AngularJS by Example

Single-page apps (SPA) have now become mainstream and there is a rapidly growing interest in learning to build them. There is no better framework than AngularJS to build these apps. Angular has everything: a powerful view engine, an exceptional data binding framework, a capable routing engine, a supertative dependency injection framework, and a neat mechanism to organize chunks of code into modules.

This book walks you through building three sample applications, each increasing in complexity. You will learn what goes into the development of a single-page app, and the building blocks of Angular including controllers, services, directives, and filters. The book rounds off with useful tips that will come in handy as you build more and more apps with Angular.

Who this book is written for
If you are interested in working on the web stack and are always looking for better ways to organize and develop code with JavaScript, then this book is for you! You will learn how AngularJS makes JavaScript development easier, more organized, and even pretty fun.

What you will learn from this book
- Explore each and every building block of Angular starting from models to controllers, filters, services, and directives as you build multiple applications
- Work your way through every aspect of app development using Angular and design, develop, deploy, and test the apps you build
- Understand the inner workings of the AngularJS view templating and data-binding capabilities
- Create directives to extend standard HTML behavior
- Get to grips with how to manage code organization using modules and component dependencies with the dependency injection framework
- Work on HTML forms and learn the Angular way to bind, validate, and submit form data

The essential DIY guide to help you get up and running with the most popular JavaScript framework
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Getting Started"
- A synopsis of the book’s content
- More information on AngularJS by Example

About the Author

Chandermani is a software craftsman with a passion for technology and is an expert on the web stack. With more than a decade of experience under his belt, he has architected, designed, and developed solutions of all shapes and sizes on the Microsoft platform.

He never believed JavaScript could be a great platform for app development until he saw Angular. Such is his love for this framework that every engagement he is part of has an Angular footprint.

Being an early adopter of the Angular framework, he tries to support the platform in every possible way, whether by writing blog posts on various Angular topics or helping his fellow developers on Stack Overflow, where he is often seen answering questions on AngularJS channels.

A former MSFT employee, he now works for Technovert, where he leads a bunch of awesome developers who build cloud-scale web applications using AngularJS and other new-age frameworks.

Writing this book has just been a surreal experience and I would like to thank my Technovert family who supported me in all possible ways, be it helping me with sample apps, reviewing the content, or offloading some of my professional commitments to make sure I get enough time to write the book.

I especially want to thank Vijay Yalamanchili, who inspired me to take up this endeavor and also made sure there are no impediments while I juggle my professional and book writing commitments.

And finally, I want to express my gratitude toward my family. I know your blessings are always with me.
AngularJS by Example

The first thing I must do is to congratulate you! You have made an excellent decision in choosing to learn this super awesome JavaScript framework: AngularJS. Rest assured you won't be disappointed, either by AngularJS, or by this book.

JavaScript has come a long way. I remember when I started using JavaScript (around 10 years ago), it was primarily used for doing some client-side form validation and some animation, nothing very serious. Developers hacked their solution using JavaScript without much understanding of the language and its capabilities.

As JavaScript became more mainstream, browsers became more capable, jQuery hit the scene, and developers started testing the limits of the language and browser capabilities.

In recent years, a new breed of JavaScript applications has hit the fancy of the developer community, Single Page Applications (SPAs). These are rich client applications with no page refresh/redirects, bookmarkable URLs and they imbibe the UX design / code patterns such as MVC, MVP, MVVM, or MV*.

AngularJS is one such SPA JavaScript framework. Open sourced by Google and actively developed and supported by a vibrant community around it, this framework has gained a lot of traction. Due to its modular design, powerful features, and great performance it has become a great choice for building business apps over the JavaScript platform.

Through this book, our aim is to teach you how to effectively build apps using the AngularJS platform. We will be building multiple apps on this platform ranging from simple ones to more complex ones.

Learning by example has its advantages; you immediately see the concept explained in action. Also, if you are like me and prefer Do It Yourself (DIY) over too much theory, then this book is a perfect fit for you.

What This Book Covers

Chapter 1, Getting Started, introduces you to the AngularJS framework. We create a super simple app in AngularJS that highlights some core features of the framework.

Chapter 2, Building Our First App – 7 Minute Workout, will teach us how to build our first real AngularJS app. In the process, we learn more about the framework's MVC constructs and are introduced to scopes and binding capabilities of the framework. We learn about code organization using modules and dependency injection, look at view routing in action, and learn how to use filters.
Chapter 3, More AngularJS Goodness for 7 Minute Workout, focuses on adding the bells and whistles to the 7 Minute Workout app and in the process touches upon into some new Angular capabilities. This chapter covers Angular services, digest cycles, animation support, filters and few other concepts.

Chapter 4, Building Personal Trainer, introduces a new exercise where we morph 7 Minute Workout into a generic Personal Trainer app. The new app has the capability to create new workout plans other than the original 7 Minute Workout app. In this chapter, we exclusively focus on AngularJS form capabilities.

Chapter 5, Adding Data Persistence to Personal Trainer, covers retrieving and saving data to the server. We augment our Personal Trainer with data load and data persistence capabilities. Server interaction and data persistence have been ignored in all the previous chapters.

Chapter 6, Working with Directives, explains directives by building some of our own custom directives for the Personal Trainer app. Directives are the most powerful and most misunderstood feature of AngularJS.

Chapter 7, Testing the AngularJS App, highlights how to use the AngularJS testing constructs within Personal Trainer as AngularJS was built with testability in mind.

Chapter 8, Handling Common Scenarios, provides some practical tips and guidance around scenarios that we might encounter while developing apps on this framework. We will cover scenarios such as structuring the app for complex views, inter controller/directive communication, common framework pitfalls, authentication and authorization, code organization for large size apps, and other similar scenarios.
Developing applications in JavaScript is always a challenge. Due to its malleable nature and lack of type checking, building a decent size application in JavaScript is difficult. Moreover, we use JavaScript for all types of processes such as User Interface (UI) manipulation, client server interaction, and business processing/validations. As a result, we end up with spaghetti code that is difficult to maintain and test.

Libraries such as jQuery do a great job of taking care of various browser quirks and providing constructs that can lead to an overall reduction in lines of code. However, these libraries lack any structural guidance that can help us when the codebase grows.

This is where architectural patterns such as Model View Controller (MVC) and frameworks such as AngularJS come into the picture. This chapter is dedicated to understanding the concept behind MVC architecture and learning how to put these principles into practice while we build a simple app using AngularJS.

The topics that we will cover in this chapter are as follows:

- **MVC basics**: We will talk in brief about each of the components of the Model-View-Controller pattern.
- **Building our first AngularJS app**: We will build a small game, Guess the Number! in AngularJS.
- **Understanding the Model, View, and Controller components**: We will dissect the game that we built to understand the MVC components of AngularJS.
- **Working with scopes**: AngularJS scopes are an important concept to understand. This chapter will introduce us to scopes and the role they play in the framework.
Getting Started

- **An introduction to some AngularJS constructs**: We will introduce some new constructs such as expressions, directives, and interpolations that we used to build the *Guess the Number!* app.

- **App initialization**: We will talk about the app initialization process in AngularJS; this is also known as App bootstrapping.

- Lastly, we will provide some resources and tools that will come in handy during AngularJS development and debugging.

So let's get started with our first topic: MVC.

### The basics of Model View Controller

MVC is a UI architectural pattern that has been around for some time. It is based on the premise of separation of responsibility. In MVC, we have the following components:

- **Model**: This stores business data
- **View**: This represents the model in UI
- **Controller**: This is responsible for coordinating between a model and view

Any change in the model is reflected in the view and any change done by the user while interacting with the view is reflected back on the model. Here, the controller acts as a coordinator and is responsible for keeping the model and view in sync.

This is an over-simplified definition of the MVC and if we search the Web, we will find numerous variations of this pattern such as MVP, Front Controller, MVVM, and maybe some others. The net effect of this separation is that the code becomes more organized, more understandable, and maintainable.

For us, the best way to understand MVC is to see it in action and hence, we are going to build our first Hello World app in AngularJS. This app will help us to become familiar with the AngularJS framework and see the MVC paradigm in action.

### The customary Hello Angular app (Guess the Number!)

As our first exercise, we want to keep things simple but still showcase the framework capabilities. Therefore, we are going to build a very simple game, *Guess the Number!*. The objective of the game is to guess a computer-generated random number in as few tries as possible.
This is how the game looks:

![Guess the Number!](image)

The gameplay is simple.

We enter the guess in the textbox. To verify whether the guess is correct, we click on **Verify**. The game gives us a hint to improve our next guess. If we are unable to guess or have guessed the number correctly, we can restart the game by clicking on **Restart**.

Before we start working on the game, it is strongly recommended that we run the sample code from a development server instead of loading the app directly from the filesystem. The browser security model has some restrictions on running script content from the local filesystem. Let's quickly look at some easy-to-install options for a development server.

### Setting up a development server

The development web server that we choose greatly depends on the platform that we work on and the backend we support. However, for apps in this book that target purely client-side development, any web server will do.

My recommendation would be to use the `http-server` module of Node.js. Since Node.js is available cross-platform, we can install Node.js from [http://nodejs.org/](http://nodejs.org/).

Once Node.js is installed, installing the `http-server` module and running the `http-server` are easy. Open the command line and type the following command:

```
$ npm install http-server -g
```

This installs the HTTP server at the global level.

To run the server, just navigate to the folder where the app code resides, or open the folder from where we want to serve static files, and type this:

```
http-server
```
And that's it! We have an HTTP server running at http://localhost:8080 that can serve files from the current directory.

The http-server module does support some startup configurations. Check the documentation at https://github.com/nodeapps/http-server.

The http-server module is just one of the many options available out there. Depending upon the platform we are on, we can also try Python's SimpleHTTPServer module, Mongoose, or any such web server.

Let's build Guess the Number!

Building Guess the Number!

The standard practice when building user interfaces is to build them top-down. Start with designing the UI and then plug in the data and behavior according to the needs. With such an approach, the UI, data, and the behavioral aspects of the app are all tightly coupled, which is a less than ideal situation!

With the MVC paradigm, things work a little differently. There is a conscious effort to design the model beforehand by looking at the UI and expected behavior, and to encapsulate the behavior aspect into the controller, thus minimizing the coupling between the controller implementation and the UI (view) that it supports.

Given these facts, we will start by identifying the model data for our app based on the feature set.

The app model

The model is the data that the view and controller work on. It represents the state of the system projected on the view. To determine the model for our own app, we need to detail the features that the app supports. These features include:

- Supporting the generation of random numbers (original)
- Supporting input for a user to guess the value (guess)
- Tracking the number of guesses already made (noOfTries)
- Giving users hints to improve their guess based on their input (deviation)
- Giving a success message if the user guesses the number correctly (deviation)
Once we have the feature list, we can now determine what data needs to be tracked and that becomes part of our model. For the preceding feature set, the elements in parentheses denote properties that will support these features and hence represent the app model.

Designing the model for an app is a very crucial process. If it is done right, we can minimize the friction between a model and view and simplify the controller implementation.

While building any app, I urge you to first think about the functionality you want to offer, then the model that can support the functionality, and lastly think about how to build a view for it. This is a good practice irrespective of the library or framework you use to build your app.

The model properties highlighted earlier need to be hosted in a script and then referenced by the view. These model properties will be defined inside a controller and hence it’s time to introduce the Angular Controller.

However, before we do that, we first need to create a file for the controller code. Due to the size of the app, we are going to create a single file that will contain everything, from the controller script to the view HTML code. To start with, this is the outline of our app HTML code:

```html
<!DOCTYPE html>
<html>
<head>
  <title>Guess The Number !</title>
  <link href="http://netdna.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
  <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.3/angular.js"></script>
</body>
</html>
```

### Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at [http://www.packtpub.com](http://www.packtpub.com). If you purchased this book elsewhere, you can visit [http://www.packtpub.com/support](http://www.packtpub.com/support) and register to have the files e-mailed directly to you.
Create an HTML file and add the preceding HTML code to it. Henceforth, everything that we outline should be appended to this file. The app HTML code itself is self-explanatory. We reference the Twitter Bootstrap CSS in the <head> section and the Angular framework inside the <body> tag.

We can now start building the controller.

The controller

The controller manages the model and the view. It is always designed with the view in mind and it is the view's behavior that drives controller functionality. In AngularJS, the controller is a JavaScript class (a constructor function) that hosts the model and exposes some behavior that the view binds to. How it binds to the view will be clear when we discuss the view implementation.

Let's start working on the controller implementation. While defining our model, we have already detailed the functional aspect of the application and we do have a fair idea about how the view should behave. Keeping that in mind, this is how the app controller looks:

```javascript
function GuessTheNumberController($scope) {
    $scope.verifyGuess = function () {
        $scope.deviation = $scope.original - $scope.guess;
        $scope.noOfTries = $scope.noOfTries + 1;
    }
    $scope.initializeGame = function () {
        $scope.noOfTries = 0;
        $scope.original = Math.floor((Math.random() * 1000) + 1);
        $scope.guess = null;
        $scope.deviation = null;
    }
    $scope.initializeGame();
}
```

Add this controller script to the file created earlier after the Angular script declaration inside its own script block.

The GuessTheNumberController function sets up some model properties that we described in the The app model section and exposes two methods: verifyGuess and initializeGame.
The `verifyGuess` function verifies whether the guess matches the original value and updates model properties `deviation` and `noOfTries` accordingly. The `initializeGame` function is used to initialize the model properties before the start of the game, and during the game whenever the user clicks on the Restart button.

The last statement in the preceding controller calls `initializeGame` to set up the game for the first time.

The overall controller implementation is self-explanatory but the only oddity seems to be the `$scope` object. This `$scope` object has been passed as a parameter to the controller function and all functions and properties are attached to `$scope`. To understand the role of the `$scope` object and how things tie together, we need to start implementing the view.

However, we are still not done with the controller yet. We will revisit the controller once we get the app running and learn a bit more about them.

## The app view

The view is nothing but a UI projection of model data. Looking at the Guess the Number! UI, we will agree that this UI may not win any User Experience (UX) award; still, this HTML is what we call the view.

Let's put some focus on the view and start building it. We have already added the controller to the app file. Add this view code inside the body tag at the top:

```html
<div class="container">
  <h2>Guess the Number !</h2>
  <p class="well lead">Guess the computer generated random number between 1 and 1000.</p>
  <label>Your Guess: </label><input type="number" ng-model="guess" />
  <button ng-click="verifyGuess()" class="btn btn-primary btn-sm">Verify</button>
  <button ng-click="initializeGame()" class="btn btn-warning btn-sm">Restart</button>
  <p ng-show="deviation<0" class="alert alert-warning">Your guess is higher.</p>
  <p ng-show="deviation>0" class="alert alert-warning">Your guess is lower.</p>
  <p ng-show="deviation===0" class="alert alert-success">Yes! That's it.</p>
  <p>No of guesses : {{noOfTries}}</p>
</div>
```
Lots of interesting stuff here, but before we dissect the view implementation we need to link the view and the controller; in fact, both the view and the controller need to be linked to the Angular framework first.

To let Angular know it needs to process the view HTML, update the existing `<body>` tag with a custom attribute `ng-app`:

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

The `ng-app` attribute tells Angular to treat everything defined inside the HTML tag (with `ng-app` attribute) as an angular app and process it accordingly. The `<body>` tag just mentioned becomes the root of the application.

The "app" value of the `ng-app` attribute tells Angular to search for a module named app and load it.

We have a new concept called **module**. Modules in the Angular framework are containers that hold various artifacts that we create or are part of the framework. Any Angular-specific script implementation always goes into one or another module.

Let's define the "app" module that the `<body>` tag references. Add the highlighted statement inside the script block containing the controller declaration:

```javascript
angular.module('app', [])
  function GuessTheNumberController($scope) {
```

We use the global Angular object that is part of the Angular framework to declare an Angular module. The first parameter is the name of the module and the second is there to provide module dependencies.

We now have a module named app. This module is linked with the view using `ng-app`, hence the Angular framework can now process the view and load the module. However, the link between the controller and the Angular framework (as well as between the view and the controller) is still missing.

To make Angular aware of the controller, we need to register the controller with the module we just created. Update the module declaration now to:

```javascript
angular.module('app', [])
  .controller('GuessTheNumberController', GuessTheNumberController);
  function GuessTheNumberController($scope) {
```

The `angular.module` function creates a module and returns a module object. This module object has a function `controller` that allows us to register a controller. The first parameter is the name of the controller and the second is the controller implementation itself. With the first two lines from the preceding code, we have declared a module app and registered the `GuessTheNumberController` function with it.
Lastly, we link the controller and the view using another attribute, `ng-controller`. Update the `div` (class=container) parameter to:

```html
<div class="container" ng-controller="GuessTheNumberController">
```

Well, the app is complete and ready to be tested! Open the file in the browser and start guessing.

If you are having trouble running the app, a working HTML is available on my GitHub account [http://goo.gl/4j6DG6](http://goo.gl/4j6DG6).

If we glance at the HTML file now, we should be mightily impressed with what we have achieved with these 40 lines. There is no DOM manipulation or model-view synchronization code and still everything works perfectly.

To understand how this app functions in the Angular context, we need to delve deeper into the view HTML as the view acts as an entry point into the app and links everything together.

Let's go back and check the app HTML. It looks like standard HTML with some new attributes (`ng-app`, `ng-controller`, `ng-model`, `ng-click`, and `ng-show`) and symbols (`,`, `{`, and `}` (`{{` and `}}`).

In the Angular world, `{` and `}` are the interpolation symbols and the `ng-*` attributes are what we call directives. The model properties and functions are either assigned to these directive attributes or referenced inside interpolation symbols. Clearly, interpolation symbols and directive attributes have some behavior attached to them and seem to be linking the view HTML and controller code. Let's try to understand what these directive attributes and interpolation symbols are actually doing.

### Interpolation

Look at this HTML fragment from the *Guess the Number!* code:

```html
<p class="text-info">No of guesses :
    <span class="badge">{{ noOfTries }}</span>
</p>
```

The model properties `noOfTries` is sandwiched between two interpolation symbols. Interpolation works by replacing the content of the interpolation markup with the value of the expression (`noOfTries`) inside the interpolation symbol.
Interpolations are declared using the syntax \{expression\}. This expression looks similar to the JavaScript expression but is always evaluated in the context of the linked $scope object. Remember we passed a $scope object to the controller function, as follows:

```javascript
function GuessTheNumberController($scope) {
}
```

And we then attached the properties noOfTries to the $scope object in the implementation. Clearly, the interpolation expression can reference such properties and link it to the view. Later in this chapter, we will discuss more about the $scope object to understand how the underlying infrastructure works and the link between the controller, $scope, and the view is established.

Another interesting aspect of interpolation is that changes made to model properties are automatically synchronized with the view. Run the app and make some guesses; the noOfTries value changes after every guess and so does the view content.

Interpolation is an excellent debugging tool in scenarios where we need to see the state of the model. With interpolation, we don't have to put a breakpoint in code just to know the model value. Since interpolation can take an expression, we can pass a function, an object, or a primitive type and see its value.

Interpolations are flexible and can be placed almost anywhere in HTML:

- Inside a tag (<div>{{noOfTries}}</div>)
- As an attribute value (<div class='cls-{{noOfTries}}'>
- As an attribute name (<input {{myAttributeName}}=

Learning interpolation was easy; now let's look at the other framework construct, directives.

**Directives**

Directives, in AngularJS, are constructs that allow us to extend the standard HTML vocabulary. They help us augment the behavior of the existing HTML elements. Also, they allow us to create our own elements as well! This makes directives the most powerful and the most celebrated feature of the framework. Directives are the way to create and package reusable components in the Angular framework.
For our game, too, we use some Angular directives, including `ng-app`, `ng-controller`, `ng-model`, `ng-click`, and `ng-show`. Each of these directives extends the behavior of the HTML element on which they are defined. Here is what each of these directives do:

- **ng-model**: This sets up a link between the scope property passed in as the attribute value ('guess' in our case) and HTML input element. As we change the input value, the `guess` variable changes; it works the other way round too. In this case, we update the `guess` value in the controller as we do in the `initializeGame` function (`$scope.guess = null;`), it clears the input value. We can use interpolation to verify if `guess` indeed is changing when input changes. Append an interpolation next to input, as follows:

```html
<input type="number" ng-model="guess"/> {{guess}}
```

Save and refresh the page. If we now enter a numeric value in the input field, the guess interpolation next to it automatically changes.

- **ng-click**: When attached to an HTML element, this directive evaluates the expression passed as an attribute value when the element is clicked. So the Verify button causes the `verifyGuess` function to execute and the Restart button causes the `initializeGame` method to execute.

- **ng-show**: When attached to an element, this shows or hides the element based on the expression's return value. If true, the element is shown, otherwise it gets hidden. In our case, we use `ng-show` on the three paragraph elements and show/hide them based on the value of `deviation`.

- **ng-controller**: As we have already seen, this directive links the controller implementation with the view.

We will cover `ng-app` later in this chapter.

Directives are everywhere! They are the most powerful feature of Angular. Using them in HTML is intuitive but creating directives requires a decent understanding of the framework. We have dedicated a complete chapter to directives where we will learn how to create our own directives and use them in views.

Similar to the expression usage in interpolation, directives too can take an expression (although it’s not mandatory). All the directives' attributes we have used in the app have an expression assigned to them and like interpolation these expressions are also evaluated in the context of the linked scope. Expressions are like input parameters to a directive.
In Angular, some directives accept interpolated expressions instead of standard expressions, for example, the ng-src directive that allows setting the src property of the image tag dynamically:

```
ng-src="http://www.google.com/images/{{imageName}}"
```

But more often than not we need to provide a non-interpolated expression to the directive. Using an expression with `{{}}` and without it is not the same and hence we should be always be careful while assigning an expression to the directive attribute.

Using interpolation expression for a directive that accepts standard expressions is a common mistake that AngularJS newbies make. `ng-click='{{verifyGuess()}}'` is invalid, `ng-click='verifyGuess()'` is correct. When in doubt, always refer to the documentation for the directive to know what is acceptable.

The usage of expressions in interpolation symbols and directives is an important concept to understand and this is what we are going to discuss in the next section on expressions.

### Expressions

Expressions in AngularJS are nothing but plain JavaScript code that are evaluated in the context of the current scope object with few limitations. The excerpt from the AngularJS documentation (https://docs.angularjs.org/guide/expression) highlights the following differences and limitations:

- **Context**: JavaScript expressions are evaluated against the global window. In Angular, expressions are evaluated against a scope object.
- **Forgiving**: In JavaScript, trying to evaluate undefined properties generates ReferenceError or TypeError. In Angular, expression evaluation is forgiving to undefined and null.
- **No Control Flow Statements**: You cannot use the following in an Angular expression: conditionals, loops, or exceptions.
- **Filters**: You can use filters within expressions to format data before displaying it.
These limitations still do not stop us from doing some nifty stuff with expressions. As we can see in the following examples, these all are valid expressions:

```javascript
// outputs the value of property
{{property}}

// outputs the result of boolean comparison. Ternary operator
{{property1 >=0?'positive': 'negative'}}

// call testMethod and outputs the return value
{{testMethod()}}

// assign value returned by testMethod to x. Creates "x" on scope if not available. Empty output
{x=testMethod()}

// calls testMethod() and testMethod2() and assign return values to x and y. Empty output
{x=testMethod();y=testMethod2()}
```

Having looked into expressions, I strongly advise you to keep your expressions simple, thus keeping the HTML readable. The `ng-show="formHasErrors()"` expression is always better than `ng-show="uname==null || email==null || emailformatInValid(email) || age < 18"`. So, when an expression starts to become complex, move it into a controller function.

That concludes our discussion on expressions. We now have a working app and a fair understanding of the model, view, and controller part of the framework.

Let's dig a bit deeper into the framework and understand how some parts of the app function.

To start with, it is evident now that the `ng-controller` directive together with interpolation and other directives such as `ng-model`, `ng-show`, and `ng-click` allow us to establish a connection between view and model properties/action methods defined on the controller.

To understand how this connection works, we need to introduce an important new concept: AngularJS data bindings.
AngularJS bindings

To keep the view and the model in sync, Angular creates bindings between view elements and model properties. Bindings are created for the interpolations and directives that we define on the view.

Interpolation bindings are easy to understand. An expression inside the interpolation symbol is evaluated and the return value replaces the interpolation. Whenever the expression value changes, the view is automatically updated.

A directive on the other hand can bind any HTML element attribute with the model depending upon the directive implementation.

For example, the `ng-show` directive that we have used is as follows:

```html
<p ng-show="deviation<0" class="alert alert-warning">Your guess is higher.</p>
```

This binds the paragraph's `<p>` class property with the model value `deviation`. When the expression `deviation<0` is false, a CSS class `.ng-hide` (with style defined as `display:none`) is appended; when true, the class is removed. This addition and removal of the class results in the previous paragraph element being shown and hidden.

Similarly, the `ng-model` directive binds the `value` attribute of the `input` with the model property it references.

Angular supports MathML and SVG markup and can perform interpolation on them.

Angular binding setup starts once the browser has rendered the raw HTML file or what we call the view template. The framework then compiles this view template and in the process sets up the necessary binding. It then does the necessary synchronization between the model and the view template that produces the final rendered output. The following screenshot depicts the transformations that happen to the view template after data binding is done for our app:
We can ourselves check the view template of the app by removing the `ng-app` attribute from the `body` tag and refreshing the app in the browser.

Where Angular differs from other template frameworks is that these bindings between a model and view are live. Angular does not merely merge the model and view templates to create the final HTML. Changes done to the model update the view and any update in the view done by the user is reflected back in the model. AngularJS never regenerates the HTML again; it just works on the relevant part of HTML and updates it on model changes. This data binding capability of the framework together with directives and interpolations makes Angular an exceptional view templating engine too.

An interesting characteristic about these bindings is that they can be one-way or two-way. Depending upon the directive used, the nature of the binding is decided.

In one-way binding, changes to the model are synchronized with the view. Following are examples of one-way binding:

```html
<p>
  <p ng-show="deviation<0" class="alert alert-warning">Your guess is higher.</p>
  <p ng-show="deviation>0" class="alert alert-warning">Your guess is lower.</p>
  <p ng-show="deviation===0" class="alert alert-success">Yes! That's it.</p>
</p>
<p class="text-info">No of guesses : <span class="badge">{{noOfTries}}</span></p>
```
Getting Started

Any change to the noOfTries and deviation properties affects these bindings and consequently the user interface content.

In two-way bindings, not only are changes done to a model reflected in the view, but the reverse also holds true. In our app, the input box defines a two-way binding between the input value and guess using the ng-model directive:

```html
<label>Your Guess: </label><input type="number" ng-model="guess"/>
```

Changing the value in the input box updates guess and, if we update the guess property in the controller, the view automatically gets updated, as happens in one-way bindings.

The bottom-line is, when adding directives and interpolation to a view, we are essentially instructing Angular to create data bindings that keep the model and view in sync.

The overall bindings in our app look like this:

![Diagram showing bindings between HTML View and Scope Properties]

It's time now to revisit the controller implementation and understand the implementation in the light of our newly acquired understanding of Angular bindings.
Revisiting the controller

Now that the app is running, why don't we get a firsthand experience of how the view and controller interact. Put a breakpoint on the JavaScript code inside verifyGuess. Enter a guess and click on Verify. The breakpoint will be hit.

We can see that the value that we added in the input is available on the $scope property guess (see the Watch Expressions section in the previous screenshot). Inside the function, we use the guess property and the original property to calculate deviation, and then update the noOfTries property. As we return from the function, the view updates immediately based on model changes.

Let's try to analyze a few important aspects of an AngularJS controller:

- To start with, if you are as observant as I am, you will have noticed that a scope object ($scope) gets passed as a parameter to the controller function (function GuessTheNumberController($scope)), almost magically! We are nowhere calling the controller function and passing any parameter. And you may have guessed it already; if we are not doing it, then the framework must be responsible. How and when is something that we will discuss when we go into scopes in greater detail.
• Also, if you are more observant than me, you will realize that the controller does not reference the view! There are no DOM element references, no DOM selection, or reading or writing. The operations that the controller performs are always on the model data. As a consequence, the controller code is more readable, hence more maintainable (and, of course, testable).

When it comes to writing a controller, the golden rule is never ever, ever, ever reference the DOM element, either using plain JavaScript or with libraries such as jQuery, in the controller. Such code does not belong to the controller.

If you want to do DOM manipulations, use existing directives or create your own.

• In AngularJS, unlike other frameworks, a model does not need to follow any specific structure. We do not have to derive it from any framework class. In fact, any JavaScript object and primitive types can act as a model. The only requirement is that the properties should be declared on a special object exposed by the framework called $scope. This $scope object acts as glue between the model and view and keeps the model and view in sync through Angular bindings (an over-simplification). This $scope object takes up some responsibilities that in traditional MVC frameworks belonged to the controller.

When we talk about a "scope object" or "scope", we're referring to the $scope object. As we will see in the next section, almost every scope object has a constrained existence and hence we simply refer to it as scope.

So far, we have been constantly discussing a scope object or $scope. This is an important concept to understand as it can save us from countless hours of debugging and frustration. The next section is dedicated to learning about scopes and using them.

Scope

Scope, as we described earlier, is a JavaScript object that binds the model properties and behavior (functions) to the HTML view.

An important thing to realize here is that the scope object is not the model but it references our model.
This is a special object for the framework as the complete view behavior is tied to this object. The framework creates these scope objects based on some AngularJS constructs. The framework can create multiple scope objects, depending on how the views are structured. The thing that we should keep in mind is that scope objects are always created in the context of a view element and hence follow a hierarchical arrangement similar to that of HTML elements (with some exceptions).

It will be interesting to see what scopes we have created for our app. To dig into the available scopes, we will use an excellent chrome extension Batarang.

To use Batarang, perform the following steps:

1. Download and install this extension from the Chrome web store.
2. Navigate to the app page.
3. Open the Chrome developer console (F12).
4. Click on the AngularJS tab
5. In this tab, enable the the Enable checkbox and we are all set!

If we now open the Models tab, we will see two scopes: the 002 and 003 IDs (the IDs may differ in your case) organized in a hierarchical manner, as seen in the following screenshot:

The parent scope (002) does not have any model properties but the child scope (003) contains all the properties and functions that we have been using as part of our app.
Whenever we talk about the parent-child scope or scope hierarchy, it actually is an inheritance hierarchy. This implies that the child scope object inherits from the parent scope object (standard JavaScript prototypal inheritance).

It seems that we have been using this very scope (003) object to bind the view to the model. We can check this by clicking on the small < arrow that precedes each scope name. This should take us to this line in the HTML:

```
<div class="container" ng-controller="GuessTheNumberController">

Well, it looks like the scope (003) is tied to the previous div element and hence is available within the start and end tags of the previous div. Any AngularJS view construct that has been declared inside the previous div element can access the scope (003) properties; in our case, this is the complete application HTML.

We can confirm this by copying the line containing the {{noOfTries}} interpolation string and pasting it outside the previous div element. The code should look something like this:

```
<div class="container" ng-controller="GuessTheNumberController">
  <!--Truncated Code-->
  <p class="text-info">No of guesses :
    <span class = "badge"> {{noOfTries}}</span></p>
  </p>
</div>
```

If we now refresh the app, we will see two lines that should be tracking the number of tries. Try to guess the number the first tracker will increment but there is no effect on the second one. Please refer to the following screenshot:

Interesting! Isn’t it? And it implies that all scope objects are constrained/scoped.

Angular does not create these scopes at random. There is a reason why the Guess the Number! app has two scopes. The next section on app bootstrapping covers the underlying cause of multiple scope creation.
We have not talked about the parent scope (002) till now. If we try to click the <link next to 002 scope, it navigates to the <body> tag with the ng-app directive. This in fact is the root scope and was created as part of application bootstrapping. This scope is the parent of all scope objects that are created during the lifetime of applications and is hence called $rootScope. We will throw some more light on this scope object in the next section where we talk about the app initialization process.

Let's summarize the key takeaways from our discussion on scope:

- Scope objects bind the view and model together.
- Scope objects are almost always linked to a view element.
- There can be more than one scope object defined for an application. In fact, in any decent size app there are a number of scopes active during any given time.
- More often than not, child scopes inherit from their parent scope and can access parent scope data.

This was a gentle introduction to scopes and I believe now we should have some basic understanding of scopes. There is more to scopes that we will be covering in the coming chapters. Nonetheless, this should be enough to get us started.

The last topic that we will cover in this chapter is the app initialization process. This will help us understand how and when these scope objects are created and linked to the view HTML.

**App bootstrapping**

One of the important topics that we have not touched on yet is how AngularJS initializes the app. To answer that, we will have to go back to the HTML view and look at the <body> tag. The <body> tag is as follows:

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

The <body> tag defines an ng-app directive attribute. Wherever Angular encounters this directive, it starts the initialization process. Since we have added ng-app to the <body> tag, everything within the <body> tag becomes part of our AngularJS app.

This construct makes it possible for an Angular app to coexist with other frameworks, as we can demarcate what part of HTML should use AngularJS.
During this bootstrapping/initialization process Angular does the following:

- It loads the module for the application. Modules are a way for AngularJS to organize and segregate code. As described earlier, these are containers for various Angular artifacts. In fact, even the framework's internal functionality is exposed through such modules.

- It sets up dependency injection (DI). DI is not very popular in the JavaScript world but is commonplace in other programming languages. It is basically a mechanism to inject dependencies into components that require it instead of the component creating it itself. For example, in the GuessTheNumberController function, we inject the dependency for $scope.

  function GuessTheNumberController($scope) {
  
- It creates a $rootScope object, which is a scope object available at the global level and not tied to any specific HTML fragment.

  $rootScope and $scope are instances of the same class (a constructor function). The difference is just the context in which they are available. $rootScope is available throughout the HTML (within ng-app) whereas $scope is always scoped by a specific HTML element.

- It compiles the DOM starting from where ng-app is declared. In this compilation process, the framework traverses the DOM, looks for all directives and interpolations, and sets up the binding between the view and model.

- Post compilation, it links the view and scope to produce a live view where changes are synced across the model and viewed in real time as we interact with the app.

This compilation and linking process can also lead to the creation of new scopes, all of which depend on the directive that is being applied to the HTML node. If you have Batarang opened, go back to the lone child scope (003) and click on the < link. We will again land up here:

  <div class="container" ng-controller="GuessTheNumberController">

Now look at the AngularJS documentation for ng-controller (https://docs.angularjs.org/api/ng/directive/ngController). There is this quote:

Directive Info

This directive creates new scope.
So, this ng-controller declaration when compiled by the framework leads to the creation of a new scope whose visibility is limited within the earlier mentioned <div> tag.

There are a number of other directives in Angular that cause the creation of a new scope; this is precisely the reason why there can be multiple scopes active during the execution of the app.

So now we know how and when these scope objects are created. We also now understand the Angular app initialization process a little better.

The last two sections cover the tools and resources that will come in handy for us while we make some killer apps on AngularJS.

**Tools**

Tools make our lives easy and we are going to share some tools that will help you with different aspects of AngularJS development, from code writing to debugging:

- **Batarang**: We have mentioned and used Batarang earlier while working with scopes. Batarang is an excellent debugging tool for an AngularJS application and we have explored only a few features of this awesome Chrome extension. We can install it from the Chrome web store (https://chrome.google.com/webstore/category/apps).

- **Browser Developer Console**: All current browsers have excellent capabilities when it comes to JavaScript debugging. Since we are working with JavaScript, we can put breakpoints, add a watch, and do everything that is otherwise possible with JavaScript. Remember, a lot of errors with code can be detected just by looking at the browser's console window.

- **jsFiddle and Plunker**: jsFiddle (http://jsfiddle.net/) and Plunker (http://plnkr.co/) are excellent playgrounds for trying out HTML, CSS, and JavaScript code. These tools also have great versioning and sharing capabilities that can come in handy if we want someone's help.

- **IDE extensions**: Many of the popular IDEs on the market have plugins/extensions to make AngularJS development easy for us. Examples include: Sublime Text package (https://github.com/angular-ui/AngularJS-sublime-package), JetBrains WebStorm 8 (http://blog.jetbrains.com/webstorm/2014/03/welcome-webstorm-8/), and TextMate bundle (https://github.com/angular-ui/AngularJS.tmbundle). Search for support for your specific IDE on the Internet to see if there is some helpful stuff available.
Resources

AngularJS is a popular framework and there is a vibrant community to support us in all our endeavors. Together with this book there are also blogs, articles, support forums, and plenty of help. Some of the prominent resources that will be useful are:

- **Framework code and documentation**: Angular documentation (https://angularjs.org/) has been criticized a lot in the past but it has got a whole lot better. And there is always the Angular source code, a great source of learning.

- **The AngularJS Google group** (angular@googlegroups.com) and the StackOverflow channel (http://stackoverflow.com/questions/tagged/angularjs): Head over here if you have any questions or are struck with some issue.

- **AngularJS Google+ Channel** (https://plus.google.com/+AngularJS): This is the source of the latest happenings on the framework.

- **Build with Angular** (https://builtwith.angularjs.org/): People have been creating some amazing apps using AngularJS. This site showcases such apps and most of them have source code available for us to have a look at.

That's it! The chapter is complete and it's time to summarize what we've learned.

Summary

The journey has started and we have reached the first milestone. Despite this chapter being entitled *Getting Started*, we have covered a lot of concepts that will be necessary for us to know so as to understand the bigger picture. Our learning was derived from our *Guess the Number!* app that we built and dissected throughout the chapter.

We learned about the MVC triad starting from the model and designed the model for our *Guess the Number!* app. We learned how to expose the model over the scope object and how the model drives the view in AngularJS.

We explored the view part of MVC and designed our view. We learned more about AngularJS binding and understood the live nature of these bindings. We explored the new AngularJS view constructs: interpolations and directives, and understood the role expressions play in the framework.
The last MVC element was the controller. We learned about the AngularJS controller and how it works in close sync with the view to provide the necessary behavior. One important consideration that came out of this discussion was that the controller does not directly refer the view or manipulate it.

Once we had a good understanding of AngularJS and the MVC component, we focused our efforts on learning about scopes. We learned how the scope is the glue between the view and the model. We saw scope objects defined inside our game and how changes in the model and view are synced by the framework.

To round things off, we learned about a very important process in AngularJS: app bootstrapping. By exploring the bootstrap process, we were able to connect the dots and answer a number of questions that related to scopes, binding, and app initialization itself.

The groundwork has been done and now we are ready for some serious app development on the AngularJS framework. In the next chapter, we will start working on a more complex exercise and expose ourselves to a number of new AngularJS constructs.
Where to buy this book
You can buy AngularJS by Example from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.