

1

Building a Simple AngularJS Application

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Creating a new AngularJS application from scratch
- Creating custom controllers, directives, and services
- Communicating with an external API server
- Storing data client-side using HTML5 LocalStorage
- Creating a simple animation with ngAnimate
- Packaging your application for distribution and deployment using GitHub Pages

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com code downloads for this chapter at <http://www.wrox.com/go/proangularjs> on the Download Code tab. For added clarity, the code downloads contain an individual directory for each step of the application building guide. The `README.md` file located in the root directory of the companion code contains additional information for properly utilizing the code for each step of the guide. Those who prefer to use GitHub can find the repository for this application, which includes Git tags for each step of the guide and detailed documentation, by visiting <http://github.com/diegonetto/stock-dog>.

WHAT YOU ARE BUILDING

The best way to learn AngularJS is to jump directly into a real-world, hands-on application that leverages nearly all key components of the framework. Over the course of this chapter, you will build StockDog, a real-time stock watchlist monitoring and management application. For the

unfamiliar, a watchlist in this context is simply an arbitrary grouping of desired stocks that are to be tracked for analytical purposes. The Yahoo Finance API (application programming interface) will be utilized to fetch real-time stock quote information from within the client. The application will not include a dynamic back end, so all information will be fetched from the Yahoo Finance API directly or, in the case of company ticker symbols, be contained within a static JSON (JavaScript Object Notation) file. By the end of this chapter, users of your application will be able to do the following:

- Create custom-named watchlists with descriptions
- Add stocks from the NYSE, NASDAQ, and AMEX exchanges
- Monitor stock price changes in real time
- Visualize portfolio performance of watchlists using charts

StockDog will consist of two main views that can be accessed via the application’s navigation bar. The dashboard view will serve as the landing page for StockDog, allowing users to create new watchlists and monitor portfolio performance in real time. The four key performance metrics displayed in this view will be Total Market Value, Total Day Change, Market Value by Watchlist (pie chart), and Day Change by Watchlist (bar graph). A sample dashboard view containing three watchlists is shown in Figure 1-1.

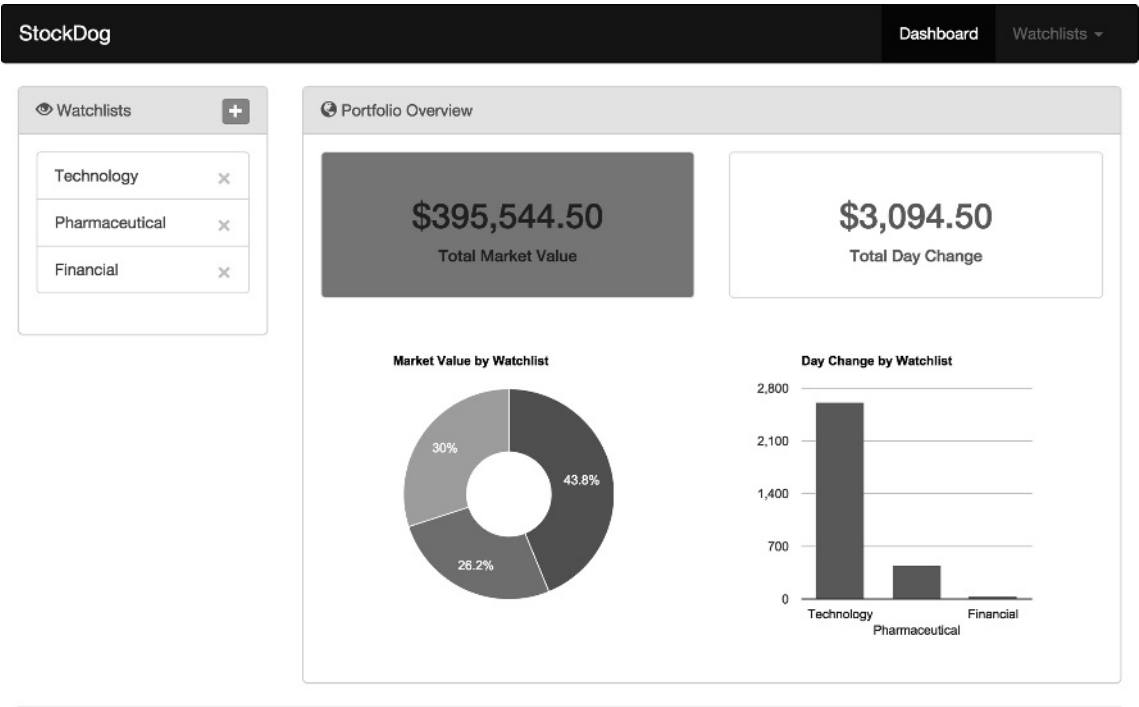
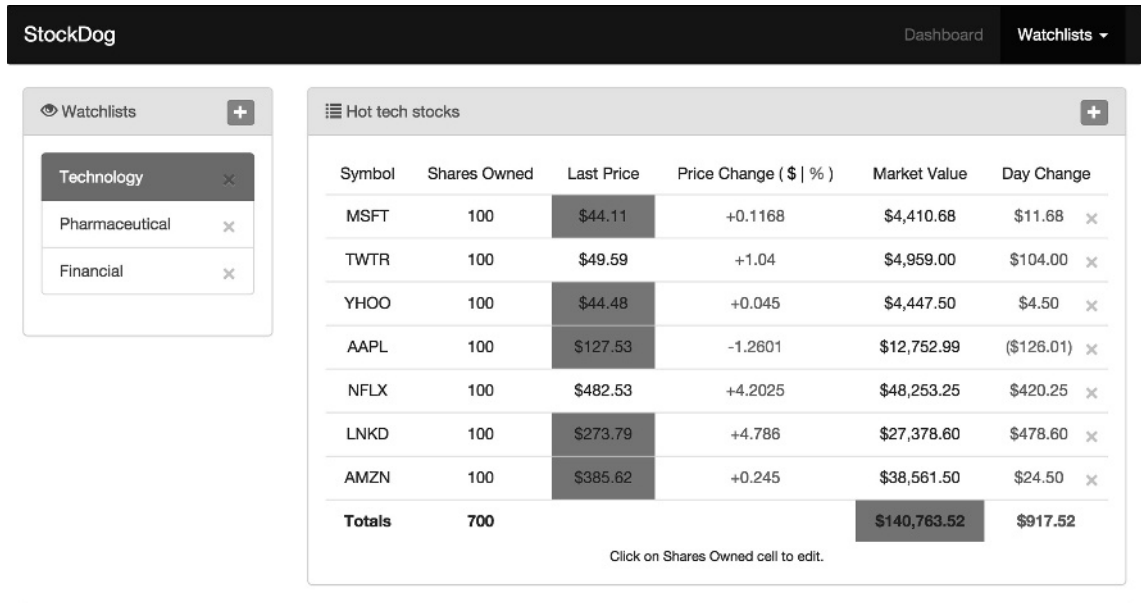


FIGURE 1-1

Each watchlist created in StockDog has its own watchlist view containing an interactive table of stock price information as well as a few basic calculations that assist in monitoring an equity position. Here, users of your application can add new stocks to the selected watchlist, monitor stock price changes in real time (during market hours), and perform in-line editing of the number of shares owned. A sample watchlist view tracking seven stocks is shown in Figure 1-2.



Built with ❤ by Diego and Val

FIGURE 1-2

The process of building this application will be described over a series of 12 steps. Each step will focus on developing a key feature of StockDog, with AngularJS components introduced along the way, because they are needed to fulfill requirements defined by the application. Before beginning the construction of StockDog, it is important to establish a high-level overview of what you will be learning.

WHAT YOU WILL LEARN

The step-by-step guide included in this chapter will go beyond basic AngularJS usage. By implementing practical, real-world examples using the main building blocks of this framework, you will be exposed to most of the components provided by AngularJS, which will then be expanded upon in detail in subsequent chapters. It is important to keep this in mind because some of the features required by StockDog will utilize advanced concepts of the framework. In these cases, specific details on how the underlying AngularJS mechanism works will be omitted, but a high-level explanation will always be provided so that you can understand how the component is being

utilized in the context of implementing the feature at hand. By the end of this chapter, you will have learned how to do the following:

- Structure a multiview single-page application
- Create directives, controllers, and services
- Configure `$routeProvider` to handle routing between views
- Install additional front-end modules
- Handle dynamic form validation
- Facilitate communication between AngularJS components
- Utilize HTML5 `LocalStorage` from within a service
- Communicate with external servers using `$http`
- Leverage the `$animate` service for cascading style sheet (CSS) animations
- Build application assets for production
- Deploy your built application to GitHub Pages

Now that the scope and high-level overview for StockDog have been discussed, you should have enough background and context to begin building the application. For those interested in viewing a working demonstration of StockDog immediately, you can find the completed application at <http://stockdog.io>.

STEP 1: SCAFFOLDING YOUR PROJECT WITH YEOMAN

Starting a brand new web application from scratch can be a hassle because it usually involves manually downloading and configuring several libraries and frameworks, creating an intelligent directory structure, and wiring your initial application architecture by hand. However, with major advancements in front-end tooling utilities, this no longer needs to be such a tedious process. Throughout this guide, you will utilize several tools to automate various aspects of your development workflow, but detailed explanations of how these tools work will be saved for discussion in Chapter 2, “Intelligent Workflow and Build Tools.” Before getting started with scaffolding your project, you need to verify that you have the following prerequisites installed as part of your development environment:

- **Node.js**—<http://nodejs.org/>
- **Git**—<http://git-scm.com/downloads>

All the tools used in this chapter were built using Node.js and can be installed from the Node Packaged Modules (NPM) registry using the command-line tool `npm` that is included as part of your Node.js installation. Git is required for one of these tools, so please ensure that you have properly configured both it and Node.js on your system before continuing.

Installing Yeoman

Yeoman is an open source tool with an ecosystem of plug-ins called generators that can be used to scaffold new projects with best practices. It is composed of a robust and opinionated client-side

stack that promotes efficient workflows which, coupled with two additional utilities, can help you stay productive and effective as a developer. Following are the tools Yeoman uses to accomplish this task:

- **Grunt**—A JavaScript task runner that helps automate repetitive tasks for building and testing your application
- **Bower**—A dependency management utility so you no longer have to manually download and manage your front-end scripts

You can find an in-depth discussion of Yeoman, its recommended workflow, and associated tooling in Chapter 2, “Intelligent Workflow and Build Tools.” For now, all you need to do to get started is to install Grunt, Bower, and the AngularJS generator by running the following from your command line:

```
npm install -g grunt-cli
npm install -g bower
npm install -g generator-angular@0.9.8
```

NOTE *Specifying the `-g` flag when invoking `npm install` ensures that the desired package will be available globally on your machine. When you're installing `generator-angular`, the official AngularJS generator maintained by the Yeoman team, version 0.9.8 is specified. This should allow you to easily follow along with the rest of the guide, regardless of the current version. For any subsequent projects, it's highly recommended that you update to the latest version. You can do this by simply running `npm install -g generator-angular` once you have completed this chapter.*

Scaffolding Your Project

With all the prerequisite tools installed on your machine, you are ready to get started scaffolding your project. Thankfully, Yeoman makes this process quick and painless. Go ahead and create a new directory named `StockDog`, and then navigate into it using your command-line application of choice. From within your newly created project directory, run the following from the command line:

```
yo angular StockDog
```

This fires up the AngularJS Yeoman generator, which asks you a few questions regarding how you want to set up your application. The first prompt asks if you want to use Sass with Compass. Although these are both incredibly useful tools for managing your style sheets, their usage is outside the scope of this chapter, so please answer no by typing `n` and then pressing Enter:

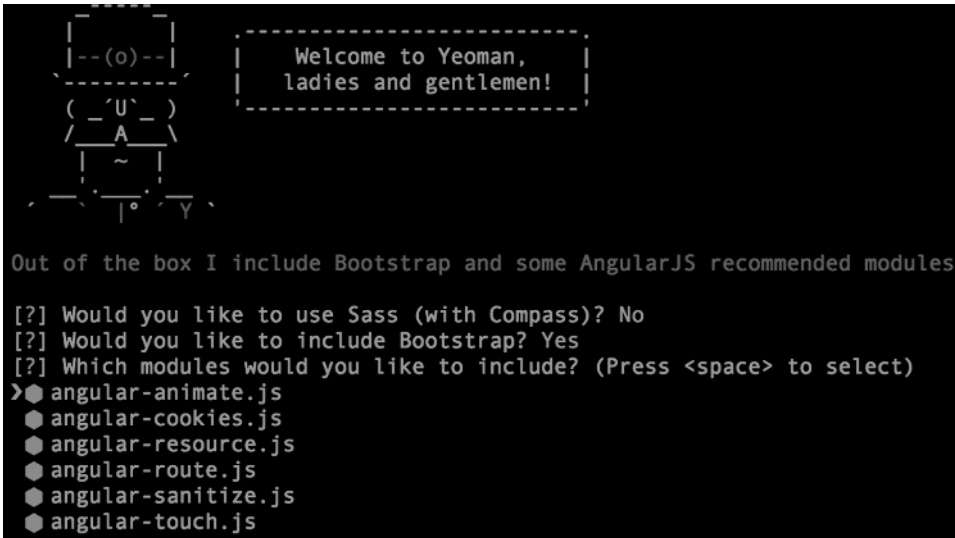
```
[?] Would you like to use Sass (with Compass)? (Y/n)
```

The next prompt asks if you want to include Bootstrap, the front-end framework created by Twitter. `StockDog` makes heavy use of the Hypertext Markup Language (HTML) and CSS assets that

Bootstrap provides, so you need to include this as part of your application. Because the default response to this prompt is yes, as expressed by the capitalized `y`, simply pressing the Enter key allows you to continue with Bootstrap included:

```
[?] Would you like to include Bootstrap? (Y/n)
```

The final prompt asks which optional AngularJS modules you want to include in your application. Although you won't necessarily be utilizing all the ones listed in Figure 1-3 for this specific project, it's recommended that you go ahead and include them anyway. You can learn more by visiting <https://docs.angularjs.org/api> and scrolling down to see what services and directives are made available for each respective module. Simply press the Enter key to continue with all the default modules and have Yeoman begin scaffolding your project, as shown in Figure 1-3.



```

  ____
  |  (o)  |
  |_____|
  |  'U'  |
  |  _A_  |
  |  ~    |
  |_____|
  |  |    |
  |_____|

Welcome to Yeoman,
ladies and gentlemen!

Out of the box I include Bootstrap and some AngularJS recommended modules.

[?] Would you like to use Sass (with Compass)? No
[?] Would you like to include Bootstrap? Yes
[?] Which modules would you like to include? (Press <space> to select)
> ☒ angular-animate.js
  ☐ angular-cookies.js
  ☐ angular-resource.js
  ☐ angular-route.js
  ☐ angular-sanitize.js
  ☐ angular-touch.js

```

FIGURE 1-3

After pressing Enter on the final prompt and waiting for Yeoman to finish running all the relevant scaffolding tasks, which will take a few brief moments, the foundation for StockDog will be ready for exploration. In the following section, you will take a closer look at the important parts of the directory structure and workflow tasks that Yeoman configured as part of the scaffolding process.

Exploring the Application

Now that your project setup is complete, take a few minutes to explore what the AngularJS Yeoman generator has provided for you. Your project's directory should be structured as follows:

```

StockDog/
├── .bowerrc
├── .editorconfig
├── .gitattributes

```

```
|— .jshintrc
|— .travis.yml
|— bower.json
|— package.json
|— Gruntfile.js
|— app/
|   |— 404.html
|   |— favicon.ico
|   |— robots.txt
|   |— index.html
|   |— images/
|   |— styles/
|   |   |— main.css
|   |— views/
|   |   |— main.html
|   |   |— about.html
|   |— scripts/
|   |   |— app.js
|   |   |— controllers/
|   |       |— main.js
|   |       |— about.js
|— node_modules/
|— bower_components/
|— test/
```

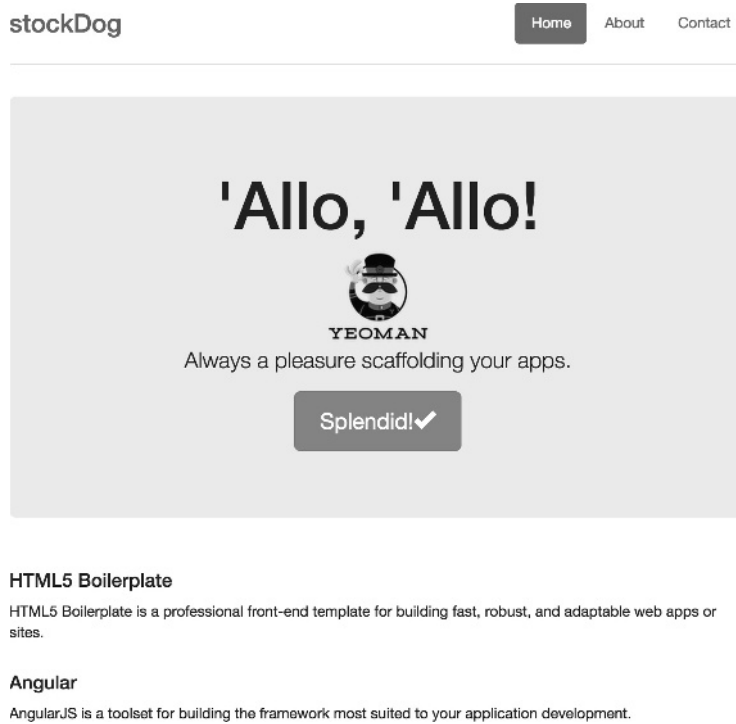
Upon first glance, this directory structure may seem overwhelming, but many of the generated files created by Yeoman are meant to help enforce best practices and can be completely ignored for the remainder of this chapter. The files and directories that you will be focusing on have been bolded for emphasis, so for now you only need to pay attention to those.

NOTE Depending on how you are viewing your project's directory, your operating system may automatically hide the files prefixed by a dot. These files are meant for configuring various tools such as Git, Bower, and JSHint.

As you have probably guessed, the bulk of your application is contained inside the `app/` directory. Here you can find the main `index.html` file, which serves as the entry point for your entire application, as well as the `styles/`, `views/`, and `scripts/` directories, which contain CSS, HTML, and JavaScript files, respectively. `Gruntfile.js` is also of particular interest because it configures several Grunt tasks that support your workflow during the development of StockDog. Go ahead and fire up your terminal application of choice and run the following from the command line:

```
grunt serve
```

This launches the local development server configured by Yeoman during the scaffolding process and opens the current skeleton application within a new tab inside your default browser. At this point, your browser should be pointed at `http://localhost:9000/#/` and displaying an application page that looks identical to Figure 1-4.

**FIGURE 1-4**

Congratulations! You have successfully finished scaffolding your project and are almost ready to begin building the first component of the StockDog application. Throughout your development process, be sure to keep the terminal session where you ran the `grunt serve` command open because it is responsible for serving all your application assets for use in your browser. Before moving onto the next section, take a minute to modify the `app/views/main.html` file by removing all its contents. Upon saving your modification, you should notice that your browser tab is refreshed with your changes instantly, which in this case should consist of a mostly empty view. Yeoman set up this automation magic when it configured your `Gruntfile.js` with tasks that watch for modifications in your application's files and refresh your browser accordingly. This functionality alone will prove to be quite helpful as you begin building components of the StockDog application.

Cleaning Up

So far in this chapter, you have seen how to use Yeoman to scaffold a new project from scratch, explored the generated project structure, and gotten a glimpse of how the provided workflow can help you stay productive during development. The last thing you need to do before moving onto the next step of this guide is to delete a few generated files that StockDog won't need and clean up any associated references. Please locate and delete the following files from your project:

```
app/views/main.html
app/views/about.html
```



```
app/scripts/controllers/main.js
app/scripts/controllers/about.js
```

Next, remove the routes Yeoman created by opening the `app/scripts/app.js` file and deleting the two `.when()` configurations of the `$routeProvider`. You can accomplish this by removing the following lines of code:

```
.when('/', {
  templateUrl: 'views/main.html',
  controller: 'MainCtrl'
})
.when('/about', {
  templateUrl: 'views/about.html',
  controller: 'AboutCtrl'
})
```

Finally, remove the references to the previously deleted `main.js` and `about.js` controller scripts by deleting the following lines from within the `app/index.html` file:

```
<script src="scripts/controllers/main.js"></script>
<script src="scripts/controllers/about.js"></script>
```

With these modifications of the generated skeleton application complete, you are now ready to begin building the Watchlist component of StockDog. To access the completed code for this step of the guide in its entirety, please refer to the `step-1` directory inside the companion code for this chapter or check out the corresponding tag of the GitHub repository.

STEP 2: CREATING WATCHLISTS

In this section, you will be implementing stock watchlists, the first major component of the StockDog application. As previously mentioned, a watchlist is simply an arbitrary grouping of desired stocks that are to be tracked for analytical purposes. Users of your application will create new watchlists in StockDog by filling out a small form, presented inside a modal, which prompts them for a name and brief description to identify the watchlist. All watchlists registered with the application have their data saved client-side in the browser using HTML5 LocalStorage. Finally, watchlists will be presented by name within a small panel in the user interface. Armed with a high-level understanding of the component's desired functionality, you will now learn how to implement watchlists using AngularJS.

The Application Module

The main entry point for all AngularJS applications is the top-level app module. So what exactly is a module? As mentioned in the official documentation, you can think of a module as a container for the different parts of your application. Although most applications have a main method that instantiates and wires together various components, AngularJS modules declaratively specify how your components should be bootstrapped. Some advantages to this approach are that modules can be loaded asynchronously in any order, and code readability and reusability are enhanced. The main application module is defined by invoking the `.module()` function, which accepts a name

and array of dependencies, located inside the `app/scripts/app.js` file. Make note of the module name, which in this case should be `stockDogApp`, because you will be referencing it shortly. For those who have used RequireJS in the past, this method of declaring module dependencies should look familiar.

Installing Module Dependencies

Currently, the only modules your application depends on should be `ngAnimate`, `ngCookies`, `ngResource`, `ngRoute`, `ngSanitize`, and `ngTouch`, all of which Yeoman installed based on your response to the third prompt of the initial scaffolding process. Later in this section, you will be using the `$modal` service exposed by AngularStrap, a third-party module containing native AngularJS bindings for various components provided by the Bootstrap framework. You can learn more about AngularStrap by visiting its documentation site located at <http://mgcrea.github.io/angular-strap/>. Because the workflow set up by Yeoman uses Bower for managing front-end scripts, installing AngularStrap is as simple as running the following from your command line:

```
bower install angular-strap#v2.1.0 -save
```

This downloads the AngularStrap library and saves it as a dependency inside your `bower.json` file. If you have left your application server running, which was launched using `grunt serve`, Grunt will have seen the modification to `bower.json` and automatically updated your `index.html` file to reference the CSS and JavaScript files that AngularStrap provides. Not bad for a simple one-line command! Now all that is left is to register the AngularStrap module, which is named `mgcrea.ngStrap`, as a dependency for your `stockDogApp` module by adding it to the array of dependencies, as shown in Listing 1-1.

LISTING 1-1: `app/scripts/app.js`

```
angular
  .module('stockDogApp', [
    'ngAnimate',
    'ngCookies',
    'ngResource',
    'ngRoute',
    'ngSanitize',
    'ngTouch',
    'mgcrea.ngStrap'
  ]);
```

NOTE Another commonly used AngularJS companion library that exposes directives for various Bootstrap components is UI Bootstrap, a project that the AngularUI organization maintains. To learn more about UI Bootstrap, please visit the documentation site located at <http://angular-ui.github.io/bootstrap/>.

Bootstrapping the Application

Now that you have seen how to define an application module and register dependencies, the next and final step in bootstrapping StockDog is to reference the `stockDogApp` module from within your HTML. Conveniently enough, Yeoman has already done this for you. Take a look inside your `app/index.html` file; on line 19, you should see the following code:

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

The `ng-app` attribute that has been attached to the page's `<body>` tag is an AngularJS directive that flags the HTML element, which should be considered the root of your application. Directives will be defined shortly, but for now, the takeaway is that to bootstrap your AngularJS application module, you must add the `ng-app` attribute to your application's HTML. Also worth mentioning is that because `ng-app` is an element attribute, you have the freedom to move it around and decide whether the entire HTML page or only a portion of it should be treated as the Angular application. With the bootstrapping of your application using the `stockDogApp` module out of the way, you will now be exposed to AngularJS services, another crucial component of the framework.

The Watchlist Service

As defined in the AngularJS documentation, services are substitutable objects that are wired together using dependency injection. Services provide a great way to organize and share encapsulated code across your application. It is worth mentioning that AngularJS services are lazily instantiated singletons, meaning that they are only instantiated when an application component depends on it, with each dependent component receiving a single instance reference generated by the service factory. For the purpose of building out the watchlists functionality for StockDog, you will be creating a custom service that handles reading and writing the watchlists model to HTML5 LocalStorage. To get started, run the following from your command line:

```
yo angular:service Watchlist-Service
```

This uses the AngularJS Yeoman generator's packaged subgenerator for scaffolding out a skeleton service contained within the newly created `watchlist-service.js` file, which is located inside the `app/scripts/services` directory. In addition, Yeoman adds a reference to this newly created script, which can be seen by the addition of the following line of code at the bottom of your `app/index.html` file:

```
<script src="scripts/services/watchlist-service.js"></script>
```

Now that you have quickly wired up an entry point for your new service, you need to install `Lodash`, a utility library that offers functional programming helpers for JavaScript, which will be used throughout the remainder of this chapter. Use Bower to install `Lodash` by running the following from your command line:

```
bower install lodash --save
```

`Lodash` was initially a fork of the `Underscore.js` project but has since evolved to become a highly configurable and performant library loaded with a plethora of additional helpers. The `watchlistService` implementation, which uses a couple of `Lodash` methods, is shown in Listing 1-2.

LISTING 1-2: app/scripts/services/watchlist-service.js

```
'use strict';

angular.module('stockDogApp')
  .service('WatchlistService', function WatchlistService() {
    // [1] Helper: Load watchlists from localStorage
    var loadModel = function () {
      var model = {
        watchlists: localStorage['StockDog.watchlists'] ?
          JSON.parse(localStorage['StockDog.watchlists']) : [],
        nextId: localStorage['StockDog.nextId'] ?
          parseInt(localStorage['StockDog.nextId']) : 0
      };
      return model;
    };

    // [2] Helper: Save watchlists to localStorage
    var saveModel = function () {
      localStorage['StockDog.watchlists'] = JSON.stringify(Model.watchlists);
      localStorage['StockDog.nextId'] = Model.nextId;
    };

    // [3] Helper: Use lodash to find a watchlist with given ID
    var findById = function (listId) {
      return _.find(Model.watchlists, function (watchlist) {
        return watchlist.id === parseInt(listId);
      });
    };

    // [4] Return all watchlists or find by given ID
    this.query = function (listId) {
      if (listId) {
        return findById(listId);
      } else {
        return Model.watchlists;
      }
    };

    // [5] Save a new watchlist to watchlists model
    this.save = function (watchlist) {
      watchlist.id = Model.nextId++;
      Model.watchlists.push(watchlist);
      saveModel();
    };

    // [6] Remove given watchlist from watchlists model
    this.remove = function (watchlist) {
      _.remove(Model.watchlists, function (list) {
        return list.id === watchlist.id;
      });
      saveModel();
    };

    // [7] Initialize Model for this singleton service
```

```
var Model = loadModel();
});
```

The first thing you should notice is the invocation of the `.service()` method on the `stockDogApp` module, which registers this service with the top-level AngularJS application. This allows your service to be referenced elsewhere by injecting `WatchlistService` into the desired component implementation function. The `loadModel()` helper [1] requests the data stored in the browser's `LocalStorage` using keys that are namespaced under `StockDog` to avoid potential collisions. The `watchlists` value retrieved from `localStorage` is an array, whereas `nextId` is simply an integer used to uniquely identify each watchlist. The ternary operator guarantees that the initial value of both these variables is properly set and correctly parsed. The `saveModel()` helper [2] simply needs to stringify the `watchlists` array before persisting its contents to `localStorage`. Another internal helper function, `findById()` [3], uses `Lodash` to find a watchlist by a given ID inside the aforementioned array.

With these internal helpers out of the way, you should now notice that the remaining functions are attached directly to the service instance by using the keyword `this`. Although using `this` can be error prone and is not always the best approach, in this case it is quite alright because Angular instantiates a singleton by calling `new` on the function supplied to `.service()`. The service `.query()` function [4] returns all watchlists in the model unless a `listId` is specified. The `.save()` function [5] increments `nextId` and pushes a new watchlist onto the `watchlists` array before delegating to the `saveModel()` helper. Finally, `.remove()` uses a `Lodash` method to accomplish the exact opposite [6]. To complete this service, a local `Model` variable is initialized using the `loadModel()` helper. At this point, your `WatchlistService` is ready to be wired up from within an AngularJS directive, which you will be creating in the following section.

NOTE *If up until this point you have left your local development server running, Grunt should be reporting warnings that '_' is not defined. This is because Lodash attaches itself to the global scope via an underscore, but the process in charge of linting your JavaScript files (checking them for errors) is not aware of this fact. Adding "": false to the globals object located at the bottom of your .jshintrc file makes these warnings go away.*

The Watchlist-Panel Directive

By now, you might have already heard about AngularJS directives and how versatile they can be if used correctly. So what exactly is a directive? As defined in the official documentation, *directives* are markers on a Document Object Model (DOM) element (such as an attribute, element name, comment, or CSS class) that tell AngularJS's HTML compiler (`$compile`) to attach a specified behavior to, or even transform, the DOM element and its children. You will take a deeper look at how directives work in Chapter 5, "Directives." For now, all you need to know is that not only can you create your own custom directives, but AngularJS also comes with a set of built-in directives ready for use, like `ng-app`, `ng-view`, and `ng-repeat`, which are all prefixed by `ng`. For the `StockDog`

application, all your custom directives are prefixed by `stk` so they are easily identifiable. You can use Yeoman's directive subgenerator to scaffold and wire up a skeleton directive by running the following from your command line:

```
yo angular:directive stk-Watchlist-Panel
```

This creates the `stk-watchlist-panel.js` file inside the `app/scripts/directives` directory and automatically adds a reference to the newly created script inside your `index.html` file. The implementation of this directive is shown in Listing 1-3.

LISTING 1-3: `app/scripts/directives/stk-watchlist-panel.js`

```
'use strict';

angular.module('stockDogApp')
  // [1] Register directive and inject dependencies
  .directive('stkWatchlistPanel', function ($location, $modal, WatchlistService) {
    return {
      templateUrl: 'views/templates/watchlist-panel.html',
      restrict: 'E',
      scope: {},
      link: function ($scope) {
        // [2] Initialize variables
        $scope.watchlist = {};
        var addListModal = $modal({
          scope: $scope,
          template: 'views/templates/addlist-modal.html',
          show: false
        });

        // [3] Bind model from service to this scope
        $scope.watchlists = WatchlistService.query();

        // [4] Display addlist modal
        $scope.showModal = function () {
          addListModal.$promise.then(addListModal.show);
        };

        // [5] Create a new list from fields in modal
        $scope.createList = function () {
          WatchlistService.save($scope.watchlist);
          addListModal.hide();
          $scope.watchlist = {};
        };

        // [6] Delete desired list and redirect to home
        $scope.deleteList = function (list) {
          WatchlistService.remove(list);
          $location.path('/');
        };
      }
    };
  });
```

The `.directive()` method handles registering the `stkWatchlistPanel` directive with the `stockDogApp` module [1]. This example illustrates the use of Angular's dependency injection mechanism, which is as simple as specifying parameters to the directive's implementation function. Note that the previously created `WatchlistService` has been injected as a dependency, along with the `$location` and `$modal` services, because it will be needed to complete the directive's implementation. The implementation function itself returns an object containing configuration options and a `link()` function. Inside this function is where the directive's scope variables are initialized [2], which include creating a modal using AngularStrap's `$modal` service. The `.query()` method of the `WatchlistService` is invoked to bind the service's model to the directive's scope [3]. Handler functions are then attached to the `$scope` and provide functionality for showing the modal [4], creating a new watchlist from the modal's fields [5], and deleting a watchlist [6]. The implementations of these handler functions are straightforward and use the injected services.

The configuration options for the `stkWatchlistPanel` directive modify its behavior by restricting it for use as an element via `restrict: 'E'` and isolating its scope so that anything attached to the `$scope` variable is available only within the context of this directive. The `templateUrl` option can reference a file that Angular loads and renders into the DOM. For this application, you will be storing templates inside the `app/views/templates` directory, so go ahead and create that now. The `watchlist-panel.html` template needed by this directive is shown in Listing 1-4.

LISTING 1-4: app/views/templates/watchlist-panel.html

```
<div class="panel panel-info">
  <div class="panel-heading">
    <span class="glyphicon glyphicon-eye-open"></span>
    Watchlists
    <!-- [1] Invoke showModal() handler on click -->
    <button type="button"
      class="btn btn-success btn-xs pull-right"
      ng-click="showModal()">
      <span class="glyphicon glyphicon-plus"></span>
    </button>
  </div>
  <div class="panel-body">
    <!-- [2] Show help text if no watchlists exist -->
    <div ng-if="!watchlists.length" class="text-center">
      Use <span class="glyphicon glyphicon-plus"></span> to create a list
    </div>
    <div class="list-group">
      <!-- [3] Repeat over each list in watchlists and create link -->
      <a class="list-group-item"
        ng-repeat="list in watchlists track by $index">
        {{list.name}}
        <!-- [4] Delete this list by invoking deleteList() handler -->
        <button type="button" class="close"
          ng-click="deleteList(list)"> &times;
        </button>
      </a>
    </div>
  </div>
</div>
```

NOTE Upon saving this HTML file, you may have noticed that your browser did not automatically refresh with the changes. That is because the current Grunt workflow is only watching for changes to HTML files in the top-level `app/views` directory. To force Grunt to recursively watch for modifications of any HTML files inside of `app/views`, change the regular expression on line 59 of your `Gruntfile.js` to the following:

```
'<%= yeoman.app %>/**/*.html',
```

The `watchlist-panel.html` template makes heavy use of the classes and icons provided by the Bootstrap framework to create a simple, yet polished, interface. The built-in AngularJS `ng-click` directive is used to invoke the `showModal()` handler when the plus button is clicked [1]. The `ng-if` directive conditionally inserts or removes a DOM element based on the evaluation of an expression, which in this case displays instruction text when the `watchlists` array is empty [2]. To iterate over the `watchlists` array, `ng-repeat` is used with the `track by $index` syntax so that Angular doesn't complain if the array contains identical objects [3]. Worth mentioning is the fact that because `ng-repeat` is attached to an HTML `<a>` tag, a unique link is created for each object in the array. The double curly braces, `{{ }}`, used to reference the current list's name, are called a *binding*, while `list.name` itself is called an *expression*. The binding tells Angular that it should evaluate the expression and insert the result into the DOM in place of the binding. A binding results in efficient continuous updates whenever the result of the expression evaluation changes. Finally, the `deleteList()` handler is wired into the interface via another button, connected once again using the `ng-click` directive [4].

Basic Form Validation

The final step in completing the implementation of the `stkWatchlistPanel` directive is to build the form that allows users to create new watchlists. If you remember, inside the directive's `link()` function, the `addListModal` variable was initialized using the `$modal` service exposed by the AngularStrap module. The `$modal` service accepts a `template` option, which renders the desired HTML inside a Bootstrap modal. Create a new file inside the `app/views/templates/` directory named `addlist-modal.html`. The implementation of this template is shown in Listing 1-5.

LISTING 1-5: `app/views/templates/addlist-modal.html`

```
<div class="modal" tabindex="-1" role="dialog">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <!-- [1] Invoke $modal.$hide() on click -->
        <button type="button" class="close"
          ng-click="$hide()" > &times;
        </button>
        <h4 class="modal-title">Create New Watchlist</h4>
      </div>
```



```

<!-- [2] Name this form for validation purposes -->
<form role="form" id="add-list" name="listForm">
  <div class="modal-body">
    <div class="form-group">
      <label for="list-name">Name</label>
      <!-- [3] Bind input to watchlist.name -->
      <input type="text"
        class="form-control"
        id="list-name"
        placeholder="Name this watchlist"
        ng-model="watchlist.name"
        required>
    </div>
    <div class="form-group">
      <label for="list-description">Brief Description</label>
      <!-- [4] Bind input to watchlist.description -->
      <input type="text"
        class="form-control"
        id="list-description"
        maxlength="40"
        placeholder="Describe this watchlist"
        ng-model="watchlist.description"
        required>
    </div>
  </div>
  <div class="modal-footer">
    <!-- [5] Create list on click, but disable if form is invalid -->
    <button type="submit"
      class="btn btn-success"
      ng-click="createList()"
      ng-disabled="!listForm.$valid">Create</button>
    <button type="button"
      class="btn btn-danger"
      ng-click="$hide()">Cancel</button>
  </div>
</form>
</div>
</div>
</div>

```

The first thing you should notice with this template is that not only does it reference the handler functions attached to the `stkWatchlistPanel` directive's scope, it also leverages the `$hide()` method exposed by the `$modal` service [1]. Because inputs are required to gather the information necessary to create a new watchlist, an HTML `<form>` is used [2]. Pay particular attention to the `name="listForm"` attribute because this is how you reference the form to check its validity. The two `<input>` tags are augmented with the `ng-model` directive, which binds the respective input values to the `$scope.watchlist` variable ([3] & [4]) initialized in the directive's `link()` function. The HTML `required` attribute is also used for both inputs because you want to ensure the user specifies both a name and a description before creating a new watchlist. Finally, the directive's `createList()` handler is invoked when the Create button is clicked, but only when the form is valid. The built-in `ng-disabled` directive disables or enables the button based on the result of evaluating the `!listForm.$valid` expression.

Using the Directive

Now that you have completed creating the `stkWatchlistPanel` directive and its associated templates, you will see how easy it is to reference it inside your HTML. Open the `app/index.html` file and insert the following code before the `<div>` tag marked with the `footer` class:

```
<stk-watchlist-panel></stk-watchlist-panel>
```

At this point, you may be wondering why this directive was used as an HTML element tag instead of an attribute. If you remember, the `stkWatchlistPanel` directive was created with the `restrict` configuration property set to `E`, which meant that the directive was to be used as an HTML element. It may also initially seem strange that, although the directive was registered using `camelCase`, it was referenced using `spinal-case` inside the HTML. This is because HTML is case insensitive, so Angular normalizes your directive's name using this convention. With the preceding modification to your `index.html` file saved, Grunt automatically triggers a browser refresh; your application should look identical to the screenshot shown in Figure 1-5.

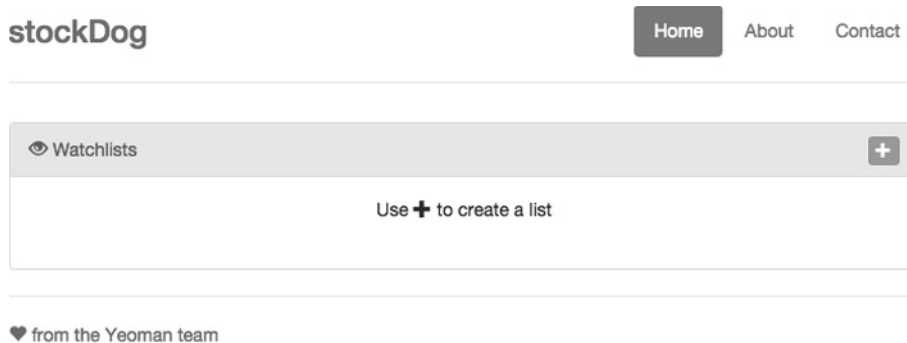


FIGURE 1-5

Clicking the green plus button inside the watchlist panel should launch the Bootstrap modal containing the watchlist creation form, as shown in Figure 1-6.

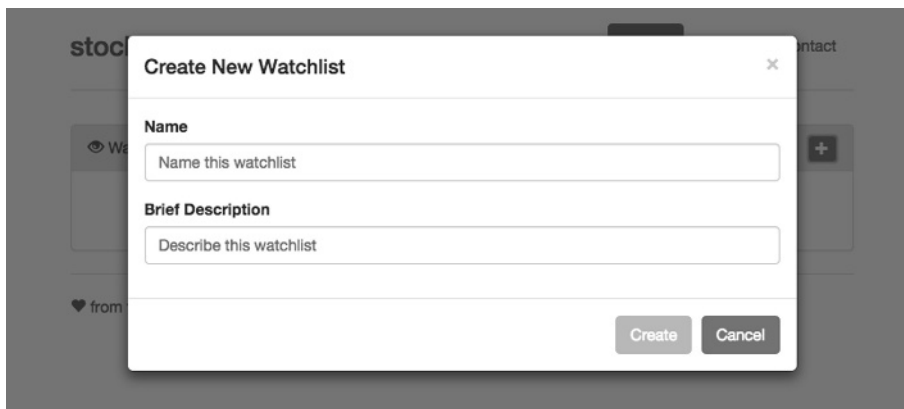


FIGURE 1-6

Congratulations! You have successfully finished implementing the watchlists feature of the StockDog application. In doing so, you have seen how to create an AngularJS service that uses HTML5 LocalStorage as well as a directive that manipulates the DOM and wires together several services. Take a minute to enjoy your handiwork thus far by creating a few watchlists, refreshing your browser to confirm that they were indeed persisted to LocalStorage, and then deleting them from the watchlist panel to ensure that everything is working as expected. If you've gotten stuck at any point during this step, take a moment to examine the completed code by referring to the `step-2` directory inside the companion code for this chapter or checking out the corresponding tag of the GitHub repository.

STEP 3: CONFIGURING CLIENT-SIDE ROUTING

Client-side routing is a critical component of any single-page application. Thankfully, AngularJS makes the task of mapping URLs to various front-end views extraordinarily simple. In its current state, StockDog does not contain additional HTML views other than the `index.html` file, which contains an embedded watchlist panel using the `stk-watchlist-panel` directive. In this section, you will see how the routing mechanism brings together AngularJS controllers and HTML templates to power the two main views of the StockDog application.

The Angular ngRoute Module

During the initial process of scaffolding the StockDog application, Yeoman asked if you wanted to install any supplemental AngularJS modules. One of these was `angular-route`, which exposes the `ngRoute` module that can be listed as a dependency for your application. You can verify that this module has been properly installed for StockDog by looking inside the `app/scripts/app.js` file and locating the reference to `ngRoute` inside the array of dependencies for the main `stockDogApp` module definition, as shown here:

```
angular
  .module('stockDogApp', [
    'ngAnimate',
    'ngCookies',
    'ngResource',
    'ngRoute',    // Include angular-route as dependency
    'ngSanitize',
    'ngTouch',
    'mgcrea.ngStrap'
  ])
```

NOTE Over the course of developing future AngularJS applications, you will undoubtedly be exposed to, and utilize, several AngularJS modules. The AngularJS team officially maintains some of these modules, like most of the ones seen in the code in the “The Angular ngRoute Module” section, with several others being created by the community. It is imperative that when you install a new module, usually via Bower, you also look at its documentation and properly include the corresponding module reference here as a dependency for your application.

The `ngRoute` module exposes the `$route` service and can be configured using the associated `$routeProvider`, which allows you to declare how your application's routes map to view templates and controllers. Providers are objects that create instances of services and expose configuration APIs that can be used to control the runtime behavior of a service. You will learn more about providers in a Chapter 7, “Services, Factories, and Providers,” but for now, the takeaway is that you can use the `$routeProvider` to define your application routes and implement deep linking, which allows you to utilize the browser's history navigation and bookmark locations within your application.

Adding New Routes

The process of adding a new route to your application consists of four distinct steps:

1. Define a new controller.
2. Create an HTML view template.
3. Call the `$routeProvider.when(path, route)` method.
4. Include a `<script>` tag reference inside `index.html` if the new controller resides within its own JavaScript file.

The fourth step is only required if your project's architecture mirrors that of the StockDog application, where each new AngularJS component resides within its own JavaScript file. Although these four steps are simple enough on their own, when working on large applications with many routes, views, and controllers, it can become a tedious process. Thankfully, the AngularJS Yeoman generator contains a subgenerator that can be used to entirely automate this four-step process. Go ahead and run the following commands from your terminal to scaffold out the AngularJS controllers, HTML templates, and `$routeProvider` configurations for the dashboard and watchlist views of the StockDog application:

```
yo angular:route dashboard
yo angular:route watchlist --uri=watchlist/:listId
```

With these two simple commands, you have instructed Yeoman to create the `dashboard.js` and `watchlist.js` files inside the `app/scripts/controllers/` directory. These files define the `DashboardCtrl` and `WatchlistCtrl`, respectively, as well as the `dashboard.html` and `watchlist.html` views inside the `app/views/` directory. Because Yeoman created two new JavaScript files for the desired route controllers, it also took the liberty of inserting the two required `<script>` tag references at the bottom of your `index.html` file. You may have noticed that the second command invoked the route subgenerator with a `--uri` flag. This instructs Yeoman to use an explicitly defined path when configuring the `$routeProvider`, which in this case was required because each watchlist created within StockDog will have its own unique view, generated from the `listId`, which will be passed as a route parameter. Looking inside `app/scripts/app.js`, you should see the following `$routeProvider.when()` configurations that Yeoman set up:

```
.when('/dashboard', {
  templateUrl: 'views/dashboard.html',
  controller: 'DashboardCtrl'
})
.when('/watchlist/:listId', {
```

```

    templateUrl: 'views/watchlist.html',
    controller: 'WatchlistCtrl'
  })

```

Before continuing onto the next section, take a moment to update the path used in the `$routeProvider.otherwise()` function located at the bottom of this file. The `redirectTo` property currently points to `'/'`, but in this case you will want to modify it to point to `'/dashboard'` because that is the main page of the StockDog application.

Using the Routes

With all the required steps accomplished for adding new client-side routes and wiring together the skeleton dashboard and watchlist views, you can now begin linking together the pages within StockDog using the configured routes. Open the `stkwatchlistpanel.js` file containing the directive that renders out the watchlist panel, and inject the AngularJS `$routeParams` service as a dependency alongside the current `$location`, `$modal`, and `WatchlistService` dependencies. The call to `.directive()` should now look something like this:

```

.directive('stkWatchlistPanel',
  function ($location, $modal, $routeParams, WatchlistService) {

```

Next, you will be adding a new `$scope` variable that will keep track of the current watchlist being displayed, as well as a `gotoList()` function that will send users to the desired watchlist view. You can accomplish this by adding the following code to the directive's implementation:

```

    $scope.currentList = $routeParams.listId;
    $scope.gotoList = function (listId) {
      $location.path('watchlist/' + listId);
    };

```

Once again, the `$location` service is used to route the user to the desired watchlist view, which includes the `listId`. At this point, you might be asking yourself where this `listId` that is passed into the `gotoList()` function is coming from. If you remember, when you first created the `watchlist-panel.html` template view, you used the built-in `ng-repeat` directive to iterate over all the watchlists fetched from the `WatchlistService`. To wire this function into the directive's template, you need to add the `ng-click` directive to the `<a>` tag, which contains a call to the `gotoList()` function that will be evaluated whenever the DOM element is clicked. Because the `stkWatchlistPanel` is used on both the main dashboard and individual watchlist views, you should also go ahead and add an `ng-class` directive to the same element, which can be used to add the active class from Bootstrap to the `<a>` tag for the list that the user is currently viewing. The modifications to the `watchlist-panel.html` file located inside the `app/view/templates/` directory are shown here:

```

<a class="list-group-item"
  ng-class="{ active: currentList == list.id }"
  ng-repeat="list in watchlists track by $index"
  ng-click="gotoList(list.id)">

```

Notice that the newly defined `currentList` variable that was attached to the `$scope` is used to evaluate whether the active class should be present on the element. In the next section, you

will be laying the foundation structure for the dashboard and watchlist views. Because the `<stk-watchlist-panel>` element is used within the context of both views, take a moment to delete its current reference from within the `index.html` file.

Template Views

At this point, you might be wondering how AngularJS knows to load the `dashboard.html` and `watchlist.html` views specified in the `$routeProvider`'s `template` option for each configured route. The key component behind this functionality is the `ngView` directive, which was included in the `index.html` file when you initially scaffolded your project with Yeoman. This directive requires the `ngRoute` module to be installed to function and handles inserting the view template defined by the `$route` service into the layout template, which in this case is the `index.html` file. It is important to note that the route's template is inserted in the exact DOM location where the `<ng-view>` element resides.

In its current state, the StockDog application is devoid of any useful functionality, so go ahead and modify your generated `dashboard.html` and `watchlist.html` files to resemble those shown in Listing 1-6 and Listing 1-7, respectively.

LISTING 1-6: app/views/dashboard.html

```
<div class="row">
  <!-- Left Column -->
  <div class="col-md-3">
    <stk-watchlist-panel></stk-watchlist-panel>
  </div>

  <!-- Right Column -->
  <div class="col-md-9">
    <div class="panel panel-info">
      <div class="panel-heading">
        <span class="glyphicon glyphicon-globe"></span>
        Portfolio Overview
      </div>
      <div class="panel-body">
      </div>
    </div>
  </div>
</div>
```

LISTING 1-7: app/views/watchlist.html

```
<div class="row">
  <!-- Left Column -->
  <div class="col-md-3">
    <stk-watchlist-panel></stk-watchlist-panel>
  </div>

  <!-- Right Column -->
```

```
<div class="col-md-9">
  </div>
</div>
```

Both the `dashboard.html` and `watchlist.html` templates use Bootstrap's grid system to create two distinct columns, with the `<stk-watchlist-panel>` being included in the left column of each view. Now that the modifications to both these files are complete, go ahead and navigate to the Dashboard view in your browser by visiting `http://localhost:9000/#/dashboard`. For testing purposes, take a moment to add a new watchlist to the panel and then click on the newly created list item. The `ngClick` directive you added should evaluate the `gotoList()` function of the `stkWatchlistPanel` directive, which will result in your application routing you to a uniquely named view for that watchlist. You should now see something along the lines of `http://localhost:9000/#/watchlist/1` inside your browser's URL bar. Pressing the Back button of your browser should take you back to the main Dashboard view.

Congratulations! You have successfully implemented the client-side routing for both views of the StockDog application. In doing so, you have seen how the `ngRoute` module can be used to implement deep linking inside an AngularJS application, as well as learning how the `ngView` directive can be used to load route templates. If you've gotten stuck at any point during this step, take a moment to examine the completed code by referring to the `step-3` directory inside the companion code for this chapter or checking out the corresponding tag of the GitHub repository.

STEP 4: CREATING A NAVIGATION BAR

With client-side routing out of the way, you can now take a few moments to spruce up the navigation bar of the StockDog application by using native Bootstrap components. In its current state, your application's navigation bar has yet to be modified from what was initially scaffolded for you by the Yeoman generator. In this section, you will replace this default navigation bar with one that is more fluid and allows for appropriate navigation between the two main views of the StockDog application.

Updating the HTML

First, you need to delete a few lines of code from your current `app/index.html` file. Go ahead and open that file and start by deleting the line containing the opening `<body ng-app="stockDogApp">` tag, located around line 19, and only stop right before the HTML comment containing `<!-- build:js(.) scripts/vendor.js -->`, located around line 61. If you have been following along with the example code, you should have deleted around 42 lines from this file.

NOTE *It is critically important that you do not delete the HTML comment containing `<!-- build:js(.) scripts/vendor.js -->` because this inline comment is used by the build system, discussed later in this chapter, to optimize the final distributable version of your application.*

Now that you have deleted the necessary lines from your application's `index.html` file, go ahead and insert the markup shown next in place of the lines that were just deleted:

```
<!-- [1] Load MainCtrl -->
<body ng-app="stockDogApp" ng-controller="MainCtrl">
  <nav class="navbar navbar-inverse" role="navigation" ng-cloak>
    <div class="container-fluid">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle"
          data-toggle="collapse" data-target="#main-nav">
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="/">Stock Dog</a>
      </div>

      <!-- Collect the nav links and other content for toggling -->
      <div class="collapse navbar-collapse" id="main-nav">
        <ul class="nav navbar-nav navbar-right">
          <!-- [2] Add active class to necessary item -->
          <li ng-class="{active: activeView === 'dashboard'}">
            <a href="/">Dashboard</a>
          </li>
          <li ng-class="{active: activeView === 'watchlist'}"
            class="dropdown">
            <a class="dropdown-toggle" data-toggle="dropdown">
              Watchlists <b class="caret"></b>
            </a>
            <ul class="dropdown-menu">
              <li ng-if="!watchlists.length" class="dropdown-header">
                No lists found
              </li>
              <!-- [3] Create a unique link for each watchlist -->
              <li ng-repeat="list in watchlists track by $index">
                <a href="/#/watchlist/{{list.id}}">{{list.name}}</a>
              </li>
            </ul>
          </li>
        </ul>
      </div><!-- /.navbar-collapse -->
    </div><!-- /.container-fluid -->
  </nav>

  <!-- Main container -->
  <div class="container-fluid" id="main">
    <div ng-view=""></div>

    <div class="footer">
      <p>Built with <span class="glyphicon glyphicon-heart"></span></p>
    </div>
  </div>
```

The first difference you should notice in this block of HTML is the use of the `ng-controller` directive on the `body` tag [1]. In the previous section, you discovered how the `ngRoute` module

could be used to load the desired controllers and views for a specific route. However, in this case, you want to force AngularJS to load the `MainCtrl` controller because it will be used for logic that should be applied to your application regardless of the current evaluated route. This approach demonstrates a simple way to encapsulate application-wide logic into a single controller.

Another addition to this markup that is worth mentioning is the use of the `ng-class` directive [2] to add the Bootstrap active class to the navigation menu links, depending on the value of the `activeView` scope variable. The final AngularJS component used in this markup for the navigation bar is the `ng-repeat` directive. It is used here [3] to create a unique `` for each list in the `watchlist` scope variable. This example shows how nav links can be dynamically generated based on data that an AngularJS controller provides. In its current state, your application should be displaying an error in your browser's console because the `MainCtrl` has yet to be defined. This issue will be resolved in the next section when you create and implement the `MainCtrl`.

Creating MainCtrl

You have seen how to use the Yeoman subgenerators to scaffold out new services, directives, and routes. Now you will be following the same process to have Yeoman scaffold out a new AngularJS controller. To accomplish this, go ahead and run the following from your command line:

```
yo angular:controller Main
```

This instructs Yeoman to create a new controller named `MainCtrl` inside the `app/scripts/controllers/main.js` file and add the appropriate `<script>` tag reference to your `app/index.html` file. Open this newly created file and replace its entire contents with the code shown in Listing 1-8.

LISTING 1-8: `app/scripts/controllers/main.js`

```
'use strict';

angular.module('stockDogApp')
  .controller('MainCtrl', function ($scope, $location, WatchlistService) {
    // [1] Populate watchlists for dynamic nav links
    $scope.watchlists = WatchlistService.query();

    // [2] Using the $location.path() function as a $watch expression
    $scope.$watch(function () {
      return $location.path();
    }, function (path) {
      if (_.contains(path, 'watchlist')) {
        $scope.activeView = 'watchlist';
      } else {
        $scope.activeView = 'dashboard';
      }
    });
  });
```

The `MainCtrl` uses both the `$location` service, provided by AngularJS, as well as the `WatchlistService`, created earlier in this chapter. The `WatchlistService` is used to populate the `$scope.watchlist` variable [1], which is used in the markup to dynamically create multiple drop-down links for the top-level Watchlists navigation item. For this controller to figure out the current application route, the `$location` service is used in conjunction with the `$scope.watch()` function so that every time the value returned from the `$location.path()` function changes, your callback function can appropriately update the `$scope.activeView` variable (using the `_.contains()` function from `Lodash`), which is used to add an active class to the navigation bar. The `$scope.$watch()` function is covered in more detail later in this book. For now, all you need to know is that it watches the value returned from the first function for changes and invokes the callback specified as its second argument on each change.

Your application's navigation bar should now be fully functional. See Figure 1-7. For testing purposes, go ahead and create a new watchlist (if you haven't already) and then navigate to it by selecting the appropriate link from the Watchlists drop-down in the nav bar. Then click the Dashboard link to return to the initial view of the StockDog application. If you've gotten stuck at any point during this step, take a moment to examine the completed code by referring to the `step-4` directory inside the companion code for this chapter or checking out the corresponding tag of the GitHub repository.

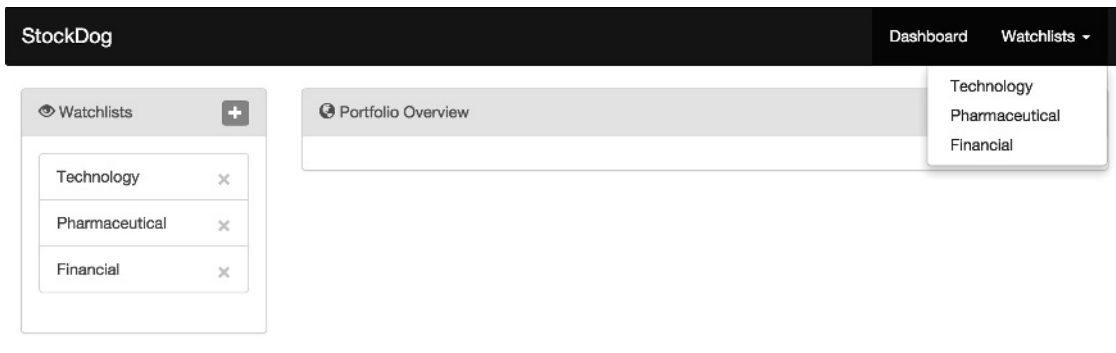


FIGURE 1-7

STEP 5: ADDING STOCKS

The next major piece of functionality that needs to be implemented for StockDog is the ability to add stocks to a watchlist. In a similar fashion to the way users can add a new watchlist to their portfolio, you will be creating a new modal that will be displayed after clicking a specific button on the watchlist view. This modal will allow users to search for companies listed on the NYSE, NASDAQ, and AMEX stock exchanges, and add them, along with a specified number of shares, to part of a desired watchlist. In this section, you will learn how to leverage the various mechanisms provided by AngularJS to accomplish this task.

Creating the CompanyService

The first order of business is to create a new AngularJS service that will be in charge of fetching a list of companies and relevant data for each of the three major exchanges. Normally, this would be accomplished by communicating with a back-end service of some kind, but for the purposes of this application, a JSON file has been created for your perusal. You can find the `companies.json` file inside the `step-5/app/` directory of the associated companion code, as well as inside the `app/` directory of the GitHub repo <https://github.com/diegonetto/stock-dog>. Once you've downloaded the file, go ahead and save it inside the `app/` directory of your local project. Next, run the following from your command line to scaffold out and wire up a new AngularJS service:

```
yo angular: service Company-Service
```

This creates a `company-service.js` file inside your `app/scripts/services` directory. The implementation for this service is shown in Listing 1-9. Notice that the `$resource` service, which creates a resource object that facilitates interaction with RESTful server-side data sources and will be covered in detail in a Chapter 8, “Server Communication,” is injected as a dependency. The takeaway at this point is that the `$resource` service is taking care of fetching the `companies.json` file from your local file system and returning an object that will allow you to query against the provided list of publicly traded companies.

LISTING 1-9: `app/scripts/services/company.js`

```
'use strict';

angular.module('stockDogApp')
  .service('CompanyService', function CompanyService($resource) {
    return $resource('companies.json');
  });
```

You will be making use of this newly created `CompanyService` shortly, but before continuing onto the next section, take a moment to open the `Gruntfile.js` located in your project's root directory and find the `src` property of the `copy` task, located around line 300. You will need to add `json` to the `src` array so that the `companies.json` file will be copied into the built distributable when you are preparing your application for production later in this chapter. The modification should leave the first entry of the `src` array looking like this:

```
'*.{ico,png,txt,json}',
```

Creating the AddStock Modal

With the `CompanyService` complete, it is time to create a new view that will serve as the modal for allowing your users to add new stocks to the currently selected watchlist. Go ahead and create a new file named `addstock-modal.html` inside your `app/views/templates/` directory. You can see the implementation for this view in Listing 1-10.

LISTING 1-10: app/views/templates/addstock-modal.html

```
<div class="modal" tabindex="-1" role="dialog">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <button type="button" class="close" ng-click="$hide()">&times;</button>
        <h4 class="modal-title">Add New Stock</h4>
      </div>

      <form role="form" id="add-stock" name="stockForm">
        <div class="modal-body">
          <div class="form-group">
            <label for="stock-symbol">Symbol</label>
            // [1] Use ng-options with label syntax and bs-typeahead directive
            <input type="text"
              class="form-control"
              id="stock-symbol"
              placeholder="Stock Symbol"
              ng-model="newStock.company"
              ng-options="company as company.label for company in companies"
              bs-typeahead
              required>
          </div>
          // [2] Only accept numbers for shares owned
          <div class="form-group">
            <label for="stock-shares">Shares Owned</label>
            <input type="number"
              class="form-control"
              id="stock-shares"
              placeholder="# Shares Owned"
              ng-model="newStock.shares"
              required>
          </div>
        </div>
        <div class="modal-footer">
          <button type="submit"
            class="btn btn-success"
            ng-click="addStock()"
            ng-disabled="!stockForm.$valid">Add</button>
          <button type="button"
            class="btn btn-danger"
            ng-click="$hide()">Cancel</button>
        </div>
      </form>
    </div>
  </div>
</div>
```

This should look fairly similar to the previous modal for adding new watchlists to StockDog. The first input [1] uses the `bs-typeahead` directive from the AngularStrap project, which utilizes the native Angular `ng-options` directive for providing the data required for the typeahead mechanism to function. The `ng-options` directive accepts multiple forms of syntax. In this case, you are forcing it to

use the `label` property of each company object in the `companies` scope variable, which will be created inside the `WatchlistCtrl` shortly, as the data to be displayed in the typeahead recommendations. The second input `[2]` simply allows users to specify the number of shares owned of a particular stock.

Updating the WatchlistService

Before continuing on to developing the `WatchlistCtrl` and associated watchlist view, you need to make a few modifications to the existing `WatchlistService`. To abstract the various calculations and interactions between watchlists and their associated stocks, you will be creating two separate objects to be used as models for the required behaviors. Inside the top of the service implementation function of your `watchlist-service.js` file, located inside the `app/scripts/services/` directory, add the following lines of code to create a `StockModel` object with a `save()` function:

```
// Augment Stocks with additional helper functions
var StockModel = {
  save: function () {
    var watchlist = findById(this.listId);
    watchlist.recalculate();
    saveModel();
  }
};
```

Because watchlists are composed of many stocks, you will also need to create a `WatchlistModel` with `addStock()`, `removeStock()`, and `recalculate()` functions, as shown here:

```
// Augment watchlists with additional helper functions
var WatchlistModel = {
  addStock: function (stock) {
    var existingStock = _.find(this.stocks, function (s) {
      return s.company.symbol === stock.company.symbol;
    });
    if (existingStock) {
      existingStock.shares += stock.shares;
    } else {
      _.extend(stock, StockModel);
      this.stocks.push(stock);
    }
    this.recalculate();
    saveModel();
  },
  removeStock: function (stock) {
    _.remove(this.stocks, function (s) {
      return s.company.symbol === stock.company.symbol;
    });
    this.recalculate();
    saveModel();
  },
  recalculate: function () {
    var calcs = _.reduce(this.stocks, function (calcs, stock) {
      calcs.shares += stock.shares;
      calcs.marketValue += stock.marketValue;
      calcs.dayChange += stock.dayChange;
    }, {});
  }
};
```

```

        return calcs;
    }, { shares: 0, marketValue: 0, dayChange: 0 });

    this.shares = calcs.shares;
    this.marketValue = calcs.marketValue;
    this.dayChange = calcs.dayChange;
}
};
```

Finally, the method in which data is serialized and unserialized from `LocalStorage` needs to be modified because you will be extending the two previously created models to create the proper data structure in memory required to power the application. Modify the existing `loadModel()` and `this.save()` functions to look like those shown here:

```
// Helper: Load watchlists from localStorage
var loadModel = function () {
  var model = {
    watchlists: localStorage['StockDog.watchlists'] ?
      JSON.parse(localStorage['StockDog.watchlists']) : [],
    nextId: localStorage['StockDog.nextId'] ?
      parseInt(localStorage['StockDog.nextId']) : 0
  };
  _.each(model.watchlists, function (watchlist) {
    _.extend(watchlist, WatchlistModel);
    _.each(watchlist.stocks, function (stock) {
      _.extend(stock, StockModel);
    });
  });
  return model;
};

// Save a new watchlist to watchlists model
this.save = function (watchlist) {
  watchlist.id = Model.nextId++;
  watchlist.stocks = [];
  _.extend(watchlist, WatchlistModel);
  Model.watchlists.push(watchlist);
  saveModel();
};
```

Implementing WatchlistCtrl

Next, you will be modifying the current `WatchlistCtrl`, which is still an empty skeleton that was created by Yeoman during the scaffolding process. Open up the `watchlist.js` file, located inside the `app/scripts/controllers/` directory, and modify it to look like Listing 1-11.

LISTING 1-11: app/scripts/controllers/watchlist.js

[illegible]

```

// [1] Initializations
$scope.companies = CompanyService.query();
$scope.watchlist = WatchlistService.query($routeParams.listId);
$scope.stocks = $scope.watchlist.stocks;
$scope.newStock = {};
var addStockModal = $modal({
  scope: $scope,
  template: 'views/templates/addstock-modal.html',
  show: false
});

// [2] Expose showStockModal to view via $scope
$scope.showStockModal = function () {
  addStockModal.$promise.then(addStockModal.show);
};

// [3] Call the WatchlistModel addStock() function and hide the modal
$scope.addStock = function () {
  $scope.watchlist.addStock({
    listId: $routeParams.listId,
    company: $scope.newStock.company,
    shares: $scope.newStock.shares
  });
  addStockModal.hide();
  $scope.newStock = {};
};
});

```

You should notice that `$routeParams`, `$modal`, `WatchlistService`, and `CompanyService` are all being injected as dependencies. The `CompanyService`'s `query()` function, provided by the object returned from using the `$resource` service as previously mentioned, is invoked to populate the `companies` scope variable, which will be utilized in the watchlist view momentarily. The rest of the code is straightforward, with the `WatchlistService` being used to initialize the `watchlist` scope variable, which is in turn used to retrieve the current watchlist variable using the `listId` passed along in the route parameters [1]. Next, the modal itself is instantiated, and definitions are made for the [2] `showStockModal()` and [3] `addStock()` functions.

Modifying the Watchlist View

Because modifications were made to the way watchlists were saved and loaded, take a moment to delete all current watchlists from your application before proceeding with the updates to the watchlist view markup. Once that is complete, go ahead and modify the existing `app/views/watchlist.html` file to include a Bootstrap panel where the list of stocks will be displayed. As it stands, this file should only contain one row comprised of two columns, with the left column being comprised of the `stk-watchlist-panel` directive. Modify the right column of this file to match the HTML markup shown in Listing 1-12.

LISTING 1-12: `app/views/watchlist.html`

```

<div class="row">
  <!-- Left Column -->

```

continues

LISTING 1-12 *(continued)*

```

<div class="col-md-3">
  <stk-watchlist-panel></stk-watchlist-panel>
</div>

<!-- Right Column -->
<div class="col-md-9">
  <div class="panel panel-info">
    <div class="panel-heading">
      <span class="glyphicon glyphicon-list"></span>
      {{watchlist.description}}
      <button type="button"
        class="btn btn-success btn-xs pull-right"
        ng-click="showStockModal()">
      <span class="glyphicon glyphicon-plus"></span>
    </button>
  </div>
  <div class="panel-body table-responsive">
    <div ng-hide="stocks.length" class="jumbotron">
      <h1>Woof.</h1>
      <p>Looks like you haven't added any stocks to this watchlist yet!</p>
      <p>Do so now by clicking the
        <span class="glyphicon glyphicon-plus"></span> located above.
      </p>
    </div>
    <!--[1] loop over all stocks and display company symbols -->
    <p ng-repeat="stock in stocks">{{stock.company.symbol}}</p>
  </div>
</div>
</div>
</div>

```

By now, you should be comfortable using the `ng-click`, `ng-hide`, and `ng-repeat` directives, the latter of which is currently being used for simply displaying the stock's company ticker symbol. This will be revisited in a later step when it comes time to build the stock table directives.

At this point, you should be able to add new stocks to a selected watchlist by clicking the green plus button in the panel heading, selecting a stock by searching for its company name or ticker symbol, and clicking the desired typeahead recommendation. See Figure 1-8. If your application is not functioning properly, be sure to check your browser's developer tools console for errors, and take a moment to review the code included in this section. You can refer to the `step-5` directory inside the companion code for this chapter or check out the corresponding tag of the GitHub repository.

STEP 6: INTEGRATING WITH YAHOO FINANCE

Now that your StockDog application is able to manage manipulating watchlists and stocks, it is time to begin fetching quote information from an external service provider—in this case Yahoo Finance. In this section, you will create a new AngularJS service that will be responsible for making asynchronous HTTP requests to the Yahoo Finance API and updating the in-memory data structure that powers the application.

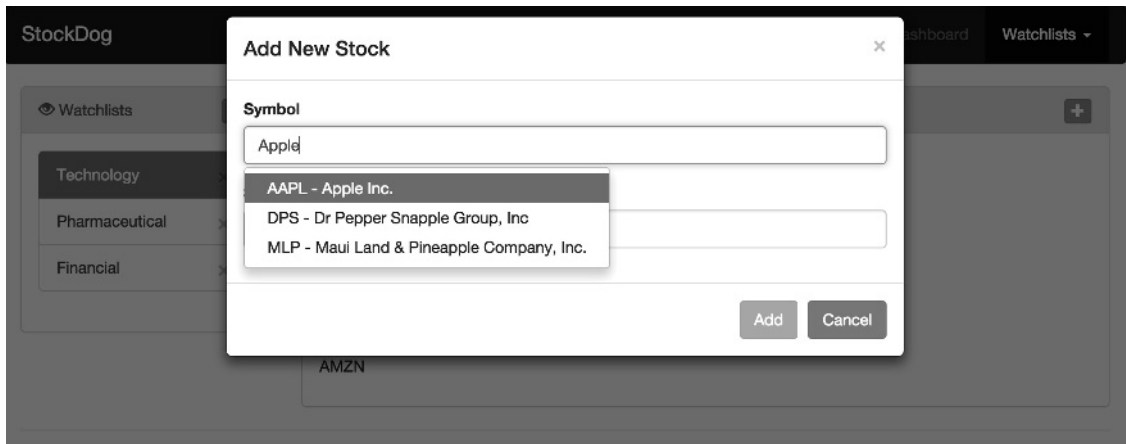


FIGURE 1-8

Creating the QuoteService

To encapsulate the HTTP requests and response parsing into a reusable component, you will be creating a new AngularJS service. Run the following command from your terminal to have Yeoman scaffold your new `QuoteService`:

```
yo angular:service Quote-Service
```

As seen several times in this chapter, this creates a skeleton implementation of, in this case, an AngularJS service named `QuoteService` inside of a newly created `quote-service.js` file located within your `app/scripts/services` directory. You can see the entire implementation for the `QuoteService` in Listing 1-13.

LISTING 1-13: `app/scripts/services/quote-service.js`

```
'use strict';

angular.module('stockDogApp')
  .service('QuoteService', function ($http, $interval) {
    var stocks = [];
    var BASE = 'http://query.yahooapis.com/v1/public/yql';

    // [1] Handles updating stock model with appropriate data from quote
    var update = function (quotes) {
      console.log(quotes);
      if (quotes.length === stocks.length) {
        _.each(quotes, function (quote, idx) {
          var stock = stocks[idx];
          stock.lastPrice = parseFloat(quote.LastTradePriceOnly);
          stock.change = quote.Change;
          stock.percentChange = quote.ChangeinPercent;
        });
      }
    };

    $interval(update, 1000);
  });
```

continues

LISTING 1-13 *(continued)*

```

        stock.marketValue = stock.shares * stock.lastPrice;
        stock.dayChange = stock.shares * parseFloat(stock.change);
        stock.save();
    });
}
};

// [2] Helper functions for managing which stocks to pull quotes for
this.register = function (stock) {
    stocks.push(stock);
};
this.deregister = function (stock) {
    _.remove(stocks, stock);
};
this.clear = function () {
    stocks = [];
};

// [3] Main processing function for communicating with Yahoo Finance API
this.fetch = function () {
    var symbols = _.reduce(stocks, function (symbols, stock) {
        symbols.push(stock.company.symbol);
        return symbols;
    }, []);
    var query = encodeURIComponent('select * from yahoo.finance.quotes ' +
        'where symbol in (\'' + symbols.join(',') + '\')');
    var url = BASE + '?' + 'q=' + query + '&format=json&diagnostics=true' +
        '&env=http://datatables.org/alltables.env';
    $http.jsonp(url + '&callback=JSON_CALLBACK')
        .success(function (data) {
            if (data.query.count) {
                var quotes = data.query.count > 1 ?
                    data.query.results.quote : [data.query.results.quote];
                update(quotes);
            }
        })
        .error(function (data) {
            console.log(data);
        });
};

// [4] Used to fetch new quote data every 5 seconds
$interval(this.fetch, 5000);
});

```

Because the `QuoteService` is in charge of communicating with the Yahoo Finance API, you'll notice that the `$http` service was injected as a dependency. The `$interval` service that was also injected is Angular's wrapper for `window.setInterval`. Internally this service keeps track of an array of stocks for which quote data should be retrieved. The `update()` function [1] handles parsing the response from Yahoo Finance into the required stock model properties. This code

also contains helper functions [2] for adding, removing, and clearing the internal array of stocks being tracked. Finally, the `fetch()` function [3] generates the appropriate Yahoo Finance query URL before invoking the `$http` service to make an asynchronous request to the desired endpoint. The response from Yahoo is then passed into the `update()` function for processing as previously described.

Invoking Services from the Console

Because your newly created `QuoteService` has not been injected and used anywhere in the `StockDog` application at this time, the easiest way to quickly spot-check this service is by typing a few lines into the console of your browser developer tools. Go ahead and open that now and paste the following lines directly into the browser console:

```
Quote = angular.element(document.body).injector().get('QuoteService')
Watchlist = angular.element(document.body).injector().get('WatchlistService')
Quote.register(Watchlist.query()[0].stocks[0])
```

This grabs a reference to the `QuoteService` and `WatchlistService` and then invokes the `QuoteService`'s `register()` function with the first stock of the first watchlist available. (So make sure you have created at least one watchlist and added at least one stock.) Within five seconds, you should see an array containing a single object. Inspecting that object should show you all the data provided by the Yahoo Finance API for that one particular stock, similar to Figure 1-9.

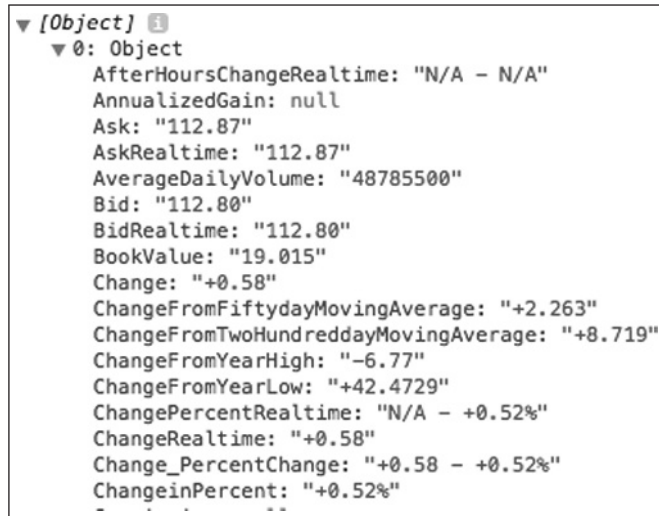


FIGURE 1-9

Now that you have finished creating the `QuoteService` and verified that it is successfully pulling data from the Yahoo Finance API, you are ready to move onto the next section and display that data in a table on the watchlist view. If your application is not functioning properly, please refer to the `step-6` directory inside the companion code for this chapter or check out the corresponding tag of the GitHub repository.

STEP 7: CREATING THE STOCK TABLE

In this section, you will be exposed to a more sophisticated use of AngularJS directives. Specifically, you will see how directives can communicate data between themselves as you build a table for displaying information on a stock's performance.

Creating the StkStockTable Directive

To get started, you will be creating a new directive for the stock table. As you've seen several times, you can do this using the AngularJS Yeoman generator by running the following from your command line:

```
yo angular:directive stk-Stock-Table
```

This creates a `stk-stock-table.js` file inside of `app/scripts/directives` and links the new JavaScript file inside of `index.html`. The implementation of the `stkStockTable` directive is shown in Listing 1-14.

LISTING 1-14: `app/scripts/directives/stk-stock-table.js`

```
'use strict';

angular.module('stockDogApp')
  .directive('stkStockTable', function () {
    return {
      templateUrl: 'views/templates/stock-table.html',
      restrict: 'E',
      // [1] Isolate scope
      scope: {
        watchlist: '='
      },
      // [2] Create a controller, which serves as an API for this directive
      controller: function ($scope) {
        var rows = [];

        $scope.$watch('showPercent', function (showPercent) {
          if (showPercent) {
            _.each(rows, function (row) {
              row.showPercent = showPercent;
            });
          }
        });

        this.addRow = function (row) {
          rows.push(row);
        };

        this.removeRow = function (row) {
          _.remove(rows, row);
        };
      },

      // [3] Standard link function implementation
```

```

    link: function ($scope) {
        $scope.showPercent = false;
        $scope.removeStock = function (stock) {
            $scope.watchlist.removeStock(stock);
        };
    }
};
});

```

The first thing you should notice is that this directive contains an object for its `scope` property [1]. By isolating the scope of a directive in this way, you can bind an attribute of the directive's DOM element. You will explore this in more detail in Chapter 4, “Data Binding,” but for now, know that when you use the `stkStockTable` directive, you must include an attribute named `watchlist` and assign it an expression to be evaluated. Also of note in this example is that this directive contains a controller property [2]. This, in a more general sense, is how you expose an API for other directives to use for communication. Because inside the controller property's implementation both the `addRow()` and `removeRow()` function are attached to the `this` object, they will be available for external use. The concept here is that the `stkStockTable` directive keeps track, internally, of all the rows in the table. This allows it to modify the rows if needed, as is the case, in this example, for toggling the `showPercent` property of each row's scope. Finally, this directive also includes the `link` property [3], which is typical for DOM manipulation, and in this case simply initializes the `showPercent` scope variable and exposes a `removeStock()` function via the top-level directive scope.

Creating the StkStockRow Directive

Now that the main `stkStockTable` directive has been created, it's time to create the directive that will be repeated for each table row. Run the following from the command line to create a new `stkStockRow` directive:

```
yo angular:directive stk-Stock-Row
```

This creates the `stk-stock-row.js` file inside the `app/scripts/directives` directory with a skeleton for the `stkStockRow` directive. The implementation for this directive is shown in Listing 1-15.

LISTING 1-15: app/scripts/directives/stk-stock-row.js

```

'use strict';

angular.module('stockDogApp')
    .directive('stkStockRow', function ($timeout, QuoteService) {
        return {
            // [1] Use as element attribute and require stkStockTable controller
            restrict: 'A',
            require: '^stkStockTable',
            scope: {
                stock: '=',
                isLast: '='
            },
            // [2] The required controller will be made available at the end

```

continues

LISTING 1-15 *(continued)*

```
link: function ($scope, $element, $attrs, stockTableCtrl) {
    // [3] Create tooltip for stock-row
    $element.tooltip({
        placement: 'left',
        title: $scope.stock.company.name
    });

    // [4] Add this row to the TableCtrl
    stockTableCtrl.addRow($scope);

    // [5] Register this stock with the QuoteService
    QuoteService.register($scope.stock);

    // [6] Deregister company with the QuoteService on $destroy
    $scope.$on('$destroy', function () {
        stockTableCtrl.removeRow($scope);
        QuoteService.deregister($scope.stock);
    });

    // [7] If this is the last 'stock-row', fetch quotes immediately
    if ($scope.isLast) {
        $timeout(QuoteService.fetch);
    }

    // [8] Watch for changes in shares and recalculate fields
    $scope.$watch('stock.shares', function () {
        $scope.stock.marketValue = $scope.stock.shares *
            $scope.stock.lastPrice;
        $scope.stock.dayChange = $scope.stock.shares *
            parseFloat($scope.stock.change);
        $scope.stock.save();
    });
}
};
});
```

For this directive, only `$timeout` and `QuoteService` are injected as dependencies. Also, you might have already noticed that the `restrict` property [1] has been set to `A`, which means that `stkStockRow` is meant to be used as an attribute of a DOM element instead of as a DOM element itself as was the case with the previously created directives. You should also make note of the use of the `require` property. This is how you tell the directive that it needs a specific controller, which in this case was defined inside the `stkStockTable` directive. The `^` prefix instructs this directive to search for controllers on its parent scopes, which is exactly what you want it to do in this case. The required controller is then available via the last parameter of the `link` function, as seen in [2]. Because each row has its own tooltip markup, this directive is a great location to put the tooltip initialization code [3]. The rest of the code takes care of registering the `$scope` for each row using the `stkStockTable` directive's `addRow()` function [4], registering the row's stock with the `QuoteService` on creation [5] and deregistering it when the row is destroyed [6], as well as immediately triggering a `QuoteService.fetch()` call if the currently created row is the last one in the table [7]. Finally, a `$watch()` is used to monitor changes to the stock's number of shares so that the appropriate calculations can be made [8].

Creating the Stock Table Template

With both the `stkStockTable` and `stkStockRow` directives now complete, the next order of business is to create a new HTML template view for the stock table. Go ahead and create a new file named `stock-table.html` inside your `app/views/templates/` directory and make it look like the markup shown in Listing 1-16.

LISTING 1-16: `app/views/templates/stock-table.html`

```
<table class="table">
  <thead>
    <tr>
      <td>Symbol</td>
      <td>Shares Owned</td>
      <td>Last Price</td>
      <td>Price Change
        <span> (
          <!-- [1] Toggle showPercent scope variable on click -->
          <span ng-disabled="showPercent === false">
            <a ng-click="showPercent = !showPercent">${</a>
          </span>|
          <span ng-disabled="showPercent === true">
            <a ng-click="showPercent = !showPercent">%</a>
          </span>)
        </span>
      </td>
      <td>Market Value</td>
      <td>Day Change</td>
    </tr>
  </thead>
  <!-- [2] Only show footer if more than one stock exists -->
  <tfoot ng-show="watchlist.stocks.length > 1">
    <tr>
      <td>Totals</td>
      <td>{{watchlist.shares}}</td>
      <td></td>
      <td></td>
      <td>{{watchlist.marketValue}}</td>
      <td>{{watchlist.dayChange}}</td>
    </tr>
  </tfoot>
  <tbody>
    <!-- [3] Use stk-stock-row to create row for each stock -->
    <tr stk-stock-row
      ng-repeat="stock in watchlist.stocks track by $index"
      stock="stock"
      is-last="!$last">
      <td>{{stock.company.symbol}}</td>
      <td>{{stock.shares}}</td>
      <td>{{stock.lastPrice}}</td>
      <td>
        <span ng-hide="showPercent">{{stock.change}}</span>
        <span ng-show="showPercent">{{stock.percentChange}}</span>
      </td>
    </tr>
  </tbody>
</table>
```

continues

LISTING 1-16 *(continued)*

```

        </td>
        <td>{{stock.marketValue}}</td>
        <td>{{stock.dayChange}}
            <button type="button" class="close"
                ng-click="removeStock(stock)">x</button>
        </td>
    </tr>
</tbody>
</table>

```

Although the markup for `stock-table.html` is not overly complicated, there are a few things worth pointing out. First, inside the `<thead>`, you should notice that the Price Change header cell contains two spans with `ng-click` directives that assign a value to the `showPercent` scope variable [1]. This is the first example using this form of an expression and is a helpful way to accomplish simple tasks, in this case toggling a Boolean without creating a scope function. You should also note the use of `ng-show` to only display the table footer if there is more than one stock in the current watchlist [2] because it contains calculated totals. Finally, although this view template is for the `stkStockTable` directive, under the hood it uses an `ng-repeat` to create `<tr>` elements containing the `stkStockRow` directive [3]. Using external directives inside another directive's template is perfectly acceptable; just take care in not overcomplicating your approach because you may run into situations in which you have to manually compile child directive templates using the `$compile` service.

Updating the Watchlist View

The only remaining task in completing this step is to invoke the `stkStockTable` directive by including it in StockDog's watchlist view. Open your project's `app/views/watchlist.html` file and locate the `<p>` tag containing the `ng-repeat` directive. Instead of simply displaying the stock's company symbol, you want to render the entire interactive table. Replace that entire line with the following code to accomplish this task:

```
<stk-stock-table ng-show="stocks.length" watchlist="watchlist">
```

Congratulations on successfully completing the first pass over the stock table! See Figure 1-10. You might be thinking that it isn't the most beautiful table you've ever created, but don't fret. Over the next three sections, you will be refining it into a more polished product. In the next section, you will see how to make individual cells editable, adding even more interactivity to your table. If your application is not functioning properly, please refer to the `step-7` directory inside the companion code for this chapter or check out the corresponding tag of the GitHub repository.

STEP 8: INLINE FORM EDITING

Now that StockDog has a functioning table that can display information on the various stocks being tracked by a watchlist, the next step is to make the application more interactive by allowing users to edit the number of shares owned for each stock. Because data is being displayed in a table, a common paradigm for editing values is to modify them inline, much like a spreadsheet. In this

section, you will see how to create a directive that can be used in conjunction with HTML5's `contenteditable` attribute to accomplish this functionality.

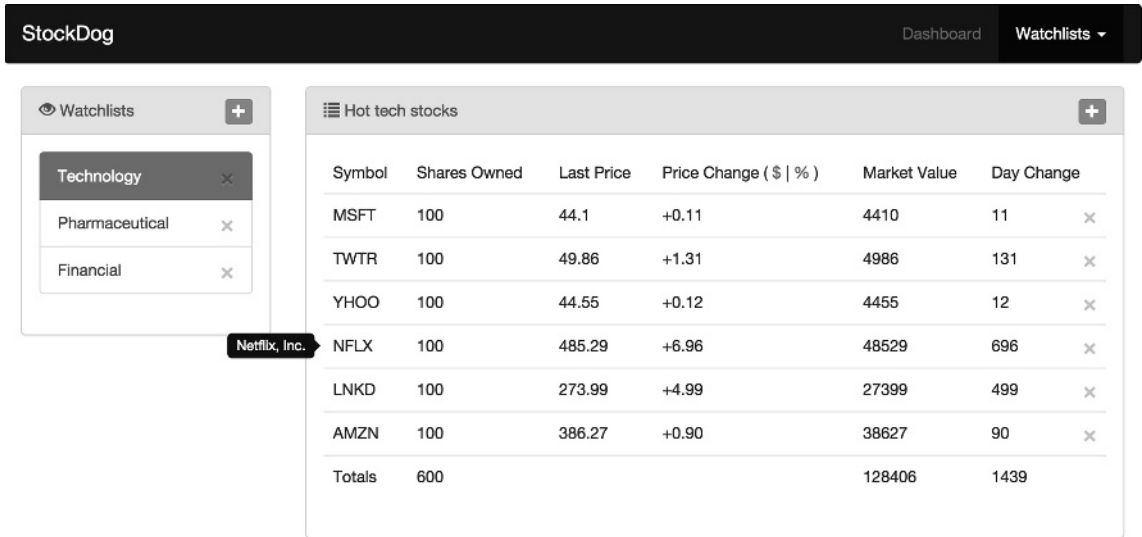


FIGURE 1-10

Creating the Contenteditable Directive

Because this new directive will be extending the `contenteditable` attribute's functionality, it must share the same name. Run the following command from your terminal to scaffold out a new AngularJS directive using Yeoman:

```
yo angular:directive contenteditable
```

This creates a new file named `contenteditable.js` inside your `app/scripts/directives/` directory. The `contenteditable` directive is restricted to an attribute and performs sanitization and validation of user-inputted data. You can find the full implementation of this new directive in Listing 1-17.

LISTING 1-17: `app/scripts/directives/contenteditable.js`

```
'use strict';

var NUMBER_REGEXP = /^\\s*(\\-|\\+)?(\\d+|(\\d*(\\.\\d*))\\s*$)/;

angular.module('stockDogApp')
  .directive('contenteditable', function ($sce) {
    return {
      restrict: 'A',
      require: 'ngModel', // [1] Get a hold of NgModelController
```

continues

LISTING 1-17: *(continued)*

```
link: function($scope, $element, $attrs, ngModelCtrl) {
    if(!ngModelCtrl) { return; } // do nothing if no ng-model

    // [2] Specify how UI should be updated
    ngModelCtrl.$render = function() {
        $element.html($sce.getTrustedHtml(ngModelCtrl.$viewValue || ''));
    };

    // [3] Read HTML value, and then write data to the model or reset the view
    var read = function () {
        var value = $element.html();
        if ($attrs.type === 'number' && !NUMBER_REGEXP.test(value)) {
            ngModelCtrl.$render();
        } else {
            ngModelCtrl.$setViewValue(value);
        }
    };

    // [4] Add custom parser-based input type (only 'number' supported)
    // This will be applied to the $modelValue
    if ($attrs.type === 'number') {
        ngModelCtrl.$parsers.push(function (value) {
            return parseFloat(value);
        });
    }

    // [5] Listen for change events to enable binding
    $element.on('blur keyup change', function() {
        $scope.$apply(read);
    });
}
};
});
```

As with the `stkStockRow` directive, the `require` property is once again used to grab a handle on an external directive's controller. In this case, `ngModel` is being required [1] because you want to take advantage of Angular's bidirectional data binding to trigger updates to the rest of the table based on the user's modification. Next, the `ngModelCtrl.$render()` function is implemented, which is required to inform the `ngModel` directive how the view should be updated. Here, the Strict Contextual Escaping service `$sce` is used, which was the only injected dependency, to sanitize user input before updating the view's HTML [2]. A `read()` function is then defined that inspects the element's current HTML value and, if its `type` property is set to `number`, tests to see if the value is a number using a regular expression [3]. In this case, your `contenteditable` directive is only used for the Shares Owned cell, so only a number type is supported, but you can easily extend this functionality to support other input types and formats. If the current value is not a number, the `ngModelCtrl.$render()` function is called, which updates the view with the previous value. However, if the user does in fact input a valid number, the directive calls `ngModelCtrl.$setViewValue()`, which handles invoking `$render()` with the new value and kicks off the `ngModel $parsers` pipeline. A custom parser is defined [4] to support number input types. It parses the `$viewValue` into a number so that `ngModel` can update the `$modelValue`,

which can then be properly used to recalculate values for the stock table. Finally, the `$element.on()` function is used to listen for the `blur`, `keyup`, and `change` events so that `read()` can be invoked after each modification [5].

Updating the StkStockTable Template

All that is left to do is update the `stock-table.html` file located in the `app/views/templates` directory to utilize this newly created `contenteditable` directive. Find the line containing `<td>{{stock.shares}}</td>` and replace it entirely with the following:

```
<td contenteditable type="number" ng-model="stock.shares"></td>
```

Notice that the `type` attribute is set to `number`, and `ng-model` is used to bind to the `shares` value of the row's stock object. Because it might not be explicitly clear to your users that you can perform inline edits on the Shares Owned cell, add the following line to the bottom of your `stock-table.html` file:

```
<div class="small text-center">Click on Shares Owned cell to edit.</div>
```

With these two quick modifications complete, take a moment to test out the inline editing functionality, an example of which is shown in Figure 1-11. Attempting to type any characters other than a number into a row's Shares Owned cell immediately resets the value. However, after each successful modification that results in a valid number, the entire stock table is recalculated in real time. If your application is not functioning properly, please refer to the `step-8` directory inside the companion code for this chapter or check out the corresponding tag of the GitHub repository.

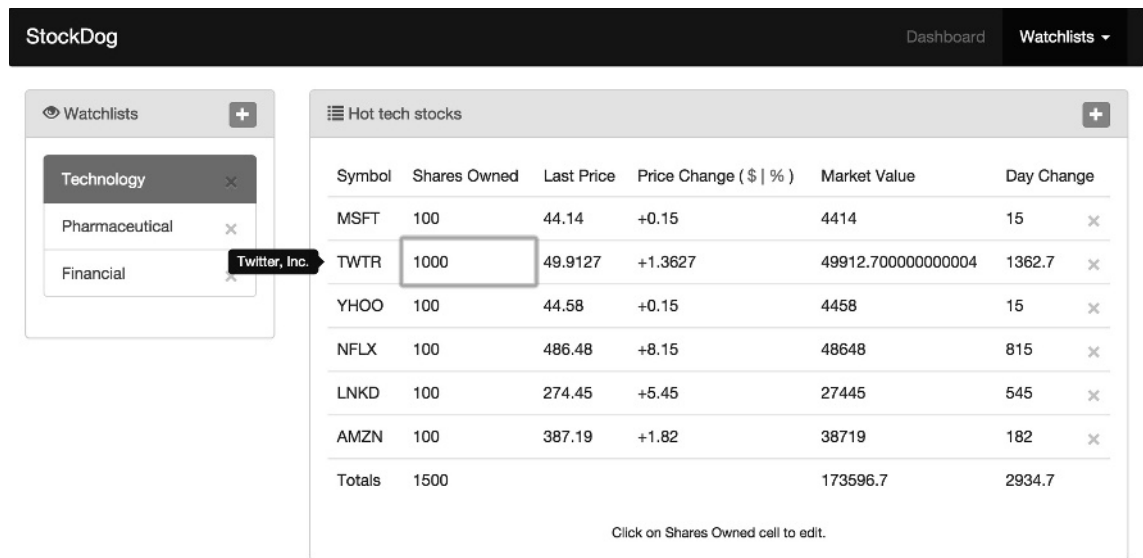


FIGURE 1-11

STEP 9: FORMATTING CURRENCY

At this point, StockDog's watchlist view is fully functional. Watchlists can be created, and stocks can be added, deleted, and edited using the stock table, but the way the data is being displayed isn't ideal. In this section you will be formatting the displayed numbers using Angular's built-in currency filter, in addition to creating a new directive that changes the number's color based on whether it is reflecting a positive or negative change.

Creating the StkSignColor Directive

The first order of business is to create a new `stkSignColor` directive that you can apply to existing elements to modify their displayed color to be either red or green. Go ahead and run the following command from your terminal to scaffold out this directive:

```
yo angular:directive stk-sign-color
```

This creates a new file named `stk-sign-color.js` inside your `app/scripts/directives/` directory. You can see the full implementation of the `stkSignColor` directive in Listing 1-18. The first thing you may notice is that instead of a `$scope.$watch()`, an `$attrs.$observe()` was used to listen to changes in the expression assigned to `stkSignColor` [1]. Because `$observe()` is a function of the `$attrs` object, it can only be used to observe/watch the value change of a DOM attribute, which in this case is exactly what you want. The rest of this directive is incredibly simple because all it has to do is update the `$element`'s `style.color` property depending on whether the expression's new value is positive or negative [2].

LISTING 1-18: `app/scripts/directives/stk-sign-color.js`

```
'use strict';

angular.module('stockDogApp')
  .directive('stkSignColor', function () {
    return {
      restrict: 'A',
      link: function ($scope, $element, $attrs) {
        // [1] Use $observe to watch expression for changes
        $attrs.$observe('stkSignColor', function (newVal) {
          var newSign = parseFloat(newVal);
          // [2] Set element's style.color value depending on sign
          if (newSign > 0) {
            $element[0].style.color = 'Green';
          } else {
            $element[0].style.color = 'Red';
          }
        });
      }
    };
  });
```

Updating the StockTable Template

In addition to adding the `stkSignColor` directive to your `stock-table.html` template, you need to use Angular's built-in currency filter. Although an in-depth discussion of Angular filters is outside

the scope of this chapter, all you need to know to move forward is that a filter formats the value of an expression for display to the user. A filter can be used in view templates, controllers, and services, and it is fairly straightforward to create your own custom filter. You can apply a filter to an expression in a view template using this syntax: `{{ expression | filter }}`. To learn more about what filters are available out of the box, visit the official documentation located at <https://docs.angularjs.org/api/ng/filter>. For this section, you will be using the currency filter, with default parameters. The full syntax for the currency filter is as follows:

```
{{ currency_expression | currency : symbol : fractionSize }}
```

Because the `symbol` defaults to `$` and the `fractionSize` to the current locale's max fraction size, using the currency filter is almost trivial. Go ahead and add `| currency` to the `watchlist.marketValue`, `watchlist.dayChange`, `stock.lastPrice`, `stock.marketValue`, and `stock.dayChange` expression bindings. Then you'll want to add the `stk-sign-color` attribute, with a binding to a value that should be watched for changes, to each `<td>` element that you want to color. In this case, you'll want to color the `watchlist.dayChange` cell in the footer, as well as the Price Change and Day Change columns in the table. Here is an example of applying the `stk-sign-color` directive to the `watchlist.dayChange` row in the footer:

```
<td stk-sign-color="{{watchlist.dayChange}}">
  {{watchlist.dayChange | currency}}
</td>
```

The application of the `stk-sign-color` directive to the remaining two cells is left as an exercise for you, the reader. Once the currency filters and `stk-sign-color` directives are properly in place, your application should look something like Figure 1-12. If you find yourself struggling with applying the directive and currency filters in the correct location of the markup, please refer to the `step-9` directory inside the companion code for this chapter or check out the corresponding tag of the GitHub repository.

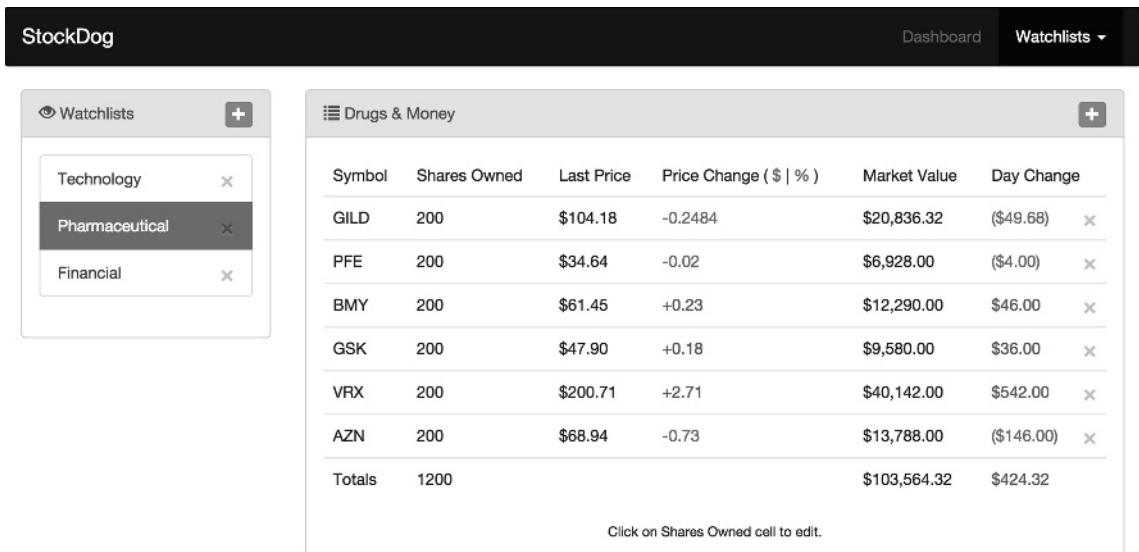


FIGURE 1-12

STEP 10: ANIMATING PRICE CHANGES

In this section, you will learn the basics of how to use Angular's `ngAnimate` module to perform an animation on StockDog's watchlist view. To visually show your users the price action of a given stock—that is, whether there has been a positive or a negative change in value—a red or green crossfade on the entire cell is performed. Although a complete discussion on creating JavaScript and CSS3 animations with Angular is outside the scope of this chapter, you can find more information by visiting the official documentation at <https://docs.angularjs.org/api/ngAnimate>.

Creating the `StkSignFade` Directive

Because the desired result is to crossfade an entire table cell, you need to create another directive that will be used as an attribute so that it can be dropped onto existing elements. To get started, run the following command from your terminal:

```
yo angular:directive stk-sign-fade
```

This creates a new `stk-sign-fade.js` file inside your `app/scripts/directives/` directory. Just as with the `stkSignColor` directive you created in the previous section, this directive will be fairly short and straightforward. You can find the complete implementation of the `stkSignFade` directive in Listing 1-19.

LISTING 1-19: `app/scripts/directives/stk-sign-fade.js`

```
'use strict';

angular.module('stockDogApp')
  .directive('stkSignFade', function ($animate) {
    return {
      restrict: 'A',
      link: function ($scope, $element, $attrs) {
        var oldVal = null;
        // [1] Use $observe to be notified on value changes
        $attrs.$observe('stkSignFade', function (newVal) {
          if (oldVal && oldVal == newVal) { return; }

          var oldPrice = parseFloat(oldVal);
          var newPrice = parseFloat(newVal);
          oldVal = newVal;

          // [2] Add the appropriate direction class, and then remove it
          if (oldPrice && newPrice) {
            var direction = newPrice - oldPrice >= 0 ? 'up' : 'down';
            $animate.addClass($element, 'change-' + direction, function() {
              $animate.removeClass($element, 'change-' + direction);
            });
          }
        });
      }
    };
  });
```

The only dependency that was injected into this directive was the `$animate` service, which is provided by the `ngAnimate` module. As you saw with the `stkSignColor` directive, `$attrs.$observe()` is once again used to watch for changes to the expression assigned to `stkSignFade` [1]. A local reference is kept to the `oldVal` so that on subsequent changes, it can be compared against the `newVal` and the appropriate direction class can be computed [2]. For this example, the `$animate` service is used to add, and then quickly remove, the `change-up` or `change-down` CSS classes from the directive's element. The `$animate` service takes an element, class name, and callback function as a parameter, which is used to remove the class after the animation for adding it has been performed. Before attempting to use this directive in the `stock-table.html` file, you must create a handful of CSS classes using the syntax that Angular requires. Add the following lines of code to the top of your `app/styles/main.css` file. A few other styles that polish up the stock table's display are also included here:

```
/* Stock Table Styles */
.table {
  text-align: center;
  margin-bottom: 5px;
}
tfoot {
  font-weight: bold;
}
a {
  cursor: pointer;
}
span[disabled="disabled"] a {
  text-decoration: none;
  color: black;
}
span[disabled="disabled"] {
  pointer-events: none;
}

/* Styles for ngAnimate animations */
.change-up-add {
  transition: background-color linear 1.5s;
  background-color: green;
}
.change-up-add.change-up-add-active {
  background-color: white;
}
.change-down-add {
  transition: background-color linear 1.5s;
  background-color: red;
}
.change-down-add.change-down-add-active {
  background-color: white;
}
```

Angular expects you to define `*-add` and `*-add-active` classes for each of your desired animation classes. In the preceding example, `change-up-add` is applied immediately, which sets the background to green. Then the `change-up-add-active` class is applied for the duration of the animation. In this case, that sets the background color to white with a 1.5s CSS transition,

ultimately creating a crossfade effect from green to white. The same approach is used for `change-down-add`, which shows a negative price action in red.

Updating the StockTable Template

Now that you have completed the `stkSignFade` directive and created the appropriate CSS classes expected by the `ngAnimate` module, it is time to modify your `stock-table.html` view template. Locate the two lines with `<td>` elements that are displaying the `watchlist.marketValue` and `stock.lastPrice`, and add the `stk-sign-fade="{{watchlist.marketValue}}"` and `stk-sign-fade="{{stock.lastPrice}}"` directive to them, respectively.

NOTE *Because the `QuoteService` is updating the `stock.lastPrice` as it fetches data from Yahoo Finance, you may run into a situation in which the market is closed and the price isn't changing, making it difficult to see your new `stkSignFade` directive in action. In this case, modify the `update()` function inside your `quote-service.js` file to randomize the `stock.lastPrice`. You can accomplish this with `Lodash` by adding `+ _.random(-0.5, 0.5)` to the line that parses the `quote.LastTradePriceOnly`. Just don't forget to remove it when you've finished testing!*

Congratulations! You have completely finished StockDog's watchlist view! See Figure 1-13. If you find yourself struggling with getting your animations to properly run, please refer to the `step-10` directory inside the companion code for this chapter or check out the corresponding tag of the GitHub repository.

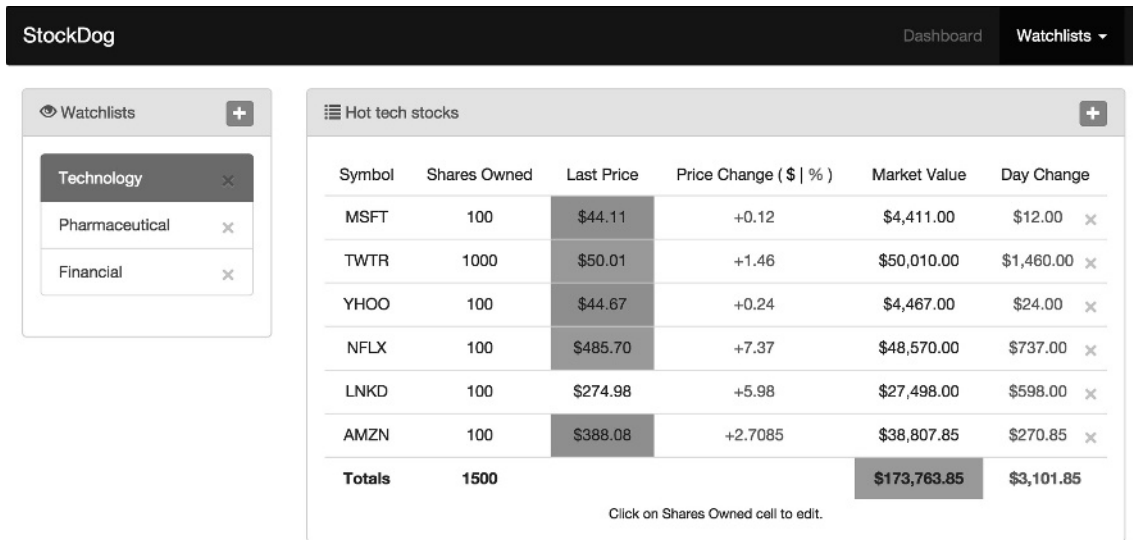


FIGURE 1-13

STEP 11: CREATING THE DASHBOARD

The final outstanding feature that remains to be implemented for the StockDog application is the dashboard view. This view aggregates performance metrics across all created watchlists and reports the analytics in four unique panels. These performance metrics are Total Market Value, Total Day Change, Market Value by Watchlist, and Day Change by Watchlist. Because no dashboard is complete without interactive graphs, you will be taking advantage of the Google Charts library to render two distinct charts.

Updating the Dashboard Controller

To use the Google Charts library from within your AngularJS application, you need to wrap and expose its functionality via directives. For the sake of simplicity, you will be using a preexisting library that has done just that, whose documentation can be found here: <https://github.com/bouil/angular-google-chart>. To get started with the `angular-google-chart` library, run the following command from your terminal to install it using Bower:

```
bower install angular-google-chart --save
```

This downloads and installs the library. It also lists it as a project dependency inside your `bower.json` file. Once that is complete, you must register this library's module with your AngularJS application by updating your `stockDogApp` module dependencies. You can do this by adding `googlechart` to the end of the dependencies array found in your `app/scripts/app.js` file, in the same manner in which the `AngularStrap` library was registered back in Listing 1-1 of Step 2 earlier in this chapter. Once that is complete, open the `dashboard.js` file located in your `app/scripts/controllers/` directory and replace its contents with the final implementation shown in Listing 1-20.

LISTING 1-20: `app/scripts/controllers/dashboard.js`

```
'use strict';

angular.module('stockDogApp')
  .controller('DashboardCtrl', function ($scope, WatchlistService, QuoteService) {
    // [1] Initializations
    var unregisterHandlers = [];
    $scope.watchlists = WatchlistService.query();
    $scope.cssStyle = 'height:300px;';
    var formatters = {
      number: [
        {
          columnNum: 1,
          prefix: '$'
        }
      ]
    }
  }
};

// [2] Helper: Update chart objects
var updateCharts = function () {
  // Donut chart
```

continues

LISTING 1-20 *(continued)*

```
var donutChart = {
  type: 'PieChart',
  displayed: true,
  data: [['Watchlist', 'Market Value']],
  options: {
    title: 'Market Value by Watchlist',
    legend: 'none',
    pieHole: 0.4
  },
  formatters: formatters
};
// Column chart
var columnChart = {
  type: 'ColumnChart',
  displayed: true,
  data: [['Watchlist', 'Change', { role: 'style' }]],
  options: {
    title: 'Day Change by Watchlist',
    legend: 'none',
    animation: {
      duration: 1500,
      easing: 'linear'
    }
  },
  formatters: formatters
};

// [3] Push data onto both chart objects
_.each($scope.watchlists, function (watchlist) {
  donutChart.data.push([watchlist.name, watchlist.marketValue]);
  columnChart.data.push([watchlist.name, watchlist.dayChange,
    watchlist.dayChange < 0 ? 'Red' : 'Green']);
});
$scope.donutChart = donutChart;
$scope.columnChart = columnChart;
};

// [4] Helper function for resetting controller state
var reset = function () {
  // [5] Clear QuoteService before registering new stocks
  QuoteService.clear();
  _.each($scope.watchlists, function (watchlist) {
    _.each(watchlist.stocks, function (stock) {
      QuoteService.register(stock);
    });
  });
};

// [6] Unregister existing $watch listeners before creating new ones
_.each(unregisterHandlers, function(unregister) {
  unregister();
});
_.each($scope.watchlists, function (watchlist) {
```

```

        var unregister = $scope.$watch(function () {
            return watchlist.marketValue;
        }, function () {
            recalculate();
        });
        unregisterHandlers.push(unregister);
    });
};

// [7] Compute the new total MarketValue and DayChange
var recalculate = function () {
    $scope.marketValue = 0;
    $scope.dayChange = 0;
    _.each($scope.watchlists, function (watchlist) {
        $scope.marketValue += watchlist.marketValue ?
            watchlist.marketValue : 0;
        $scope.dayChange += watchlist.dayChange ?
            watchlist.dayChange : 0;
    });
    updateCharts();
};

// [8] Watch for changes to watchlists.
$scope.$watch('watchlists.length', function () {
    reset();
});
});
};

```

For the implementation of this `DashboardCtrl`, both `WatchlistService` and `QuoteService` are injected as dependencies. Next, some initializations are made to populate the `$scope.watchlists` variable using the `WatchlistService`, with chart style and formatting options also being defined [1]. An `updateCharts()` function is then created [2] that sets up both a `donutChart` and a `columnChart`. The required properties and available configuration options for these objects are defined by the Google Chart library documentation, which can be found here <https://developers.google.com/chart/>. This function also handles looping over each watchlist being tracked by `StockDog` and adding the appropriate data onto the respective chart object [3] before attaching both chart structures to the controller's `$scope`. A `reset()` function [4] is then defined that is used to clear the controller's state. This function clears all tracked stocks from the `QuoteService` before registering each stock for each existing watchlist [5]. It then unregisters all existing `$watch` listeners, whose references are stored in a local array, before creating new `$watch` targets on each watchlist's `marketValue` [6]. This is used to invoke the `recalculate()` function [7], which handles computing new aggregate market value and day change metrics each time a watchlist's computed value changes.

Each time `recalculate` is invoked, a call to `updateCharts()` is made so that the existing charts can be redrawn by the Google Chart library with the newest data. Finally, a `$watch` target is set on the `watchlists.length` property so that when a watchlist is created or deleted, the `reset()` function can be triggered to appropriately rebuild the entire controller's state [8]. It's worth mentioning that the `watchlists.length` expression is used instead of the entire `watchlists` object because deep-watching large data structures can seriously degrade your application's performance.

Updating the Dashboard View

Now that the `DashboardCtrl` implementation is complete, the next order of business is to update `StockDog`'s dashboard view to render the new data and chart objects that have been created. As it stands, the `app/views/dashboard.html` file only contains a reference to the `stkWatchlistPanel` directive and an empty `Portfolio Overview` panel. You can find the missing markup for this panel in the completed dashboard view, shown in Listing 1-21.

LISTING 1-21: `app/views/dashboard.html`

```
<div class="row">
  <!-- Left Column -->
  <div class="col-md-3">
    <stk-watchlist-panel></stk-watchlist-panel>
  </div>

  <!-- Right Column -->
  <div class="col-md-9">
    <div class="panel panel-info">
      <div class="panel-heading">
        <span class="glyphicon glyphicon-globe"></span>
        Portfolio Overview
      </div>
      <div class="panel-body">
        <!-- [1] Display some helpful text to guide new users -->
        <div ng-hide="watchlists.length && watchlists[0].stocks.length"
          class="jumbotron">
          <h1>Unleash the hounds!</h1>
          <p>
            StockDog, your personal investment watchdog, is ready
            to be set loose on the financial markets!
          </p>
          <p>Create a watchlist and add some stocks to begin monitoring.</p>
        </div>

        <div ng-show="watchlists.length && watchlists[0].stocks.length">
          <!-- Top Row -->
          <div class="row">
            <!-- Left Column -->
            <div class="col-md-6">
              <!-- [2] Use sign-fade directive on wrapper element -->
              <div stk-sign-fade="{{marketValue}}" class="well">
                <h2>{{marketValue | currency}}</h2>
                <h5>Total Market Value</h5>
              </div>
            </div>

            <!-- Right Column -->
            <div class="col-md-6">
              <!-- [3] Use sign-color directive on wrapper element -->
              <div class="well" stk-sign-color="{{dayChange}}">
                <h2>{{dayChange | currency}}</h2>
                <h5>Total Day Change</h5>
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

```

        </div>
    </div>
    <!-- [4] Use google-chart directive and reference chart objects -->
    <div class="row">
        <!-- Left Column -->
        <div class="col-md-6">
            <div google-chart chart="donutChart" style="{{cssStyle}}"></div>
        </div>

        <!-- Right Column -->
        <div class="col-md-6">
            <div google-chart chart="columnChart" style="{{cssStyle}}"></div>
        </div>
    </div>
</div>
</div>
</div>
</div>
</div>

```

The new markup inside the `panel-body` starts by including some helpful text to guide new users when they first open StockDog and have yet to create any watchlists [1]. You should also notice that both columns of the top row contain references to the `stkSignFade` [2] and `stkSignColor` [3] directives, but the directives have been applied to a wrapper element—in this case, Bootstrap wells. Finally, the `googleChart` directive, exposed by the previously installed `angular-google-chart` library, is used in both columns of the bottom row, with the chart objects created in the `DashboardCtrl` being used as the value for each respective element's `chart` attribute [4]. To polish up the completed dashboard view, the only remaining modification you'll need to make is to add the following CSS to the top of your `app/styles/main.css` file:

```

/* Dashboard View Styles */
.well {
    background-color: white;
    text-align: center;
}

```

Congratulations! If you have successfully made it through the entirety of this section, you have finally finished building the entire StockDog application! See Figure 1-14. Take a moment to appreciate your hard work and play around with the application by creating several new watchlists, adding new stocks, and monitoring your portfolio's performance from the dashboard view. For the completed application source code, please refer to the `step-11` directory inside the companion code for this chapter or check out the corresponding tag of the GitHub repository.

PRODUCTION DEPLOYMENT

Now that you have finished building StockDog, the time has come to unleash the hounds and package the distributable application before deploying it to the Internet so that your users around the world can better manage their stock portfolios. Although an in-depth discussion of production deployment and all the associated intricacies is outside the scope of this section, there are a few simple tasks that can be accomplished to get your application ready for the masses.

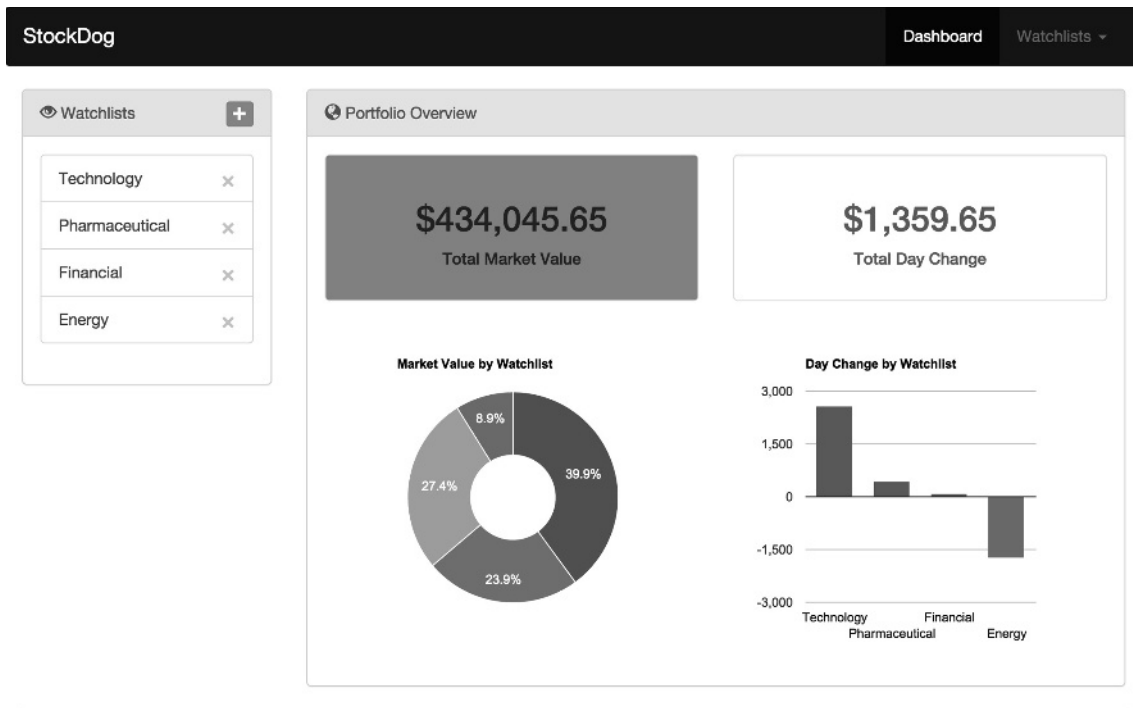


FIGURE 1-14

Because your application was developed using the AngularJS Yeoman generator, your project already includes a sophisticated build system. You will learn more about how this system works in the Chapter 3, “Architecture,” but for now, just run the following command from your terminal to run the build system:

```
grunt build
```

This concatenates, obfuscates, and minifies all of StockDog’s source files and creates a new `dist/` directory in your project’s root folder with the optimized assets. The `dist/` directory contains everything needed for users to run your application, so deployment is as simple as uploading this folder to your hosting service of choice. However, for the purpose of this section, you will be deploying StockDog to GitHub Pages, a hosting service provided free of charge for GitHub-based projects. If you haven’t already uploaded your project to GitHub, take a few minutes to do so, consulting <https://help.github.com/articles/adding-an-existing-project-to-github-using-the-command-line/> if you need any further assistance.

Once your project has been uploaded to GitHub, open your `.gitignore` file and remove the line containing `dist`. Out of the box, Yeoman has set up your project to follow best practices by ignoring files generated by the automated build task. However, because you will be hosting your `dist/` directory on GitHub, it must be committed as part of your project. Go ahead and add the

`dist/` directory to your repository, commit, and then push it upstream. Now you are ready to deploy your application to GitHub using the `git subtree` command. Run the following command from your terminal to create a new `gh-pages` branch for your project consisting of all the files residing inside your `dist/` directory:

```
git subtree push --prefix dist origin gh-pages
```

Once that is complete, your application will be publicly available at `http(s)://<username>.github.io/<projectname>`. For example, you can find the StockDog application running at `http://diegonetto.github.io/stock-dog`. One caveat to this approach is that your Dashboard and Watchlist links must be prefixed with your `<projectname>` because of the nature of the GitHub Pages URL. Another approach is to set up a custom URL for your project by uploading a new CNAME file to your `dist/` directory that contains your custom domain. This is how `http://www.stockdog.io/` has been set up to point to `http://diegonetto.github.io/stock-dog`. After uploading your CNAME file and redeploying your site using the `git subtree` command shown earlier, all that is left is to modify the `www` CNAME record (assuming you want to use the `www` subdomain) of your DNS provider to point to `username.github.io`. If you have successfully followed these steps, congratulations! Your application should be live and ready to share with the rest of the world.

NOTE *GitHub recommends using a subdomain and not an apex domain when configuring custom URLs for hosted project pages. If you wish to use your apex domain (`http://stockdog.io` in the above example) for your deployed application, the best way to accomplish this is to use the `www` subdomain CNAME DNS entry with your provider as described and then enable domain forwarding from your apex domain to your `www` URL. In this case, `http://stockdog.io` has been set up to forward to `http://www.stockdog.io`.*

CONCLUSION

The journey through this chapter has exposed you to a real-world application of AngularJS by building StockDog, an application that leverages nearly all-key components of the framework. From scaffolding a starter project using the Yeoman AngularJS generator to deploying your application using GitHub Pages, this step-by-step guide should have given you the confidence and instant gratification to inspire a deeper dive into this elegant framework. Along the way, you learned how to structure a multiview single-page application; created several controllers, directives, and services; installed additional front-end modules; handled dynamic form validation; communicated with an external API; and brought your application to life with a simple animation. In the following chapters of this book, you will explore in detail how the various components of the AngularJS framework function and be exposed to various tools, services, and technologies that can be used to create robust, reliable, and maintainable projects for professional consumption.

