClient) { }
  public fetchData(){
                return this.httpClient.get(`${this.SERVER_URL}/addresses`);
        }
}
```

In this code, we imported and injected the `HttpClient` service through the service constructor, defined the `SERVER_URL` variable, and assigned it the address of the REST API server that is running locally.

Next, define a `fetchData()` method that will send an HTTP `GET` request to the REST API endpoint for address, that is, `/address`.

Step 10: Injecting and Calling the Service

Next, we need to use this service in the home component. Open *src/app/home/home.component.ts* and add the code shown in Code Snippet 19.

Code Snippet 19: *home.component.ts*

```
import { Component, OnInit } from '@angular/core';
import { ApiService } from '../api.service';
@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {
 addresses = [];
   constructor(private apiService: ApiService) { }
   ngOnInit() {
       this.apiService.fetchData().subscribe((data: any[])=>{
           this.addresses = data;
       })
   }
}
```

The code imports and injects ApiService and uses `fetchData()` method of the service for fetching data from the REST API server and assigns the data to an array variable `addresses`.

Step 11: Rendering the Fetched Data with ngFor

The view for the Home component should display the fetched data, that is, the address information for each entry in the JSON file. The HTML markup that will render this view is shown in Code Snippet 20 and should be added in *src/app/home/home.component.html* file.

Code Snippet 20: *home.component.html*

```
<div style="padding: 13px;">
  <ul *ngFor="let address of addresses" style="margin-top:10px;">
    <table border = "0">
      <thead>
        <td>City</td>
        <td>Street </td>
        <td>Street Address</td>
        <td>zipCode</td>
```

```
        <td>Country</td>
        <td> </td>
      </thead>
      <tr>
        <td>{{address.city}}</td>
        <td>{{address.streetName}} </td>
        <td>{{address.streetAddress}}</td>
        <td>{{address.zipCode}}</td>
        <td> {{address.country}}</td>
      </tr>
    </table>
  </ul>
</div>
```

Then, we specify a style for the table that will be rendered in the Home view. Refer to Code Snippet 21.

Code Snippet 21: *home.component.css*

```
thead{
    background-color: #c4c4c4;
}
```

For the About view, we just display a text stating About Page. Refer to Code Snippet 22.

Code Snippet 22: *about.component.html*

```
<p style="padding: 20px;"> About Page </p>
```

There is no change made to *about.component.ts* and the default code is retained as is. Code Snippet 23 shows the default code.

Code Snippet 23: *about.component.ts*

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-about',
  templateUrl: './about.component.html',
  styleUrls: ['./about.component.css']
})
export class AboutComponent implements OnInit {
  constructor() { }
  ngOnInit(): void {
  }
}
```

Finally, we reach the stage where we can execute the application. Remember that in Step 8, we specified the command to start the server. Hence, the REST API server is currently running. To deploy and launch the application, run `ng serve` as shown in Figure 9.9 and then, type http://localhost:4200/home in the browser.

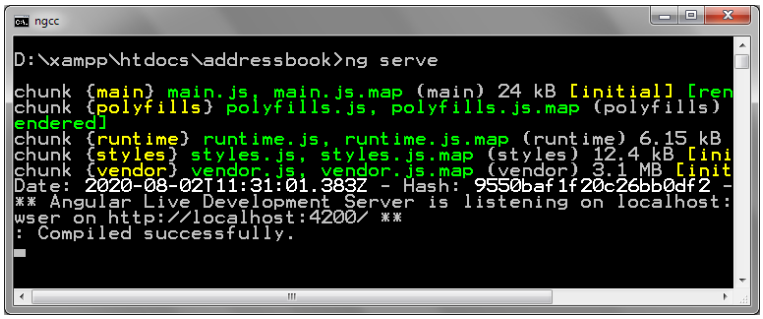


Figure 9.9: Deploying the AddressBook Application

The output will be as shown in Figure 9.10.

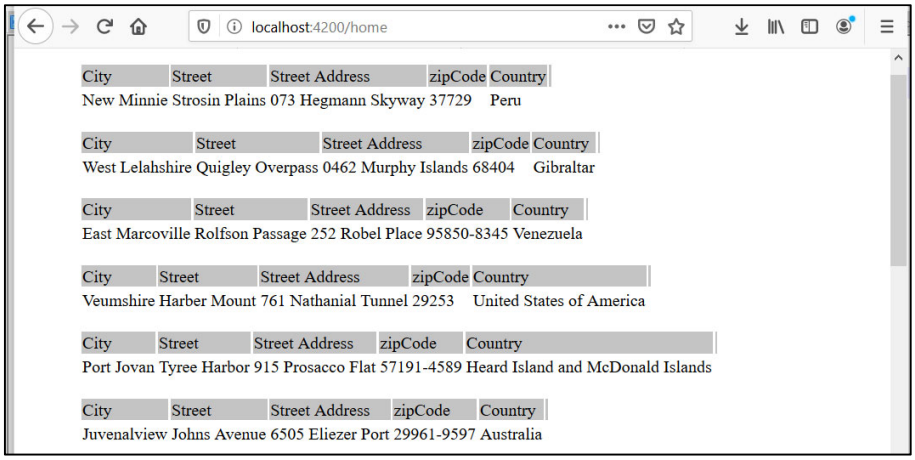


Figure 9.10: Output of AddressBook Application

9.3 Working with the Animation API in Angular 9

The Animations API in Angular provides a declarative API to build and reuse animations throughout our components.

Angular Animation API is not part of the default Angular installation and is shipped as a separate package. It should be installed as follows:

```
npm install @angular/animations
```

Our aim is to design an application that will display two images in sliding fashion, based on a button click.

To begin with, create an Angular 9 application named **AnimationApp**. Then, create a component under it named **slide-panel**. Install the Angular Animation API.

Next, import `BrowserAnimationsModule` in the root module of the Angular application as shown in Code Snippet 24.

Code Snippet 24: *app.module.ts*

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { AppComponent } from './app.component';
import { SlidePanelComponent } from './slide-panel';

@NgModule({
  imports     : [ BrowserModule, BrowserAnimationsModule ],
  declarations: [ AppComponent, SlidePanelComponent ],
  bootstrap   : [ AppComponent ]
})
export class AppModule {}
```

The view for the application will comprise a custom layout with two ‘panes’ formed using div element and each pane will contain an image. Refer to Code Snippet 25 for the code.

Code Snippet 25: *app.component.html*

```
<my-slide-panel [activePane]="isLeftVisible ? 'left' : 'right'">
<div leftPane><img src="https://cdn.pixabay.com/photo/2020/07/10/12/08/hot-air-balloon-
5390487_960_720.jpg" height="140" width="209" alt="Hot Air Balloon" />
 </div>
  <div rightPane> <img src="https://cdn.pixabay.com/photo/2020/06/24/17/20/blue-flower-
5337027_960_720.jpg" height ="200" width="200" alt="Flowers" />
</div>
```

```
</my-slide-panel>
<button (click)="isLeftVisible = !isLeftVisible">Toggle panes</button>
```

In *app.component.ts*, we set `isLeftVisible` to true for the initial view so that the left pane will be visible in the beginning. Refer to Code Snippet 26.

Code Snippet 26: *app.component.ts*

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  styleUrls: [ './app.component.css' ],
  templateUrl: './app.component.html'
})
export class AppComponent {
  isLeftVisible = true;
}
```

The codes for the respective stylesheets of the main application and the component are given in Code Snippets 27 and 28.

Code Snippet 27: app.component.css

```
:host {
  display: block;
}
button {
  font-size: 24px;
  margin-top: 16px;
}
my-slide-panel {
  border: 3px solid #0c1414;
  max-width: 500px;
  height: 150px;
  div {
    font: 20px sans-serif;
    height: 100%;
    display: flex;
    align-items: center;
    justify-content: center;
  }
}
/deep/ my-slide-panel .panes {
  border: 6px dashed #4990E2;
  box-sizing: border-box;
}
[leftPane]  { background-color: #FFF897; }
[rightPane] { background-color: #6CE6CB; }
```

Code Snippet 28: *slide-panel.component.scss*

```
:host {
  display: block;
  overflow: hidden;
}
.panes {
  height: 100%;
  width: 200%;
  display: flex;
  div {
    flex: 1;
  }
}
```

The HTML markup for the component should be as given in Code Snippet 29.

Code Snippet 29: *slide-panel.component.html*

```
<div class="panes" [@slide]="activePane">
  <div><ng-content select="[leftPane]"></ng-content></div>
  <div><ng-content select="[rightPane]"></ng-content></div>
</div>
```

Finally, we apply animation in the code. Refer Code Snippet 30. Here, we listen for any changes happening in the view and then, based on whether 'left' or 'right' pane needs to be shown, the appropriate pane is slided over. The transition delay is set to 300 milliseconds.

Code Snippet 30: *slide-panel.component.ts*

```
import { ChangeDetectionStrategy, Component, Input } from '@angular/core';
import { animate, state, style, transition, trigger } from
'@angular/animations';
type PaneType = 'left' | 'right';
@Component({
  selector: 'my-slide-panel',
  styleUrls: [ './slide-panel.component.scss' ],
  templateUrl: './slide-panel.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush,
  animations: [
    trigger('slide', [
      state('left', style({ transform: 'translateX(0)' })),
      state('right', style({ transform: 'translateX(-50%)' })),
      transition('* => *', animate(300))
    ])
  ]
})
export class SlidePanelComponent {
  @Input() activePane: PaneType = 'left';
}
```

Upon deploying and launching the application in the browser, the output will be as shown in Figures 9.11 and 9.12.

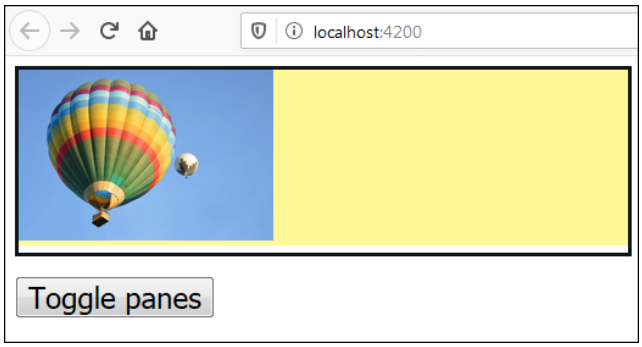


Figure 9.11: Initial Output of AnimationApp

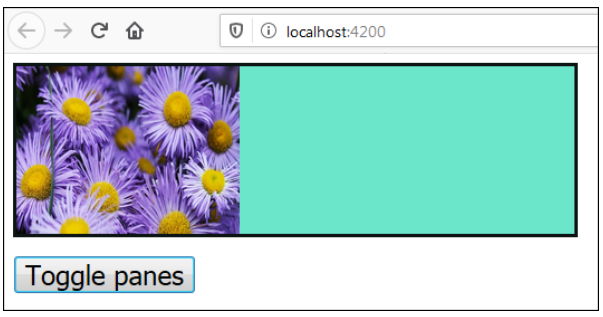


Figure 9.12: Output of AnimationApp After Clicking Button

Quick Test 9.1

1. Animations API should be installed locally using the command `npm install @angular/animations`
   a. True
   b. False
2. Fake.js is a readymade JavaScript library that generates massive amounts of realistic fake data in Node.js and the browser.
   a. True
   b. False
3. When we import `FormsModule` in our application, the `ngForm` directive becomes active by default on all `<form>` tags.
   a. True
   b. False

Summary

- Angular 9 provides two types of approaches for handling user input through forms: reactive and template-driven.
- Reactive Forms are scalable, reusable, and testable, more robust and have a structured data model.
- Template-driven forms are simple forms with logic manageable in template, they are not suitable for much scaling, and have an unstructured data model.
- When we import `FormsModule` in our application, the `ngForm` directive becomes active by default on all `<form>` tags.
- The `ngSubmit` event is notified when user triggers a form submission by clicking **Submit** button.
- Angular provides the `HttpClient` service as an injectable class for communicating with a remote server over HTTP. The primary purpose of `HttpClient` is to perform HTTP requests.
- Tools such as *json-server* enable to simulate a fully working backend API.
- The Animations API in Angular provides a declarative API to build and reuse animations throughout our components.

9.4 Exercise

1. Angular 9 provides two types of forms: _____ and _____.

   a. React forms and Template-driven forms
   b. Reactive forms and Template-driven forms
   c. Reactive forms and dynamic forms
   d. Reactive forms and test-driven forms

2. Which of the following represents the correct code to add form functionality to your module?

   a. `import {Forms} from '@angular/forms'`
   b. `import {FormModule} from '@angular/forms'`
   c. `import {FormsModule} from '@angular/forms'`
   d. `import {FormsModule} from '@angularforms'`

3. Identify the valid properties of `route` JavaScript object.

   a. `path`
   b. `pathMatch`
   c. `redirectTo`
   d. `componentroute`

4. Which module should you import into your root module to add Animation functionality?

   a. `BrowserAnimationsModule`
   b. `AnimationsModule`
   c. `AnimationsAppModule`
   d. All of these

5. Transition delay for animations is usually specified in _____.

   a. Milliseconds
   b. Seconds
   c. Minutes
   d. Nanoseconds

9.5 Do It Yourself

1. Create an Angular 9 application that makes use of a form to accept Customer details such as Name, Age, Address, and Phone Number.

2. Create an Angular 9 Http application similar to the one created in the session, however, it should generate fake ‘company’ information using Faker.js and then, retrieve it using HTTP requests.

Answers to Exercise

1. Reactive forms and Template-driven forms
2. `import {FormsModule} from '@angular/forms'`
3. `path`, `pathMatch`, and `redirectTo`
4. `BrowserAnimationsModule`
5. Milliseconds

Answers to Quick Test

Quick Test 9.1

1. True
2. False
3. True

Session 10

AngularJS Material

In this session, students will learn to:

- Describe AngularJS Material
- Learn about Customizations in AngularJS Material
- Explain how to use API references in AngularJS Material
- Explain the process of migration from AngularJS to Angular

10.1 What is AngularJS Material?

AngularJS Material is a User Interface (UI) Component library designed as an implementation of Google's Material Design Specification from the year 2014 to 2017. For year 2018 and later, the updated version is referred to as Angular Material and is applicable to Angular v 2.0 onwards. This session will focus on AngularJS Material, which is used with AngularJS.

AngularJS Material provides a set of reusable, well-tested, and accessible UI components for AngularJS developers, some of which are depicted in Figure 10.1.

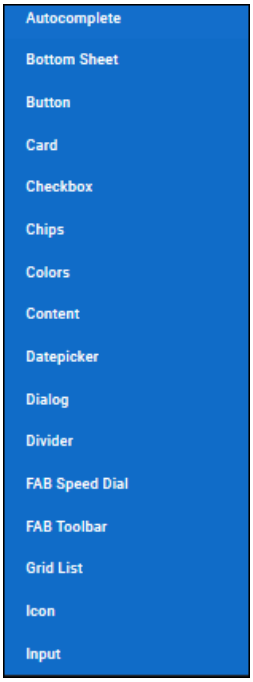


Figure 10.1: UI Components Provided by AngularJS Material

AngularJS Material is mature, stable, and production-ready.

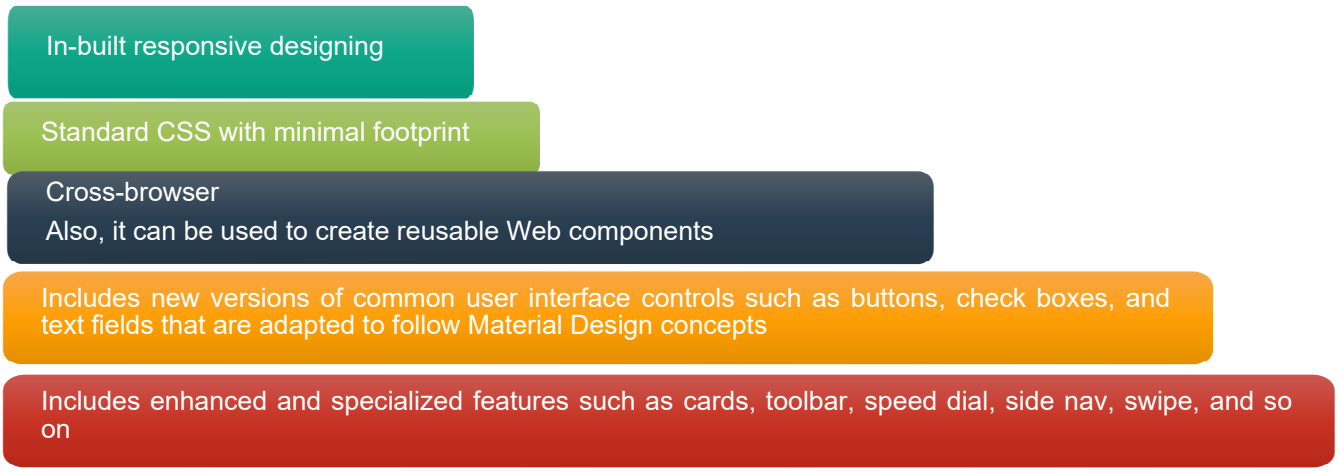
Pre-requisite: You need to have any version of AngularJS between 1.7.2 to 1.8.x, in order to work with AngularJS Material.

This session will use AngularJS 1.7.9.

10.2 Features and Benefits of AngularJS Material

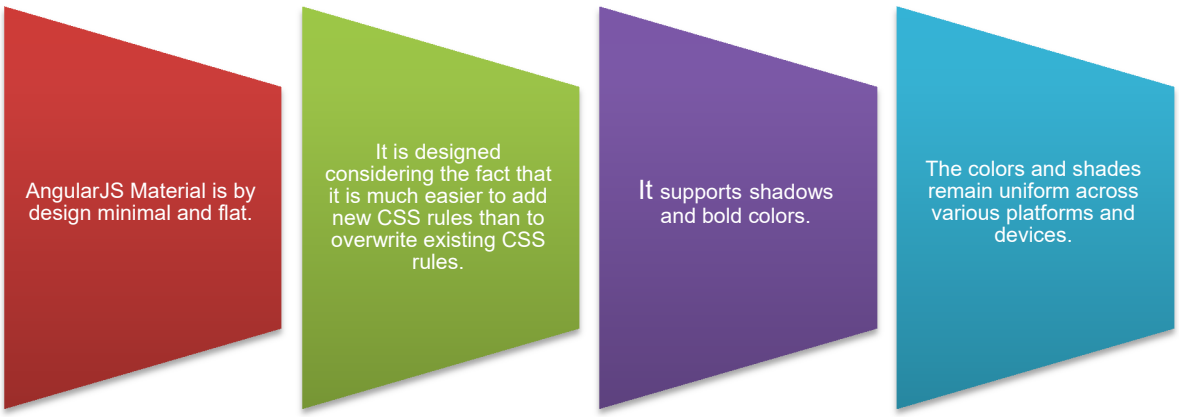
The reusable UI components provided by AngularJS Material aid developers in building attractive, consistent, and functional Web pages. At the same time, it helps developers to adhere to modern Web designing principles such as browser portability and device independence.

Some of the key features of AngularJS Material are as follows:



With built-in responsive design, Websites created using AngularJS Material can redesign dynamically as per device size. AngularJS Material classes are created such that Websites can fit any screen size. Websites created using AngularJS Material are fully compatible with desktop PCs, tablets, and mobile devices.

AngularJS Material also provides extensible capabilities, which are outlined as follows:



Another key benefit of AngularJS Material is that it is free to use.

10.1.1 Installing AngularJS Material

Developers can install AngularJS Material locally using `npm` or directly use Google CDN links in their application.

In the first approach, everything is installed explicitly on the local machine using `npm`. This includes AngularJS installation too. To begin with, create a folder named *MaterialDemo* in your XAMPP htdocs path. Navigate to the folder.

Open a new command prompt and give the command `npm init`. This will help to initialize your project. You will be prompted with several options. Leave the options as default by pressing Enter key against each as shown in Figure 10.2.

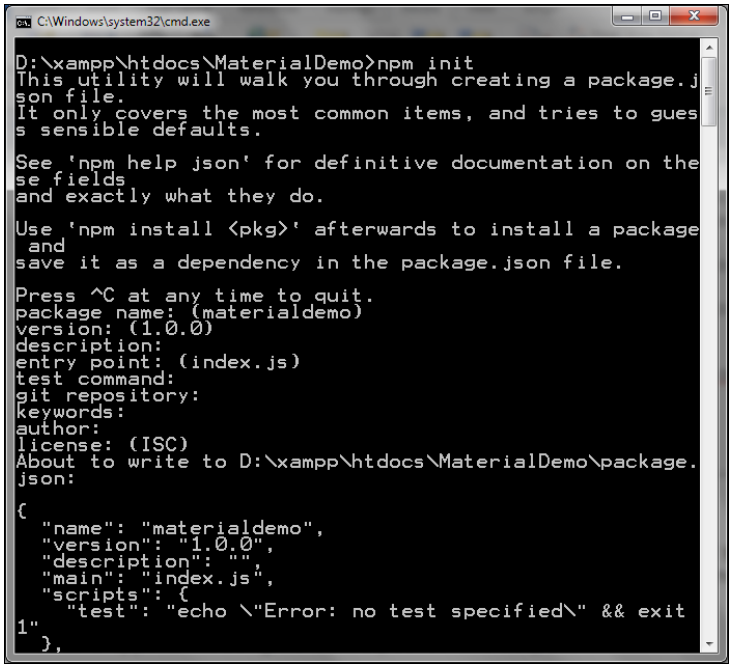


Figure 10.2: Initializing the Project

Next, install AngularJS libraries locally using the command `npm install angular@1.7.9` as shown in Figure 10.3.

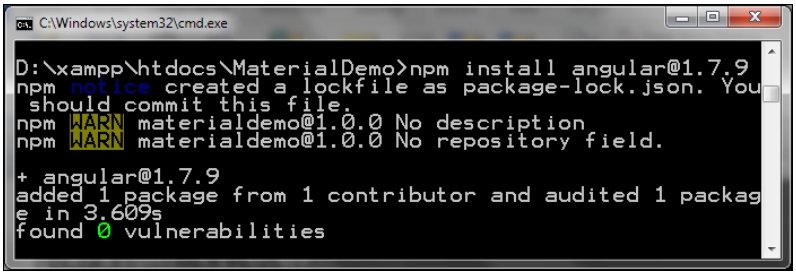


Figure 10.3: Installing AngularJS

Then, install angular-messages, angular-aria, and angular-animate libraries which are dependencies for AngularJS Material 1.1.22. Refer to Figures 10.4 to 10.6.

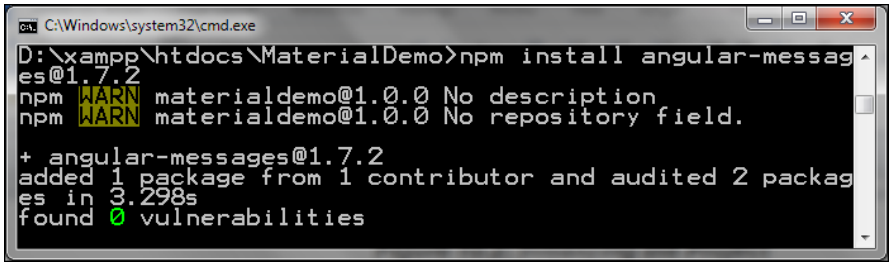


Figure 10.4: Installing AngularJS Messages

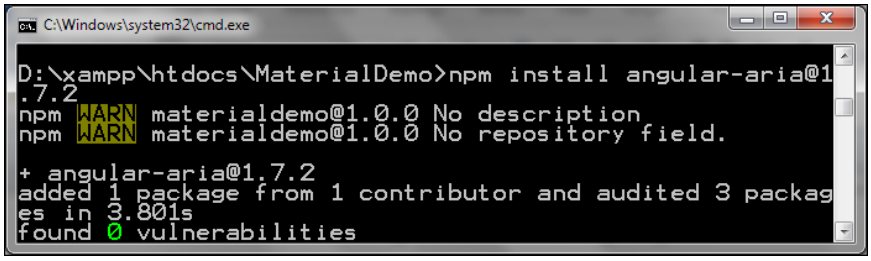


Figure 10.5: Installing AngularJS Aria

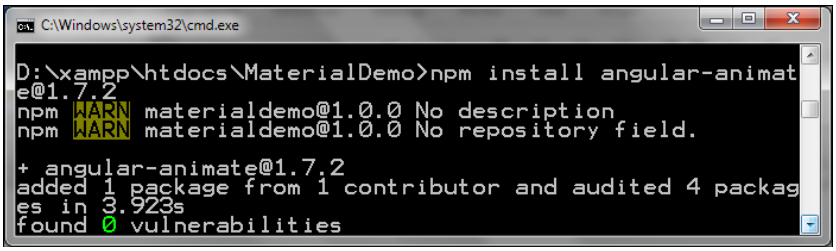


Figure 10.6: Installing AngularJS Animate

Since we wish to install an old version of Material that can work with AngularJS 1.7.9, we explicitly specify the version of AngularJS Material as 1.2.22. Refer to Figure 10.7.

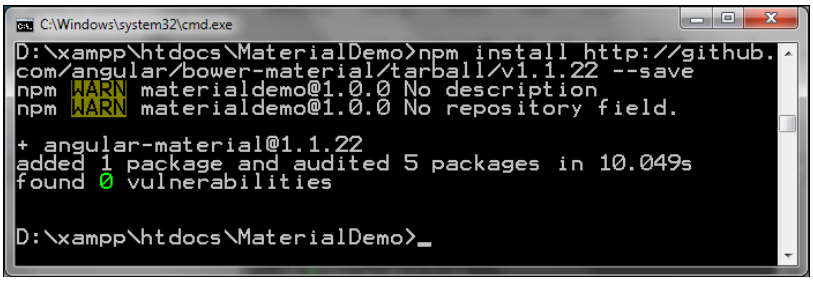


Figure 10.7: Installing AngularJS Material

`npm` will install all these libraries and packages under */node_modules/angular-material/* path of your project. After AngularJS libraries are installed, you must include the stylesheet and relevant scripts in your main HTML file, in proper order. An example of this is shown in Code Snippet 1.

Code Snippet 1: Including the Local Stylesheet and Scripts

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="initial-scale=1, maximum-scale=1,
user-scalable=no" />
  <link rel="stylesheet"
href="node_modules/angular-material/angular-material.css">
</head>
  <body ng-app="MaterialDemo">
  Beginning to work with AngularJS Material
  <div ng-controller="MatController">
  </div>
  <script src="node_modules/angular/angular.js"></script>
  <script src="node_modules/angular-aria/angular-aria.js"></script>
  <script src="node_modules/angular-animate/angular-animate.js"></script>
  <script src =
"node_modules/angular-messages/angular-messages.js"></script>
  <script src =
```

```
"node_modules/angular-material/angular-material.js"></script>
  <script>
    // Include app dependency on ngMaterial
    angular.module('MaterialDemo', ['ngMaterial', 'ngMessages'])
      .controller("MatController", function() {
});
  </script>
</body>
</html>
```

As of now, only the libraries are referenced in the code, but Material functionality is not yet added so the output will not yield anything fancy, only a line 'Beginning to work with AngularJS Material'. Refer to Figure 10.8.

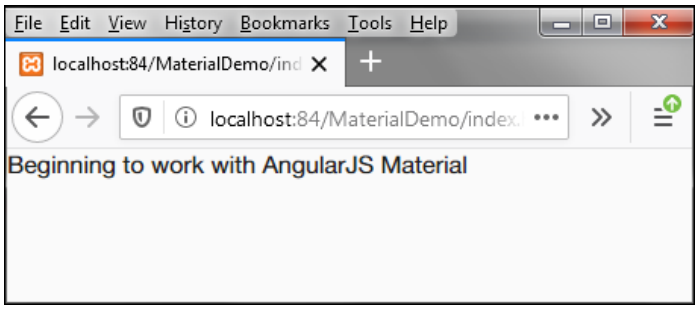


Figure 10.8: Output of Code Snippet 1

This approach, as you can see, is quite cumbersome, involving many local installations. Hence, developers prefer to use the CDN approach.

Using the CDN

With Google CDN, developers will not require to download local copies of the libraries. Instead, they can provide references to CDN URLs in their application files and use functionalities that are provided.

Create a folder named *MaterialExample* under XAMPP htdocs path. Add the code shown in Code Snippet 2 into an HTML file and save as *index.html*.

Code Snippet 2: Including the CDN Stylesheet and Scripts

```
<!DOCTYPE html>
<html>
<head>
 <!-- AngularJS Material CSS via Google CDN; version 1.1.22 is used here -->
    <link rel="stylesheet"
href="https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angular-
material.min.css">
</head>
<body ng-app="MaterialExample">
 Beginning to work with AngularJS Material
    <!--Required AngularJS Material Dependencies -->
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js"></
script>
    <script src =
"https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
animate.min.js">
</script>
    <script src =
"https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-aria.min.js">
</script>
```

```
 <script src =
"https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
messages.min.js"></script>
<!-- AngularJS Material JavaScript via Google CDN version 1.1.22 used here-->
    <script
src="https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angular-
material.min.js"></script>
</body>
</html>
```

The output will be similar to Figure 10.8. However, this approach has been easier and simpler to implement.

10.2 Customization

Customizations allow developers to make changes according to specific requirements. In AngularJS Material, customizations are allowed in terms of following components:

- Cascading Stylesheets
- Themes
- Performance

10.2.1 CSS

These classes are provided for Typography, Buttons, and Checkboxes.

- **Typography**

  It can be used to create visual consistency in AngularJS applications.

  General Typography

  Roboto font will be used by AngularJS Material for all of its components. AngularJS Material will not load this automatically. Mainly developers are responsible for loading all fonts used in their applications.

  Code Snippet 3 shows a sample markup for loading Roboto font using CDN.

  Code Snippet 3:

```
<!DOCTYPE html>
<html>
<head>
<!-- AngularJS Material CSS via Google CDN version 1.1.22 is used here -->
<link rel="stylesheet" href =
 "https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angular-
material.min.css">
<link href =
   "http://fonts.googleapis.com/css?family=Roboto:400,100,100italic,300,
   300 italic,400italic,500,500italic,700,700italic,900italic,900"
   rel="stylesheet" type="text/css">
</head>
<body style="font-family: 'Roboto', sans-serif;font-size: 48px;">
    Using Roboto Font
  <script src =
   "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js
   ">
    </script>
    <script src =
   "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
   animate.min.js">
```

```
    </script>

<script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
    aria.min.js"></script>
   <script src =
  "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
  messages.min.js"></script>
   <script src =
      "https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angu
      lar-material.min.js"></script>
</body>
</html>
```

If in case Roboto font is not loaded, the typography will fall back to Sans-Serif. Figure 10.9 shows the output of Code Snippet 3.

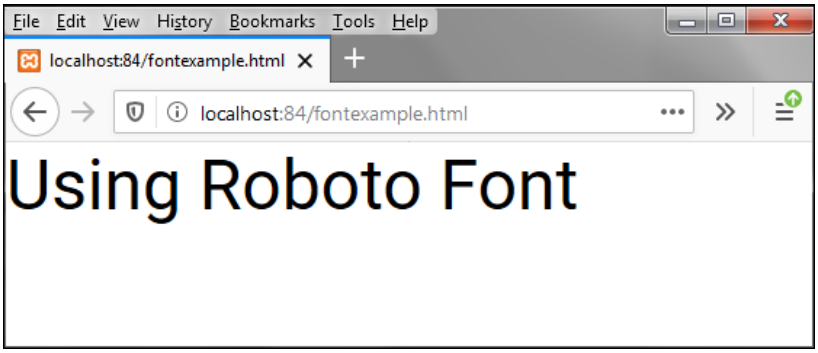


Figure 10.9: Using Roboto Font

Heading Styles

Heading tags <h1> - <h6> will be styled with styling class selectors as shown in Figure 10.10. The prefix md is applied to indicate Material Design.

Figure 10.10: Heading Styles

Code Snippet 4 shows use of various heading styles with typography selectors.

Code Snippet 4:

```
<!DOCTYPE html>
<html>
<head>
   <link rel="stylesheet"
href="https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angular-
material.min.css">
</head>
<body>
  <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js"></
script>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
animate.min.js"></script>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
aria.min.js"></script>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
messages.min.js"></script>
   <script
src="https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angular-
material.min.js"></script>
<h5 flex="" aria-describedby="headings-output" class="md-display-4 docs-
output flex">Light 112px</h5>
<h5 aria-describedby="headings-output" class="md-display-3 docs-
output">Regular 56px</h5>
<h5 aria-describedby="headings-output" class="md-display-2 docs-
output">Regular 45px</h5>
<h5 aria-describedby="headings-output" class="md-display-1 docs-
output">Regular 34px</h5>
<h5 aria-describedby="headings-output" class="md-headline docs-
output">Regular 24px</h5>
<h5 aria-describedby="headings-output" class="md-title docs-output">Medium
20px</h5>
<h5 aria-describedby="headings-output" class="md-subhead docs-output">Regular
16px</h5>
</body>
</html>
```

Figure 10.11 depicts the output of the code.

Figure 10.11: Heading Styles with Typography Selectors

In addition, body styles can be used. Figure 10.12 lists some of the styles.

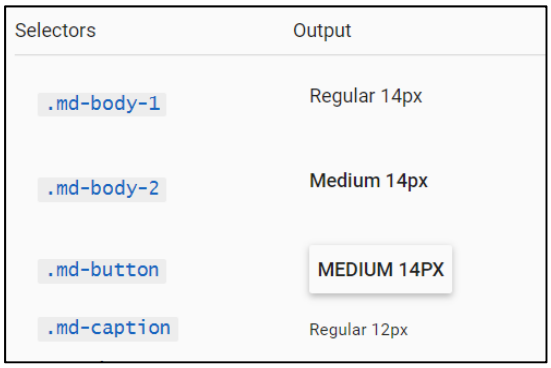


Figure 10.12: Body Styles

Code Snippet 5 shows some examples of using body styles.

Code Snippet 5:

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" href =
"https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angular-
material.min.css">
</head>
<body>
<script src =
"https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js"></scri
pt>
<script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
animate.min.js">
</script>
<script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
aria.min.js">
</script>
<script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
messages.min.js"></script>
<script src =
"https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angular-
material.min.js"></script>
<p class="md-body-2">Body style with medium weight.</p>
<md-button>Button</md-button>
<p class="md-body-1">Regular body style <small class="md-caption">with small
text</small>.</p>
<span class="md-caption">Caption</span>
</body>
</html>
```

Buttons

Code Snippet 6 shows use of AngularJS Material <md-button> classes with the default themes and standard options. It is useful to create different types of buttons as per requirement inside your application. For the icon buttons, you can use various icons. The complete list of icons with their names is given at: https://material.io/resources/icons/?style=baseline.
Save Code Snippet 6 as *materialbuttons.html*.

Code Snippet 6:

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" href =
    "https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angular-
    material.min.css">
    <script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
    </script>
    <script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
    animate.min.js"></script>
    <script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
    aria.min.js"></script>
    <script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
    messages.min.js"></script>
    <script src =
    "https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angular-
    material.min.js"></script>
      <link rel = "stylesheet" href =
     "https://fonts.googleapis.com/icon?family=Material+Icons">
     <style>
         .buttondemo section {
            background: #f7f7f7;
            border-radius: 3px;
            text-align: center;
            margin: 1em;
            position: relative !important;
            padding-bottom: 10px;
         }
         .buttondemo md-content {
            margin-right: 7px;
         }
         .buttondemo section .md-button {
            margin-top: 16px;
            margin-bottom: 16px;
         }
         .buttondemo .label {
            position: absolute;
            bottom: 5px;
            left: 7px;
            font-size: 14px;
            opacity: 0.54;
         }
    </style>
    <script language = "javascript">
         angular
            .module('MatApplication', ['ngMaterial'])
            .controller('buttonController', buttonController);
         function buttonController ($scope) {
            $scope.title1 = 'Button';
            $scope.title4 = 'Warn';
            $scope.isDisabled = true;
            $scope.googleUrl = 'http://google.com';
         }
   </script>
   </head>
```

```
   <body ng-app = "MatApplication">
      <div class = "buttondemo" ng-controller = "buttonController">
         <md-content>
            <section layout = "row" layout-sm = "column" layout-align =
            "center center" layout-wrap>
               <md-button>{{title1}}</md-button>
               <md-button md-no-ink class = "md-primary">Active (md-no-
               ink)</md-button>
               <md-button ng-disabled = "true" class =
                "md-primary">Disabled</md-button>
               <md-button class = "md-warn">{{title4}}</md-button>
               <div class = "label">Flat Buttons</div>
            </section>
            <section layout = "row" layout-sm = "column" layout-align =
              "center center" layout-wrap>
               <md-button class = "md-raised">Button</md-button>
               <md-button class = "md-raised md-primary">Active</md-button>
               <md-button ng-disabled = "true" class = "md-raised
                md-primary">
                  Disabled</md-button>
               <md-button class = "md-raised md-warn">Warn</md-button>
               <div class = "label">Raised Buttons</div>
            </section>
            </section>
            <section layout = "row" layout-sm = "column" layout-align =
             "center center" layout-wrap>
               <md-button ng-href = "{{googleUrl}}" target = "_blank">
                  Default Link</md-button>
               <md-button class = "md-primary" ng-href = "{{googleUrl}}"
                  target = "_blank">Primary Link</md-button>
               <md-button>Default Button</md-button>
               <div class = "label">Link vs. Button</div>
            </section>
            <section layout = "row" layout-sm = "column" layout-align =
               "center center" layout-wrap>
               <md-button class = "md-icon-button md-primary" aria-label =
               "Add">
               <md-icon class = "material-icons">alarm</md-icon>
               </md-button>
               <md-button class = "md-icon-button md-accent" aria-label =
               "Add">
               <md-icon class = "material-icons">build</md-icon>
               </md-button>
               <md-button class = "md-icon-button" aria-label = "Add">
               <md-icon class = "material-icons">backup</md-icon>
               </md-button>
               <md-button href = "http://google.com"
                  title = "Open Google.com in new window"
                  target = "_blank"
                  ng-disabled = "true"
                  class = "md-icon-button launch" >
                  <md-icon  class = "material-icons">gavel</md-icon>
               </md-button>
               <div class = "label">Icon Buttons</div>
            </section>
         </md-content>
      </div>
   </body>
</html>
```

Code Snippet 6 shows how we can define different kinds of button styles, such as plain buttons, buttons with colorful fonts, raised buttons, link buttons, and icon buttons. These button styles can help to make our Web applications visually appealing. Based on application purpose and requirements, specific styles can be used.

The output of Code Snippet 6 is shown in Figure 10.13. The first row shows plain/flat buttons. It also shows how we can apply ink color to buttons. The second row shows raised buttons. The third row shows link buttons and the fourth row shows icon buttons with different icon styles.

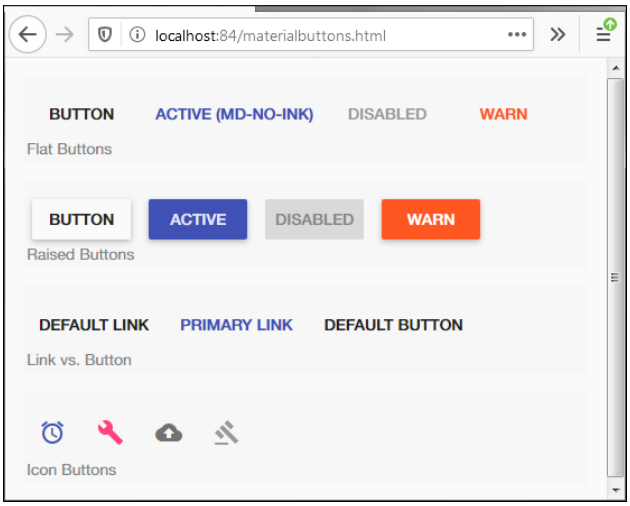


Figure 10.13: Button CSS Classes

Check boxes

CSS declaration and usage of `<md-checkbox>` component is shown in Code Snippet 7. Its corresponding output is shown in Figure 10.14. Following are some of the key properties of this component:

- **Ink color**
  It is useful to change the color when the check box is checked as shown in Figure 10.14.
- **Disabled**
  It is useful to change the color when a disabled check box is checked as shown in Figure 10.14.
- **Borders**
  It is useful to add a custom borders as shown in Figure 10.14.

Code Snippet 7:

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href =
   "https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angular-
   material.min.css">
  <script src =
  "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js">
  </script>
   <script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
    animate.min.js"></script>
    <script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
    aria.min.js"></script>
    <script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
```

```
    messages.min.js"></script>
    <script src =
    "https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angular-
material.min.js"></script>
    <style>
        .right {
         display: inline-block;
         width: 90%;
          float: right;
         }
      md-checkbox.md-checked.green .md-icon {
            background-color: rgba(0, 255, 0, 0.87);
      }
      md-checkbox.md-checked[disabled].red .md-icon {
            background-color: rgba(255, 0, 0, 0.26);
      }
      md-checkbox.dotted .md-icon {
            border-width: 1px;
            border-style: dashed;
      }
     </style>
     <script language = "javascript">
         angular.module('MatApplication', ['ngMaterial'])
            .controller('checkBoxController', checkBoxController);
         function checkBoxController ($scope) {
         }
      </script>
   </head>
  <body ng-app = "MatApplication">
   <div class="right">
      <h5>Checkboxes</h5>
      <md-checkbox ng-model = "isChecked" aria-label = "Married">
         Married
      </md-checkbox>
      <md-checkbox md-no-ink ng-model = "hasInk" aria-label = "No Ink
       Effects">
         Single
      </md-checkbox>
      <md-checkbox ng-disabled = "true" ng-model = "isDisabled" aria-label
       = "Disabled">Other (Disabled)</md-checkbox> </br>
      <h5>INK Checkbox</h5>
      <md-checkbox class="green">Green Checkbox</md-checkbox> </br>
      <h5>Disabled Checkbox</h5>
       <md-checkbox ng-disabled="true" class="red" ng-model="data.cb4"
      ng-init="data.cb4=true">
      Checkbox: Disabled, Checked
      </md-checkbox>
      <h5>Bordered Checkboxes</h5>
      <div>
      <md-checkbox ng-model="data.cb2" aria-label="Checkbox 2" ng-true-
       value="'yup'" ng-false-value="'nope'">
      Default Border
      </md-checkbox>
      </div>
      <div>
       <md-checkbox ng-model="data.cb2" aria-label="Checkbox 2"
        ng-true-value="'yup'" class="dotted" ng-false-value="'nope'">
      Custom Border
      </md-checkbox>
      </div>
     </div>
```

```
    </body>
</html>
```

Code Snippet 7 shows the code to create different styles of check boxes. Instead of using default styles, we can enhance the look and feel of our UI pages with various styles. The output is shown in Figure 10.14.

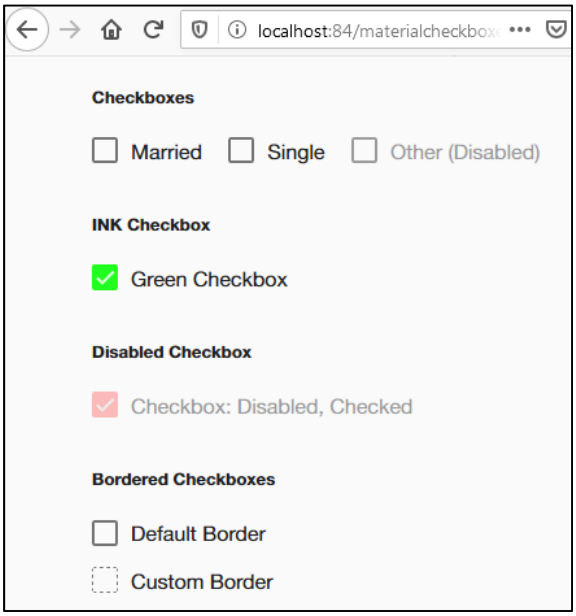


Figure 10.14: Checkbox CSS Classes

10.2.2 Theming

To innovate User Experiences (UX) and User Interface (UI) elements, Material Design visual language is used with certain specifications. Color, tones, and contrasts are conveyed through themes, that are similar to layouts.

Themes have color palettes, alphas, and shadows to deliver a consistent tone in application and provide unified feel for all AngularJS Material components. Thus, theming allows customizing the look and feel of AngularJS application. For custom styling, it is required to write CSS rules with custom selectors, or build a custom version of angular-material.css file using Syntactically awesome Style Sheets (SaSS) which extends CSS to add more features and custom variables.

Theming Approach

AngularJS Material makes it easy to design an application that follows design patterns suggested by Material style guide. Selection of colors should be limited. Choose three color hues from the primary palette and one accent color from the secondary palette. Accent color may or may not require fallback options. It useful to centralize AngularJS Material approaches for theming.

Following are few important terms related to themes:

Hues or Shades	Palettes	Color Intentions
It refers to a single color within a palette.	It is a collection of hues.	It is a mapping of a palette for a given intention into the application.

Following are valid intentions in AngularJS Material:

- primary - represent primary interface elements for a user
- accent - represent secondary interface elements for a user
- warn - represent interface elements that the user should be careful of

Theme Configuration

AngularJS Material components use the intention group classes such as `md-primary`, `md-accent`, and `md-warn`. Additional classes for color differences are include `md-hue-1`, `md-hue-2`, or `md-hue-3`.

Components used with earlier mentioned classes are as follows:

- `md-button`
- `md-checkbox`
- `md-progress-circular`
- `md-progress-linear`
- `md-radio-button`
- `md-slider`
- `md-switch`
- `md-tabs`
- `md-text-float`
- `md-toolbar`

Themes are configured using `$mdThemingProvider` during configuration process of applications.

Following properties are used to assign different color palettes:

- `primaryPalette`
- `accentPalette`
- `warnPalette`
- `backgroundPalette`

Code Snippets 8 and 9 show the use of default and custom themes in AngularJS application.

Code Snippet 8:

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" href =
   "https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angula
     r-material.min.css">
   <script src =
  "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js"
    >
    </script>
    <script src =
  "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
  animate.min.js"></script>
    <script src =
  "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
  aria.min.js">
  </script>
  <script src =
  "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
  messages.min.js">
 </script>
    <script
   src="https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/an
     gular-material.min.js"></script>
         <script language = "javascript">
```

```
         angular
            .module('MatApplication', ['ngMaterial'])
            .controller('themeController', themeController)
            .config(function($mdThemingProvider) {
               $mdThemingProvider.theme('customTheme')
               .primaryPalette('grey')
               .accentPalette('orange')
               .warnPalette('red');
            });
         function themeController ($scope) {
         }
      </script>
   </head>
      <body ng-app = "MatApplication">
           <div id = "themeContainer" ng-controller = "themeController as
         ctrl" ng-cloak>
         <md-toolbar class = "md-primary">
            <div class = "md-toolbar-tools">
               <h2 class = "md-flex">Default Theme</h2>
            </div>
         </md-toolbar>
         <hr/>
          <md-card>
            <md-card-content layout = "column">
               <md-toolbar class = "md-primary">
                  <div class = "md-toolbar-tools">
                     <h2 class = "md-flex">Using md-primary style</h2>
                  </div>
               </md-toolbar>
                <md-toolbar class = "md-primary md-hue-1">
                  <div class = "md-toolbar-tools">
                     <h2 class = "md-flex">Using md-primary md-hue-1
                    style</h2>
                  </div>
               </md-toolbar>
                <md-toolbar class = "md-primary md-hue-2">
                  <div class = "md-toolbar-tools">
                     <h2 class = "md-flex">Using md-primary md-hue-2
                    style</h2>
                  </div></md-toolbar>
                <md-toolbar class = "md-accent">
                  <div class = "md-toolbar-tools">
                     <h2 class = "md-flex">Using md-accent style</h2>
                  </div>
               </md-toolbar>
                 <md-toolbar class = "md-accent md-hue-1">
                  <div class = "md-toolbar-tools">
                     <h2 class = "md-flex">Using md-accent md-hue-1
          style</h2>
                  </div>
               </md-toolbar>
                  <md-toolbar class = "md-accent md-hue-2">
                  <div class = "md-toolbar-tools">
                     <h2 class = "md-flex">Using md-accent md-hue-2
                  style</h2>
                  </div>
               </md-toolbar>
                  <md-toolbar class = "md-warn">
                  <div class = "md-toolbar-tools">
                     <h2 class = "md-flex">Using md-warn style</h2>
                  </div>
```

```
               </md-toolbar>
                  <md-toolbar class = "md-warn md-hue-1">
                  <div class = "md-toolbar-tools">
                   <h2 class = "md-flex">Using md-warn md-hue-1
                   style</h2>
                  </div>
               </md-toolbar>
                     <md-toolbar class = "md-warn md-hue-2">
                  <div class = "md-toolbar-tools">
                     <h2 class = "md-flex">Using md-warn md-hue-2
                     style</h2>
                  </div>
               </md-toolbar>
            </md-card-content>
         </md-card>
      </div>
   </body>
</html>
```

Code Snippet 9:

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" href =
   "https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angula
     r-material.min.css">
   <script src =
  "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js"
    >
    </script>
    <script src =
  "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
  animate.min.js"></script>
    <script src =
  "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
  aria.min.js">
  </script>
  <script src =
  "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
  messages.min.js">
 </script>
   <script
  src="https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/an
    gular-material.min.js"></script>
        <script language = "javascript">
        angular
           .module('MatApplication', ['ngMaterial'])
           .controller('themeController', themeController)
           .config(function($mdThemingProvider) {
              $mdThemingProvider.theme('customTheme')
              .primaryPalette('grey')
              .accentPalette('orange')
              .warnPalette('red');
           });
        function themeController ($scope) {
        }
     </script>
  </head>
     <body ng-app = "MatApplication">
          <div md-theme = "customTheme">
```

```
            <md-toolbar class = "md-primary">
               <div class = "md-toolbar-tools">
                  <h2 class = "md-flex">Custom Theme</h2>
               </div>
            </md-toolbar>
            <hr/>
             <md-card>
               <md-card-content layout = "column">
                  <md-toolbar class = "md-primary">
                     <div class = "md-toolbar-tools">
                        <h2 class = "md-flex">Using md-primary style</h2>
                     </div>
                  </md-toolbar>
                  <md-toolbar class = "md-primary md-hue-1">
                  <div class = "md-toolbar-tools">
                 <h2 class = "md-flex">Using md-primary md-hue-1 style</h2>
                     </div>
                  </md-toolbar>
                  <md-toolbar class = "md-primary md-hue-2">
                  <div class = "md-toolbar-tools">
                 <h2 class = "md-flex">Using md-primary md-hue-2 style</h2>
                     </div>
                  </md-toolbar>
                   <md-toolbar class = "md-accent">
                     <div class = "md-toolbar-tools">
                        <h2 class = "md-flex">Using md-accent style</h2>
                     </div>
                  </md-toolbar>
                  <md-toolbar class = "md-accent md-hue-1">
                <div class = "md-toolbar-tools">
                <h2 class = "md-flex">Using md-accent md-hue-1 style</h2>
                </div>
                </md-toolbar>
                <md-toolbar class = "md-accent md-hue-2">
                <div class = "md-toolbar-tools">
                <h2 class = "md-flex">Using md-accent md-hue-2 style</h2>
                </div>
                  </md-toolbar>
                   <md-toolbar class = "md-warn">
                     <div class = "md-toolbar-tools">
                        <h2 class = "md-flex">Using md-warn style</h2>
                     </div>
                  </md-toolbar>
                    <md-toolbar class = "md-warn md-hue-1">
                     <div class = "md-toolbar-tools">
                    <h2 class = "md-flex">Using md-warn md-hue-1 style</h2>
                     </div>
                  </md-toolbar>
                    <md-toolbar class = "md-warn md-hue-2">
                     <div class = "md-toolbar-tools">
                    <h2 class = "md-flex">Using md-warn md-hue-2 style</h2>
                     </div>
                  </md-toolbar>
               </md-card-content>
            </md-card>
           </div>
      </div>
   </body>
</html>
```

Figures 10.15 and 10.16 display the outputs for Code Snippet 8 and 9 respectively that are used to create default and custom themes.

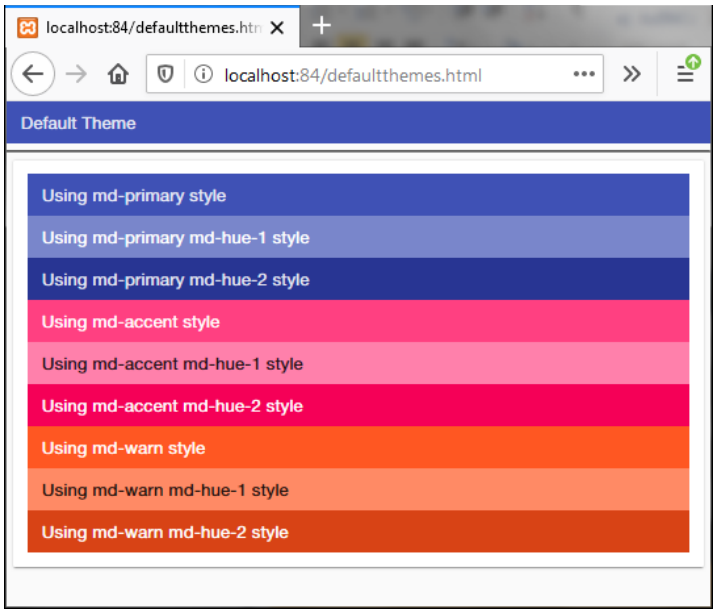


Figure 10.15: Default Theme

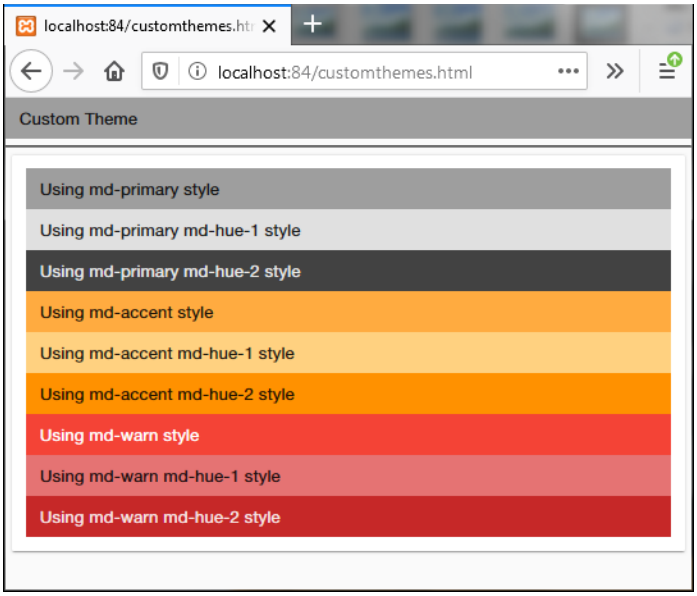


Figure 10.16: Custom Themes

10.2.3 Browser Support and Performance Issues

AngularJS Material is supported on latest versions of most major browsers such as Chrome, EDGE, Opera, Safari, Firefox, and IE.

Table 10.1 depicts this in detail.

Browser	Supported Versions
Chrome Chrome for Android Edge Opera	Last 2 versions

Browser	Supported Versions
Firefox	Last 2 versions ESR
IE IE Mobile	11
Safari	11.x 12.x
iOS	11.x 12.x
Firefox for Android UC QQ Baidu	Latest version
Android Browser	4.4.3-4.4.4 Latest version
Samsung Internet	4 6.2 7.2
Opera for Android	Mini all

Table 10.1: Angular Material Browser Support

However, Internet Explorer 11 has some performance issues with few features of AngularJS Material, such as layout calculations, animations, and border rendering.

10.3 Layouts and Containers

Layouts

Features of layout enable developers to easily create modern and responsive layouts. API for layout contains a set of AngularJS directives which are applied to any HTML element in an application.

Attributes are useful to define layout and CSS classes are useful to assign styling. Layout directive on top of the container element is useful to specify layout direction for its children.

Following values are used for the Layout Directive:

row

Items are arranged horizontally. `max-height` is given as 100% and `max-width` is set as the width of container in which item is present.

column

Items are arranged vertically. `max-width` is given as 100% and `max-height` is the height of container in which items are present.

Following APIs can be used to set the layout direction for devices with their view widths:

- **layout**
  It sets default layout direction until another breakpoint is provided.
- **layout-xs**
  width < 600px
- **layout-gt-xs**
  width >= 600px
- **layout-sm**
  600px <= width < 960px

- **layout-gt-sm**
  width >= 960px
- **layout-md**
  960px <= width < 1280px
- **layout-gt-md**
  width >= 1280px
- **layout-lg**
  1280px <= width < 1920px
- **layout-gt-lg**
  width >= 1920px
- **layout-xl**
  width >= 1920px

Code Snippet 10 displays use of layout directive.

Code Snippet 10:

```
<!DOCTYPE html>
<html>
   <head>
    <link rel="stylesheet"
    href="https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/an
    gular-material.min.css">
    <script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js"
     >
    </script>
    <script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
    animate.min.js">
    </script>
    <script src =
    "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
    aria.min.js"></script>
    <script src =
   "https://ajax.googleapis.com/ajax/libs/angularjs/1.7.2/angular-
   messages.min.js"></script>
    <script src =
    "https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angular
    -material.min.js">
    </script>
    <style>
         .box {
            color:white;
            padding:10px;
            text-align:center;
            border-style: inset;
         }
         .orange {
            background:orange;
         }
        .gray {
            background:gray;
         }
     </style>
     <script language = "javascript">
         angular
            .module('firstApplication', ['ngMaterial'])
            .controller('layoutController', layoutController);
         function layoutController ($scope) {
         }
```

```
    </script>
  </head>
  <body ng-app = "firstApplication">
 <div id = "layoutContainer" ng-controller = "layoutController as ctrl"
       style = "height:100px;" ng-cloak>
       <div layout = "row" layout-xs = "column">
          <div flex class = "orange box">1st Row: Item 1</div>
          <div flex = "50" class = "gray box">1st Row: Item 2</div>
       </div>
           <div layout = "column" layout-xs = "column">
          <div flex = "33" class = "orange box">1st Column: item 1</div>
          <div flex = "66" class = "gray box">1st Column: item 2</div>
       </div>
 </div>
</body>
</html>
```

Figure 10.17 depicts the output.

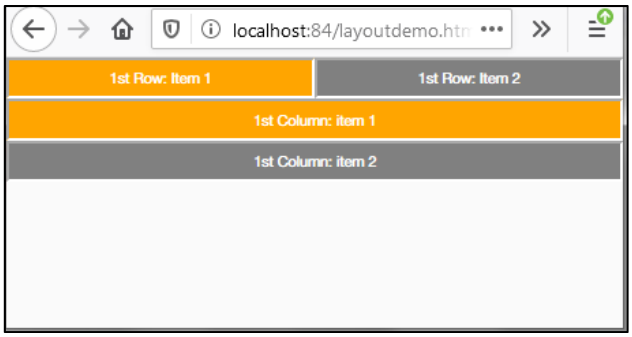


Figure 10.17: Use of Layout

`flex` directive is useful to customize size and position of the HTML elements. It helps to define and adjust size with respect to the parent container and other elements within the container.

Following values can be used for `flex` directive:

- 5 – 5, 10, 15 ... 100
- 33 – 33%
- 66 – 66%

Containers

Containers are generally used to create overlaying elements such as dialogs, menus, navigation bars, and so on. Container components are programmable and used to show or hide elements. Code Snippet 11 demonstrates how to use `md-sidenav` programmatically.

Code Snippet 11:

```
<!DOCTYPE html>
<html>
   <head>
      <link rel="stylesheet"
href="https://ajax.googleapis.com/ajax/libs/angular_material/1.1.22/angular-
material.min.css">
      <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.0/angular.min.js"></
script>
```

```
      <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.0/angular-
animate.min.js"></script>
      <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.0/angular-
aria.min.js"></script>
      <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.0/angular-
messages.min.js"></script>
      <script src =
"https://ajax.googleapis.com/ajax/libs/angular_material/1.0.0/angular-
material.min.js"></script>
      <link rel = "stylesheet" href =
"https://fonts.googleapis.com/icon?family=Material+Icons">
        <style>
         .orange {
            background:orange;
         }
         .pink {
            background:pink;
         }
      </style>
        <script language = "javascript">
         angular
            .module('firstApplication', ['ngMaterial'])
            .controller('sideNavController', sideNavController);

         function sideNavController ($scope, $mdSidenav) {
            $scope.openLeftMenu = function() {
               $mdSidenav('left').toggle();
            };

            $scope.openRightMenu = function() {
               $mdSidenav('right').toggle();
            };
         }
      </script>
    </head>
    <body ng-app = "firstApplication">
      <div id = "sideNavContainer" ng-controller = "sideNavController as
     ctrl"  layout = "row" layout-align="center" ng-cloak>
         <md-sidenav md-component-id = "left" class = "md-sidenav-left;
     orange" >
            <div>Welcome to Container</div></md-sidenav>
            <md-content>
            <md-button ng-click = "openLeftMenu()">Click to Open Left Menu
      </md-button>
            <md-button ng-click = "openRightMenu()">Click to Open Right
      Menu</md-button>
         </md-content>
        <md-sidenav md-component-id = "right" class = "md-sidenav-right">
            <md-button href = "http://google.com" class = "pink">Visit
        Google</md-button>
         </md-sidenav>
      </div>
   </body>
</html>
```

Figures 10.18 to 10.20 depict the output of the code.

Figure 10.18: Main Menu

Figure 10.19: Left Menu

Figure 10.20: Right Menu

10.4 Migration to Angular Material

Migration from AngularJS Material to Angular Material involves multiple aspects such as features that have been moved to Angular Component Development Kit (CDK), changes in theming system, separate library for AngularJS material, and so on.

AngularJS Material has not yet entered Long Term Support (LTS) whereas AngularJS has and due to this, Angular Components team's resources are focused on Angular Material and Angular CDK (CDK). For applications with long-term development and support plans, considerations are made for migration in the latest version of Angular, Angular Material, and CDK.

Official ngUpgrade guide covers general practices around AngularJS to Angular migration. This document provides additional guidance specific to AngularJS Material (v1.1.9+). The https://angular.io/guide/upgrade#preparation and https://material.angularjs.org/latest/migration guides cover a number of important steps and other details for preparing applications for the migration.

Quick Test 10.1

1. Flex directive is useful to customize size and position of the HTML elements.
    a. True
    b. False

2. Themes are configured using `$mdThemingProvider` during configuration process.
    a. True
    b. False

10.5 Summary

- AngularJS Material is a UI Component library that provides a set of reusable, well-tested, and accessible UI components for AngularJS developers.
- Typography is used to create visual consistency in AngularJS applications.
- `md-button` class is used to create different types of buttons such as Fab, Flat, Raised, and so on.
- `md-checkbox` class is used to create check boxes with different styles.
- Layout enables developers to create modern, responsive layouts on CSS3 flex box.
- `flex` directive is useful to customize size and position of the HTML elements.
- Overlaying elements are created using containers.
- AngularJS Material to Angular Material migration is supported through various helpful documents and guides.

10.6 Exercise

1. You need to have any version of AngularJS between _____ in order to work with AngularJS Material.

    a. 1.7.2 to 1.8.x
    b. 1.8 to 1.9
    c. 1.6 to 1.7
    d. 1.6.2 to 2.0x

2. _____ is a collection of hues.

    a) Shades
    b) Color Intention
    c) Palettes
    d) CSS

3. Which of the following values are used by Layouts?

    a) Row
    b) Height
    c) Column
    d) Width

4. Which of the following types of buttons can be created with class md-button?

    a) Flat
    b) Fab
    c) Raised
    d) All of these

5. Ink color is useful to change the color only when the _____ is checked.

    a) Checkbox
    b) Link
    c) Button
    d) Textbox

10.7 Do It Yourself

1. Jenny has started her own cupcake baking business. She has sought out your aid in creating an Order application for her. Create an AngularJS 1.7.9 application that accepts orders via form for the following:
    a. Customer Name
    b. Customer Address
    c. Order Date
    d. Email Id
    e. Number of Cupcakes

    Enhance it with visually appealing user interface elements using AngularJS Material.

2. Peter is setting up a company JobSeekers for aspiring job seekers. You have to help him by creating a page that contains a basic Registration form. You do this by creating an AngularJS 1.7.9 application that uses AngularJS Material to present the UI in an appealing manner.

Answers to Exercise

1. 1.7.2 to 1.8.x
2. Palettes
3. Row and Column
4. All of these
5. Checkbox

Answers to Quick Test

Quick Test 10.1

1. True
2. True