Learn With

Knowledge to Boost Your Career

AngularJS, Bootstrap, & ColdFusion

WWW.LEARN-WITH.COM

Part of the DotComIt Brain Trust

Learn With
AngularJS, Bootstrap, and ColdFusion

By Jeffry Houser

http://www.learn-with.com

http://www.jeffryhouser.com

About the Author

Jeffry Houser is a technical entrepreneur that likes to share cool stuff with other people.

In the days before business met the Internet, Jeffry obtained a Computer Science degree. He has solved a problem or two in his programming career. In 1999, Jeffry started DotComIt; a company specializing in custom application development.

During the Y2K era, Jeffry wrote three books for Osborne McGraw-Hill. He is a Member of the project management committee for Apache Flex, and created Flextras; a library of Open Source Flex Components. Jeffry has spoken all over the US. He has produced hundreds of podcasts, written over 30 articles, and written a slew of blog posts.

In 2014, Jeffry created Life After Flex; an AngularJS training course for Flex Developers. Since then, he has worked with multiple clients building AngularJS applications.

Preface

I was a Flex developer for what seems like a long time. However, times have changed and Adobe’s Flash Platform is not as relevant as it once was. A smart developer will spend time to educate himself on new technologies to continue to have healthy job prospects. While educating myself, I decided to write about my experiences. With this series of books, you can leverage my experience to learn quickly.

This book is about my experiences building AngularJS 1.x applications. AngularJS is a JavaScript framework built by Google. It has data binding features to build dynamic views in HTML5. It is fully testable, which is important to many enterprise-level applications. It has good documentation and a large developer community, ready to help with problems. I like AngularJS.

This book will help you learn how to build a Task Manager application using AngularJS and Bootstrap. It will show you how to use real services built with ColdFusion and a SQL Server database. I have worked with ColdFusion for over 15 years, and it is a simple way to build dynamic web pages.

Introduction

What is this Book Series About?

The purpose of this series is to teach by example. The plan is to build an application using multiple technologies. These books will document the process; each book focusing on a specific technology or framework. This entry will focus on AngularJS as the application framework, and Bootstrap as the UI component library.

The application built in this book will focus on common functionality required when I build applications for enterprise consulting clients. You'll receive step-by-step instructions on how to build a task manager application. It will integrate with a service layer. A login will be required. Functionality will be turned off or on based on the user's role. Data will be displayed in a DataGrid, because all my enterprise clients love DataGrids. Common tasks will be implemented for creating, retrieving, and updating data.

Who Is This Book For?

Want to learn about building HTML5 applications? Are you interested in AngularJS or Bootstrap? Do you want to learn new technologies by following detailed examples with runnable code? If you answer yes to any of these questions, then this book is for you!

Here are some topics we'll touch on in this book, and what you should know before continuing:

- **JavaScript**: You should have some general knowledge of JavaScript. I won't be detailing the language's syntax. If you haven't any, don't worry. I try to keep things simple.
- **AngularJS**: The primary focus of this book is on AngularJS, so we are going assuming you have no experience with it.
- **JSON**: The data returned from the services will be done so as JSON packets. Even if you have no experience with JSON, it should be very easy to understand. This book will provide a basic primer when the topic comes up.
- **Bootstrap**: This is a CSS framework that helps create things such as popups and date choosers. We'll use Bootstrap in conjunction with AngularJS to help flesh out the application's user interface.
- **ColdFusion**: The services for this book are built using CFML; the language of ColdFusion. This book only requires basic knowledge of ColdFusion. If you're adventurous, you should be able to rework the services of this app in a technology of your choice—and I'd love for you to share that code if you do so.
- **SQL**: A SQL Server database is used as the storage mechanism for this book. As such, the SQL language will be used to communicate with the database from ColdFusion. There aren't any advanced SQL concepts in this book, but you'll get the most of this if you have a general understanding of SQL.

How to Read This Book

Each chapter of this book represents one aspect of the application's user interface, such as logging in, or editing a task. The chapters are split up into four parts each:

- **Building the UI**: This section will cover the creation of the UI elements of the chapter.
- **The Database**: Each chapter will have a section to review the data that the chapter's functionality deals with. The database storage tables will be examined, as well as an explanation of the data types.
- **The Services**: This section will cover the APIs of the services that need to be interacted with. This book will create services using ColdFusion. If you're feeling adventurous, you should be able to build out the services to any language of your choice.
- **Connecting the UI to the Services**: This section will cover how the UI code will call the services, and handle the results.

Common Conventions

I use some common conventions in the code behind this book.

- **Classes**: Class names are in proper case; the first character of the class in uppercase, and the start of each new compound word being in uppercase. An example of a class name may be **MyClass**. When referencing class names in the book text, the file extension is usually referenced. For JavaScript files that contain classes the extension will be JS.
- **Variables**: Variable names are also in proper case, except the first letter is always lowercase. This includes class properties, private variables, and method arguments. A sample property name may be **myProperty**.
- **Constants**: Constants are in all uppercase, with each word separated by an underscore. A sample constant may be **MY_CONSTANT**. JavaScript doesn't have a universal implementation of constants, but if we create a variable intended to be a constant this is the convention that will be used.
- **Method or Function Names**: Method names use the same convention as property names; the first character in lower-case, and then proper case is used. When methods are referenced in text, open and close parentheses are put after the name. A sample method name may be **myMethodName()**.
- **Package or Folder Names**: The package names— or folders— are named using proper case, where the first letter of the package is in lowercase and the first letter of any following words are uppercase. All other letters are in lowercase. In this text, package names are always referenced as if they were a directory relative to the application root. A sample package name may be **com/dotComIt/learnwith/myPackage**.

Caveats

The goal of this book is to help you become productive creating HTML5 apps with a focus on AngularJS. It leverages my experience building business apps, but is not intended to cover everything you need to know about building HTML Applications. This book purposely focuses on the AngularJS framework, not the tool chain. You should approach this book as part of your learning process and not the last thing you'll ever need to know. Be sure that you keep learning. I know I will.

Want More?

You should check out this book's web site at www.learn-with.com for more information, such as:

- **Source Code**: You can find links to all the source code for this book and others.
- **Errata**: If we make mistakes, we plan on fixing them. You can always get the most up-to-date content available from the website. If you find mistakes, please let us know.
- **Test the Apps**: The web site will have runnable versions of the app for you to test.
- **Bonus Content**: You can find more articles and books expanding on the content of this book.

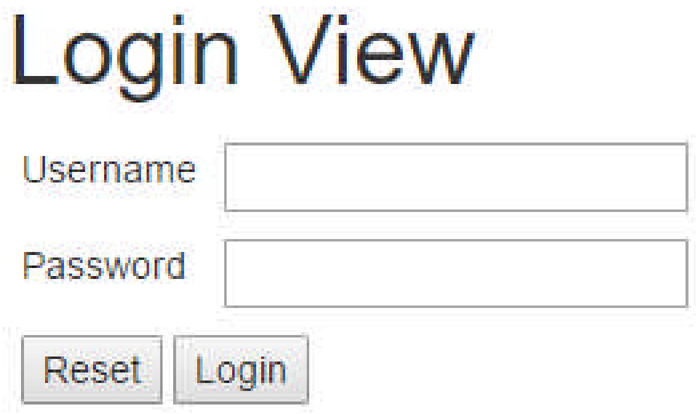
Chapter 1: The Application Overview and Setup

This chapter will examine the full scope of the application this book builds. It will flesh out the application infrastructure. Each subsequent chapter will dive deeper into one piece of specific functionality.

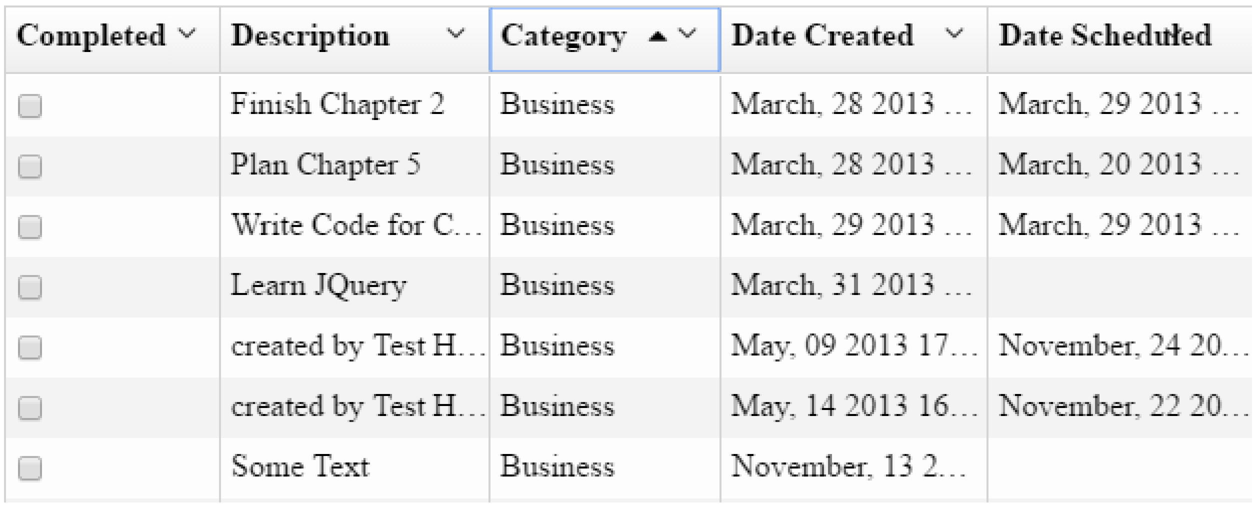
Introducing the Task Manager Application

This book will build a Task Manager application. It will start at ground zero, and create a finished application. The application will include these functionalities:

- A Login Screen so that different users, or types of users, can have access to the applications functionality and data.

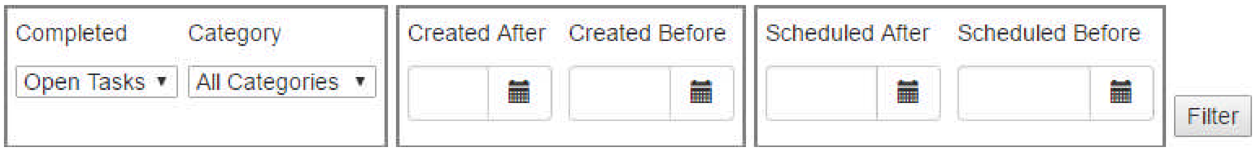


- The ability to load tasks and display them to the user.

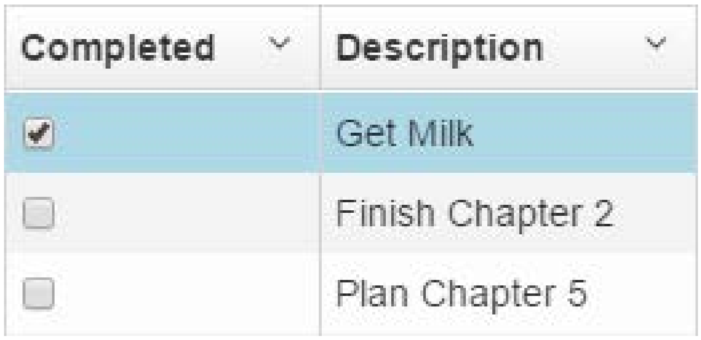


Completed	Description	Category	Date Created	Date Scheduled
☐	Finish Chapter 2	Business	March, 28 2013 …	March, 29 2013 …
☐	Plan Chapter 5	Business	March, 28 2013 …	March, 20 2013 …
☐	Write Code for C…	Business	March, 29 2013 …	March, 29 2013 …
☐	Learn JQuery	Business	March, 31 2013 …	
☐	created by Test H…	Business	May, 09 2013 17…	November, 24 20…
☐	created by Test H…	Business	May, 14 2013 16…	November, 22 20…
☐	Some Text	Business	November, 13 2…	

- The ability to filter tasks so that only a subset of the tasks will be shown in the UI; such as all tasks scheduled on a certain day.

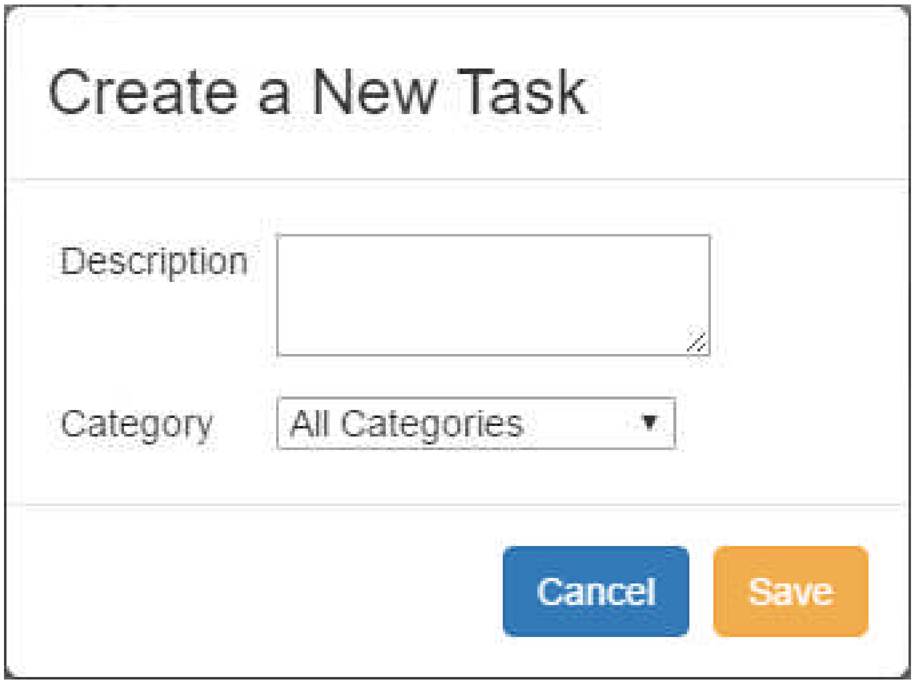


- The ability to mark a task completed.

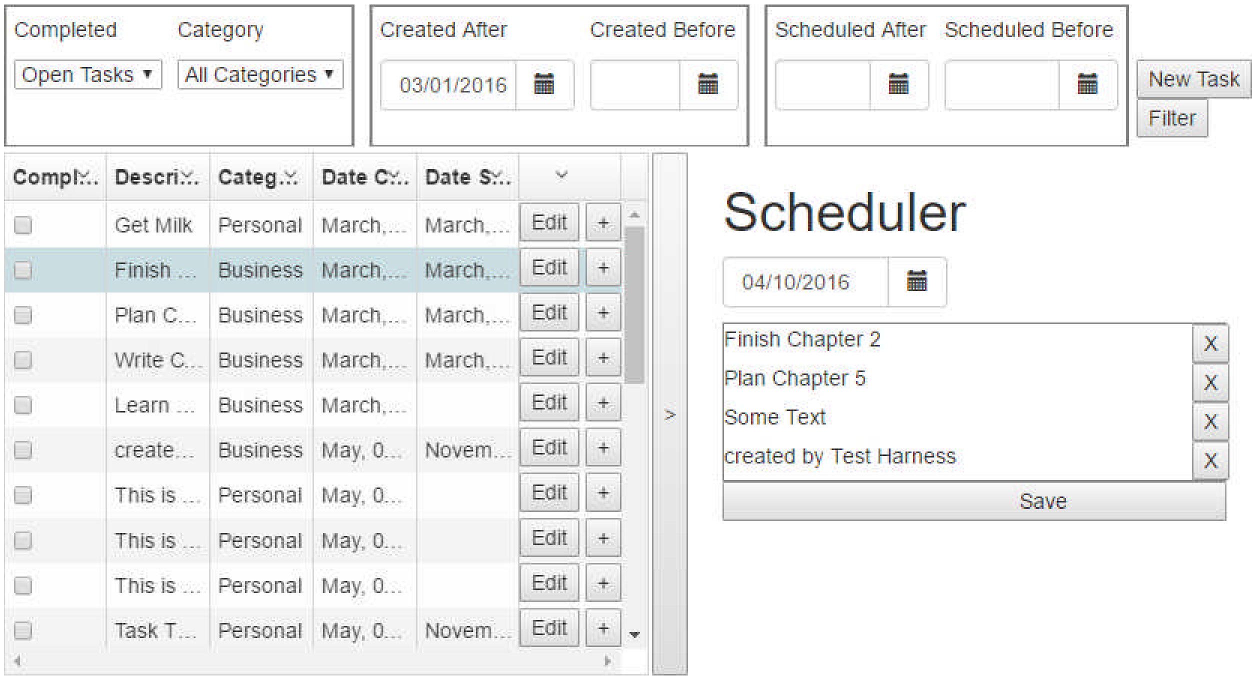


Completed	Description
✓	Get Milk
☐	Finish Chapter 2
☐	Plan Chapter 5

- The ability to create or edit tasks.



- The ability to schedule a task for a specific day.



Completed	Category	Created After	Created Before	Scheduled After	Scheduled Before
Open Tasks	All Categories	03/01/2016			

Compl...	Descri...	Categ...	Date C...	Date S...
	Get Milk	Personal	March,...	March,...
	Finish ...	Business	March,...	March,...
	Plan C...	Business	March,...	March,...
	Write C...	Business	March,...	March,...
	Learn ...	Business	March,...	
	create...	Business	May, 0...	Novem...
	This is ...	Personal	May, 0...	
	This is ...	Personal	May, 0...	
	This is ...	Personal	May, 0...	
	Task T...	Personal	May, 0...	Novem...

Scheduler

04/10/2016

Finish Chapter 2
Plan Chapter 5
Some Text
created by Test Harness

Each chapter of this book will focus on a different aspect of the application.

Create the Application Skeleton

This section will create the basic application skeleton of the AngularJS application. It will discuss the use of Angular's **$routeProvider** service in order to change the application's current view.

Start with a Basic HTML Page

Start by creating a simple HTML page:

```
<html>
<head>
</head>
<body>
</body>
</html>
```

I named this page **index.html**. This page doesn't display anything yet, but is about as simple as they come.

In the body of the document, you should import the AngularJS library:

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.3/angular.min.js">
</script>
```

If you want a local version of the AngularJS library you can download it from the Angular web site. Instead of copying the library local I used a version available on the Google Content Delivery Network (CDN). When building applications, it is considered a best practice to use libraries from a CDN because users of your application may already have it cached. That is one less item your users need to download.

Define the AngularJS Application

The next step is to define the Angular application within the context of the HTML page. Angular works by using directives that extend normal HTML elements. The **ngApp** directive defines the Angular application. We'll add it to the body tag of the **index.html**:

```
<body ng-app="learnWith">
</body>
```

The value of the **ngApp** directive is **learnWith**. This code tells AngularJS that the application named **learnWith** has access to the contents of the body tag.

If you load the page as is, you'll see an error in your web browser's console. I recommend using Chrome for its robust Developer tools. The error happens because we have told Angular to look for the learnWith app, but we have not defined the app yet. When I built my first Angular application I had to determine how much code should be kept in each file. I decided I would prefer to have lots of small files with discrete code chunks over a smaller number of comprehensive files.

The first file we're going to make is named **LearnWithApp.js**. Put this file in the **com/dotComIt/learnWith/app** directory. This code will define the application:

```
angular.module('learnWith', []);
```

The line of code calls the **module()** function in the Angular library and creates a module named "learnWith". The first parameter of the **module()** function is the name of the Angular application. The second parameter is an array of options available to the new module. Options are other Angular libraries. At this point, no options are being used. This module name matches the name of the application referenced in the body tag.

The **LearnWith.js** file was created in the **com/dotComIt/learnWith/app** directory. Go back to your **index.html** file and load the library in the file header using a script tag:

```
<script src="com/dotComIt/learnWith/app/LearnWith.js"></script>
```

This script tag should be added after the script tag which loads the Angular library. The script tag is, roughly, like a class import in other languages; such as Java, C#, or ActionScript. Keep in mind that the order of script tags in JavaScript is important.

Creating Templates and Navigation

This application will contain two main views. The first will be a login view, and the second will be the primary task view. To create the two views we will use an Angular **$routeProvider**. Based on the application's URL, the **$routeProvider** will determine which template to display. A template is a portion of an HTML document. Along with the template, the **$routeProvider** can also specify a controller for each template. The controller is the JavaScript code that powers this HTML template.

The Task Manager application will have two URLs:

- **Domain/index.html#/login**: The URL for the Login screen
- **Domain/index.html#/tasks**: The URL for the Task Management screen

The URLs tell the **$routeProvider** which template to display and how to react if the URL is unknown. This section will code the **$routeProvider** and two templates and two controllers.

The first step is to import the route provider library:

```
<script
   src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.3/angular-route.min.js">
</script>
```

This route provider library comes from the same DCN that we're loading the Angular library from. The **$routeProvider** used to be part of the main Angular library, but was separated in order to make the AngularJS more modular and lightweight.

Next, we need to tell our main application to use the route library. Open up the **LearnWith.js** file from the **com/dotComIt/learnWith/app** directory:

```
angular.module('learnWith', ['ngRoute']);
```

Earlier in this chapter we created the **learnWith** module without passing any config options into it. Now, we are passing a config option. The config option, **ngRoute**, refers Angular's **$routeProvider** library.

Now go back to the main HTML and add a div to display the templates. Put this in your **index.html** template. It should go after the body tag that contains the **ngApp** directive, but before the Script tags which load our libraries:

```
<div ng-view></div>
```

This code uses the **ngView** directive tells Angular that this div should display the contents of a template based on the **$routeProvider** of the **learnWith** application.

The **$routeProvider** will be defined in a file named **Nav.js**. I put this file in the **com/dotComIt/learnWith/navigation** directory. Add the file into the **index.html** using a script tag:

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.3/angular.min.js">
</script>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.3/angular-route.min.js">
</script>
<script src="com/dotComIt/learnWith/app/LearnWith.js"></script>
<script src="com/dotComIt/learnWith/navigation/Nav.js"></script>
```

I repeated the Angular libraries and the **learnWith** script imports in this segment to clarify a point. The order of the script tags in the **index.html** file is important. The main Angular library must be first. The Angular route library is next. The application definition, **LearnWith.js**, is third. The **Nav.js** file, with the **$routeProvider**, must come after the application definition. This is because the **Nav.js** will have dependencies upon the **learnWith** application; just as the **learnWith** application definition has dependencies upon the AngularJS framework.

The **Nav.js** file defines the **$routeProvider** config:

```
angular.module('learnWith').config(['$routeProvider',
 function ($routeProvider) {
    $routeProvider
     .when('/login',
          {templateUrl:'com/dotComIt/learnWith/views/login/Login.html',
           controller: 'LoginCtrl'})
     .when('/tasks',
          {templateUrl: 'com/dotComIt/learnWith/views/tasks/MainScreen.html',
           controller: 'MainScreenCtrl'})
     .otherwise({redirectTo: '/login'});
    }
]);
```

First, I want to highlight that this template references the **learnWith** module. This is similar code to when the module was created, but you'll notice that the options array is missing here. That is an important distinction. This line creates a new application:

```
angular.module('learnWith', [])
```

This line returns an existing application:

```
angular.module('learnWith')
```

It is an important thing to remember when you are accessing the same Angular module across multiple files.

The code calls the **config()** function on the Angular module. The config function contains a single argument, which is an array. Angular's version of dependency injection is used to insert the Angular **$routeProvider** service into the config function. In short hand, it is like this:

```
['$routeProvider', function ($routeProvider) {}]
```

The array is passed as a parameter into the **config()** function. The array primarily contains strings, but the final argument is a function. For each string in the array, AngularJS will look for an AngularJS service. It will either be defined by Angular itself, or something you created. The services are then passed into the function; which is always the last argument of the array. We'll see this approach to dependency injection used throughout this book.

The config function defines the **$routeProvider**. The **$routeProvider** is like a one big switch statement. When the URL postfix is equal to "login", it will display the **Login.html** template and activate the controller named **LoginCtrl**. When the URL postfix is named 'tasks', the **MainScreen.html** template will be displayed with the controller **MainScreenCtrl**. If the URL equals neither of those, then the app redirects back to "login". This is a very powerful concept.

Next, we are going to create the two controllers. Afterwards, we'll examine the two templates. For the moment, both the controllers and the templates are going to be stubs with very little functionality. We'll expand on them in later chapters.

The Controllers

The next step is to create the two controllers referenced in the **$routeProvider** of the **Nav.js**. I created a directory **com/dotComIt/learnWith/controllers** to contain the controllers. The first file to examine is the **LoginCtrl.js**:

```
angular.module('learnWith').controller('LoginCtrl', ['$scope',
  function($scope){
    $scope.title = 'Login View';
  }
]);
```

A controller is defined using the **controller()** method of the angular module. The logic is located in a **controller()** function. The first argument, **LoginCtrl**, is the name of the controller. It matches the same name referenced in the **$routeProvider**. We see the dependency injection syntax used here to inject the **$scope** variable into the controller function. **$scope** is a special variable in the context of AngularJS Any values or functions in the **$scope** can easily be accessed inside a view template. For now, only a single variable is created in the login controller; one named title.

The next file to examine is the **MainScreenCtrl.js** file. It mimics the **LoginCtrl** setup:

```
angular.module('learnWith').controller('MainScreenCtrl', ['$scope',
  function($scope){
    $scope.title = 'Main View';
  }
]);
```

The title variable in this controller is set to Main View instead of Login View. We'll use the title view to differentiate our two templates.

The main application needs to know about both controllers, so you'll have to import them into the main **index.html** page using the HTML script tag:

```
<script src="com/dotComIt/learnWith/controllers/LoginCtrl.js"></script>
<script src="com/dotComIt/learnWith/controllers/MainScreenCtrl.js"></script>
```

These two script tags should be added to the main page after the application is defined. I put these two scripts after the **Nav.js** script tag.

AngularJS HTML Templates

There are two templates we need to create for our AngularJS app; one for the login screen and one for the main screen. As with the controllers they'll start with simple templates which we will expand upon in later chapters. I created a views directory for the view templates at **com/dotComIt/learnWith/views**. All views will go under this directory, or more likely in a subdirectory.

The **Login.html** file was added in a login directory under the views directory. This is it:

```
<h1>{{title}}</h1>
<a href="#/tasks">Go to View 2</a>
```

There are some important things to note about this. First, the title variable is displayed between a double set of curly brackets that will look like this: {{}}. This is special AngularJS syntax to say, "look for a variable named title in the controller's **$scope** and put its value here." You can use this approach to bind to text, like we are here, but also to input controls or to anything you can put on the HTML page. You'll see a lot of these in the book.

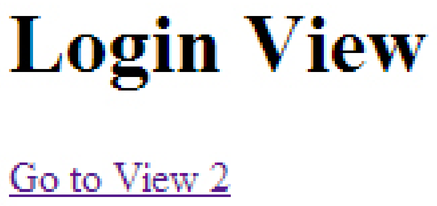
The other aspect of the **Login.html** template is a link. This link adds the "tasks" text to the URL; which the Angular **$routeProvider** will use to load the second template. I added this strictly for debugging purposes at the moment and it will be removed as the application develops.

This is the **MainScreen.html** template:

```
<h1>{{title}}</h1>
<a href="#/login">Go to View 1</a>
```

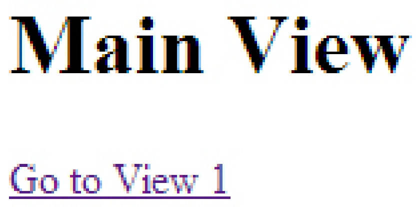
This template is located in **com/dotComIt/learnWith/views/tasks** directory. It mirrors the **Login.html** for now. It displays the **title** variable and provides a link to switch to the login view.

At this point you can load the app and you should see this view:


Login View

Go to View 2

Clicking on the Go to View 2 link will bring this the other template:


Main View

Go to View 1

Feel free to switch between the two templates by clicking the links.

Create the Database

I built the database behind this application in SQL Server, as that is what most of my clients have used over the years. If you want to set up your own local environment, there are two SQL Scripts in the database directory of the code archive:

- **GenerateDatabaseJustSchema.sql**: This script will create the database schema without creating any data.
- **GenerateDatabaseSchemaAndData.sql**: This script will create the database schema and pre-populate all tables with some test data.

You can run either file that you desire, though I recommend the second one. These scripts will not create a database file, so you'll have to create the database first and then run these scripts against the database you create. Table structure details will be covered in future chapters.

Setup ColdFusion

This section will set up the **Application.cfc**, which is the only code infrastructure needed for the ColdFusion application.

Installing ColdFusion

The first thing you'll need to do is install ColdFusion on your machine if you haven't already. The Adobe documentation on this is going to be more complete than anything I could offer here. You can use the developer edition at no cost to test this code. Once you download the binary, stepping through the installation wizard should be a walk in the park.

Once you have ColdFusion installed, you can create a DNS entry in the ColdFusion administrator to point to local database we created in the previous section. The ColdFusion docs explain how to do this in detail. ColdFusion has built in support for most types of databases, without us having to change our CFML code at all.

The Application.cfc

ColdFusion has a special file, the **Application.cfc**, which sets up some important structural information about the application. It defines the application's memory space along with other information; how to set up sessions, error handling, and application specific data. The **Application.cfc** should go in the root directory of the web server.

This ColdFusion application will only be used to deliver services for our UI code, so session management is not an issue. It will use a few Application specific variables. This is the start of the **Application.cfc** code:

```
<cfcomponent displayname="ApplicationCFC" output="true" >
  <cfscript>
      this.name = "learnWith";
      this.applicationTimeout = createTimeSpan(1,1,0,0);
      this.sessionManagement = "false";
   </cfscript>
</cfcomponent>
```

ColdFusion is primarily a tag-based language if you haven't seen it before. A **cfcomponent** is analogous to a class. The **Application.cfc** is a special class that the ColdFusion application server knows to look for. In the **CFScript** block at the top of the file, the name of the application is defined—**learnWith**. The application timeout is set to one day and one hour. This means that any application specific data will time out after twenty-five hours of inactivity. Session management is turned off, as the server will not be handling sessions in the same way that a traditional HTML app might make use of sessions. The UI will handle all our session management.

Next in the file is the **onApplicationStart** method:

```
<cffunction name="onApplicationStart" returntype="boolean" output="true">
      <cfset application.dsn = "learnWith">
      <cfset application.debugMode = 0>
      <cfset application.componentPrefix = "coldFusion">
      <cfreturn true>
</cffunction>
```

The **onApplicationStart** method will execute whenever the application is used after the application has timed out. For this application, I am setting three values. One is the **dsn**, which will point to the data source set up in the ColdFusion administrator. The second is the **datasource**, which should point to the database you created using one of the SQL Scripts. The last is a **debugMode** variable. I will use the **debugMode** variable inside the service code to determine whether debugging output should be displayed to the screen or not. When returning JSON to a UI making use of the service call, you want debug mode to be turned off. However, I find it useful when testing the services from within ColdFusion. The final value is named **componentPrefix**. If you build your ColdFusion code in the root directory, this can be an empty string. In order to keep the code for each chapter separate, I'll use this when instantiating other CFCs.

The **Application.cfc** allows for code to be executed at the start of a request using **onRequestStart()** and at the end of the request using **onRequestEnd()**. This application isn't using **onRequestEnd()**, however it does make use of **onRequestStart()**:

```
<cffunction name="onRequestStart" returntype="boolean" output="false">
      <cfargument name="thePage"type="string"required="true">
      <cfif isDefined('url.reinit')>
            <cfset onApplicationStart()>
      </cfif>
      <cfreturn true>
</cffunction>
```

This **onRequestStart()** method provides an easy back door if you want to reset the application variables. An approach like this is common in many ColdFusion applications I have worked on. Just load any page with the URL variable **reinit** and the **onApplicationStart()** method will be manually called.

The next method is the **onError()** method. This method just emails the error to you as a quick form of logging. I have also worked on apps that will display the error to the screen which is good for debugging but less ideal for production:

```
<cffunction name="onError" returnType="void" output="true">
      <cfargument name="exception" required="true">
      <cfargument name="eventname" type="string" required="true">
      <cfset var to="myEmail@mydomain.com">
      <cfset var from="myEmail@mydomain.com">
      <cfmail to="#to#" from="#from#" replyto="#from#"
            subject="Learn With Error #cgi.HTTP_HOST#" type="html">
            Error Date: #now()#<Br/><br/>
            Exception:
            <cfdump var="#arguments.exception#">
            EventName:
            <cfdump var="#arguments.eventname#">
            CGI:
            <cfdump var="#cgi#">
            Request
            <Cfdump var="#request#">
      </cfmail>
</cffunction>
```

If we were using session management in this application, we can run code in the start and end of a session using **onSessionStart()** and **onSessionEnd()**. But for this application, neither method would apply.

Creating and putting the **Application.cfc** file in the root directory of your ColdFusion web application server will get you on your way to building this application. Some values that referenced in the **Application.cfc** are set up in the ColdFusion Administrator. A couple of them are the DNS name pointing at the SQL Server database, and the mail server that is used to send the email.

Final Thoughts

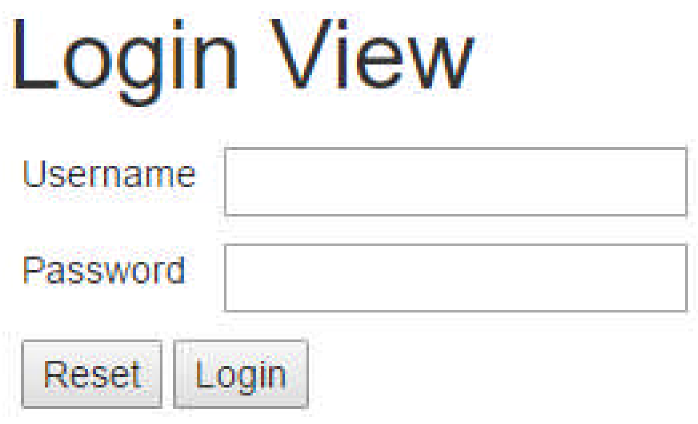
This chapter showed you the full scope of the application that will be built, including screenshots of each main section of the app. It also built up the main application shell, showing you how to use templates and controllers with a routing provider to display aspects of the application. The next chapter will implement the login functionality

Chapter 2: Login

This chapter will examine the authentication aspect of the Task Manager application. It will build out the User Interface and show you how to connect to a service. It will build upon the application skeleton created in the previous chapter.

Create the User Interface

This section will take a look at building the login form for our application. The finished Login screen will look like this:



This layout is simple and should be no problem if you have moderate HTML skills. Open up the **Login.html** file from the **com/dotComIt/learnWith/views/login** directory. I put the elements in a two column table. The table has three rows with the final row spanning across both columns:

```
<table>
    <tr>
        <td>Username</td>
        <td><input type="text" /></td>
    </tr>
    <tr>
        <td>Password</td>
        <td><input type="password" /></td>
    <tr>
        <td colspan="2">
            <input type="button" value="Reset" ng-click="onReset()" />
            <input type="button" value="Login" ng-click="onLogin()" />
        </td>
    </tr>
</table>
```

I’m sure there is a different way to create this layout using CSS, but my goal is to be effective and quick without obsessing over layouts.

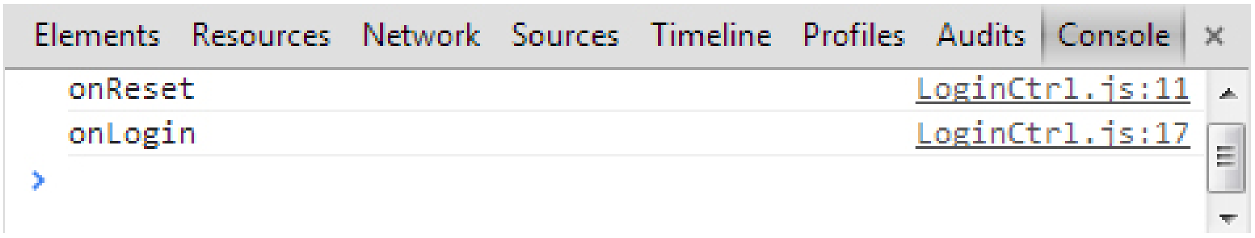
A new Angular directive is used in this template. The buttons at the bottom of the template have an **ngClick**. When one of the buttons is clicked, this directive tells Angular to execute the specified function in the controller.

For now, let's add some placeholders for each function. These functions should be put inside the **LoginCtrl** from the **com/dotComIt/learnWith/controllers** directory:

```
$scope.onReset = function onReset(){
    console.log('onReset');
}
$scope.onLogin= function onLogin(){
    console.log('onLogin');
}
```

The name of the function called in the template is the name of the **$scope** variable that contains the function.

If you're adventurous, you can run the app again. Open up a browser and bring up some web developer tools—I use F12 in Chrome on Windows. Click the buttons and you'll see the console output is logged:



```
onReset                                    LoginCtrl.js:11
onLogin                                    LoginCtrl.js:17
```

A Quick Introduction to JSON

Before jumping into the services section, I wanted to give a brief review of JSON. As an experienced developer, JSON should be easy for you to understand. JSON is the most standardized format when building HTML5 applications. A JSON packet looks like a JavaScript object.

The services will always return a JSON object. It consists of a bunch of name value pairs but can also support arrays and nested objects. This is a simple JSON packet:

```
{
  “stringProperty”: “value1”,
  “numberproperty”: 1
}
```

The curly brackets, ‘**{**‘ and ‘**}**’ distinguish the start and end of an object. Properties inside the object consist of the property name, followed by a colon, ending with the value. Property names are enclosed in double quotes. Properties are put into a comma-separated list. This JSON object is a single object, which contains two properties: **stringProperty** and **numberProperty**.

A JSON packet can also support nested objects:

```
{
  “stringProperty”: “value1”,
  “numberProperty”: 1,
  “objectProperty”: {
                    “embeddedStringProperty”:”embeddedValue”,
                    “embeddedNumberProperty”:2
                    }
}
```

The new JSON packet expands on the previous one, just adding a new property; **objectProperty**. The value of **objectProperty** is an embedded object with two properties of its own. In essence, this is an object containing another object. The embedded object has an identical syntax to the main object. Elements can be embedded for as many layers down as you need.

JSON can also support arrays of values. This packet includes an array of numbers as an example:

```
{
  “stringProperty”: “value1”,
  “numberProperty”: 1,
  “objectProperty”: {
                      “embeddedStringProperty”:”embeddedValue”,
                      “embeddedNumberProperty”:2
                     },
  “arrayProperty”: [1,2,3,4,5,6,7,8,9,10]
}
```

Arrays are distinguished in a JSON packet using the opening square bracket, ‘**[**‘**,** and closing square bracket, ‘**]**’. In the case above, the array contains a comma-separated list of numbers. An array can also contain strings or objects.

In this sample the JSON packet is an array:

```
[
 {
  “stringProperty”: “value1”,
  “numberProperty”: 1,
  “objectProperty”: {
                      “embeddedStringProperty”:”embeddedValue”,
                      “embeddedNumberProperty”:2
                     },
  “arrayProperty”: [1,2,3,4,5,6,7,8,9,10]
 },
 {
  “stringProperty”: “value2”,
  “numberProperty”: 3,
  “objectProperty”: {
                      “embeddedStringProperty”:”embeddedValue2”,
                      “embeddedNumberProperty”:4
                     },
  “arrayProperty”: [10,9,8,7,6,5,4,3,2,1]
 }
]
```

Instead of starting with a curly bracket to represent an object, it starts with a square bracket to represent an array. The elements of the array are objects. In this case they are identical to each other, a common occurrence when building applications. The objects are identical, but the property values are different.

Examine the Database

Applications often use a database for permanent storage. Most of my clients use SQL Server, so I used it here; but any database server of your choice will work. This application should support role-based authentication. This means the user will login and will be assigned a role. Then functionality within the app will turn on or off based on those roles. For the sake of this sample, I made the assumption that each user will only have a single role. This would be a one-to-many relationship in database-speak.

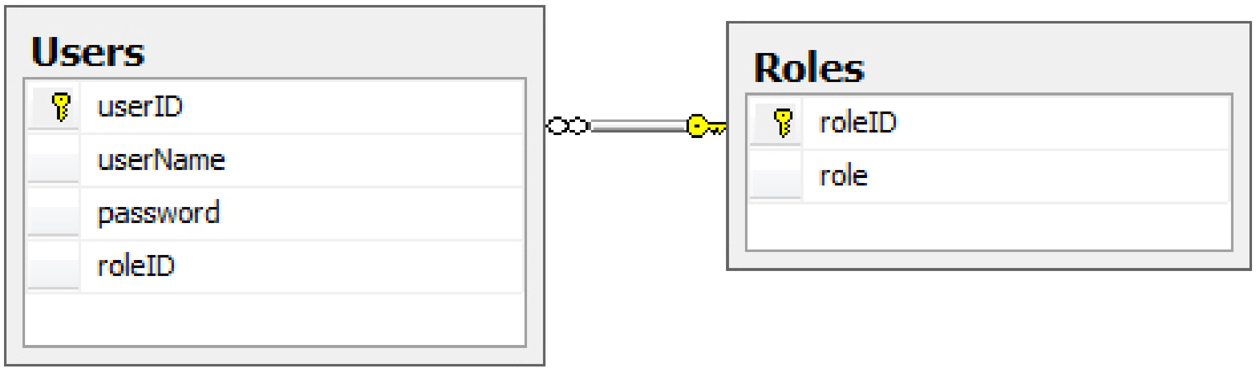
The data to store for users is this:

- **userID**: The primary key for this user.
- **userName**: The name that this user will use to login.
- **password**: This column will store the user's hashed password. I'll speak more about the password hashing in a bit.
- **roleID**: This column will be a foreign key; used to associate the user with their specific permissions from the role table.

The data stored for the user roles is this:

- **roleID**: The primary key for the role. This field is mirrored in the users table as a foreign key and is used to relate the two tables.
- **role**: This is a text field for the role which is any text descriptor you wish to use.

The table structure looks like this:



In the UI, the user will enter their username and password, which will get sent to the ColdFusion service. The service will then run a query to select the user information. The SQL Query will be like this:

```
select *
from users
where username = 'inputUsername' and password ='inputPassword'
```

The **inputPassword** will have to be pre-hashed, as we will not be storing passwords in plain text. The **inputUsername** can be plain text. For the purposes of authentication, we don't need to join the two tables because the name of the role is needed. The UI can use only the **roleID** to control access to certain features.

For the purposes of this sample, I have created two roles for this application:

- **Tasker**: This is the role that can create and edit tasks. In our imaginary world, this is me.
- **Creator**: This role can view tasks, and can create tasks. They cannot edit tasks, schedule tasks, or mark tasks as completed. In our imaginary world, this is my wife, who wants to know what I'm up to on any given day.

Following this, I have created two users in the database:

- **Me/Me**: The user with the **RoleID** of "1", who will have full access to the application.
- **Wife/Wife**: The wife user will have the **RoleID** of "2", and will be able to view and create tasks, but not edit or schedule tasks.

These two roles allow us to mimic a more complicated system that may exist within the context of an Enterprise client.

Write the Services

This section introduces the implementation of the authentication service that powers the login screen of the Task Manager application. It will cover the ColdFusion code as well as the database structure. In addition, it will create the AngularJS services used to integrate into the app.

The Generic Return Object

Somewhere in my own development travels, I decided that it is best to always return a consistent object from my services. This helps when building UI services because there is always a consistent object I know to look for in the results from the service. The class name of this object is **ResultObjectVO**, and it contains two properties:

- **error**: This is a Boolean value. If the error flag is "true", then the UI will know that there is an error, and it will stop further processing. When this happens, the UI will display a message to the user. If the error flag is "false", then the UI should know there is no error and it can continue processing as normal.
- **resultObject**: This property is a generic object. If there is an error, then the **resultObject** will contain a string description of the error that the UI displays to the user. If there is no error, then the **resultObject** will be the value expected from the service. In our authentication service, it will contain user details. When retrieving task information, it will contain an array of tasks. The UI will know how to process the **resultObject** based on the type of call made, and the information expected back.

Even though the plan is to return JSON from the services, there is a reason to create a value object with these properties. ColdFusion makes it really simple to convert a CFC into JSON. I put this CFC inside the **com/dotComIt/learnWith/vos** directory. This is the CFC:

```
<cfcomponent displayname="ResultObjectVO"
             hint="I represent the response from a remote service call"
             alias="com.dotcomit.learnWith.vos.ResultObjectVO">
      <cfproperty name="error" type="Boolean" >
      <cfproperty name="resultObject" type="any" >
</cfcomponent>
```

The CFC is defined by the **cfcomponent** tag. There is a **displayname**, and a hint which are used primarily for documentation purposes. The alias tag is used for AMF conversion from server side to client side objects, but it won't make a difference in our JSON conversions. The two properties are defined with the **cfproperty** tag. The error is a Boolean value, and the **resultObject** can be of any type.

The User Value Object

Before jumping into the CFC used to create the authentication code, I wanted to cover the **UserVO**. The **UserVO** will be used by a service similar to the **ResultObjectVO** from the previous section. The **resultObject** property of the **ResultObjectVO** returned by the service will contain a **UserVO** instance if the authentication is successful.

I put the **UserVO** in the **com/dotComIt/learnWith/vos** directory. The database query will return all items in the user table; the username, the password, the role, and the **userID**. Since there is no need to store the password in the UI, only three of those properties will be in our **UserVO**:

```
<cfcomponent displayname="UserVO"
             alias="com.dotcomit.learnWith.vos.UserVO">
      <cfproperty name="userID" type="numeric" />
      <cfproperty name="username" type="String" />
      <cfproperty name="roleID" type="numeric" />
</cfcomponent>
```

The component is defined with the **cfcomponent** tag, and has the **displayname** and hint for documentation purposes. I also added an alias property—which defines the path name that would be used to create an instance of the object. This is for documentation purposes, as well. The three properties are defined with the **cfproperty** tag. The **userID** and **roleID** are numeric. The username is a **String**.

The Authentication Service

ColdFusion uses CFC files, which are like classes, to create and expose services for use by remote applications. It easily supports AMF, SOAP, and REST web services. When building back ends for HTML applications, I like to use REST and JSON services.

For the authentication service, I created an **Authentication.cfc** in the package **com/dotComIt/learnWith/services**.

This is the empty component:

```
<cfcomponent displayname="AuthenticationService"
             hint="I handle authentication for the LearnWith test app">
</cfcomponent>
```

The **displayName** and **hint** on the **cfcomponent** tag are just metadata that is added to the built-in component documentation.

Inside the component, I'll add a function declaration:

```
<cffunction name="authenticate" returnformat="plain" access="remote"
            output="true"
            hint="I authenticate the user for the LearnWith test app" >
    <cfargument name="username" type="string" required="true" />
    <cfargument name="password" type="string" required="true"
                hint="I am the hashed password" />
</cffunction>
```

The function is named **authenticate**. The **returnFormat** is plain. I do this because I will handle the conversion of our results to JSON manually. The access is remote and this means that the function can be called by remote applications; such as our browser. The hint on the **cffunction** tag is just some documentation.

The function accepts two input parameters as defined by the **cfargument** tag; the username, and password. Both are strings, and the comment specifies that the password should already be hashed. When it comes to hooking up this method to our app, we'll handle the password hashing inside of it.

The first order of business inside this function is to query the database:

```
<cfquery datasource="#application.dsn#" name="local.dataQuery" >
    select * from users
    where username = <cfqueryparam cfsqltype="cf_sql_varchar"
                                   value="#arguments.username#"> and
          password = <cfqueryparam cfsqltype="cf_sql_varchar"
                                   value="#arguments.password#">
</cfquery>
<cfif application.debugMode is 1>
    <cfdump var="#local.dataQuery#" /><Br/>
</cfif>
```

This uses the ColdFusion **cfquery** tag to query the database. It references the **application.dsn** variable, which is the application specific variable used to store our **datasource** name. It saves the results of the query in the variable that is local to the function; **local.dataQuery**. The query text is similar to what we reviewed earlier in the Database Storage section of this chapter. The only difference is that we used **cfqueryparam** around the method arguments. This is ColdFusion's built-in method for scrubbing data from malicious attacks.

After the query, we make use of our application specific **debugMode** parameter. If it is set to "1", then the results of the query are displayed as part of the output of this function. If it is set to "0", then nothing is displayed. Whenever calling the method that looks for real JSON output, you'll want to be sure that **application.debugMode** is set to "0".

The next step is to process the resulting query by storing the data in the **UserVO** value object, and then storing the **UserVO** object in a **ResultObjectVO** instance. Afterwards, everything is converted to JSON and returned.

First, create an instance of the **ResultObjectVO**:

```
<cfset local.resultObject =
      createObject('component','com.dotComIt.learnWith.vos.ResultObjectVO')/>
```

If there are no results returned in the query, then the user did not authenticate correctly. If there are results, then the user authenticated correctly and we need to store the results as part of the **resultObject**. In ColdFusion, we can determine if there are results returned by using the **recordCount** variable on the **dataQuery**:

```
<cfif local.dataQuery.recordcount EQ 1>
  <cfset local.tempObject = createObject('component',
                                'com.dotComIt.learnWith.vos.UserVO')/>
  <cfset local.tempObject['userID'] = dataQuery.userID />
  <cfset local.tempObject['username'] = dataQuery.username />
  <cfset local.tempObject['role'] = dataQuery.roleID />
  <cfset local.resultObject['error'] = 0 />
  <cfset local.resultObject['resultObject'] = local.tempObject />
<cfelse>
  <cfset local.resultObject['error'] = 1 />
</cfif>
```

For authentication purposes, we want to be sure that the **recordCount** is “1”. If there are fewer records returned, then no user was found, and the credentials were incorrect. If there are more records returned, than those credentials were verified for more than one user which is indicative of an underlying system problem.

The final piece of code for the authenticate method is to convert the **local.resultObject** into JSON:

```
<cfset local.resultString = SerializeJSON(local.resultObject) />
<cfreturn local.resultString>
```

ColdFusion’s **SerializeJSON()** method is doing the hard work for us. The last line returns the resulting JSON values from the method.

Testing the Authentication Service

I created a test case to test the service. To keep things simple, I used a simple cfm page which will create an instance of the service, all the authenticated method on it, and output the results. The page is **AuthenticationTest.cfm** in the **com/dotComIt/learnWith/tests** directory. I can test the service by loading that page in a browser:

```
<cfset application.debugMode = 1>
<h1>Test Authentication Success</h1>
<cfset service = createObject('component',
                'com.dotComIt.learnWith.services.AuthenticationService')>
<cfset passwordHashed = #hash('me')#/>
<cfset results = service.authenticate('me',passwordHashed)>
<Br/><Br/>
Results: <Br/>
<cfdump var="#results#" expand="false" />

<h1>Test Authentication Fail</h1>
<cfset results = service.authenticate('me','me')>
<Br/><Br/>
Results: <Br/>
<cfdump var="#results#" expand="false" />
<cfset application.debugMode = 0>
```

Executing the code will show results, like this for the successful test:

```
{
 "resultObject":
     {"role":1,
      "username":"me",
      "userID":1},
 "error":0.0
}
```

The results are a JSON packet; a group of comma-separated name value pairs enclosed in curly brackets. They can get more complex with arrays and nested objects. The above results show that no error was returned, because the error property is listed as “0”. It also shows that a **UserVO** object was returned in the **resultObject** property, containing three user related properties.

These are the results for a failure test:

```
{
 "error":1.0
}
```

It simply has the error property set to “1”, and no **resultObject** is returned.

Hashing the Password

I wanted to highlight one aspect of the ColdFusion code related to passwords; the database requires the passwords to be hashed, and this app doesn’t include services to create and edit users. So how do you get a hashed password for entry into the database? I created a simple function for that in the Authentication service:

```
<cffunction name="hashPassword" type="string" required="true"
            hint="I am a helper function for hashing passwords" >
     <cfargument name="password" type="string" required="true"
                 hint="I am an unhashed password"/>
     <cfreturn hash(arguments.password) />
</cffunction>
```

ColdFusion already has a default hashing function, so this code is just a wrapper around it. There is no access attribute set on this function, so it will not be accessible remotely. Passwords should not be sent over the Internet in clear text.

To create the passwords in the database, I just wrote a hard-coded template to create the hashed versions of the passwords I wanted:

```
<h1>Test Hash Password Me</h1>
<cfset service = createObject('component',
                'com.dotComIt.learnWith.services.AuthenticationService')>
<cfset results = service.hashPassword('me')>
<cfdump var="#results#" expand="false" />

<h1>Test Hash Password Wife</h1>
<cfset results = service.hashPassword('wife')>
<cfdump var="#results#" expand="false" />
```

This gives the results I needed to create the database entries manually:

Test Hash Password Me
AB86A1E1EF70DFF97959067B723C5C24

Test Hash Password Wife
BB4694A26A39DF7501F8BB8CBDD13E38

If you use the script I provided to populate your own database, then you won't have to worry about entering the data manually.

Review the API

After building the CFC, you'll need to know how to make calls to it. The API is a REST API and the app will make service calls using an HTTP Post request. You need to know the endpoint, and the parameters to send in. Each form variable of the HTTP post will represent an input variable to the method.

Assuming that the app and the service are served off the same domain, you can use a relative URL for the endpoint. In this case, I used the following:

```
/com/dotComIt/learnWith/services/AuthenticationService.cfc
```

The name of the service method at this end point is **authenticate()**, which is identical to the name the method was given in the CFC. It accepts three inputs:

- **method**: When making an HTTP call to a ColdFusion CFC, this parameter needs to be specified. This tells ColdFusion which method should be executed at the end point. In this case the value should be "authenticate".
- **username**: This is the name the user uses to authenticate.
- **password**: This is the hashed string of letters and numbers entered for the user to authenticate. The password must be hashed before being sent over the wire. A hash is a type of encryption, sometimes called an MD5 hash. With an MD5 hash, the encrypted value cannot be turned back into the original. A good practice is to use a MD5 hash for passwords in security systems, so if the password store gets compromised, the original passwords cannot be retrieved.

The AngularJS code will use the end point and the parameters to call the service method.

Access the Service

This section will create an AngularJS service that will call the ColdFusion service. To perform the call, we will use the post method on Angular's **$http** service. First, we want to create the AngularJS object to contain the service. I created a new file named **AuthenticationService.js** in the **com/dotComIt/learnWith/services/coldFusion** directory.

The primary tasks for this file are to create the service, and implement the authentication method inside the service. This is the code outline for the service:

```
angular.module('learnWith').service('AuthenticationService',
  ['$http',function($http){
     var services = {
        authenticate : authenticate
     }
     return services;
  }
]);
```

The **learnWith** application is referenced using Angular's **module()** function. The **service()** method is executed against the application instance. The **service()** method is similar to the method we used to create controllers. It accepts two arguments; the first being the name of the service, and the second being an options array. This options array is Angular's version of dependency injection. Here, we are injecting a single argument—Angular's **$http** service—into our custom **AuthenticationService**. The last argument of the options array is a function which is executed the first time the service is retrieved by Angular.

Inside the **AuthenticationService** function, an object is created named **services**. The object contains a list of functions and variables that can be accessed by the service. In this case, we only have one function; authentication.

Let us implement the **authenticate()** method inside the **authenticationService** object:

```
this.authenticate = function authenticate(username, password){
}
```

This method accepts two arguments, a username and a password.

There are three different parameters that we need to pass to the remote service; the username, the password, and the method name. All parameters were discussed in more depth earlier in this chapter. To add these parameters to the call, you'll need to put them in a string; like a query string that you may find in a URL:

```
var parameters =  "username" + '=' + username + '&';
parameters += "password" +'='+ hex_md5(password) + "&";
parameters += "method" + "=" + "authenticate" ;
```

The previous code block creates a parameter variable and adds the service method arguments to it one by one. The username and method are just used as strings. However, the password must be encrypted with an MD5 hash before sending it. This is for security purposes. It is never a good idea to send a plain text password over the wire. To perform the MD5 hash, I found an open source encryption library named **jshash**. I had to import the library in the main **index.html** file, like this:

```
<script src="js/libraries/jshash/2.2/md5-min.js"></script>
```

The final parameter string will look like this:

```
username=me&password=ab86a1e1ef70dff97959067b723c5c24&method=authenticate
```

The last step is to make the method call. There are some issues in how AngularJS makes a post call to a remote service. By default, Angular does not add the individual parameters to the call. Therefore, on the server side, they cannot be accessed as separate inputs. This is because the default content-type header used by AngularJS is **application/json**. For the purposes of this app, we need to set it to **application/x-www-form-urlencoded**. I do this in a configuration block.

First, create a file named **HTTPServiceConfig.js** in the **com/dotComIt/learnWith/services/coldFusion** directory. Add this code:

```
angular.module('learnWith').config(['$httpProvider',
 function ($httpProvider) {
     $httpProvider.defaults.headers.post['Content-Type'] =
                          'application/x-www-form-urlencoded; charset=UTF-8';
 }
]);
```

This code executes a **config** block on the **learnWith** module. An **$httpProvider** is passed in as the service. The **$httpProvider** object is used to configure the **$http** object. Inside the function, the content-type header is specified. When our service calls are executed against the **$http** service; it will inherit the **config** from the **$httpProvider** service.

Be sure to add the **HTTPServiceConfig.js** file to the main index file:

```
<script src="com/dotComIt/learnWith/services/coldFusion/HTTPServiceConfig.js">
</script>
```

A more in-depth explanation to problems with accessing server side variables through **$http** service posts, and a comprehensive solution, can be found on my personal blog. For this chapter, I set the header manually because we are only passing simple values. In future chapters, we will expand to use a global transform function.

Here is the remote call for the **AuthenticationService**:

```
return $http.post('com/dotComIt/learnWith/services/AuthenticationService.cfc',
          parameters)
```

The first argument to the post method is the endpoint of the service. The second argument is the parameter string we created earlier. The **$http.post()** returns a promise object, and the promise object is returned from the method. A promise object is part of how Angular implements asynchronous service calls. Behind the scenes, AngularJS will wait for a response from the service call. Based on the response, it will either call a success or failure method—if we set one up. You will shortly learn how to do just that.

To use this service, open the main **index.html** file and add the script tag:

```
<script src="com/dotComIt/learnWith/services/coldFusion/AuthenticationService.js">
</script>
```

If you're looking at the code archive for this series of books, you'll see that I created a separate file that uses ColdFusion services named **index_ColdFusion.html**. However, in the bulk of this text I'll be referring to it as **index.html**.

Wire Up the UI

This section will complete the implementation of our authentication method in the Angular app. It will wire up the service layer to the user interface and successfully authenticate the user. Along the way it will also talk about storing data across multiple controllers and views.

Creating a UserModel

This section will create a **UserModel**. It will be an Angular service that is used for sharing user authentication data between the different parts of this application.

First, create a new directory at **com/dotComIt/learnWith/model**. I created a JavaScript file in this called **UserModel.js**.

Start with this:

```
angular.module('learnWith').service('UserModel', [function(){
}]);
```

This creates an Angular service named **UserModel**, which is similar to the **AuthenticationService** created earlier. The only difference is our intent. The **AuthenticationService** was designed to encapsulate remote access functionality. The **UserModel** is designed as a client side data store. Notice that nothing is injected into this service. For the purposes of the **UserModel**, no external services are needed.

Next, define and return the **UserModel** object:

```
var userModel = {
    user : {
        userID : 0,
        username : '',
        password : '',
        roleID : 0
    }
}
return userModel;
```

The **userModel** object contains a user property, which is itself an object. The user is like a value object, containing data points that represent the user. The username and password will be used to sync to the UI elements as the user fills out the login form. The **userID** and **roleID** will come later.

Now turn your attention back to the **index.html** page. Using the JavaScript script tag, you can include the **UserModel** and **ModelLocator** into the template:

```
<script src="com/dotComIt/learnWith/app/LearnWith.js"></script>
<script src="com/dotComIt/learnWith/model/UserModel.js"></script>
<script src="com/dotComIt/learnWith/services/mock/AuthenticationService.js">
</script>
```

I put the **UserModel** script tag between the **LearnWith.js** main application and the **AuthenticationService.js** script tag.

Add the Model to your Controller

Now that we have a **UserModel** object created inside the **UserModel** service, the object needs to be passed into the controller so that it can be used in both the controller and in the view template. To do this, open up the **LoginCtrl.js** file from the controller directory.

When defining the service in **UserModel.js**, it was given a name of **UserModel**. Simply add that as an argument onto the controller definition:

```
angular.module('learnWith').controller('LoginCtrl',
  ['$scope','AuthenticationService','UserModel',
    function($scope,AuthenticationService,UserModel){
```

This passes the **UserModel** argument into the **LoginCtrl**. AngularJS automatically knows how to find the **UserModel** service because it was defined on the same **learnWith** module. The **UserModel** is not yet available to the view. To do that, we'll have to add the UserModel into the **$scope**:

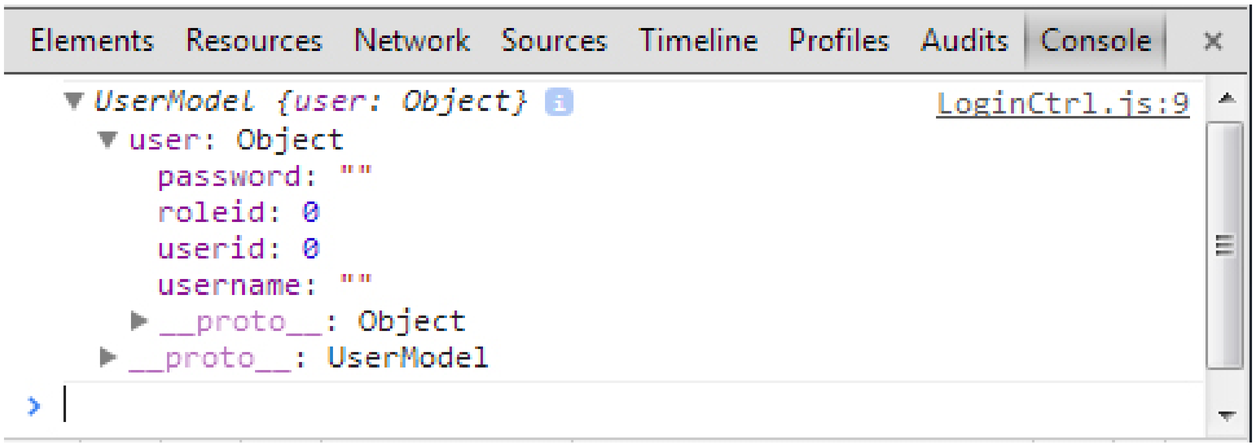
```
$scope.userModel = UserModel;
```

This now makes the user model available to the **Login.html** view template that was associated with the **LoginCtrl** controller via the **$routeProvider** definition we created in Chapter 1.

In order to prove that this all works as I say, I'm going to add a **console.log()** statement to output the **userModel**:

```
console.log($scope.userModel);
```

This code can go right after the spot where the **$scope.UserModel** is defined. Since the code isn't in a function it will be executed as the app loads. You can load the app and you'll see the output in the browser's console:



```
▼ UserModel {user: Object}                    LoginCtrl.js:9
  ▼ user: Object
      password: ""
      roleid: 0
      userid: 0
      username: ""
    ▶ __proto__: Object
  ▶ __proto__: UserModel
>
```

Even though we haven't made any changes to the values in the **UserModel**'s user object, the defaults are all there.

Accessing the Model Values from Within the View

For this next step, you'll need to switch over to the **Login.html** template in the **com/dotComIt/learnWith/views.login** directory. We need to tie the input variable to the model. For that, we'll use an AngularJS directive named **ngModel**.

Here is the code:

```
<input type="text" ng-model="userModel.user.username"/>
```

The **ngModel** is an Angular directive. It changes the value of the input when the variable changes. It also changes the variable if the input changes. Angular knows to look in the controller's **$scope** for the **userModel** variable. In this case, it accesses **userModel.user.username**; drilling down into the **UserModel** service to get to the user object, and finally to the object's username property.

The same approach is used for the password:

```
<input type="password" ng-model="userModel.user.password"/>
```

This is an easy way to sync changes between the input fields for the username and password with the values in our **UserModel** singleton.

Implementing the Reset Button

There are two buttons to implement in the UI; the reset button and the login button. The reset button is easier to implement, so let's start there. You already saw the **ngClick** on the reset button and created a method stub for **onReset()**. The code for the **onReset()** function is inside the **LoginCtrl**. Previously, we had created a method stub.

Here is the full implementation:

```
$scope.onReset = function onReset(){
   $scope.userModel.user.username = '';
   $scope.userModel.user.password = '';
}
```

The code inside the button accesses the **userModel** values for username and gives them blank values. The Angular form of binding automatically knows to update the view template with the new, blank values.

Implementing the Login Handler

The last step in the Authentication process is to wire up the login button to a method, integrate the service call inside the method, and handle the results. Potentially, the results could be errors so we need a way to display errors to the user. Open up the **Login.html** template in the **com/dotComIt/learnWith/views/login** directory. Here, we are going to add more table cells to the layout table.

Add this table cell after the username input:

```
<td class="error">{{usernameError}}</td>
```

The **userNameError** variable is something we'll define shortly inside the Controller.

After the table cell with the password input, add this:

```
<td class="error">{{passwordError}}</td>
```

The **passwordError** is something we'll add in the **LoginCtrl** shortly. You'll notice that the two table cells have a class associated with them; **error**. This is a CSS Style. Although the purpose of this book is not to focus on design aspects of HTML5, I didn't want to leave the app completely dry. I created a new directory for **com/dotComIt/learnWith/styles** and put a **styles.css** file in there. The file contains one simple CSS class:

```
.error {
    color: #ff0000;
}
```

Go to the main **index.html** page and load the style sheet within the document header:

```
<link href="com/dotComIt/learnWith/styles/styles.css" rel="stylesheet"
      type="text/css" />
```

Now we can move away from **Login.html** template and into the **LoginCtrl.js** file. This file contains the controller with the code behind the login template. First, you need to create the two variables to contain the error:

```
$scope.usernameError = '';
$scope.passwordError = '';
```

These two variables can be put in the JavaScript after the **userModel** variable is created in the **$scope**.

Now it is time to implement the **onLogin()** method. When we last saw this method, it had nothing more than an output to the console. Here is the method signature again:

```
$scope.onLogin= function onLogin(){
}
```

The first step in the method is to perform some error checking and if there are errors, populate the error variables, and exit the method. First, create a Boolean value to determine whether an error was found or not:

```
var errorFound = false;
```

The default value of the **errorFound** is "false". If we find errors, the **errorFound** variable will change to "true". First, check to make sure that a username was entered. Simple conditional logic is used:

```
if($scope.userModel.user.username == ''){
    $scope.usernameError = 'You Must Enter a Username'
    errorFound = true;
} else {
    $scope.usernameError = '';
}
```

The conditional checks the username property of the user object on the **userModel** variable. If it has no value, change the **usernameError** variable— which will in turn update the view. It also sets the **errorFound** variable to "true". If a username was entered, then the **usernameError** value is set to blank. This will effectively remove the error from the user's display.

Error checking for the password acts similarly:

```
if($scope.userModel.user.password == ''){
    $scope.passwordError = 'You Must Enter a Password';
    errorFound = true;
} else {
    $scope.passwordError = '';
}
```

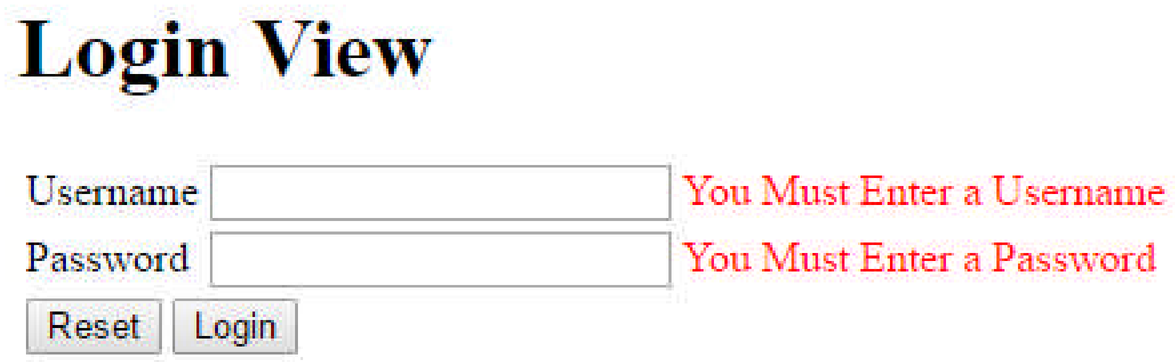
The reason that the **errorFound** is not set to “false” in the **else** condition is because it is possible the user could have entered a password, but not a username. In which case setting the **errorFound** variable to “false” would essentially negate the first round of tests.

Here is the last step in error checking:

```
if(errorFound == true){
    return;
}
```

This code segment checks the **errorFound** variable. If it is “true”, it uses the “return” keyword. “return” tells the method to stop executing and return a value. Though, in this case, we aren’t returning any value.

You can test this out in a browser. Click the “Login” button without entering any values to see this:



The last step is to execute the authenticate method on the **AuthenticationService**:

```
AuthenticationService.authenticate($scope.userModel.user.username,
                                   $scope.userModel.user.password)
.then(onLoginSuccess,onLoginError);
```

This calls the **authenticate** method in our mock service. It passes in the username and password. The result is a promise object. The **then()** function is called on the promise object. The **then()** function accepts two arguments; both functions. The first function, **onLoginSuccess()**, is executed when a successful result is achieved. The second, **onLoginError()**, is called when the method fails. Failure would usually be a problem making the service call. In most cases, even a failed login would return a valid result from the server. However, I have worked on one application where the service layer would return an HTTP status code 403—meaning forbidden access—if a login failed. That situation would execute the failure method instead of the success method. I've grown fond of that approach. Though, it is uncommon among my clientele.

This is the **onLoginError()** method:

```
var onLoginError = function onLoginError(response){
    console.log('failure');
    console.log(response);
    alert(response.data);
}
```

The **onLoginError()** method strictly logs the error to the console and shows the user an alert. There is a single argument to this method, response. The data is an object containing the following:

- **data**: This argument represents the response from the service.
- **status**: This status argument represents the HTTP status code of the response.
- **headers**: This header's argument represents the headers of the remote service call's response.
- **config**: This represents the configuration object used to generate the request.
- **statusText**: This property contains the HTTP status text included in the response.

For the purpose of this **error** method, only the data is used to display to the user. The **success** and **error** methods have the same object as an argument. It is something you'll see on a lot of the calls from the app.

The **onLoginSuccess()** method has two priorities; to save the **resultObject**'s values into the **UserModel**, and to redirect to the main task screen.

This is the method:

```
var onLoginSuccess = function onLoginSuccess(response){
    if(response.data.error == 1){
        alert("We could not log you in");
        return;
    }
    $scope.userModel.user = response.data.resultObject;
    $location.path( "/tasks" );
}
```

First the method checks to see if the error flag in the returned object is "1". If it is, then it displays an error to the user with an alert box and stops processing. If there is no error, then the returned **resultObject** is stored in the **userModel** as a replacement for the default user object. Finally, the **$location** service is used to redirect the main view. The **$location** variable is an AngularJS service that can be used for modifying the application's URL.

You'll have to modify the **LoginScreenCtrl** definition to make use of the service:

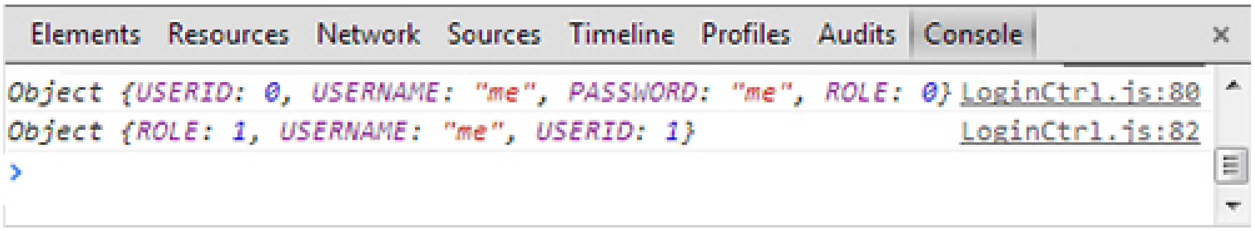
```
angular.module('learnWith').controller('LoginCtrl',
  ['$scope','AuthenticationService','UserModel','$location',
    function($scope,AuthenticationService,UserModel,$location){
```

The path method on the **$location** service is used, and the new path is added to the URL. The next chapter will start the implementation of the default tasks view.

You can add some **console.log** statements to the **onLoginSuccess()** method to view the user object of the **UserModel** before and after the method call:

```
console.log($scope.userModel.user);
$scope.userModel.user = response.data.resultObject;
console.log($scope.userModel.user);
```

You'll see something like this:



```
Object {USERID: 0, USERNAME: "me", PASSWORD: "me", ROLE: 0} LoginCtrl.js:80
Object {ROLE: 1, USERNAME: "me", USERID: 1}                 LoginCtrl.js:82
>
```

It is worth noting that the returned object does not include the password. This is for security purposes. After the user successfully logs in the app no longer needs the password.

Final Thoughts

This chapter implemented a login screen which communicated with a remote service. We created view templates and controllers and shared data between the two using the Angular **$scope** object. If this is your first look at AngularJS, I wanted to give a full accounting of what happens when you load the AngularJS App:

1. The main HTML page loads and all the JavaScript files are loaded in the order that their script tags are listed on the page.
2. The **LearnWith.js** template defines the Angular app and associates itself with the **ngApp** directive on the HTML Body.
3. The **UserModel.js** creates a **UserModel** singleton as part of the AngularJS app so it can easily be accessed from the controllers.
4. The **Nav.js** file uses the routing provider to examine the URL and determine where to go. On first load, it is assumed that the URL is not known, so the user is redirected to the login page. The **LoginCtrl** is initialized at this time.
5. The user can interact with the login page, enter a username and password, and click the login button.
6. The login button uses **ngClick** to execute the **onLogin()** method in the **LoginCtrl** file.
7. The **onLogin()** method will validate the inputs. If there is an error, it updates the related error variable which will in turn automatically show the error in the view. If there is no error then the authentication details are sent to service.
8. The service creates and returns a promise object. It validates the data, performs a service call (or simulates one in the mock service), and calls the **onLoginSuccess()** method.
9. The promise object in the **LoginCtrl** file triggers the **onLoginSuccess()** method when results are returned from the service.
10. The **onLoginSuccess()** method checks the response from the service. If the user did not authenticate, then an error is displayed. If the user's login was a success, then the user object is saved into the **UserModel** and the **$location** is used to direct the user to the main screen.

The next chapter will focus on loading tasks and displaying them in a DataGrid.

Chapter 3: Displaying the Tasks

This chapter will focus on loading the task data and displaying it to the user. A JavaScript grid library, **uiGrid**, will be used for display. We will also review the services needed to retrieve the data, and wire up everything to display data from the database. Since this is the first screen beyond the login screen, we will also implement code to make sure the user cannot view the tasks screen without logging in.

Create the User Interface

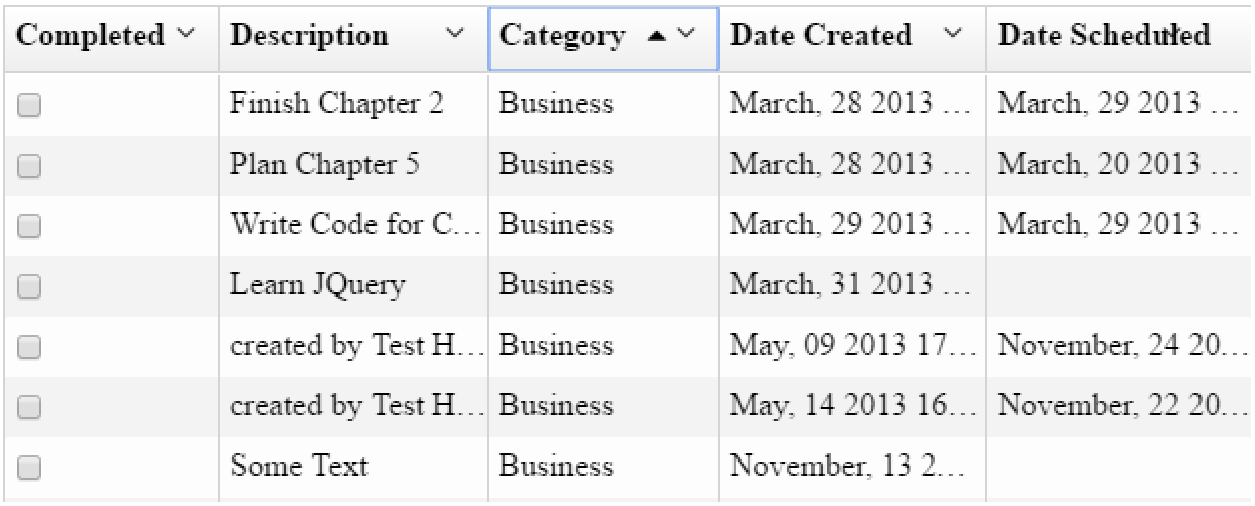
This section will show you how to create the grid within the AngularJS application. It will review the data to display in the grid, and show you how to create a **cellTemplate** to customize a column's display.

What Goes in the Grid?

This is a review of the data that needs to be displayed in the task grid. The data is what makes up a task for the context of this application. Here are the fields:

- **Category**: Tasks can be categorized and the grid will display the task category to the user.
- **Description**: This column will contain the details of the task.
- **Completed**: The grid will display whether or not the item was completed. This column will be displayed as a checkbox. If the item is checked, that means the item was completed. If the item is not checked, it is not completed. This will also provide an easy way for the user to mark the item completed. Chapter 7 will implement the ability to mark an item completed.
- **Date Created**: The grid will display the date that the task was created.
- **Date Scheduled**: The grid will display the date that the task was scheduled for. Chapter 5 will build the interface for scheduling tasks.

The final grid will look like this:



Completed	Description	Category ▲	Date Created	Date Scheduled
☐	Finish Chapter 2	Business	March, 28 2013 …	March, 29 2013 …
☐	Plan Chapter 5	Business	March, 28 2013 …	March, 20 2013 …
☐	Write Code for C…	Business	March, 29 2013 …	March, 29 2013 …
☐	Learn JQuery	Business	March, 31 2013 …	
☐	created by Test H…	Business	May, 09 2013 17…	November, 24 20…
☐	created by Test H…	Business	May, 14 2013 16…	November, 22 20…
☐	Some Text	Business	November, 13 2…	

Import the uiGrid Library and Styles

The grid component we will use in this application is named uiGrid. The **uiGrid** component is an AngularJS directive. It isn't part of the main AngularJS library, but is designed to be used with AngularJS. I am impressed with how capable it is.

The first step to implementing the grid into the Task Manager application is to add the new JavaScript imports into the **index.html** file. First, import the **uiGrid** library with a script tag:

```
<script
    src="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-grid/3.1.1/ui-grid.min.js">
</script>
```

The **uiGrid** import was placed after the angular library import and before the **learnWith** app definition. Because the **uiGrid** is not hosted on Google's CDN, I used a secondary CDN as the remote location of the grid.

The **uiGrid** library comes with some built-in style sheets explicitly for the grid. Include them, too:

```
<link
  href="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-grid/3.1.1/ui-grid.min.css"
  rel="stylesheet"
  type="text/css" />
```

I put this before the style sheet we created in the previous chapter. Our own style sheet, **style.css**, also needs two additions for the **uiGrid**. The first is some sizing of the actual grid:

```
.gridStyle {
    border: 1px solid rgb(212,212,212);
    width: 100%;
    height: 97vh;
}
```

These styles add a border around the grid and also specify the height and width. Width is set to "100%". However, the height is set to "97vh", which is 97% of the view port height. This expands the grid to fill all the available space.

The next CSS styles we need are these:

```
.ui-grid-row:nth-child(odd):hover .ui-grid-cell{background:lightblue}
.ui-grid-row:nth-child(even):hover .ui-grid-cell{background:lightblue}
```

This adds some visual appeal, so when the mouse rolls over a row in the grid, that grid will be highlighted

Add the Grid to the LearnWith Application

The next step is to tell our **learnWith** application about the grid library that we just imported. This can be done when the application is setup in the **LearnWith.js** file from the **com/dotComIt/learnWith/app** directory:

```
angular.module('learnWith', ['ngRoute','ui.grid','ui.grid.selection']);
```

Two new options were added to the module's option array. The first, **ui.grid**, is the main **uiGrid** directive. The second, **ui.grid.selection**, is an extension to the **uiGrid** that contains logic for selecting rows.

Next, you can open the **MainScreen.html** template. This is the view template displayed after the user successfully logs in. Erase everything we had previously put in the template, and replace it with this:

```
<div ui-grid="taskGridOptions" class="gridStyle" ui-grid-selection></div>
```

The **uiGrid** is placed on a **div** by using the **ui-grid** attribute. The value of this attribute is an object—**taskGridOptions**—that will be defined in the controller. The **ui-grid-selection** is a directive that enables this grid to have rows selected. The **div** also references the **gridStyle** CSS class created in our **style.css**.

Before we delve into the **taskGridOptions**, we need a **TaskModel** to store task information; such as the data to be displayed in the **uiGrid**.

Creating a TaskModel

The **TaskModel** will be an Angular service that will contain task information. I created a **TaskModel.js** file in the **com/dotComIt/learnWith/model** directory. Start by defining the service:

```
angular.module('learnWith').service('TaskModel', [function(){
   var taskModel = {
      tasks: []
   }
   return taskModel;
}]);
```

Conceptually, this is similar to the **UserModel** created in Chapter 2. The **service()** method is called on the **learnWith** module. The service is named **TaskModel**, and a function returns an object which will represent the **TaskModel** singleton. The only value in the **TaskModel** is an array named tasks. This will be used to fill the **uiGrid**. It is defaulted to an empty array, but will be replaced from data loaded by our service layer. Be sure to add a script tag to the **index.html** to import the **TaskModel** class:

```
<script src="com/dotComIt/learnWith/model/TaskModel.js"></script>
```

I put this file after the **UserModel** import, and before the **AuthenticationService** import.

The last step is to add the **TaskModel** into the controller for the main screen. Open up the **MainScreenCtrl.js** file and add the **TaskModel** to the controller definition:

```
angular.module('learnWith').controller('MainScreenCtrl',
   ['$scope','UserModel','TaskModel',
     function($scope,UserModel,TaskModel){
```

Finally, save the **taskModel** to the **$scope** for easy local access inside the controller, or the view template:

```
$scope.taskModel = TaskModel;
```

The app will also need the **UserModel**, and you'll notice that was injected into the **MainScreenCtrl**. You can save that to the **$scope** too:

```
$scope.userModel = UserModel;
```

This will be important in later chapters when we remove and add functionality based on the user's login role. The next step is to examine the grid's options.

Creating the Grid Options

taskGridOptions is a variable inside the **MainScreenCtrl**. It is referenced by the **uiGrid** directive inside the **MainScreen.html** template. The variable is an object that contains a lot of options about how the grid should work. First, create the object:

```
$scope.taskGridOptions = {
}
```

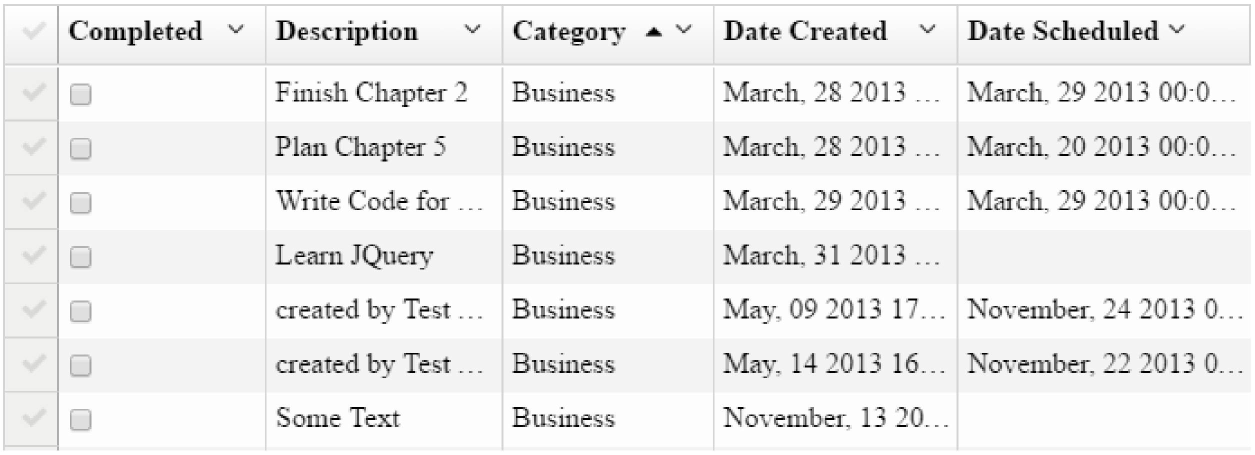
This is easy enough, and something you have seen many times already in this book. The first property to add to this object is named **data**. It contains an array of the objects that will be displayed in the grid:

```
data: 'taskModel.tasks',
```

The next set of options relate to how we can select items in the grid:

```
enableFullRowSelection :  true,
enableRowHeaderSelection : false,
multiSelect: false
```

The **enableFullRowSelection** tells the grid that a row can be selected by clicking on any cell in the row. Meanwhile, the **enableRowHeaderSelection** property is used to toggle the display of a row header, which can be used to select a row:



✓	Completed	Description	Category	Date Created	Date Scheduled
✓	☐	Finish Chapter 2	Business	March, 28 2013 …	March, 29 2013 00:0…
✓	☐	Plan Chapter 5	Business	March, 28 2013 …	March, 20 2013 00:0…
✓	☐	Write Code for …	Business	March, 29 2013 …	March, 29 2013 00:0…
✓	☐	Learn JQuery	Business	March, 31 2013 …	
✓	☐	created by Test …	Business	May, 09 2013 17…	November, 24 2013 0…
✓	☐	created by Test …	Business	May, 14 2013 16…	November, 22 2013 0…
✓	☐	Some Text	Business	November, 13 20…	

I chose to remove the gutter by setting **enableRowHeaderSelection** to false.

Finally, the **multiSelect** property is set to “false”. By default, this property is set to “true”. For the purposes of this app, I do not want to allow for multiple selection of items in the grid

The last item to define in our **taskGridOptions** is known as **columnDefs**, and it defines the details of the grid’s columns. If you leave this property out, then every property in the grid’s data objects will be added to the grid. You have a lot more power if you define them manually and individually.

We’ll start with our four simple columns:

```
columnDefs: [
    {field: 'description', displayName: 'Description'},
    {field: 'taskCategory', displayName: 'Category'},
    {field: 'dateCreated', displayName: 'Date Created'},
    {field: 'dateScheduled', displayName: 'Date Scheduled'},
]
```

The **columnDefs** is an array. Each element of the array is an object. In each object, we define a **field** property and a **displayName** property. The **field** property refers to the value from the **data** property's objects that will be displayed in the grid. The other property is the **displayName** property; which is used to populate the column header.

If you are paying very close attention, you'll notice that only four columns are defined above. The "completed" column is missing. This is because there is no default way to add a checkbox into a column in the **uiGrid**. I wanted to pay special attention to the checkbox.

Creating a Checkbox Cell Template

To create a checkbox column in the grid, we will need to create **cellTemplate**. This will reference an in-line or external HTML template that is used to draw the grid data.

A **cellTemplate** is just an HTML file. First we'll create the file, and then we'll see how to add it into the **uiGrid**. This is the **cellTemplate**:

```
<div class="ngSelectionCell">
    <input type="checkbox"
           class="ngSelectionCheckbox"
           ng-checked="row.entity.completed"
    />
</div>
```

I named this file **CompletedCheckBoxRenderer.html**, and put it in the **com/dotComIt/learnWith/views/tasks** folder. The renderer contains a **div** and is given the class **ngSelectionCell**. This refers to a CSS Class in the special **ui-grid.css** file that is part of the grid library and helps create a consistent look for the grid cells.

Inside the **div** is a simple input, with the type "checkbox". This is how you create a checkbox in HTML. The CSS class on the input also relates back to the **ui-grid.css** file. There is an Angular directive on the checkbox named **ngChecked** that is used to determine the value of the checkbox. Its value refers to a property in the **cellTemplate**'s entity. The entity relates to an object in the data array that populates the grid.

The last step here is to define the "completed" column definition as part of the **taskGridOptions** in the main screen controller:

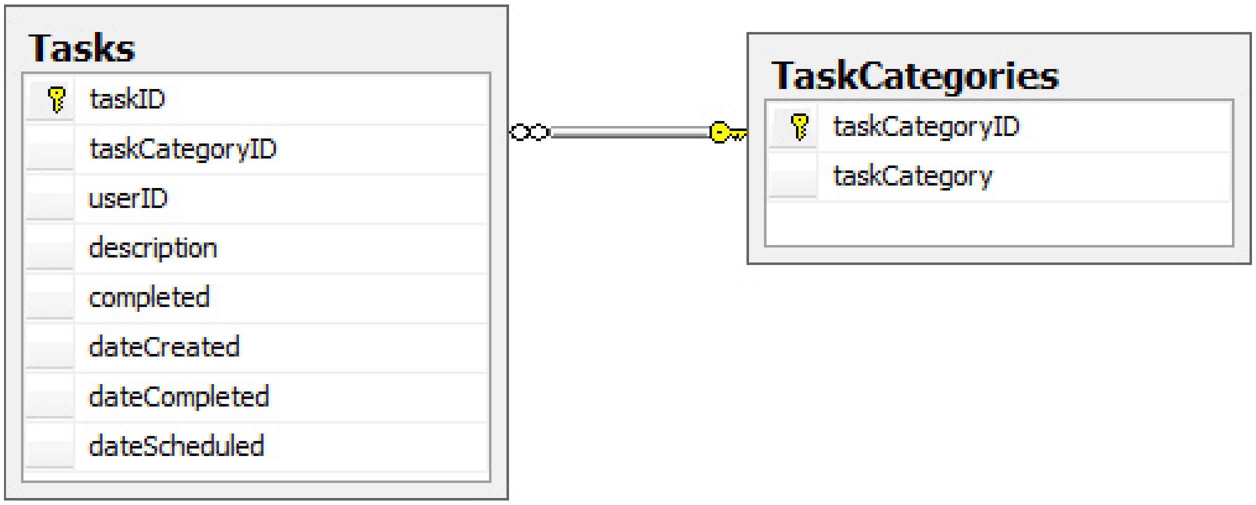
```
{
  field: 'completed',
  displayName: 'Completed',
  cellTemplate:'/com/dotComIt/learnWith/views/tasks/CompletedCheckBoxRenderer.html'
},
```

The column definition includes two properties you have seen before. The field relates to a property on the object populating the grid. The **displayName** property specifies the value to display in the column header. The next property is the **cellTemplate**, which specifies the renderer.

You can try to load the app now and log in. You won’t see any data yet, but you should see the grid outline and the column headers in place. Next we’ll review the service that we want to communicate with and finally we’ll wire up the service to the UI.

Examine the Database

The database behind this application is a SQL Server database. Two tables reside behind the tasks:



The data will support the UI. It also has a few additions to accommodate for internal values, such as the primary keys, which are not usually explicitly displayed to the user.

This is the data in the tables:

- **taskID**: This is the primary key for the task.
- **taskCategoryID**: This is the primary key for the task category. It represents a one-to-many relationship; meaning that a task can only be in a single category, but a category can have many tasks associated with it. This field shows up in two tables.
- **taskCategory**: This column contains the name of the category that a task was put into. A category from the **TaskCategories** table is connected to the **Tasks** table using the **taskCategoryID**.
- **userID**: This is the user's primary key. Although the user table is not shown in the above diagram, this column represents a one-to-many relationship between tasks and users. A user can have many tasks, but each task is associated with only a single user.
- **description**: This is the primary task text.
- **dateCreated**: This column is a date column that keeps track of when the task was created.
- **dateCompleted**: This column keeps track of when a task was marked as completed. Marking tasks complete will be detailed in a future chapter.
- **dateScheduled**: This column keeps track of when the task was scheduled. A future chapter will focus on how to schedule tasks for a specific date.

A query with a **join** can be used to select the data:

```
select * from tasks
join taskCategories on (tasks.taskCategoryID = taskCategories.taskCategoryID)
```

This query will return the data that is needed to populate the columns in our UI's **DataGrid**.

Write the Services

This section will explain the ColdFusion services that will be used to retrieve the task data from the database. It will create two ColdFusion value objects; one for the task, and one for the task filtering process. These objects will be used by ColdFusion before converting the data to JSON and returning it the UI. It will also create a new CFC—**TaskService**—and examine the process used to load the filtered tasks.

The Task Value Object

The task object represents a single task as returned from the database. The object will be converted to JSON and passed to the UI where it will represent a single row in the front end of the app.

I named this object **TaskVO** and put it in the **com/dotComIt/learnWith/vos** directory. This is the class:

```
<cfcomponent displayname="TaskVO" hint="I represent a single task."
            alias="com.dotcomit.learnWith.vos.TaskVO">
      <cfproperty name="category" type="String" />
      <cfproperty name="completed" type="Boolean" />
      <cfproperty name="dateCompleted" type="date" />
      <cfproperty name="dateCreated" type="date" />
      <cfproperty name="dateScheduled" type="date" />
      <cfproperty name="description" type="String" />
      <cfproperty name="taskCategoryID" type="numeric" />
      <cfproperty name="userID" type="numeric" />
</cfcomponent>
```

This class is a **CFComponent**, as are all classes in ColdFusion. It lists all the properties in alphabetical order. Each property mirrors a database column. This value object is the container for a single task in the context of our application.

The TaskFilter Value Object

Where the **TaskVO** class is the output of our method to retrieve task data, the **TaskFilterVO** is the input into the same method. It includes various bits of information that will be used to filter the result set. It is not usually practical for an application to load up and retrieve every single piece of data it has. A way to limit the data to the pertinent information the user wants to see is an important consideration.

To determine how the method for retrieving tasks should limit the data, this app will use a value object. I named this object **TaskFilterVO** and put it in the **com/dotComIt/learnWith/vos** directory. The CFC class starts similar to what you saw for the **TaskVO** object in the previous section:

```
<cfcomponent displayname="TaskFilterVO"
             hint="I represent filter criteria for retrieving tasks."
             alias="com.dotcomit.learnWith.vos.TaskFilterVO">
</cfcomponent>
```

What properties are needed inside this task object? For starters, the user may want to filter on the category:

```
<cfproperty name="taskCategoryID" type="numeric" />
```

The **taskCategoryID** is the best way to filter a query based on the category.

Next, the user may want to filter on whether tasks were completed, or not:

```
<cfproperty name="completed" type="numeric" />
```

Even though "completed" is a bit field in the database, I did not make the "completed" filter property a Boolean. When we write the code in the service to query the database, you'll need to know whether the "completed" value was defined or not. If it isn't defined, you can return tasks that are complete and incomplete. Unfortunately, Boolean values are always true or false. There is no undefined state for Boolean variables. By making the completed field as a number, we'll be able to easily tell if the value is undefined or not. A "0" value returns incomplete tasks. A "1" value returns complete tasks. Any other value will return tasks that are both incomplete and complete.

One easy way to focus the UI on data that the user wants to see is by date filtering. This class has an **endDate** property and a **startDate** property for this purpose. When the query is performed, we can look for created dates later than or equal to the start date, or earlier than or equal to the end date:

```
<cfproperty name="endDate" type="date" />
<cfproperty name="startDate" type="date" />
```

Another important element to filter against is the scheduled date. A user may want to see all the tasks scheduled for today or all tasks scheduled for this month. These properties are used to allow for this functionality:

```
<cfproperty name="scheduledEndDate" type="date" />
<cfproperty name="scheduledStartDate" type="date" />
```

The **scheduledStartDate** and **scheduledEndDate** properties are used to define a range of tasks.

The Task Service

The **TaskService** class is going to be used for a variety of actions throughout the UI. For now, we will create a method—**getFilteredTasks()**—for retrieving the task data. The method will accept a **TaskFilterVO**, in JSON form, as the input argument. The method will output an array of **TaskVO** objects wrapped in a **ResultObjectVO** instance. The output will be converted to JSON before being returned.

Since this is the first time in this book where the **TaskService** has come up, we can create the file in the **com/dotComIt/learnWith/services** directory.

```
<cfcomponent displayname="TaskService"
             hint="I handle task related service calls">
</cfcomponent>
```

Inside the CFC there is a method declaration:

```
<cffunction name="getFilteredTasks" returnformat="plain" access="remote"
            output="true"
            hint="I retrieve tasks from the db that meet the criteria.">
  <cfargument name="filter" type="String" required="true"
              hint="I am a JSON Packet that represents a TaskFilterVO"/>
</cffunction>
```

The function name is **getFilteredTasks()**. The **returnFormat** is plain, which means simple text will be returned from the method. The access is remote, which means the function can be accessed by applications other than ColdFusion, such as our AngularJS application. There is one argument to this method which is a filter. The argument will be sent in as JSON, so the argument type is a string. However, the method assumes that the argument will be a JSON instance with properties from the **TaskFilterVO** class.

The first step in the method is to convert the filter argument from JSON into something native to ColdFusion. ColdFusion's **deserializeJSON()** function will do the job:

```
<cfset local.json = deserializeJSON(arguments.filter)/>
```

The second and most complicated step in this method is to create the database query with all the various conditions based on properties defined in the **TaskFilterVO**. First, before jumping into the query code, create two variables:

```
<cfset setWhere = 'true' />
<cfset firstOne = 'true' />
```

The **setWhere** variable will be used to determine if the "where" clause needs to be added to the query. The **firstOne** variable will be used to determine if the condition being added to the query is the first condition. If it isn't the first condition, then an "and" will be added between two statements.

Here comes the first part of the query:

```
<cfquery datasource="#application.dsn#" name="local.dataQuery" >
      select *
      from tasks join taskCategories on (tasks.taskCategoryID =
                                          taskCategories.taskCategoryID)
```

Primarily, this is the same query that was shown earlier in this chapter. It is preceded by a **cfquery** tag, which is the ColdFusion tag for querying a database. Next up, comes the query filters. Only the filters for "completed" and **startDate** will be implemented now, with the rest being filled in for later chapters.

The check for the "completed" flag is complex. Though we built it as a numerical field in the **TaskFilterVO**, the competed column in the database is a bit column. In order to avoid problems by sending a numeric value to a database bit column, I sent a hard-coded value to the database instead:

```
<cfif StructKeyExists(local.json,'completed')>
      <cfif setWhere is true >where <cfset setWhere = 'false'></cfif>
      <cfif firstOne is true><cfset firstOne = false><cfelse>and</cfif>
      <cfif local.json['completed'] is 0>
            completed = <cfqueryparam cfsqlType="cf_sql_bit" value="0">
      <cfelse>
            completed = <cfqueryparam cfsqlType="cf_sql_bit" value="1">
      </cfif>
</cfif>
```

The code checks the **local.json** variable to verify that the completed property exists. If it doesn't exist, then the query doesn't need to check it, and will thus return all tasks. If it does exist, then the code does need to include a completed comparison in the query. The code checks the **setWhere** value. If the value is “true” it adds the “where” statement and sets the **setWhere** to “false”. On the other hand, if the value is already “false”, nothing is added. The next line checks for **firstOne**. If the value is “true”, the code sets it to “false”. Otherwise the code adds the “and” statement to the clause. The final section of this segment actually creates the completed condition. If the completed value in the converted JSON packet is “0”, then send a “0”—false—in the database for the completed value. Otherwise, assume it is “true” and send a “1”.

The date fields are handled using a similar approach. For the moment we only care about the **startDate** filter property, which compares against the **dateCreated** column.

```
<cfif StructKeyExists(local.json,'startDate')>
      <cfif setWhere is true >where <cfset setWhere = 'false'></cfif>
      <cfif firstOne is true><cfset firstOne = false><cfelse>and</cfif>
      dateCreated >= <cfqueryparam cfsqltype="cf_sql_date"
                             value="#local.json['startDate']#">
</cfif>
```

This code checks if the **startDate** is defined. If it isn’t, then no condition needs to be added to the query. If it does, then the condition is added. The same **setWhere** and **firstOne** checks are added. Then the **dateCreated** is compared against the **startDate** using a greater than or equal condition.

This is the end of the query:

```
      order by dateCreated
</cfquery>
```

The data is ordered by the **dateCreated** and the **cfquery** tag is closed.

The third step in this method is to process the database results. The first step is to create an instance of the **resultObjectVO**:

```
<cfset local.resultObject =
          createObject('component','com.dotComIt.learnWith.vos.ResultObjectVO')/>
<cfset local.results = arrayNew(1) />
```

A local array is also created to contain the array of **TaskVO** Objects.

Next, the query results will be looped over and a **TaskVO** object will be created for each record. Each **TaskVO** will be added to the results array:

```
<cfoutput query="local.dataQuery">
  <cfset local.tempObject =
          createObject('component','com.dotComIt.learnWith.vos.TaskVO')/>
  <cfset local.tempObject['taskID'] = dataQuery.taskID />
  <cfset local.tempObject['taskCategoryID'] = dataQuery.taskCategoryID />
  <cfset local.tempObject['taskCategory'] = dataQuery.taskcategory />
  <cfset local.tempObject['userID'] = dataQuery.userID />
  <cfset local.tempObject['description'] = dataQuery.description />
  <cfset local.tempObject['completed'] = dataQuery.completed />
  <cfset local.tempObject['dateCreated'] =
                            dateFormat(dataQuery.dateCreated,"mm/dd/yyyy") />
  <cfset local.tempObject['dateCompleted'] =
                          dateFormat(dataQuery.dateCompleted,"mm/dd/yyyy") />
  <cfset local.tempObject['dateScheduled'] =
                          dateFormat(dataQuery.dateScheduled,"mm/dd/yyyy") />
  <cfset arrayAppend(local.results,local.tempObject)/>
  <cfset local.resultObject['error'] = 0/>
</Cfoutput>
```

In the previous block of code, the database record set—**local.dataQuery**—is looped over using **cfoutput**. For each item in the resulting query, a **TaskVO** is created. The dates are configured into the standard US-based date formatting and the times are removed. Otherwise the code performs no special processing.

The final step in this method is to turn the database results into JSON and return them to the calling service:

```
<cfset local.resultObject['resultObject'] = local.results />
<cfset local.resultString = convertToJSON(local.resultObject) />
<cfreturn local.resultString>
```

First, an instance of the **ResultObjectVO** is created. Next, the results array is added to it as the **resultObject**. Lastly, the **resultObject** is converted to JSON using a custom function. This is the function:

```
<cffunction name="convertToJSON" returnformat="plain" access="public">
  <cfargument name="value" type="any" required="true"/>
  <cfset local.resultString = SerializeJSON(arguments.value) />
  <cfreturn local.resultString>
</cffunction>
```

The function converts the value to JSON using the **SerializeJSON** string. This method converts the returned JSON string, which the main method returns to the calling service. The final line of the method closes the **cffunction** tag that started things in the beginning.

Testing the TaskService

I created a bunch of different test scripts that can be used to test the aspects of the **getFilteredTasks()** function, but will only share two in this book. The code is in the **getFilteredTasks.cfm** test in the **com/dotComIt/learnWith/tests** directory. First create the service:

```
<cfset service =
 createObject('component','com.dotComIt.learnWith.services.TaskService')>
```

The first test I'll show you is one to get all the tasks that have not yet been completed:

```
<h1>Get Not Completed Tasks</h1>
<cfset tempObject =
        createObject('component','com.dotComIt.learnWith.vos.TaskFilterVO')/>
<cfset tempObject.completed = 0 />
<cfset resultString = SerializeJSON(tempObject) />
<cfset results = service.getFilteredTasks(resultString)> <Br/>
Results: <Br/>
<cfdump var="#results#" expand="false" />
```

The test creates an instance of the **TaskFilterVO**. It sets the completed property on it to “0”. Then it turns the filter object into JSON, and calls the **getFilteredTasks()** method on the service. It outputs the results, a JSON String, using the **cfdump** tag:

```
{
 "resultObject":[
  {
   "taskcategoryID":2,
   "description":"Get Milk",
   "taskcategory":"Personal",
   "dateScheduled":"03\/29\/2013",
   "dateCompleted":"",
   "taskID":1,
   "dateCreated":"03\/27\/2013",
   "completed":0,
   "userID":1
  },
  {
   "taskcategoryID":1,
   "description":
   "Finish Chapter 2",
   "taskcategory":"Business",
   "dateScheduled":"03\/29\/2013",
   "dateCompleted":"",
   "taskID":2,
   "dateCreated":"03\/28\/2013",
   "completed":0,
   "userID":1
  },
  {
   "taskcategoryID":1,
   "description":"Plan Chapter 5",
   "dateScheduled":"03\/20\/2013",
   "taskcategory":"Business",
   "dateCompleted":"",
   "taskID":5,
   "dateCreated":"03\/28\/2013",
   "completed":0,
   "userID":1
  }
 ],
 "error":0.0
}
```

The other test I want to show is one that gets tasks created after a certain date. The code behind this is very similar:

```
<h1>Get Tasks After Date</h1>
<cfset tempObject =
    createObject('component','com.dotComIt.learnWith.vos.TaskFilterVO')/>
```

```
<cfset tempObject.startDate = createDate(2013,3,29) />
<cfset resultString = SerializeJSON(tempObject) />
<cfset results = service.getFilteredTasks(resultString)> <Br/>
Results: <Br/>
<cfdump var="#results#" expand="false" />
```

Once again, a **TaskFilterVO** object is created. The **startDate** is specified, as that is the data used to specify the results of this test. The **TaskFilterVO** is serialized into JSON. The **getfilteredTasks()** method is called on the **TaskService**. The results are displayed on screen:

```
{
 "resultObject":[
  {"taskcategoryID":1,
   "description":"Write Code for Chapter 3",
   "dateScheduled":"03\/29\/2013",
   "taskcategory":"Business",
   "dateCompleted":"",
   "taskID":3,
   "dateCreated":"03\/29\/2013",
   "completed":0,
   "userID":1
  },
  {"taskcategoryID":1,
   "description":"Write Chapter 4",
   "taskcategory":"Business",
   "dateScheduled":"03\/20\/2013",
   "dateCompleted":"",
   "taskID":4,
   "dateCreated":"03\/30\/2013",
   "completed":0,
   "userID":1
  },
  {"taskcategoryID":1,
   "description":"Learn JQuery",
   "taskcategory":"Business",
   "dateScheduled":"",
   "dateCompleted":"",
   "taskID":6,
   "dateCreated":"03\/31\/2013",
   "completed":0,
   "userID":1
  }
 ],
 "error":0.0
}
```

The various other tests in the file are used to validate the other aspects of the filtering.

Review the API

To integrate this service into the JavaScript application, you need to know the endpoint and the method arguments. If the app and service are served off the same domain, you can use a relative URL for the endpoint. In this case, I used this:

```
/com/dotComIt/learnWith/services/TaskService.cfc
```

Calls to the API will need to pass two separate parameters:

- **method**: This parameter is the name of the method to be called at the endpoint. In this case the value should be set to “getFilteredTasks”.
- **taskFilterVO**: The second parameter will be a JSON packet representing a **TaskFilterVO** object.

The Angular code will use this endpoint, and parameters, to call the service method to load the application’s data.

Access the Service

This section will discuss how to create the AngularJS code needed to access the application's backend services.

Turning a JavaScript Object into a JSON String

Before looking at the method for loading tasks, I want to review a few important points. First, in Chapter 2, I wrote about setting the proper HTTP header so that the variables would show up on the server side as form variables. I made mention of a blog post which spoke of creating an HTTP **transformRequest** function to universally set the values. I have set up a **transformRequest** in the **HTTPServiceConfig.js** file to handle this conversion. I'm not going to show you the full code here as it is just something I borrowed from the interwebs.

The second thing I want to remind you is that our service is expecting a filter string which represents a JSON object. If we run a JavaScript object through the transform function it will be treated as an object and not a string. So, before sending the object parameter as part of the call, it must be converted.

To do this, I created a new JavaScript library, named **Utils.js**. It is in the **com/dotComIt/learnWith/utils** directory. This will contain an Angular service. Here is the outline:

```
angular.module('learnWith').service('SharedUtils', [function(){
    var utils = {
    }
    return utils;

}]);
```

The service accesses the **learnWith** module and calls the **service()** method on it. The service is named **SharedUtils** and the service function does not need any other services. The service creates, and returns, an object to contain the utility function.

Only a single method is required for the **SharedUtils** service, named **objToJSONString**. Add it to the **utils** object like this.

```
objToJSONString : objToJSONString
```

Define the method after the return. Here is the method signature:

```
function objToJSONString (obj) {
```

It accepts a single argument—"obj"—which is the object which needs to be converted into a JSON string. The first line of the method creates a string to contain the results:

```
var str = '';
```

Next we need to look over all the properties in the argument object:

```
for (var p in obj) {
```

Each property name is enclosed in double quotes. A colon is used to separate the property name and the property value. After the colon, the object's value is displayed, also surrounded by quotes. To include the quote as part of the finished string; the forward slash ('\') had to be used to escape the quotes:

```
 str += "\"" + p + "\":\"" + obj[p] + "\",";
}
```

The end curly bracket closes off the for loop.

Each property/value pair combination in a JSON string is separated by a comma. The loop above will add a comma after every property. At the end of the loop, our string will have an extra comma at the end. The next operation of this method is to remove the final comma from the string.

```
if(str.length > 0){
    str = str.substr(0,str.length-1)
}
```

It checks the length of the result string. If the length is listed as greater than "0"; then we can assume that the object had at least one property and there is a comma at the end. Some string processing removes it.

Finally, return the result string, adding the curly brackets on each end of it:

```
return '{' + str + '}';
}
```

I found this method useful when dealing with the back end. In its current state, this method will not support nested objects. However, it should be able to be modified easily enough if that is needed.

Be sure to include this file in your **index.html**:

```
<script src="com/dotComIt/learnWith/utils/Utils.js"></script>
```

I know I've said order of the imports in JavaScript is important. In this case, our **Utils.js** file has a dependency on the Angular framework, so it must be placed after the AngularJS script import. I put it at the end.

Accessing the loadTask( ) Service

This section will create a **TaskService** file that will integrate with the **getFilteredTasks()** method from the ColdFusion service. I put the **TaskService.js** file in the **com/dotComIt/learnWith/services/coldFusion** directory. This file will create an Angular service to contain all task related functions. It is a parallel to the ColdFusion **TaskService.cfc**.

This is the file's outline:

```
angular.module('learnWith').service('TaskService',
   ['$http', '$filter', 'SharedUtils',function($http,$filter,sharedUtils){
       var services = {
       }
       return services;
   }]
);
```

The first few line references the **learnWith** Angular module. It calls the **service()** function off the module instance to create our custom service. The **service()** function accepts two values; a name—**TaskService**—and an options array. The options array is used to inject the **$http** service, the **$filter** service, and the **sharedUtil** service into the anonymous service function. The service function creates an object named **services**, and returns it. This is the object that Angular will create and store; an instance of when the **TaskService** is referenced in other controllers. The **$http** service will be used to make calls to ColdFusion. The **$filter** will be used for date formatting in future chapters. The **sharedUtils** service will be used to convert the **TaskFilterVO** to JSON before passing it to ColdFusion.

The services object does not have anything in it yet, so let's add a **loadTasks()** function:

```
var services = {
  loadTasks : loadTasks
}
```

After the return statement, create the **loadTasks()** function. A single argument is passed to **loadTasks()**, which is a **taskFilter** object :

```
function loadTasks(taskFilter){
  var parameters = {
      method : "getFilteredTasks",
      filter : sharedUtils.objToJSONString(taskFilter)
  }
  return $http.post('/com/dotComIt/learnWith/services/TaskService.cfc',
              parameters )
}
```

The first step in this method is to create a parameter object. The first element of the parameter object will be the method argument, **getFilteredTasks**, to send to the remote service. The second argument is the JSON filter object. The **objToJSONString()** method is run on the **TaskModel**'s **taskFilter** object in order to get the JSON string.

Then the HTTP post is called; specifying the endpoint for the **TaskService** service and adding the parameters object. The HTTP **transform** function will convert the parameters object to something that the service will recognize. When this method is executed from the controller, the service is called. Then either the **failure** or **result** method is executed based on the results from the **post()** call.

To use the service, open up the **main index_ColdFusion.html** file and add this tag:

```
<script src="com/dotComIt/learnWith/services/coldFusion/TaskService.js">
</script>
```

ColdFusion should be all setup now.

Wire Up the UI

This section will show you how to load the task data from the remote service and display it in the **uiGrid**. Also, it will add in a security check so users can't open the tasks screen without first logging in.

Validate the User before Loading Data

This section will show you how to check if the user is logged in or not before attempting to load the data. To determine whether the user is logged in, you can check the **userID** property of the user object in the **UserModel**. If the **userID** is "0", then the user has not logged in. If it is anything else, then the user has logged in fine.

You can add the method **validateUser()** to the **MainScreenCtrl.js** file:

```
function validateUser(){
    if($scope.userModel.user.userID == 0){
        return false;
    }
    return true;
}
```

This method performs the check. If the **userID** is "0"; then it returns false. Otherwise, it returns true.

Now we want to call this method when the view initially loads. In JavaScript, any code that is not in a function will execute as it is found. So, to run a method during the initial setup of the main screen, you just have to put the function call as part of the main screen controller. First, let's create an **onInit()** function:

```
function onInit (){
    if(!validateUser()){
        $location.path( "/login" );
    } else {
        loadTasks();
    }
}
```

Then, execute it:

```
onInit();
```

The **onInit()** method calls the **validateUser()** method. If the user does not validate, it uses the **$location** service to redirect the **routeProvider** back to the login screen. The **$location** service is an AngularJS service that can be used to redirect the user in the application. You'll have to modify the **MainScreenCtrl** controller to inject the **$location**.

```
angular.module('learnWith').controller('MainScreenCtrl',
    ['$scope','$location','UserModel','TaskModel','TaskService',
      function($scope,$location,UserModel,TaskModel,TaskService){
```

If the user does validate properly, then a method called **loadTasks()** is called.

Loading the Tasks

The final step is to load the tasks. Go back to the **MainScreenCtrl.js** file and create the **loadTasks** method:

```
function loadTasks(){
  TaskService.loadTasks($scope.taskModel.taskFilter)
              .then(onTaskLoadSuccess, onTaskLoadError);
}
```

This method calls the **loadTasks()** in the **TaskService** object. The argument is the **taskFilter** object, which is defined in the **taskModel**. A promise object is returned, and the **then()** function is used to define functions to be called on a successful result and on a failed result.

First, the error method; **onTaskLoadError()**:

```
function onTaskLoadError(response){
    alert(response.data);
}
```

This method merely displays the error to the user using the JavaScript alert. We did something almost identical in Chapter 1 when authenticating the user.

The successful result method— **onTaskLoadSuccess()**— is slightly more interesting:

```
function onTaskLoadSuccess(response){
    if(response.data.error == 1){
        alert("We could not load the task data");
        return;
    }
    $scope.taskModel.tasks = response.data.resultObject;
}
```

The **onTaskLoadSuccess()** method first examines at the returned data object. If the error field of the data object is one, then it displays a message to the user about the error and stops processing the method. If there is no error, then it will instead store the **resultObject** in the **taskModel.tasks** variable. The **resultObject** should contain an array of task objects. When the **TaskModel** is updated, the **uiGrid** updates as well, displaying the data.

Final Thoughts

This chapter should have given you an understanding of how to use the **uiGrid** component with an AngularJS application. It also integrated with the **getFilteredTasks()** service method, and you learned how to convert a JavaScript object into a JSON string.

This summarizes the actions from within this chapter:

1. The app loads and a **taskModel** instance is created.
2. Then user logs in. After successful login, the **MainScreen.html** template loads and **MainScreenCtrl** controller is initialized with the **taskModel**. Along with the **MainScreenCtrl** initialization, the **taskGridOptions** are created.
3. The **onInit()** method is executed on the **MainScreenCtrl**. It validates that the user has indeed logged in. If not, it redirects them to the login page. If the user has successfully logged in, it calls the **loadTasks()** method to call the remote service to load the tasks.
4. The **loadTasks()** service method is called with the **taskFilter** parameters initially set up in the **TaskModel**. The **taskFilter** parameter is converted to a string before calling the remote method.
5. The tasks are filtered, and the success method is executed. The success method stores the returned tasks in the **TaskModel.tasks** array.
6. Using binding, the **uiGrid** will populate itself with the rows in the MainScreen.html template.
7. The next chapter will focus on creating the filter which will allow for viewing different tasks based on different criteria.

The next chapter will focus on creating the filter which will allow for viewing different tasks based on different criteria.

Chapter 4: Filtering the Tasks

This chapter will demonstrate how to filter the list of tasks in the application. It will create the User Interface to allow for filtering tasks and review the services that power it. This chapter will modify the **loadTasks()** service method covered in the previous chapter and introduce a new service to load the task categories.

Create the User Interface

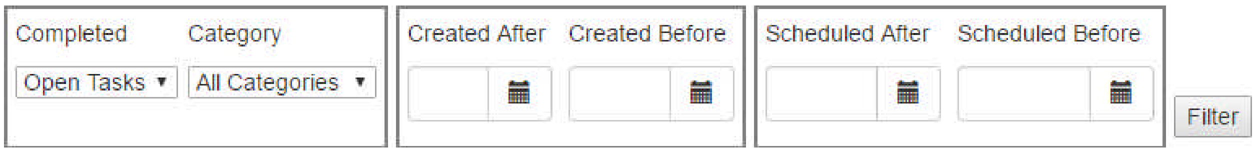
This section will review the user interface that we're going to build and then show you how to build it. It will cover how to populate a select box's data using AngularJS and will also introduce UI Bootstrap, in order to add a **DateChooser** to the application.

What Data Do We Filter On?

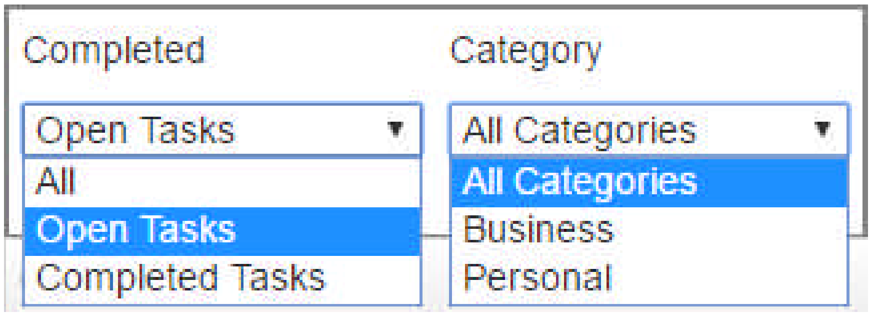
Chapter 3 integrated with a service method named **loadTasks()** to populate the **uiGrid**. At the time, we only loaded data with some default values; a completed property and a start date. However, there are more fields we want to allow the user to implement to filter the task grid:

- **Category**: The user should be able to filter the data on a specific category. Perhaps they want to see all the tasks they can do at home, all the work-related tasks, all the business tasks, everything related to clothes shopping, or however they decide to categorize their tasks.
- **Date Scheduled**: The user should be able to filter the data based on the date that the task was scheduled for. Two properties in the **TaskFilterVO** relate here; **scheduledStartDate** and **scheduledEndDate**.
- **Completed**: The user should be able to filter completed tasks, or incomplete tasks. In Chapter 3, we set the default of this property to "0"; assuming the user would want to see tasks which have not yet been completed.
- **Date Created**: The user should be able to filter on the date that a task was created. This relates to two properties; the **startDate** and **endDate**

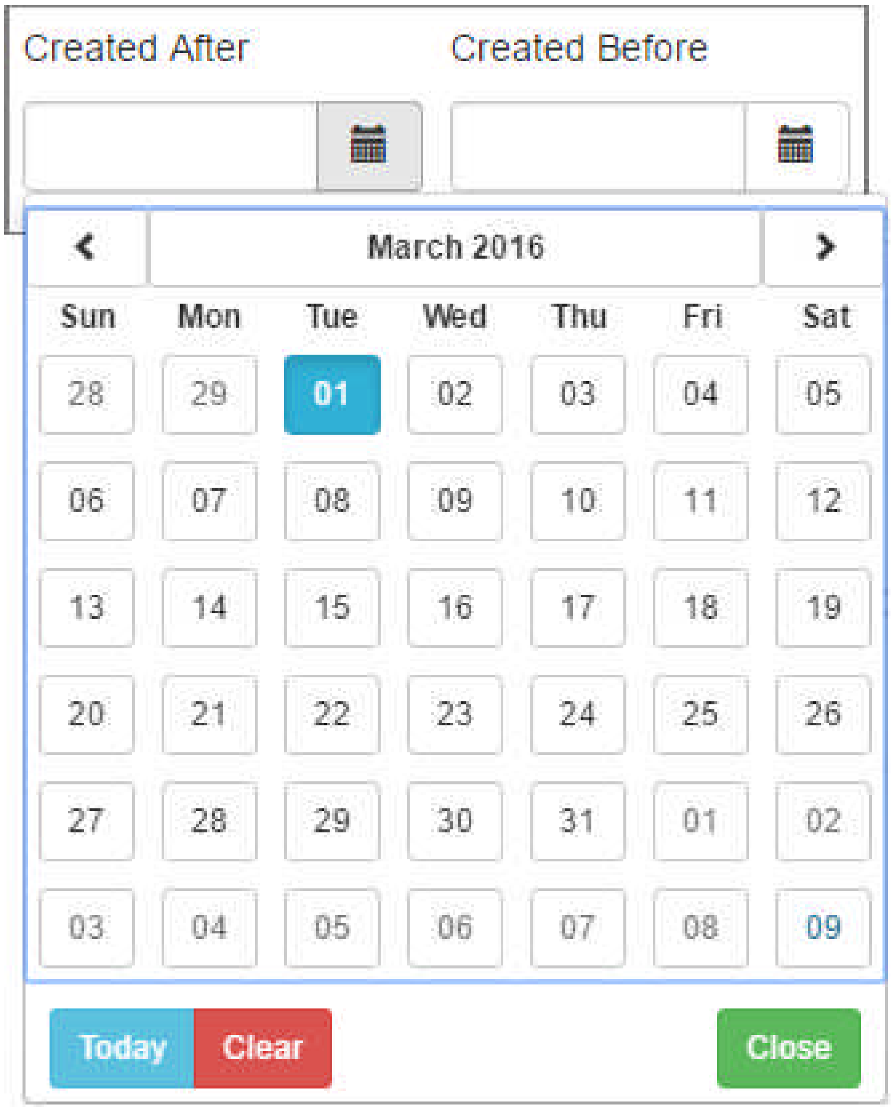
This is what the final UI will look like:



This grouping will be placed above the **uiGrid**. The completed select box will be populated with hard-coded data, but the category select box will be populated with a service call:



This is the date chooser, implemented as a popup:



These represent the UI we'll build throughout this chapter.

Using Angular Includes in the MainScreen

I was looking at the code thus far, and the code we still have to write, and realized that the **MainScreen** would become very complicated with all the code inside it. I looked for a way to encapsulate functionality inside our AngularJS application. In order to simplify the **MainScreen.html** template, I decided to use Angular's **ngInclude** directive. The **ngInclude** directive allows you to import one HTML template into another.

First, I'm going to move the **uiGrid** into its own sub template. The template is named **TaskGrid.html**, and is in the **com/dotComIt/learnWith/views/tasks** folder. The template has a single line:

```
<div ui-grid="taskGridOptions" class="gridStyle" ui-grid-selection></div>
```

It simply copies the code that was in **MainScreen.html** into a new file. The modified **MainScreen.html** must now include a template:

```
<div ng-include="'com/dotComIt/learnWith/views/tasks/TaskGrid.html'">
</div>
```

It uses a **div**, with the **ngInclude** directive. The value of the directive is the location of the **TaskGrid.html** file. With this simple approach we can break the **MainScreen.html** into multiple discrete files with a more streamlined purpose.

Next, you can create an include file for the Filter UI that we'll create in this chapter. I named the file **TaskFilter.html** and put it in **com/dotComIt/learnWith/views/tasks** folder. At the moment the file has nothing inside it, but here is the include directive:

```
<div ng-include="'com/dotComIt/learnWith/views/tasks/TaskFilter.html'">
</div>
```

We could assign a different controller to an **ngInclude** using an **ngController** directive, but I decided to rely on the **MainScreenCtrl** for this purpose.

It is important to note that the use of the **ngInclude** will create a sub-scope of the controller's scope. Simple **$scope** variables created in the **MainScreenCtrl** will not update if accessed or bound to variables inside an included file. However, objects inside the **MainScreenCtrl** will be inherited into the include file's new scope. To accommodate for this we had to make a few changes to the code in our **MainScreenCtrl**, primarily related to the **TaskModel** instance.

In the original code, we had the following:

```
$scope.taskModel = TaskModel;
```

When using **ngIncludes**, the **taskModel** will not be accessible inside the include file because it is not an object. In order to set defaults of values inside the include files, we must nest it inside an object. Then JavaScript will expose the value to inheritance. This is the modified code:

```
$scope.taskModelWrapper = {
    taskModel : TaskModel
}
```

All references to the **taskModel** instance variable must be changed within **MainScreenCtrl** to reference this change. The **onTaskLoadSuccess()** method must be changed in how it stores the results of a successful call to load the task objects:

```
$scope.taskModelWrapper.taskModel.tasks = data.resultObject;
```

The **taskGridOptions** object must be modified to reflect the new structure of the data:

```
data: 'taskModelWrapper.taskModel.tasks',
```

Since the **taskGridOptions** variable is already an object, it will already be properly inherited, and accessible by the new **TaskGrid.html** template.

Populating A Select with Angular

There are two drop down lists that need to be added to the **TaskFilter.html** template; one for completed tasks, and one for categories. Completed tasks will use a hard-coded data source, and the category drop down will be populated from a service call. In both cases, the data source will be populated with a property from the **TaskModel**.

This is the **dataProvider** for the completed drop down:

```
taskCompletedOptions : [
    {"id":-1,"label":"All"},
    {"id":0,"label":"Open Tasks"},
    {"id":1,"label":"Completed Tasks"}
]
```

It is an array with three elements, each one an object and containing a label and an ID. Since the **taskCategories** won't be loaded yet, we'll just populate that with an empty array:

```
taskCategories : []
```

Moving onto the **TaskFilter.html** template, we can use a table to layout items. The first row of the table will contain the headers and the second row will contain the input elements.

This is the start of the table, containing just the top row, and the headers for the first two drop down lists:

```
<table>
    <tr>
        <td>Completed</td>
        <td>Category</td>
    <tr/>
</table>
```

The second row will contain the select boxes. Before we jump into that, I want to refresh your memory on how a normal select box would be populated in HTML:

```
<select>
    <option value="-1">All</option>
    <option value="0">Open Tasks</option>
    <option value="1">Completed Tasks</option>
</select>
```

The top level tag is the **select**. Each option in the drop down is defined with an **option** tag. The text between the open and close option is displayed in the drop down list of the UI. The **value** is something that can be accessed through JavaScript.

When creating a select box in Angular, the approach is slightly different:

```
<tr>
 <td>
   <select ng-model="taskModelWrapper.taskModel.taskFilter.completed"
           ng-options="item.id as item.label for item in
                          taskModelWrapper.taskModel.taskCompletedOptions">
    </select>
</td>
```

The first property to the select box is an **ngModel**. This syncs the selected value in the select box with a variable in the **$scope**. In this case, the completed property of the **taskModel**'s **taskFilter** objects. Remember, in order for the **taskModel** to be accessible in the include file it had to be wrapped in an object.

The **ngOptions** directive is up next. This directive is used to create the option tags. We do not create the option tags manually. This is the directive's value:

```
item.id as item.label for item in
taskModelWrapper.taskModel.taskCompletedOptions
```

This is like a loop, but has more meaning. An option tag will be created with the display text of **label** for every item inside the **taskCompletedOptions** array. When an item is selected, the **id** field from the **taskCompletedOptions** object will be bound to the **ngModel** value.

The drop down for the categories operates in a similar manner:

```
 <td>
  <select ng-model="taskModelWrapper.taskModel.taskFilter.taskCategoryID"
          ng-options="item.taskCategoryID as item.taskCategory for item in
                                  taskModelWrapper.taskModel.taskCategories">
  </select>
 </td>
</tr>
```

This select box operates in the exact same manner that the previous one did. However, it has a different list of categories, and binds the selected value to a different variable within the **taskModel**.

Adding a DateChooser

If you've done HTML Development, you probably know that a **DateChooser** is not a native HTML control. I have come to like the UI Bootstrap library. It has a **DateChooser**-type component named **DatePicker**. UI Bootstrap is an Angular library that makes use of Twitter Bootstrap. Bootstrap is a common framework in HTML5 development, and UI Bootstrap includes other functionality that will be useful later in this book.

The first step is to include the **Bootstrap.css** file and the UI Bootstrap modular in the **index.html** of this application:

```
<link
    href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.3.6/css/bootstrap.min.css"
    rel="stylesheet"
    type="text/css" />
```

I put this file after the **ui-grid.css**. As with other libraries we've used, I am pulling this out of a CDN. The next step is to include the UI Bootstrap library:

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-bootstrap/1.3.1/ui-bootstrap-tpls.min.js">
</script>
```

I placed the UI Bootstrap script tag between the **uiGrid** script tag and the **learnWith** module script. I want to note that this app is using Bootstrap 3.3.6; the current version at the time of this writing.

The next step is to make UI Bootstrap available to the Angular module. Open up the **LearnWith.js** file in the **com/dotComIt/learnWith/app** directory:

```
angular.module('learnWith',['ngRoute','ui.grid','ui.grid.selection',
'ui.bootstrap']);
```

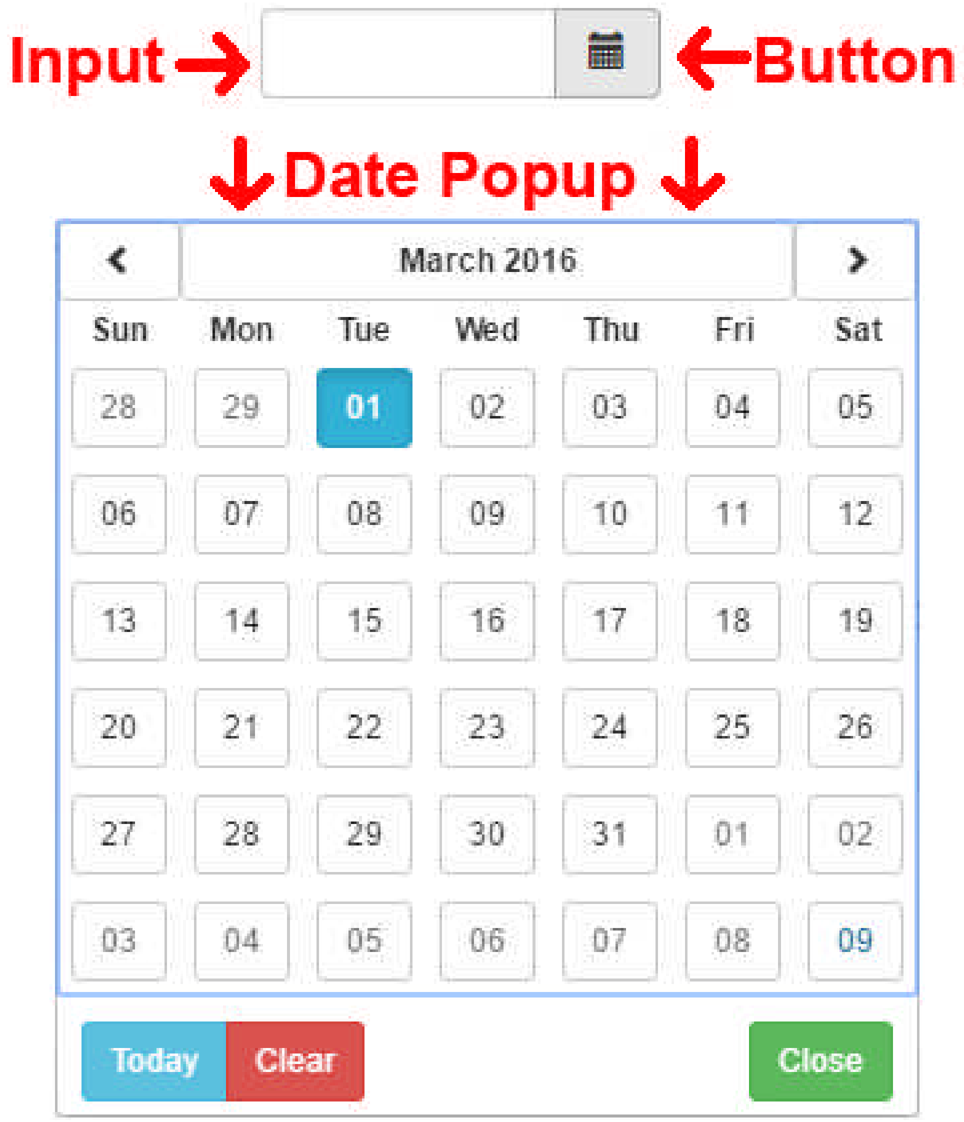
The second argument is an array of other modules that the **learnWith** app is dependant upon. We had already listed the **ngRoute**, and two **uiGrid** directives Now we can add **ui.bootstrap** This makes the **DateChooser** available to the **TaskFilter** template.

You'll remember that the **TaskFilter.html** template is using an HTML table to lay out items. The first step is to add additional headers:

```
<td>Created After</td>
<td>Created Before</td>
<td></td>
<td>Scheduled After</td>
<td>Scheduled Before</td>
```

These table data will go in the first row of the table tag. The empty table data between "Created After" and "Created Before" is just for some space between the borders, which will be added in the next section.

I want to review the aspects of the **DateChooser** component before we look at the code:



There are three different aspects to the **DatePicker** component. The first is the text Input. This will display the currently selected item. The second is a button. The button is used to open and close the date popup. The final aspect is the actual date popup which is displayed or removed either by clicking the button or by giving focus to the input. All three aspects could be used separately, but this project will combine them to a single user experience.

Now, we need to add the **DateChooser** components. I'll start with one, discuss it in details and then give you the code for the other three:

```
<td>
  <p class="input-group">
    <input type="text" class="form-control"
           uib-datepicker-popup="MM/dd/yyyy"
           ng-model="taskModelWrapper.taskModel.taskFilter.startDate"
           is-open="createdAfterDateProperties.opened"
           datepicker-options="dateOptions"
           close-text="Close"
           />
    <span class="input-group-btn">
      <button type="button" class="btn btn-default"
              ng-click="openDatePicker(createdAfterDateProperties)">
                <i class="glyphicon glyphicon-calendar"></i>
      </button>
    </span>
  </p>
</td>
```

The whole thing is wrapped in a **td** tag, which is in the second table row of the **TaskFilter.html**. The **td** tag comes after the two tags for the select inputs. Inside the **td** is a **div** with the class input-group, a CSS style from Bootstrap CSS that allows us to easily extend inputs by putting buttons next to it. We're putting a calendar button up against the text input. Next is the input element. The input will display the selected date. The input has multiple properties on it:

- **type**: This attribute specifies the type of input this element represents. In this case it is a text input. This is a standard HTML property to the tag and is independent of Angular.
- **class**: This attribute specifies the CSS class to be applied to this element. This uses a bootstrap style, form-control.
- **uib-datepicker-popup**: This is used to tell the UI Bootstrap library that this input is part of a date picker. The value here is a date mask which will be used to format the input date. I set it to format in normal US style dates.
- **ng-model**: This is a standard Angular directive we've used in past chapters. It binds the selected date to the specified item. In this case it goes to the **startDate** property in the **TaskModel**'s **taskFilter** object.
- **is-open:** This property is used to determine whether the date popup is displayed or hidden. It is bound to a new object in the **MainScreenCtrl** named **createdAfterDateProperties**. This is the object:

```
$scope.createdAfterDateProperties = {
    opened : false
};
```

This is a simple object, which only contains the opened property. If the opened property is set to "true", then the date popup will be opened. If it is set to "false", then the date popup will be closed. This property will be toggled by clicking the button.

- **datepicker-options**: This is a property that references an object in the controller that specifies properties of the **DateChooser** component:

```
$scope.datePickerOptions = {
   startingDay: 0,
   showWeeks: false
};
```

There are three properties set here. The first is the **startingDay** property. It is set to "0" and refers to the order that days are displayed in the popup. With a value of "0", the first day of the week is Sunday. If you set the start-day value to "1", then the first day displayed will be Monday, and so on. The last value is **showWeeks** which tells the **datePicker** component whether or not to display week numbers inside the date picker popup. I have it set to "false".

After the **DatePicker**, you'll see a button in the HTML. Inside the button is an **i** tag with the class **icon-calendar**. This is part of Bootstrap and provides you with the graphic that displays on the button. The button has an **ngClick** directive. This means that when the button is clicked, it will call the function **openDatePicker** inside the **MainScreenCtrl**. The value passed is the model's property object for this particular **DatePicker** instance. Here is the function:

```
$scope.openDatePicker = function(data) {
    data.opened = !data.opened;
};
```

The function toggles the opened property on the **DatePicker**'s custom data object, which will in turn open the date popup. The reason for adding the **DatePicker**'s custom data argument as an object was so that the same open function could be reused with multiple **DatePickers**.

The remaining **DatePickers** are implemented similarly. For completeness this is the code:

```
<td>
  <p class="input-group">
    <input type="text" class="form-control"
           uib-datepicker-popup="MM/dd/yyyy"
           ng-model="taskModelWrapper.taskModel.taskFilter.endDate"
           is-open="createdBeforeDateProperties.opened"
           datepicker-options="dateOptions"
           close-text="Close"
    />
    <span class="input-group-btn">
       <button type="button" class="btn btn-default"
               ng-click="openDatePicker(createdBeforeDateProperties)">
           <i class="glyphicon glyphicon-calendar"></i>
       </button>
    </span>
  </p>
</td>
<td></td>
<td>
  <p class="input-group">
    <input type="text" class="form-control"
        uib-datepicker-popup="MM/dd/yyyy"
        ng-model="taskModelWrapper.taskModel.taskFilter.scheduledStartDate"
        is-open="scheduledAfterDateProperties.opened"
        datepicker-options="dateOptions"
        close-text="Close"
    />
    <span class="input-group-btn">
```

```
       <button type="button" class="btn btn-default"
                ng-click="openDatePicker(scheduledAfterDateProperties)">
          <i class="glyphicon glyphicon-calendar"></i>
       </button>
    </span>
  </p>
</td>
<td>
  <p class="input-group">
    <input type="text" class="form-control"
         uib-datepicker-popup="MM/dd/yyyy"
         ng-model="taskModelWrapper.taskModel.taskFilter.scheduledEndDate"
         is-open="scheduledBeforeDateProperties.opened"
         datepicker-options="dateOptions"
         close-text="Close"
    />
    <span class="input-group-btn">
      <button type="button" class="btn btn-default"
              ng-click="openDatePicker(scheduledBeforeDateProperties)">
         <i class="glyphicon glyphicon-calendar"></i>
      </button>
    </span>
  </p>
</td>
```

Each **DatePicker** instance will also need a related property for custom objects in the **MainScreenCtrl.js** file:

```
$scope.createdBeforeDateProperties = {
    opened : false
};
$scope.scheduledAfterDateProperties = {
    opened : false
};
$scope.scheduledBeforeDateProperties = {
    opened : false
};
```

These three objects mirror the **createdAfterDateProperties** object shown earlier in this chapter.

The Filter Button

There is only one more aspect to add to our **TaskFilter.html**; the filter button. It will cause the UI to load new tasks with the modified criteria In the second row of the filter form's table, place this at the end:

```
<td>
    <input type="button" value="Filter" />
</td>
```

Right now the button doesn't do anything. We'll implement the functionality in a bit.

Adding Styles

The last aspect of creating the user interface for this chapter is to add some styles. I know this book is not intended to be a design book, but I did want to add some basic layout and sizing. All styles will be added to the **styles.css** file from the **com/dotComIt/learnWith/styles** directory.

First, I felt the default lengths of the **datePicker** input and the select drop downs were too large. To address that, I created some styles which would shorten their widths:

```
.datePicker {
    width:150px;
}
.completedDropDown {
    width:150px;
}
.taskCategoryDropDown {
    width:150px;
}
```

These styles can be added to the inputs in the **TaskFilter.html** file. First, here is the completed drop down:

```
<select
        // other properties
        class="completedDropDown">
```

I replaced many of the properties you have already seen with a standard JavaScript comment. Be sure not to use this approach in your real code. I only did it here so I would not bombard you with duplicate information. It uses the HTML class attribute to refer to the CSS style. The same happens for the task category drop down:

```
<select
        // other properties
        class="taskCategoryDropDown">
```

The **DatePicker** style is put on the paragraph which encloses the **DatePicker**:

```
<p class="input-group datePicker">
```

With the date picker code, the internal elements are set to expand to 100% of their width from using the Bootstrap styles. We control the width of the date picker by controlling the width of the date picker's parent container. Since the paragraph already had a Bootstrap CSS style on it, I added our custom style to the class after the Bootstrap style, separating it with a space. The same **datePicker** class is used on all of the **DatePicker** inputs, but since the code is identical, you don't need to see the other three additions.

Next, I added some basic styles to the table and table data:

```
table{
    border-collapse: collapse;
}
td {
    vertical-align: top;
    padding:5px;
}
```

These are tag-level styles and will be immediately picked up by all table and table data tags within the application. The border collapse on the table means that visible borders in the table will ignore any spacing or padding when creating the visual border. The default **td** aligns items to the top of the cell and adds some padding around each cell. This made the **TaskFilter** component feel less crowded from a visual perspective.

There is one table cell where I wanted to align the elements to the bottom instead of the top; the button's table cell:

```
.alignBottom {
    vertical-align: bottom;
}
```

It is applied like this:

```
<td class="alignBottom">
    <input type="button" value="Filter" />
</td>
```

The final styling step to do is to add the borders. To do this, we'll create custom styles for the table cells that add borders on some combination of the top, bottom, left, and/or right. First, let's look at what is needed:

- To the left of, and above the completed header cell.
- Above, and to the right of the category header cell.
- To the left of, and above the created after-header cell.
- Above, and to the right of the created before-header cell
- To the left of, and above the scheduled-after cell.
- Above, and to the right of the scheduled-before cell
- To the left, and below the completed drop down.
- To the right, and below the category drop down.
- To the left, and below the created-after **DatePicker**.
- Below, and to the right of the created-before **DatePicker**.
- To the left, and below the scheduled-after **DatePicker**.
- Below, and to the right of the scheduled-before **DatePicker**.

This is the combination of styles that are needed to create the square borders that span multiple table cells. These are the styles:

```
.border-top-left {
    border-left: solid 2px grey;
    border-top: solid 2px grey;
}
.border-top-right {
    border-right : solid 2px grey;
    border-top: solid 2px grey;
}
.border-bottom-right {
    border-bottom: solid 1px grey;
    border-right : solid 2px grey;
}
.border-bottom-left {
    border-bottom: solid 1px grey;
    border-left : solid 2px grey;
}
```

This creates six different styles that will add a border to a table cell, or **div**, or other HTML elements. We'll use them on **td** tags in order to create borders around the **date-created** properties and **date-scheduled** properties.

First, in the header row:

```
<td class="border-top-left">Completed</td>
<td class="border-top-right">Category</td>
<td></td>
<td class="border-top-left">Created After</td>
<td class="border-top-right">Created Before</td>
<td></td>
<td class="border-top-left">Scheduled After</td>
<td class="border-top-right">Scheduled Before</td>
```

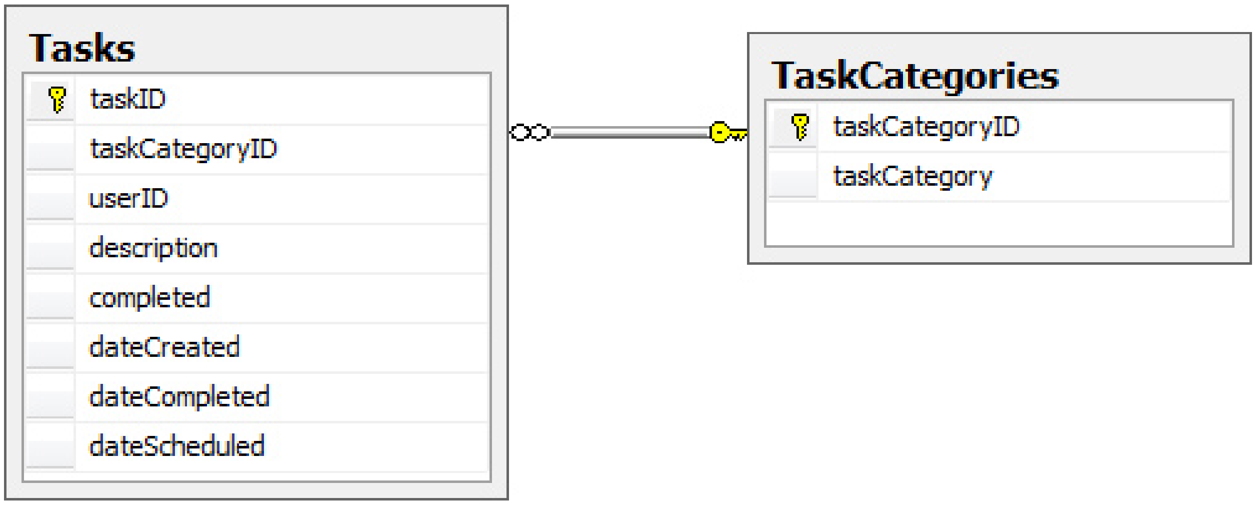
Then, this is the bottom row, which contains the full **DatePicker** components:

```
<td class="border-bottom-left">
  // completed drop down
</td>
<td class="border-bottom-right">
    // category drop down
</td>
<td></td>
<td class="border-bottom-left">
  // created after DatePicker
</td>
<td class="border-bottom-right">
    // created before DatePicker
</td>
<td></td>
<td class="border-bottom-left">
    // scheduled after DatePicker
</td>
<td class="border-bottom-right">
   // scheduled Before DatePicker
</td>
```

This completes the section on the styling and CSS that we applied to this app.

Examine the Database

There is no new database structure to examine in this chapter; as the categories were already shown in the previous chapter. Here is a quick review:



The focus here is on querying the **TaskCategories** table. There is no need for a "where" clause in this query:

```
select * from taskCategories
order by taskCategory
```

The SQL will return all items in the **TaskCategories** table.

Write the Services

This section focuses on the service needed to load the task categories. It will cover the ColdFusion services required, and the AngularJS code for integrating with the service. The task category list is used in the UI for filtering objects. It is also used in the screen for creating or editing a task. This chapter introduces some new fields for filtering the tasks as well, so we'll modify the **getFilteredTasks()** service method we started in the previous chapter.

Revisit the getFilteredTasks( ) Method

Chapter 4 of the main book is focused on building out the code to control the task display in the main grid. It will introduce many new fields that can be used to filter the main task grid:

- **taskCategoryID**: This property is used to filter the task list based on a category.
- **endDate**: This property is used to filter the task list based on the **dateCreated** database property. The query should make sure that the **dateCreated** comes before the **endDate** for the task to show up in the result.
- **scheduledEndDate**: This is used to filter the task list based on the **dateScheduled** column. If this is specified, then the query should make sure that the **dateScheduled** property is earlier than the **scheduledEndDate** value.
- **scheduledStartDate**: This compares against the **dateScheduled** column. If this is specified, then the query should make sure that the **dateScheduled** property is later than the **scheduledStartDate** value.

I'm not going to review the full **getFilteredTasks()** method in the **TaskService.cfc**. I'm just going to add these new elements to the query. The **TaskService** class is in the **com/dotComIt/learnWith/services** directory.

This is the code to check for the **taskCategoryID**:

```
<cfif StructKeyExists(local.json,'taskCategoryID') and
      local.json['taskCategoryID'] NEQ 0>
  <cfif setWhere is true >where <cfset setWhere = 'false'></cfif>
  <cfif firstOne is true><cfset firstOne = false><cfelse>and</cfif>
  taskCategories.taskCategoryID = <cfqueryparam cfsqltype="cf_sql_integer"
                                     value="#local.json['taskCategoryID']#">
</cfif>
```

The code checks the **local.json** value for the **taskCategoryID**. If it exists, then a filter should be added to the "where" clause of the query. If it doesn't exist, then nothing should be added to the "where" clause of the query. The **taskCategoryID** code also needs to check that the **taskCategoryID** is not a "0" value. A "0" value in the UI will be used to communicate that no category was selected and all categories should be shown. When filtering tasks the user will be able to select an all categories option.

The code checks the **setWhere** value. If it is "true", then it adds the "where" statement and sets the **setWhere** to "false". If it is "false", then nothing is added. The next line checks for **firstOne**. If it is "true", then it sets it to "false". Otherwise it adds the "and" statement to the clause. The **firstOne** and **setWhere** values are used to properly set the "where" and "and" statement when building the query.

When running this code, we'll never know what value will be first. The final line of this segment adds the condition to the query checking for equality of the **taskCategoryID**. A **cfqueryparam** is used, as that is one ColdFusion method of scrubbing user input of bad stuff.

The previous chapter examined the **startDate** value and compared it against the **dateCreated** database column. This also includes the **endDate** comparison. The **dateCreated** should be less than or equal to the **endDate**:

```
<cfif StructKeyExists(local.json,'endDate')>
      <cfif setWhere is true >where <cfset setWhere = 'false'></cfif>
      <cfif firstOne is true><cfset firstOne = false><cfelse>and</cfif>
      dateCreated <= <cfqueryparam cfsqlType="cf_sql_date"
                                   value="#local.json['endDate']#">
</cfif>
```

The **scheduledStartDate** and **scheduledEndDate** are processed the same way that the **startDate** and **endDate** were processed. They are used to compare against a different column in the database; **dateScheduled**:

```
<cfif StructKeyExists(local.json,'scheduledStartDate')>
  <cfif setWhere is true >where <cfset setWhere = 'false'></cfif>
  <cfif firstOne is true><cfset firstOne = false><cfelse>and</cfif>
  dateScheduled >= <cfqueryparam cfsqltype="cf_sql_date"
                               value="#local.json['scheduledStartDate']#">
</cfif>
<cfif StructKeyExists(local.json,'scheduledEndDate')>
  <cfif setWhere is true >where <cfset setWhere = 'false'></cfif>
  <cfif firstOne is true><cfset firstOne = false><cfelse>and</cfif>
  dateScheduled <= <cfqueryparam cfsqlType="cf_sql_date"
                               value="#local.json['scheduledEndDate']#">
</cfif>
```

The TaskCategory Value Object

A CFC Object will be needed to store the category data. To do so, you can create a **TaskCategory** value object:

```
<cfcomponent displayname="TaskCategoryVO" hint="I represent the order count."
             alias="com.dotComIt.learnWith.vos.TaskCategoryVO">
      <cfproperty name="taskCategoryID" type="numeric" />
      <cfproperty name="taskCategory" type="String" />
</cfcomponent>
```

The CFC goes in the **com/dotComIt/learnWith/vos** directory and was named **TaskCategoryVO**. It includes two properties, the **taskCategoryID** and the **taskCategory**. Each property reflects a column in the database table. In terms of value object CFCs reviewed in this book, this is as simple as they come.

In applications where there are a lot of queries which populate drop down lists, radio buttons, or similar UI elements, I will sometimes create a generic object strictly for containing controlled vocabulary data. The generic object includes an ID and a label, and I name it **ControlledVocabularyVO**. In this case, however, an explicit object for the task category data seemed better.

Loading the Task Categories

The method for loading task categories is placed in the same **TaskService** class used by the **getFilteredTasks()** method. This method is named **getTaskCategories()**:

```
<cffunction name="getTaskCategories" returnformat="plain" access="remote"
            output="true" required="true" hint="I retrieve task Categories" >
</cffunction>
```

No arguments need to be passed into this method, as this method will always return all category data. The first thing the method must do is query the database:

```
<cfquery datasource="#application.dsn#" name="local.dataQuery" >
      select * from taskCategories
      order by taskCategory
</cfquery>
```

The query uses the same SQL Query text shown earlier in this chapter. It is a simple query to retrieve all tasks in the **TaskCategory** database table.

Next, create the result object and a results array:

```
<cfset local[‘resultObject’] = createObject('component',
                                'com.dotComIt.learnWith.vos.ResultObjectVO') />
<cfset local.results = arrayNew(1) />
```

This code creates two variables local to the function. One is a **resultObject**, which contains an instance of the **ResultObjectVO** which will be used to wrap all the results from this application’s service calls. There is also a results array, which will include all the **TaskCategoryVO** objects.

With the tasks categories, you can add hard-coded value which will be used in the UI for selecting all categories:

```
<cfset local.tempObject = createObject('component',
                                 'com.dotComIt.learnWith.vos.TaskCategoryVO')/>
<cfset local.tempObject['taskCategoryID'] = 0 />
<cfset local.tempObject['taskCategory'] = "All Categories" />
<cfset arrayAppend(local.results,local.tempObject)/>
```

The hard-coded category includes a **taskCategoryID** of “0”, and a **taskCategory** of “All Categories”. The **taskCategoryID** of “0” was the property used in the **getFilteredTasks()** service method to determine that no “where” clause related to categories should be displayed.

Next, loop over the results of the database query, and create a **TaskCategoryVO** for each item:

```
<Cfoutput query="local.dataQuery">
  <cfset local.tempObject = createObject('component',
                                 'com.dotComIt.learnWith.vos.TaskCategoryVO')/>
  <cfset local.tempObject['taskCategoryID'] = dataQuery.taskCategoryID />
  <cfset local.tempObject['taskCategory'] = dataQuery.taskCategory />
  <cfset arrayAppend(local.results,local.tempObject)/>
</Cfoutput>
```

The new **TaskCategoryVO** is stored in the results array. The results array needs to be added to the **resultObject** instance:

```
<cfset local.resultObject['resultObject'] = local.results />
```

Set the **resultObject** error value to “false”, as a precautionary measure:

```
<cfset local.resultObject['error'] = false/>
```

In this method, the code doesn’t perform an error checking. It, for the most part, assumes that the method will run without errors.

Finally, convert the **resultObject** to JSON and return it:

```
<cfset local.resultString = convertToJSON(local.resultObject) />
<cfreturn local.resultString>
```

The method we created in the previous chapter, **convertToJSON()**, is called to perform the final JSON conversion. This is a simple service method, especially compared to what we have previously created.

Testing getTaskCategories()

It is time to test the method to **getTaskCategories()**. You can find a test file, **getTaskCategories.cfm**, in the **com/dotComIt/learnWith/tests** directory. First, create an instance of the service:

```
<cfset service =
 createObject('component','com.dotComIt.learnWith.services.TaskService')>
```

Since this new service method has no arguments, it is easy to just call the method on the service instance:

```
<cfset results = service.getTaskCategories()>
```

Finally, output the results:

```
Results: <Br/>
<cfdump var="#results#" expand="false" />
```

You should see results similar to this on the screen:

```
{
 "resultObject":
  [
      {"taskCategoryID":0.0, "taskCategory":"All Categories"},
      {"taskCategoryID":1,   "taskCategory":"Business"},
      {"taskCategoryID":2,   "taskCategory":"Personal"}
  ],
  "error":false
}
```

This is a JSON packet. It contains two properties; the **resultObject** property, and the **error** property. The **resultObject** is an array. Arrays are delineated with square brackets. Each element of the array is like a mini JSON packet.

Review the API

To integrate this service into the JavaScript applications, you need to know the end point and the arguments. If the app and service are served off the same domain, you can use a relative URL for the endpoint. In this case, I used:

```
/com/dotComIt/learnWith/services/TaskService.cfc
```

The **getTaskCategories()** method has no input arguments, but when making the service call you still need to tell the server which method to execute at the end point. You do that by specifying the method parameter and naming it **getTaskCategories**.

Access the Service

This section will cover the new AngularJS code needed to retrieve the task categories from the server. Start by opening up the **TaskService.js** file from the **com/dotComIt/learnWith/services/coldFusion** directory. Add a new method inside the **taskService** object:

```
function loadTaskCategories(){
    var parameters = {
        method : "getTaskCategories"
    };
    $http.post('/com/dotComIt/learnWith/services/TaskService.cfc',
               parameters )
};
```

This method accepts no arguments, as the service will return all task categories, and needs no other special criteria. It creates a parameter object with a single parameter containing the name of the method; **getTaskCategories**. Then it makes a **post()** call on the **$http** object. It is really that simple.

Be sure to add the **loadTaskCategories()** method to the **TaskService** service object:

```
var services = {
    loadTasks : loadTasks,
    loadTaskCategories : loadTaskCategories
}
```

This will make the **loadTaskCategories()** method available to the controllers that need it.

Since the **TaskService.js** is already included into the main index file, you shouldn't need to make any changes to your index file to make use of this code. For the appropriate imports, you can look at **index_ColdFusion.html** in our code archive.

Wire Up the UI

The final step of this chapter is to wire up the services to the user interface and make everything work. This section will examine the code behind loading the task categories and will show you how to hook up the “Filter Tasks” button.

Loading Task Categories

You can create a method in the **MainScreenCtrl.js** file to load the task categories:

```
function loadTaskCategories(){
   TaskService.loadTaskCategories()
              .then(onTaskCategoriesLoadSuccess,onTaskLoadError)
}
```

This will execute the **loadTaskCategories()** method inside the **TaskService** service. A promise object is returned from the service. Based on the promise object, if the result is a success, then the **onTaskCategoriesLoadSuccess()** method will be executed. If there is an error, then the **onTaskLoadErrors()** method will be executed. The **onTaskLoadErrors()** method is the same method used in previous chapters, so I am not showing it again.

The purpose of the **onTaskCategoriesLoadSuccess()** method is to check to see if a successful result was received from the service. If the call was successful, then you should save the task category array in the **taskModel**. If the call was a failure, then display a message to the user:

```
function onTaskCategoriesLoadSuccess(response){
  if(response.data.error == 1){
      console.log('we could load the task categories');
      alert("We could not load the task categories");
      return;
  }
  $scope.taskModelWrapper.taskModel.taskCategories =
                                                  response.data.resultObject;
}
```

The method checks the error property of the returned value. If it is “1”, then there was an error. An alert is displayed to the user. If it is “0”, then the **resultObject** is saved in the **taskModelWrapper.taskModel.taskCategories** variable. This will cause Angular to automatically update the select drop down that contains the task categories.

The task categories need to be loaded after the user logs in. To do this, you can revisit the **init()** method we created in the **MainScreenCtrl.js**. Just add this line after the **loadTasks()** method is called:

```
loadTaskCategories();
```

This will call the method which loads the task categories, which are then populated in model’s **taskCategories** array, which in turn will update the **TaskFilter.html**.

Since the categories drop down now has values in it, we should also set a default value. Open up the **TaskModel.js** file from **com/dotComIt/learnWith/model** directory. Add a default **taskCategoryID** to the **taskFilter** object:

```
taskFilter : {
    completed : 0,
    startDate : "3/1/2013",
    taskCategoryID : 0
}
```

The **taskCategoryID** of “0” represents the “all categories” option

Triggering the Filter

The final step for this chapter is to implement the code behind the filter button. First, the button inside the **TaskFilter.html** template will have to respond to the click event:

```
<input type="button" value="Filter" ng-click="onFilterRequest()" />
```

An **ngClick** directive was added. When clicked, the method **onFilterRequest()** will be executed inside the **MainScreenCtrl**. The purpose of the **onFilterRequest()** method is to put together a filter object and call the **loadTasks()** method.

In the original implementation of **loadTasks()** in Chapter 3, it accessed the **TaskModel**’s **taskFilter** object directly. This chapter changes that to accept a **taskFilter** object as an argument The reason for this is because we want to conditionally include properties in the filter object sent to the service. If the object has no property given, we do not want to include it in the service call. The first step is to modify the **loadTask()** method. This is the existing method:

```
function loadTasks(){
   TaskService.loadTasks($scope.taskModelWrapper.taskModel.taskFilter)
               .then(onTaskLoadSuccess, onTaskLoadError);
}
```

We will add a **taskFilter** argument now. This will, in turn, be passed onto the **loadTasks()** method in the **TaskService**:

```
function loadTasks(taskFilter){
    TaskService.loadTasks(taskFilter)
                .then(onTaskLoadSuccess, onTaskLoadError);
}
```

Make sure to go to the **onInit()** method and modify the initial **loadTasks()** trigger:

```
loadTasks($scope.taskModelWrapper.taskModel.taskFilter);
```

Now, we can continue to implement the **onFilterRequest()** method. First step here is to create a customized filter object. Here is the method signature:

```
$scope.onFilterRequest = function onFilterRequest(){
}
```

The method needs no arguments. The first step is to create an empty **taskFilter** object:

```
var localTaskFilter = {};
```

Then we can start populating the **taskFilter** object with properties based on values in the **taskModel**'s **taskFilter** object. This can be done in the same order that the elements appear in the UI. First the two drop downs:

```
if($scope.taskModelWrapper.taskModel.taskFilter.completed != -1){
 localTaskFilter.completed = $scope.taskModelWrapper.taskModel.taskFilter.completed;
}
localTaskFilter.taskCategoryID=
                        $scope.taskModelWrapper.taskModel.taskFilter.taskCategoryID;
```

The completed value is set to a default of "0", which represents all open tasks. The completed property is like a Boolean value on the service layer representing open and completed tasks. However, in the UI it must account for a third value, which is "all tasks". If the completed value is not "-1", then add it to the local **taskFilter** object. If the value is set to "-1", then the service should return all tasks, and the completed property should be left out of the current object.

The next line sets the **taskCategoryID** property on the **localTaskFilter**. The **taskCategoryID** will be set in the results regardless of its value.

Next, we need to examine the date fields:

```
if($scope.taskModelWrapper.taskModel.taskFilter.startDate){
    localTaskFilter.startDate = $filter('date')
             ($scope.taskModelWrapper.taskModel.taskFilter.startDate,'shortDate');
}
if($scope.taskModelWrapper.taskModel.taskFilter.endDate){
    localTaskFilter.endDate = $filter('date')
              ($scope.taskModelWrapper.taskModel.taskFilter.endDate, 'shortDate');
}
if($scope.taskModelWrapper.taskModel.taskFilter.scheduledStartDate){
    localTaskFilter.scheduledStartDate = $filter('date')
   ($scope.taskModelWrapper.taskModel.taskFilter.scheduledStartDate, 'shortDate');
}
if($scope.taskModelWrapper.taskModel.taskFilter.scheduledEndDate){
    localTaskFilter.scheduledEndDate = $filter('date')
     ($scope.taskModelWrapper.taskModel.taskFilter.scheduledEndDate, 'shortDate');
}
```

You'll notice that this code references a **$filter** object. Before explaining the code, you'll want to make sure that the **$filter** object is passed into the Controller in the **controller** definition:

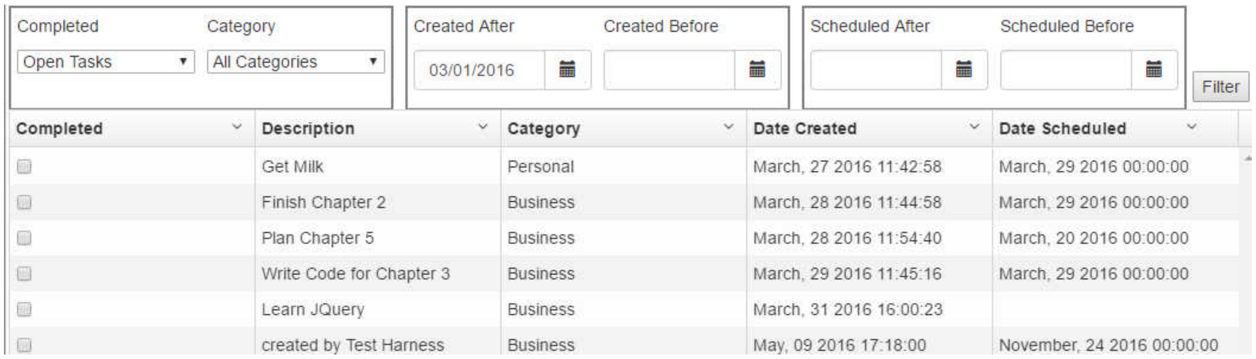
```
angular.module('learnWith').controller('MainScreenCtrl',
 ['$scope','$location','$filter','UserModel','TaskModel','TaskService',
   function($scope,$location,$filter,UserModel,TaskModel,TaskService){
```

The **$filter('date')** command is used to format the date and by specifying the **shortDate** value as the argument. The date will be formatted in the "month/day/year" format, which is what the service needs.

Finally, the **loadTasks()** method must be called:

```
loadTasks(localTaskFilter);
```

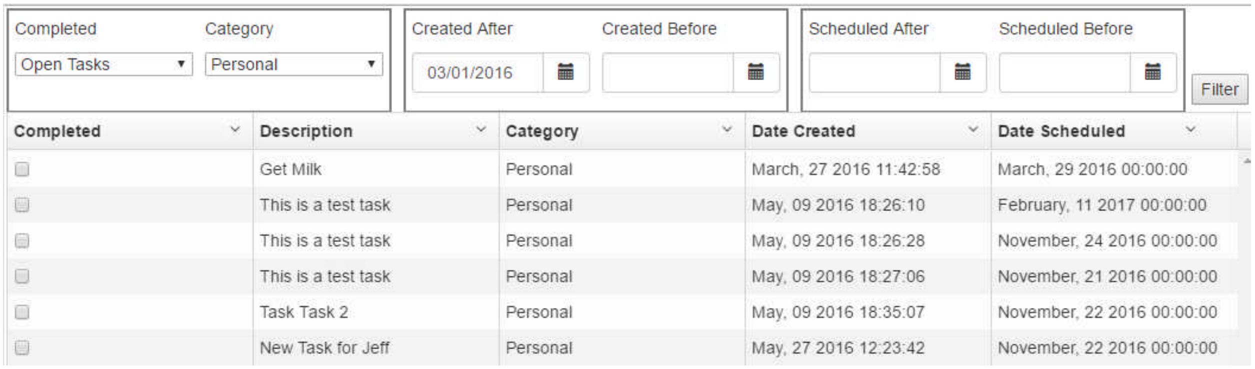
This completes the method and the code I wanted to cover in this chapter. I want to show you some different screenshots of the grid with different properties. First, this is a default load of the grid:



Completed	Category	Created After	Created Before	Scheduled After	Scheduled Before
Open Tasks	All Categories	03/01/2016			

Completed	Description	Category	Date Created	Date Scheduled
☐	Get Milk	Personal	March, 27 2016 11:42:58	March, 29 2016 00:00:00
☐	Finish Chapter 2	Business	March, 28 2016 11:44:58	March, 29 2016 00:00:00
☐	Plan Chapter 5	Business	March, 28 2016 11:54:40	March, 20 2016 00:00:00
☐	Write Code for Chapter 3	Business	March, 29 2016 11:45:16	March, 29 2016 00:00:00
☐	Learn JQuery	Business	March, 31 2016 16:00:23	
☐	created by Test Harness	Business	May, 09 2016 17:18:00	November, 24 2016 00:00:00

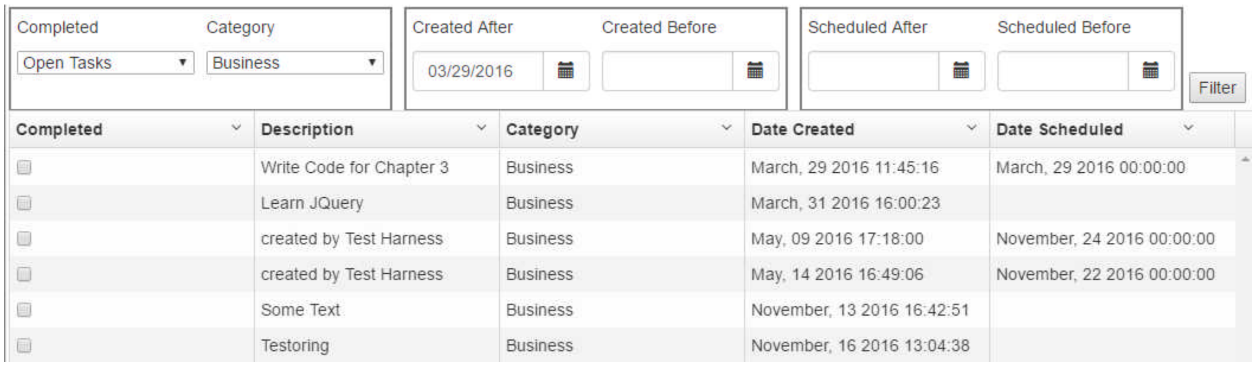
Now, change the category to "Personal" and press the Filter button:



Completed	Category	Created After	Created Before	Scheduled After	Scheduled Before
Open Tasks	Personal	03/01/2016			

Completed	Description	Category	Date Created	Date Scheduled
☐	Get Milk	Personal	March, 27 2016 11:42:58	March, 29 2016 00:00:00
☐	This is a test task	Personal	May, 09 2016 18:26:10	February, 11 2017 00:00:00
☐	This is a test task	Personal	May, 09 2016 18:26:28	November, 24 2016 00:00:00
☐	This is a test task	Personal	May, 09 2016 18:27:06	November, 21 2016 00:00:00
☐	Task Task 2	Personal	May, 09 2016 18:35:07	November, 22 2016 00:00:00
☐	New Task for Jeff	Personal	May, 27 2016 12:23:42	November, 22 2016 00:00:00

Finally, try changing the category to "Business" and setting the "Created After" date to 3/29:



Completed	Category	Created After	Created Before	Scheduled After	Scheduled Before
Open Tasks	Business	03/29/2016			

Completed	Description	Category	Date Created	Date Scheduled
☐	Write Code for Chapter 3	Business	March, 29 2016 11:45:16	March, 29 2016 00:00:00
☐	Learn JQuery	Business	March, 31 2016 16:00:23	
☐	created by Test Harness	Business	May, 09 2016 17:18:00	November, 24 2016 00:00:00
☐	created by Test Harness	Business	May, 14 2016 16:49:06	November, 22 2016 00:00:00
☐	Some Text	Business	November, 13 2016 16:42:51	
☐	Testoring	Business	November, 16 2016 13:04:38	

These are just a few different options on how you can filter the tasks that are displayed to the user.

Final Thoughts

After this chapter, you should have an understanding of the UI Bootstrap project and how to use it within AngularJS to create a **DatePicker** component. We also covered how to use Angular to populate drop down boxes for the first time. Some services were reviewed, and everything was wired up to create a working and functional UI. The next chapter will focus on the system for creating and editing tasks.

Chapter 5: Creating and Editing Tasks

This chapter will show you how to create and edit tasks. It will present the user interface that we will create using AngularJS, and how to create a modal popup using Bootstrap and AngularJS. It will show you how to create service methods for saving and updating a task. Finally, it will tie everything together by wiring up the UI to the services.

Create the User Interface

This section creates a popup window that can be used for both creating new tasks, and editing existing ones. It will start by showing what the UI popup should look like. Next, it will expand on the implementation details behind that UI and how to create the popup within the Angular application.

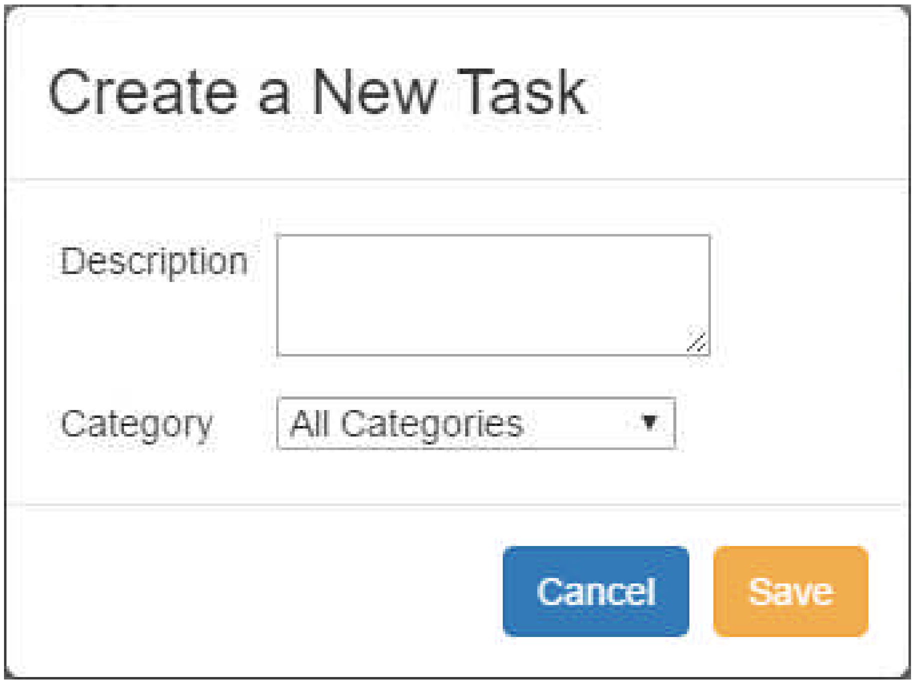
The Task Window

There are two elements that are important to create a new task:

- **Task Description**: This element is the main text which makes up the task.
- **Task Category**: This element contains the categorization that the task will be put in.

Other task-related data—such as the completed status, the task creation date, and the task scheduled date—will not be edited manually when creating the task. These extra fields are kept out of this UI in order to keep things simple. Marking a task completed and scheduling a task are both important but will be addressed in future chapters.

This is the popup screen for creating a new task:



The window is implemented as a popup. The edit task window is almost identical, except the input fields will be populated with data based on the selected task; instead of reading, "Create a New Task," the window title will now say, "Edit Task."

Implement the Popup

The first thing that you need to do is to create the popup file. I named the file **TaskCU.html** and put it in the **com/dotComIt/learnWith/views/tasks** directory. The file has three parts; a header, a content area, and a footer area.

This is the header:

```
<div class="modal-header">
    <h3 class="modal-title">{{title}}</h3>
</div>
```

The header is represented by a **div**, and it is given a Bootstrap CSS class; **modal-header**. The content of the header is a variable title which will reference a variable inside the popup's controller. The popup will have a custom controller which will contain the code need to load, save, and edit task data

The next element of the popup is the content area of the form. It starts out with a **div**:

```
<div class="modal-body">
</div>
```

The class name for the content area is called **modal-body**; a special Bootstrap style. The content area of the form contains the description text area and the category drop down. I put them in a table for easy layout. First, the description:

```
<table>
    <tr>
        <td>Description</td>
        <td>
            <textarea ng-model="taskVO.description"></textarea>
        </td>
    </tr>
```

The text area is a simple HTML tag. It uses the **ngModel** tag to bind the value of the **TextArea** to a value inside the popup's controller; **taskVO.description**.

Next is the category drop down. This implementation is similar to the category drop down used in the task filter component:

```
<tr>
  <td>Category</td>
  <td>
    <select ng-model="taskVO.taskCategoryID"
            ng-options="item.taskCategoryID as item.taskCategory for item in
                        taskModel.taskCategories"
            class="taskCategoryDropDown">
    </select>
  </td>
</tr>
</table>
```

The table row contains a label, “Category”, and an HTML select box. The HTML select box is populated with Angular, using the **taskCategories** array that was loaded from the service layer in the previous chapter. The data was cached in the **taskModel** class. The selected value of the category select box is bound to the **taskVO.taskCategoryID** value in the controller using the **ngModel** tag. You’ve seen all of this before.

The final section of the **TaskCU.html** file is the footer. It contains the cancel and save buttons:

```
<div class="modal-footer">
  <button class="btn btn-warning" type="button" ng-click="onClose()">
    Cancel
  </button>
  <button class="btn btn-primary" type="button" ng-click="onSave()">
    Save
  </button>
</div>
```

The class for the footer **div** is **modal-footer**. Once again, this refers to a Bootstrap CSS class. Two buttons are defined here; one for the cancel button, and one for the save button. Both buttons are styled using Bootstrap styles. Each button has a generic button style named **btn**. There are two secondary styles—one on each button. The cancel button uses **btn-warning** as the style, while the save button uses **btn-primary**. The HTML blocks combine to create the popup.

The Popup Controller

The **TaskCU.html** file refers to a lot of values in an Angular controller. The next step is to create that controller. I created a file named **TaskCUCtrl.js** and put it in the **com/dotComIt/learnWith/controllers** directory. First, define the controller:

```
angular.module('learnWith').controller('TaskCUCtrl',
  ['$scope','$uibModalInstance','TaskModel','UserModel','TaskService',
   'title','taskVO',
   function($scope,$uibModalInstance,TaskModel,UserModel,TaskService,
            title,taskVO){
   }
  ]
);
```

There are multiple services injected into the function definition:

- **$scope**: This is the Angular directive that allows us to share items with the view template.
- **$uibModalInstance**: This directive is not something we’ve used yet. It is a reference to the popup. We’ll use this to close the popup when the cancel button is pressed, or a new task is updated.
- **TaskModel**: This refers to the factory that was created in previous chapters. It contains task specific information—such as the list of categories.
- **UserModel**: This refers to the factory that was created in previous chapters. It contains the information about the user who is logged into the app.

- **TaskService**: This refers to the service which will be used to update or create the task. For the purposes of this chapter, it will refer to the Mock service.
- **title**: This is a value we pass into the popup when creating it. It will be used to toggle the title between creating a new task, and editing an existing one.
- **taskVO**: This is another value that is passed into the popup when creating it. It represents the task that is being created, or edited.

The next step is to save many of the values locally in the **$scope**:

```
$scope.title = title;
$scope.taskModel = TaskModel;
$scope.taskVO = taskVO;
```

This allows the html template to access these variables. Although we won't implement the **onSave()** method until later in this chapter, you can add the stub in:

```
$scope.onSave = function onSave () {
    console.log('on save');
}
```

The **onSave()** method will have to integrate with the remote service for updating tasks. The **onClose()** method does not need any remote integration, so we can implement it here:

```
$scope.onClose = function onClose() {
    $uibModalInstance.dismiss();
};
```

When the cancel button in the **TaskCU.html** template is clicked, the **onClose()** method will be executed. This method will call the **dismiss()** method on the **$uibModalInstance** property. This will remove the popup from display. It will also call a function in the controller for the Main Screen meaning that the window was closed without completing its task.

The final step is to add the **TaskCUCtrl.js** file into the main **index.html** file:

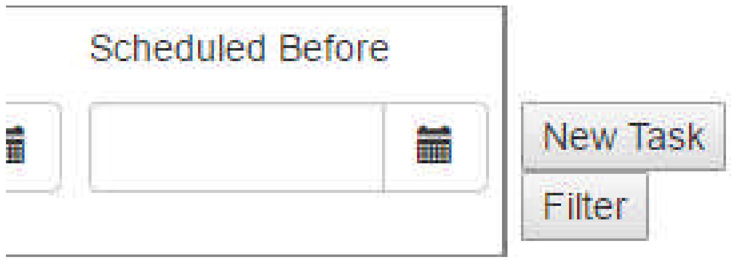
```
<script src="com/dotComIt/learnWith/controllers/TaskCUCtrl.js"></script>
```

This should be placed after the other controllers in the index file.

Opening the New Task Window

There are two different ways to open the create-task popup. One is going to be with an edit button in the **uiGrid**. That will be examined in the next section. The other is going to be with a new task button in the **TaskFilter.html**, which I'll show you now.

This is the button:



The button implementation is fairly simple. Open up the **TaskFilter.html** file from the **com/dotComIt/learnWith/views/tasks** directory. At the bottom of the file, edit the table cell that contains the filter button to add the new task button above it:

```
<td class="alignBottom">
    <input type="button" value="New Task" ng-click="onNewTask()" /><br/>
    <input type="button" value="Filter" ng-click="onFilterRequest()" />
</td>
```

The New Task button uses the **ngClick** directive to call the **onNewTask()** method inside the **MainScreenCtrl.js** file:

```
$scope.onNewTask = function onNewTask(){
    var newTask = {
        taskCategoryID : 0,
        taskID : 0
    };
    openTaskWindow('Create a New Task',newTask);
}
```

This method creates an empty **taskVO** object. The object will be used inside the **modal** to define the default values inside the popup. Then the **openTaskWindow()** method is called. Two parameters are passed into this method, a string representing the title displayed inside the popup and the task object.

Before examining the **openTaskWindow()** method, you'll have to add the **$uibModal** service to the **MainScreenCtrl** definition:

```
angular.module('learnWith').controller('MainScreenCtrl',
  ['$scope','$location', '$filter','$uibModal',
   'UserModel','TaskModel','TaskService',
   function($scope,$location,$filter,$uibModal,
            UserModel,TaskModel,TaskService){
```

The **$uibModal** value is a special value from UI Bootstrap that can be used to create the popup. The **openTaskWindow()** method starts with the function definition:

```
function openTaskWindow(title,task){
}
```

It has two arguments briefly mentioned earlier:

- **title**: This argument will be used in the header of the popup to distinguish between editing a task, or creating a new one.

- **task**: This argument represents the task being modified in the popup.

The next step is to use an **open()** method on the **$modal** value to create the popup. The **open()** method has a single argument which defines the options of the popup. Although not a complete list, these are the options that you'll define:

- **templateURL**: This is the URL of the template that will represent the popup.
- **controller**: This represents the controller which will be created to handle the popup functionality.
- **resolve**: This is an object which will contain a collection of named functions. Each value represents a single value passed into the controller's definition. This is how the title and **taskVO** are passed from the **MainScreenCtrl** to the **TaskCUCtrl**.

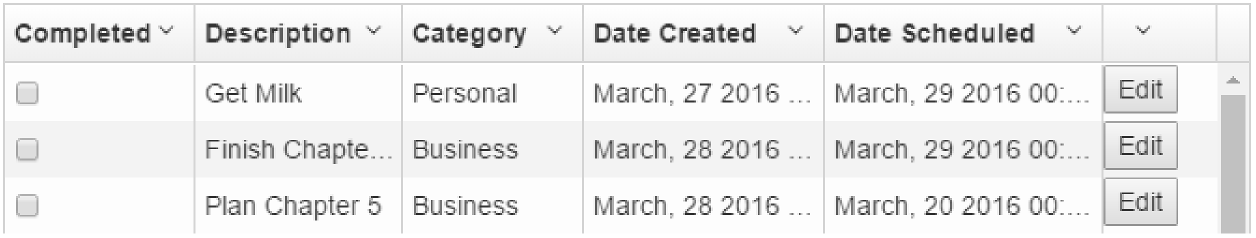
This line opens the popup:

```
var modalInstance = $uibModal.open({
    templateUrl:'com/dotComIt/learnWith/views/tasks/TaskCU.html',
    controller: 'TaskCUCtrl',
    resolve: {
        title : function(){ return title },
        taskVO : function(){ return task }
    }
});
```

At this point, you should be able to run your app, click the new task button, and see the popup show. It should even close if you click the cancel button.

Opening the Edit Task Window

The edit button is going to be placed inside the task grid with a new column:



Completed	Description	Category	Date Created	Date Scheduled	
☐	Get Milk	Personal	March, 27 2016 ...	March, 29 2016 00:...	Edit
☐	Finish Chapte...	Business	March, 28 2016 ...	March, 29 2016 00:...	Edit
☐	Plan Chapter 5	Business	March, 28 2016 ...	March, 20 2016 00:...	Edit

To create this new column, we will create a new column definition template similar to how we created the completed checkbox. First, we'll create the template. Then, we'll modify the column definitions.

I created a file named **EditButtonRenderer.html** and put it in the **com/dotComIt/learnWith/views/tasks** directory. The file contains a single button representing the edit button:

```
<button ng-click="grid.appScope.onEditTask(row.entity)">Edit</button>
```

When clicking the button, a method named **onEditTask()** will be called in the **MainScreenCtrl**. Before looking at the click handler, let's look at the modified **columnDefs** of the **taskGridOptions** object. The **taskGridOptions** are defined in the **MainScreenCtrl**:

```
columnDefs: [
  {
    field: 'completed',
    displayName: 'Completed',
    cellTemplate:'com/dotComIt/learnWith/views/tasks/CompletedCheckBoxRenderer.html'
  },
  {field: 'description', displayName: 'Description'},
  {field: 'taskCategory', displayName: 'Category'},
  {field: 'dateCreated', displayName: 'Date Created'},
  {field: 'dateScheduled', displayName: 'Date Scheduled'},
  {
    name: 'Controls',
    displayName : '',
    enableSorting : false,
    cellTemplate:'com/dotComIt/learnWith/views/tasks/EditButtonRenderer.html'
  },
]
```

The first five columns remain unchanged. The fifth column is a new one. The header is defined using the **displayName** property. It is left blank. The **uiGrid** requires either a name or field value for each column. Since this column is not displaying a field, I gave the column a name of controls. Since this is just a button column, the **enableSorting** is turned off. The main property of note is the **cellTemplate**. That one line of code will add the edit button inside the task grid.

Finally, look at the **onEditTask()** method:

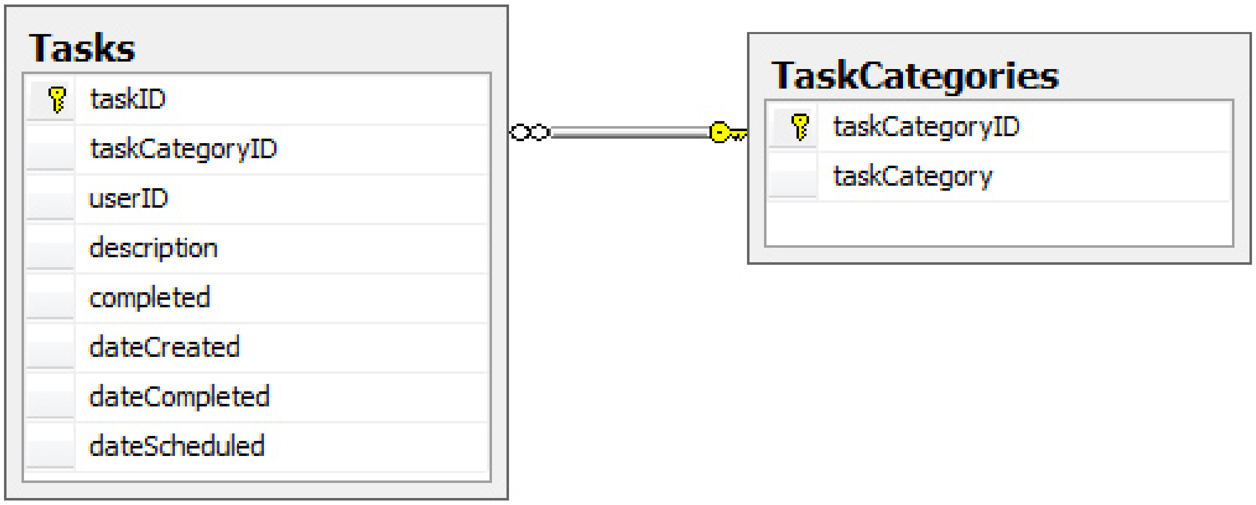
```
$scope.onEditTask = function onEditTask(task){
    openTaskWindow('Edit Task',task);
}
```

This method goes in the **MainScreenCtrl.js** file. The **onEditTask()** method accepts a single task as the argument. It then calls the **openTaskWindow()** method; passing the modal title and the task as arguments.

After implementing this last bit, the app should run and open the dialog popup for both a new task and for editing an existing one. The next steps in this chapter are to review the services that we need to integrate with, and to hook up the save button in order to complete the UI.

Examine the Database

Before delving into the service code, I wanted to provide you a refresher on the database tables behind the tasks. You have seen this diagram used in previous chapters, but here it is again:



This chapter will only be dealing with the Tasks table, and only a few columns at that. This table shows the list of columns, and how they are affected by the create task or update task:

Data	On Create	On Update
taskID	Created by the database	Passed along by the UI with no user input
taskCategoryID	User editable	User editable
userID	Passed along by the UI with no user input	Not changed during update
Description	User editable	User editable
completed	Defaulted to false	Not changed during a task update, although this functionality to toggle this value will be added to the app in later chapters
dateCreated	Defaulted to current date with no user input	Not changed during update
dateCompleted	Set to null with no user input	Not changed during a task update, although this functionality will be added to the app in later chapters
dateScheduled	Set to null with no user input	Not changed during task update, although this functionality will be added to the app in later chapters

These are the columns that get updated when we implement the services which create and update our task data.

Write the Services

This section will focus on the services for creating and editing tasks. Two new service methods will be created in the service layer. The user interface will only create a single service method, using the **taskID** property to determine whether the data change is creating a new task or updating an existing one.

Modify the getFilteredTasks() method

Before looking at the code to update or create a task, we need to make a change to the **getFilteredTasks()** method. The **getFilteredTasks()** method was created in Chapter 3, and expanded upon in Chapter 4. It loads an array of tasks based on some specified filters. We are going to add a filter for the **TaskID** now. This will make it easy to return the new or updated task after the created or update process is completed.

Open up the **TaskService.cfc** in the **com/dotComIt/learnWith/services** directory. Add this new clause as part of the database query:

```
<cfif StructKeyExists(local.json,'taskID')>
      <cfif setWhere is true >where <cfset setWhere = 'false'></cfif>
      <cfif firstOne is true><cfset firstOne = false><cfelse>and</cfif>
      taskID = <cfqueryparam cfsqltype="cf_sql_numeric"
                             value="#local.json['taskID']#">
</cfif>
```

The code examines the **json** variable to see if the **taskID** is in it. If it is, then it adds a check to the SQL "where" clause to check for **taskID** equality.

Creating a New Task

When creating the **createTask()** method and the **updateTask()** service methods, I decided to use individual arguments instead of sending in a full task object, converted to a JSON string. This is because much of the task data is not being changed, so this approach minimizes the amount of data sent over the wire.

This is the signature for **createTask()** method:

```
<cffunction name="createTask" returnformat="plain"
            access="remote" output="true"
            required="true" hint="I save, or update, a single task." >
      <cfargument name="taskCategoryID" type="numeric" required="true" />
      <cfargument name="userID" type="numeric" required="true" />
      <cfargument name="description" type="String" required="true" />
</cffunction>
```

There are three arguments to the **createTask()** method; the **taskCategoryID**, the **userID**, and the **description**. All attributes are required. The method body will use a database query to insert the values into the database, select the new **TaskID**, and return the data in JSON format.

First, the query:

```
<cfquery datasource="#application.dsn#" name="local.dataQuery" >
  insert into tasks(
    taskCategoryID, userID, description, completed, dateCreated
  ) values(
  <cfif taskCategoryID NEQ 0>
      <cfqueryparam value="#arguments.taskCategoryID#"
                    cfsqltype="cf_sql_integer" >,
  <cfelse>
```

```
      <cfqueryparam null="true" cfsqltype="cf_sql_integer" >,
  </cfif>
  <cfqueryparam value="#arguments.userID#" cfsqltype="cf_sql_integer" >,
  <cfqueryparam value="#arguments.description#" cfsqltype="cf_sql_varchar" >,
  <cfqueryparam value="0" cfsqltype="cf_sql_bit" >,
  <cfqueryparam value="#createODBCDateTime(now())#"
                cfsqltype="cf_sql_timestamp" >
  )
  SELECT SCOPE_IDENTITY() as TaskID
</cfquery>
```

This is a standard SQL Server database query for an insert. It starts with the “insert” statement, and specifies which table the data is inserted into. In this case, you use the tasks table, and then a list of all fields to be updated.

The “values” keyword comes next, followed by the list of values. After the keyword, **taskCategoryID** comes first. If the **taskCategoryID** is “0”, then a “null” value is passed to the database. Next, you’ll see the **userID** and description values. The first three values are from the arguments in the method. The next two values, **completed** and **dateCreated**, are hard-coded values and do not come from user input. The **completed** column is set to “0”, which means that the task is not yet completed. The **dateCreated** is set to the current date.

The last piece of the query makes use of a SQL Server function to get the identity column of the record recently created; the new task. There are other ways to do this that may or may not work with your database of choice.

The next step is to take the results from the query—which will be the **taskID**—and use that to get the full task object which you can return as the results of the service:

```
<cfset tempObject =
    createObject('component','com.dotComIt.learnWith.vos.TaskFilterVO')/>
<cfset tempObject['taskID'] = local.dataQuery.TaskID />
<cfset JSONedObject = SerializeJSON(tempObject) />
<cfset results = getFilteredTasks(JSONedObject)>
<cfreturn results />
```

First a new **TaskFilterVO** object is created. The **taskID** property on it is set. The object is converted to JSON with ColdFusion’s **SerializeJSON()** method. Then, the resulting JSON is used as the input for the **getFilteredTasks()** method. If you recall, the **getFilteredTasks()** method was introduced in Chapter 3, and modified in Chapter 4. It seemed to make sense to reuse the existing methods instead of writing new code. The values returned from this method will be a **ResultObjectVO** with the **resultObject** property, a **TaskVO** object, converted to JSON.

Test Creating a Task

To test this, just create a simple CFM page that calls the **createTask()** method:

```
<h1>Create Task</h1>
<cfset service =
 createObject('component','com.dotComIt.learnWith.services.TaskService')>
```

```
<cfset results = service.createTask(1,1,'created by Test Harness')>
Results: <Br/>
<cfdump var="#results#" expand="false" />
```

This page creates an instance of the **TaskService** class, then calls the **createTask()** method. The first argument passed in is the **taskCategoryID** “1”, which stands for business tasks in our test data. The second argument relates to the **userID**. The final argument is the task description. The results from this call are the following:

```
{
 "resultObject":
 [
  {
   "taskCategoryID":1,
   "description":"created by Test Harness",
   "dateScheduled":"",
   "taskCategory":"Business",
   "dateCompleted":"",
   "taskID":13,
   "dateCreated":"05\/14\/2013",
   "completed":0,
   "userID":1
  }
 ],
 "error":0.0
}
```

This is an array of **TaskVO**, converted to JSON. Because you sent in a **taskID** as the filter criteria, you can safely assume only a single entity will be returned in the array.

Updating an Existing Task

The next service method to examine is the one to update a task. This method is, conceptually, similar to the method for creating a new task. Here is the method signature:

```
<cffunction name="updateTask" returnformat="plain" access="remote"
            output="true" >
      <cfargument name="taskID" type="numeric" required="true" />
      <cfargument name="taskCategoryID" type="numeric" required="true"/>
      <cfargument name="description" type="String" required="true" />
</cffunction>
```

The method has three arguments. The description and **taskCategoryID** are arguments similar to the **createTask()** method. Instead of the **userID** used in **createTask()**, the third argument is the **taskID**. This represents the unique task that needs to be updated. Here is the query code:

```
<cfquery datasource="#application.dsn#" name="local.dataQuery"
         result="local.results">
  update tasks
  set
  <cfif taskCategoryID NEQ 0>
```

```
   taskCategoryID = <cfqueryparam value="#arguments.taskCategoryID#"
                                  cfsqltype="cf_sql_integer">,
  <cfelse>
   taskCategoryID = null,
  </cfif>
  description = <cfqueryparam value="#arguments.description#"
                              cfsqltype="cf_sql_varchar">
  where taskID = <cfqueryparam value="#arguments.taskID#"
                               cfsqltype="cf_sql_integer">
</cfquery>
```

This query is a standard database update query. It has the update keyword, followed by the table that is being updated. Then, the set keyword is there, followed by the list of columns being updated.

A conditional is added to the **taskCategoryID** query parameter. If the value is not “0”, then the provided value is used. If it is “0”, then change the **taskCategoryID** to “null”. This means it will have no value. The description argument is changed based on what the user entered. A “where” clause is on the **update** statement, using the **taskID** to make sure only a single task is updated by this method.

The rest of the method processes the results of the query and returns a **ResultObjectVO** with a **TaskVO** inside it as the **resultObject** property. This section of code is almost identical to the method for saving a task:

```
<cfset tempObject =
    createObject('component','com.dotComIt.learnWith.vos.TaskFilterVO')/>
<cfset tempObject.taskID = arguments.taskID />
<cfset JSONedObject = SerializeJSON(tempObject) />
<cfset results = getFilteredTasks(JSONedObject)>
<cfreturn results />
```

The only difference these last few lines have between updating and saving a task is that the **taskID** argument of the **TaskFilterVO** instance comes from the arguments sent in, not from the results of the query.

Test Updating a Task

The script to test the task update uses a simple approach we’ve seen previously. Create an instance of the service, call the method, and output the results. The file is named **UpdateTask.cfm** and is in the **com/dotComIt/learnWith/tests** directory:

```
<h1>Update Task</h1>
<cfset service =
 createObject('component','com.dotComIt.learnWith.services.TaskService')>
<cfset results = service.updateTask(1,2,'Get Milk')>
Results: <Br/>
<cfdump var="#results#" expand="false" />
```

This snippet will update the task with a **TaskID** of “1”. The category will be set to “Personal”, and the text will be set to “Get Milk”. You’ve probably seen this task show up in various screenshots in this book.

This is the resulting JSON output from the code snippet:

```
{
 "resultObject":
  [
   {
    "taskCategoryID":2,
    "description":"Get Milk",
    "dateScheduled":"",
    "taskCategory":"Personal",
    "dateCompleted":"",
    "taskID":1,
    "dateCreated":"03\/27\/2013",
    "completed":0,
    "userID":1
  }
 ],
 "error":0.0
}
```

The **resultObject** property contains the updated **TaskVO**, with all related properties. The error property of the **ResultObjectVO** instance is set to “0”. This means no error occurred.

Review the API

To integrate this service in the JavaScript applications, you'll need to know the endpoint to use, and the arguments to pass in. In this case, both the "create service" and "update service" use the same services endpoint seen in previous chapters. If the app and service are served off the same domain you can use a relative URL for the endpoint. In this case, I used this:

```
/com/dotComIt/learnWith/services/TaskService.cfc
```

The **createTask()** method will need these three end points:

- **method**: This argument will tell the ColdFusion application server which method to execute at the endpoint. In this case it will be "createTask".
- **taskCategoryID**: This argument will be the primary key for the category, and is a value the user will select in the UI.
- **userID**: This argument will be the **userID** which the UI Stored after successful authentication.
- **description**: This argument will contain the main text of the task, as entered by the user.

The arguments needed for updating a task are

- **method**: This argument will tell the ColdFusion application server which method to execute at the endpoint. In this case it will be "updateTask".
- **taskID**: This argument will be the primary key of the task, which was loaded into the UI via the **getFilteredTasks()** function.
- **taskCategoryID**: This argument will be the primary key for the category, and is a value the user will select in the UI.
- **description**: This argument will contain the main text of the task, as entered by the user.

In both cases, the arguments you need to send to the service mirror the arguments defined in the ColdFusion code.

Access the Services

This section will cover the JavaScript code needed to integrate with the service for updating or creating a task using an AngularJS **post()** method. Open up the **TaskService.js** file from **com/dotComIt/learnWith/services/coldFusion**. We are going to add a new method to the **taskService** object; **updateTask()**:

```
function updateTask(taskVO, user){
```

This method accepts two arguments; the task object being updated, and the user who updated the task.

This method is going to have to determine whether the **createTask()** or **updateTask()** service method is called. It can do so by checking the **taskID** value of the **taskVO** object. If the **taskID** is “0”, then you’ll need to call the **createTask()** service method. Otherwise, you’ll have to call the **updateTask()** service method.

The first piece of code in the method determines which item to call:

```
var method = "createTask";
if(taskVO.taskID != 0 ){
    var method = "updateTask";
}
```

The code creates a variable named “method”. The default value is **createTask**, for creating a new task. If the **taskVO**’s **taskID** value is not equal to “0”, then the method is changed to **updateTask**. This value will be used in the parameter object.

Next, we want to make sure that the **taskCategoryID** was selected:

```
if(!(typeof taskVO.taskCategoryID === 'number' )){
    taskVO.taskCategoryID = 0;
}
```

In certain situations, the UI may not force the user to select a task, and the **taskCategoryID** will therefore be empty. We want to make sure we send the service layer a “0” **taskCategoryID** in those situations, not an empty string.

Then create the parameter object:

```
var parameters = {
    method : method,
    taskCategoryID : taskVO.taskCategoryID,
    description : taskVO.description,
    taskID : taskVO.taskID,
    userID : user.userID
}
```

The parameter object defines all the parameters that will be passed to the remote service. The first one is the method variable. Then, the **taskCategoryID** and description are defined. These are values required for both the **createTask()** and **updateTask()** methods.

The last two parameters are the semi-optional parameters; the **taskID** and the **userID**. The **taskID** is required for the **updateTask()** method, but not for the **createTask()** method. The **userID** is required for the **createTask()** method, but not for the **updateTask()** method. Thankfully, passing the non-required parameters has no effect on the backend. They will just be ignored. Thus, we can include the parameters with every request without any adverse effect on the server side.

The last step on this method is to make the remote call:

```
return $http.post('/com/dotComIt/learnWith/services/TaskService.cfc',
           parameters )
```

The Angular **$http** variable is used to make the **post()** call. The promise object from the **post()** is returned from the service. This is so that the invoking code can call a "result" or "failure" function when the asynchronous call is completed.

Be sure to add this new **updateTask()** method to the **TaskService**'s services object:

```
var services = {
    loadTasks : loadTasks,
    loadTaskCategories : loadTaskCategories,
    updateTask : updateTask
}
```

As a reminder, the **TaskService.js** file should already be referenced in the **index.html** file if you followed code from previous chapters. In our code archive, you will find that the **Index_ColdFusion.html** file already has the proper script included.

Wire Up the UI

The final section of this chapter will wire up the popup dialog to the services. This will allow for the creation and update of the tasks. It will also determine how to update the main task grid with the results of the popup interactions.

Clicking the Save Button

To start, we will flesh out the **onSave()** method in the **TaskCUCtrl** file. This method is executed when the save button is clicked inside the task popup. This method only needs to call the **updateTask()** method in the **TaskService**.

The heavy lifting is handled inside the service:

```
$scope.onSave = function onSave () {
   TaskService.updateTask($scope.taskVO,UserModel.user)
              .then(onTaskUpdateSuccess, onTaskUpdateError)
}
```

The **updateTask()** service method returns a promise object. Using that promise object, we line up to two result methods with the **then()** function. On a successful result, the **onTaskUpdateSuccess()** method will be executed. If there is a problem, the **onTaskUpdateError()** method will be called. These are both methods defined in the **TaskCUCtrl.js** file.

Error handler methods appear to be shorter, so this is what happens if there is an error:

```
function onTaskUpdateError(response){
    alert(response.data);
}
```

The code merely displays an alert to the user with the results. This method does not close the popup, so that the user can attempt to resubmit their changes.

The **onTaskUpdateSuccess()** method is a bit more complicated than the **onTaskUpdateError()** method. It must process the returned value to see if there was an error. If there was an error, it will display a message to the user. If there was no error; it will close the popup and notify the parent controller of the new object.

First, this is the method signature:

```
function onTaskUpdateSuccess(response){
}
```

The first step in the method is to make sure that no error exists:

```
if(response.data.error == 1){
    alert("We could not create the task data");
    return;
}
```

The data argument is the value returned from the server side, which should be an instance of the **ResultObjectVO**. If the error value of it is “1”, then an error has occurred. A warning message is displayed to the user in that case. If no error occurred, then the popup is closed:

```
$uibModalInstance.close(response.data.resultObject[0]);
```

The **$modalInstance** is a value passed into the controller and is a reference to the popup. The **close()** method tells the **MainScreenCtrl** that the was closed successfully. It passes the updated object as an argument to the parent’s **close()** method. The next step is to execute methods when the popup is closed or dismissed.

Handling the createTask() Result

Open up the **MainScreenCtrl.js** file and find the **openTaskWindow()** method. This method was created earlier in this chapter and it creates a popup. After the popup is created, there is some additional code that needs to be implemented. This new code will allow you to execute a function when the popup is closed or dismissed.

First, you need to determine which result method you want to execute:

```
var resultFunction = updateTaskComplete;
if(task.taskID == 0){
    resultFunction = createTaskComplete;
}
```

The code creates a variable named **resultFunction**. It will refer to other methods defined in the controller. The **resultFunction** variable is defaulted to **updateTaskComplete()** which will be the method to execute when the update task process is completed. If the task’s **taskID** value is “0”, then a new task is being created. In this case, the **createTaskComplete()** method will be executed instead.

The reason for two different methods is that, when the task is updated, we want to update the task displayed in the grid. When a new task is created, we want to add it to the current grid. However, each process needs a different implementation.

The last piece of this method handles what happens when the popup is closed:

```
modalInstance.result.then(resultFunction, onPopUpDismissal);
```

This uses the **modalInstance** variable, which was returned from the **$uibModal.open()** method. It accesses the result property, which is a promise object. It uses the **then()** method to execute success or failure functions. The result method is listed first, and is followed by the dismissal method.

In our code we aren’t doing anything when the popup is cancelled, but I left the method there for the sake of demonstration:

```
function onPopUpDismissal(){
    console.log('Modal dismissed at: ' + new Date());
}
```

It logs some output to your browser console so you know that the popup was canceled. Next comes the **createTaskComplete()** method implementation:

```
function createTaskComplete (updatedTask) {
    TaskModel.tasks.push(updatedTask);
}
```

This method adds the newly created task to the **taskGrid**. It does so by accessing the tasks array in the **TaskModel**. It also uses the **push** method to add the new item to the array. This will automatically update the **uiGrid** to add the new task.

Handling the updateTask() Result

The final task of this chapter is to implement the **updateTaskComplete()** method. This method will execute after the edit task popup is closed. The purpose of this method is to find the task being edited in the grid's **dataProvider** and replace it.

```
updateTaskComplete = function updateTaskComplete (updatedTask) {
 for (index = 0; index < TaskModel.tasks.length;index++){
  if(TaskModel.tasks[index].taskID == updatedTask.taskID){
     TaskModel.tasks[index] = updatedTask;
     break;
  }
 }
}
```

The function loops over all the tasks in the **TaskModel**'s array. When it finds the one with the appropriate **taskID** it updates it with the new item and breaks out of the loop. This will update the data in the grid.

Final Thoughts

This is the process going on with the popup:

1. User loads app and logs in.
2. They click the “Edit Task” or “New Task” button.
3. Methods inside the **MainScreenCtrl** will open the popup dialog. If creating a new task, then an empty **TaskVO** object is created. The title value and the task object are passed into the dialog.
4. If the user clicks the “cancel” button then the **dismiss()** function is called, closing the popup and executing the **onPopUpDismissal()** method in **MainScreenCtrl**.
5. Otherwise the user performs their edits and clicks the “save” button. In this case the remote service is called to save—or update— the task from the **TaskCUCtrl**.
6. The results come back and the popup is closed.
7. The **MainScreenCtrl** executes the successful closure method for updating or creating the task. On the **createTaskComplete()** method, a new task will be added to the task grid. On the **updateTaskComplete()** method, the grid task is updated with its new values.

The next chapter will implement the ability to schedule tasks

Chapter 6: Scheduling Tasks

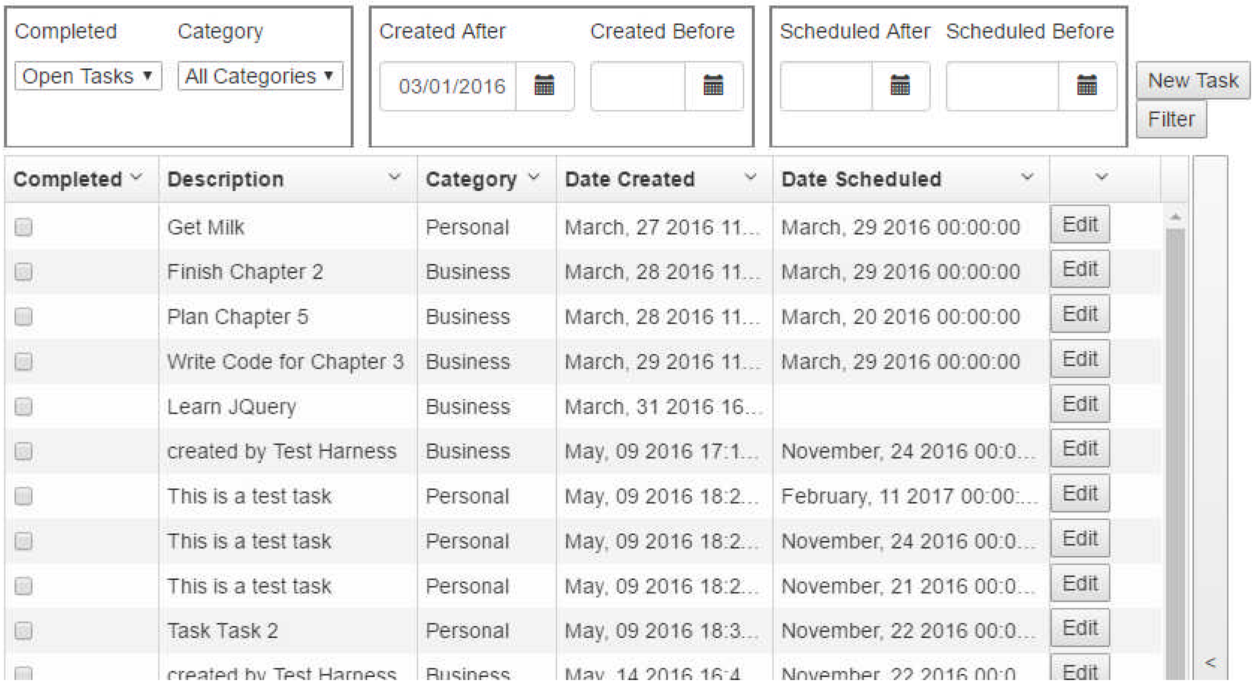
This chapter will implement the ability to schedule tasks inside an app. That means being able to assign a task to a specific day. It will show you the user interface for this and discuss the services required. Then, it will wire everything up to make things work, and revisit the service method for loading task data. This chapter will also introduce a few new angular directives; **ngClass**, **ngShow**, and **ngRepeat**.

Create the User Interface

This section will show you the user interface for scheduling tasks. It will show you the implementation, and we'll also revisit some of our past CSS Styles to tweak the layouts.

The Task Scheduler Window

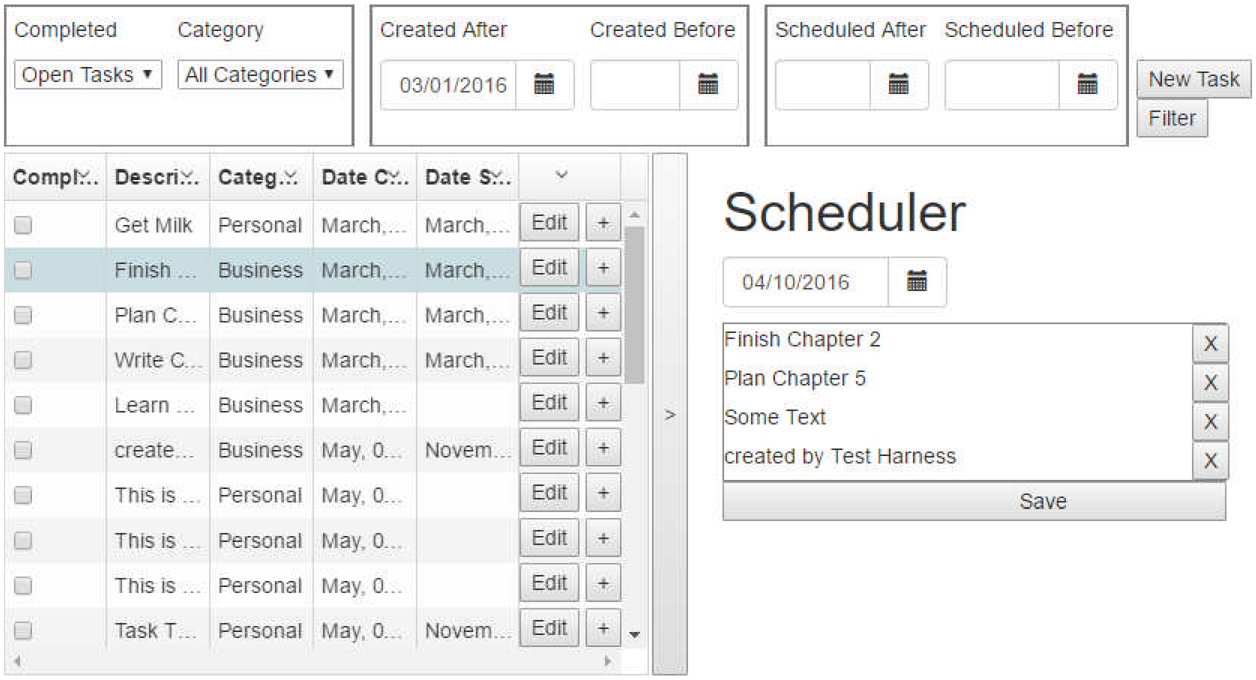
The scheduler will be shown, or hidden, based on a button click. The first change to the Main screen is to add that particular button:



Completed	Category	Created After	Created Before	Scheduled After	Scheduled Before
Open Tasks	All Categories	03/01/2016			

Completed	Description	Category	Date Created	Date Scheduled	
	Get Milk	Personal	March, 27 2016 11...	March, 29 2016 00:00:00	Edit
	Finish Chapter 2	Business	March, 28 2016 11...	March, 29 2016 00:00:00	Edit
	Plan Chapter 5	Business	March, 28 2016 11...	March, 20 2016 00:00:00	Edit
	Write Code for Chapter 3	Business	March, 29 2016 11...	March, 29 2016 00:00:00	Edit
	Learn JQuery	Business	March, 31 2016 16...		Edit
	created by Test Harness	Business	May, 09 2016 17:1...	November, 24 2016 00:0...	Edit
	This is a test task	Personal	May, 09 2016 18:2...	February, 11 2017 00:00:...	Edit
	This is a test task	Personal	May, 09 2016 18:2...	November, 24 2016 00:0...	Edit
	This is a test task	Personal	May, 09 2016 18:2...	November, 21 2016 00:0...	Edit
	Task Task 2	Personal	May, 09 2016 18:3...	November, 22 2016 00:0...	Edit
	created by Test Harness	Business	May, 14 2016 16:4	November, 22 2016 00:0	Edit

The button to show the scheduler is on the right of the screen. After pressing the button, you'll see this:



Completed	Category
Open Tasks	All Categories

Created After	Created Before
03/01/2016	

Scheduled After	Scheduled Before

Compl…	Descri…	Categ…	Date C…	Date S…
	Get Milk	Personal	March,…	March,…
	Finish …	Business	March,…	March,…
	Plan C…	Business	March,…	March,…
	Write C…	Business	March,…	March,…
	Learn …	Business	March,…	
	create…	Business	May, 0…	Novem…
	This is …	Personal	May, 0…	
	This is …	Personal	May, 0…	
	This is …	Personal	May, 0…	
	Task T…	Personal	May, 0…	Novem…

Scheduler

04/10/2016

Finish Chapter 2
Plan Chapter 5
Some Text
created by Test Harness

The grid shrinks down to half the screen. The Scheduler is shown with the date chooser and the list of tasks to be scheduled. There is a “save” button now below the scheduler, and a row of “+” buttons added into the grid beside each “edit” button. This “+” button only shows up when the scheduler is displayed. It is used to add an item to the scheduler.

These are the screens and functionality that you’ll implement in this chapter.

Modifying the Main Screen

The first step in the process is to modify the **MainScreen.html** template in the **com/dotComIt/learnWith/views/tasks** directory. This is the template that is displayed after the user logs in. It displays the **TaskFilter.html** template and the **TaskGrid.html** template. This section will add the button used to expand and collapse the scheduler. It will also add the scheduler template.

Before jumping into that, we are going to add some style changes. It was a bit tricky to get the “expand/collapse” button to spread the full height of the grid. It is easy to set “100%” height to the button, but that puts it to “100%” height of its container and that sort of bubbles up to the root HTML tag. So, the first style changes will set the HTML and Body tags to “100%” height and width. These changes go into the **style.css** file located in the **com/dotComIt/learnWith/styles** directory:

```
html{
    height:99%;
    width:100%;
}
body{
    height:100%;
    width:100%;
    padding:5px;
}
```

Since the styles are set on the element, not through a class, no additional work is needed for them to apply. Next up is the **ngView div** inside the **index.html** file. This is the **div** that will display the contents defined by the **routerProvider** based on the URL. This will need a new style, and for that we created a custom CSS class:

```
.horizontal-layout-100x100{
    height:100%;
    width:100%;
}
```

The class is named **horizontal-layout-100x100**. The height and width are set to "100%" Put it on the **ng-view div**:

```
<div ng-view class="horizontal-layout-100x100"></div>
```

This will expand the main application view to the full height and width of the browser.

Now you can turn your attention to the **MainScreen.html** file in the **com/dotComIt/learnWith/views/tasks** directory. Currently, the file should only have two **ngInclude** directives in it; one for the **TaskFilter** template, and one for the **TaskGrid** template. They are listed side by side. First, we're going to add a new style class—**taskFilter**—to the **TaskFilter** component:

```
<div ng-include="'com/dotComIt/learnWith/views/tasks/TaskFilter.html'"
     class="taskFilter">
</div>
```

This is the style definition from **styles.css**:

```
.taskFilter{
    height:22%;
    width:100%;
    max-height:100px;
}
```

The height of the **taskFilter** is set to "22%" and the width is set to "100%". The max-height is set to **100px**. The remaining application's height will be available for the grid and scheduler.

A new **div** is needed to encompass the bottom half of the screen:

```
<div class="mainScreenContainer">
</div>
```

The style class **mainScreenContainer** sets the height and width:

```
.mainScreenContainer{
    height: calc(100% - 100px);
    width:100%;
}
```

The width is set to "100%". For the height, I used the CSS **calc()** function. This calculates and specifies the height to "100%" of the available screen, but leaves room for the task filter portion of the app.

Inside the **mainScreenContainer** is going to be three different elements; the **TaskGrid**, the "expand/collapse" button, and the scheduler. First, examine the **taskGrid**—which you have seen before.

Change the **MainScreen.html** file in the same directory:

```
<div ng-include="'com/dotComIt/learnWith/views/tasks/TaskGrid.html'"
     ng-class="gridContainerStyle">
</div>
```

There is a new Angular directive on the **TaskGrid** include; **ngClass**. The **ngClass** directive allows you to set a style class on the **div** by using a variable defined in the controller. There are two different CSS classes that will be switched between, each representing a different width. The first is named **horizontal-layout-94**:

```
.horizontal-layout-94{
    position: relative;
    display: inline-block;
    vertical-align: top;
    width: 94%;
    min-height: 100%;
}
```

This class will be used in the default state when the scheduler is hidden. The style sets the **position** to "relative". This means that the element will be positioned based on its parent container. It also sets the **display** to "inline-block". Divs are usually placed as vertical elements; one on top of the other. By setting the **display** style to "inline-block", they can be put side by side. The **vertical-align** is set to "top", so the grid will be positioned at the top of the container. The width is set to "94%", which leaves room for the "expand/collapse" button. Finally, the **min-height** is set to "100%". This will expand the height of the grid to fill up the available space.

The second style is **horizontal-layout-40**. This style cuts the width of the layout in half to make room for the scheduler component:

```
.horizontal-layout-40{
    position: relative;
    display: inline-block;
    vertical-align: top;
    width:40%;
    min-height: 100%;
}
```

The styles will be the value to the **gridContainerStyle** variable in the **MainScreenCtrl.js** file. The file is in the **com/dotComIt/learnWith/controllers** directory. Open it up to find the definition:

```
$scope.gridContainerStyle = 'horizontal-layout-94';
```

When the button is clicked, this variable will be able to toggle between the two values—changing the width of the grid. This button is what comes next in the file:

```
<div class="horizontal-layout-4">
  <button class="height100" ng-click="onToggleScheduler()">
```

```
    {{schedulerShowButtonLabel}}
  </button>
</div>
```

The button is in a **div**. It uses the class **horizontal-layout-4**. This is the CSS Class:

```
.horizontal-layout-4{
    display: inline-block;
    vertical-align: top;
    width: 4%;
}
```

The class gives a small **width**, but a “100%” **height**. It makes the button **div** the same height as the task grid. The button itself has a simple style:

```
.height100 {
    height:100%
}
```

The style stretches the height of the button to the full height of the **div** which it is enclosed in. The button’s label is an Angular variable defined in the **MainScreenCtrl.js** file:

```
$scope.schedulerShowButtonLabel = "<"
```

The default value for the button is the less-than sign; “<”. When the button is clicked, and the screen is re-oriented, the label will change to the greater-than sign; “>”. This is intended to symbolize that clicking the button will expand things to the left because the less-than sign looks like a left pointing arrow. Clicking the button in the expanded state will expand things to the right because the greater-than sign resembles an arrow pointing right.

The last element of the **mainScreenContainer div** is the scheduler component:

```
<div class="horizontal-layout-40"
     ng-include="'com/dotComIt/learnWith/views/tasks/Scheduler.html'"
     ng-show="scheduler.schedulerState" >
</div>
```

The **div** is styled with a CSS class named **horizontal-layout-40**. This is the same class used for the task grid in the expanded state, so I won’t repeat the instructions on it. Then there is an **ngInclude** which contains the scheduler component. The **Scheduler.html** file will be examined a bit later in this chapter.

The final element of the **div** is the **ngShow** directive. This directive determines whether the **div** should be shown or hidden based on a variable in the controller. This is the variable:

```
$scope.scheduler = {
    schedulerState : false,
}
```

The variable is inside the object so that it can be accessed inside the task grid’s include file, as you will see later on. The default value is “false”, which means the scheduler component is hidden by default.

Clicking the Expand Button

What happens when the expand button is clicked? Well, a method in the **MainScreenCtrl** is executed when this happens—**onToggleScheduler()**. This method will perform three tasks:

- Changes the **gridContainerStyle** to shrink or expand the task grid accordingly.
- Changes the button's label to display one that represents "expand", or the one that represents "collapse".
- Changes the scheduler state variable which will display the scheduler component.

All in all, the method is fairly simple:

```
$scope.onToggleScheduler = function onToggleScheduler(){
    if($scope.scheduler.schedulerState == true){
        $scope.scheduler.schedulerState = false;
        $scope.gridContainerStyle = 'horizontal-layout-94';
        $scope.schedulerShowButtonLabel = "<"
    } else {
        $scope.scheduler.schedulerState = true;
        $scope.gridContainerStyle = 'horizontal-layout-40';
        $scope.schedulerShowButtonLabel = ">"
        loadSchedulerTasks();
    }
}
```

The method is toggling a bunch of different variables here. First, it checks the current scheduler state. If the value is "true", it is changed to "false". Then, the method changes the **gridContainerStyle** to the longer style and changes the scheduler button's label to "<". If the current scheduler state is "false", then it changes to "true", sets the **gridContainerStyle** to the smaller width style, and sets the button's label to ">". If the scheduler is being shown, a method—**loadSchedulerTasks()**—is called to load the tasks for the given date. This method will be examined in more detail later in this chapter.

If you try the code as is, you'll notice one major problem; the task grid is not resizing when the state changes. However, resizing the browser window fixes it immediately. How can we fix this? My solution is two pronged. First, we need a hook into the **gridApi**:

```
$scope.taskGridOptions.onRegisterApi = function(gridApi){
  $scope.gridApi = gridApi;
}
```

The **onRegisterApi** function is called when the **uiGrid** is initialized. By saving the **gridApi** reference into the **$scope**, we can easily access it elsewhere in this controller

Then, we can add this at the end of the **onToggleScheduler()** method:

```
$timeout(function() {
    $scope.gridApi.core.handleWindowResize();
},1);
```

I found similar approaches suggested multiple times when researching the issue. This causes the app to think that the window has resized even though it has not. It is a kludgy fix, but it works

Adding the Schedule Button to the TaskGrid

In the expanded scheduler state, the task grid shows an extra button in the last column. This new button is a "+" which will be used to add the current task into the scheduler. The button is defined in a cell template defined in the **taskGridOptions**. Since we're adding the new button to an already existing cell template, there is no need to review the **taskGridOptions**.

Open up the **EditButtonRenderer.html** file in the **com/dotComIt/learnWith/views/tasks** directory. Add this new button on the line after the "edit" button:

```
<button ng-click="grid.appScope.onScheduleTask(row.entity)"
        ng-show="grid.appScope.scheduler.schedulerState">
  +
</button>
```

The button uses the **ngShow** directive to determine when it should be hidden or displayed. It ties into the **schedulerState** variable of the **MainScreenCtrl**. This state variable is used to hide the button when the scheduler template is hidden, or show it when the scheduler template is displayed. The button also has an **ngClick** handler to call the **onScheduleTask()** method when the button is clicked.

The purpose of the **ngClick** handler is to add the task to the currently displayed schedule. Since it does not rely on a remote service, let's look at the code behind the **onScheduleTask()** method now. But, before that, open up the **TaskModel.js** file from the **com/dotComIt/learnWith/model** directory. Two variables were added to the class:

```
addedTasks : [],
scheduledTasks : []
```

The first variable is a **scheduledTasks** array. This will be used to determine the tasks which should be displayed in the Scheduler component's list. The second array, **addedTasks**, is used to keep track of components that were added manually to the current day. If the day changes before items are saved we don't want to erase the currently added, but unsaved, items. It would be a frustrating user experience to remove all items the user just added before realizing they had selected the wrong date. The **addedTasks** array is used to keep track of these extra items and show them in the list after the date changes and new results are retrieved from the server.

Back to the **MainScreenCtrl** and the **onScheduleTask()** method:

```
$scope.onScheduleTask = function onScheduleTask(task){
  var found= false;
  for (var index = 0; index < TaskModel.scheduledTasks.length; index++){
    if(TaskModel.scheduledTasks[index].taskID == task.taskID){
          found= true;
          break;
    }
  }
  if(!found){
      TaskModel.scheduledTasks.push(task);
      TaskModel.addedTasks.push(task);
  }
}
```

The first step in the method is to verify that the item being added to the **scheduledTasks** list is not already in there. If it is, it should not be added again. This is done by looping over the **taskModel.scheduledTasks** array and comparing the **taskID** values. If they are equal, then a Boolean value is set to "true", indicating that the item was found.

After the loop, if the item was not found, then it is added to both the **ScheduledTasks** array and the **addedTasks** array. If the item was found, then nothing happens. This approach allows you to easily add tasks to the currently scheduled list.

The Scheduler Template

The final UI piece to build out for this chapter is the **Scheduler** component template. This includes four separate parts. The first is the text header which is trivial to create. The second is the date chooser template, which is no different than what we are already using in the **TaskFilter** template. Next up is the list of tasks that are scheduled, or need to be scheduled, for the current selected day. We'll spend some time explaining this. The final piece is the "save" button. We've seen a lot of AngularJS powered buttons, so this should be simple enough.

Create the **Scheduler.html** component in the **com/dotComIt/learnWith/views/tasks** directory. First, start with the header:

```
<h1>Scheduler</h1>
```

This is simple HTML and is doesn't include any specific AngularJS functionality.

The second piece is the current date **datePicker**:

```
<div class="form-horizontal">
    <input type="text" datepicker-popup="MM/dd/yyyy"
           ng-model="scheduler.schedulerDate"
           is-open="currentDateProperties.opened"
           datepicker-options="datePickerOptions"
           show-weeks="dateOptions.showWeeks"
           class="datePicker" ng-change="onSchedulerDateChange()"
            />
    <button class="btn" ng-click="openDatePicker(currentDateProperties)">
        <i class="icon-calendar"></i>
    </button>
</div>
```

I won't go into another explanation about the **datePicker**, as we have seen it in previous chapters, but there are a few things to make note of. The first is that this uses **ngModel** to bind to **scheduler.schedulerDate**. The **schedulerDate** must be defined as part of the scheduler object inside the **MainScreenCtrl** file:

```
$scope.scheduler = {
    schedulerState : false,
    schedulerDate : new Date()
}
```

This is the scheduler object from earlier in the chapter; when we initially defined the **schedulerState** value. It takes the same object, and adds a new property to it. Also inside the **MainScreenCtrl.js** file is a new scope variable named **currentDateProperties**:

```
$scope.currentDateProperties = {
   opened : false
};
```

This variable is used to determine when the popup window should be opened or not, and is triggered in the **openDatePicker()** function that is part of the calendar button. The final element to point out on this **datePicker** is the **ngChange** directive. It will call a brand new method in the controller whenever the date changes:

```
$scope.onSchedulerDateChange = function onSchedulerDateChange(){
    $scope.loadSchedulerTasks();
}
```

This **onSchedulerDateChange()** method merely calls the **loadSchedulerTasks()** method. This method will be implemented later in this chapter. It will load the new set of tasks scheduled for the newly selected day.

The next step is to display the list of tasks that are associated with the current day. Each list item is a **div** element and each contains two divs inside it. The first **div** encompasses the task description, while the second includes the “X” button—which is used for deleting a task. The whole thing is wrapped in a **div** that provides the border.

You can implement from the outside in, so here is the **div** that does nothing but provide the border:

```
<div class="border-top-left-bottom-right">
</div>
```

The previous code references a CSS Style, put in the **styles.css** file:

```
.border-top-left-bottom-right {
    border : solid 1px grey;
}
```

The style just defines a border. Inside the **div** is where the magic happens:

```
<div ng-repeat="task in taskModelWrapper.taskModel.scheduledTasks"
     class="width100">
  <div class="horizontal-layout-94">{{task.description}}</div>
  <div class="horizontal-layout-4">
     <button ng-click="onTaskUnschedule(task)">X</button>
  </div>
</div>
```

The first **div** uses the **ngRepeat** directive. This directive is like an HTML for loop. It tells AngularJS that for every task in the **scheduledTasks** array, it will create an instance of the **div**. The **div** has a CSS class of “width100”:

```
.width100 {
    width:100%
}
```

The style sets the width of the **div** to “100%”. Inside the **div** is another **div** to display the task description. The task variable referenced inside the repeat loop refers back to the task definition defined in the **ngRepeat**. You reference it the same way you would a value in the templates controller.

The second internal **div** contains the “delete” button. This new button is used to remove an item from the scheduled task list. It will, essentially, null out the tasks scheduled date. If the task was already scheduled, it will have to call a service to fix it. If not, then it just needs to be removed from scheduler display:

```
$scope.onTaskUnschedule = function onTaskUnschedule(task){
    if(task.dateScheduled){
        task.dateScheduled = null;
        scheduleTask(task);
    } else {
        deleteTaskFromSchedule(task);
    }
}
```

If the task has a **dateScheduled** value, then that value is set to “null” and the **scheduleTask()** method is called. The **scheduleTask()** method will be covered later in this chapter when we wire up the services. If the **dateScheduled** property has no value, then the **deleteTaskFromSchedule()** method is called:

```
function deleteTaskFromSchedule(task){
   var itemIndex = TaskModel.scheduledTasks.indexOf(task);
   if(itemIndex >= 0){
       TaskModel.scheduledTasks.splice(itemIndex,1);
   }
   itemIndex = TaskModel.addedTasks.indexOf(task);
   if(itemIndex >= 0){
       TaskModel.addedTasks.splice(itemIndex,1);
   }
}
```

The purpose of the **deleteTaskFromSchedule()** method is to remove the task from the **scheduledTasks** and **addedTasks** arrays stored in the **taskModel**. It does this by using the **indexOf()** method on the array to get the index. If the **itemIndex** has a value greater than “0”; it uses the **splice()** method to remove the item. The **splice()** method accepts two parameters; the **itemIndex** and the number of items to be removed. In this case, only one item is to be removed.

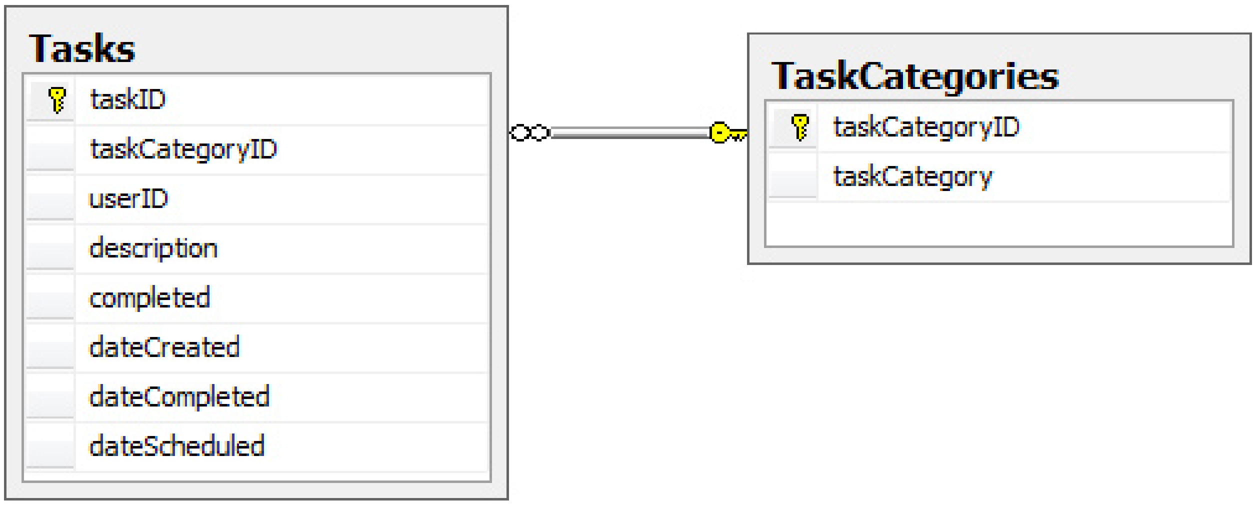
The final piece of the scheduler is the “save” button:

```
<button class="width100" ng-click="onTaskListSchedule()">Save</button>
```

The save button uses the **width100** CSS class which extends the button to “100%” width. It also calls a method, **onTaskListSchedule()**, which will save all items in the current scheduler list to the current date. After reviewing the services, we’ll explore this method in a bit more detail.

Examine the Database

The code in this chapter does not need any new tables, but I wanted to refresh your memory of the database structure:



The new services needed for this chapter deal with updating the **dateScheduled** property in the Tasks table. Two service methods will be created; one for editing a single task, and one for editing multiple tasks.

This is the SQL behind the editing of a single task:

```
update tasks
set dateScheduled = someDate
where taskID = someTaskID
```

This is the SQL behind editing multiple tasks:

```
update tasks
set dateScheduled = someDate
where taskID in (someCommaSeparatedListOfTasks)
```

The two SQL procedures are very similar, but the “where” clause is different.

Write the Services

This section will explain the services that help the UI assign or remove tasks to a specific day. There will be two services created; one to schedule or cancel a single task, and one to schedule a collection of tasks.

Update getFilteredTasks() Method

Before jumping into the ability to schedule tasks, we need to make one change to the **getFilteredTasks()** method. This is the service method that is used to load tasks based on certain criteria. In this case, we need to add a way to load all tasks based on a scheduled date. We'll do that by supporting a **scheduledEqualDate** property of the **taskFilter** argument.

First, open up the **TaskFilterVO** object from the **com/dotComIt/learnWith/vos** directory. This is the CFC representing all the properties that may be filtered on when loading tasks. Add the **scheduledEqualDate** property:

```
<cfproperty name="scheduledEqualDate" type="date" />
```

Now, move on to the **TaskService.cfc** method in the **com/dotComIt/learnWith/services** directory. Find the **getFilteredTasks()** method, and add this code to the database query:

```
<cfif StructKeyExists(local.json,'scheduledEqualDate')>
  <cfif setWhere is true >where <cfset setWhere = 'false'></cfif>
  <cfif firstOne is true><cfset firstOne = false><cfelse>and</cfif>
  dateScheduled = <cfqueryparam cfsqlType="cf_sql_date"
                                value="#local.json['scheduledEqualDate']#">
</cfif>
```

This performs a simple comparison against the **dateScheduled** column and the **scheduledEqualDate** property from our **TaskFilterVO** object. Adding these conditions to the server does not need any additional change to the UI code.

Schedule a Single Task

There is a new service method to schedule a single task, named **scheduleTask()**. The service goes in the **TaskService.cfc** class which is located in the **com/dotComIt/learnWith/services** package. The method signature accepts two arguments:

```
<cffunction name="scheduleTask" returnformat="plain" access="remote"
            output="true" >
      <cfargument name="taskID" type="numeric" required="true" />
      <cfargument name="dateScheduled" type="date" required="false" />
</cffunction>
```

The two arguments for the **scheduleTask()** method are the **taskID**, and the **dateScheduled**. The **taskID** is an integer representing the task that needs to be updated. The second argument is the date that the **dateScheduled** column in the database will be changed to. The **dateScheduled** argument is not required, and this is the first time in this book where a service method has had a non-required argument. If the argument is not specified, then the **dateScheduled** database column will be “null”; essentially removing the scheduled date from the task, thereby cancelling it.

Next in the method comes the database query:

```
<cfquery datasource="#application.dsn#" name="local.dataQuery"
         result="local.results">
  update tasks
 <cfif isDefined('arguments.dateScheduled')>
    set dateScheduled = <cfqueryparam
                         value="#createODBCDateTime(arguments.dateScheduled)#"
                         cfsqltype="cf_sql_timestamp" />
 <cfelse>
   set dateScheduled=<cfqueryparam null="true"
                                   cfsqltype="cf_sql_timestamp"/>
 </cfif>
   where taskID = <cfqueryparam value="#arguments.taskID#"
                                cfsqltype="cf_sql_integer" >
</cfquery>
```

The query is an update query, as shown earlier in this chapter. It uses the **cfqueryparam** tag to scrub the arguments from any nefarious user input.

This method will return a new task object. The easiest way to get that is to call the **getFilteredTasks()** method with the **taskID** as input:

```
<cfset tempObject =
    createObject('component','com.dotComIt.learnWith.vos.TaskFilterVO')/>
<cfset tempObject.taskID = arguments.taskID />
<cfset local.JSONedObject = SerializeJSON(tempObject) />
<cfset local.results = getFilteredTasks(local.JSONedObject)>
<cfreturn local.results />
```

This will finish off the service method for scheduling a single task.

Testing the Single Task Scheduler

I set up two tests for the scheduled task method. Both go in the **ScheduledTask.cfm** file in the **com/dotComIt/learnWith/tests** directory. The first will schedule a task, and the second will remove the scheduled date from a task. The process for testing the method is to create an instance of the service class, call the method, and output the results. This is the code that will test the scheduling of a task:

```
<h1>Schedule Task</h1>
<cfset service =
 createObject('component','com.dotComIt.learnWith.services.TaskService')>
<cfset results = service.scheduleTask(1,createDate(2013,3,29))>
Results: <Br/>
<cfdump var="#results#" expand="false" />
```

The output of the method call is a JSON array, representing the updated task:

```
{
 "resultObject":
 [
  {"taskCategoryID":2,
   "description":"Get Milk",
   "dateScheduled":"03\/29\/2013",
   "taskcategory":"Personal",
   "dateCompleted":"",
   "taskID":1,
   "dateCreated":"03\/27\/2013",
   "completed":0,
   "userID":1
  }
 ],
 "error":0.0
}
```

The code to cancel the task is similar:

```
<h1>Cancel Task</h1>
<cfset service =
 createObject('component','com.dotComIt.learnWith.services.TaskService')>
<cfset results = service.scheduleTask(1)>
Results: <Br/>
<cfdump var="#results#" expand="false" />
```

The only difference between them is the missing schedule date argument from the **scheduleTask()** method call. These are the results:

```
{
 "resultObject":
 [
  {"taskCategoryID":2,
   "description":"Get Milk",
   "dateScheduled":"",
   "taskcategory":"Personal",
   "dateCompleted":"",
   "taskID":1,
   "dateCreated":"03\/27\/2013",
   "completed":0,
   "userID":1
  }
 ],
 "error":0.0
}
```

You'll notice that the **dateScheduled** property in the results had a value in the original results, but it is an empty string after executing the second test. This proves that the method to schedule a single task can be used to both schedule a task and remove a task from the schedule.

Schedule a Lot of Tasks at Once

A second service method was needed to support the UI for scheduling tasks. This method is named **scheduleTaskList()**. It will be used for scheduling a lot of tasks at once and will be called when a save button is pressed on the scheduler component in the UI.

The method signature:

```
<cffunction name="scheduleTaskList" returnformat="plain" access="remote"
            output="true" >
      <cfargument name="taskIDList" type="string" required="true" />
      <cfargument name="dateScheduled" type="date" required="false" />
</cffunction>
```

The **scheduleTaskList()** method accepts two arguments. The first is a **taskIDList**. This is expected to be a comma separated list of **TaskIDs**. The second is the **dateScheduled**. As with the **scheduleTask()** method, **dateScheduled** is not required and if it is left out, all tasks will have their **scheduledDate** set to “null”.

The **taskIDList** comes into the method as a string, but you’ll need to convert it to a comma separated list of numbers. The first step of the method is to perform that conversion and validate that each **taskID** is indeed numerical:

```
<cfset local.taskIDsForSQL = ""/>
<cfloop list="#arguments.taskIDList#" index="local.tempTaskID" >
  <cfif isNumeric(local.tempTaskID)>
    <cfset local.taskIDsForSQL =
                         ListAppend(local.taskIDsForSQL,local.tempTaskID)/>
  </cfif>
</cfloop>
```

This code loops over the **taskIDList** argument. It verifies that each element of the list is numeric. If it is, then the code adds it to the **taskIDsForSQL** list variable. If not, the code ignores it. This, essentially, helps to scrub potentially invalid input. The **taskIDsForSQL** variable is used in the query statement, which comes up next:

```
<cfquery datasource="#application.dsn#" name="local.dataQuery"
         result="local.results">
  update tasks
  <cfif isDefined('arguments.dateScheduled')>
      set dateScheduled = <cfqueryparam
                        value="#createODBCDateTime(arguments.dateScheduled)#"
                        cfsqltype="cf_sql_timestamp" />
  <cfelse>
    set dateScheduled = <cfqueryparam null="true"
                                      cfsqltype="cf_sql_timestamp" />
  </cfif>
  where taskID in (#local.taskIDsForSQL#)
</cfquery>
```

This is a standard update query. If the **dateScheduled** argument is defined, then the **dateScheduled** column is set to its value. Otherwise, it is set to “null”. The SQL statement’s “where” clause uses the “in” keyword with the **taskIDsForSQL** list to restrict the number of tasks that are updated.

This method doesn’t need to return any special data:

```
<cfset local.resultObject =
  createObject('component','com.dotComIt.learnWith.vos.ResultObjectVO')/>
```

```
<cfset local.resultObject['error'] = false />
<cfset local.resultString = convertToJSON(local.resultObject) />
<cfreturn local.resultString />
```

Additional error checking could be added to make sure that the query does not cause “error”, but I kept things simple here.

Testing the Multiple Task Scheduler

The process for testing the **scheduleTaskList()** is just to create an instance of the service, call the method, and output the results. The main application does not use this method to cancel a task by nulling out **scheduleDate** columns, so I didn’t test that specific scenario. The test is in the file **ScheduleTaskList.cfm** in the **com/dotComIt/learnWith/tests** directory. Here is the test code:

```
<h1>Schedule Task List</h1>
<cfset service =
 createObject('component','com.dotComIt.learnWith.services.TaskService')>
<cfset results = service.scheduleTaskList("1,2,3",createDate(2013,3,29))>
Results: <Br/>
<cfdump var="#results#" expand="false" />
```

This is the output of the method:

```
{
 "error":false
}
```

This means the code is working as expected.

Review the API

To integrate the services into the AngularJS application, you just need to know the endpoint and the method arguments. Since both of these services were added to the **TaskService** class, the end point is the same as seen in some previous chapters. If you have set up your own environment, you can use a local endpoint:

```
/com/dotComIt/learnWith/services/TaskService.cfc
```

The **scheduleTask()** method needs three arguments:

- **method**: This argument tells the application server which method to execute at the endpoint. In this case, the value should be “scheduledTask”.
- **taskID**: This argument is used in the method’s query to determine which task should be updated in the database.
- **dateScheduled**: This is an optional argument. If specified, it determines what date the task should be scheduled for. If not specified, the **dateScheduled** task property is set to “null”, cancelling the task.

The **scheduleTaskList()** requires these three arguments:

- **method**: This argument tells the application server which method to execute at the endpoint. In this case, the value should be “scheduleTaskList”.
- **taskIDList**: This argument is a comma separated list of **taskIDs**. All the tasks are updated in the database.
- **dateScheduled**: This is an optional argument. If specified, it determines what date the task should be scheduled for. If not specified, the **dateScheduled** task property is set to “null”, cancelling the task.

With the exception of **method**, all arguments mimic the defined arguments of the CFC Methods.

Access the Services

This section will show you the AngularJS code that integrates the services to schedule a task.

Accessing the scheduledTask( ) Service

Open up the **TaskService.js** file from the **com/dotComIt/learnWith/services/coldFusion** directory. You'll want to add a method named **scheduleTask()** to the **taskService** variable:

```
function scheduleTask(task){
    var parameters = {
        method : "scheduleTask",
        taskID : task.taskID
    };
    if(task.dateScheduled){
        parameters.dateScheduled = task.dateScheduled
    }
    return $http.post('/com/dotComIt/learnWith/services/TaskService.cfc',
               parameters )
};
```

The method accepts a single argument; the task to be updated. First, the method creates a parameter object with two values specified; the method name "scheduleTask", and the **taskID**. Then it checks the **dateScheduled** property on the task. If it exists, the **dateScheduled** property is added to the outgoing parameter list. In the case of removing a task from a schedule, the parameter will already be removed from the task and will not be added to the parameter object. Finally this method makes the call using the **$http.post()**. The function returns the results of the **$http()**, which will be a promise object. This allows the invoking code to execute "result" and "failure" functions against the service call when it is complete.

We need to remember to add the **scheduledTask()** method to the **TaskService** object:

```
var services = {
    loadTasks : loadTasks,
    loadTaskCategories : loadTaskCategories,
    updateTask : updateTask,
    scheduleTask : scheduleTask
}
```

This will make the **scheduleTask()** method available to the controller that uses the **TaskService** service.

Accessing the scheduleTaskList( ) Service

The **scheduledTaskList()** method needs to be put in the same **TaskService** file used in the previous section. This is the method:

```
function scheduleTaskList(taskArray, schedulerDate){
    var taskIDList = '';
    for (index = 0; index < taskArray.length; ++index) {
        taskIDList += taskArray[index].taskID + ","
    }
    taskIDList = taskIDList.substr(0,taskIDList.length-1);
    var parameters = {
        method : "scheduleTaskList",
        taskIDList : taskIDList,
        dateScheduled : $filter('date')(schedulerDate, 'shortDate')
    }
    return $http.post('/com/dotComIt/learnWith/services/TaskService.cfc',
               parameters )
};
```

The method accepts two arguments; an array of tasks to be updated, and the new date these tasks should be scheduled for.

The first thing the method does is to make a comma separated list of all the **taskIDs** by looping over the **taskArray** argument. Then, it removes the last comma after the list is completed. Next, a parameter object is created with the three arguments needed to execute the service method. The **dateScheduled** property is formatted using an AngularJS **$filter** object. Finally, the **post()** call is made. The **post()** call's promise object is returned so the invoking code can use it to run other code when the service call completes.

We need to remember to add the **scheduleTaskList()** method to the **TaskService** object here:

```
var services = {
    loadTasks : loadTasks,
    loadTaskCategories : loadTaskCategories,
    updateTask : updateTask,
    scheduleTask : scheduleTask,
    scheduledTaskList : scheduleTaskList
}
```

This will make the **scheduleTask()** method available to the controller that uses the **TaskService** service.

Remember, you can find the code behind this in our archive. The **index_ColdFusion.html** code contains all the special references required to run the UI app using ColdFusion services.

Wire Up the UI

This section will show you how to connect the AngularJS UI to the services. We'll be populating the contents of a lot of methods we referenced earlier in this chapter, but haven't implemented yet. We'll make some changes to the method used to load filtered tasks, and call that method when the scheduled date changes. We'll add service calls to the **scheduleTaskList()** when the "save" button is clicked, and to **scheduleTask()** when the "delete" button is clicked.

Modifying the loadTasks() Method to Accommodate Different Result Handlers

A **loadTasks()** method was implemented in a previous chapter to load the filtered tasks when the "filter" button was clicked from the task filter section of the main UI. The method will be used, once again, to create the initial load of tasks for the scheduler component. Different result handlers are needed when dealing with filtering of tasks versus loading those scheduled on a certain date. This is because the results will be stored in different variables in the **TaskModel**. As such, the **loadTasks()** method needs to be tweaked to add additional arguments for the success and error handler methods. Open up the **MainScreenCtrl** file from **com/dotComIt/learnWith/controllers**:

```
function loadTasks(taskFilter, onSuccess , onError ){
    TaskService.loadTasks(taskFilter).then(onSuccess , onError);
}
```

The method accepts three parameters now. The **taskFilter** was already one of the arguments. The new arguments represent the **onSuccess** and **onError** methods. Previously we were using hard-coded values for the success and error handler methods.

The **onFilterRequest()** method of the **MainScreenCtrl** must also be modified. Previously, it didn't have to include the two new arguments. So, if we want the code to continue to work, we'll have to add them. I won't repeat the full method, but here is the modified call to the **loadTasks()** method:

```
loadTasks(localTaskFilter, onTaskLoadSuccess, onTaskLoadError);
```

The **onInit()** method will need a similar update:

```
loadTasks($scope.taskModelWrapper.taskModel.taskFilter,
          onTaskLoadSuccess, onTaskLoadError);
```

You may remember we created an **onToggleScheduler()** method from earlier in this chapter. This method would trigger the UI changes that shrink down the size of the task grid and bring the scheduler component into view. When the scheduler component is shown, it calls a method named **loadSchedulerTasks**. This is the method:

```
function loadSchedulerTasks(){
  var localTaskFilter = {};
  localTaskFilter.scheduledEqualDate =
                 $filter('date')($scope.scheduler.schedulerDate, 'shortDate');
  loadTasks(localTaskFilter,onSchedulerTaskLoadSuccess,onTaskLoadError);
}
```

The code creates a new, empty object named **localTaskFilter**. Then it specifies the **scheduledEqualDate** on the object. It uses the **scheduler.schedulerDate** property as the value. It also runs it through a date format function with filter—like we've seen in the chapter covering the task filter. Afterwards, it calls the **loadTasks()** method; passing in the error handler and the success handler. The error handler is identical to what you've seen in the past. It will merely displays a message to the user.

The purpose of the **onSchedulerTaskLoadSuccess()** method is to set the resulting array of tasks to the **scheduledTasks** array in the **taskModel** instance. If there are any **addedTasks** the user has entered into the scheduler that have not yet been saved, they are added to the result:

```
function onSchedulerTaskLoadSuccess(response){
  if(response.data.error == 1){
      alert("We could not load the task data");
      return;
  }
  TaskModel.scheduledTasks = response.data.resultObject;
  TaskModel.scheduledTasks = TaskModel.scheduledTasks.concat(
                                                    TaskModel.addedTasks);
}
```

The **resultObject** is saved directly into the **scheduledTasks** array. The JavaScript array method—**concatenate()**—is then used to combine the **addedTasks** with the **scheduledTasks,** and save them back into the **scheduledTasks** variable. This will create the array used to populate the list of tasks displayed in the scheduler.

The **loadScheduledTasks()** also needs to be called when the scheduler date changes. This is triggered with an **ngChange** handler on the scheduler's date picker component. It calls the **onSchedulerDateChange()** method:

```
function onSchedulerDateChange(){
    $scope.loadSchedulerTasks();
}
```

The **onSchedulerDateChange()** method in turn calls the **loadSchedulerTasks()** method, which was shown above.

Implement the Delete Task from Scheduler Button

When the "X" button is clicked in the task schedule template, the **onTaskUnschedule()** method is called—as was shown earlier in this chapter. If this button is clicked and the task already has an associated **dateScheduled**, then a service call must be made to null the **dateScheduled** for the associated task. This is done in the **scheduleTask()** method:

```
function scheduleTask(task){
    TaskService.scheduleTask(task)
               .then(onTaskScheduleSuccess,onTaskScheduleError);
}
```

The task object's **dateScheduled** property will already be updated to the most current, so no additional processing of it is needed before calling the service method.

If there is a problem with the call, the **onTaskScheduleError()** method is called:

```
function onTaskScheduleError(response){
    alert(response.data);
}
```

The **onTaskScheduleError()** method merely displays an alert to the user, similar to how other error handlers have operated.

The success method is named **onTaskScheduleSuccess()**:

```
function onTaskScheduleSuccess(response){
 if(response.data.error == 1){
    alert("We could not schedule the task data");
    return;
 }
 replaceTask(TaskModel.tasks, response.data.resultObject[0]);
 for (index = 0; index < TaskModel.scheduledTasks.length; ++index){
   if(TaskModel.scheduledTasks[index].taskID ==
                                        response.data.resultObject[0].taskID){
         deleteTaskFromSchedule(TaskModel.scheduledTasks[index]);
   }
 }
}
```

If there is an error value returned from the service, then a message is displayed to the user. If not, the method continues with the processing. First, it calls a secondary method named **replaceTask()**. This method is a helper function that will replace a task object in an array with another, and will assume the same **taskID**. We are calling it on the main tasks array, which populates the task grid:

```
function replaceTask(taskArray,task){
    for (index = 0; index < taskArray.length; ++index) {
        if(taskArray[index].taskID == task.taskID){
            taskArray[index] = task;
            break;
        }
    }
}
```

The **replaceTask()** method loops over the given array. If the item is found, it replaces it and stops the loop.

Back to the **onTaskScheduleSuccess()** method; it then looks over the **scheduledTasks** displayed in the current scheduler list. If it finds the task, it then calls the **deleteTaskFromSchedule()** method. This is a method shown earlier in this chapter that will remove the task from the **scheduledTasks** list, as well as the **addedTasks** list. This is all that is needed to remove a task from the schedule.

Saving all Scheduled Tasks

The last element of this chapter is to look at the method for scheduling all the current tasks in the scheduler's list to the currently selected date. Clicking the "save" button in the scheduler template will call a method named **onTaskListSchedule()**:

```
$scope.onTaskListSchedule = function onTaskListSchedule(){
  TaskService.scheduleTaskList(TaskModel.scheduledTasks,
                                $scope.scheduler.schedulerDate)
             .then(onTaskListScheduleSuccess,onTaskListScheduleError);
}
```

The **onTaskListSchedule()** method calls the **scheduleTaskList()** method in the **TaskService**. It passes in the list of **scheduledTasks** and the date they are scheduled for. The service returns a promise object; which is used to queue the success and error handler functions.

The **onTaskListScheduleError()** method simply displays an alert to the user letting them know an error occurred:

```
function onTaskListScheduleError(response){
    alert(response.data);
}
```

We've seen methods like this all throughout the app. The **onTaskListScheduleSuccess()** method is a bit more complex. It will check for an error. If there is one, it will display a message to the user. However, if there is no error, it compares the tasks array for the task grid, and the **scheduledTasks** array from the scheduler template to find the same item. If an item exists in both arrays, then the item in the tasks array must be updated with the new **schedulerDate**. This will, in turn, update the task grid's visual display with the proper scheduled date:

```
function onTaskListScheduleSuccess(response){
  if(response.data.error == 1){
      alert("We could not schedule the tasks");
      return;
  }
  for (var masterTaskIndex = 0;
       masterTaskIndex < TaskModel.tasks.length;
       ++masterTaskIndex)
  {
    for (scheduledTaskIndex = 0;
         scheduledTaskIndex < TaskModel.scheduledTasks.length;
         scheduledTaskIndex++)
    {
     if(TaskModel.tasks[masterTaskIndex].taskID ==
        TaskModel.scheduledTasks[scheduledTaskIndex].taskID)
        TaskModel.tasks[masterTaskIndex].dateScheduled =
                        $filter('date')($scope.scheduler.schedulerDate, 'M/d/yyyy');
     }
    }
  }
  TaskModel.addedTasks = [];
}
```

The first piece of this method checks to see if an error was returned. If so, a message is displayed to the user and the method completes. If not, then the task grid's tasks array is looped over. Inside it, the **scheduledTasks** array is looped over. The two tasks are then compared; looking for items where the **taskID** matches from each array. If a **taskID** match is found, the task grid's task has its **dateScheduled** property updated to the current selected date within the scheduler component. This will, subsequently, update the task grid to display proper values.

Final Thoughts

This is the most complicated chapter of the book, as it deals with a lot of new concepts; **ngClass**, **ngShow**, and **ngRepeat**, while also introducing multiple service calls, and changes to existing service calls. Surprisingly, when building for this chapter, I had more problems implementing the layout than the business logic. Getting the “expand/collapse” scheduler button to size properly compared to the rest of page’s components was a challenge. I hope you learned something. There are two chapters left in this book. The next one will show you how to mark a task as completed, and the final one will discuss how to handle different levels of authentication.

Chapter 7: Marking a Task Completed

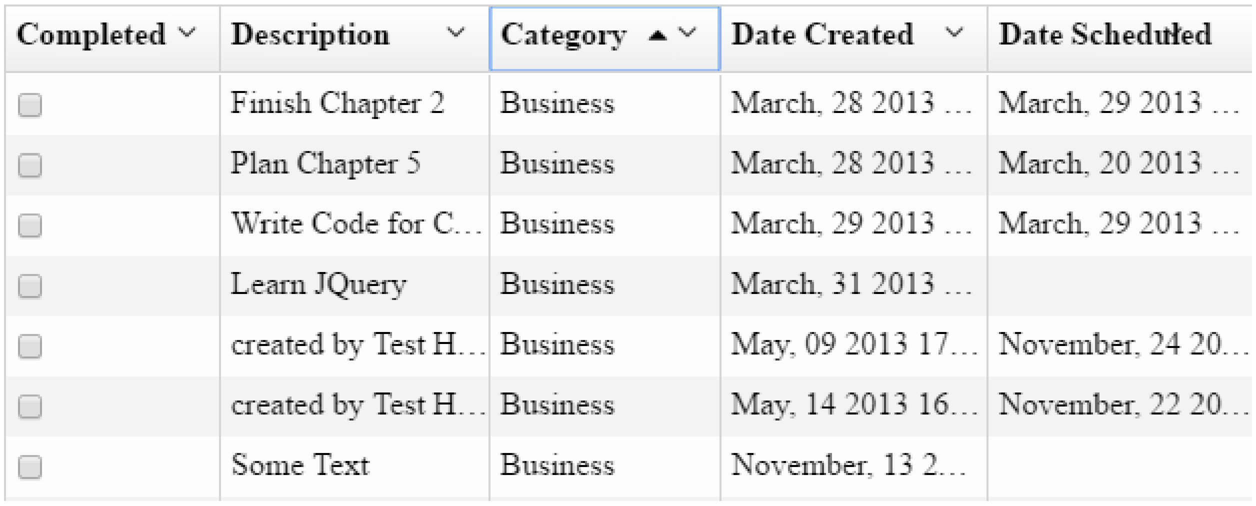
This chapter will cover the code behind marking a task complete. There are not any new UI elements introduced in this chapter. Instead, we are taking the checkbox from the task grid component and hooking it up to a new service which will mark the task as “completed” or “uncompleted”.

Create the User Interface

This section will review the UI features that we already built in the application in relation to marking a task as “completed”.

The Completed Checkbox

When we built the task grid, back in Chapter 3, we gave it a “completed” column. The “completed” column contains a checkbox. Its checked state can be used to determine if the task has already been completed or not. This is a recap of the grid from Chapter 3:



Completed	Description	Category	Date Created	Date Scheduled
☐	Finish Chapter 2	Business	March, 28 2013 …	March, 29 2013 …
☐	Plan Chapter 5	Business	March, 28 2013 …	March, 20 2013 …
☐	Write Code for C…	Business	March, 29 2013 …	March, 29 2013 …
☐	Learn JQuery	Business	March, 31 2013 …	
☐	created by Test H…	Business	May, 09 2013 17…	November, 24 20…
☐	created by Test H…	Business	May, 14 2013 16…	November, 22 20…
☐	Some Text	Business	November, 13 2…	

The checkbox can perform double duty. Instead of just using it as a visual way to show whether or not the task is completed, the user can also interact with it by clicking on the checkbox. When the checkbox is clicked, we can call a service to change the completed state of the checkbox.

The Checkbox Implementation

It has been a few chapters since you’ve seen it, so I wanted to review the Checkbox implementation. First open up the **MainScreenCtrl.js** file in the **com/dotComIt/learnWith/controllers** directory. Find the **taskGridOptions** object. This is the object which defines the properties that make up the **datagrid**. The property we care about is the **columnDefs**:

```
columnDefs: [
  {
    field: 'completed',
    displayName: 'Completed',
    cellTemplate:'com/dotComIt/learnWith/views/tasks/CompletedCheckBoxRenderer.html'
  },
  {field: 'description', displayName: 'Description'},
  {field: 'taskCategory', displayName: 'Category'},
  {field: 'dateCreated', displayName: 'Date Created'},
  {field: 'dateScheduled', displayName: 'Date Scheduled'},
  {
      name : 'Controls',
      displayName : '',
      enableSorting : false,
      cellTemplate:'com/dotComIt/learnWith/views/tasks/EditButtonRenderer.html'
  },
],
```

The **columnDefs** property on the **taskgridOptions** object defines the columns of the **uiGrid**. The first column is the “completed” column. It uses a **cellTemplate,** which is equivalent to an **itemRenderer**.

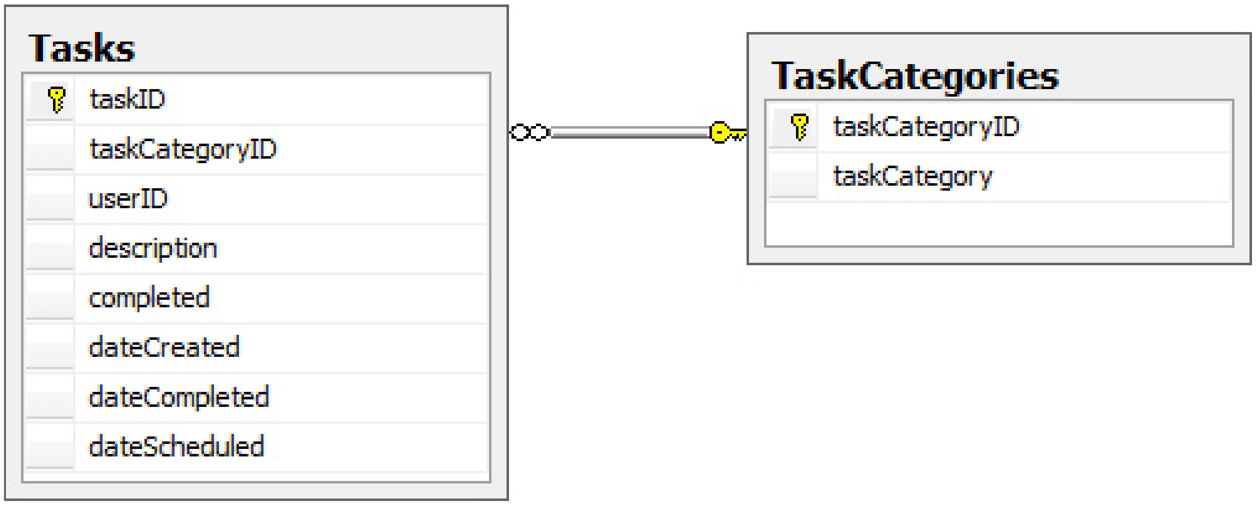
The cell template is **CompletedCheckBoxRenderer.html** and is located in the **com/dotComIt/learnWith/views/tasks** directory:

```
<div class="ngSelectionCell">
    <input tabindex="-1" class="ngSelectionCheckbox" type="checkbox"
           ng-checked="row.entity.completed"
           ng-click="grid.appScope.onCompletedCheckBoxChange(row.entity)"/>
</div>
```

The **ngClick** is new. When the checkbox is clicked, the **onCompletedCheckBoxChange()** method is executed inside the **MainScreenCtrl.js** file. We’ll look at the implementation right after creating the service.

Examine the Database

This is a review of the data structure surrounding tasks:



In this chapter, the only database columns that need updating are the **completed** column and the **dateCompleted** column. When a task is marked as "completed", the **completed** column should be set to "1" or "true", and the **dateCompleted** column should be set to the current date. When an item is being marked as "not completed", the **completed** column should be set to "0" or "false", and the **dateCompleted** column should be set to "null", erasing the value.

This is a sample query to set a task to being "completed":

```
update tasks
set completed = 1,
    dateCompleted = '5/20/2013'
Where taskID = 1
```

When we implement this in code, the values that are variable are the **taskID** and the **completed** column's value. The **dateCompleted** value will always be set to the current value.

Write the Services

This section will show you how to create the service method used to mark a task as "completed", and the JavaScript code needed to integrate with the service.

The completeTask( ) Service Method

You can add a method to the **TaskService.cfc** to allow tasks to be marked as "completed". The **TaskService.cfc** is located in the **com/dotComIt/learnWith/services** directory. This is the method signature:

```
<cffunction name="completeTask" returnformat="plain" access="remote"
output="true" >
      <cfargument name="taskID" type="numeric" required="true"/>
      <cfargument name="completed" type="Boolean" required="true" />
</cffunction>
```

The service method is named **completeTask()**. It accepts two arguments; the **taskID** to be updated, and the "completed" value. These two arguments allow the method to set a task to the "completed" state, or the "not-completed" state. A task may need to go into the "not-completed" state if the UI checkbox was accidently clicked incorrectly, marking the task as "complete". In that case, the user would need to click the checkbox again to mark the task as "open".

This method must perform two tasks; first, it will execute a query against the database, and then it will return a **ResultObjectVO** object containing the updated task.

First, the query:

```
<cfquery datasource="#application.dsn#" name="local.dataQuery"
         result="local.results">
  update tasks
  set completed = <cfqueryparam value="#arguments.completed#"
                                cfsqltype="cf_sql_bit" >,
  <cfif completed is true>
      dateCompleted = <cfqueryparam value="#createODBCDateTime(now())#"
                                    cfsqltype="cf_sql_timestamp" >
  <cfelse>
      dateCompleted = <cfqueryparam null="true"
                                    cfsqltype="cf_sql_timestamp" >
  </cfif>
  where taskID = <cfqueryparam value="#arguments.taskID#"
                               cfsqltype="cf_sql_integer" >
</cfquery>
```

The query performs a database update on the tasks table. It updates the "completed" field and the **dateCompleted** value. The next step is to return the updated task:

```
<cfset local.tempObject =
    createObject('component','com.dotComIt.learnWith.vos.TaskFilterVO')/>
<cfset local.tempObject.taskID = arguments.taskID />
<cfset local.JSONedObject = SerializeJSON(tempObject) />
```

```
<cfset local.results = getFilteredTasks(local.JSONedObject)>
<cfreturn local.results />
```

The previous code block is something you may remember from previous chapters. A new **TaskFilterVO** object is created, and the **taskID** property is set on it. The **getFilteredTasks()** method is used with the **TaskFilterVO** instance in order to retrieve the update task object. The value returned from the **getFilteredTasks()** method is returned to the UI call.

Testing the completeTask( ) Service Method

The process for testing the **completeTask()** method is to create an instance of the **TaskService**, call the method, and output the results. I created a file named **CompleteTask.cfm** in the **com/dotComIt/learnWith/tests** directory. This is a simple process which mimics the approach used in previous chapters. The first step is to create the instance of the service:

```
<cfset service =
 createObject('component','com.dotComIt.learnWith.services.TaskService')>
```

Then, the service method is called:

```
<cfset results = service.completeTask(1,1)>
```

The arguments into the service are the **TaskID**, “1”, and the completed value, “1”. This should mark the task with a **taskID** of “1” to complete. The final step is to output the results:

```
Results: <Br/>
<cfdump var="#results#" expand="false" />
```

The results are the task with a current **dateScheduled** property, and the completed field set to “1” or “true”:

```
{
 "resultObject":
  [
   {
    "taskcategoryID":2,
    "description":"Get Milk",
    "taskCategory":"Personal",
    "dateScheduled":"03\/29\/2013",
    "dateCompleted":"05\/26\/2013",
    "taskID":1,
    "dateCreated":"03\/27\/2013",
    "completed":1,
    "userID":1
  }
 ],
 "error":0.0
}
```

To set the task as “not-completed”, you can make the same service call, but replace the second parameter with a “0” instead of a “1”:

```
<cfset results = service.completeTask(1,0)>
Results: <Br/>
<cfdump var="#results#" expand="false" />
```

The results are output using an identical **cfdump** call:

```
{
 "resultObject":
 [
  {
   "taskcategoryID":2,
   "description":"Get Milk",
   "taskCategory":"Personal",
   "dateScheduled":"03\/29\/2013",
   "dateCompleted":"",
   "taskID":1,
   "dateCreated":"03\/27\/2013",
   "completed":0,
   "userID":1
  }
 ],
 "error":0.0
}
```

In the results you can successfully see that the “completed” field is back to “0”, and the **dateCompleted** value is an empty string.

Review the API

To integrate the services into the JavaScript application, you need to know the end point and the method arguments. Since both of these new services were added to the **TaskService** class, the end point is the same. If you have set up your own environment, you can use a local endpoint:

```
/com/dotComIt/learnWith/services/TaskService.cfc
```

The **completeTask()** method has three arguments when being called as a service:

- **method**: This parameter must be included as part of the service call in order for the UI to tell the service which method to execute at the specified endpoint.
- **taskID**: This argument refers to the task which must be modified by the method.
- **completed**: This argument tells the method whether the task should be marked as “completed” or not. If the value is “1”, then the task will be marked as “completed” and the **dateCompleted** will be set. If the value is “0”, then the task will be marked as “open”.

The first argument method is not explicitly defined as an argument to the **completeTask()** method. It could be considered a system variable needed for ColdFusion to determine which method to execute at the endpoint. The “taskID” and “completed” arguments, however, mirror the defined arguments into the **completeTask()** method.

Access the Service

This section will show you the JavaScript code needed to call the **completeTask()** service. First, open up the **TaskService** file in the **com/dotComIt/learnWith/services/coldFusion** directory. In the **taskService** object, create a new **completeTask()** method:

```
function completeTask(task){
    var parameters = {
        method : "completeTask",
        taskID : task.taskID,
        completed : !task.completed
    }
    $http.post('/com/dotComIt/learnWith/services/TaskService.cfc',
               parameters )
}
```

The method accepts a single argument; the **task** object which will be marked as “complete”, or “not complete”. It creates a **parameters** object with three properties. The **method** property is a hard-coded name; “completeTask”. The **taskID** value is taken from the **task** object passed into the function. The “completed” value is the opposite of the **taskID** object passed into the function.

Then the **post()** call is made on the **$http** service. The service call will modify the **task** in the database, and return a modified **task**. A promise object is returned from the function, which will allow the invoking code to execute stuff.

Remember to add the **completeTask()** function to the **TaskService**’s object:

```
var services = {
    loadTasks : loadTasks,
    loadTaskCategories : loadTaskCategories,
    updateTask : updateTask,
    scheduleTask : scheduleTask,
    scheduleTaskList : scheduleTaskList,
    completeTask : completeTask
}
```

Doing this will expose the **completeTask()** method to the controllers that use the **TaskService** object.

Wire Up the UI

This section will show you how to integrate the service into the UI code to mark the task completed. When the checkbox is clicked the method **onCompletedCheckBoxChange()** will execute. The method is inside the **MainScreenCtrl.js** file, located at **com/dotComIt/learnWith/controllers**.

This is the **onCompletedCheckBoxChange()** method:

```
$scope.onCompletedCheckBoxChange = function onCompletedCheckBoxChange (task){
  TaskService.completeTask(task,onTaskCompletedSuccess,onTaskCompletedError);
}
```

This method accepts a single argument, the task whose completed value needs to be toggled. It calls the **completeTask()** method of the **TaskService** object; passing in the task to be updated, the success method, and the failure method.

The error handler is similar to what we've been using throughout this book:

```
function onTaskCompletedError(response){
    alert(response.data);
}
```

The **onCompletedError()** method simply displays the returned data to the user.

The **onTaskCompletedSuccess()** method has a bit more to it, but not by much:

```
function onTaskCompletedSuccess(response){
  if(response.data.error == 1){
      alert("We could not load the task data");
      return;
  }
  replaceTask($scope.taskModel.taskModel.tasks,response.data.resultObject[0])
}
```

The success handler method checks to see if an error was returned from the server. If so, an alert is displayed to the user, and method processing stops. If no error was returned, a replacement task object must have been returned. The **replaceTask()** method is called. This is a method we built in the previous chapter, and will use to replace a task object inside the specified array. When scheduling tasks, we used it to modify the task in the main grid after the newly scheduled ones had been saved. This time, we're using it to update the task object after the it's completed property has been changed.

Final Thoughts

This chapter, while short, represents an important piece of the task manager application we've been building. It doesn't introduce any new concepts or ideas, and at this point in the book we are just applying what we know. There is only a single chapter left in this book and it will focus on the security aspects of the application, learning how to disable or enable functionality based on the user's role.

Chapter 8: Implementing User Roles

This chapter will focus on tweaking the application's UI based on the role of the user who had signed in. There are no new services to cover for this chapter, so the structure of this chapter will be a bit different than previous chapters. The bulk of this chapter relates to conditionally modifying the UI. A new Angular directive is introduced; **ngDisabled**.

Review User Roles

This section will review the user roles that pertain to this app, and then define how the UI needs to change based on those roles.

Role Review

You may remember, back in Chapter 1, we defined two user roles for this application:

- **Tasker**: This is the administrator user who has full access to the application. They can create new tasks, edit tasks, and mark tasks as "completed".
- **Creator**: This is the limited permission user. Users with this role can view tasks, and create new tasks. However, this user cannot edit tasks; including scheduling a task for a certain date, or marking tasks "completed".

The user's **RoleID** is loaded into the application along with other user data after the user logs in. They are stored as part of the user object in the **UserModel**. Two user accounts were set up here; one for each role. The **Tasker** account is "me/me" and the **Creator** account is "wife/wife".

What UI Changes Are Needed?

The **Tasker** role will see the app we have created without any further changes. However, the **Creator** role will need to see a different type of functionality. These are the items that need to be changed:

- **Disable Completed Checkbox**: The **Creator** user role should be able to see the "completed" checkbox—so they know the status of the tasks—but they should not be able to interact with it.
- **Scheduler**: The **Creator** user role should not be able to access the scheduler. This will be accomplished by hiding the button that will display or hide the scheduler.
- **Edit Task**: The **Creator** user role should not be able to edit tasks. This will be accomplished by hiding the column in the **uiGrid** that shows the "edit task" button.

These are simple changes throughout the app. Overall, they provide a more robust experience. In many of the applications I have built for Enterprise clients, controlling who can interact with what data is often very important.

Modify the UI

This section will show the code specifics on the UI changes that need to be made to support the different user roles.

Modifying the UserModel

The first step is to add an **isUserInRole()** function to the **UserModel**. This function will take a number argument representing a role and check to see if the user is in that role or not. If so, it will return true. If not, it will return false. This is a helper function to encapsulate the role-checking functionality throughout the app.

First, open the **UserModel.js** file from the **com/dotComIt/learnWith/model** directory. Add the **isUserInRole()** function to the **UserModel** object:

```
var userModel = {
    isUserInRole : isUserInRole,
    user : {
        userID : 0,
        username : '',
        password : '',
        roleID : 0
    }
}
return userModel;
```

The **isUserInRole** function is defined as part of the **userModel** object. The default **user** object is still there from previous chapters. Define the **isUserInRole** function after the return **userModel** statement:

```
function isUserInRole(roleToCompare){
    if(userModel.user.role == roleToCompare){
        return true;
    }
    return false;
}
```

This function will be used throughout our implementation of the new functionality. There are two separate roles in this app. For encapsulation sake, I defined them both in the **userModel**:

```
TASKER_ROLE = 1,
CREATOR_ROLE = 2
```

When reviewing code, it will be a lot easier to understand what **TASKER_ROLE** means than it will to understand what “1” means. In other languages, I might define these as constants. However, JavaScript does not have an implementation of constants, so they are variables instead. By using the convention of all caps, this should hopefully distinguish our constant variables from regular variables which use camel case.

Let’s turn our attention to the **MainScreenCtrl.js** file in the **com/dotComIt/learnWith/controllers** directory. It has an instance of the **UserModel** passed into it, and it is saved in a variable in the local **$scope**, like this:

```
$scope.userModel = UserModel;
```

Using this approach, the **userModel** will not be accessible inside our view template. This is because of how the include templates inherit data from the main controller. Objects are inherited, but simple values are not; and the **userModel** is a simple variable even though it points to an object. The solution is to move this into an object:

```
$scope.userModelWrapper = {
    userModel : UserModel
}
```

By making this change, we also have to change all the references to the **userModel** instance. In the **MainScreenCtrl** file, there is only one reference to the **validateUser()** function. Let's move this function to the **UserModel** class since we're making changes. This is the modified function:

```
function validateUser(){
    if(userModel.user.userID == 0){
        return false;
    }
    return true;
}
```

The **validateUser()** method was used by the **MainScreenCtrl** to validate whether the user has logged in or not. If they haven't logged in they are redirected back to the login page. Instead of accessing the **userModel** in the scope—as the original function did—this modified function references the instance variable of the user inside the **UserModel**.

The **MainScreenCtrl.js** will need to modify its usage of the **validateUser()** function to reference the function inside the **UserModel** instead of in the local function:

```
function onInit (){
    if(!UserModel.validateUser()){
        $location.path( "/login" );
    } else {
        loadTasks($scope.taskModelWrapper.taskModel.taskFilter,
                   onTaskLoadSuccess, onTaskLoadError);
        loadTaskCategories();
    }
}
```

This finishes our changes to the **UserModel** and we're ready to tackle the other changes.

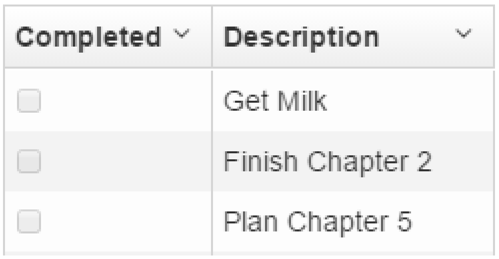
Disabling the Completed Checkbox

The "completed" checkbox column was implemented using a cell template. With the changes we made to the **MainScreenCtrl** in the previous section, the "completed" checkbox's cell template can now access the renderer. The name of the cell renderer is **CompletedCheckBoxRenderer.html** in the directory **com/dotComIt/learnWith/views/tasks**. This is the template:

```
<div class="ngSelectionCell">
  <input class="ngSelectionCheckbox" type="checkbox"
```

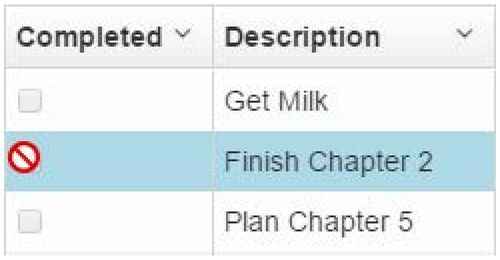
```
        ng-checked="row.entity.completed"
        ng-click="grid.appScope.onCompletedCheckBoxChange(row.entity)"
        ng-disabled="grid.appScope.userModelWrapper.userModel
        .isUserInRole(grid.appScope.userModelWrapper.userModel.CREATOR_ROLE)"
  />
</div>
```

The only new attribute on the checkbox is the **ngDisabled** attribute. If "true", the checkbox will be disabled. If "false", the checkbox will be enabled. To get the value for the **ngDisabled** property, the **isUserInRole()** function is called with the **CREATOR_ROLE** as an argument. When this occurs, AngularJS knows to automatically enable or disable the checkbox based on the user log in. You've seen enabled checkboxes all throughout this book, so far:



Completed	Description
☐	Get Milk
☐	Finish Chapter 2
☐	Plan Chapter 5

The disabled Checkboxes have a slightly different look; they are greyed out. When you roll the mouse over it, you see a visual mouse cursor telling you that you can't interact with it. This is the disabled checkboxes screen, with mouse cursor:



Completed	Description
☐	Get Milk
🚫	Finish Chapter 2
☐	Plan Chapter 5

Removing the Show Scheduler Button

The next step is to toggle the "scheduler" button display. The tasker role will be able to see it, but the creator will not. This is easy to do with an **ngShow** directive in the **MainSCreen.html** file, right on the **div** that contains the button:

```
<div class="horizontal-layout-4"
     ng-show="userModelWrapper.userModel.
                        isUserInRole(userModelWrapper.userModel.TASKER_ROLE)">
    <button class="height100" ng-click="onToggleScheduler()">
        {{schedulerShowButtonLabel}}
    </button>
</div>
```

We've already seen the **ngShow** directive, as it is used to hide or display the "scheduler" component. Here, the value of the component acts similar to the **ngDisabled** directive on the "completed" checkbox. The only difference is that we're testing for the **TASKER_ROLE** instead of the **CREATOR_ROLE**.

Removing the button leaves some empty space on the right side of the screen where the button used to be. So, to expand the task grid to make use of all the space available, we can create a new style. Put this in the **styles.css** file from the **com/dotComIt/learnWith/styles** directory:

```
.horizontal-layout-100{
    position: relative;
    display: inline-block;
    vertical-align: top;
    width:100%;
    min-height:100%;
}
```

The style on the task grid is set using the **ngClass** directive and the main screen controller's **gridContainerStyle** variable. You'll need to set the **gridContainerStyle** to something other than the default. You can do so in the **init()** method of the **MainScreenCtrl.js** file:

```
if(UserModel.isUserInRole(UserModel.CREATOR_ROLE)){
    $scope.gridContainerStyle = 'horizontal-layout-100';
}
```

It checks if the user is in the creator role. If they are, it sets the **gridContainerStyle** to the new value.

Removing the Edit Task Column

The final step is to remove the column with the edit button for the creator role while keeping it in for the tasker role. The solution to this is to set the **columnDefs** property on the **taskGridOptions** object dynamically based on the user role. Currently, the value is being set like this:

```
$scope.taskGridOptions = {
  columnDefs: [
    {
     field: 'completed',
     displayName: 'Completed',
     cellTemplate:'com/dotComIt/learnWith/views/tasks/CompletedCheckBoxRenderer.html'
    },
    {field: 'description', displayName: 'Description'},
    {field: 'taskCategory', displayName: 'Category'},
    {field: 'dateCreated', displayName: 'Date Created'},
    {field: 'dateScheduled', displayName: 'Date Scheduled'},
    {
      name : 'Controls',
      displayName : '',
      enableSorting : false,
     cellTemplate:'com/dotComIt/learnWith/views/tasks/EditButtonRenderer.html'
    },
  ],
  data: 'taskModelWrapper.taskModel.tasks',
  enableFullRowSelection :  true,
  enableRowHeaderSelection : false,
  multiSelect: false
}
```

This is a hard-coded object inside of the **MainScreenCtrl.js** file. We are going to remove the **columnDefs** from the **taskGridOptions** entirely:

```
$scope.taskGridOptions = {
  data: 'taskModelWrapper.taskModel.tasks',
  enableFullRowSelection :  true,
  enableRowHeaderSelection : false,
  multiSelect: false
}
```

And then create two separate versions of the column definitions:

```
var taskerGridColumnDefs = [
  {
    field: 'completed',
    displayName: 'Completed',
    cellTemplate:'com/dotComIt/learnWith/views/tasks/CompletedCheckBoxRenderer.html'
   },
   {field: 'description', displayName: 'Description'},
   {field: 'taskCategory', displayName: 'Category'},
   {field: 'dateCreated', displayName: 'Date Created'},
   {field: 'dateScheduled', displayName: 'Date Scheduled'},
   {
    name : 'Controls',
    displayName : '',
    enableSorting : false,
    cellTemplate:'com/dotComIt/learnWith/views/tasks/EditButtonRenderer.html'
   },
]
var creatorGridColumnDefs = [
   {
    field: 'completed',
    displayName: 'Completed',
    cellTemplate:'com/dotComIt/learnWith/views/tasks/CompletedCheckBoxRenderer.html'
   },
   {field: 'description', displayName: 'Description'},
   {field: 'taskCategory', displayName: 'Category'},
   {field: 'dateCreated', displayName: 'Date Created'},
   {field: 'dateScheduled', displayName: 'Date Scheduled'},
]
```

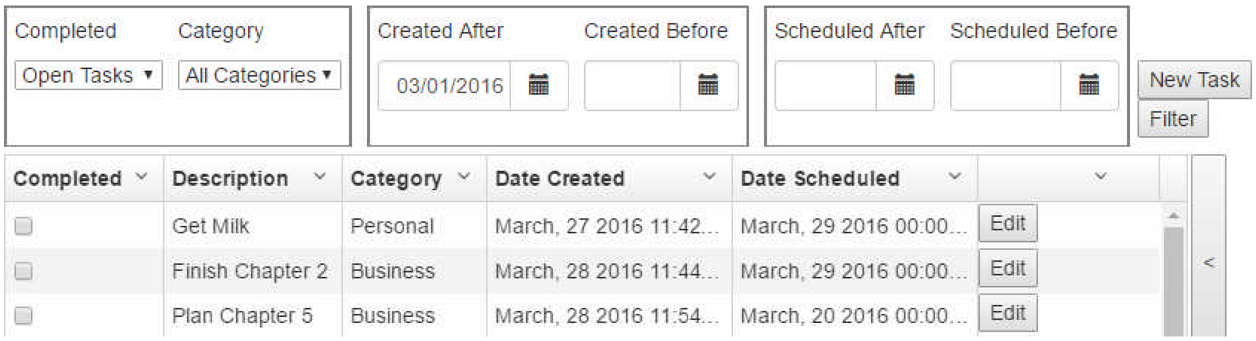
The first variable—**taskerGridColumnDefs**—is an array containing the column definitions for the Tasker's grid. It includes the **cellTemplate** for the "edit" button. The second variable—**creatorGridColumnDefs**—does not contain the final column, although the other columns are identical.

The next step is to set the **columnDefs** value on the **taskGridOptions**. This can be done during the **onInit()** method of **MainScreenCtrl**. We can use the same approach that was used in the previous section to set the **gridContainerStyle**. This is a mod to the same code snippet:

```
if(UserModel.isUserInRole(UserModel.CREATOR_ROLE)){
    $scope.gridContainerStyle = 'horizontal-layout-100';
    $scope.taskGridOptions.columnDefs = creatorGridColumnDefs;
} else {
    $scope.taskGridOptions.columnDefs = taskerGridColumnDefs;
}
```

The **isUserInRole()** function is used to make sure the user is in the creator role. If so, then the **gridContainerStyle** is set—as we saw in the previous section—and the **taskGridOptions.columnDefs** property is set to the **creatorGridColumnDefs**. Otherwise, the **taskGridOptions.columnDefs** property is set to the **taskerGridColumnDefs**.

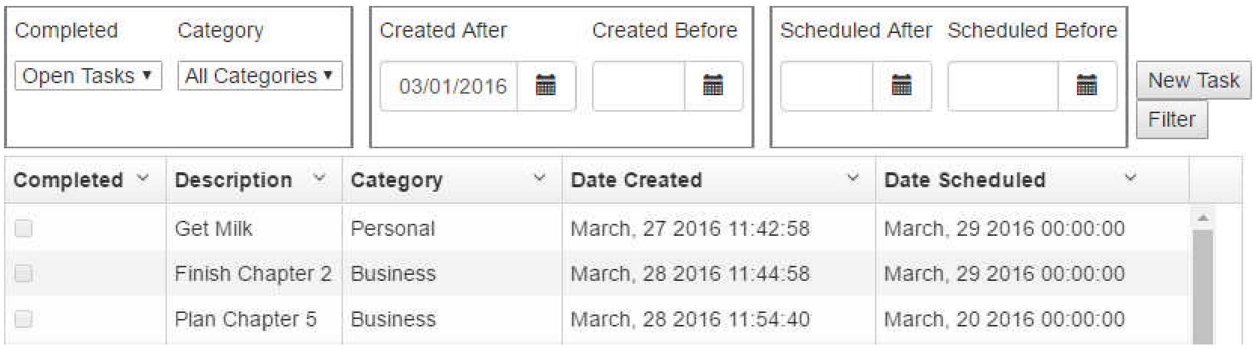
If the user logs in as a tasker, this is what they'll see:



Completed	Description	Category	Date Created	Date Scheduled	
☐	Get Milk	Personal	March, 27 2016 11:42...	March, 29 2016 00:00...	Edit
☐	Finish Chapter 2	Business	March, 28 2016 11:44...	March, 29 2016 00:00...	Edit
☐	Plan Chapter 5	Business	March, 28 2016 11:54...	March, 20 2016 00:00...	Edit

The scheduler "expand" button is on the right and the "edit" button is clearly visible in the default grid.

If the user logs in as a creator, this is what they'll see:



Completed	Description	Category	Date Created	Date Scheduled
☐	Get Milk	Personal	March, 27 2016 11:42:58	March, 29 2016 00:00:00
☐	Finish Chapter 2	Business	March, 28 2016 11:44:58	March, 29 2016 00:00:00
☐	Plan Chapter 5	Business	March, 28 2016 11:54:40	March, 20 2016 00:00:00

The scheduler's "expand" button is hidden here, and there is no edit button in the task grid.

Final Thoughts

At this point, we have finished building our task list manager in AngularJS. You should be ready to tackle your first AngularJS project. Good luck! Let me know how it goes.

Afterword

I wrote this book to document my own learning process building HTML5 applications. This book is actually the second iteration, and encompasses a lot of my experience building apps for real clients. I hope you benefited from my experience.

If you want more information be sure to check out **www.learn-with.com**. You can test the app we created in this book, get the source code for each chapter, get the most up to date version of the books, and browse some of our other titles which will build the same app using different technologies

If you need personal mentoring or have a custom consulting project, we'd love to help out, so please reach out

Send me an email at jeffry@dot-com-it.com and tell me how this book helped you become a success.