



Learn With
Knowledge to Boost Your Career

Angular 4, Bootstrap, & NodeJS



WWW.LEARN-WITH.COM

Part of the DotComIt Brain Trust

LEARN WITH ANGULAR, BOOTSTRAP, AND NODEJS

By Jeffry Houser

<https://www.learn-with.com>

<https://www.jeffryhouser.com>

<https://www.dot-com-it.com>

Copyright © 2017 by DotComIt, LLC

About the Author

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

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

During the Y2K era, Jeffrey wrote three books for Osborne McGraw-Hill. He is a member of the Apache Flex Project, and created Flextras; a library of Open Source Flex Components. Jeffrey 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, Jeffrey created Life After Flex; an AngularJS training course for Flex Developers. In 2016, Jeffrey launched the Learn With series with books focusing on using AngularJS with other technologies. Jeffrey has worked with multiple clients building AngularJS applications.

Table of Contents

[Learn With Angular, Bootstrap, and NodeJS](#)

[About the Author](#)

[Preface](#)

[Introduction](#)

[What is this Book Series About?](#)

[Who Is This Book for?](#)

[How to Read This Book](#)

[Common Conventions](#)

[Caveats](#)

[Want More?](#)

[Chapter 1: The Application Overview and Setup](#)

[Introducing the Task Manager Application](#)

[Setup Your Environment](#)

[Prerequisites](#)

[Get the Project Seed](#)

[Understand the Project Seed](#)

[Create the Database](#)

[Create the NodeJS Application](#)

[Create a Web Server in NodeJS](#)

[Create a Request Router](#)

[Create a Response Handler](#)

[Create the Main Application File](#)

[Create the Application Skeleton](#)

[Start with a Basic HTML Page](#)

[Set up the SystemJS Config](#)

[Setup the Angular Module](#)

[Set up the Routes](#)

[Create the Login Component](#)

[Create the Tasks Component](#)

[Create the Routing Module](#)

[Put it all Together](#)

[Final Thoughts](#)

[Chapter 2: Login](#)

[Create the User Interface](#)

[Creating Value Objects](#)

[The Generic Return Object](#)

[Create a User Value Object](#)

[Examine the Database](#)

[Write the Services](#)

[Install MSSQL NodeJS Driver](#)

[Creating a DatabaseConnection Package](#)

[Creating JSON in NodeJS](#)

[Create the AuthenticationService](#)

[Test in a Browser](#)

[Access the Services](#)

[Hashing the Password with Angular](#)

[Create the Service](#)

[Implement the authenticate\(.\) method](#)

[Turn the Service into a Provider](#)

[Wire Up the UI](#)

[Creating a UserModel](#)

[Accessing Component Values from Within the View](#)

[Implementing the Reset Button](#)

[Implementing the Login Handler](#)

[Final Thoughts](#)

[Chapter 3: Displaying the Tasks](#)

[Create the User Interface](#)

[What Goes in the Grid?](#)

[Setup the Grid](#)

[Tell Angular how to find the Grid Component](#)

[Creating a TaskModel and Other Value Objects](#)

[Create a Grid Component](#)

[Create the DataGrid](#)

[Creating a TaskFilter Object](#)

[Examine the Database](#)

[Write the Services](#)

[Install a DateFormatter](#)

[Create the Task Service](#)

[Testing the getFilteredTasks\(\) Service](#)

[Create the TaskService Stub](#)

[Turning the Object into a JSON String](#)

[Accessing the loadTask\(.\) Service](#)

[Wire Up the UI](#)

[Validate the User before Loading Data](#)

[Loading the Tasks](#)

[Final Thoughts](#)

[Chapter 4: Filtering the Tasks](#)

[Create the User Interface](#)

[What Data Do We Filter On?](#)

[Setup ng-bootstrap](#)

[Tell Angular how to find ng-bootstrap](#)

[Modify the TaskFilterVO](#)

[Create the TaskFilter component](#)

[Create the TaskFilter Template](#)

[Populating a Select with Angular](#)

[Adding a DateChooser](#)

[The Filter Button](#)

[Adding Styles](#)

[Examine the Database](#)

[Write the Service](#)

[Revisit the getFilteredTasks\(\) Method](#)

[Loading Task Categories](#)

[Testing Task Categories](#)

[Access the Service](#)

[Wire Up the UI](#)

[Loading Task Categories](#)

[Triggering the Filter](#)

[Catching the filterRequest Event](#)

[Test the Filtering](#)

[Final Thoughts](#)

[Chapter 5: Creating and Editing Tasks](#)

[Create the User Interface](#)

[The Task Window](#)

[Create the Popup Component](#)

[Populate the Popup Template](#)

[Opening the New Task Window](#)

[Opening the Edit Task Window](#)

[Examine the Database](#)

[Write the Services](#)

[Modify the getFilteredTasks\(.\) method](#)

[Creating a New Task](#)

[Testing Task Creation](#)

[Updating a Task](#)

[Testing Task Updates](#)

[Access the Services](#)

[Wire Up the UI](#)

[Clicking the Save Button](#)

[Handle the updateTask\(.\) Result](#)

[Final Thoughts](#)

[Chapter 6: Scheduling Tasks](#)

[Create the User Interface](#)

[The Task Scheduler Window](#)

[Create the TaskScheduler component](#)

[Create the Scheduler Template](#)

[Modifying the Main Screen](#)

[Clicking the Expand Button](#)

[Adding the Schedule Button to the TaskGrid](#)

[Examine the Database](#)

[Write the Services](#)

[Revisit the getFilteredTasks\(.\)](#)

[Scheduling a Single Task](#)

[Testing Scheduling a Single Task](#)

[Scheduling a Lot of Tasks](#)

[Testing Scheduling a Lot of Tasks](#)

[Access the Services](#)

[Use the scheduleTask\(.\) Service](#)

[Use the scheduleTaskList\(.\) Service](#)

[Wire Up the UI](#)

[Loading Tasks when Scheduler is Opened](#)

[Loading Tasks when the Scheduler Date Changes](#)

[Implement the Delete Task from Scheduler Button](#)

[Saving all Scheduled Tasks](#)

[Final Thoughts](#)

[Chapter 7: Marking a Task Completed](#)

[Create the User Interface](#)

[The Completed Checkbox](#)

[The Checkbox Implementation](#)

[Examine the Database](#)

[Creating the Service](#)

[The completeTask\(.\) Service Method](#)

[Testing the completeTask\(.\) service](#)

[Complete Tasks from Angular](#)

[Wire Up the UI](#)

[Final Thoughts](#)

[Chapter 8: Implementing User Roles](#)

[Review User Roles](#)

[Role Review](#)

[What UI Changes Are Needed?](#)

[Modify the UI](#)

[Modifying the UserModel](#)

[Disabling the Completed Checkbox](#)

[Removing the Show Scheduler Button](#)

[Removing the Edit Task Column](#)

[Final Thoughts](#)

[Afterword](#)

Preface

I was a Flex developer for a long time; however, Adobe's Flash Platform is no longer relevant. A smart developer will spend time educating himself on new technologies to keep up with a changing market, and to make sure he has healthy job prospects in the future. While cultivating these new skills, 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 Angular applications. Angular is a JavaScript framework built by Google for building smart user interfaces. It is built on TypeScript and allows you to build dynamic views in HTML5. It is fully testable, which is important to many enterprise-level applications. It has a large developer community, ready to help with problems. I greatly enjoy building applications with Angular.

This book will show you how to build a Task Manager application using Angular and Bootstrap. It uses REST services built with NodeJS, so Angular can interact with a fully functional backend.

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, with each book focusing on a specific technology or framework. This entry will focus on Angular 4 as the application framework, and Bootstrap as the primary 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 Angular or Bootstrap? Do you want to learn new technologies by following detailed examples with runnable code? If you answered 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:

- **TypeScript:** This is the language behind Angular. TypeScript is a statically typed language that compiles to JavaScript. The more you know about it, the better. If you are not familiar with it yet, check out [our tutorial lesson](#) on learning the basics of TypeScript.
- **NodeJS:** We use these scripts to compile our TypeScript into JavaScript, process CSS, and copy files. We'll also use NodeJS to build rest services which the main UI will integrate with. Familiarity with NodeJS will be beneficial, but is not required.
- **JavaScript:** TypeScript compiles to JavaScript to run in the browser. Aside from that, we touch on JavaScript routinely through the book in order to configure NodeJS scripts, and SystemJS; a module loader used by Angular. You should be familiar with JavaScript.
- **Angular:** The primary focus of this book is on Angular, so we are going assuming you have no experience with it. At the time of this writing, the most current version is Angular 4. If you're looking for information on the AngularJS 1.x code base, check out some of the other books in this series.
- **JSON:** The data returned from the services will be done so as JSON packets. JSON should be easy to understand, but if you have no experience with it, check out our [free introduction](#).
- **Bootstrap:** This is a CSS framework that helps create things such as popups and date choosers. We'll use it in conjunction with Angular to help flesh out the application's user interface.
- **SQL:** This is the database Server used as the storage mechanism for this book. As such, the SQL language will be used to communicate with the database from NodeJS. There aren't any advanced SQL

concepts in this book, but you should have a general understanding of this type of database Server.

How to Read This Book

Each chapter of this book represents one aspect of the application's user interface; logging in, editing a task, etc. Each chapter is split up into these parts:

- **Building the UI:** This section will show you how to create the UI elements of each chapter.
- **The Database:** There will be sections to review the data that each chapter's functionality deals with. The database storage tables will be examined here, 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 help you create services using NodeJS. 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 show you 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 is uppercase, and the start of each new compound word is uppercase. An example of a class name is **MyClass**. When referencing class names in the book text, the file extension is usually referenced. For TypeScript files that contain classes the extension will be “ts”. For JavaScript files, the extension is “js”.
- **Variables:** Variable names are also in proper case, except the first letter of the first compound word; it is always lowercase. This includes class properties, private variables, and method arguments. A sample property name is **myProperty**.
- **Constants:** Constants are in all uppercase, with each word separated by an underscore. A sample constant may be **MY_CONSTANT**.
- **Method or Function Names:** Method names use the same convention as property names. When methods are referenced in text, open and close parentheses are typed after the name. A sample method name may be **myMethodName()**.
- **Package or Folder Names:** The package names—or folders—are named using proper case again. 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 Angular. It leverages my experience building business apps, but is not intended to cover everything you need to know about building HTML5 Applications. This book purposely focuses on the Angular framework, not the tool chain. If you want to learn more about the tool chain, [check out our bonus book](#). You should approach this book as part of your learning process and not as the last thing you'll ever need to know. Be sure that you keep educating yourself. 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 code 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.

Login View

Username

Password

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

Completed	Description	Category	Date Created	Date Scheduled	
<input type="checkbox"/>	Copy edit Chapter 1	Business	12/5/2017	4/17/2017	<input type="button" value="Edit Task"/>
<input type="checkbox"/>	Finish Chapter 2	Business	3/28/2017	4/17/2017	<input type="button" value="Edit Task"/>
<input type="checkbox"/>	Write Code for Chapter 3	Business	3/29/2017	3/29/2017	<input type="button" value="Edit Task"/>
<input type="checkbox"/>	Plan Chapter 5	Business	3/28/2017	3/20/2017	<input type="button" value="Edit Task"/>

- 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.

Completed: Category:

Created After: Created Before:

Scheduled After: Scheduled Before:

- The ability to mark a task completed.

Completed	Description
<input checked="" type="checkbox"/>	Copy edit Chapter 1
<input type="checkbox"/>	Finish Chapter 2

- The ability to create or edit tasks.

Create a New Task ×

Description

Category

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

Completed: Open Tasks

Category:

Created After:

Created Before:

Scheduled After:

Scheduled Before:

New Task

Filter

Completed	Description	Category	Date Created	Date Schedule	
<input type="checkbox"/>	Clean Kitchen	Personal	5/9/2017	11/21/2017	+
<input type="checkbox"/>	Wish Mom a b	Personal	5/9/2017	11/22/2017	+
<input type="checkbox"/>	Followup with	Business	5/14/2017	11/22/2016	+
<input type="checkbox"/>	Call Brother	Personal	5/27/2017	11/22/2017	+
<input type="checkbox"/>	Call Sister	Personal	5/27/2017	11/22/2017	+
<input type="checkbox"/>	Start Chapter	Business	11/13/2017		+
<input type="checkbox"/>	Update StackO	Business	11/16/2017		+

Scheduler

Copy edit Chapter 1	X
Finish Chapter 2	X
Create React Proof of Concept	X
Call Sister	X
Start Chapter 6	X
Update StackOverflow Profile	X

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

Setup Your Environment

When creating a JavaScript application, you can write code that will immediately run in the browser. Although build tools are common in HTML5 applications, they are not required. However, when writing a TypeScript application, you need a process to compile the TypeScript into JavaScript so you can run your code in the browser. This section will tell you how to set your environment up for writing and compiling our TypeScript Angular application.

Prerequisites

Before you start you'll need to install some prerequisites:

- **NodeJS:** Our build process will make use of [NodeJS](#), so I strongly recommend you get this set and configured locally. It is necessary for modern web development. When you install NodeJS, the process will also install the NodeJS Package Manager; “npm”. This will allow you to easily install NodeJS modules that someone else created. We'll use npm in the next chapter to install a SQL Server driver, as it is the database behind the app.
- **Web Server:** I use the [Apache Web Server](#), but you should be fine using [Express](#), [IIS](#), or any web server of your choice. Just configure the web root, or a virtual directory, to point to your project files.
- **Git:** This is a source control. Installing [Git](#) is optional for this book. We'll just use it in this chapter to setup the project seed. If you don't want to install Git, you can download the project seed files directly from [our GitHub.com repository](#).

The installation instructions on the related web sites will explain how to install the software for your environment better than anything I could offer here.

Get the Project Seed

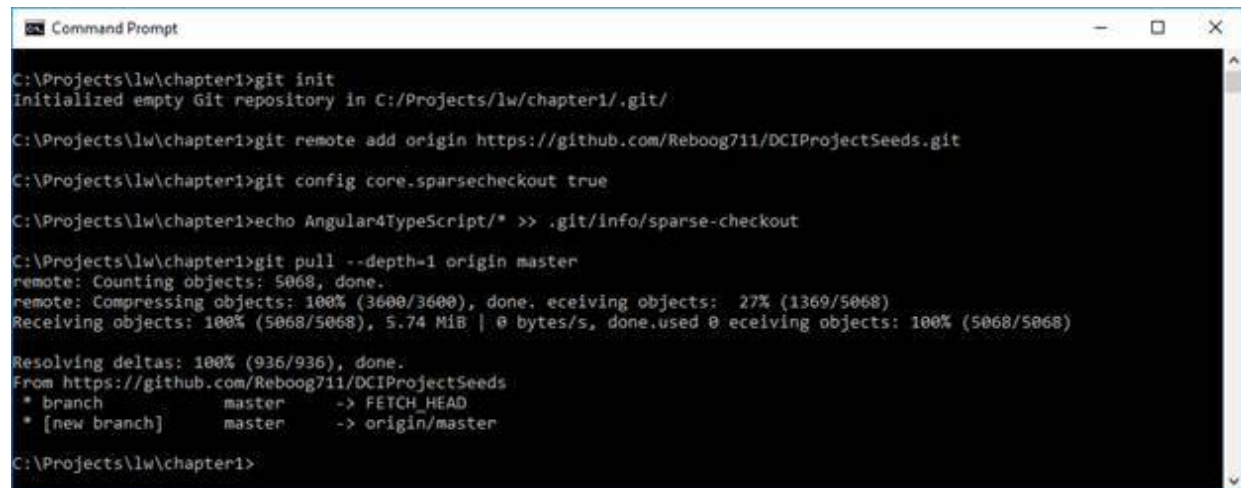
An explanation of creating the build scripts would be very long. To keep the discussion relevant to writing code, we're going to use a project seed to jump start the process. The Angular team provides a [seed project](#) you can use, and it is great to get started quickly. There is also an [Angular CLI](#) project that is popular. I'm going to use the [one I wrote](#). It combines some NodeJS

and Gulp scripts. A full explanation of how I built the seed project is available as part of the [Angular bonus book](#) to this series.

First, you need to check out the seed. If you have Git installed, you can run this script from the directory where you plan to write your code:

```
git init
git remote add origin
https://github.com/Reboog711/DCIProjectSeeds.git
git config core.sparsecheckout true
echo Angular4TypeScript/* >> .git/info/sparse-checkout
git pull --depth=1 origin master
```

Your command line should look like this:



```
Command Prompt
C:\Projects\lw\chapter1>git init
Initialized empty Git repository in C:/Projects/lw/chapter1/.git/
C:\Projects\lw\chapter1>git remote add origin https://github.com/Reboog711/DCIProjectSeeds.git
C:\Projects\lw\chapter1>git config core.sparsecheckout true
C:\Projects\lw\chapter1>echo Angular4TypeScript/* >> .git/info/sparse-checkout
C:\Projects\lw\chapter1>git pull --depth=1 origin master
remote: Counting objects: 5068, done.
remote: Compressing objects: 100% (3600/3600), done, receiving objects: 27% (1369/5068)
Receiving objects: 100% (5068/5068), 5.74 MiB | 0 bytes/s, done, used 0 receiving objects: 100% (5068/5068)
Resolving deltas: 100% (936/936), done.
From https://github.com/Reboog711/DCIProjectSeeds
 * branch      master      -> FETCH_HEAD
 * [new branch] master      -> origin/master
C:\Projects\lw\chapter1>
```

Now, you have to setup the directory. Run these two NodeJS commands:

```
npm install -g gulp
npm install
```

The first will install Gulp as a global install. This is required to run Gulp commands from the command line. The second will install all the required NodeJS modules for this project. You'll see results similar to this:

```
Command Prompt
C:\Projects\lw\chapter1\Angular4TypeScript>npm install -g gulp
npm WARN deprecated minimatch@2.0.10: Please update to minimatch 3.0.2 or higher to avoid a RegExp DoS issue
npm WARN deprecated minimatch@0.2.14: Please update to minimatch 3.0.2 or higher to avoid a RegExp DoS issue
npm WARN deprecated graceful-fs@1.2.3: graceful-fs v3.0.0 and before will fail on node releases >= v7.0. Please update to graceful-fs@4.0.0 as soon as possible. Use 'npm ls graceful-fs' to find it in the tree.
C:\Users\jhouse\AppData\Roaming\npm\gulp -> C:\Users\jhouse\AppData\Roaming\npm\node_modules\gulp\bin\gulp.js
C:\Users\jhouse\AppData\Roaming\npm
|-- gulp@3.9.1

C:\Projects\lw\chapter1\Angular4TypeScript>npm install
npm WARN deprecated minimatch@2.0.10: Please update to minimatch 3.0.2 or higher to avoid a RegExp DoS issue
npm WARN deprecated minimatch@0.2.14: Please update to minimatch 3.0.2 or higher to avoid a RegExp DoS issue
npm WARN deprecated graceful-fs@1.2.3: graceful-fs v3.0.0 and before will fail on node releases >= v7.0. Please update to graceful-fs@4.0.0 as soon as possible. Use 'npm ls graceful-fs' to find it in the tree.
DCITypeScriptProjectSeed@0.0.1 C:\Projects\lw\chapter1\Angular4TypeScript
+-- @angular/common@4.0.2
+-- @angular/compiler@4.0.2
+-- @angular/core@4.0.2
+-- @angular/forms@4.0.2
+-- @angular/http@4.0.2
+-- @angular/platform-browser@4.0.2
+-- @angular/platform-browser-dynamic@4.0.2
+-- @angular/router@4.0.2
+-- angular-in-memory-web-api@0.3.1
+-- core-js@2.4.1
+-- del@2.2.2
| +-- globby@5.0.0
| | +-- array-union@1.0.2
| | |-- arrify@1.0.1
| | +-- is-path-cwd@1.0.0
| | +-- is-path-in-cwd@1.0.0
| | |-- is-path-inside@1.0.0
| | |-- path-is-inside@1.0.2
| +-- object-assign@4.1.1
```

Understand the Project Seed

The seed project comes with some default code, so you can run your first build now:

```
gulp build
```

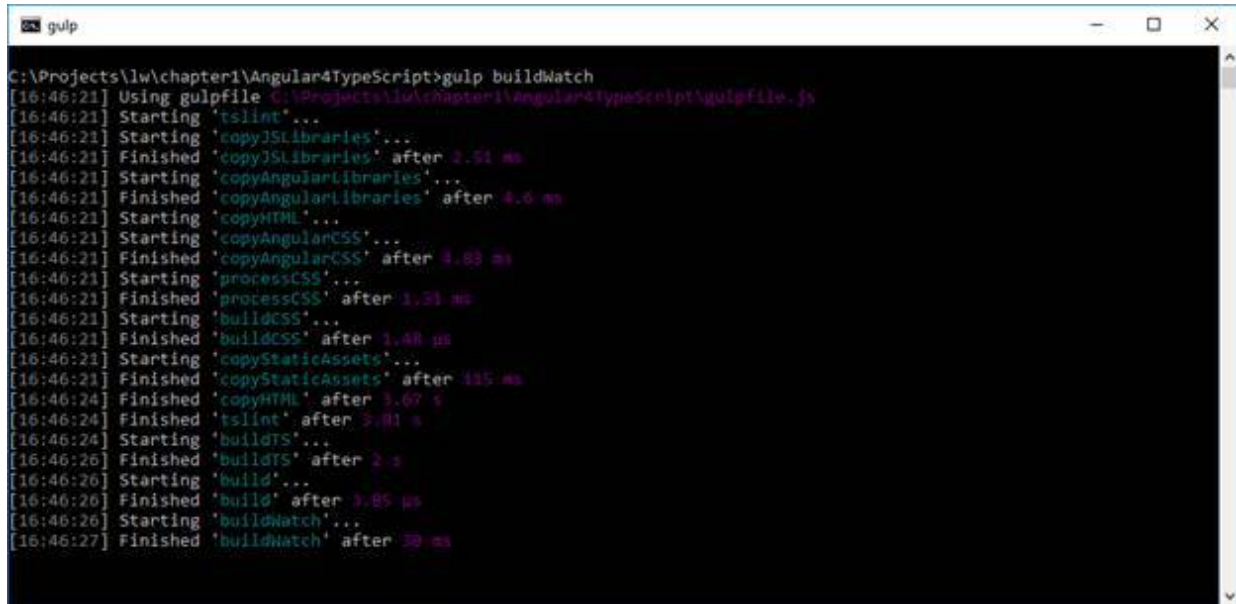
The command line should show you this:

```
Command Prompt
C:\Projects\lw\chapter1\Angular4TypeScript>gulp build
[16:42:33] Using gulpfile C:\Projects\lw\chapter1\Angular4TypeScript\gulpfile.js
[16:42:33] Starting 'tslint'...
[16:42:33] Starting 'copyJSLibraries'...
[16:42:33] Finished 'copyJSLibraries' after 2.32 ms
[16:42:33] Starting 'copyAngularLibraries'...
[16:42:33] Finished 'copyAngularLibraries' after 4.02 ms
[16:42:33] Starting 'copyHTML'...
[16:42:33] Starting 'copyAngularCSS'...
[16:42:33] Finished 'copyAngularCSS' after 4.7 ms
[16:42:33] Starting 'processCSS'...
[16:42:33] Finished 'processCSS' after 1.49 ms
[16:42:33] Starting 'buildCSS'...
[16:42:33] Finished 'buildCSS' after 1.48 ms
[16:42:33] Starting 'copyStaticAssets'...
[16:42:33] Finished 'copyStaticAssets' after 116 ms
[16:42:36] Finished 'copyHTML' after 3.55 s
[16:42:37] Finished 'tslint' after 2.87 s
[16:42:37] Starting 'buildTS'...
[16:42:39] Finished 'buildTS' after 1.99 s
[16:42:39] Starting 'build'...
[16:42:39] Finished 'build' after 3.26 ms
C:\Projects\lw\chapter1\Angular4TypeScript>
```

The gulp **build** command runs a lot of different commands including compiling TypeScript, copying Angular libraries and static files, and parsing CSS. Most of the time when developing you'll use the **buildWatch** script:

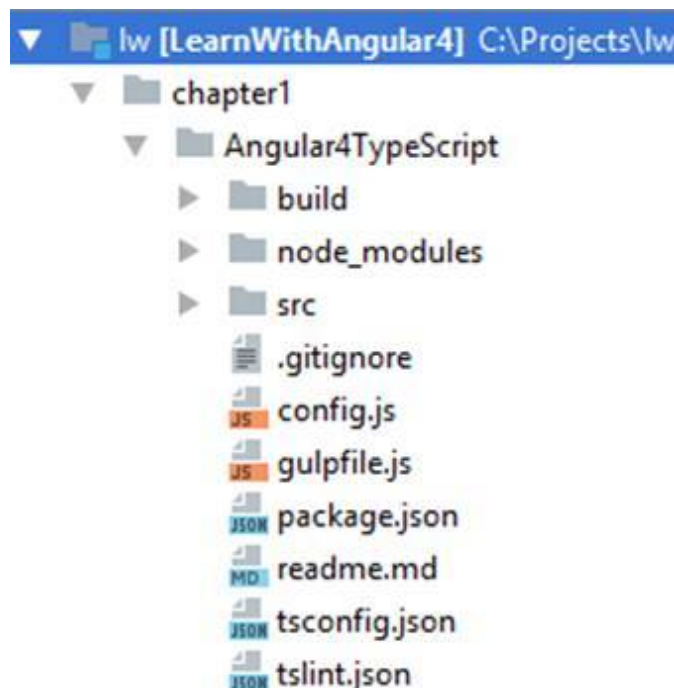
```
gulp buildWatch
```

This will run in the background and recompile the app whenever there are code changes:



```
C:\Projects\lw\chapter1\Angular4TypeScript>gulp buildWatch
[16:46:21] Using gulpfile C:\Projects\lw\chapter1\Angular4TypeScript\gulpfile.js
[16:46:21] Starting 'tslint'...
[16:46:21] Starting 'copyJSLibraries'...
[16:46:21] Finished 'copyJSLibraries' after 2.51 ms
[16:46:21] Starting 'copyAngularLibraries'...
[16:46:21] Finished 'copyAngularLibraries' after 4.6 ms
[16:46:21] Starting 'copyHTML'...
[16:46:21] Starting 'copyAngularCSS'...
[16:46:21] Finished 'copyAngularCSS' after 4.83 ms
[16:46:21] Starting 'processCSS'...
[16:46:21] Finished 'processCSS' after 1.24 ms
[16:46:21] Starting 'buildCSS'...
[16:46:21] Finished 'buildCSS' after 1.48 ms
[16:46:21] Starting 'copyStaticAssets'...
[16:46:21] Finished 'copyStaticAssets' after 135 ms
[16:46:24] Finished 'copyHTML' after 3.67 s
[16:46:24] Finished 'tslint' after 2.81 s
[16:46:24] Starting 'buildTS'...
[16:46:26] Finished 'buildTS' after 2 s
[16:46:26] Starting 'build'...
[16:46:26] Finished 'build' after 3.85 ms
[16:46:26] Starting 'buildWatch'...
[16:46:27] Finished 'buildWatch' after 39 ms
```

Check out the directory structure and you'll see something like this:



The important elements you need to understand:

- **Root Directory:** Contains a lot of the Node and Gulp configuration files. You probably won't have to worry about this for the most part, but we'll modify the config in future chapters as we add more libraries to the project, such as NG Bootstrap.
- **build:** Contains the build of the application created by the build scripts. When testing the code in a browser, you'll open up the **index.html** file in here, but otherwise will not need to modify these.
- **node_modules:** Contains all the NodeJS modules. You won't need to touch this.
- **src:** This directory contains all your source code files including HTML Files, TypeScript files, CSS files, image assets, and JavaScript libraries. You'll spend a lot of time editing code here throughout this book.

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.

Create the NodeJS Application

This section will introduce the architecture of our NodeJS application. It will show you how to build three NodeJS modules that will work together as the backbone of our app. The first module is a **server** module, which will accept incoming requests. The second is a **router** module that will be used by the web server to determine how to handle incoming requests. The third will be a **response** handler module, which will be used to determine how to handle the request, and what data to send back to the browser. Finally, we will build a main application file to tie everything together.

Create a Web Server in NodeJS

When creating the NodeJS application, I decided to use inspiration from the directory structure of the other applications built in this series. Our NodeJS web server will be in the **Server.js** file in the **com/dotComIt/learnWith/server** directory. The first line of the file loads the NodeJS **http** library:

```
var http = require('http');
```

The “require” statement is a cross between an “import” statement and a variable definition. This code loads the NodeJS **http** module and stores a reference to it in the **http** variable. The **http** module is one of the modules that included as part of NodeJS.

Our server will also need access to the NodeJS **url** library:

```
var url = require("url");
```

The **url** library is another built in NodeJS library, and it will be used to determine what URL the user requested, along with the query string of the request.

Next, we can create the HTTP server:

```
http.createServer(function (request, response) {  
  // other code here  
}).listen(8080);
```

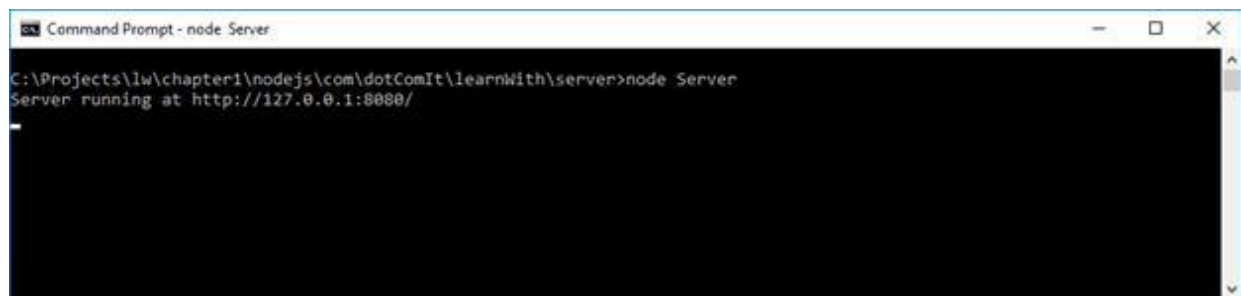
To create the server, use the **createServer()** method on the **http** object. The method has two values; the **request**, and **response**. The **request** object is used to determine what the client requested, and the **response** object is

used to send data back to the client. In a moment, we'll populate the contents of the method. First, jump to the last line of the code segment. The dot syntax is used to call the **listen** method on the **http.createServer()** object. It specifies the port that the app will listen at. Port 80 is the common HTTP port, however on my own machine I have that port tied up by other web servers. As such, I chose to use port 8080 for this server.

This code should execute as is. First, add a console log at the end of the file:

```
console.log('Server running at http://127.0.0.1:8080/');
```

This will show in the NodeJS console, so you can verify that your code has run. Now try to run the code, and you should see something like this:



```
Command Prompt - node Server
C:\Projects\lw\chapter1\nodejs\com\dotComIt\learnWith\server>node Server
Server running at http://127.0.0.1:8080/
_
```

If you were to try to load a page with the current server implementation, the request would eventually time out without loading anything. This is because we have not coded the server to respond to any explicit requests. We need to flesh out the anonymous function argument to the **createServer()** method.

First, **parse** the request:

```
var url_parts = url.parse(request.url, true);
```

The **parse** method is on the **url** object, which we imported into this file using the **require()** method. The first argument to the **parse()** function is the "url", which we get off the **request** object. The second method is a Boolean, which we set to "true". It will say, "Yes, parse the query string".

Now, get the **pathname**:

```
var pathname = url_parts.pathname;
console.log("Request for " + pathname + " received.");
```

The file path is stored to the local variable, named **pathname**. The path will include everything after the domain name and port. If you request this:

```
http://localhost:8080/MyPage.html
```

Then the **pathname** will be:

```
/MyPage.html
```

If you were to request this, instead:

```
http://localhost:8080/dotComIt/learnWith/someRandomPage/
```

Then the **pathname** would be:

```
/dotComIt/learnWith/someRandomPage/
```

The **pathname** will be used within the router to determine what the request was for, and how to handle it.

Next, get the query string:

```
var queryString = url_parts.query;
```

The query string can be taken off the same **url_parts** object created by the **parse()** function.

The last step in the method is to call the request router's **route()** method:

```
route(handlers, pathname, response, queryString);
```

We have not created the **route()** method yet, but will in the next section. The input in the router is a **handlers** variable—which we also haven't created yet. It also takes the **pathname**, the **response**, and the **queryString**.

Place the **createServer()** method inside another function named **start**:

```
function start(route, handlers){  
  // http.createServer() implementation here  
}
```

The **start** method has two arguments; a **route** function—which will be created in the next section—and a **handlers** structure—which will be created later in this chapter.

The last line of our server module is:

```
exports.start = start;
```

This file is a NodeJS module of our own creation. The **exports** command defines the API that external modules or NodeJS applications will use to

interface with this custom module. In this case, we are exposing the **start()** method. This will all come together after we create the index.

Create a Request Router

This section will create a request router file. The purpose of this file is to take in the requested **pathname**, the query string, the **response** object, and the **handlers** object. It will determine if a handler exists for the relevant path. If so, it calls the **handlers** function. Otherwise, it will need to return a **404** response.

I created a file **RequestRouter.js** in the **com/dotComIt/learnWith/server** directory. First, create the **function** signature:

```
function route(handlers, pathname, response, queryString) {  
}
```

This **function** definition holds the same signature as the **route()** function call from the previous section. That is because this file represents the implementation of that function.

The **handlers** variable is like an object we use in client side JavaScript. The key of the handler will be the request, and the value will be a function that handles the request. The first step is to check for the existence of the **handler** function:

```
if (typeof handlers[pathname] === 'function') {  
    handlers[pathname](response, queryString);  
}
```

The code uses the JavaScript **typeof** keyword. It determines if the key—AKA **pathname**—in the **handlers** object is a function. If it is, then the function is executed; passing in the **response** object, and the **queryString**.

If no handler exists for the **pathname**—or if the handler is not a function—then the code does not know how to handle the request. In this case, a **404** response must be returned. This can be done in the **else** condition:

```
else {  
    response.writeHead(404, {"Content-Type": "text/plain"});  
    response.write("404 Not found");  
    response.end();  
}
```

The response first writes the header, which specifies **404**. Then it writes some body text to the response, which just specifies, "404 Not Found". Finally, it calls the **end()** method, which closes the request and sends the final response back to the browser.

For the **RequestRouter** to be used as a component, it must export the function:

```
exports.route = route;
```

That completes the **RequestRouter**. Next, we will create a **ResponseHandler**, and then we'll put all the files together into a single app by through a main index file.

Create a Response Handler

This section will show you how to create a sample response handler for **index.html**, and then create a **ResponseHandler** package. First, create a file named **IndexService.js** in the **com/dotComIt/learnWith/services** directory:

```
function execute(response, queryString) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write('The Index');
    response.end();
};
exports.execute = execute;
```

This file represents the handler for processing a request for **/index.html**. It does not need a query string, so that parameter is ignored in the main function. It uses the **response** variable to write a text header, some content, and then close the request. This is similar to how the **404** error was handled.

This file is used in a response handler file. Create a file named **ResponseHandlers.js** in the **com/dotComIt/learnWith/server** directory. The purpose of this file is to create the **handlers** associative array, which will be passed into the server. First, create the associative array as an object:

```
var handlers = {};
```

The **handlers** associative array is initialized as a simple object. Next, load the index service using the **require()** method:

```
var indexService = require("../services/IndexService");
```

Add the `"/index.html"` handler to the **handlers** object, with the value of **indexService.execute**:

```
handlers["/index.html"] = indexService.execute;
```

Of course, we have to export the **handlers** variable:

```
exports.handlers = handlers;
```

In previous sections, we have only exported functions. However, it is perfectly valid to export variables, which is what is done in this case. Next, we must put this all together; creating a main file that can be used to launch the application.

Create the Main Application File

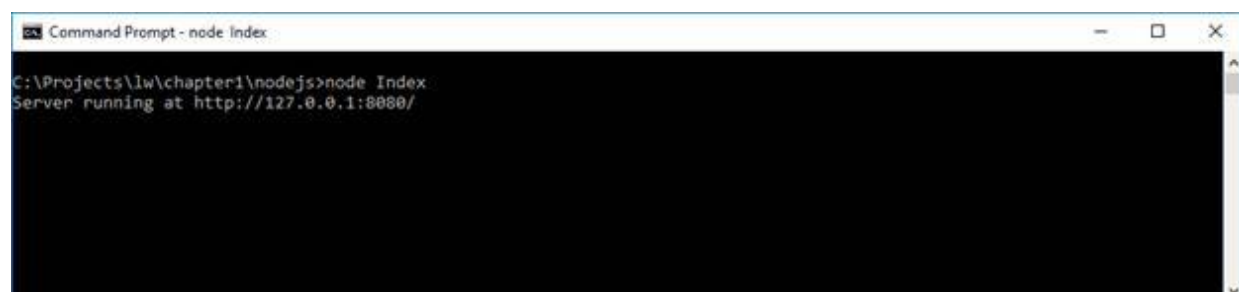
The main application file is **Index.js** and can be found in the root directory. The purpose of this file is to load all the other modules and start the server:

```
var server = require("../com/dotComIt/learnWith/server/Server");
var requestRouter =
require("../com/dotComIt/learnWith/server/RequestRouter");
var requestHandlers =
require("../com/dotComIt/learnWith/server/ResponseHandlers");
server.start(requestRouter.route, requestHandlers.handlers);
```

First, the server component is loaded. Then the **RequestRouter** is loaded. Next, the **ResponseHandlers** are loaded. All methods use the **require()** method, and store the results in a variable.

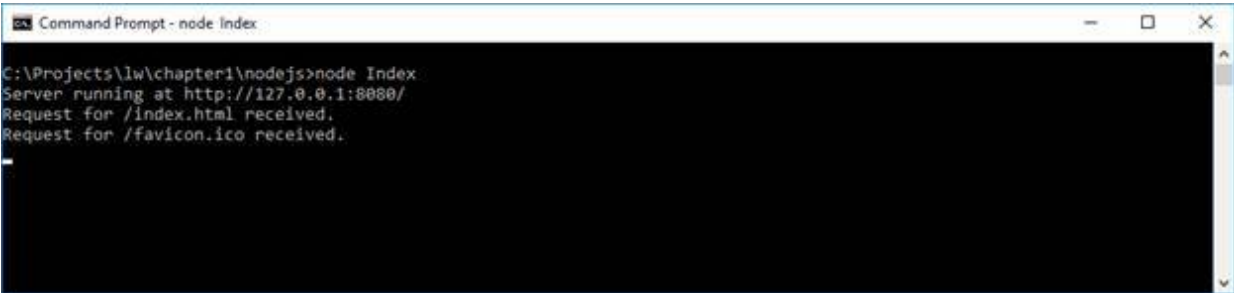
The final line of the class calls the **start()** method on the server variable. This will start the server and allow it to wait for requests. It passes in two values; the **route** function—which is retrieved from the **requestRouter** instance—and the **handlers** object—that is retrieved from the **requestHandlers** instance.

Once you have this code in place, you should be able to execute it:



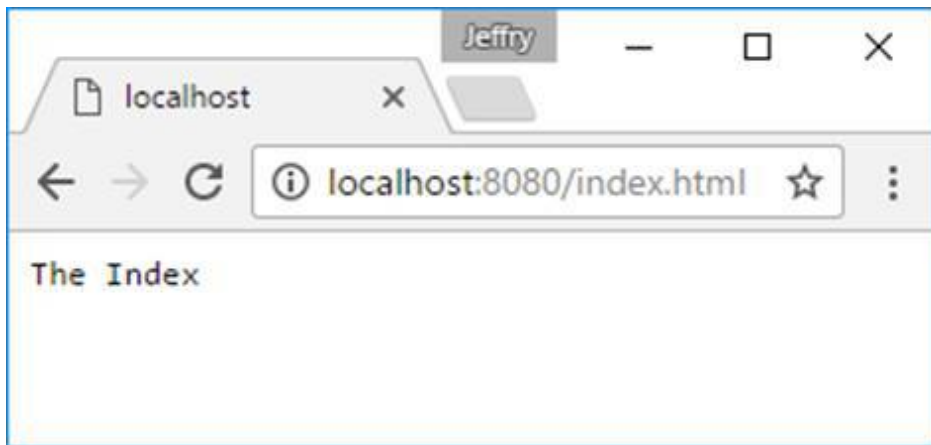
```
Command Prompt - node Index
C:\Projects\lw\chapter1\nodejs>node Index
Server running at http://127.0.0.1:8080/
```

Try to load **localhost:8080/index.html** in the browser, and you'll see the request get logged to the console:



```
Command Prompt - node Index
C:\Projects\lw\chapter1\nodejs>node Index
Server running at http://127.0.0.1:8080/
Request for /index.html received.
Request for /favicon.ico received.
```

In the browser, you should see the results of the request:



You should be all set with the primary infrastructure for the NodeJS service layer for the *Learn With* application.

Create the Application Skeleton

In this section, we will create the basic application skeleton of the Angular application. It will show you how to bootstrap your Angular application and use the routing modules to create two different screens. For the purposes of this book, I recommend you start with an empty **src** directory.

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**, and it is in the **src** directory. This page doesn't display anything yet, but is about as simple as they come. As part of the head, load the style sheet:

```
<link rel="stylesheet" href="app.min.css">
```

The **app.min.css** file will be generated by our build script by compiling all the relevant CSS. Create a **styles** directory and put an empty **styles.css** in it.

Next, we need to load the JavaScript libraries that power Angular:

```
<script src="js/core-js/client/shim.min.js"></script>
<script src="js/zone.js/dist/zone.js"></script>
<script src="js/reflect-metadata/Reflect.js"></script>
<script src="js/systemjs/dist/system.src.js"></script>
```

Our build script will copy these files from the **node_modules** directory into the **build** directory.

Set up the SystemJS Config

In AngularJS 1.x applications, it was important to minimize our libraries and combine them into a big, tight file. Angular has moved to a module loading system. This means that libraries are loaded as needed instead of loading the whole app at once. This cuts down on load times, because less files are

being loaded at once. It also requires some up-configuration. We're going to have to tell SystemJS how to find Angular and related libraries.

Create a file named **systemjs.config.js** in the **js/systemJSConfig** directory. Start by creating an Immediately-Invoked Function Expression (IIFE):

```
(function (global) {  
})(this);
```

The **this** value is sent into the function, representing a reference to the page's global space. Next, create the **config** object inside the IIFE:

```
System.config({  
});
```

The **config** is an object which will tell SystemJS how to discover the libraries. We're going to set three different values in the **config** object. The first is the **paths**:

```
paths: {  
  'js:': 'js/'  
},
```

The **paths** object is just an alias that points to the root for all our JavaScript files. Next is a **map** object. The **map** tells SystemJS that when it finds a specific path—such as **@angular/core**—it should look for the library at the specified location—such as “js:@angular/core/bundles/core.umd.js”. Notice that the path location makes use of the path alias defined in the previous code snippet. Here is the map section:

```
map: {  
  app: 'com',  
  '@angular/core': 'js:@angular/core/bundles/core.umd.js',  
  '@angular/common': 'js:@angular/common/bundles/common.umd.js',  
  '@angular/compiler':  
  'js:@angular/compiler/bundles/compiler.umd.js',  
  '@angular/platform-browser':  
  'js:@angular/platform-browser/bundles/platform-  
browser.umd.js',  
  '@angular/platform-browser-dynamic':  
  'js:@angular/platform-browser-dynamic/bundles/platform-  
browser-dynamic.umd.js',  
  '@angular/http': 'js:@angular/http/bundles/http.umd.js',  
  '@angular/router': 'js:@angular/router/bundles/router.umd.js',  
  '@angular/forms': 'js:@angular/forms/bundles/forms.umd.js',
```

```
'rxjs': 'js:rxjs'
},
```

This defines our local application’s code—in the **com** directory—all the Angular libraries, and rxjs which is used under the hood by Angular.

Finally, the packages are defined:

```
packages: {
  app: {
    main: './dotComIt/learnWith/main/main.js',
    defaultExtension: 'js'
  },
  rxjs: {
    defaultExtension: 'js'
  }
}
```

Two packages are defined here. One for the rxjs library. The second is for our main application’s library. This specifies the default extension of the code files—**js**—and the main entry point of the application, placed in the **com/dotComIt/learnWith/main/main.js** directory. The **defaultExtension** property, and the **main** property refer to “js”. Our application files will be TypeScript files with the extension “ts”. However, the compilation process will turn them into “js” files, and those need to be referenced here.

Be sure to load the SystemJS config file in the main **index.html**:

```
<script src="js/systemJSConfig/systemjs.config.js"></script>
```

That prepares the SystemJS module loader for finding and loading the Angular modules, and our custom application files.

Setup the Angular Module

It is time to build the Angular glue that will create the base of our application. We’re going to create three files. The first is the application’s main component, which will contain the main display. The second will set up the application module, and the third will load the module.

First, create the main component—**app.component.ts**—in the **com/dotComIt/learnWith/main** directory. Next, import the **Component** class from the **@angular/core** library:

```
import { Component } from '@angular/core';
```

This makes the **Component** class available for use. Now, create the **Component**:

```
@Component({
  selector: 'lw-app',
  template: '<h1>Hello World</h1>',
})
```

The **@Component** text is called an annotation in TypeScript world. An annotation adds declarative data to the component. In this case, we are telling Angular that the name of this component is **lw-app**; short for LearnWith application. We are also telling it that when it encounters this selector as part of HTML in our app, it should display the template. In this case, a simple “Hello World” message. It is analogous to an **ngApp** directive of an Angular 1 application.

Finally, export the class:

```
export class AppComponent { }
```

The class portion of the component is where we’d put all our code or business logic. The class is analogous to the controller of an Angular 1 application. For now, we’re keeping the class empty, but we will populate them with code throughout this book.

Now, create the Angular module. Create a file named **app.module.ts** in the **com/dotComIt/learnWith/main** directory. To begin, import two Angular classes:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
```

The **NgModule** class is the one that sets up the main Angular application. The **BrowserModule** class is the browser specific module that loads the Angular application into the browser. It contains shared code and compile time. Next, import the custom **AppComponent** we created earlier:

```
import { AppComponent } from './app.component';
```

This is so we can tell the module to load our custom component.

With imports complete, we want to set up the **@NgModule** annotation:


```
@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:   [ AppComponent ]
})
```

There are three aspects of the **@NgModule** annotation. First, the **BrowserModule** class is imported as part of the module. Then, our **AppComponent** is declared. The final step is to bootstrap our **AppComponent**. This tells Angular that the **AppComponent** is the main entry component of this application. By loading that, it will load all the relevant pieces of the app as needed.

I wanted to explain the difference between imports and declarations. They are very similar, but subtly different. If we need to use functionality wrapped up in its own module, then we must import it into our own module. If we just need to use a component that's not in its own module, then we use the **declarations** tag. When preparing a set of components for reuse, we'd put them in a module of their own. This way, they live in their own namespace, independent of our own app. For the most practical purposes, we won't need to create our own modules.

The final step in the file is to export the class:

```
export class AppModule { }
```

That completes the **app.module.ts** file.

Finally, create the **main.ts** file in the **com/dotComIt/learnWith/main** directory. This file was mentioned in our SystemJS configuration file as the entry point of our application. First, import the Angular **platformBrowserDynamic** library:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
```

The **platformBrowserDynamic** library takes care of run-time processing or templates including features such as binding other interactions. Then, load the **appModule**:

```
import { AppModule } from './app.module';
```

The last step is to put it all together:

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

Call the **bootstrapModule()** function on the **platformBrowserDynamic()** to call and pass in the **AppModule**. This will get the Angular app working to load views, services, or whatever else is needed.

Before we can compile and run the app, we need to make a few additions to the **index.html** in the **src** directory. First, use the SystemJS library to load the app:

```
<script>
  System.import('app').then(null, console.error.bind(console));
</script>
```

Remember that **app** is defined in the SystemJS configuration to point to the **com/dotComIt/learnWith/main/main.js** file, and the **main.js** file is actually the compiled version of **main.ts**. The **main.ts** file references the **app.module.ts**, which in turn references the **app.component.ts**. A real application will have more files and components.

The last step is to add the app selector to the main body:

```
<body>
  <lw-app>Loading AppComponent content here ...</lw-app>
</body>
```

The **lw-app** tag is our custom HTML tag which was created by the selector in the **app-component.ts**. Now you can compile your app and load it in a browser. If you have the **buildWatch** script running, everything should already be compiled. If not, just run this:

```
gulp build
```

Then load your browser:



Congratulations! You just compiled your first Angular application.

Set up the Routes

With a basic structure in place, you'll want to create an infrastructure for building the two main screens of this application. The first will be the login screen, and the second will be the main task display. For now, we'll just use placeholder screens and implement them in future chapters.

Create the Login Component

The first step is to create one component for each view, starting with the login component. Each component will have three parts:

- A TypeScript class containing the Angular code.
- An HTML File containing the view template.
- A CSS file containing style specific to this component.

Start by creating a new directory at **com/dotComIt/learnWith/views/login**. Create the TypeScript file first; **login.component.ts**. Import the **Component** class from the **@angular/core** library:

```
import { Component } from '@angular/core';
```

Next, create the **@Component** annotation:

```
@Component({
  selector: 'login',
  templateUrl :
  './com/dotComIt/learnWith/views/login/login.component.html',
  styleUrls: [
  './com/dotComIt/learnWith/views/login/login.component.css' ]
})
```

This specifies the selector; login. Previously when we created a component we specified a **template** value and created an in-line template. In this case, I specified a **templateUrl**. It is easier to write and change extended HTML templates in their own file. The path to the HTML component is relative to the root directory of the app, where the **index.html** page will be. A **styleUrl** property is also specified. This value is an array; loading all the styles associated with this component. A cool thing about **styleUrls** is that Angular keeps the style sheets separate. This way, if you load two components that have similarly named styles, they will not overwrite each other.

To finish the component, export the **LoginComponent** class:

```
export class LoginComponent { }
```

Next, create the **login.component.css** file. For now, it is a placeholder and will be blank. Create the **login.component.html** file:

```
<h1>Login View</h1>  
<a [routerLink]="['/tasks']">Go Back to Tasks</a>
```

This is the simplest template we can create. It includes a header stating which view is being displayed and includes a link to the tasks view. The **a** tag uses a **routerLink** property instead of the traditional **href**. Behind the scenes, Angular automatically decides how to direct the anchor to the proper route.

The Gulp build script automatically knows how to copy the new HTML file and CSS file into the build directory with one caveat. As you create new HTML pages, the **buildWatch** task will not automatically copy them over. It only noticed changed files, not new files. You'll have to manually rerun the task for it to be aware of the new files you just added.

The final step is to load the **LoginComponent** in the **app.module.ts** file. Open it up and import the **LoginComponent**:

```
import {LoginComponent} from "../views/login/login.component";
```

The path to find the **login.component.ts** file is relative to the **app.module.ts** file, so it moves a directory up from **main** to **learnWith**, and then traverses down to **views/login/**. Now, add the **LoginComponent** to the declarations:

```
declarations: [  
  AppComponent,  
  LoginComponent  
],
```

The **LoginComponent** is ready to be used in the app, but nothing is coded to make it show up yet. We'll fix that after creating the Tasks Component.

Create the Tasks Component

As with the **LoginComponent**, the **TasksComponent** will be made up of a TypeScript file, a CSS file, and an HTML Template file. Create a new directory at **com/dotComIt/learnWith/views/tasks**. Add an empty file named **tasks.component.css**. CSS will be populated in later chapters as required.

Create the TypeScript file next, **tasks.component.ts**. Import the **Component** class from the **@angular/core** library:

```
import { Component } from '@angular/core';
```

Next, create the **@Component** annotation:

```
@Component({
  selector: 'tasks',
  templateUrl :
  './com/dotComIt/learnWith/views/tasks/tasks.component.html',
  styleUrls: [
  './com/dotComIt/learnWith/views/tasks/tasks.component.css' ]
})
```

This specifies the selector, **tasks**. The **templateUrl** is pointed to the **tasks.component.html** file, relative to the **index.html** location. The **styleUrl** array points to the **tasks.component.css** file previously created.

To finish the component, export the **TasksComponent** class:

```
export class TasksComponent { }
```

The third file is the **login.component.html** file:

```
<h1>Tasks View</h1>
<a [routerLink]="['/Login']">Go Back to Login</a>
```

This template parallels the login template. It includes a header stating the tasks view is being displayed and includes a link to the login view. The **routerLink** property is used once again to create the link. The Angular router will do its magic under the hood to load a new view without redirecting the page.

The final step is to load the **TasksComponent** in the **app.module.ts** file. Open it up and import the **TasksComponent**:

```
import { TasksComponent } from "../views/tasks/tasks.component";
```

The path to find the **tasks.component.ts** file is relative to the **app.module.ts** file, so it moves up a directory from **main** to **learnWith**, and then traverses down to **views/login/**. Now, add the **TasksComponent** to the declarations:

```
declarations: [
  AppComponent,
  LoginComponent,
```

```
    TasksComponent
  ],
```

Now both views are ready to be used in the app, but nothing is coded to make it show up yet. Next, we'll write the code to create the routing module.

Create the Routing Module

The Routing Module is the way the app tells the browser which components to load based on which view is loaded. Create the file **routing.module.ts** in the directory **com/dotComIt/learnWith/nav**. First, import some Angular classes:

```
import { NgModule }      from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
```

First the **NgModule** is imported from **@angular/core**. The route is created as a module instead of a component. Next, two classes from the **@angular/router** are imported; the **RouterModule** and **Routes**. The **RouterModule** represents the underlying code to change and load routes. The **Routes** class represents an array of route objects. Each **route** will be defined as a single object. Create the **routes** array:

```
const ROUTES : Routes = [
  { path: 'login', component: LoginComponent },
  { path: 'tasks', component: TasksComponent },
  { path: '', redirectTo: 'login', pathMatch: 'full' },
  { path: '**', redirectTo: 'login' }
];
```

This array is created as a constant using the **const** keyword. Constants are like variables, except their values are set once and do not change. The value of the **ROUTES** constant is of type **Routes**, the class we imported. Its value is an array of objects, each one representing a different route.

The first route is the login route, which will load the **LoginComponent**. The second is the tasks route, which will load the **TasksComponent**. The third route is where things start to get tricky. The path is an empty string, which you'll see on the application's initial load. It does not load a route. Instead, it redirects to the login path. The final path uses a wild card to discover the root. If the root is unknown, it also redirects to the login route.

Next, create the **@NgModule** annotation:

```
@NgModule({
  imports: [ RouterModule.forRoot(ROUTES) ],
  exports: [ RouterModule ]
})
```

This defines **imports** and **exports**. The **imports** calls the **forRoot()** method on the **RouterModule** to return a **routes** module with our defined **routes**, and a **router** service. The **exports** of the **RouterModule** means that other modules which import this new module will have access to use the **RouterModule**'s classes and components within their own templates.

Finally, export the class:

```
export class AppRoutingModule {}
```

With the routing module created, we must now load it in the **app.module.ts** file. First, import the class:

```
import { AppRoutingModule } from '../nav/routing.module';
```

Now, inside the **@NgModule** imports, list the **AppRoutingModule**:

```
imports: [
  BrowserModule,
  AppRoutingModule
],
```

That is the only change required to the main module. However, I want to show you one more. The **RouterModule** modifies the browser's URL as the user moves around the app. There are a few different strategies for this change, and the default one must be done in conjunction with server-side configuration. Otherwise the user may see 404 errors when reloading a screen. For example, if we were to load the app, like this—

```
http://dev.learn-with.com/build/index.html
```

—it would redirect to the login screen:

```
http://dev.learn-with.com/build/index.html/login
```

Now reload the page, and you'll get a 404. That is because **index.html/login** is not a valid directory. The solution is to tell Angular to use page anchors as

part of the URL. First, import the **HashLocationStrategy** and **LocationStrategy** classes from **@angular/common**:

```
import { HashLocationStrategy, LocationStrategy } from
 '@angular/common';
```

Then, add a provider as part of the **@NgModule** annotation:

```
providers : [{provide: LocationStrategy,
 useClass:HashLocationStrategy}],
```

This will add anchor tags as part of the URL redirect. So loading this—

```
http://dev.learn-with.com/build/index.html
```

—would redirect to the login screen, like this:

```
http://dev.learn-with.com/build/index.html#/login
```

This is a perfectly valid URL for reloading without any special web server configuration.

Put it all Together

We have to make a few more changes before we're ready to test the app. First, open up the **app.component.ts** file. Change the template to include the **router-outlet** tag:

```
@Component({
  selector: 'lw-app',
  template: `<router-outlet></router-outlet>`,
})
```

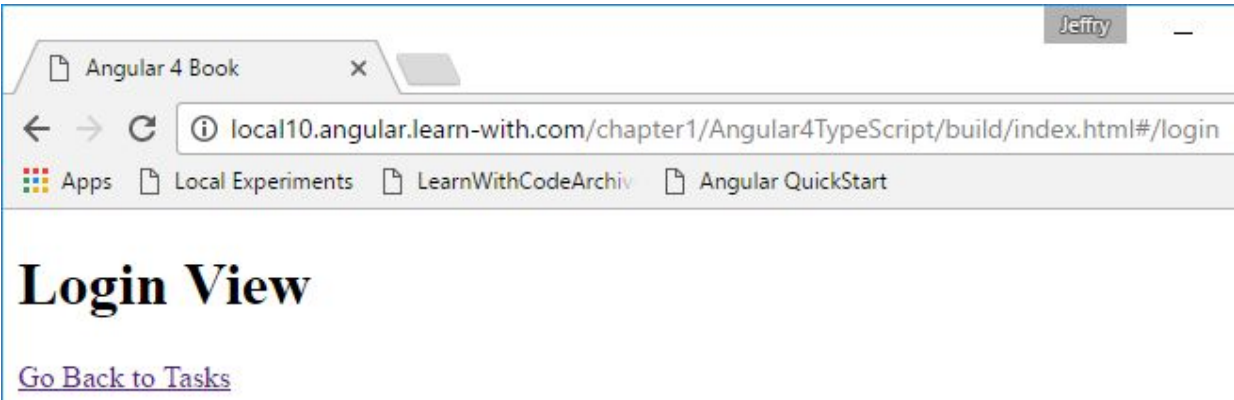
The **router-outlet** tag is a custom Angular tag, just like our **lw-app** tag. It comes from the routing module and tells Angular to put the routing views here.

Next, open the **index.html**. We'll need to add a base **href** for the router to correctly change the URL. I use some JavaScript to make it work:

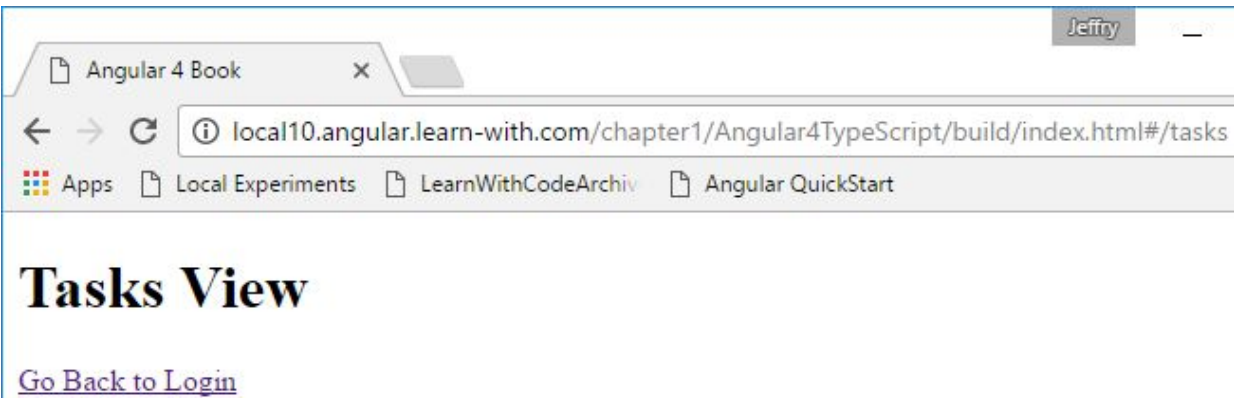
```
<script>document.write('<base href="' + document.location + '" />');
</script>
```

The short JavaScript snippet finds the location of the main index file, and the router modules will use it to change the URL after that specified location.

Now, recompile your app and load it in a browser:



We can see on initial load we see the Login view and the URL specifies the login route. As we expected. Click the “Go Back to Tasks” link:



We can see that the URL has changed to show the tasks route, and the Tasks template is displayed.

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 helped you set up the application infrastructure—including a project seed—and showed you how to use the build scripts. Finally, it explained how to build the main application shell; showing you how to create components with templates and use a routing module to navigate between screens of your 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 show you how to build out the User Interface, and 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 build the login form for our application. The finished Login screen will look like this:

Login View

Username	<input type="text"/>
Password	<input type="password"/>
<input type="button" value="Reset"/>	<input type="button" value="Login"/>

This layout is simple, and should be no problem if you have moderate HTML skills. Open the **login.component.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 one 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>
  <tr>
    <td colspan="2">
      <input type="button" value="Reset" (click)="onReset()" />
      <input type="button" value="Login" (click)="onLogin()" />
    </td>
  </tr>
</table>
```

I could have created this layout using CSS, but my goal is to be effective and quick without obsessing over layouts.

The buttons at the bottom of the template have an Angular directive named **click**. The directive is surrounded by parenthesis, which is Angular's way of

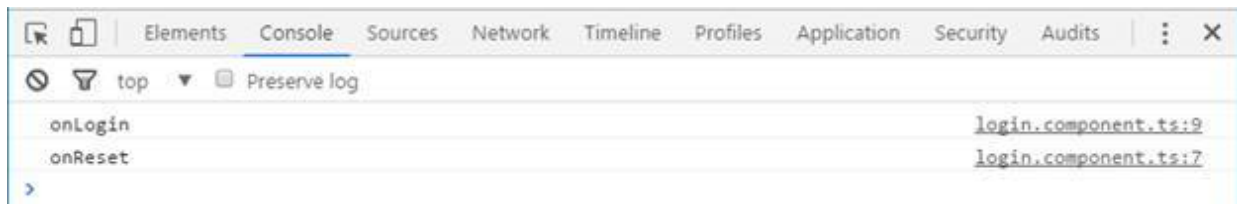
responding to an event. When one of the buttons is clicked, this directive tells Angular to execute the specified function in the login component's class.

Let's add some placeholders for each function. These functions should be put inside the **login.component.ts** from the **com/dotComIt/learnWith/views/login** directory:

```
export class LoginComponent {
  onReset(): void {
    console.log('onReset');
  }
  onLogin(): void {
    console.log('onLogin');
  }
}
```

The functions do not return anything, so the return element is void. No arguments are passed in, so nothing is specified.

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



We now have a login screen, but it isn't working just yet.

Creating Value Objects

Before jumping into the service code, I wanted to create some value objects used by the UI. A Value Object is, basically, a data container.

The Generic Return Object

We'll create a class to encapsulate the values returned from services. That way the service handlers in the application know to look at the consistent format and determine whether there is an error or not. I named this object **ResultObjectVO** on the server-side entries to this series, and will name the TypeScript version the same. There are two properties in this object:

- **error**: This is a Boolean value. If the error flag is true, then the UI needs to stop processing and show an error. If the error flag is false, then the UI can continue processing.
- **resultObject**: This property is a generic object. If there is an error, then this property would contain the error text. If there is no error, then this will contain the result of the service. For our authentication service, this would contain the user details that the application needs.

A sample JSON packet for a successful service may look like this:

```
{
  "resultObject": [
    {
      "someData": 1,
      "someOtherData": "me",
    },
    {
      "someData": 2,
      "someOtherData": "you",
    }
  ], "error": 0
}
```

The error property is set to "0", meaning no error occurred. The **resultObject** is set to an array of objects.

A sample packet from a failed authentication call may be:

```
{
  "error": 1,
}
```

```
"responseObject": "Something Bad Happened"
}
```

The error property on the object is set to "1" and the **responseObject** contains a string describing the error.

Let's create the class. Create a new directory in **com/dotComIt/learnWith/vo**. Inside that directory, create a new TypeScript file—**ResultObjectVO.ts**. This is the contents of the file:

```
export class ResultObjectVO {
  responseObject : any;
  error : boolean;
};
```

The class definition for the **ResultObjectVO** is created, with two properties defined. The **responseObject** has a type of "any" and the **error** has a type of "boolean". This can easily be used by other components.

Create a User Value Object

Since TypeScript is statically typed, I'm also going to create a user object to represent a user who signed into the app. The user object will contain these properties:

- **userID**: This property contains a unique ID that represents the user.
- **username**: This property contains the user's login in name.
- **password**: This property contains the user's passwords. For most practical purposes, we would not send the password back to the app after authentication the user.
- **role**: This property will become relevant in later chapters when we implement role based authentication.

Create a file named **UserVO.ts** in the **com/dotComIt/learnWith/vo** directory. This is how to create it:

```
export class UserVO {
  userID: number;
  username: string;
  password: string;
  role: number;
}
```


The four properties are defined, and the **UserVO** class is exported. It is a simple class.

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 application, each user will only have a single role. This would be a one-to-many relationship in database-speak.

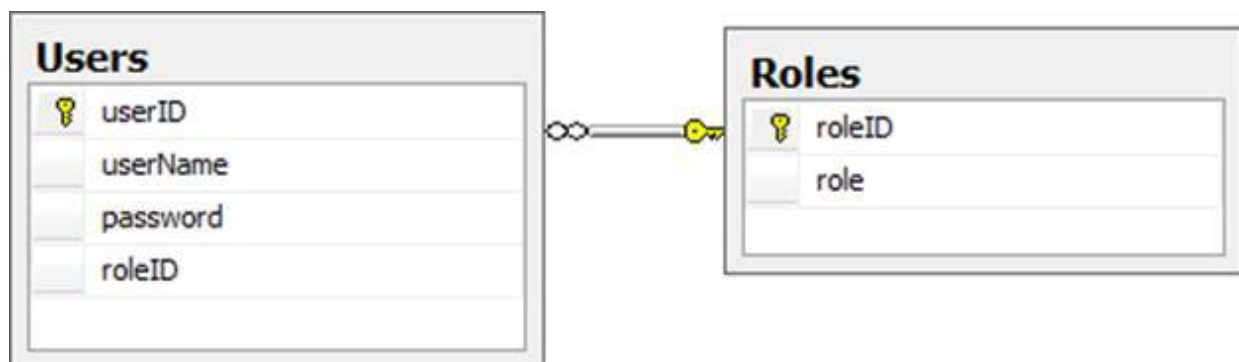
The data to store for users is this:

- **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 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 will cover the NodeJS code needed for the authentication. It will talk about the database driver used to access a Microsoft SQL Server database, and show you how to use that in an item. It will also explain how to create the JSON results that will be returned from the services. Lastly, it will cover the actual service code needed to perform authentication.

Install MSSQL NodeJS Driver

Most of my consulting clients over the years have used SQL Server, which is why I use it here. However, the database structure and SQL is simple enough that any database should work for your purposes. Setting up the NodeJS connection will require a specific package based on the database server of your choosing.

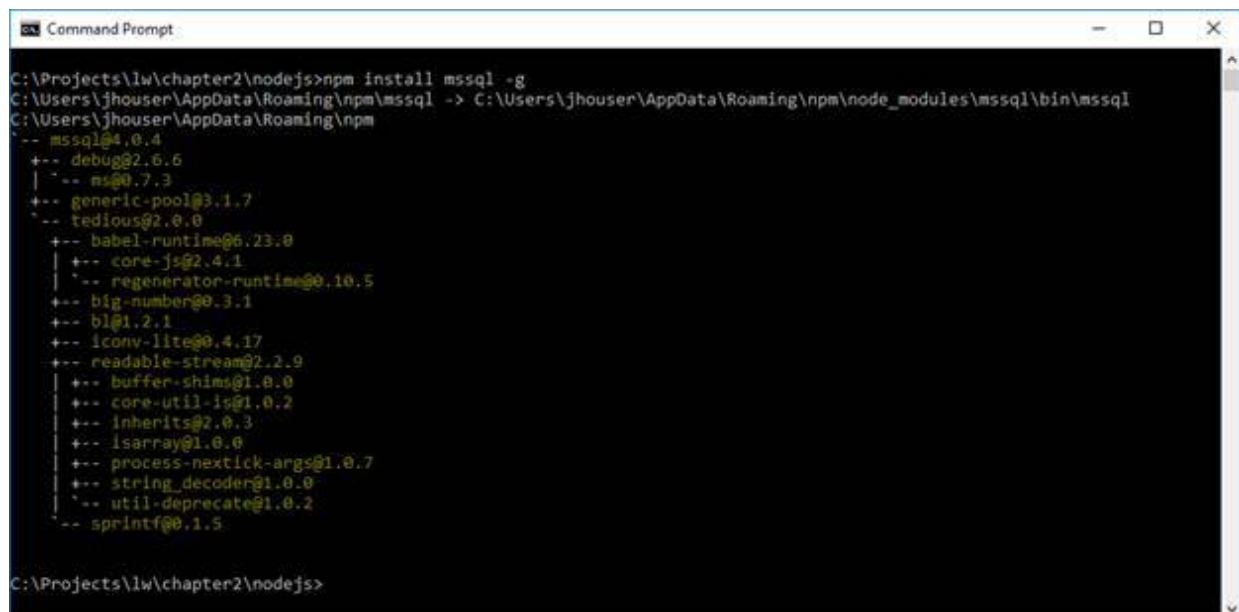
The package I'm using is named [mssql](#). To install the package for use in NodeJS, just open up a console window and run this:

```
npm install mssql
```

Optionally, you can use the "-g" flag to install it globally:

```
npm install mssql -g
```

You should see a result similar to this:



```
Command Prompt
C:\Projects\lw\chapter2\nodejs>npm install mssql -g
C:\Users\jhouse\AppData\Roaming\npm\mssql -> C:\Users\jhouse\AppData\Roaming\npm\node_modules\mssql\bin\mssql
C:\Users\jhouse\AppData\Roaming\npm
  -- mssql@4.0.4
  +-+ debug@2.6.6
  | +-+ ms@0.7.3
  +-+ generic-pool@3.1.7
  +-+ tedious@2.0.0
  +-+ babel-runtime@6.23.0
  | +-+ core-js@2.4.1
  | +-+ regenerator-runtime@0.10.5
  +-+ big-number@0.3.1
  +-+ bl@1.2.1
  +-+ iconv-lite@0.4.17
  +-+ readable-stream@2.2.9
  | +-+ buffer-shims@1.0.0
  | +-+ core-util-is@1.0.2
  | +-+ inherits@2.0.3
  | +-+ isarray@1.0.0
  | +-+ process-nextick-args@1.0.7
  | +-+ string_decoder@1.0.0
  | +-+ util-deprecate@1.0.2
  +-+ sprintf@0.1.5
C:\Projects\lw\chapter2\nodejs>
```

Now you should now be able to use the **mssql** package inside of your NodeJS application.

Creating a DatabaseConnection Package

When originally architecting the NodeJS application, I wanted to try to encapsulate as much as possible. So, I created a single NodeJS module that would handle all the database interactions. Ideally, this will make it easy to replace the module with one that uses a different database driver. I created the file **DatabaseConnection.js** in the folder **com/dotComIt/learnWith/database**.

First, we need to load the **mssql** driver:

```
var sql = require("mssql");
```

This uses the **require** statement to load the package we installed in the previous section. This package will need a **config** object that can be used to tell the driver how to find the database:

```
var config = {
  user: 'LearnWithUser',
  password: 'password',
  server: 'dev.mySQLServer.com',
  database: 'LearnWithApp',
  port : 1433
};
```

The **config** object specifies a username, password, server location, and the name of the database. You'll have to replace these values with the ones from your own setup.

Next create an **executeQuery()** function:

```
var executeQuery = function(query, resultHandler, failureHandler){
};
```

The **executeQuery()** function will be the method used by others packages to execute queries. It accepts three arguments:

- **query**: This argument is a string that represents the query we want to execute.
- **resultHandler**: This argument represents a function—or callback—which will be executed upon successful execution of the query.

- **failureHandler**: This argument represents a function—or callback—to be executed if there is an error running the query.

Populate the body of the **executeQuery()** method:

```
var connection = new sql.ConnectionPool(config);
connection.connect(function() {
    new sql.Request(connection).query(query)

.then(resultHandler).catch(failureHandler);
});
```

First, the **connection** object is created with the **ConnectionPool()** method of the **sql** class. This accepts the **config** object as an argument. The **connect()** function is executed off of that connection object. It accepts a single argument; the function to **run** when the connection is made.

Inside the function, a new instance of **sql.Request()** is created. The **query()** method is run against it. This will execute the query. This is also done asynchronously. If a successful result is returned from the query, then the **resultHandler** callback is executed inside the **then()** command. If an error occurs, the **catch()** function is executed, and the **failureHandler** argument's callback is executed. The primary reason for the use of callbacks is so that we can use the same code to execute the query, but handle results differently for different types of calls. Service calls that load tasks will handle results differently than the service call to authenticate a user. This allows us a lot of flexibility in our implementation, while still encapsulating the database call code.

The final step in this package is to expose the **executeQuery** function:

```
exports.executeQuery = executeQuery;
```

This will allow for the **executeQuery()** method to be called from other packages.

Creating JSON in NodeJS

In my own development, I often return a consistent object from all my services. This helps when building UI Services because I always have a convention that I can use for the type of data that came back. NodeJS doesn't have the concept of classes in the same manner a statically typed

language would, but we can still create a standard to represent a generic result object.

These are the properties of the result object:

- **error**: This is a Boolean value. If the **error** flag is “true”, then the UI will know that there is an error, and to stop further processing. In this case, it 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.
- **responseObject**: This property is a generic object. If there is an **error**, then the **responseObject** will contain a string description of the error that the UI can display to the user. If there is no **error**, then the **responseObject** 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 **responseObject** based on the type of call made and the information expected back.

In NodeJS, we are going to create this as a regular JavaScript object, and then convert it to JSON before returning it to the user. I created a NodeJS module to handle the generic conversions. Create the file **JSONResponseHandler.js** and put it in the **com/dotComIt/learnWith/server** directory.

First, create an **execute()** function:

```
function execute(response, data, callback){  
}
```

This function accepts three parameters. The first, “response”, represents the **response** object that can be used to send data back to the user. The second argument is “data”, which represents the value needed to be sent to the user. The third argument is “callback”. This is a string argument, which represents a **callback** that will wrap the JSON in case we want to use JSONP. When calling the NodeJS code from the Angular app, JSONP will be used.

Next, populate the response header:

```
response.writeHead(200, {"Content-Type": "application/javascript"});
```

The response header is a **200** code, and is of type "application/javascript". Then convert the data string into JSON:

```
var json = JSON.stringify(data);
```

The JSON object is a JavaScript object and the **stringify()** method is used to create a JSON string. This will be the results we want to send back to the user.

Next, check for a **callback**. If it exists, wrap the JSON string in the **callback** function call:

```
if(callback != ''){  
    json = callback + "(" + json + ")";  
}
```

Finish off the **execute** method by writing the final output with the **write()** method, and closing the response with the **end()** method.

```
response.write(json);  
response.end();
```

The last step in our **JSONResponseHandler.js** file is to export the execute method:

```
exports.execute = execute;
```

This module exists as a helper function used to process the output of our service calls.

Create the AuthenticationService

This section will create the actual login service. Create a file named **AuthenticationService.js** in the **com/dotComIt/learnWith/services** directory. The **AuthenticationService** will make use of the **DatabaseConnection** module to talk to the database, and the **JSONResponseHandler** module to return a JSON result. The first step is to import those two modules with the **require()** method:

```
var databaseConnection =  
require("../database/DatabaseConnection");  
var responseHandler = require("../server/JSONResponseHandler")
```

Next, create the handler function:


```
function login(response, queryString) {  
}
```

The handler function for the authentication method is named **login()**. It accepts a “response” argument, and a “queryString” argument. The first step inside this method is to initialize a **response** object:

```
var responseObject = {};
```

The **responseObject** variable will represent the **ResultObjectVO**, which contains an error property and a **responseObject** property. Next, see if a callback is specified as part of the query string:

```
var callback = '';  
if(queryString.callback !== undefined){  
    callback = queryString.callback;  
}
```

The **callback** variable is initialized to an empty string. Then, we check to see if the **queryString** argument contains a **callback** variable. If it does, then the **callback** is copied over to the local function. If it is not, then the **callback** remains as an empty string.

Next, the method needs to validate the input. If the username and password are not defined in the **queryString**, then there is no need to attempt authentication:

```
if((queryString.username === undefined) ||  
    (queryString.password === undefined)){  
    responseObject.error = 1;  
    responseHandler.execute(response, responseObject, callback);  
}
```

If this error occurs, then the **responseObject** value is given an **error** property with the value of “1”. Then the **JSONResponseHandler** is called to convert the **responseObject** into JSON, add the **callback**—if it exists—and return the information to the client.

```
} else {  
    query = "select * from users ";  
    query = query + "where username = '" + queryString.username + "'";  
    query = query + "and password = '" + queryString.password + "'";  
}
```

This uses some string concatenation to create the database query. Notice, the **else** statement group is not closed with a curly bracket yet. Next, the **databaseConnection** object is used to execute the query:

```
var dataQuery = databaseConnection.executeQuery(query,
function(result) {
    if(result.recordset.length == 1){
        responseObject.error = 0;
        responseObject.resultObject = {};
        responseObject.resultObject.userID =
result.recordset[0].userID;
        responseObject.resultObject.username =
result.recordset[0].userName;
        responseObject.resultObject.role = result.recordset[0].roleID;
    } else {
        responseObject.error = 1;
    }
    responseHandler.execute(response, responseObject, callback);
},
function(err){
    console.log('AuthenticationServiceExecuteFailureHandler')
    console.log(err);
    responseObject.error = 1;
    responseHandler.execute(response, responseObject, callback);
});
} // this ends the else statement
```

The **executeQuery()** method was the one we created in the **DatabaseConnection** class. It accepts three arguments; a string argument representing the query to execute, a result handler function, and an error handler function. I created both the result handler and the error handler functions in-line.

First, the result handler function checks to see if any items were returned in the result object's **recordset**. If so, then a **success** object is created, copying the user data into the **responseObject.resultObject** property. Finally, the **responseHandler** is used to create the JSON result and send that back to the user. If not, the **error** property of the **responseObject** is set. The final aspect of the result handler method is to use the **responseHandler** to return the JSON result to the calling entity.

The failure handler method assumes there is an error. It sets the **error** property on the result object to "1", and then returns the JSON to the

calling entity.

The last step in this file is to export the login function:

```
exports.login = login;
```

This completes the **AuthenticationService**. But, we still have to tell our main app to use it. Open the **ResponseHandlers.js** class from the **com/dotComIt/learnWith/server** directory. Load the **AuthenticationService**:

```
var authenticationService =  
require("../services/AuthenticationService");
```

And add the handler:

```
handlers["/login"] = authenticationService.login;
```

Now, restart your NodeJS application and you should be good to go.

Test in a Browser

The easiest way to test this code is to launch a browser and test some URLs. Try this URL:

```
http://localhost:8080/login
```

This should give you an **error** because it has no query string parameters:

```
{"error":1}
```

Try this URL, which should log in the "me/me" user:

```
http://localhost:8080/login?  
username=me&password=AB86A1E1EF70DFF97959067B723C5C24
```

You should see a result like this:

```
{  
  "error":0,  
  "responseObject":{  
    "userID":1,  
    "username":"me",  
    "role":1  
  }  
}
```

You can try a few different options to see what the code returns. The next step is to integrate this service into the Angular application.

Access the Services

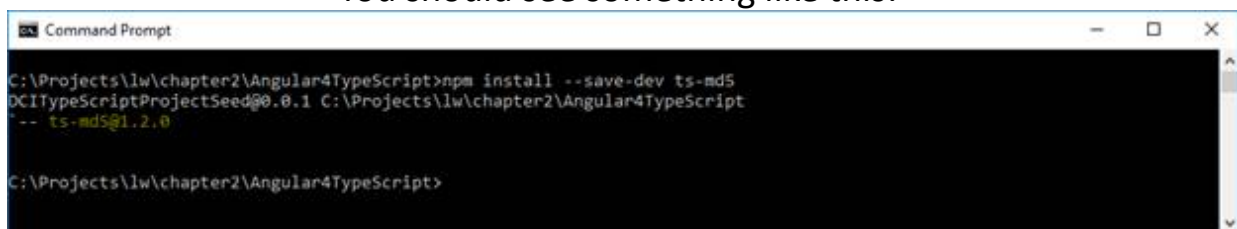
This section will create the UI Service code.

Hashing the Password with Angular

Before we examine the UI's **authenticate()** method, we'll need to setup a hashing library so that we can hash the password before it is sent over the wire to the server. We're going to use [ts-md5 library](#). Install it:

```
npm install --save-dev ts-md5
```

You should see something like this:



```
Command Prompt
C:\Projects\lw\chapter2\Angular4TypeScript>npm install --save-dev ts-md5
DCITypeScriptProjectSeed@0.0.1 C:\Projects\lw\chapter2\Angular4TypeScript
-- ts-md5@1.2.0
C:\Projects\lw\chapter2\Angular4TypeScript>
```

This library is a TypeScript implementation of the md5 hash and can be used directly from TypeScript.

We'll have to tell the build scripts how to move the file from the install location to the build location. Open the **config.js** file in the project root and find the **angularLibraries** variable:

```
angularLibraries : [
  'core-js/client/shim.min.js',
  'zone.js/dist/**',
  'reflect-metadata/Reflect.js',
  'systemjs/dist/system.src.js',
  '@angular/**/bundles/**',
  'rxjs/**/*.*',
  'angular-in-memory-web-api/bundles/in-memory-web-api.umd.js',
  'ts-md5/dist/**.*'
],
```

I added a glob to represent the ts-md5 distribution folder and related files at the end. Our build script will know to look for these values in the **node_modules** directory and move them to the **js** library of the **build** folder.

Open the **SystemJS** configuration file in **js/systemJSConfig**. We need to tell **SystemJS** how to find this library. Find the map object. At the end, add this

entry:

```
'ts-md5' : 'js:ts-md5 '
```

Now add the ts-md5 to the packages object:

```
'ts-md5': {  
  main: '/md5.js'  
}
```

Now, when the app rebuilds, you'll be able to import this library into the **AuthenticationService** and hash the password.

Create the Service

The next thing we need to cover is how to access the NodeJS services from the Angular application. To do this, we will make use of JSONP. Our app code and service code are located on different domains, since we aren't using NodeJS to deliver our build directory to the application.

JavaScript cannot access services on a different domain due to cross site scripting restrictions implemented by the browser. We can get around this a few different ways; JSONP and CORS. JSONP can be done in the calling code while CORS needs to be set up based on the remote service. We're going to use JSONP for this book, as it is easier to set up. JSONP works by using the HTML script tag to load items from different domains, but Angular has libraries that handle the complexities of it.

Create a file named **AuthenticationService.ts** and put it in the **com/dotComIt/learnWith/services/nodejs** directory. We're going to need to import a bunch of things. Start with the **Injectable** library from the **@angular/core**:

```
import {Injectable} from "@angular/core";
```

This is used to specify that this library can be used as a provider, and that it may have other providers injected into it upon creation. Now, a class from the **@angular/http** library:

```
import {Jsonp} from "@angular/http";
```

The **Jsonp** class is the Angular library for loading remote data via JSONP.

We'll need to import a few things from the rxjs library. This is a library that Angular uses under the hood to help create observables:

```
import {Observable} from "rxjs/Observable";  
import 'rxjs/add/operator/map';
```

The **Observable** property is a class. The map import will be used on the class to process the results of the service call and pass them back to the invoking code's success function. Finally, import our **ResultObjectVO**:

```
import {ResultObjectVO} from "../../vo/ResultObjectVO";
```

Before we start the class, create a constant for a service URL:

```
const server : string = 'http://127.0.0.1:8080/';
```

This points to the location of the NodeJS services.

Now, create the class:

```
@Injectable()  
export class AuthenticationService {  
}
```

This specifies the **@Injectable()** annotation. Create the constructor:

```
constructor(private jsonp: Jsonp) {}
```

The **Jsonp** service will be injected into this class via the constructor. This makes it available later for making the service call.

For the **Jsonp** service to be injected into this class, we need to set it up as a provider as part of the main module. Open up **app.module.ts** in **com/dotComIt/learnWith/main**. Import the **JsonpModule**:

```
import { JsonpModule } from '@angular/http';
```

The **JsonpModule** is a helper module that includes the **Jsonp** service. Set it up as an import as part of the **@ngModule** annotation:

```
imports: [  
  BrowserModule,  
  AppRoutingModule,  
  FormsModule,  
  JsonpModule  
],
```

Now, the **Jsonp** service will inject into the **AuthenticationService** class.

Implement the `authenticate()` method

Now, implement the **authenticate()** method inside the **AuthenticationService** object:

```
authenticate(username : string, password : string) :  
  
Observable<ResultObjectVO> {  
}
```


This method accepts two string arguments; a username and a password. It returns an **Observable** of type **ResultObjectVO**.

There are two parameters we need to pass to the remote service; the username, the password. A third parameter—**callback**—is included as part of the string. The value of this is **JSONP_CALLBACK**, and it is a default Angular value. The NodeJS code will use the **callback** to wrap the resulting JSON in the function call for **JSONP_CALLBACK**, and Angular will automatically execute the function to get the returned data.

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:

```
let parameters : string = "username" + '=' + username + '&';
parameters += "password" + '=' + Md5.hashStr(password) + "&";
parameters += "callback" + "=" + "JSONP_CALLBACK" ;
```

The code block creates a parameter variable and adds the service method arguments to it one by one. The username and callback 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. Don't forget to import the MD5 hash library:

```
import {Md5} from 'ts-md5/dist/md5';
```

The final parameter string will look like this:

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

Create a variable to hold the URL to make the call:

```
let url = server + 'login?' + parameters;
```

The last step is to make the method call.

```
return this.jsonp.request(url)
    .map((result) => result.json() as ResultObjectVO );
```

The argument to the post method is the **url** variable, representing the endpoint of the service. The **Jsonp** call returns an **Observable**, which is returned from the method. We use the dot notation to run a **map()** function against the results. When something is returned, the **map()**

function will be executed. The results of the map will be returned to the calling code's success method. Our **map()** turns the JSON into a **ResultObjectVO** instance. Behind the scenes, Angular will wait for the service to return data before running the **map()** function.

Turn the Service into a Provider

We want to set up the **AuthenticationService** as a provider for our application. Open the **app.module.ts** file in the **com/dotComIt/learnWith/main** directory. Import the Authentication service:

```
import {AuthenticationService} from
    "../services/nodejs/authentication.service";
```

Then load the **AuthenticationService** as a provider:

```
providers : [
    {provide: LocationStrategy, useClass:HashLocationStrategy},
    UserModel,
    AuthenticationService
],
```

If you're reviewing the code archive for this series of books, you'll notice that the code for this book is shared between other books in this series. That is because the bulk of it is identical, with a few minor tweaks. I did create some NodeJS specific files:

- **index_NodeJS.html**: This is for the main HTML file. It loads a NodeJS specific version of the SystemJS config, but otherwise is not changed. For the bulk of this text, I'll refer to it as **index.html**.
- **systemjs.config.nodejs.js**: A NodeJS specific version of the SystemJS configuration. It changes the main app to the NodeJS main file. For the bulk of the text I'll leave out the **nodejs** in the file name.
- **main.nodejs.ts**: This is the NodeJS main application file, used to load the NodeJS main module. As with other files, I won't mention the **nodejs** when referring to this file.
- **app.module.nodejs.js**: This is the application's main module. It is used to load specific services and providers for accessing a **NodeJS** server and ignore code for other servers.

If you build the app from scratch with the book, which I strongly recommend, you won't run into problems. But, when you review the code archive for each chapter, this is something to be aware of.

Wire Up the UI

This section will demonstrate how to complete the implementation of the login functionality. It will explain how to wire the service layer up to the user interface, and successfully authenticate the user. Along the way, it will also show you how to share data between multiple components, access user input forms, and conditionally hide HTML elements.

Creating a UserModel

This section will show you how to create a **UserModel**. It will be a TypeScript class that is used for sharing user data between the different parts of this application.

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

First, the file requires us to import the **UserVO** class:

```
import {UserVO} from "../vo/UserVO";
import {Injectable} from "@angular/core";
```

I also imported the **Injectable** class from the **@angular/core** library. This will allow us to easily set up this new class as an Angular provider.

Then, create the class definition:

```
@Injectable()
export class UserModel {
  user : UserVO;
};
```

This is a TypeScript class with no dependencies upon the Angular framework. It contains a single object; the user. To use this class in our application, we need to set it up as a provider in the **app.module.ts**.

First, import it:

```
import {UserModel} from "../model/usermodel";
```

Then, modify the providers array of the **@NgModule** annotation:

```
providers : [
  {provide: LocationStrategy, useClass:HashLocationStrategy},
  AuthenticationService,
```

```
UserModel  
],
```

This will make an instance of the **UserModel** class available to all our underlying components.

Accessing Component Values from Within the View

Next, we want to associate the input values for username and password to values in the login component. First, open up the **login.component.ts** file in the **com/dotComIt/learnWith/views/login** directory. Create two properties on the **LoginComponent** class:

```
username = '';  
password = '';
```

Then, switch over to the **login.component.html** template in the same directory. We are going to use a directive named **ngModel**:

```
<input type="text" [(ngModel)]="username">
```

The **ngModel** is an Angular directive. It is enclosed in array notation—`[]`—and parenthesis notation—`()`. This is Angular short form for two-way data binding. When the value of the input changes, the variable changes in the component. On the other hand, when the variable changes in the component, the value displayed in the HTML template changes. Do the same thing for the password:

```
<input type="password" [(ngModel)]="password">
```

The **ngModel** directive is part of the Angular **FormsModule** library. That library will need to be loaded in the **app.module.ts** class. First, import it:

```
import { FormsModule } from '@angular/forms';
```

Then add it to the imports:

```
imports: [  
  BrowserModule,  
  AppRoutingModule,  
  FormsModule,  
],
```

This step is required, or else the **ngModel** will throw runtime errors suggesting an unknown property on the input tag.

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. Earlier in the chapter, you saw the **click** on the reset button and created a method stub for **onReset()**. The code for the **onReset()** function is inside the **login.component.ts** file. Previously, we had created a method stub.

Here is the full implementation:

```
onReset(): void {  
    this.username = '';  
    this.password = '';  
}
```

The code inside the button accesses the local **username** and **password** variables and sets them to blank strings. 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 the login button up 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 the **Login.component.html** template in the **com/dotComIt/learnWith/views/login** directory. Here, we are going to add more table cells to the layout table.

First, add a new table row to the top of the table:

```
<tr *ngIf="loginError">  
    <td colspan="3">  
        <div class="alert alert-danger warningAlert ">  
            <h2>{{loginError}}</h2>  
        </div>  
    </td>  
</tr>
```

This will display any login error returned from the service. It is displayed as a Bootstrap alert. Bootstrap is a CSS library, so we'll need to load that in the main **index.html** file:

```
<link rel="stylesheet"  
      href="//maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
```

```
alpha.6/css/bootstrap.min.css" />
```

Another thing to notice is the ***ngIf** directive on the **tr** tag. It is a way for Angular to conditionally include a template, or not. In this case, we are using it to hide the **loginError** table row when there is no login error, and show it when there is.

We are also going to include individual errors to make sure that the user filled out both the username and password:

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

The **userNameError** variable is something we'll define shortly, inside the login component.

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** soon. 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 **styles** and put a **styles.css** file in there. The file contains one simple CSS class:

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

This style will automatically be processed by our Gulp build scripts and turned into a file named **app.min.css**. This should already be loaded in the main **index.html** page. If not, you can add this here:

```
<link rel="stylesheet" href="app.min.css">
```

Now open the **login.component.ts** file to edit the login component's class. First, you need to create the three variables to contain the error:

```
usernameError = '';  
passwordError = '';  
loginError = '';
```

I placed them right under the definition of the username and password variables. The login method will require use of our providers, so we need to

make sure they are available to the login component.

Create a constructor:

```
constructor (  
  private authenticationService: AuthenticationService,  
  private router: Router,  
  private userModel :UserModel  
) {}
```

The constructor defines local private component variables that Angular will automatically inject into the component. The **authenticationService** and **userModel** variables point to shared data sources created as providers on the **@NgModule**. The router is just an instance of the **Router** variable which can will be used to redirect the user to the tasks view after successful sign in.

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:

```
onLogin(): void {  
}
```

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

```
let errorFound : boolean = false;
```

The default value of the **errorFound** is “false”. If we find errors, the **errorFound** variable will change to “true”. Then, default the **loginError**:

```
this.loginError = '';
```

This will reset the **loginError** variable to an empty string and hide the display if it is already displayed.

First, check to make sure that a username was entered. Simple conditional logic is used:

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



```
    this.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 be blank. This will effectively remove the error from the user’s display.

Error checking for the password acts similarly:

```
if ( this.password === '' ) {  
    this.passwordError = 'You Must Enter a Password';  
    errorFound = true;  
} else {  
    this.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. This keyword 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:

Login View

Username	<input type="text"/>	You Must Enter a Username
Password	<input type="password"/>	You Must Enter a Password
<input type="button" value="Reset"/>	<input type="button" value="Login"/>	

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

```
this.authenticationService.authenticate( this.username,
this.password )
  .subscribe(
    result => {
      // result code
    }, error => {
      // error code
    }
  )
)
```

This calls the **authenticate()** method in the service. It passes in the username and password. The result is an **Observable** object. The **subscribe()** function is called on the observable and it contains one function that will execute upon success, and one for failure. Both functions are created with the TypeScript lambda operator and the arguments are passed into the subscribe function.

This is the success code:

```
if ( result.error) {
  this.loginError = 'We could not log you in';
  return;
}
this.userModel.user = result.resultObject as UserVO;
this.router.navigate(['tasks']);
```

This code checks for the error property on the result. If it is “true”, then it displays an error to the user using the **loginError** variable. If it is “false”, then we had a successful login. Save the user object the **userModel**, and use the router to redirect to the tasks route.

This is the error function will execute upon promise rejection:

```
this.loginError = 'There was an authentication service error';
```

For simplicity, it just sets the **loginError** value, which in turn will activate the UI table row and will display the error to the user. Here is an example of the **loginError** displayed:

Login View

We could not log you in

Username	<input type="text" value="asdf"/>
Password	<input type="password" value="...."/>
<input type="button" value="Reset"/>	<input type="button" value="Login"/>

The service integration code is setup so that the only way a promise rejection would occur is if there was a network or other unexpected error. Even a failed login will return a successful result. I 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.

Final Thoughts

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

1. The main HTML page loads and all the JavaScript files that represent Angular dependencies are loaded; Shim.js, Zone.js, Reflect-Metadata, and SystemJS.
2. Our custom SystemJS config is loaded. It is an Immediately-Invoked Function Expression, so it will run immediately. It tells SystemJS where to find all the Angular libraries and our custom application.
3. The `System.import('app')` code is run inside the **index.html**. This loads our main Angular file.
4. The main Angular file bootstraps the app module, which loads the relevant application components and classes.
5. Angular introspects the HTML page, finds **lw-app** and replaces it with the appropriate Angular view.
6. The view contains a router, so the router initializes and changes the URL to load the default login route.
7. The login component template is loaded, waiting for user input.
8. User enters credentials and clicks the login button. This triggers a service call to the authentication service. The authentication service returns a promise object to the component, and the component waits for it to resolve or reject.
9. If the promise object resolves, the result is processed and the user is logged in and redirected to the main tasks view, or shown an error.
10. If the promise object is rejected, the user is shown an error.

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. An open source Angular grid library—**ngx-datatable**—will be used for display. We will also review the services needed to retrieve the data, and wire up everything to display. 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

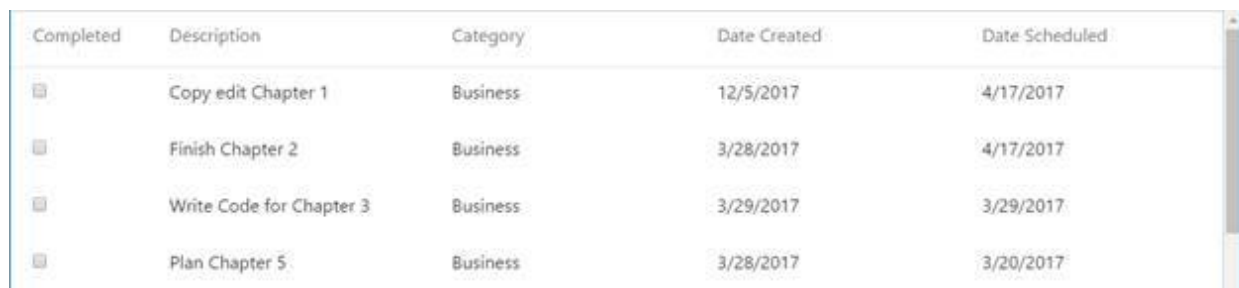
This section will show you how to create the grid within the Angular application. It will review the data to display in the grid, and show you how to customize a column's display.

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:

- **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.
- **Description:** This column will contain the details of the task.
- **Category:** Tasks can be categorized and the grid will display each task category to the user.
- **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 show you how to build the interface for scheduling tasks.

The final grid will look like this:



Completed	Description	Category	Date Created	Date Scheduled
<input type="checkbox"/>	Copy edit Chapter 1	Business	12/5/2017	4/17/2017
<input type="checkbox"/>	Finish Chapter 2	Business	3/28/2017	4/17/2017
<input type="checkbox"/>	Write Code for Chapter 3	Business	3/29/2017	3/29/2017
<input type="checkbox"/>	Plan Chapter 5	Business	3/28/2017	3/20/2017

Setup the Grid

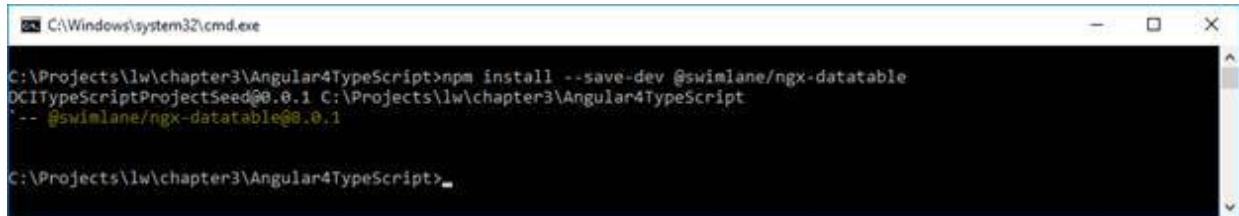
First order of business is to configure out app to load the [ngx-datatable](#). To do this, first we'll have to install the libraries via NodeJS. Then, we'll have to

modify our Gulp scripts to copy over the related assets. After that's done, we'll have to modify our SystemJS config so it knows how to load the **ngx-datatable** component.

First, install the grid library by running this Node command on the console:

```
npm install --save-dev @swimlane/ngx-datatable
```

You'll see something like this:



```
C:\Windows\system32\cmd.exe
C:\Projects\lw\chapter3\Angular4TypeScript>npm install --save-dev @swimlane/ngx-datatable
DCITypeScriptProjectSeed@0.0.1 C:\Projects\lw\chapter3\Angular4TypeScript
'-- @swimlane/ngx-datatable@0.0.1
C:\Projects\lw\chapter3\Angular4TypeScript>
```

Node will automatically download the latest version of the library and update the **package.json** which tells Node what libraries it needs.

Next, modify the Gulp scripts to copy the relevant CSS and JavaScript behind the grid. Open the **config.js** file in the root directory of the project. The **ngx-datatable** was written in TypeScript, but includes the compiled JavaScript version of it in their release build. We just need to copy that. Find the **angularLibraries** variable in the **config.js** file. It should look like this:

```
angularLibraries : [
  'core-js/client/shim.min.js',
  'zone.js/dist/**',
  'reflect-metadata/Reflect.js',
  'systemjs/dist/system.src.js',
  '@angular/**/bundles/**',
  'rxjs/**/*.*js'
],
```

This array is a bunch of [globs](#). A glob is a pattern that specifies how to find files. The script uses this variable to copy the relevant libraries from the **node_modules** directory to the **build/js** directory. The **node_modules** is not specified, because it is assumed in the copy task. We just need to add the **ngx-datatable** library at the end of this array:

```
'@swimlane/ngx-datatable/release/index.js'
```

Next, we need to handle the **cssSource**. This is what it looks like in its current state:

```
cssSource : [baseDirs.sourceRoot + '**/*.css',
             '! ' + baseDirs.sourceRoot + baseDirs.codeRoot +
             '/**/*.css'
            ],
```

The variable looks for all CSS in the source root—**src**—but avoids anything that is in the code root—**com**. The build script assumes anything in the **com** directory will be part of an Angular component, and does not need to be processed into the app's primary CSS file; Angular will load it when needed. Two additional globs to this array to accommodate for the **ngx-datatable** specific styles:

```
'node_modules/@swimlane/ngx-datatable/release/**/*.css',
'!node_modules/@swimlane/ngx-datatable/release/**/*.app.css'
```

The first glob looks for all the all the CSS files in the **ngx-datatable** release directory. The second tells to ignore the **app.css** file in the same directory. At the time of this writing, the **ngx-datatable** release directory contains a style sheet for the demo app of the component instead of styles just for the component itself.

The **ngx-datatable** CSS has some references to external font files, so they will need to be copied from the **node_modules** directory to the build directory also. The gulp script isn't set up to handle font files by default, so we'll create a new task strictly for the grid assets. Open the **gulpfile.js** in your project's root directory and add this:

```
gulp.task('copyGridAssets', function () {
  return gulp.src('node_modules/@swimlane/ngx-
  datatable/release/assets/fonts/*.*)
    .pipe(gulp.dest(config.destinationPath + '/fonts'));
});
```

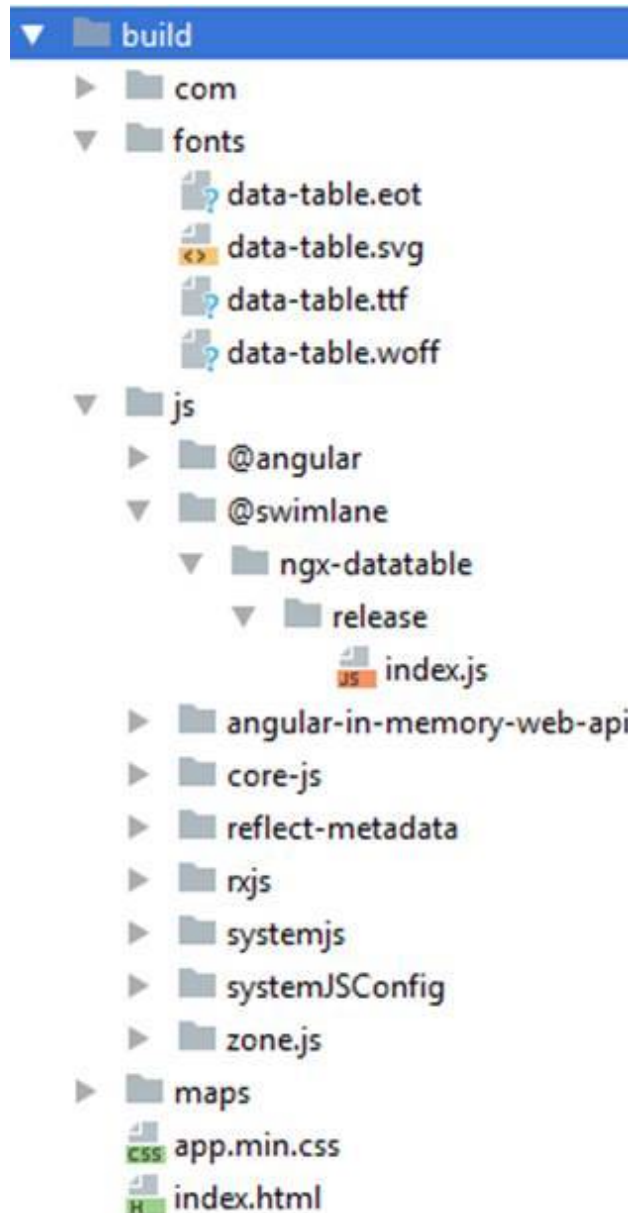
Add this new **copyGridAssets** task to the build script:

```
gulp.task("build", ['buildTS', 'copyJSLibraries',
'copyAngularLibraries',
                 'copyHTML', 'buildCSS', 'copyStaticAssets',
                 'copyGridAssets']);
```

Now, run your gulp build script. It should copy over the new, relevant files.


```
C:\Windows\system32\cmd.exe
C:\Projects\lw\chapter3\Angular4TypeScript>gulp cleanBuild
[11:35:09] Using gulpfile C:\Projects\lw\chapter3\Angular4TypeScript\gulpfile.js
[11:35:09] Starting 'cleanBuild'...
[11:35:09] Starting 'clean'...
[11:35:09] Finished 'clean' after 6.12 ms
[11:35:09] Finished 'clean' after 236 ms
[11:35:09] Starting 'tslint'...
[11:35:09] Starting 'copyJSLibraries'...
[11:35:09] Finished 'copyJSLibraries' after 3.91 ms
[11:35:09] Starting 'copyAngularLibraries'...
[11:35:09] Finished 'copyAngularLibraries' after 4.33 ms
[11:35:09] Starting 'copyHTML'...
[11:35:09] Starting 'copyAngularCSS'...
[11:35:09] Finished 'copyAngularCSS' after 4.96 ms
[11:35:09] Starting 'processCSS'...
[11:35:09] Finished 'processCSS' after 1.0 ms
[11:35:09] Starting 'buildCSS'...
[11:35:09] Finished 'buildCSS' after 1.78 ms
[11:35:09] Starting 'copyStaticAssets'...
[11:35:09] Starting 'copyGridAssets'...
[11:35:09] Finished 'copyStaticAssets' after 127 ms
[11:35:14] Finished 'copyGridAssets' after 4.67 s
[11:35:14] Finished 'copyHTML' after 4.73 s
[11:35:14] Finished 'tslint' after 4.76 s
[11:35:14] Starting 'buildTS'...
[11:35:16] Finished 'buildTS' after 2.35 s
[11:35:16] Starting 'build'...
[11:35:16] Finished 'build' after 3.85 ms
C:\Projects\lw\chapter3\Angular4TypeScript>
```

Check out the build directory structure:



You'll see the **fonts** directory was successfully copied over, as was the **@swimlane/ngx-datatable/release/** JavaScript file. The **ngx-datatable's** CSS was added to **app.min.css**. With the files handled, the component is ready. Next, we'll tell Angular how to find it.

[Tell Angular how to find the Grid Component](#)

There are two code changes required to tell Angular about this new component. The first is to configure SystemJS about the component. The second is to load the component into the application's main module.

Open the **systemjs.config.js** file in the **js/systemJSConfig/** directory. Look for the **System.config** command. It creates accepts object defining SystemJS's config. Look for the **map** property. This tells SystemJS how to find all the Angular libraries. We just need to add the new library now:

```
'@swimlane/ngx-datatable' : 'js:@swimlane/ngx-datatable/release/index.js'
```

This is an important step when adding new libraries, but an easy one to forget.

Open the **app.module.ts** file and import the **ngx-datatable**:

```
import { NgModule } from '@swimlane/ngx-datatable';
```

Then, add it to the import array in the **@NgModule** annotation:

```
imports: [
  BrowserModule,
  AppRoutingModule,
  FormsModule,
  NgModule
],
```

That completes the basic install.

Creating a TaskModel and Other Value Objects

We are going to create a **TaskModel** for this app. It will be a class that contains task related data which can be easily shared across multiple components. Create a **taskmodel.ts** file in the **com/dotComIt/learnWith/model** directory.

Start by defining the class:

```
export class TaskModel {
};
```

For the moment, the only property we are going to add to the **TaskModel** is an array of tasks. This will be used by the grid to display the tasks to the user. Before we do that, we need a task object. Create a file named **TaskVO** in **com/dotComIt/learnWith/vo**:

```
export class TaskVO {
  completed : boolean;
  dateCompleted : Date;
  dateCreated : Date;
```

```
dateScheduled : Date;
description : string;
taskCategory : string;
taskCategoryID : number;
taskID : number;
userID : number;
};
```

This class contains all the properties we discussed in relation to the task grid earlier; category, completed state, date completed, date scheduled, and description. It also contains the date created, the primary ID for the task—**taskID**—the primary ID for the user—**userID**—and the primary key for the category—**taskCategoryID**. There is no advanced functionality in the value object class; it is just a data container.

Import the **TaskVO** into the **TaskModel**:

```
import {TaskVO} from "../vo/TaskVO";
```

And create the tasks array as a property on the **TaskModel**:

```
tasks : TaskVO[];
```

Since this is a model class to be used as part of Angular's dependency injection process, let's make it injectable with the annotation. First, import the **Injectable** class:

```
import {Injectable} from "@angular/core";
```

And add the **@Injectable()** annotation before the class definition:

```
@Injectable()
export class TaskModel {
```

It is considered good practice to set up all our provider classes with the **@Injectable()** annotation.

To make use of the **TaskModel** in the Angular application, you need to set it up as a provider in the **@NgModule** annotation from the **app.module.ts** file. First, import the **TaskModel** class:

```
import {TaskModel} from "../model/taskmodel";
```

Then, add it to the providers array:

```
providers : [
  {provide: LocationStrategy, useClass:HashLocationStrategy},
```

```
AuthenticationService,  
UserModel,  
TaskModel  
],
```

Now we can use the **TaskModel** across multiple views.

Create a Grid Component

Let's create a new component to display the task grid. In the **com/dotComIt/learnWith/views/tasks** folder, create a file named **taskgrid.component.ts**. Import the Angular component library and the **TaskModel**:

```
import {Component} from '@angular/core';  
import {TaskModel} from '../../model/taskmodel';
```

Next, create the **@Component** annotation:

```
@Component({  
  selector: 'taskgrid',  
  templateUrl :  
  './com/dotComIt/learnWith/views/tasks/taskgrid.component.html',  
  styleUrls: [  
  './com/dotComIt/learnWith/views/tasks/taskgrid.component.css']  
})
```

The component's name is **taskgrid**. This is the HTML tag we'll use in our views to display the grid. A **templateUrl** and **styleUrls** are both specified. Create two empty files in the same view directory; **taskgrid.component.html** and **taskgrid.component.css**.

Next, create the class:

```
export class TaskGrid {  
  constructor(private taskModel :TaskModel) {  
  }  
}
```

The class is named **TaskGrid**. For now, I just created a constructor, which includes an instance of the **TaskModel**.

Let's add an array of tasks now:

```
public tasks : TaskVO[];
```

I defined this below the constructor, not as part of it. With the new class, also add the import:

```
import {TaskVO} from "../../vo/TaskVO";
```

The tasks array will be used to hold the grid's data.

Be sure to load the **TaskGrid** component in the **app.module.ts** file. First, import it:

```
import {TaskGrid} from "../views/tasks/taskgrid.component";
```

Then, add it to the **@NgModule**'s declarations array:

```
declarations: [  
  AppComponent,  
  LoginComponent,  
  TasksComponent,  
  TaskGrid  
],
```

After that, open the **tasks.component.html** in the **com/dotComIt/learnWith/views/tasks** directory. Delete all the contents of the file and replace it with:

```
<taskgrid></taskgrid>
```

If you rebuild the app and load it in a browser, the **TaskGrid** component should display to the screen. Because it is currently an empty template, you won't see anything. Next, we can create the Grid.

Create the DataGrid

Open up the **taskgrid.component.html** in the **com/dotComIt/learnWith/views/tasks** directory. Create an empty **div**:

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

Then, create the **ngx-datatable**:

```
<ngx-datatable #taskGrid  
  class="material gridStyle"  
  [rows]="tasks"  
  [headerHeight]="50"  
  [rowHeight]="50"  
  [columnMode]='force' "
```

```
>  
</ngx-datatable>
```

The first thing you'll notice is the **#taskgrid** property. This is not an HTML property; it is a way for the Angular class to get a hook into grid. We won't need that for this chapter, but it will come into play in chapter 6 when we create the scheduler component.

Next, two CSS classes are specified. The first is **material**, which is part of the **ngx-datatable** styles. The second is **gridStyle**, and that is something we need to create. Open the **taskgrid.component.css** file and add the styles:

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

This tells the grid to stretch the height and width of its container. It also adds a border around the grid.

Back to the main grid properties. You'll notice that all remaining properties contain square brackets around them. This is Angular notation for data binding to a property, so these properties are custom properties implemented as part of the **ngx-datatable** component. The **rows** property is set to the **tasks** array from the **taskgrid** component. The **headerHeight** and **rowHeight** are both hard coded to 50 pixels high. The **columnMode** is set to **force**, which means the columns will distribute evenly across the grid.

We're going to define the columns as part of the HTML file. Start with the description:

```
<ngx-datatable-column name="Description" >  
</ngx-datatable-column>
```

This uses the **ngx-datatable-column** component to create the description column. The component will introspect the **tasks** array to find this column, and the word "Description" will display in the header. If we want to specify different header text than the property, we can do that too:

```
<ngx-datatable-column name="Category" prop="taskCategory" >  
</ngx-datatable-column>
```

The name property will show up in the header, while the contents of the column will show the **taskCategory** property of the row's object.

Next, create the completed column. This one requires a custom template:

```
<ngx-datatable-column name="Completed" >
  <ng-template let-value="value" ngx-datatable-cell-template>
    <input type="checkbox" [checked]="value" />
  </ng-template>
</ngx-datatable-column>
```

The **ngx-datatable-column** definition is the same as what we used for previous columns. The header's name uses the name property for display, and will introspect the **TaskVO** objects to find the completed property. The body of the **ngx-datatable-column** is no longer empty. It includes an **ng-template** directive, along with an **ngx-datatable-cell-template** distinguisher. The **let-value** property tells Angular to pass the value for the column—completed—into the template with the name "value". Inside the template is an input checkbox. The checked property is set to "value". With this done, when we load the tasks, the completed property will automatically check or uncheck based on value in the object. I put completed as the first column.

Now, create a column for the **dateCreated** value:

```
<ngx-datatable-column name="Date Created" prop="dateCreated" >
  <ng-template let-value="value" ngx-datatable-cell-template>
    {{value | date: 'shortDate'}}
  </ng-template>
</ngx-datatable-column>
```

This uses another template. In this case, we are using an Angular pipe to format the date. An Angular pipe is analogous to a filter in AngularJS. We display the data column followed by a vertical pipe character. Then the filter information. In this case, the [date pipe](#) is used, along with the default "shortDate" filter. This will format the date to the common US date format; month/day/year. Angular does have facilities for us to create our own pipes. However, it is not needed in this case.

Finally, create the **dateScheduled** column:

```
<ngx-datatable-column name="Date Scheduled" prop="dateScheduled" >
  <ng-template let-value="value" ngx-datatable-cell-template>
    {{value | date: 'shortDate'}}
  </ng-template>
</ngx-datatable-column>
```



```
</ng-template>  
</ngx-datatable-column>
```

This mirrors the **dateCreated** column. For now, that completes our grid.

Creating a TaskFilter Object

Before jumping into the services, I want to create a value object. The service method we are going to call is a **loadTasks()** method. It will retrieve tasks based on specific filter criteria. In our initial load of tasks, we only care about two values in the filter:

- **startDate**: The start date will filter against the date that an item was created. For the app's initial load, it will be hard-coded to March 1, 2017 because that was the date the database initially used when creating a bunch of data for this app. In a real-world app, we'd probably default it to today's date.
- **completed**: The completed property will be used to allow users the ability to see tasks that are either completed, or those that remain to be done. When loading the default, it is set to "0" to retrieve all the incomplete tasks.

There are other values that may be contained in the **taskFilter** class, however these are the only two needed for this chapter. We want to contain these values in a UI object.

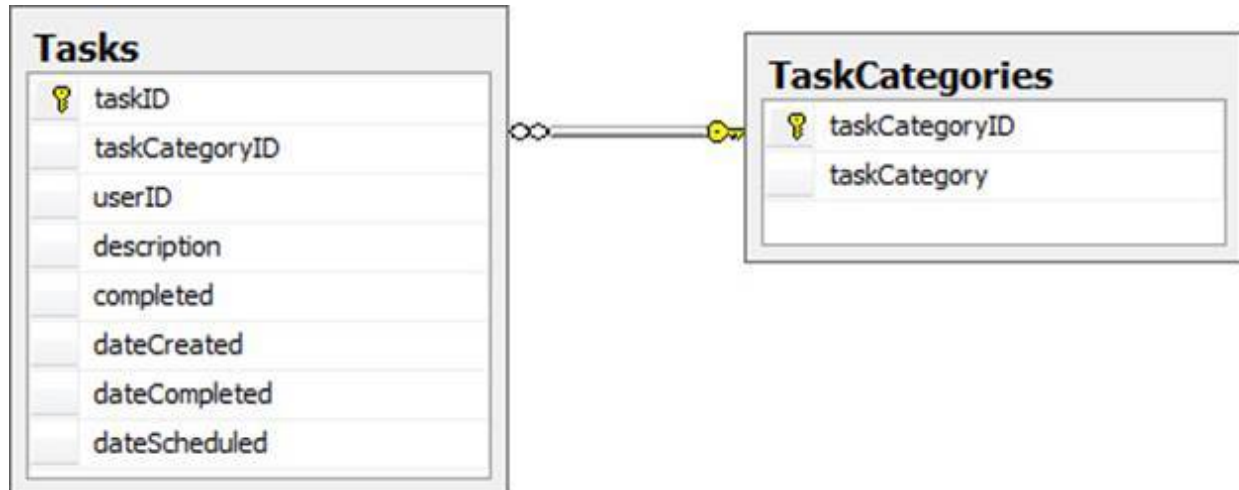
Create a new file named **TaskFilterVO.ts** in the **com/dotComIt/learnWith/vo** directory:

```
export class TaskFilterVO {  
    completed : boolean;  
    startDate : Date;  
};
```

We'll create an instance of this class to pass into the service that performs the filtering.

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

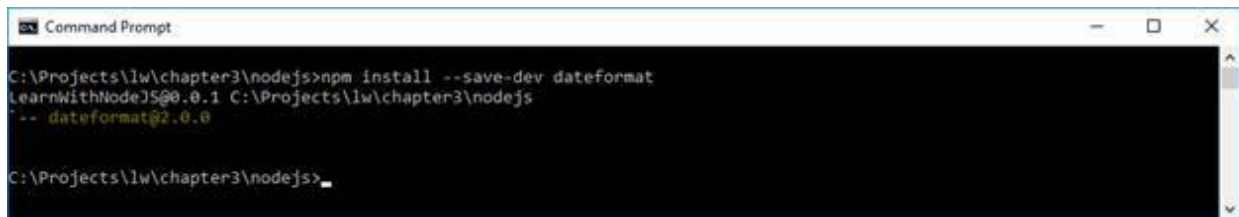
This section will cover the NodeJS services needed to load and filter tasks. It will teach you how to create a new NodeJS service, and use a new NodeJS module for date formatting purposes.

Install a DateFormatter

The first step is to install a NodeJS formatter module. The purpose of this is to format date values before sending them back to the UI. Instead of writing my own format function, I decided to look for a NodeJS package that can do the same. I found one, [node-dateformat](#). You can install it by running this on your command line:

```
npm install --save-dev dateformat
```

You'll see the install like this:



```
Command Prompt
C:\Projects\lw\chapter3\nodejs>npm install --save-dev dateformat
LearnWithNodeJS@0.0.1 C:\Projects\lw\chapter3\nodejs
├-- dateformat@2.0.0

C:\Projects\lw\chapter3\nodejs>
```

Now you should be good to go to import it into a NodeJS module, like this:

```
var dateFormatter = require('dateformat');
```

Then you can use it to format dates in your NodeJS code, like this:

```
dateFormatter(new Date(), "mm/dd/yyyy");
```

Create the Task Service

The **TaskService** module that we will create will be the basis for most of our task related services. This includes loading tasks, editing tasks, and marking tasks as “completed”. In this section, we will focus on loading tasks. First create the file—**TaskService.js**—in the **com/dotComIt/learnWith/services** directory.

The file starts with some imports:

```
var dateFormatter = require("dateformat");
var databaseConnection =
require("../database/DatabaseConnection");
var responseHandler = require("../server/JSONResponseHandler");
```

This makes the global **dateformat** package available for use. It also loads our local **DatabaseConnection** and **JSONResponseHandler** modules from the last chapter.

Next create the function signature for loading the tasks:

```
function getFilteredTasks(response, queryString) {  
}
```

This method uses the same signature as previous handlers we have created. It accepts the response object and a **queryString** object.

Inside the **getFilteredTasks()** method, we are going to see some common boilerplate:

```
var resultObject = {};  
var callback = '';  
if(queryString.callback !== undefined){  
    callback = queryString.callback;  
}
```

This creates a generic **resultObject** that will contain the results we need to send back to the application. It also saves the **callback** function name, if defined. We used similar code in the previous chapter when creating the authentication service.

Next comes some **error** checking. The required argument into this service call will be a JSON object named **filter**. Make sure that the **filter** parameter is defined:

```
if((queryString.filter === undefined)){  
    resultObject.error = 1;  
    responseHandler.execute(response, resultObject, callback);  
} else {  
    // Process the query; we'll do that next  
}
```

If the **filter** argument is not defined, add the **error** value on the **resultObject** and return results to the client.

If the **filter** argument is defined, we need to create the query and process it. The **filter** object will come in as a JSON string, so start by turning it into a native JavaScript object:

```
var json = JSON.parse( queryString.filter );
```

The **JSON.parse()** function will turn a JSON string into a JavaScript object. Next, we'll start the query creation. First, step up a variable named **firstOne**:

```
var firstOne = true;
```

The database query we are going to create has a lot of conditions. The **firstOne** variable will be used to determine if we need to add the "where" clause—for the first condition—or an "and" clause to append conditions.

Start the query:

```
var query = "select tasks.*, taskCategories.taskCategory  
            from tasks left outer join taskCategories on  
            (tasks.taskCategoryID =  
taskCategories.taskCategoryID)";
```

This is the base query, and it will return all tasks. However, we need to be able to limit the number of returned tasks based on the user input. For the purposes of this section, we'll implement the **completed** property to return completed tasks, and the **start date** to return tasks created after a specific date.

First, check for the **completed** property:

```
if(json.completed != undefined) {  
    if(firstOne) {  
        query = query + "Where "  
        firstOne = false;  
    } else {  
        query = query + "and "  
    }  
    if(json.completed == true) {  
        query = query + " completed = 1 ";  
    } else {  
        query = query + " completed = 0 ";  
    }  
}
```

If the **firstOne** variable is "true", then add a "where" clause to the query, and set the **firstOne** value to "false". Otherwise, an "and" is added to concatenate the new condition with the previous conditions. The **completed** property is a Boolean property, or a bit property in the database. As such, it always uses a "1" or "0" for the comparison no matter what the value for the **completed** argument.

Next, we want to use the **startDate** property to compare against the **dateCreated** database value:

```
if(json.startDate !== undefined) {
  if(firstOne) {
    query = query + "Where "
    firstOne = false;
  } else {
    query = query + "and "
  }
  query = query + " dateCreated >= '" + json.startDate + "' ";
}
```

This assumes that the dates will already be properly formatted to be recognized as dates by the SQL query, so we do not do any processing on the dates here. The **startDate** must be before, or equal to the database's **dateCreated** date.

The final aspect of the query is to sort the results:

```
query = query + " order by dateCreated ";
```

Next, we can use the **databaseConnection** object to execute the query against the database:

```
var dataQuery = databaseConnection.executeQuery(query,
  function(result) {
    // Process Query Results here
  },
  function(err) {
    // Process Error Results here
  }
)
```

If the result handler is executed, then we get back a successful query result and want to send a successful response to the browser:

```
responseObject.error = 0;
```

This sets the error value of the **responseObject** to "0", meaning we had a successful query. Next, you'll want to format the dates in the result set, so loop over the returned **recordset**:

```
for (var x=0;x < result.recordset.length ; x++ ){
  result.recordset[x].dateCreated =
  dateFormatter(result.recordset[x].dateCreated, "mm/dd/yyyy");
}
```

```

    if(result.recordset[x].dateCompleted) {
        result.recordset[x].dateCompleted =
dateFormatter(result.recordset[x].dateCompleted, "mm/dd/yyyy");
    } else {
        result.recordset[x].dateCompleted = '';
    }
    if(result.recordset[x].dateScheduled) {
        result.recordset[x].dateScheduled.setDate(
result.recordset[x].dateScheduled.getDate() + 1);
        result.recordset[x].dateScheduled =dateFormatter(
result.recordset[x].dateScheduled, "mm/dd/yyyy");
    } else {
        result.recordset[x].dateScheduled = '';
    }
}

```

Inside the query loop, each “date” column is replaced with a formatted version of itself, removing the time stamp. The **dateFormatter** module is used to format the date. For the “dateCompleted” and “dateScheduled” columns, we check to make sure that a value exists before trying to format it. If the value doesn’t exist, it is set to an empty string. With the **dateScheduled** property, I ran into some time zone related issues which would set the formatted date a day behind the one specified in the UI. As an ad-hoc solution, I added a single day to the **dateScheduled** before formatting it. I felt this was the best solution, as I didn’t want to make changes and rewrites to existing code in the UI.

Next, add the final **recordset** to the **responseObject**:

```
responseObject.resultObject = result.recordset;
```

Finally, call the **responseHandler** to convert the **responseObject** into JSON and pass it back to the user:

```
responseHandler.execute(response, responseObject, callback);
```

This is the error handler function:

```
responseObject.error = 1;
responseHandler.execute(response, responseObject, callback);
```


The error handler only executes if something bad happened with the query. An error response is returned to the client.

That completes the **getFilteredTasks()** method in the **TaskService**. We will expand on this method in future chapters as more filtering criteria is added. However, we still need to export this method for use by other modules:

```
exports.getFilteredTasks = getFilteredTasks;
```

Next, open the **ResponseHandlers** file from the **com/dotComIt/learnWith/server** directory. Import the **taskService** method:

```
var taskService = require("../services/TaskService");
```

And add the **getFilteredTasks()** method into the **handlers** object:

```
handlers["/taskService/getFilteredTasks"] =  
taskService.getFilteredTasks;
```

Testing the getFilteredTasks() Service

At this point, you can restart your NodeJS server and the new **TaskService** should be ready to accept and respond to **getFilteredTasks** requests. Try a few out by loading things in the browser. This URL should load all the tasks which are not completed:

```
http://127.0.0.1:8080/taskService/getFilteredTasks?  
method=getFilteredTasks  
&filter={"completed":"0"}
```

You should see results similar to this:

```
{  
  "responseObject": [  
    {"taskcategoryID":2,  
     "description":"Get Milk",  
     "taskcategory":"Personal",  
     "dateScheduled":"03\29\2016",  
     "dateCompleted":"","  
     "taskID":1,  
     "dateCreated":"03\27\2016",  
     "completed":0,  
     "userID":1  
    },  
    {"taskcategoryID":1,  
     "description":
```

```

    "Finish Chapter 2",
    "taskcategory":"Business",
    "dateScheduled":"03\29\2016",
    "dateCompleted":"","
    "taskID":2,
    "dateCreated":"03\28\2016",
    "completed":0,
    "userID":1
  },
  {
    "taskcategoryID":1,
    "description":"Plan Chapter 5",
    "dateScheduled":"03\20\2016",
    "taskcategory":"Business",
    "dateCompleted":"","
    "taskID":5,
    "dateCreated":"03\28\2016",
    "completed":0,
    "userID":1
  }
],
"error":0.0
}

```

This URL should show you tasks created after March 29, 2017:

```

http://127.0.0.1:8080/taskService/getFilteredTasks?
method=getFilteredTasks
&filter={"startDate":"3/29/2017"}

```

It should show results similar to this:

```

{
  "responseObject": [
    {
      "taskcategoryID":1,
      "description":"Write Code for Chapter 3",
      "dateScheduled":"03\29\2017",
      "taskcategory":"Business",
      "dateCompleted":"","
      "taskID":3,
      "dateCreated":"03\29\2017",
      "completed":0,
      "userID":1
    },
    {
      "taskcategoryID":1,
      "description":"Write Chapter 4",
      "taskcategory":"Business",
      "dateScheduled":"03\20\2017",
      "dateCompleted":"","
      "taskID":4,

```

```
"dateCreated": "03\30\2017",
"completed": 0,
"userID": 1
},
{
  "taskcategoryID": 1,
  "description": "Learn JQuery",
  "taskcategory": "Business",
  "dateScheduled": "",
  "dateCompleted": "",
  "taskID": 6,
  "dateCreated": "03\31\2017",
  "completed": 0,
  "userID": 1
}
],
"error": 0.0
}
```

You can tweak the filter parameter to experiment with different values and their return sets.

Create the TaskService Stub

First, you'll want to create a stub class for the **TaskService**. Create the file **TaskService.ts** in the **com/dotComIt/learnWith/services/nodejs** directory, and add some imports:

```
import {Injectable} from "@angular/core";  
import {Jsonp } from "@angular/http";
```

This imports the **Injectable** class so we can have Angular inject providers into this class.

Add a constant for the server location:

```
const SERVER : string = 'http://127.0.0.1:8080/';
```

We'll use this to tell our service calls how to find the NodeJS source directory relative to the current Angular application.

Now, create the class stub:

```
@Injectable()  
export class TaskService {  
}
```

This uses the **@Injectable()** annotation and defines the class name.

Next, create the constructor:

```
constructor(private jsonp: Jsonp) {  
}
```

The constructor has an instance of **Jsonp** service as an argument. This is already created as provider in the Angular module definition.

Now open up the **app.module.ts** file in the **com/dotComIt/learnWith/main** directory. Import the **TaskService**:

```
import {TaskService} from "../services/nodejs/task.service";
```

Now, add both classes as providers as part of the **@NgModule** annotation:

```
providers : [  
  {provide: LocationStrategy, useClass:HashLocationStrategy},  
  UserModel,  
  AuthenticationService,
```

```
TaskModel,  
TaskService  
],
```

This will allow the **TaskService** class to be used within the application. However, we still need to implement the **loadTasks()** method and some conversion functions for turning our objects into HTTP friendly strings.

Turning the 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, we created a special string to pass values from the Angular code to the NodeJS Server using JSONP. We did this for the authentication service integration. For task service integration, we are going to have to take it a step further and perform a translation of our object arguments into something that can be easily sent to NodeJS. We're going to create a function to convert a TypeScript object into a JSON String.

Create a new class file named **httpUtils.ts** in the **com/dotComIt/learnWith/services/nodejs** directory. Start with two imports:

```
import {DatePipe} from "@angular/common";  
import {isObject} from "rxjs/util/isObject";
```

This imports the **DatePipe**, which will be used for date formatting, and an **isObject()** function which can be used to determine if a variable contains an object. Create the class stub:

```
export class HttpUtils {  
}
```

The **objToJSONString()** function will accept an object and turn it into a JSON string. The service is expecting a string which represents a **TaskFilterVO** object in JSON form. This is the **objToJSONString()** signature:

```
static objToJSONString (obj : any) :string {  
}
```

It accepts a single argument—"obj"—which is the object that needs to be converted into a JSON string. The first line of the method creates an instance of the **DatePipe**:

```
let datePipe : DatePipe = new DatePipe('en-US');
```

I'm using an Angular functionality in a class which otherwise has no Angular dependencies. The argument into the **DatePipe** constructor is the locale. In this case I set it to "en-US". Now, create a string for the results:

```
let str = '';
```

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

```
for (var p in obj) {  
  if (obj.hasOwnProperty(p)) {  
  }  
}
```

The first thing this does is check that the **obj** has the property “p”. This check is done for performance reasons, because a for-in loop will include all values or functions in the object’s hierarchy. We only want to check properties on the current object.

First, check if the **obj[p]** value is numeric:

```
if (isNumeric(obj[p])) {  
  str += "\"" + p + "\":\"" + obj[p] + "\",\"";  
}
```

This uses the rxjs **isNumeric()** function, so be sure to import it:

```
import {isNumeric} from "rxjs/util/isNumeric";
```

The **isNumeric()** check is performed because if the value is a number, we want to be sure it is included in the final result. If that number is “0”—which is entirely possible given some of our drop-down sources—a Boolean check or is-defined check will return “false”. We want to avoid that trap. This check is there primarily because the “all categories” item in the task drop-down may be “0”.

Next, check if the value is Boolean:

```
else if (typeof(obj[p]) === "boolean") {  
  str += "\"" + p + "\":\"" + obj[p] + "\",\"";  
}
```

There isn’t a handy rxjs library function for checking if a value is a Boolean, so I fell back on using the JavaScript **typeof** check. This is there to check the value of the completed drop-down. If it is “false”; it will fail a “does this value exist” check. A “false” Boolean value does exist.

Now, check if the value exists:

```
} else if (obj[p]) {  
  if (obj[p] instanceof Date) {
```

```

    str += "\"" + p + "\":" + datePipe.transform(obj[p], 'yyyy-
MM-dd') +
        "\", ";
  } else {
    str += "\"" + p + "\":" + obj[p] + "\", ";
  }
}

```

If it does, dig deeper. If the **obj[p]** is a date object, we want to use the **datePipe** to convert it to a date string, removing any time stamps. If the **obj[p]** is not a date object, then translate it into a string with no additional processing.

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.

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 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 import this class into the **task.service.ts**:

```

import {HttpUtils} from "../httpUtils";

```


So we can access the **transformRequest()** function and **objToJSONString()** functions when creating our service call.

Accessing the `loadTask()` Service

We're going to need some more imports to implement the **loadTask()** method:

```
import {Observable} from "rxjs/Observable";
import {ResultObjectVO} from "../../vo/ResultObjectVO";
import 'rxjs/add/operator/map';
import {TaskFilterVO} from "../../vo/TaskFilterVO";
```

Here, we import the **Observable** class, and the **ResultObjectVO** class. Both will be part of the return type from the **loadTasks()** method. The **map()** function is brought in from the rxjs library to process the results. Finally, the **TaskFilterVO** is imported. It will be an argument to the **loadTasks()** method.

Here is the method:

```
loadTasks(taskFilter : TaskFilterVO) : Observable<ResultObjectVO> {
    let parameters : string = "filter" + '=' +
        HttpUtils.objToJSONString(taskFilter) + '&';
    parameters += "callback" + "=" + "JSONP_CALLBACK";
    let url = SERVER + 'taskService/getFilteredTasks?' +
parameters;
    return this.jsonp.request(url)
        .map((result) => result.json() as ResultObjectVO);
}
```

The first step in the function is to create a parameter string with the **filter** and **callback** arguments. The **objToJSONString()** method turns the **taskFilter** into a URL friendly object. The **callback** argument is not needed when accessing files from the same domain. It is only needed when accessing services from a remote domain. The value of the **callback** argument is **JSONP_CALLBACK**, which is a built-in Angular value.

Then, the **url** string value is created using the **SERVER** constant, the service **endpoint**, and the parameters. After that, the **jsonp.request()** function is called, and its **Observable** is returned. A **map()** is called on the observable to translate results from JSON into a **ResultObjectVO** before the subscriber's result method is called.

Wire Up the UI

This section will show you how to load the task data from the service and display it in the **ngx-datatable**. It will also 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 or not the user is logged in, you can check to see if the **user** property of the **UserModel** has been initialized. If not, redirect the user to the login screen. If so, the user has logged in fine, and redirection is not required.

You can add the method **validateUser()** to the **usermodel.ts** file in the **com/dotComIt/learnWith/model/** directory. Add the function inside the **UserModel** class:

```
validateUser() : boolean {
  if ( !this.user) {
    return false;
  }
  return true;
}
```

This method performs the check. If the **user** object is not initialized, then it returns false. Otherwise, it returns true.

Now we want to call this method when the tasks component initially loads. Open up the **tasks.component.ts** file in the **com/dotComIt/learnWith/views/tasks** directory. To perform our check, we are going to tie into the Angular component lifecycle. To do that, the **Component** will have to implement the **OnInit** interface. First, import the interface:

```
import {Component, OnInit} from '@angular/core';
```

The **Component** was already imported from the **@angular/core** library, but we can implement both classes from the same library with the same import statement. While we're at it, import the **Router**:

```
import {Router} from "@angular/router";
```

The **Router** will be used to perform the redirect in case the user is not properly logged in.

Now, add the interface after the class definition using the **implements** keyword:

```
export class TasksComponent implements OnInit {
```

An interface is a way to tell other classes that your class implements specific methods. In this case, we need the **ngOnInit()** method:

```
ngOnInit(): void {  
    if ( !this.userModel.validateUser() ) {  
        this.router.navigate(['/login']);  
    }  
}
```

Because our component implements the **OnInit** interface, Angular knows to execute the **ngOnInit()** method after Angular displays the data-bound properties and sets the component's input. If the user is logged in, nothing happens here. If the user is not logged in, then they'll be redirected to the login screen.

Loading the Tasks

The final step is to load the tasks. Go back to the **taskgrid.component.ts** file. Additionally, we're going to set up this component to implement the **OnInit()** method. First, perform the import:

```
import {Component, OnInit} from '@angular/core';
```

Then, implement the **OnInit** interface as part of the class definition:

```
export class TaskGrid implements OnInit {
```

Now, implement the **ngOnInit()** method:

```
ngOnInit(): void {  
}
```

The method is empty. Its purpose is to create a **TaskFilterVO** instance, set the default properties on it for **completed** and **startDate** and then load the tasks. First, create the **taskFilter** object:

```
let taskFilter : TaskFilterVO = new TaskFilterVO();
```

Then, set defaults on it:

```
taskFilter.completed = false;
taskFilter.startDate = new Date('3/1/2017');
```

Next, load the tasks:

```
this.loadTasks(taskFilter);
```

I decided to encapsulate the method for loading tasks instead of putting it in **ngOnInit()**. Before we jump into details of the **loadTasks()** method, we need to create a few variables on the **TaskGrid** class:

```
public tasks : TaskVO[];
public taskLoadError :string = '';
```

The **tasks** variable is an array of **TaskVO** objects and populates the grid. The new value is **taskLoadError** which is a string that will be used to display an error to the user if there is a problem loading data. Be sure the constructor injects both the **taskModel** and the **taskService** providers:

```
constructor(private taskModel :TaskModel, private taskService :
TaskService) {}
```

Now, let's look at the **loadTasks()** method. Here is its signature:

```
loadTasks(taskFilter:TaskFilterVO):void{
}
```

The first thing this method will do is set the **taskLoadError** property to an empty string:

```
this.taskLoadError = '';
```

Then, call the **loadTasks()** method on the **TaskService**:

```
this.taskService.loadTasks(taskFilter).subscribe(
  result => {
    // result code here
  },
  error => {
    // error code here
  }
);
```

The **loadTasks()** method on the **taskService** will return an observable object. Based on that, we'll run the first function if a successful response is returned, or the second function if an error occurs.

We'll start by implementing the **error()** method, because those are easier:

```
this.taskLoadError = 'We had an error loading tasks.';
```

This uses the TypeScript lambda operator. If there is an error, it sets the **taskLoadError** value and does nothing else.

The result handler does slightly more:

```
if ( result.error ) {  
    this.taskLoadError = 'We could not load any tasks.';  
    return;  
}  
this.tasks = this.taskModel.tasks = result.resultObject as TaskVO[];
```

If the result object contains an error, the **taskLoadError** value is set; thus, displaying an error to the user. Otherwise, the **resultObject**—which should be an array of **TaskVO**'s already—is set to the local **tasks** array, which will in turn update the grid to display the relevant rows.

Let's make sure we add the warning message to the **taskgrid.component.html** template:

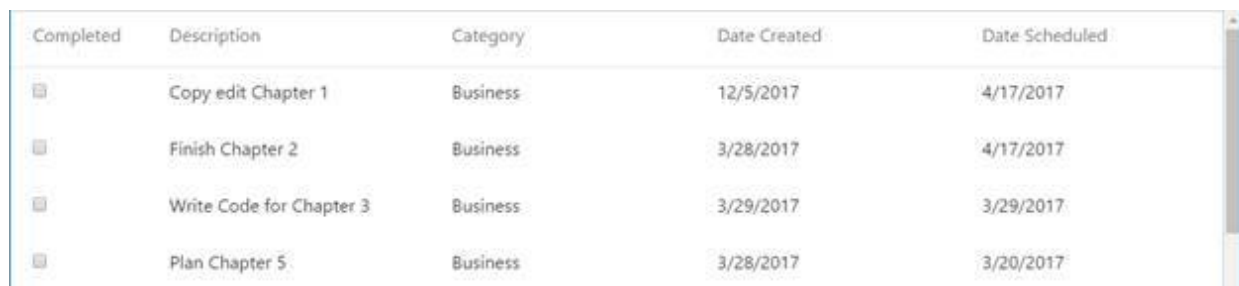
```
<div class="alert alert-danger" *ngIf="taskLoadError">  
    <h2>{{taskLoadError}}</h2>  
</div>
```

Put this at the top of the page, above the grid. If an error occurs, it should show:



Completed	Description	Category	Date Created	Date Scheduled
No data to display				

If the tasks load successfully, you'll see something like this:



Completed	Description	Category	Date Created	Date Scheduled
<input type="checkbox"/>	Copy edit Chapter 1	Business	12/5/2017	4/17/2017
<input type="checkbox"/>	Finish Chapter 2	Business	3/28/2017	4/17/2017
<input type="checkbox"/>	Write Code for Chapter 3	Business	3/29/2017	3/29/2017
<input type="checkbox"/>	Plan Chapter 5	Business	3/28/2017	3/20/2017

Final Thoughts

This chapter should have given you an understanding of how to use the **ngx-datatable** component with an Angular application, while introducing how to add and configure new custom components to your application. It also started the implementation of the **loadTasks()** service method and made use of an Angular component lifecycle hook, **OnInit**.

This summarizes the actions from within this chapter:

1. The app loads, and a **taskModel** and **TaskService** instances are created.
2. Then, user logs in. After successful login, the **TasksComponent** template loads.
3. The **ngOnInit()** method is executed in the **TasksComponent**. 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, then Angular will process the view, see the **TaskGrid** component is in the view.
4. The **TaskGrid** component is initialized, the **ngOnInit()** method is run. This creates a **TaskFilterVO** object, and calls a local **loadTasks()** method, which, in turn, calls the **loadTasks()** method in the **taskService**.
5. The **loadTasks()** service method retrieves the tasks based on the filter, and the success method is executed in Angular. The success method stores the returned tasks in the **TaskModel.tasks** array.
6. Using binding, the **ngx-datatable** will populate itself with the rows defined in the **taskgrid.component.html** template.

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 show you how to create the User Interface to allow for filtering tasks and review the services that power it. This chapter will explain modifications to 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 also show you how to populate a select box's data using Angular, and will introduce the **ng-bootstrap** library 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 **ngx-datatable**. 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 "false"; assuming the user would want to see tasks which that have not been completed yet.
- **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:





The image shows a filter UI for a task grid. It consists of several input fields and a button. On the left, there are two dropdown menus: 'Completed' with a value of 'Open Tasks' and 'Category' with a value of 'All Categories'. In the middle, there are two date choosers labeled 'Created After' and 'Created Before', each with a placeholder 'yyyy-mm-dd' and a calendar icon. On the right, there are two more date choosers labeled 'Scheduled After' and 'Scheduled Before', also with a placeholder 'yyyy-mm-dd' and a calendar icon. A 'Filter' button is located to the right of the date choosers.

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

Completed	Category
Open Tasks ▾	All Categories ▾
All	All Categories
Open Tasks	Business
Completed Tasks	Personal

This is the date chooser, implemented as a popup:

Created After	Created Before																																																	
yyyy-mm-dd 	yyyy-mm-dd 																																																	
<div style="display: flex; justify-content: space-between; align-items: center;"> < <div style="text-align: center;"> Apr ▾ 2017 ▾ </div> > </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <thead> <tr> <th style="color: #0070C0;">Mo</th> <th style="color: #0070C0;">Tu</th> <th style="color: #0070C0;">We</th> <th style="color: #0070C0;">Th</th> <th style="color: #0070C0;">Fr</th> <th style="color: #0070C0;">Sa</th> <th style="color: #0070C0;">Su</th> </tr> </thead> <tbody> <tr> <td>27</td><td>28</td><td>29</td><td>30</td><td>31</td><td>1</td><td>2</td> </tr> <tr> <td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td> </tr> <tr> <td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td> </tr> <tr> <td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td> </tr> <tr> <td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td> </tr> <tr> <td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td> </tr> </tbody> </table>		Mo	Tu	We	Th	Fr	Sa	Su	27	28	29	30	31	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	1	2	3	4	5	6	7
Mo	Tu	We	Th	Fr	Sa	Su																																												
27	28	29	30	31	1	2																																												
3	4	5	6	7	8	9																																												
10	11	12	13	14	15	16																																												
17	18	19	20	21	22	23																																												
24	25	26	27	28	29	30																																												
1	2	3	4	5	6	7																																												

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

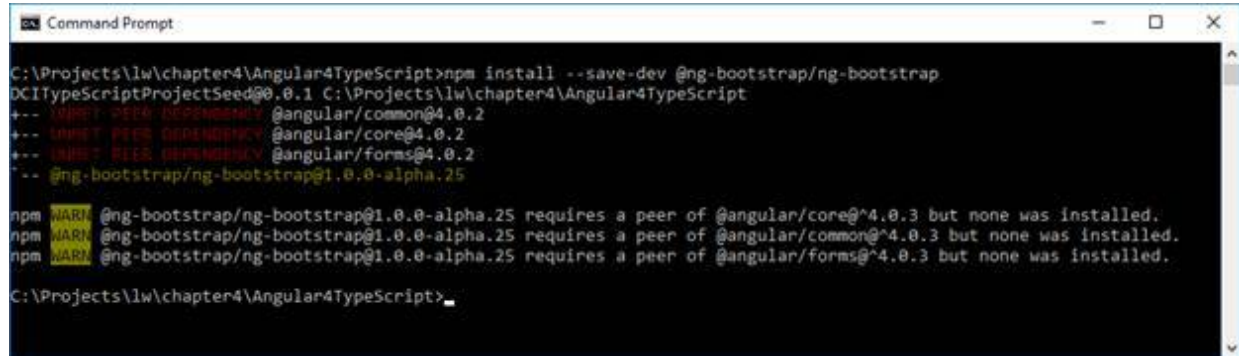
Setup ng-bootstrap

To create the **DatePicker**, we are going to use the **ng-bootstrap** library. This is an Angular version of Bootstrap; a CSS library. We used Bootstrap in earlier chapters to show fancy warning messages to the user when something went wrong. For the more advanced components, we are going to rely on the **ng-bootstrap**.

First, install it by running this on your command line:

```
npm install --save-dev @ng-bootstrap/ng-bootstrap
```

You'll see something like this:



```
Command Prompt
C:\Projects\lw\chapter4\Angular4TypeScript>npm install --save-dev @ng-bootstrap/ng-bootstrap
DCITypeScriptProjectSeed@0.0.1 C:\Projects\lw\chapter4\Angular4TypeScript
+-- @angular/common@4.0.2
+-- @angular/core@4.0.2
+-- @angular/forms@4.0.2
+-- @ng-bootstrap/ng-bootstrap@1.0.0-alpha.25

npm WARN @ng-bootstrap/ng-bootstrap@1.0.0-alpha.25 requires a peer of @angular/core@^4.0.3 but none was installed.
npm WARN @ng-bootstrap/ng-bootstrap@1.0.0-alpha.25 requires a peer of @angular/common@^4.0.3 but none was installed.
npm WARN @ng-bootstrap/ng-bootstrap@1.0.0-alpha.25 requires a peer of @angular/forms@^4.0.3 but none was installed.

C:\Projects\lw\chapter4\Angular4TypeScript>
```

Next, you need to tell the Gulp script to copy the **ng-bootstrap** JavaScript from the **node_modules** install directory into the build directory. Open the **config.js** file and look for the **angularLibraries** variable:

```
angularLibraries : [
  'core-js/client/shim.min.js',
  'zone.js/dist/**',
  'reflect-metadata/Reflect.js',
  'systemjs/dist/system.src.js',
  '@angular/**/bundles/**',
  'rxjs/**/*.*js',
  '@swimlane/ngx-datatable/release/index.js',
  '@ng-bootstrap/ng-bootstrap/bundles/ng-bootstrap.js'
],
```

The only thing I added was the final element of the array. That is all that is needed to copy the **ng-bootstrap** library.

There is one more thing you need to add. Download this [image](#) and put it in the **img** directory. We'll use it as the button to display and hide the calendar popup. It is not included in the build we downloaded with Node, but is used in the samples on the **ng-bootstrap** site.

Tell Angular how to find ng-bootstrap

We need to tell Angular how to find the **ng-bootstrap** library. There are two things we need to do. First, configure the SystemJS to be aware of **ng-bootstrap**. Second, load the library in the application's main module.

Open the **SystemJS.config.js** file in the **js/systemJSConfig** directory. Look for the **System.config** command. It creates and accepts object that defines SystemJS's config. Look for the **map** property. This tells **SystemJS** how to find all the Angular libraries. We just need to add the new library:

```
'@ng-bootstrap/ng-bootstrap': 'js:@ng-bootstrap/ng-bootstrap/bundles/ng-bootstrap.js'
```

Next, open the **app.module.ts** file in the **com/dotComIt/learnWith/main/** directory. Import **ng-bootstrap**:

```
import {NgbModule} from '@ng-bootstrap/ng-bootstrap';
```

Then, initialize the library as part of the **@NgModule's imports** list:

```
imports: [
  BrowserModule,
  AppRoutingModule,
  FormsModule,
  NgxDatatableModule,
  NgbModule.forRoot()
],
```

The **ng-bootstrap** library is initialized differently than other libraries we used in the past. Instead of just listing the module, it calls the **forRoot()** static method on the module. The **ng-bootstrap** library is a complex library with lots of components and other modules associated together. This approach just combines them so there is one easy import that gives our app access to the full library.

Modify the TaskFilterVO

Now we want to add our new properties to the **TaskFilterVO** class. Open the **TaskFilterVO.ts** file from the **com/dotComIt/learnWith/vo** directory:

```
export class TaskFilterVO {
  completed : boolean;
  endDate : Date;
  scheduledEndDate : Date;
  scheduledStartDate : Date;
  startDate : Date;
  taskCategoryID : number;
};
```

The class already had the **completed** property and the **startDate** property. We added a property for **endDate** to represent the end of the date range

for days that the component was created. The **scheduledEndDate** and **scheduledStartDate** properties will compare against the date the task was scheduled. Finally, the **taskCategoryID** will be to compare against the task's categorization.

Create the TaskFilter component

First, we're going to create a component to contain the filter UI. As with past components, we'll create three files for this; A CSS File, an HTML Template file, and a TypeScript class. Create the CSS file in the **com/dotComIt/learnWith/views/tasks** directory and name it **taskfilter.component.css**. For now, leave it as an empty file. Create the HTML template in the same directory and name it **taskfilter.component.html**. It can also be kept an empty file.

Now, create the **taskfilter.component.ts** file. To do this, you'll first, add some imports:

```
import {Component, OnInit} from "@angular/core";
import {TaskService} from "../../services/mock/task.service";
import {TaskModel} from "../../model/taskmodel";
```

This imports the **Component** and **OnInit** classes from **@angular/core**. It also includes our custom **TaskService** and **TaskModel** classes.

Now, create the **@Component** annotation:

```
@Component({
  selector: 'taskfilter',
  templateUrl :
    './com/dotComIt/learnWith/views/tasks/taskfilter.component.html',
  styleUrls:
    ['./com/dotComIt/learnWith/views/tasks/taskfilter.component.css']
})
```

The selector is named **taskfilter**, and will be used in the main task component view. The **templateUrl** and **styleUrls** values point to the HTML template and CSS template we just created.

Create the **TaskGrid** class at the end of the file:

```
export class TaskFilter implements OnInit {
  constructor(private taskModel :TaskModel,
               private taskService : TaskService) {
  }
}
```

```
ngOnInit(): void {  
  }  
}
```

This class contains implements the **OnInit** interface. It also injects instances for the **TaskService** which we'll use to call the service to populate the category drop-down, and the **TaskModel** will be used to save the category list for future use. The **ngOnInit()** method is added, but is left empty for the moment.

Now, switch back to the **app.module.ts**. Import the new class:

```
import {TaskFilter} from "../views/tasks/taskfilter.component";
```

And add it to the **@NgModule** declarations:

```
declarations: [  
  AppComponent,  
  LoginComponent,  
  TasksComponent,  
  TaskGrid,  
  TaskFilter  
],
```

While we're at it, open up the **tasks.component.html** template in the **com/dotComIt/learnWith/views/tasks** directory. Add the **TaskFilter** component:

```
<div class="wrapper">  
  <taskfilter class="taskFilter"></taskfilter>  
  <div class="mainScreenContainer">  
    <taskgrid></taskgrid>  
  </div>  
</div>
```

This template previously just displayed the **taskgrid**. Now it has the task filter, and some **div** wrappers. First, there is a wrapper class. Put this in the **tasks.component.css** file:

```
.wrapper {  
  display: flex;  
  height: 100%;  
  flex-direction: column;  
}
```

The **wrapper** sets up a Flexbox display. We are using Flexbox to display the filter, and have the task grid take up the rest of the remaining height.

This is the **taskFilter** CSS class:

```
.taskFilter{  
  width:100%;  
}
```

This tells the browser to size the **taskFilter** component to extend the full width. No height is specified so it can extend to the height it needs. The **TaskGrid** component was wrapped in a **maincontainer** class:

```
.mainScreenContainer{  
  flex : 1;  
  padding-top : 5px;  
}
```

This CSS class specifies the flex value as “1”. This is short hand for setting **flex-grow** and **flex-shrink** to “1”. It basically expands the **mainScreenContainer** to fill the rest of the height. This also adds some padding so it doesn’t bump up directly against the filter component. The **mainScreenContainer** will have more elements added to it in later chapters; such as the expand button and the scheduler component, which is why grid is wrapped in the **mainScreenContainer** instead of applying directly to the grid.

For the sake of completeness, let’s set up the same structure in the **TaskGrid** component. This component has two elements; the error alert and the actual grid. Open the **taskgrid.component.css** in the **com/dotComIt/learnWith/views** directory. Add a **taskGridWrapper** style:

```
.taskGridWrapper {  
  display: flex;  
  height: 100%;  
  flex-direction: column;  
}
```

Open up the **taskgrid.component.html** in the same directory. Wrap the full contents in the **taskGridWrapper** grid:

```
<div class="taskGridWrapper">  
  <!-- alert Here -->  
  <!-- grid Here -->  
</div>
```

This will just help smooth out the layout as needed.

Create the TaskFilter Template

Now we can turn our attention to the **taskfilter.component.html** template. We'll start by displaying an error alert before going into the filter specific UI elements:

```
<div class="taskFilterWrapper">
  <div class="alert alert-danger warningAlert taskFilterAlert"
    *ngIf="filterError">
    <h2>{{filterError}}</h2>
  </div>
</div>
```

The whole template is placed in a div with the CSS class **taskFilterWrapper**. Add this to the **taskfilter.component.css** file:

```
.taskFilterWrapper {
  display: flex;
  height: 100%;
  flex-direction: column;
}
```

This sets up the whole task filter for Flexbox sizing, then the div that creates the alert. It has a custom class after the regular classes:

```
.taskFilterAlert {
  padding: .2rem 1.25rem;
}
```

This just minimizes the padding so the **TaskFilter** alert doesn't take up too much space. Then, the div uses a ***ngIf** to hide the alert if the **filterError** value is empty. Create the **filterError** value in the **taskfilter.component.ts** file:

```
filterError : string;
```

Finally, the alert code displays the alert text, if relevant, and that starts our task filter template.

Populating a Select with Angular

Two select lists are required; 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**.

For the completed options, let's create a value object class to contain an option. Create the class **CompletedOptionVO.ts** in the **com/dotComIt/learnWith/vo** directory:

```
export class CompletedOptionVO {
  id : number;
  label : string;
  value :boolean;
};
```

This class contains three separate properties. The first is an **id** property which will represent a primary key for the value. The second will be a **label**, which is the display text. The final property is a **value** property, which will represent the actual value sent to the services. Let's add a constructor here to make it easy to create an instance of this class:

```
constructor(id :number, label :string, value:boolean) {
  this.id = id;
  this.label = label;
  this.value = value;
};
```

This constructor accepts three arguments, and sets all the class properties based on those values.

Now, switch over to the **taskmodel.ts** in the **com/dotComIt/learnWith/model** directory. Import the **CompletedOptionVO**:

```
import {CompletedOptionVO} from "../vo/CompletedOptionVO";
```

Inside the class definition, create the **taskCompletedOptions**:

```
taskCompletedOptions : CompletedOptionVO[] = [
  new CompletedOptionVO(-1, 'All', null),
  new CompletedOptionVO(0, 'Open Tasks', false),
  new CompletedOptionVO(1, 'Completed Tasks', true)
];
```

The array has three elements; each one an instance of the **CompletedOptionVO**. The value of the **completed** property is actually a tri-state property. It can be "true" for completed tasks, "false" for open tasks, or "null" for all tasks. If it is "null", the condition will not be added to the final class.

Let's create a **TaskCategoryVO** class, too. Create the file **TaskCategoriesVO.ts** in the **com/dotComIt/learnWith/vo** directory:

```
export class TaskCategoriesVO {
  taskCategoryID : number;
  taskCategory : string;
};
```

This class is simpler than the **CompletedOptionVO**. Since we won't be creating these instances all at once, I did not add a fancy constructor.

Since the **taskCategories** won't be loaded yet, we'll just populate that with an empty array inside the **TaskModel**:

```
taskCategories : TaskCategoryVO[]
```

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 [ (ngModel) ]="completed" >
      <option *ngFor="let task of taskModel.taskCompletedOptions"
        [value]="task.value">
        {{task.label}}
      </option>
    </select>
  </td>

```

The table row and table data are simple HTML tags. The select merely specifies the **ngModel**, which ties to a property on the **taskFilter** instance in the **TaskFilter** component class. Next up is the **options** tag. It has a new Angular directive; ***ngFor**. The ***ngFor** directive tells Angular to perform a loop and for each entry in the loop to create a new options tag. This is the syntax inside the ***ngFor**:

```
let task of taskCompletedOptions
```

It says to loop over the **taskCompletedOptions** array. As you loop, create an option tag for each element in the array. Inside the loop, use the **task** variable to access the current element. We do use the **task** variable in two places. The first is to set the **value** of the **option** tag using the **task.value** property. The second is to display the **label** inside the **option** tag. That creates our select drop-down for completed properties.

Where do the **completed** and **taskCompletedOptions** values come from? The **taskCompletedOptions** is from the **taskModel** which is injected into the constructor. The **completed** property needs definition, though. Create it in the **taskfilter.component.ts** file:

```
completed : string;
```

Inside the **ngOnInit()** method, set the **completed** property to "false":

```
this.completed = "false";
```

If you compile and run the app now, you should see the completed drop-down populate. However, our code is still in flux for the purposes of this chapter. Next, create the select for the category drop-down:

```

<td>
  <select [ (ngModel) ]="taskCategoryID" >
    <option *ngFor="let category of taskCategories"

```

```
        [value]="category.taskCategoryID">
            {{category.taskCategory}}
        </option>
    </select>
</td>
<tr>
```

This uses the same exact approach with slightly different values. The **ngModel** attaches itself to the **taskFilter.taskCategoryID**, and the options array is made from **taskCategories**; a local variable in the **TaskFilter** class. The value attribute of options points towards the **category.taskCategoryID**, and the **taskCategory** is displayed inside the option tag. You won't be able to see this drop-down work until later in the chapter after we implement and hook up to the service.

Adding a DateChooser

You probably know that a **DateChooser** is not a native HTML control. We are going to use the Bootstrap **DatePicker** from the **ng-bootstrap** library. Earlier, we set up UI Bootstrap, so it should be ready to use in our application.

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:

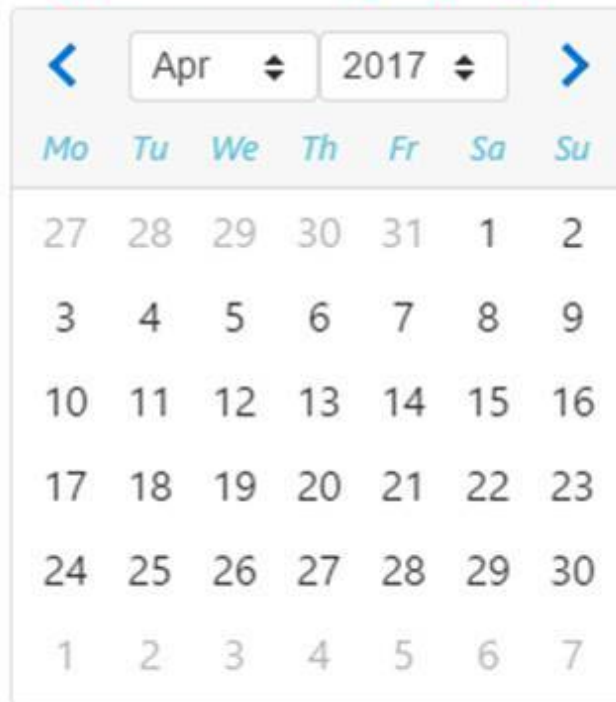
```
<td>Created After</td>
<td>Created Before</td>
<td></td>
<td>Scheduled After</td>
<td>Scheduled Before</td>
```

This 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:

Input →  ← Button

↓ Date Popup ↓



The image shows a date picker popup. At the top, there are navigation arrows (left and right) and two dropdown menus for the month (currently 'Apr') and the year (currently '2017'). Below this is a calendar grid with days of the week (Mo, Tu, We, Th, Fr, Sa, Su) as column headers. The grid shows dates from 27 to 30 in the first row, 3 to 9 in the second, 10 to 16 in the third, 17 to 23 in the fourth, 24 to 30 in the fifth, and 1 to 7 in the sixth row.

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 **DatePicker** components. I'll start with one, discuss it in detail, and then give you the code for the other three:

```
<td>
  <form class="form-inline">
    <div class="form-group">
      <div class="input-group"
        <input class="form-control" placeholder="yyyy-mm-dd"
          name="createdAfterDP" ngbDatepicker
          #createdAfterDP="ngbDatepicker"
          [(ngModel)]="startDate">
```

```

        <div class="input-group-addon"
            (click)="createdAfterDP.toggle()" >
            
        </div>
    </div>
</div>
</form>
</td>

```

The whole thing is wrapped in a **td** tag, which is in the second table row of the **taskfilter.component.html**. The **td** tag comes after the two tags for the select inputs. Inside the **td** is a **div** with the class **form-inline**, a CSS style from Bootstrap that is used to display items in a horizontal row. Next is the **form-group**, which adds some automatic spacing to the forms. The **input-group** follows, which uses Flexbox to help arrange things to the available space.

Finally, we get to the text input. It has multiple properties in it:

- **class**: This is a standard HTML property and refers to the CSS style **form-control**, which is just some Bootstrap styling.
- **placeholder**: This specifies a prompt to the user. Currently, the date format is specified.
- **name**: This property is required if you are going to use **ngModel** on the tag, which we do. I gave it a name; **createdAfterDP**.
- **ngbDatePicker**: This option tells Angular and **ng-bootstrap** that this input is part of a **DatePicker**. It does not have a value.
- **#createdAfterDP**: This property is a special name which Angular can use to reference the component inside of code. We gave it a value of **ngbDatepicker**, so we know it is a **DatePicker** component.
- **ngModel**: This property will be used to save the selected date to a variable defined in the class. It uses two-way binding, specified by the square brackets and parentheses. The **startDate** property has to be of type **ngbDateStruct**; which is like a value object with properties for the year, day, and month. I wish the value of this was a date object, but alas, it isn't.

Next in the code you'll notice another **div**. It has the CSS Class of **input-group-addon**. This is another Bootstrap style for making the image look like

a button. The div responds to a click event which will call the **toggle()** function on the **createdAfterDP** class. This is code behind the scenes to display the date popup.

Switch back to the **taskfilter.component.ts** to create the **startDate** property. First, import the class:

```
import { NgbDateStruct } from "@ng-bootstrap/ng-bootstrap";
```

Then, inside the class, create the instance:

```
startDate : NgbDateStruct;
```

While we're in there, let's create the class properties for the remaining class structures:

```
endDate : NgbDateStruct;  
scheduledStartDate : NgbDateStruct;  
scheduledEndDate : NgbDateStruct;
```

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

```
<td>  
  <form class="form-inline">  
    <div class="form-group">  
      <div class="input-group" >  
        <input class="form-control" placeholder="yyyy-mm-dd"  
          name="createdBeforeDP"  
          ngDatepicker #createdBeforeDP="ngbDatepicker"  
          [(ngModel)]="endDate">  
        <div class="input-group-addon"  
          (click)="createdBeforeDP.toggle()" >  
            
        </div>  
      </div>  
    </div>  
  </form>  
</td>  
<td></td>  
<td>  
  <form class="form-inline">  
    <div class="form-group">  
      <div class="input-group" >  
        <input class="form-control" placeholder="yyyy-mm-dd"  
          name="scheduledAfterDP"
```

```

        ngbDatepicker #scheduledAfterDP="ngbDatepicker"
        [(ngModel)]="scheduledStartDate">
        <div class="input-group-addon"
            (click)="scheduledAfterDP.toggle()">
            
            </div>
        </div>
    </div>
</form>
</td>
<td>
    <form class="form-inline">
        <div class="form-group">
            <div class="input-group" >
                <input class="form-control" placeholder="yyyy-mm-dd"
                    name="scheduledBeforeDP"
                    ngbDatepicker
#scheduledBeforeDP="ngbDatepicker"
                    [(ngModel)]="scheduledEndDate">
                <div class="input-group-addon"
                    (click)="scheduledBeforeDP.toggle()" >
                    
                </div>
            </div>
        </div>
    </form>
</td>

```

Your bootstrap **DatePickers** should be good to go.

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>

```

The button doesn't do anything yet, but we'll implement the functionality a bit later in this chapter.

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 **taskfilter.component.css** file from the **com/dotComIt/learnWith/views/tasks** 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:175px;
}
.completedDropDown {
    width:150px;
}
.taskCategoryDropDown {
    width:150px;
}
```

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

```
<select [(ngModel)]="taskFilter.completed" class="completedDropDown"
>
```

It uses the HTML **class** attribute to refer to the CSS style. The same happens for the task category drop-down:

```
<select [(ngModel)]="taskFilter.taskCategoryID"
class="taskCategoryDropDown">
```

The **datePicker** style is put on the paragraph which encloses it:

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

With the **DatePicker** code, the internal elements are set to expand to 100% of their width from using the Bootstrap styles. We control the width of the **DatePicker** by controlling the width of the **DatePicker'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 **datePicker** class is used on all the **DatePicker** inputs. However, 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. Since these are generic for the full application, I added them to the **styles.css** in the **styles** directory:

```
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.

Now back to the **taskfilter.component.css** for some task filter specific styles. 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 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, **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>
```

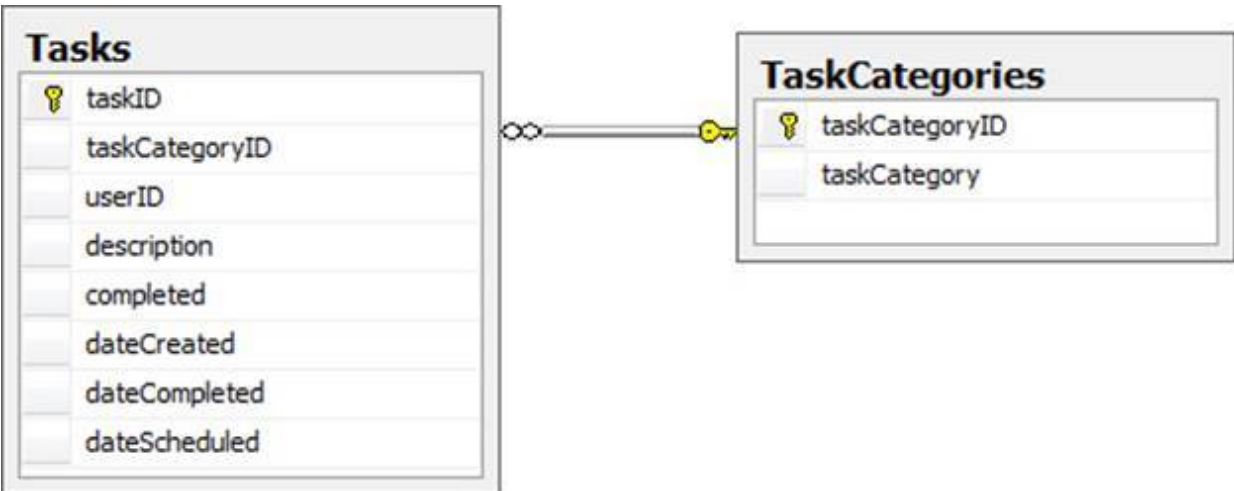
Finally, this is the bottom row, which contains the full **DatePicker** components:

```
<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 Service

This section will cover the NodeJS code needed to load the **TaskCategories** from the database and send them to the browser. It will also show you some new criteria to the method for filtering and loading tasks.

Revisit the `getFilteredTasks()` Method

Here are the fields that will 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** is chronologically 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.js** file. I'm just going to add these new elements to the **query**. The **TaskService** class is in the **com/dotComIt/learnWith/services** directory.

First, add a condition for the **taskCategoryID**:

```
if((json.taskCategoryID != undefined) && (json.taskCategoryID != "0")){
  if(firstOne){
    query = query + "Where "
    firstOne = false;
  } else {
    query = query + "and "
  }
  query = query + " taskCategories.taskCategoryID = " +
```

```
    json.taskCategoryID + " ";  
}
```

This starts with a condition that checks for the existence of the **taskCategoryID** property of the **json** filter object. If the **taskCategoryID** is "0", then the **taskCategory** parameter is ignored. This is done so that an "all" category can be added to the "category" drop down in the UI without having to store an "all" category record in the database. The condition starts by checking the **firstOne** property, which is used to determine if this is the first clause of the **query** or a continuation of existing clauses.

The previous chapter examined the **startDate** value and compared it to the **dateCreated** database column. This new code will also check for an **endDate** value, compared against the same column:

```
if(json.endDate != undefined){  
    if(firstOne){  
        query = query + "Where "  
        firstOne = false;  
    } else {  
        query = query + "and "  
    }  
    query = query + " dateCreated <= '" + json.endDate + "' ";  
}
```

This assumes that the dates will already be properly formatted to be recognized as dates by the SQL **query**, so we do not do any processing on the dates here. The **endDate** must be less than or equal to the database's **dateCreated**. I put this after the **startDate** condition which was created for the **query** in previous chapters.

The final condition of the **query** relates to the date the task was scheduled. This uses two separate values to compare against. The **scheduledStartDate** and **scheduledEndDate** are implemented similarly to the **startDate** and **endDate**.

```
if(json.scheduledStartDate != undefined){  
    if(firstOne){  
        query = query + "Where "  
        firstOne = false;  
    } else {  
        query = query + "and "  
    }  
}
```

```

    query = query + " dateScheduled >= '" + json.scheduledStartDate
+ "' ";
}
if(json.scheduledEndDate != undefined){
    if(firstOne){
        query = query + "Where "
        firstOne = false;
    } else {
        query = query + "and "
    }
    query = query + " dateScheduled <= '" + json.scheduledEndDate +
"' ";
}

```

I placed these conditions after the end date check in the **query**.

Loading Task Categories

The method to load task categories will be added to the **TaskService.js** file in the **com/dotComIt/learnWith/services** directory. First, add the method outline:

```

function getTaskCategories(response, queryString) {
}

```

The **getTaskCategories()** method uses the same method signature of all our request handler methods. In this case, a **queryString** will not be needed, but it is kept there for consistency. The other argument is the **response** argument that is used to send data back as part of the request.

Inside the method, we have some boilerplate code to define the **responseObject** and **callback**:

```

var responseObject = {};
var callback = '';
if(queryString.callback != undefined){
    callback = queryString.callback;
}

```

Next, create the **query**:

```

var query = "select * from taskCategories order by taskCategory";

```

The query is simple without any complicated filtering. It just retrieves all categories from the **TaskCategory** table. Next, execute the **query** using the **databaseConnection** module:


```
var dataQuery = databaseConnection.executeQuery(query,
  function(result) {
    // query result function here
  },
  function(err) {
    // query error function here
  }
)
```

The **databaseConnection**'s **executeQuery()** function accepts three arguments; the **query**, the result handler function, and the error handler function.

The result handler needs to process the results. Create the **responseObject**:

```
responseObject.error = 0;
responseObject.resultObject = result.recordset;
```

The **error** value of the **responseObject** is "0", and the **responseObject** is equal to the **recordset**. Finally, return the results to the browser using the **JSONResponseHandler** instance:

```
responseHandler.execute(response, responseObject, callback);
```

The error handler function sets the **error** property on the **responseObject** to "0", and then returns it to the calling entity:

```
responseObject.error = 1;
responseHandler.execute(response, responseObject, callback);
```

If an error exists, send back a **responseObject** to the browser with the **error** set to "1".

That completes the **getTaskCategories()** method, but don't forget to export it:

```
exports.getTaskCategories = getTaskCategories;
```

Next, you need to tell the app when to call this method. Open up the **ResponseHandlers** file in the **com/dotComIt/learnWith/dotComIt/server** directory. Add this line:

```
handlers["/taskService/getTaskCategories"] =
  taskService.getTaskCategories;
```

Restart your NodeJS application, and whenever a request is received for `"/taskService/getTaskCategories"`, the `getTaskCategories()` method will be executed.

Testing Task Categories

You can test this easily by loading a URL in the browser:

```
http://127.0.0.1:8080/taskService/getTaskCategories
```

You should see results similar to this:

```
{
  "responseObject":
  [
    {"taskCategoryID":0.0, "taskCategory":"All Categories"},
    {"taskCategoryID":1, "taskCategory":"Business"},
    {"taskCategoryID":2, "taskCategory":"Personal"}
  ],
  "error":0
}
```

Access the Service

This section will cover the new Angular code needed to retrieve the task categories from the server. Open the **TaskService.ts** class in the **com/dotComIt/learnWith/services/nodejs** directory.

Create the **loadTaskCategories()** method:

```
loadTaskCategories() : Observable<ResultObjectVO> {  
  let parameters = "callback" + "=" + "JSONP_CALLBACK" ;  
  let url = SERVER + 'taskService/getTaskCategories?' + parameters;  
  return this.jsonp.request(url)  
    .map((result) => result.json() as ResultObjectVO);  
}
```

No arguments are passed into the **loadTaskCategories()** function. The only parameter sent to the service is the **callback**. The URL is created by concatenating the **SERVER** constant with the method endpoint and the parameter array. Then, the **jsonp** service initiates the request. The **Observable** object is returned, and the **map()** function converts the results from JSON into an **instance** of the **ResultObjectVO**.

That's it. This one is pretty simple.

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 **taskfilter.component.ts** file in the **com/dotComIt/learnWith/views/tasks** directory:

```
loadTaskCategories():void {
    this.taskService.loadTaskCategories().subscribe(
        result => {
        },
        error => {
        }
    );
};
```

This will execute the **loadTaskCategories()** method inside the **taskService**. An observable is returned from the service. Based on the observable resolution, the first function argument will run if we get a successful result, and the second function will run if there is an error. Both the result and failure methods are empty stubs using the lambda notation to create the result function.

Let's populate the error function first:

```
this.filterError = 'There was a task category service error';
```

It just sets the **filterError** so the user know that something had happened. Easy enough. Now, the success method:

```
if ( result.error) {
    this.filterError = 'Error loading task Categories';
    return;
}
this.taskModel.taskCategories = result.resultObject as
TaskCategoryVO[];
this.taskCategories = Object.assign( [],
this.taskModel.taskCategories );
let allTask = new TaskCategoryVO();
allTask.taskCategoryID = 0;
allTask.taskCategory = "All Categories";
```

```
this.taskCategories.unshift(allTask);  
this.taskFilter.taskCategoryID = 0;
```

This method does a few things. First, it checks the **result.error** property. If the property is true, then an error occurred, so the **filterError** value is set. This causes the alert to show in the view, and processing stops. If there is no error, then the **TaskCategoryVO** array is saved to the **taskModel**. This is for future reference when we are creating and editing new tasks.

Next, I copy the **taskModel.taskCategories** array using the **Object.assign()**. This is a way to deep copy objects built into the ES6 specification of JavaScript. Most browsers support it natively, but the Polyfill library included as an Angular requirement should make it work universally without issues.

The reason for creating a deep copy of the task category array is to add a new item. Create a new instance of the **TaskCategoryVO**. Give its **taskCategoryID** the value of "0", and the **taskCategory** the value of "All Categories". It is important in the UI to be able to select the all categories option when viewing tasks. Then, use **unshift()** to add the new category onto the local **taskCategories** array. Finally, this method sets the default **taskCategoryID** in the **taskFilter** instance. We created the **taskFilter** instance earlier in this chapter.

Now, make sure that the new method is called from the **ngOnInit()**:

```
ngOnInit(): void {  
    this.taskFilter.completed = false;  
    this.loadTaskCategories();  
};
```

We were already setting the completed property to "false". This is all we need to load and use the categories.

Triggering the Filter

The final step for this chapter is to implement the code behind the filter button. The button is inside the **taskfilter.component.html** template in the **com/dotComIt/learnWith/views/tasks** directory. It needs to respond to the click event:

```
<input type="button" value="Filter" (click)="filter()"/>
```

A **click** directive was added. When clicked, the method **filter()** will be executed inside the **taskfilter.component.ts** file. The purpose of the **filter()** method is to put together a filter object and call the **loadTasks()** method. We already implemented a **loadTasks()** method inside the **taskgrid.component.ts** file, so we need to trigger that somehow.

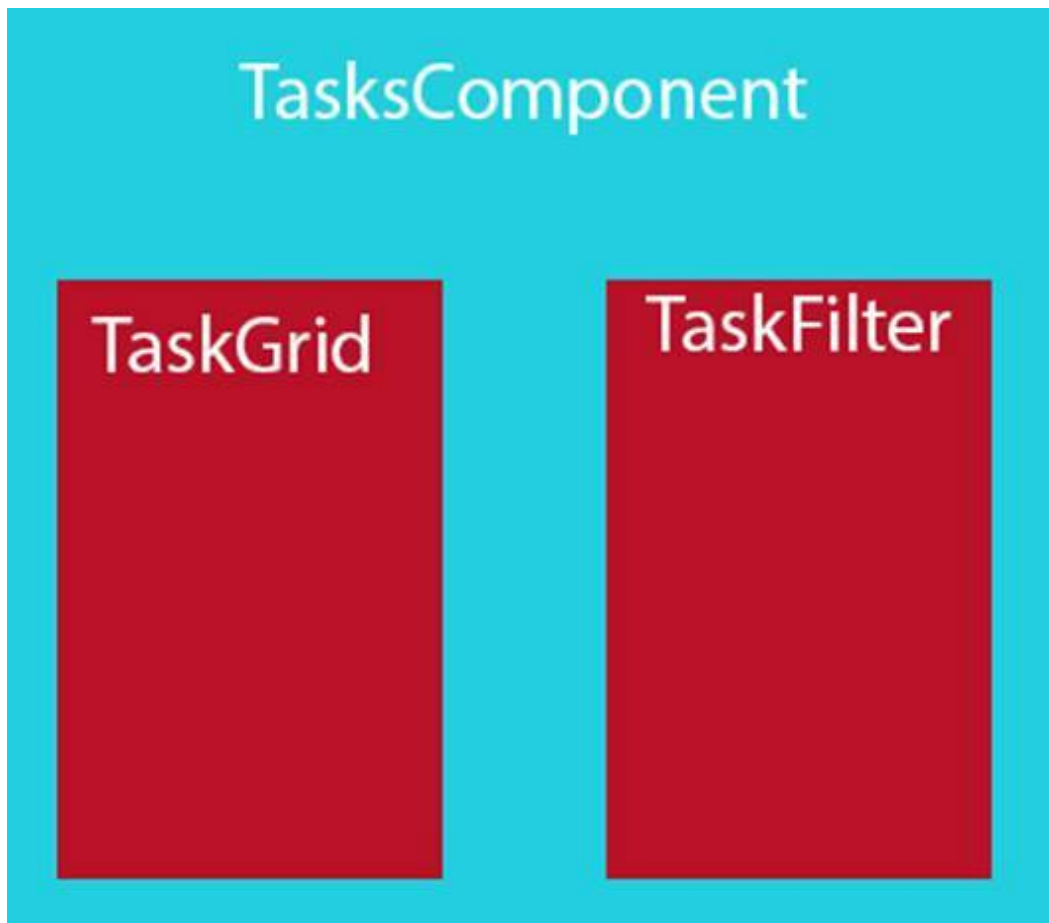
We will add a **taskFilter** argument now. This will, in turn, be passed onto the **loadTasks()** method in the **TaskService**:

```
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);
```

The architecture of the components is like this:



The **TaskGrid** and the **TaskFilter** components are children of the **TasksComponent**. Using standard encapsulation principles, the two children should not be allowed to talk to each other, so how does the **TaskFilter** trigger a method call inside the **TaskGrid**? We'll declare an output property of the **TaskFilter**. This is like creating our own event to dispatch, similar to how a **click** works on buttons. The **TasksComponent** will listen to the event on the **TaskFilter**, then execute a method on the **TaskGrid**.

First let's setup the **TaskFilter** to dispatch the event. First, import the **Output** class from the **@angular/core** library:

```
import {Component, EventEmitter, OnInit, Output} from
"@angular/core";
```

The **EventEmitter** class is also imported. **Output** is used to define the external event name while **EventEmitter** is used to dispatch the event. Create the output event:

```
@Output() filterRequest = new EventEmitter<TaskFilterVO>();
```

This uses the **@output()** annotation before the event name. The name of the event is **filterRequest** and it represents an **EventEmitter** of type **TaskFilterVO**.

Before the event is dispatched, we need to create the **TaskFilterVO** instance. Create the **filter()** method:

```
filter():void {
}
```

Inside the method, create the new variable:

```
let taskFilter : TaskFilterVO = new TaskFilterVO();
```

We need to copy the input values from the view into the properties of this new instance. Start with the **startDate**:

```
if (this.startDate) {
    taskFilter.startDate = new Date(this.startDate.month + '/' +
                                   this.startDate.day + '/' +
                                   this.startDate.year) ;
} else {
    taskFilter.startDate = null;
}
```

If the **startDate** is initialized, then we need to copy over the value. Remember, the **DatePicker**'s selected value is of type **NgbDateStruct**, not **Date**, so we have to convert it. I did that by creating a date string and sending that to the **Date** constructor. If no **startDate** is specified, the **taskFilter** property is set to **null**, which will tell the service to ignore the property.

The other date properties are created similarly:

```
if (this.endDate) {
    taskFilter.endDate = new Date(this.endDate.month + '/' +
                                this.endDate.day + '/' +
                                this.endDate.year) ;
} else {
    taskFilter.endDate = null;
}
if (this.scheduledStartDate) {
    taskFilter.scheduledStartDate = new
Date(this.scheduledStartDate.month +
                                '/' +
this.scheduledStartDate.day +
                                '/' +
this.scheduledStartDate.year) ;
} else {
    taskFilter.scheduledStartDate = null;
}
if (this.scheduledEndDate) {
    taskFilter.scheduledEndDate = new
Date(this.scheduledEndDate.month + '/' +
                                this.scheduledEndDate.day
+ '/' +
                                this.scheduledEndDate.year) ;
} else {
    taskFilter.scheduledEndDate = null;
}
```

The **completed** value was a different challenge. It should have three possible values; true, false, or undefined. However, those values were being translated into strings. As such, I had to do some fancy conversion:

```
if (this.completed === "null") {
    taskFilter.completed = null;
} else if (this.completed === "false" ) {
```



```
    taskFilter.completed = false;
  } else {
    taskFilter.completed = true;
  }
```

I was hoping for an easier approach, but this suffices.

The **taskCategoryID** had a similar problem regarding strings and numbers. I formally converted the selected value to the **taskFilter** property:

```
taskFilter.taskCategoryID = Number(this.taskCategoryID);
```

The last step is to emit the event. This is done by calling the **emit()** function on the **filterRequest EventEmitter** instance:

```
this.filterRequest.emit(taskFilter);
```

That ends the **filter()** method.

Catching the filterRequest Event

Handling the **filterRequest** event is no different than handling a built in Angular event; such as the click. Open the **tasks.component.html** in **com/dotComIt/learnWith/views/tasks** directory.

```
<taskfilter class="taskFilter"
(filterRequest)="filterRequest($event)">
</taskfilter>
```

The **filterRequest** is added as an attribute to the **taskfilter** tag. It is surrounded by parentheses, meaning it is being bound to an event handler. The method **filterRequest()** is called, and the event is passed in.

While we're in the **tasks.component.html** template, add a name to the **taskgrid**:

```
<taskgrid #taskgrid></taskgrid>
```

The hash tag name allows us to access this view inside the **TasksComponent** class. Open that up now, from the file **tasks.component.ts**. Import the **ViewChild** class from **@angular/core**:

```
import {Component, OnInit, ViewChild} from '@angular/core';
```

Inside the class, get a hook to the instance of the **TaskGrid**:

```
@ViewChild(TaskGrid)
private taskgrid : TaskGrid;
```

The **@ViewChild** annotation tells Angular to look for a view child of the type **TaskGrid** in this component's view template. The private variable name tells it to look for the **TaskGrid** instance named **taskgrid**. Once we have this reference, the **TasksComponent** can execute public methods or properties on the **TaskGrid**.

Now, create the **filterRequest()** method:

```
filterRequest(filter:TaskFilterVO):void {  
    this.taskgrid.loadTasks(filter);  
}
```

The event argument is an instance of the **TaskFilterVO**, which is the input to the **loadTasks()** method in the **TaskGrid**. This method is just a pass through.

This is the process we've developed in this section:

1. User clicks the filter button.
2. The **TaskFilter** component creates an instance of the **TaskFilterVO** and emits an event with the **taskfilter** as event data.
3. The **TasksComponent** listens to the event and executes an event handler; **requestFilter()**.
4. The **TasksComponent** uses the **@ViewChild** declaration to access an instance of the **taskGrid**.
5. The **requestFilter()** method, executes the **loadTasks()** method on the **taskGrid** child, sending it to the **TaskFilterVO** instance created in the **TaskFilter**, and passed as part of the event mechanism.
6. The **TaskFilter** component's **loadTask()** methods calls the **loadTask()** service, processes the data, and updates the grid.

It is good to understand how these things are structured.

Test the Filtering

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	Description	Category	Date Created	Date Scheduled
<input type="checkbox"/>	Copy edit Chapter 1	Business	12/5/2017	4/17/2017
<input type="checkbox"/>	Finish Chapter 2	Business	3/28/2017	4/17/2017
<input type="checkbox"/>	Write Code for Chapter 3	Business	3/29/2017	3/29/2017
<input type="checkbox"/>	Plan Chapter 5	Business	3/28/2017	3/20/2017
<input type="checkbox"/>	Create React Proof of Concept	Business	3/31/2017	

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

Completed	Description	Category	Date Created	Date Scheduled
<input type="checkbox"/>	Buy Milk	Personal	5/9/2017	2/11/2017
<input type="checkbox"/>	Buy Eggs	Personal	5/9/2017	11/24/2017
<input type="checkbox"/>	Clean Kitchen	Personal	5/9/2017	11/21/2017
<input type="checkbox"/>	Wish Mom a Happy Birthday!	Personal	5/9/2017	11/22/2017
<input type="checkbox"/>	Call Brother	Personal	5/27/2017	11/22/2017

Finally, try changing the category to “Business”, and setting the “Created After” date to 4/21:

Completed	Description	Category	Date Created	Date Scheduled
<input type="checkbox"/>	Copy edit Chapter 1	Business	12/5/2017	4/17/2017
<input type="checkbox"/>	Do Something	Business	5/9/2017	11/24/2016
<input type="checkbox"/>	Followup with client on project	Business	5/14/2017	11/22/2016
<input type="checkbox"/>	Start Chapter 6	Business	11/13/2017	
<input type="checkbox"/>	Update StackOverflow Profile	Business	11/16/2017	

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 how to setup the **ng-bootstrap** project for use within your Angular application, and how to create a **DatePicker** component. We also used Angular to populate select boxes. We covered how to communicate between components using events and **ViewChildren** annotations. Some services were reviewed, a new one created, and everything was wired up to create a working and functional UI.

The next chapter will focus on the system for creating and editing task.

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 with Angular, and show you how to create a modal popup using ng-bootstrap and Angular. We'll create new service methods for saving and updating a task. Finally, it will show you how to tie everything together by wiring up the UI to the services.

Create the User Interface

This section helps you create a popup window that can be used for both creating new tasks, and editing existing ones. It will start by showing you 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.

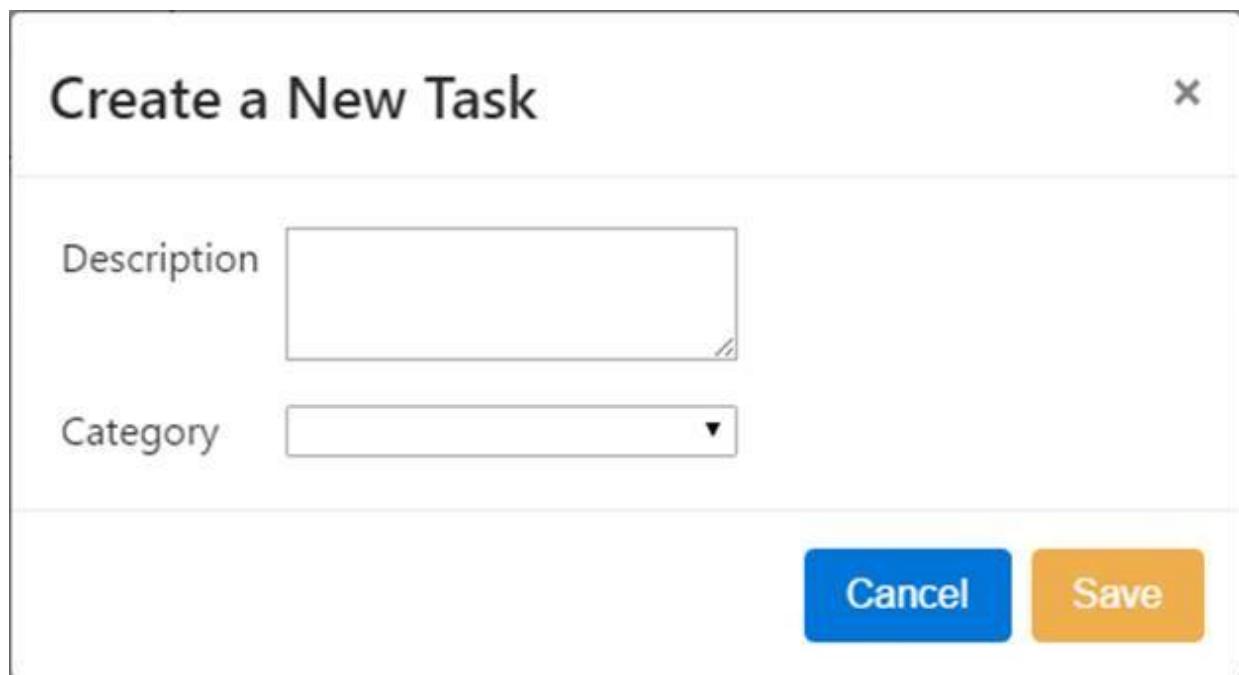
The Task Window

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, task creation date, and 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 image shows a modal dialog box titled "Create a New Task" with a close button (X) in the top right corner. The dialog contains two input fields: "Description" with a text input box and "Category" with a dropdown menu. At the bottom right, there are two buttons: "Cancel" (blue) and "Save" (orange).

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

Create the Popup Component

The first thing that you need to do is to create a new component to represent the popup. As with past Angular components we’ve created, it will contain a CSS file, an HTML template, and a TypeScript file. Create the three files in the **com/dotComIt/learnWith/views/tasks** directory; **taskcu.component.html**, **taskcu.component.css**, and **taskcu.component.ts**.

Start in the TypeScript file and add a few imports:

```
import {Component, Input, OnInit} from '@angular/core';
import {NgbActiveModal} from "@ng-bootstrap/ng-bootstrap";
```

This imports the **Component** class, the **Input** class, and the **OnInit** class from **@angular/core**. Next, the **NgbActiveModal** class is imported from **@ng-bootstrap/ng-bootstrap**. The **NgbActiveModal** is a reference to the current popup, and we will use this to close it after processing is complete.

Now, import a few of our own classes:

```
import {TaskVO} from "../../vo/TaskVO";
import {TaskService} from "../../services/mock/task.service";
import {TaskModel} from "../../model/taskmodel";
import {UserModel} from "../../model/usermodel";
```

We’ll use these classes to implement the popup’s functionality. Now, create the **@Component** annotation:

```
@Component({
  selector: 'taskcu',
  templateUrl :
  './com/dotComIt/learnWith/views/tasks/taskcu.component.html',
  styleUrls: [
  './com/dotComIt/learnWith/views/tasks/taskcu.component.css' ]
})
```

This gives the Component a selector of **taskcu**, defines the HTML template, and references the CSS Stylesheet.

Finally, we can create the class:

```
export class TaskCU implements OnInit {  
}
```

The class is named **TaskCU**. The CU in the name is an acronym for create and update. The class requires two input values:

```
@Input()  
title : string;  
@Input()  
task :TaskVO;
```

The title will be displayed as part of the modal's header. It will change dependent upon whether we are editing a class, or creating a class. The other element is the task we are editing. If it is blank, the component will assume we are creating a new task. Notice that each input value has the annotation **@Input()**. This is how Angular says it is expecting these values to be provided to the component.

Then, add a local variable:

```
taskUpdateError :string;
```

This will be used to display an alert error to the user in case there are service problems saving or update the task. Now, add the constructor:

```
constructor(private activeModal: NgbActiveModal,  
             private taskService:TaskService,  
             private userModel:UserModel,  
             private taskModel:TaskModel) {  
}
```

Though the constructor does not have any functionality, it does pass three services into this component's class. To update or create the task upon save, the **taskService** will be used. The **taskModel** will be used to populate the task categories drop-down in the view template. Finally, should the need arise, the **activeModal**—which comes from **ng-bootstrap**—will be used to close the modal.

Finally, add the **ngOnInit()** method:

```
ngOnInit(): void {  
    if (!this.task) {  
        this.task = new TaskVO;  
    }  
};
```


This is a quick check to see if the task input is defined or not. If not, then create a new empty **TaskVO** instance.

While we are in here, let's create a stub for the save method:

```
onSave() :void {  
}
```

We'll loop back to this method later in the chapter after we created the service.

Populate the Popup Template

The file has three parts; a header, a content area, and a footer area. Open the **taskcu.component.html** file in the **com/dotComIt/learnWith/views/tasks** directory.

This is the header:

```
<div class="modal-header">  
  <h4 class="modal-title">{{title}}</h4>  
  <button type="button" class="close" aria-label="Close"  
    (click)="activeModal.dismiss('Cross click')">  
    <span aria-hidden="true">&times;</span>  
  </button>  
</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 the property inside the **TaskCU** class. This also includes an "X" button, which calls the **activeModal** service to dismiss the popup.

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. Within the content area of the form is the error alert, the description text area, and the category drop-down. First, the error alert:

```
<div class="alert alert-danger" *ngIf="taskUpdateError">  
  <h2>{{taskUpdateError}}</h2>  
</div>
```

The editable content is in a table for easy layout. Next, the description:

```
<table>
  <tr>
    <td>Description</td>
    <td>
      <textarea [ (ngModel) ]="task.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 class's task object.

Lastly, 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 [ (ngModel) ]="task.taskCategoryID"
      class="taskCUDropDown">
      <option *ngFor="let category of
taskModel.taskCategories"
        [value]="category.taskCategoryID">
        {{category.taskCategory}}
      </option>
    </select>
  </td>
</tr>
</table>
```

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

The category select uses a new CSS style. Create it in the **taskcu.component.css** file:

```
.taskCUDropDown{
  width:100%
}
```

It stretches the width of the drop-down to 100% of the table cell it is in.

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

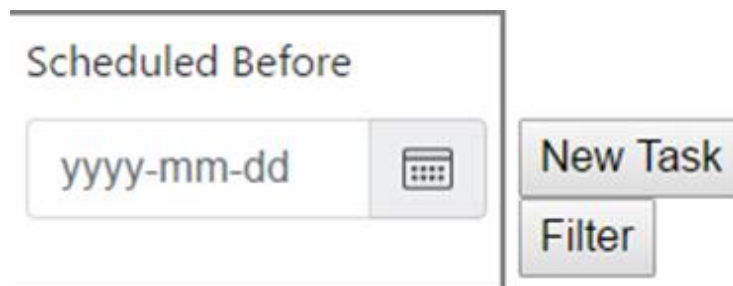
```
<div class="modal-footer">
  <button class="btn btn-primary" type="button"
    (click)="activeModal.dismiss('Close click')">
    Cancel
  </button>
  <button class="btn btn-warning" type="button"
    (click)="onSave()">
    Save
  </button>
</div>
```

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. Additionally, the close button calls on the **activeModal** instance to close the popup, while the save button will call the **onSave()** method inside the **TaskCU** component.

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.component.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" (click)="newTask()"/>
  <input type="button" value="Filter" (click)="filter()"/>
</td>
```

The new task button uses the **click** directive to call the **newTask()** method inside the **TaskFilter** class:

```
newTask() : void {
  this.newTaskRequest.emit();
}
```

All this does is emit a new task request. Be sure to define that new output event in the class:

```
@Output()
newTaskRequest = new EventEmitter();
```

None of the code to maintain the popup is in the **TaskFilter** component; it is in the **TaskComponent**. Notice the **newTaskRequest** has an **@Output()** annotation. Just like **@Input()** defines a value intended to be sent into the component, **@Output()** defines an event which will be sent out of the component. Be sure to import the **Output** class from the **@angular/core** library:

```
import {Component, EventEmitter, OnInit, Output} from
"@angular/core";
```

Now, open the **tasks.component.html** file:

```
<taskfilter class="taskFilter"
  (filterRequest)="filterRequest($event)"
  (newTaskRequest)="newTask()">
</taskfilter>
```

The **filterRequest** was already there, but now we've added a **newTaskRequest** which will call the **newTask()** method inside the **tasks.component.ts** file. Here is the **newTask()** method:

```
newTask() : void {
    this.openTaskWindow('Create a New Task');
}
```

This is just a proxy to a secondary method named **openTaskWindow()**:

```
private openTaskWindow(title:string, task:TaskVO = null) {
}
```

The **openTaskWindow()** method accepts two arguments; each one, a parallel to the two input arguments of the **TaskCU** component. They are the title, and the task to be edited. Notice that the task argument is optional and will default to a null string if not specified.

Now, create the popup:

```
const modalRef = this.modalService.open(TaskCU );
```

What is the **modalService**? We haven't injected it into this class yet. Do so as part of the constructor:

```
constructor(private userModel :UserModel,
            private router: Router,
            private modalService: NgbModal) {
}
```

And you'll have to import the **NgbModal** class:

```
import {NgbModal} from "@ng-bootstrap/ng-bootstrap";
```

Import the **TaskCU** component while you're dealing with imports:

```
import {TaskCU} from "../taskcu.component";
```

Back to the **openTaskWindow()** method. We call an **open()** method on the **modalService** component and pass in the actual component type—not an instance of the component. The **ng-bootstrap** library will create an instance of this component at runtime.

To allow Angular to create component instances at runtime, they must be defined as such when we set up the module. So, open up the **app.module.ts** file from the **com/dotComIt/learnWith/main** directory. Import the **TaskCU**:

```
import {TaskCU} from "../views/tasks/taskcu.component";
```

Add it to the **declarations** array of the **@NgModule**:

```
declarations: [  
  AppComponent,  
  LoginComponent,  
  TasksComponent,  
  TaskGrid,  
  TaskFilter,  
  TaskCU  
],
```

Then, add an **entryComponents** array top the **@NgModule**:

```
entryComponents: [TaskCU]
```

The **entryComponents** is how we tell Angular we may be creating instances of this at runtime.

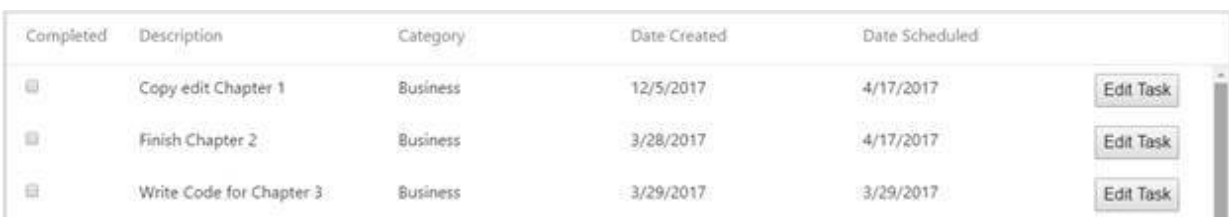
Go back to the **openTaskWindow()** method in the **tasks.component.ts** file. We created the modal reference, but still need to pass in our title and task:

```
modalRef.componentInstance.title = title;  
modalRef.componentInstance.task = task;
```

At this point, you should be able to run the app, click the new task button, and see the popup. It should even close if you click the cancel—or “X”—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	
<input type="checkbox"/>	Copy edit Chapter 1	Business	12/5/2017	4/17/2017	Edit Task
<input type="checkbox"/>	Finish Chapter 2	Business	3/28/2017	4/17/2017	Edit Task
<input type="checkbox"/>	Write Code for Chapter 3	Business	3/29/2017	3/29/2017	Edit Task

To create this, we will create a new column template similar to how we created the completed checkbox, or formatted the date. Open up the **taskgrid.component.html** file. Add the new column to the end of the **ngx-datatable**:

```
<ngx-datatable-column maxWidth="150" >  
  <ng-template let-row="row" ngx-datatable-cell-template>  
    <button (click)="onEditTask(row)">Edit Task</button>  
  </ng-template>  
</ngx-datatable-column>
```

The column starts with the **ngx-datatable-column** list. I specified a **maxWidth** for the column at 150 pixels on the column. The **ng-template** defines the button container. In the past, we specified a **let-value** to pass a value into the template. This time, we specified a **let-row** property. This will pass the full row object into the template. The button just executes a click to run the **onEditTask()** method. The row value, which represents a **TaskVO** instance, is passed to the **onEditTask()** method.

In the **taskgrid.component.ts** file, create the **onEditTask()** method:

```
onEditTask(value:any) :void {  
    this.editTaskRequest.emit(value);  
}
```

This just emits the **editTaskRequest @output** event. Create it as part of the class:

```
@Output() editTaskRequest = new EventEmitter<TaskVO>();
```

The **TaskVO** is passed as part of the event. Make sure that this component imports **Output** and **EventEmitter**:

```
import {Component, EventEmitter, OnInit, Output} from  
'@angular/core';
```

To respond to the **editTaskRequest** event, open the **tasks.component.html** file:

```
<taskgrid #taskgrid (editTaskRequest)="editTask($event)"></taskgrid>
```

The **taskgrid** component now has calls the **editTask()** method in response to the **editTaskRequest** event.

Go to the **tasks.component.ts** file to find the **editTask()** method:

```
editTask(task:TaskVO) : void {  
    this.openTaskWindow('Edit Task', Object.assign( {}, task));  
}
```

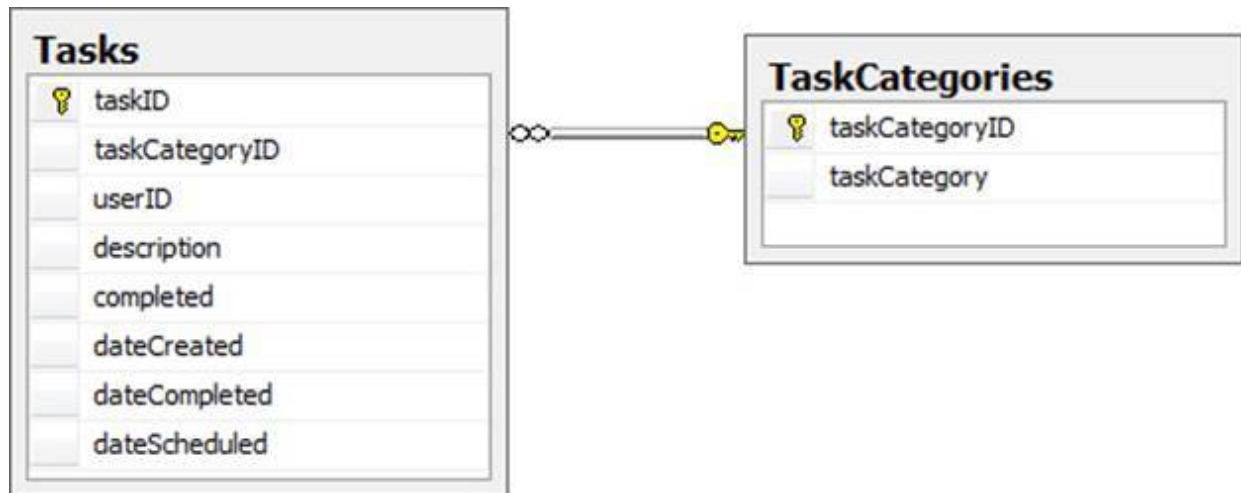
This is a pass-through for the **openTaskWindow()** method we created earlier in this chapter. This time, instead of passing in a null task, we pass in the value that was propagated up from the **TaskGrid**. Notice that we use **Object.assign()** to create a deep copy of the task. If we do not create a copy, then the grid may be updated before the service updates the task. This way,

we do not have to worry about rolling back UI visual changes if the user decides to cancel the operation and not save their task updates.

After implementing this last bit, the app should run and open the dialog popup for both a new task, and 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 explain the NodeJS code needed for the services. It will show you how to create methods in the **TaskService** for creating, and updating the task. It will show some test URLs you can use to create or update tasks.

Modify the `getFilteredTasks()` method

Before looking at the code to update or create, a task we need to make another change to is the **getFilteredTasks()** method. The **getFilteredTasks()** method was originally 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 new filter for the **TaskID**. This will make it easy to return the new or updated task after the process is completed.

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

```
if(json.taskID != undefined) {
    if(firstOne) {
        query = query + "Where "
        firstOne = false;
    } else {
        query = query + "and "
    }
    query = query + " taskID = " + json.taskID + " ";
}
```

If the **firstOne** variable is “true”, then add a “where” clause to the query, and set the **firstOne** value to “false”. Otherwise, an “and” is added to concatenate the new condition with the previous conditions. Finally, the **taskID** check is added. This is an equality comparison of the **taskID** in the database with the **taskID** passed in as part of the filter.

Creating a New Task

Open up the **TaskService** class in the **com/dotComIt/learnWith/services** directory, then create a method stub for the **createTask()** method:

```
function createTask(response, queryString) {
}
```

The **createTask()** method will be one of our URL handlers. Therefore, it accepts a **response** object to send information back to the user, and a **queryString** object that contains the information that the user sent in to the method.

First in the method is some boilerplate code:

```
var resultObject = {};  
var callback = '';  
if(queryString.callback != undefined){  
    callback = queryString.callback;  
}
```

This creates a generic return object—**resultObject**—and also saves the callback string, if available, in a **callback** variable.

Next, make sure that the relevant URL variables are present in the **queryString** value:

```
if((queryString.taskCategoryID == undefined) ||  
    (queryString.userID == undefined) ||  
    (queryString.description == undefined)){  
    resultObject.error = 1;  
    responseHandler.execute(response, resultObject, callback);  
} else {  
    // process query here  
}
```

The variables needed for this method are the **taskCategoryID**, the **userID**, and the **description**. If none of those are defined, then the **error** response is sent back to the user. If all values are defined as needed, it is time to create the query:

```
query = "insert into tasks(taskCategoryID, userID, description,  
completed,  
                                dateCreated) values( "  
if(queryString.taskCategoryID != 0) {  
    query = query + " " + queryString.taskCategoryID + ", ";  
} else {  
    query = query + " null, ";  
}  
query = query + " " + queryString.userID + ", ";  
query = query + " '" + queryString.description + "', ";  
query = query + " 0, ";  
query = query + " GETDATE() ) ";  
query = query + " SELECT SCOPE_IDENTITY() as taskID ";
```

The query starts with an **insert** statement, listing out all the values; **taskCategoryID**, **userID**, **description**, **completed**, and **dateCreated**. Three of these values will be sent in by the UI and available as part of the URL's query string. The **completed** value will be set to "0" when the task is initially created; meaning it will be "not complete". The **dateCreated** value will be set to the current date and time, using the SQL Server **getDate()** function.

The final line of the query retrieves the unique ID of the task that was just created. This will be returned in the **recordset** variable of the database connection's result handler:

```
var dataQuery = databaseConnection.executeQuery(query,
function(result) {
    var queryStringData = {};
    queryStringData.taskID = result.recordset[0].taskID;
    queryStringData = JSON.stringify(queryStringData);
    mockQueryString = {};
    mockQueryString.filter = queryStringData;
    mockQueryString.callback = callback;
    getFilteredTasks(response, mockQueryString)
},
function(err) {
    responseObject.error = 1;
    responseHandler.execute(response, responseObject, callback);
}
);
```

The result handler makes use of the **getFilteredTasks()** method that we created in previous chapters to return the updated task. It manually creates a query string object with a filter property set to the **taskID**. Passing our fake query string object and the response object into the **getFilteredTasks()** method allows this method to do the rest of the work for us, and a **result** object is returned to the browser with no additional work needed in the **createTask()** method.

The third argument of the **executeQuery()** method is the error handler. It sets the **error** property on the **responseObject** object to "1", then returns it to the invoking client. The last thing to do in this file is to export the **createTask()** method:

```
exports.createTask = createTask;
```

Then, add the exported handler into the **ResponseHandler** file in the **com/dotComIt/learnWith/server** directory:

```
handlers["/taskService/createTask"] = taskService.createTask;
```

Restart the app, and you'll be ready to accept task creation service calls.

Testing Task Creation

To test the ability to create a task, just load this URL in your browser:

```
http://localhost:8080/taskService/createTask?  
taskCategoryID=1&  
userID=1&  
description=Created by Test Harness
```

Remove the line breaks, of course. The URL contains the file location of the request that was specified in the **ResponseHandler** file in the previous section. It also contains three URL arguments; the **taskCategoryID**, the **userID**, and the **description**. You should see a result in the browser similar to this:

```
{  
  "responseObject":  
  [  
    {  
      "taskCategoryID":1,  
      "description":"Created by Test Harness",  
      "dateScheduled":"","  
      "taskCategory":"Business",  
      "dateCompleted":"","  
      "taskID":13,  
      "dateCreated":"05\14\2016",  
      "completed":0,  
      "userID":1  
    }  
  ],  
  "error":0  
}
```

This response object returns an updated task object inside the **responseObject** instance. The **error** is set to "0".

Updating a Task

Next, we want to create a method for updating a task. Once again, open up the **TaskService** class in the **com/dotComIt/learnWith/services** directory.

Create a method stub for the **updateTask()** method:

```
function updateTask(response, queryString) {  
}
```

The **updateTask()** method is a URL handler and accepts a **response** object to send information back to the user, and a **queryString** object that contains the information that the user sent in to the method.

The method opens with some boilerplate code:

```
var responseObject = {};  
var callback = '';  
if(queryString.callback !== undefined){  
    callback = queryString.callback;  
}
```

This creates a generic return object—**responseObject**—and also saves the callback string, if available, in a **callback** variable.

Next make sure that the relevant URL variables are present in the query string:

```
if((queryString.taskCategoryID == undefined) ||  
    (queryString.taskID == undefined) ||  
    (queryString.description == undefined)){  
    responseObject.error = 1;  
    responseHandler.execute(response, responseObject, callback);  
} else {  
    // process query here  
}
```

The variables needed for this method are the **taskCategoryID**, the **taskID**, and the **description**. If none of those are defined, then the **error** response is sent back to the user. If all values are defined as “needed”, it is time to create the query:

```
query = "update tasks set ";  
if(queryString.taskCategoryID !== 0) {  
    query = query + " taskCategoryID = " +  
    queryString.taskCategoryID + ", ";  
} else {  
    query = query + " taskCategoryID = null, ";  
}  
query = query + " description='" + queryString.description + "' ";  
query = query + " where taskID = " + queryString.taskID;
```

The query represents an update statement. It starts with the **update** keyword, and then provides a list of columns to be updated and their new values. The **taskCategoryID** and description are updated. The **taskCategoryID** could be set to "null" if the value is "0", meaning the "All Categories" option was selected in the UI. The **taskID** property used to determine which task is updated with a "where" clause on the query.

Next, use the **dataConnection** object to execute the query:

```
var dataQuery = databaseConnection.executeQuery(query,
  function(result) {
    var queryStringData = {};
    queryStringData.taskID = queryString.taskID;
    queryStringData = JSON.stringify(queryStringData);
    mockQueryString = {};
    mockQueryString.filter = queryStringData;
    mockQueryString.callback = callback;
    getFilteredTasks(response, mockQueryString)
  },
  function(err) {
    responseObject.error = 1;
    responseHandler.execute(response, responseObject, callback);
  }
);
```

The result handler makes use of the existing **getFilteredTasks()** method to return the updated task. It manually creates a query string object with a **filter** property set to the **taskID**. Passing the fake query string object and the **response** object into the **getFilteredTasks()** method lets this method to do the rest of the work for us. A **result** object is returned to the browser with no additional work needed in the **createTask()** method.

The error handler adds the **error** property to the **responseObject** with a value of "1" to depict that there is an error. Then it returns it with the browser.

The last thing to do in this file is to export the **updateTask()** method:

```
exports.updateTask = updateTask;
```

Be sure to add the **updateTask()** handler into the **ResponseHandler** file in the **com/dotComIt/learnWith/server** directory:

```
handlers["/taskService/updateTask"] = taskService.updateTask;
```


Restart the app and you'll be ready to update tasks.

Testing Task Updates

To test the ability to create a task, just load this URL in your browser:

```
http://localhost:8080/taskService/updateTask?  
taskCategoryID=1&  
taskID=13&  
description=Updated by Test Harness
```

Remove the line breaks when you put together the URL. The URL contains the file location of the request that was specified in the **ResponseHandler** file in the previous section, which is "taskService/updateTask". It also contains three URL arguments; the **taskCategoryID**, the **taskID**, and the **description**. You should see a result in the browser similar to this:

```
{  
  "responseObject":  
  [  
    {  
      "taskCategoryID":1,  
      "description":"Updated by Test Harness",  
      "dateScheduled":"","  
      "taskCategory":"Business",  
      "dateCompleted":"","  
      "taskID":13,  
      "dateCreated":"05\14\2016",  
      "completed":0,  
      "userID":1  
    }  
  ],  
  "error":0  
}
```

This response object returns an updated task object inside the **responseObject** instance. The error is set to "0".

Access the Services

This section will examine the Angular code needed to talk to the NodeJS services for saving and updating tasks. We'll use a single method—**updateTask()**—for both saving and updating tasks from the Angular app. That method is in the **TaskService.ts** class in the **com/dotComIt/learnWith/services/nodejs** directory:

```
updateTask(task :TaskVO, user :UserVO): Observable<ResultObjectVO>
{
};
```

The method accepts two arguments; the task that is being created or updated, and the user who is doing the updates. The method returns an **Observable** commitment for a **ResultObjectVO**.

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.

First, we want to make sure that the **taskCategoryID** was selected:

```
if(!isNumeric(task.taskCategoryID)) {
    task.taskCategoryID = 0;
}
```

This uses the rxjs **isNumeric()** function, so be sure to import it:

```
import {isNumeric} from "rxjs/util/isNumeric";
```

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.

Next, create a parameter query string:

```
let parameters = "taskCategoryID" + "=" + task.taskCategoryID +
'&';
parameters += "description" + "=" + task.description + '&';
parameters += "callback" + "=" + "JSONP_CALLBACK" + '&';
```

The query string contains the **taskCategoryID**, the **description**, and the **callback**. All three items are required for both “create” and “update” calls.

Now, determine which service method to call:

```
let method : string = "createTask";
if (task.taskID) {
    method = "updateTask";
    parameters += "taskID" + "=" + task.taskID + '&';
} else {
    parameters += "userID" + "=" + user.userID + '&';
};
```

The code creates a variable named “method”. The default value is **createTask**, for creating a new task. If the **task’s taskID** value is defined, then change it to **updateTask**. This code also adds the optional parameters. If we are updating the task, then add the **taskID** to the parameter string. On the other hand, if we are creating a new item, add the **userID**.

Now, put together the final URL:

```
let url = SERVER + 'taskService/' + method + '?' + parameters;
```

This code uses the **server**, **method**, and **parameter** variables to piece together the URL. The final step in the method is to trigger the remote call:

```
return this.jsonp.request(url)
    .map((result) => result.json() as ResultObjectVO);
```

The Angular **jsonp** variable triggers the call. The Observable object 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. Finally, using the dot notation, the **map()** function is called to parse the JSON results into a **ResultObjectVO** before sending them back.

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 **taskcu.component.ts** file. This method is executed when the save button is clicked inside the task popup. This method calls the **updateTask()** method in the **TaskService** and processes the results. Here is the method outline:

```
onSave():void {
    this.taskUpdateError = '';
    this.taskService.updateTask(this.task,
this.userModel.user).subscribe(
    result => {
    },
    error => {
    }
);
}
```

First, the error message is cleared. The **updateTask()** service method returns an observer, and the **subscribe()** method is executed upon it. We line up two result methods; the first for success, and the second to handle errors. Upon failure, we just set the **taskUpdateError** value:

```
this.taskUpdateError = 'There was a problem saving the task.';
```

This will update the alert method in the popup, warning the user that something bad happened.

The result method is similar to what we've seen before:

```
if ( result.error) {
    this.taskUpdateError = 'There was a problem saving the task.';
    return;
}
this.activeModal.close(result.resultObject);
```

If an error is returned, then update the **taskUpdateError** so the user is notified. If no error exists, then the **activeModal** service is used to close

the dialog. The returned **task** object is passed as an argument to the `close` method. The next step is to execute methods when the popup is closed or dismissed.

Handle the `updateTask()` Result

It is time to implement the `close()` and `dismiss()` functions for the modal. Open up the `tasks.component.ts` file in the `com/dotComIt/learnWith/views/tasks` directory. Find the `openTaskWindow()` method. At the end, add the result handlers to the `modalRef`:

```
modalRef.result.then(  
  (result) => {  
  }  
)  
.catch(  
  (result) => {  
  }  
)  
;
```

The `modalRef.result` is a promise object, similar to the observable we used when processing service responses. The `then()` method will execute when the window is closed. The `catch()` method will execute when the window is dismissed. The `dismiss()` method can be kept blank; if the user cancels any updates in the modal, the app doesn't need to address any functionality.

The `close()` method needs some more work. There are two options. If we have a new task, we need to add it to the current tasks array in the `TaskGrid` component so that it will immediately display. If a task was updated, we need to update the tasks array in the `TaskGrid`.

We can check whether a task was created or updated by checking the task argument into the `openTaskWindow()` method. If it is a null, we created a new task:

```
if (!task) {  
  this.taskgrid.tasks.push(result[0]);  
}
```

If no task is defined, then add the result array onto the `taskgrid.tasks` property. Remember, earlier we exposed the `TaskGrid`'s properties using the `@ViewChild()` annotation.

If the task is defined, we are performing an update. This is done with a loop over that **TaskGrid**'s tasks array:

```
else {
  for (let index = 0; index < this.taskgrid.tasks.length; ++index) {
    if (this.taskgrid.tasks[index].taskID === result[0].taskID) {
      this.taskgrid.tasks[index].description =
result[0].description;
      this.taskgrid.tasks[index].taskCategory =
result[0].taskCategory;
      this.taskgrid.tasks[index].taskCategoryID =
result[0].taskCategoryID;
      break;
    }
  }
}
```

Instead of replacing the item, I had to change the properties individually or else the grid would not refresh. I would have preferred to reference the **tasks** array in the **TaskModel**, however that was not updating, either.

Final Thoughts

This is the process going on with the popup:

1. User loads app and logs in.
2. They click the “Edit Task” in the **TaskGrid**, or “New Task” button in the **TaskFilter**.
3. Events are emitted from their respective components up to the **TaskComponent**.
4. The **TaskComponent** 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.
5. If the user clicks the “cancel” button, then the **dismiss()** function is called; closing the popup and performing no other actions.
6. If the user performs edits and clicks the “save” button, then the popup calls the service to save—or update—the task.
7. The results come back and the popup is closed, passing the new task as an argument.
8. The **TaskComponent** executes the successful method for updating or creating the task. If a new task was created, that new task is added to the task grid. If the task is updated, the grid task properties are changed so that the grid can update its display to reflect new values.

The next chapter will implement the ability to schedule tasks.

Chapter 6: Scheduling Tasks

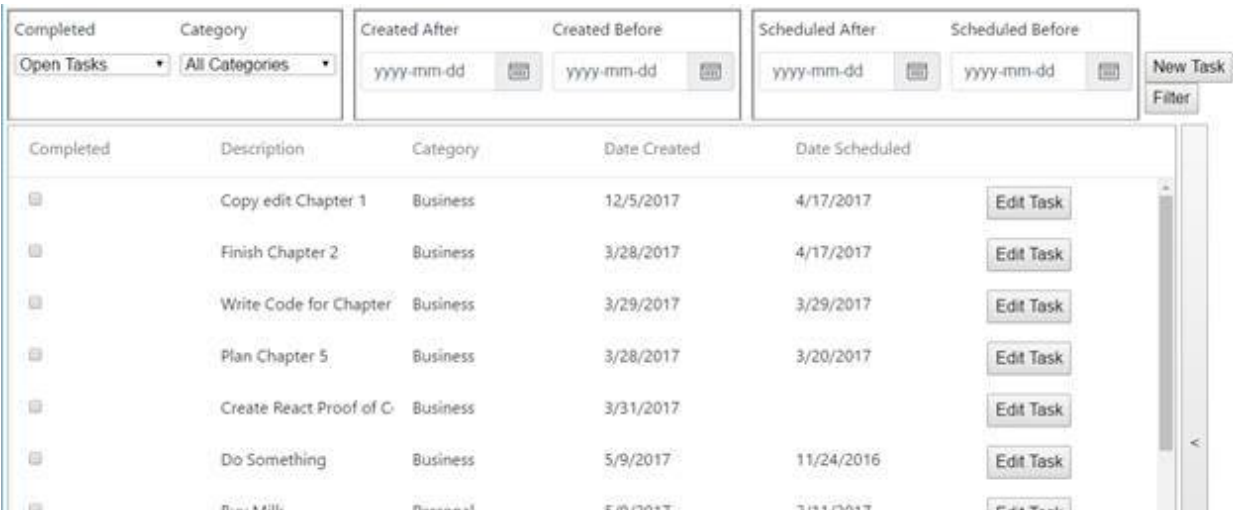
This chapter will implement the ability to schedule tasks inside an app. That means being able to assign a task to a specific day. It will show you the user interface for this, and discuss the services required. Then, it will show you how to wire everything up to make things work, and revisit the service method for loading task data. This chapter will also introduce a few new angular directives; **hidden** will show or hide aspects of the UI, **ngClass** will dynamically change style sheets, and **ngModelChange** will execute a function when the **ngModel** value changes.

Create the User Interface

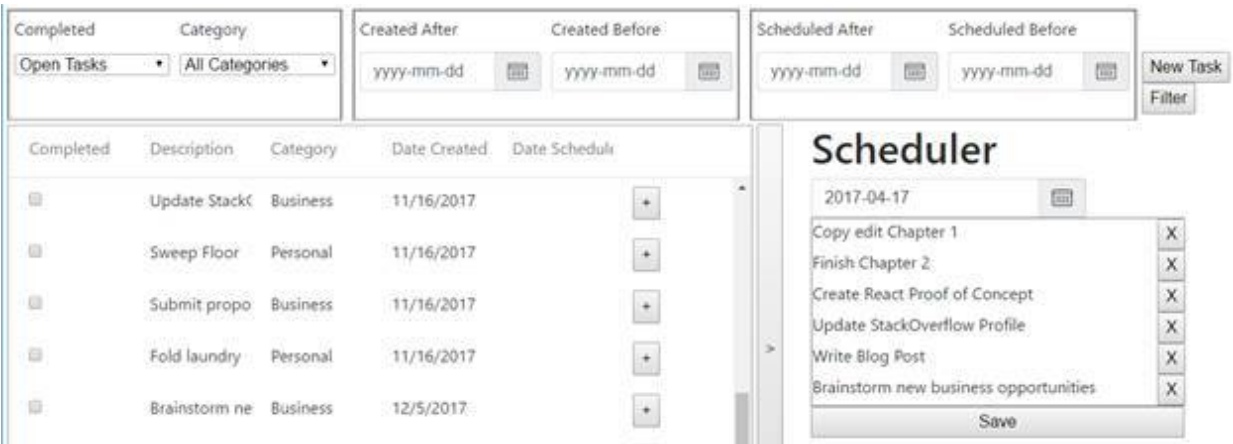
This section will show you the user interface for scheduling tasks. It will demonstrate to you the implementation, and we'll also revisit some of our past CSS Styles to tweak the layouts.

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 button:



The button to show the scheduler is on the right of the screen. After pressing the button, you'll see this:



The grid shrinks down to half the screen. Here, 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.

Create the `TaskScheduler` component

The `TaskScheduler` component will be fueled by three files; `taskscheduler.component.css`, `taskscheduler.component.html`, and `taskscheduler.component.ts`. Create all of them in the `com/dotComIt/learnWith/views/tasks` directory. Let’s start with the TypeScript file:

```
import {Component, OnInit} from '@angular/core';
import {TaskService} from "../../services/mock/task.service";
import {TaskVO} from "../../vo/TaskVO";
import {NgbDateStruct} from "@ng-bootstrap/ng-bootstrap";
```

This imports the `Component` and `OnInit` from `@angular/core`, the `NgbDateStruct` from `@ng-bootstrap/ng-bootstrap`, and our local `TaskService` and `TaskVO` libraries. Now, create the `@Component` definition:

```
@Component({
  selector: 'taskscheduler',
  templateUrl :
  './com/dotComIt/learnWith/views/tasks/taskscheduler.component.html',
  styleUrls: [
  './com/dotComIt/learnWith/views/tasks/taskscheduler.component.css' ]
})
```

The selector is named `taskscheduler`, and that is the tag we’ll use in HTML to show the component. The `templateUrl` and `styleUrl` values point to the empty HTML template and CSS files we just created.

Now, create the class:

```
export class TaskScheduler implements OnInit {
  schedulerDate : NgbDateStruct;
  schedulerError : string;

  constructor(private taskService :TaskService,
               private taskModel :TaskModel) {
  }

  ngOnInit(): void {
```

```

    }

    onSchedulerDateChange():void {
    }

    onTaskUnschedule(): void {
    };
}

```

The class is named **TaskScheduler**, and implements the **OnInit** interface. We'll use the **ngOnInit()** method to load the tasks scheduled for the current date.

Two properties are created on the class. The **schedulerDate** will contain the current selected date. The **schedulerError** will contain the text of an error to be displayed to the user, most likely coming from a service call.

The constructor injects the **TaskService** into the component and the **TaskModel**. We need to add two variables to the **taskmodel.ts** file in **com/dotComIt/learnWith/model** directory:

```

scheduledTasks: TaskVO[] = [];
addedTasks: TaskVO[] = [];

```

The **scheduledTasks** value will contain an array of tasks scheduled for the selected date. The **addedTasks** array will be tasks added to the selected date, but not yet scheduled. 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.

A method was added for when the selected scheduler date changes, and one for when a task was removed from the current date. For the moment that is it. We'll fill the method functionality in later on in this chapter.

Make sure you load this component in the **app.module.ts** from the **com/dotComIt/learnWith/main** directory. First, import it:

```

import {TaskScheduler} from
"../views/tasks/taskscheduler.component";

```

Then, add the **TaskScheduler** to the declarations:

```
declarations: [  
  AppComponent,  
  LoginComponent,  
  TasksComponent,  
  TaskGrid,  
  TaskFilter,  
  TaskCU,  
  TaskScheduler  
],
```

This makes the new component ready to use within the application.

Create the Scheduler Template

Let's turn our attention to the **taskscheduler.component.html** file in the **com/dotComIt/learnWith/views/tasks** directory. This template includes four separate parts. First is the text header which is trivial to create. Second is the date chooser template, which is almost identical to what we used 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 going over this. The final piece is the "save" button. We've seen a lot of Angular powered buttons, so this should be simple enough.

Start with a **div** wrapper:

```
<div class="taskSchedulerWrapper">  
  <h1>Scheduler</h1>  
</div>
```

I added a header too. Be sure to define the **taskSchedulerWrapper** in the **taskscheduler.component.css** file:

```
.taskSchedulerWrapper {  
  display: flex;  
  height: 100%;  
  flex-direction: column;  
}
```

This sets up this component for a Flexbox display similar to what we've used elsewhere in the app. It is interesting to note that Angular prevents similarly named styles across multiple components from interfering with each other.

Back to the HTML template. Add the error alert:

```
<div class="alert alert-danger" *ngIf="schedulerError">  
  <h2>
```

```

    {{schedulerError}}
  </h2>
</div>

```

We've seen this approach in previous chapters, so not much new content to point out here. Then, add in the Bootstrap **DatePicker**:

```

<form class="form-inline">
  <div class="form-group">
    <div class="input-group"
      <input class="form-control" placeholder="yyyy-mm-dd"
        name="schedulerDP" ngbDatepicker
        #schedulerDP="ngbDatepicker"
        [(ngModel)]="schedulerDate"
        (ngModelChange)="onScheduleDateChange()" >
      <div class="input-group-addon"
        (click)="schedulerDP.toggle()" >
        
        </div>
      </div>
    </div>
  </form>

```

We've used **DatePicker**'s in the chapter on filtering, so this should be familiar. There are a bunch of layout elements before getting to the actual input. The input is named **schedulerDP** and the **ngModel** binds to the **schedulerDate** property inside the component class. Something new here is the **ngModelChange** directive. Whenever the model changes, this method will be executed. We'll use this to trigger the loading of tasks scheduled for the current date.

Next, we want to show a list of all the tasks scheduled for the current date:

```

<div class="border-top-left-bottom-right">
  <div *ngFor="let task of taskModel.scheduledTasks"
    class="width100">
    <div class="horizontal-layout-94">{{task.description}}</div>
    <div class="horizontal-layout-4">
      <button (click)="onTaskUnschedule(task)">X</button>
    </div>
  </div>
</div>

```

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.

The first **div** has a new style which gives it a border all around. Put the style in the **taskscheduler.component.css** file:

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

The internal **div** uses the ***ngFor** directive to loop over the scheduled tasks. This directive is like an HTML for loop, and we used it to create select boxes in previous chapters. It tells Angular 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%  
}
```

As always, put the CSS information inside the **taskscheduler.component.css** file. This sets the width of the **div** to “100%” of its parent container. Inside the wrapper are two more **div** layers. The first one displays the task description. The task variable referenced inside the repeat loop refers to the task definition defined in the ***ngFor**. 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. The final two internal **div** layers have their own custom styles:

```
.horizontal-layout-94{  
    position: relative;  
    display: inline-block;  
    vertical-align: top;  
    width:94%;  
    height : 100%;  
}  
.horizontal-layout-4{  
    display: inline-block;  
    vertical-align: top;  
    width:4%;
```

```
height:100%;  
}
```

These two will be reused elsewhere, so I put them in **styles.css** in the **styles** directory. This would be a no-go if I were trying to optimize the components for reuse. However, in this case, I did not want to have duplicate versions of multiple styles.

For details on the style, set 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”. **Div** elements 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%” for the text, and 4% for the button. This leaves some room for padding in the display. Finally, the **height** is set to “100%”. This will expand the height of the grid to fill up the available space.

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.

Modifying the Main Screen

Let’s modify the main tasks screen to include the scheduler component and a button to expand or collapse the screen. Open the **tasks.component.html** file in the **com/dotComIt/learnWith/views/tasks** directory. This is the template that is displayed after the user logs in. It already displays the **TaskFilter TaskGrid** components.

Let’s review the current component:

```
<div class="wrapper">  
  <taskfilter class="taskFilter"  
(filterRequest)="filterRequest($event)"  
  (newTaskRequest)="newTask()"></taskfilter>  
  <div class="mainScreenContainer">
```

```

        <taskgrid #taskgrid (editTaskRequest)="editTask($event)">
</taskgrid>
    </div>
</div>

```

The whole component is wrapped in a **div**; used to set up the flexbox positioning. There is a **TaskFilter** component instance up top, and the **TaskGrid** instance is on the bottom. **TaskGrid** is wrapped in a **div** with the class, **mainScreenContainer**. Our new components will go inside this **div**; below the **TaskGrid**.

First, I'm going to wrap the **TaskGrid** in its own container:

```

<div [ngClass]="gridContainerStyle">
    <taskgrid #taskgrid (editTaskRequest)="editTask($event)" >
</taskgrid>
</div>

```

This introduces a brand new angular directive; **ngClass**. The **ngClass** directive allows you to set a style class on the **div** by using a variable defined in the component class. There are two different CSS classes that will be switched between—each representing a different width. The first is named **horizontal-layout-94**, which we defined earlier. This will be used when the component is in the expanded, or normal state. A similar style, **horizontal-layout-60** will be used when the **TaskGrid** is shortened in the scheduler state:

```

.horizontal-layout-30{
    position: relative;
    display: inline-block;
    vertical-align: top;
    width:30%;
    height:100%;
}

```

Create the **gridContainerStyle** variable inside the **TasksComponent** class so it has a default value:

```

public gridContainerStyle : string = 'horizontal-layout-94';

```

While we're at it, create two other variables required to maintain the state:

```

public schedulerState : boolean = false;
public schedulerShowButtonLabel : string = "<";

```


The **schedulerState** value will be used to show or hide the **TaskScheduler** component. The **schedulerShowButtonLabel** will be used as the text inside the button.

This expand/collapse button comes next in the file:

```
<div class="horizontal-layout-4">
  <button class="height100" (click)="onToggleScheduler()" >
    {{schedulerShowButtonLabel}}
  </button>
</div>
```

The button is in a **div** which uses the class **horizontal-layout-4**, created earlier in the chapter. 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 the variable inside the component we defined a few pages ago. It defaults to 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 button will execute the **onToggleScheduler()** method in the **TasksComponent** class. We'll examine it shortly.

The last element of the **mainScreenContainer div** is the scheduler component:

```
<div class="horizontal-layout-30" [hidden]="!schedulerState">
  <taskscheduler></taskscheduler>
</div>
```

The **div** is styled with a CSS class named **horizontal-layout-30**. This is similar to the other sizing classes we have used:

```
.horizontal-layout-30{
  position: relative;
  display: inline-block;
```

```
vertical-align: top;
width:30%;
height:100%;
}
```

The final element of the **div** is the **hidden** directive. This directive determines whether the **div** should be shown or hidden based on the **schedulerState** variable in the controller. The scheduler is hidden by default.

Clicking the Expand Button

What happens when the expand button is clicked? Well, a method in the **TasksComponent** is executed when this happens—**onToggleScheduler()**. This method will perform four 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 **schedulerState** variable, which will display or hide the scheduler component.
- Forces the **TaskGrid** to resize itself so columns fit into the available width.

All in all, the method is fairly simple:

```
onToggleScheduler(): void {
  if ( this.schedulerState === true) {
    this.schedulerState = false;
    this.gridContainerStyle = 'horizontal-layout-94';
    this.schedulerShowButtonLabel = "<";
  } else {
    this.schedulerState = true;
    this.gridContainerStyle = 'horizontal-layout-60';
    this.schedulerShowButtonLabel = ">";
  }
  setTimeout(() => this.taskgrid.taskGrid.recalculate(), 100);
}
```

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 needs to be called to load the tasks, but we’ll get to that later.

Finally, the **setTimeout()** is used to call the **recalculate()** method to resize the **taskGrid**. The **taskgrid** variable was already setup as a **@ViewChild** to the current component. We drill down three levels to get to the actual grid. I delayed the **recalculate()** call so that the grid would have time to resize before we forced it to redraw. Ideally, the grid should update automatically—as it does when the browser resizes—but at the time of this writing, we had to force it to prevent the grid columns from expanding beyond the grid container.

Adding the Schedule Button to the TaskGrid

In the expanded scheduler state, the task grid shows a “+” button in the last column instead of the edit button. This new button will add the current task into the scheduler. The button is defined as part of the **ngx-datatable** column template. This is the updated column:

```
<ngx-datatable-column>
  <ng-template let-row="row" ngx-datatable-cell-template>
    <button (click)="onEditTask(row)" [hidden]="schedulerState">
      Edit Task
    </button>
    <button (click)="onScheduleTaskRequest(row)"
[hidden]="!schedulerState">
      +
    </button>
  </ng-template>
</ngx-datatable-column>
```

It is in the **taskgrid.component.html** file from the **com/dotComIt/learnWith/views/tasks** directory.

The button uses the **hidden** directive to determine when it should be hidden or displayed. The “Edit” button and the “+” button both use the same variable, but with conditions reversed. This ties into a **schedulerState** variable. Add one to the **taskgrid.component.ts** file:

```
public schedulerState :boolean = false;
```

This variable is not inherently tied to the one in the **TasksComponent**. Go to the **onTogglerScheduler()** method in the **tasks.component.ts** file. Each time we set the **schedulerState**, we also want to drill down to the **taskgrid** to set the **schedulerState**. This will do it when **schedulerState** is being set to false:

```
this.schedulerState = this.taskgrid.schedulerState = false;
```

And this sets it to true:

```
this.schedulerState = this.taskgrid.schedulerState = true;
```

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 **click** handler to call the **onScheduleTaskRequest()** method when the button is clicked.

The purpose of the **click** 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 **onScheduleTaskRequest()** method now:

```
onScheduleTaskRequest(task:any):void {  
    this.taskModel.onScheduleTaskRequest(task);  
}
```

I decided to put the actual functionality inside of the **TaskModel**, so the **TaskGrid** component is just like a pass through. This is the **TaskModel** function:

```
onScheduleTaskRequest(task:TaskVO) {  
    let found :boolean = false;  
    for (let index :number = 0; index < this.scheduledTasks.length;  
index++) {  
        if (this.scheduledTasks[index].taskID === task.taskID) {  
            found = true;  
            break;  
        }  
    }  
    if (!found) {  
        this.scheduledTasks.push(task);  
        this.addedTasks.push(task);  
    }  
}
```

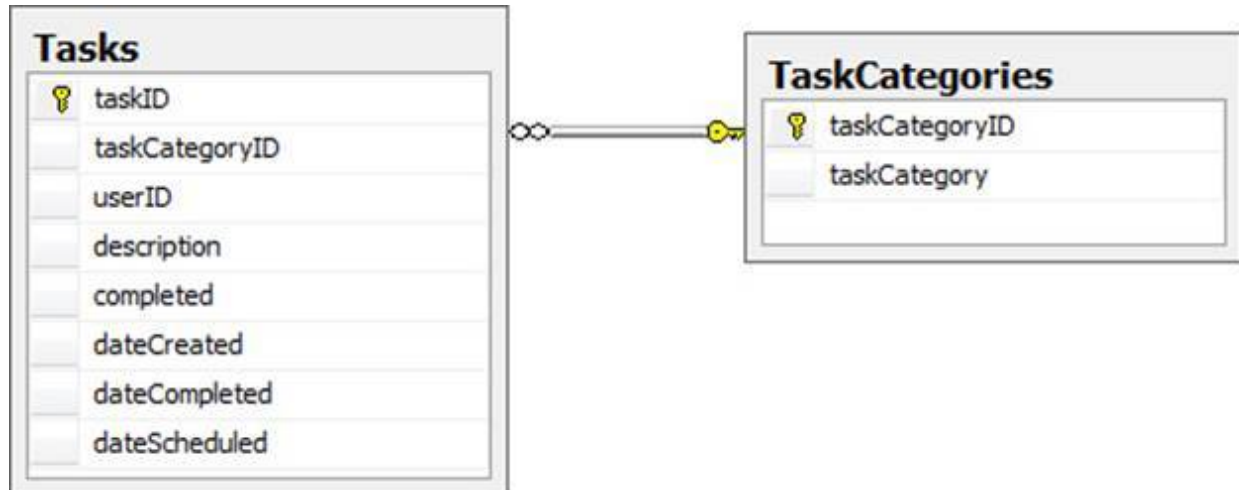
The first step in the method is to verify that the item being added to the **scheduledTasks** list is not already in there. If it is, it should not be added

again. This is done by looping over the **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.

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 delve into the NodeJS code for scheduling or canceling a single task. It will also introduce a method for scheduling a lot of tasks at once, and the URLs that will be used to call the service. For generic testing, we'll call the URL in the browser. Later on, we'll call them programmatically from Angular.

Revisit the `getFilteredTasks()`

For this chapter, we need to add one more filter to the code for loading up tasks. Open up the `TaskService.js` file from the `com/dotComIt/learnWith/services` folder. Look at the `getFilteredTasks()` method. This method creates a **query** based on multiple criteria and returns a list of all relevant tasks.

We need to add a new condition to this **query**; one for the **scheduledEqualDate**. The **scheduledEqualDate** will retrieve all the tasks scheduled on a certain day. This is the code:

```
if(json.scheduledEqualDate !== undefined) {
    if(firstOne) {
        query = query + "Where "
        firstOne = false;
    } else {
        query = query + "and "
    }
    query = query + " dateScheduled = '" + json.scheduledEqualDate
+ "' ";
}
```

This code makes the assumption that the **scheduledEqualDate** will already be formatted for use in the query before the service call is made. If the **scheduledEqualDate** value is undefined, then this condition is not added to the final query. Otherwise, the condition is added to the query. The **firstOne** variable is used to determine if this is the first query clause, or if the condition should be appended to other queries. I put this condition at the end of the query after the **scheduledEndDate** check, but before the order by statement.

Scheduling a Single Task

The first method to investigate is the one for scheduling a single task. Open up the **TaskService** class in the **com/dotComIt/learnWith/services** directory. Scroll to the end of the file and create the new handler method:

```
function scheduleTask(response, queryString) {  
}  
exports.scheduleTask = scheduleTask;
```

The **scheduleTask()** method signature follows the pattern of our previous handler methods, accepting a **response** and a **queryString**. I also added the line that exports the method so it can be called from other components. Now let's start filling in the method's implementation:

```
var resultObject = {};  
var callback = '';  
if(queryString.callback !== undefined) {  
    callback = queryString.callback;  
}
```

The method starts with the boilerplate code that defines the **resultObject** and takes note of the **callback**, if specified.

The next step is to check to make sure that the **taskID** is defined. If it isn't, an **error** response needs to be sent back to the browser:

```
if((queryString.taskID === undefined) ) {  
    resultObject.error = 1;  
    responseHandler.execute(response, resultObject, callback);  
} else {  
    // Complete request here  
}
```

This request requires two separate **query** string **parameters**. The **taskID** relates to the task being scheduled. The other is the **dateScheduled**. However, if the **dateScheduled** is left out, then it's database column will be set to "null"; essentially canceling the task. That is why we check for the **taskID** to be defined here, but do not check for the **dateScheduled**.

Assuming the **taskID** is defined, go into the **else** condition and create the **query**:

```
query = "update tasks set ";  
if(queryString.dateScheduled !== undefined) {  
    query = query + " dateScheduled = '" +  
    queryString.dateScheduled + "' ";  
}
```



```

} else {
    query = query + " dateScheduled = null ";
}
query = query + " where taskID = " + queryString.taskID;

```

The query is a typical SQL update statement. It sets the **dateScheduled** property based on the **taskID**. If the **dateScheduled** property is defined, then it is specified. Otherwise, the **dateScheduled** column is set to “null”.

Next, execute the query:

```

var dataQuery = databaseConnection.executeQuery(query,
function(result) {
    var queryStringData = {};
    queryStringData.taskID = queryString.taskID;
    queryStringData = JSON.stringify(queryStringData);
    mockQueryString = {};
    mockQueryString.filter = queryStringData;
    mockQueryString.callback = callback;
    getFilteredTasks(response, mockQueryString)
},
function(err) {
    responseObject.error = 1;
    responseHandler.execute(response, responseObject, callback);
},
);

```

Our custom **databaseConnection** object is used to run the query. We send in the query string, the **result** function, and a **failure** function. The **result** function creates a mock query string, then calls the **getFilteredTasks()** method to retrieve the single task and return it to the calling entity. The error handler sends an **error** response back to the client.

The last thing to change is in the **ResponseHandlers** class in the **com/dotComIt/learnWith/services** directory. Add this code:

```

handlers["/taskService/scheduleTask"] = taskService.scheduleTask;

```

This tells the app to call the **taskService.scheduledTask()** method when the "taskService/scheduledTask" is requested.

Testing Scheduling a Single Task

To test this, just start your node application, and then load some URLs in the browser. This URL will schedule the **taskID** of “1” for March 29, 2016:

```
http://127.0.0.1:8080/taskService/scheduleTask
?taskID=1
&dateScheduled=3/29/2016
```

You should get a result similar to this:

```
{
  "responseObject":
  [
    { "taskCategoryID":2,
      "description":"Get Milk",
      "dateScheduled":"03\29\2013",
      "taskcategory":"Personal",
      "dateCompleted":"",
      "taskID":1,
      "dateCreated":"03\27\2013",
      "completed":0,
      "userID":1
    }
  ],
  "error":0
}
```

You can cancel a task by removing the **dateScheduled** from the above URL:

```
http://127.0.0.1:8080/taskService/scheduleTask?taskID=1
```

You should see JSON results that look like this:

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

In the first set the **dateScheduled** had a value. In the second set, it did not, proving that the method can be used to cancel tasks.

Scheduling a Lot of Tasks

The second service method needed to support the UI for scheduling tasks is one that will schedule a lot of tasks at once. This method is named **scheduleTaskList()** and you can find it in the **TaskService** file in the **com/dotComIt/learnWith/services** directory. Here is the method signature and its import statement:

```
function scheduleTaskList(response, queryString) {  
}  
exports.scheduleTaskList = scheduleTaskList;
```

The method accepts a response object and a **queryString** object, as with all other handler methods we have seen in this book. The method starts with some boilerplate code; creating a **responseObject** object, and storing the **callback** value in a local variable:

```
var responseObject = {};  
var callback = '';  
if(queryString.callback != undefined){  
    callback = queryString.callback;  
}
```

This method requires two arguments; a **taskIDList**, and the **dateScheduled**. The **taskIDList** will be a comma-separated list of **taskIDs**. The **dateScheduled** argument will be the date that we need to schedule those tasks on. This method does not support canceling a lot of tasks at once because that functionality was not implemented in the UI code. As such, it is assumed that a **dateScheduled** property will be present in the **queryString**:

```
if((queryString.taskIDList == undefined) ||  
    (queryString.dateScheduled == undefined)){  
    responseObject.error = 1;  
    responseHandler.execute(response, responseObject, callback);  
} else {  
    // Execute Query Here  
}
```

If the **taskIDList** or the **dateScheduled** are missing from the **queryString**, then an **error** response is returned to the user. Otherwise, the **query** is created and executed, with the results being returned to the user.

First, create the **query**:

```
query = "update tasks set ";
query = query + "dateScheduled = '" + queryString.dateScheduled + "
' ";
query = query + "where taskID in (" + queryString.taskIDList + ")";
```

The query updates the **dateScheduled** property. Next, execute the query with the **databaseConnection** object:

```
var dataQuery = databaseConnection.executeQuery(query,
function(result) {
    responseObject.error = 0;
    responseHandler.execute(response, responseObject, callback);
},
function(err) {
    responseObject.error = 1;
    responseHandler.execute(response, responseObject, callback);
    return;
},
);
```

In the success handler, a **success** result is sent back to the UI. In this case, no modified data is included in the return request. It merely sends back the **error** property with a value of "0". When the **scheduleTaskList()** method is called, the UI already has everything it needs to update the UI based data, so the service does not need to return anything new. The error handler sends a result to the browser, with the **error** property of the **responseObject** set to "1".

The final thing we need to do is register the **scheduleTaskList()** method as a handler in the **ResponseHandlers** file of the **com/dotComIt/learnWith/server** directory:

```
handlers["/taskService/scheduleTaskList"] =
taskService.scheduleTaskList;
```

Restart your node application and you should be good to go.

Testing Scheduling a Lot of Tasks

You can test out this new method by loading a URL in your browser. Here is a sample URL:

```
http://127.0.0.1:8080/taskService/scheduleTaskList
?taskIDList=1,2,3
&dateScheduled=3/29/2013
```

This URL will tell the service to schedule the tasks with **taskID**'s—"1", "2", and "3"—for March 29, 2013. Load this in the browser, and you'll see this:

```
{  
  "error":0  
}
```

The **error** return "0" means that the code is working as expected.

Access the Services

This section will examine the Angular code needed to integrate the **scheduleTask()** and **scheduledTaskList()** services into our task manager application.

Use the `scheduleTask()` Service

You can use JSONP to access the NodeJS application server. Open the **TaskService** class in the **com/dotComIt/learnWith/services/nodejs** directory. Add a **scheduleTask()** method to the **taskService** object:

```
scheduleTask(task :TaskVO): Observable<ResultObjectVO> {
    let parameters = "taskID" + "=" + task.taskID + '&';
    if (task.dateScheduled) {
        parameters += "dateScheduled" + "=" + task.dateScheduled
+ '&';
    }
    parameters += "callback" + "=" + "JSONP_CALLBACK" ;
    let url = SERVER + 'taskService/scheduleTask?' + parameters;

    return this.jsonp.request(url)
        .map((result) => result.json() as ResultObjectVO);
};
```

This method's argument is the task being modified. The **parameter** object represents the **query** string of the service call. It adds the **taskID** and the **callback** onto the final string. If a **dateScheduled** is part of the **task** argument, then that property is added to the query string.

A URL is created using the **SERVER** constant, the service's method name, and the **parameter** string. Then, the **request()** method is called on the **jsonp** service. When results are returned, the **map()** function is executed to translate the results from **JSON** into a **ResultObjectVO** instance. The **Observable** object is returned, allowing the invoking code to specify the result and error handlers as needed.

Use the `scheduleTaskList()` Service

The final thing to look at for this chapter is to call the **scheduleTaskList()** method. The code is in the same **TaskService.js** class from the **com/dotComIt/learnWith/services/nodejs** directory:

```

scheduleTaskList(tasks :TaskVO[], schedulerDate:Date):
Observable<ResultObjectVO> {
    let datePipe : DatePipe = new DatePipe('en-US');
    let taskIDList = '';
    for (let index = 0; index < tasks .length; ++index) {
        taskIDList += tasks [index].taskID + ",";
    }
    taskIDList = taskIDList.substr(0,taskIDList.length - 1);
    let parameters = "taskIDList" + "=" + taskIDList + '&';
    parameters += "dateScheduled" + "=" +
        datePipe.transform(schedulerDate,
'shortDate') + '&';
    parameters += "callback" + "=" + "JSONP_CALLBACK" ;

    let url = SERVER + 'taskService/scheduleTaskList?' +
parameters;
    return this.jsonp.request(url)
        .map((result) => result.json() as ResultObjectVO);
};

```

The method accepts two arguments; an array of tasks to be updated, and the date that the tasks should be scheduled for. First, the method creates a comma-separated list of IDs by looping over the **tasks** array. The final comma is removed from the list.

Then, a **parameter** string is created to represent the query string on the request. The query string contains the **taskIDList**, the **dateScheduled**, and the **callback**. Next, the URL is created with the **server** constant, the endpoint, and the **parameters** string. After that, the **json** service is invoked. The **map()** function is called on the results of the request to transform the results into a **ResultObjectVO**. Finally, the **Observable** object is returned.

Wire Up the UI

This section will show you how to connect the Angular 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 load data for tasks scheduled on the specified day, add service calls to the **scheduleTaskList()** when the "save" button is clicked, and to **scheduleTask()** when the "delete" button is clicked.

Loading Tasks when Scheduler is Opened

We already have a method for loading tasks in the **TaskGrid** component. We want to mimic that approach inside the **TaskScheduler** component. Open up **taskscheduler.component.ts** from the **com/dotComIt/learnWith/views/tasks** directory. Create a new method named **loadTasks()**:

```
loadTasks(taskFilter:TaskFilterVO):void {
    this.schedulerError = '';
    this.taskService.loadTasks(taskFilter).subscribe(
        result => {
        }, error => {
        }
    );
}
```

The method first clears out the **schedulerError** variable, then calls the **loadTasks()** method on the **TaskService**. This returns an **Observable** object, and we are set up to handle both the failure condition, and the success condition using the TypeScript lambda notation.

The failure condition is the second method, after the comma. It is easy to handle:

```
this.schedulerError = 'We had an error loading tasks.';
```

It just sets the **schedulerError** value so a message is displayed to the user.

The success condition is bit more complex:

```
if ( result.error) {
    this.schedulerError = 'We could not load any tasks.';
    return;
}
```



```
this.taskModel.scheduledTasks = result.resultObject as TaskVO[];
this.taskModel.scheduledTasks =
this.taskModel.scheduledTasks.concat(this.taskModel.addedTasks);
```

The purpose of this method is to set the resulting array of tasks to the **scheduledTasks** array in the **taskModel** instance. If there are any **addedTasks** the user entered into the scheduler that have not yet been saved, they are added to the result.

The **resultObject** value is saved directly into the **scheduledTasks** array in the **taskModel**. The array method—**concat()**—is 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.

We want to load the tasks when the scheduler is initialized. Contrary to what we've done with our other components, we can't do this in **ngOnInit()** because the view is initialized before it is shown. Let's create a new method, which we can trigger from the **TasksComponent**:

```
initialLoad(): void {
  let taskFilter : TaskFilterVO = new TaskFilterVO();
  taskFilter.scheduledEqualDate = new Date();
  this.schedulerDate = {day:
taskFilter.scheduledEqualDate.getUTCDate(),
                        month:
taskFilter.scheduledEqualDate.getUTCMonth() + 1,
                        year:
taskFilter.scheduledEqualDate.getUTCFullYear()};
  this.loadTasks(taskFilter);
}
```

The code creates a new **TaskFilterVO** object and sets the **scheduledEqualDate** value to the current date. Remember that the class used by the **DatePicker** for the selected date is not a **Date** class; we need to create an instance of **NgbDateStruct**. This instance is not a class, but an interface, so we are fudging the approach a bit by creating a generic object with all the required properties. The day, month, and year properties of the **NgbDateStruct** are created by introspecting the **scheduledEqualDate** to get the valid values for them. The month must be incremented by 1, because the **Date** class uses a zero-base array for months with January being month 0. However, the **DatePicker** expects a one-base array, where

January is month 1. The last step in the method is to call the **loadTasks()** method, passing in the **taskFilter** value.

We need to make sure we tell the **TasksComponent** to execute this method whenever the scheduler is open. Open the **tasks.component.ts** file in the **com/dotComIt/learnWith/views/tasks** directory. Define the **TaskScheduler** as a **@ViewChild**:

```
@ViewChild(TaskScheduler)
private taskscheduler : TaskScheduler;
```

Be sure to import the **TaskScheduler**:

```
import {TaskScheduler} from "../taskscheduler.component";
```

Now scroll down to the **onToggleScheduler()** method:

```
if ( this.schedulerState === true) {
  // close scheduler code
} else {
  // open scheduler code
  this.taskscheduler.initialLoad(new Date());
}
```

To open the scheduler at the end of the “else” condition, call the **initialLoad()** method on the task scheduler. Send in a new **Date** object, initialized to the current day. That should get the tasks properly loading when the date is shown.

Loading Tasks when the Scheduler Date Changes

When the user selects a new scheduler date, the **onScheduleDateChange()** method in the **TaskScheduler** component executes. This is the method stub:

```
onScheduleDateChange(): void {
};
```

It is entirely possible that this will execute if an invalid date is entered. In this case, the **schedulerDate** value will have the value of a string. To make sure that we have an **NgbDateStruct** object, we’ll check for the existence of the year property:

```
if (this.schedulerDate.year) {
}
```

Inside the condition, define the **taskFilter**:

```
let taskFilter : TaskFilterVO = new TaskFilterVO();
```

Convert the **schedulerDate** to an actual **Date** object:

```
taskFilter.scheduledEqualDate = new Date(this.schedulerDate.month +  
'/' +  
                                     this.schedulerDate.day +  
'/' +  
                                     this.schedulerDate.year);
```

And then call the **loadTasks()** method:

```
this.loadTasks(taskFilter);
```

This should set us up for properly loading tasks, and holding onto tasks that have not yet been scheduled.

Implement the Delete Task from Scheduler Button

When the “X” button is clicked in the task schedule template, the **onTaskUnschedule()** method is called. We showed an empty stub earlier in this chapter, but now it’s time to implement it. 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. If the button is clicked and the task has not yet been associated with a **dateScheduled**, then just remove it from the scheduler display without any further processing.

Let’s look at the **onTaskUnschedule()** method:

```
onTaskUnschedule(task: TaskVO): void {  
    if (task.dateScheduled) {  
        task.dateScheduled = null;  
        this.scheduleTask(task);  
    } else {  
        this.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:

```
deleteTaskFromSchedule(task: TaskVO): void {
    let itemIndex : number =
this.taskModel.scheduledTasks.indexOf(task);
    if (itemIndex >= 0) {
        this.taskModel.scheduledTasks.splice(itemIndex, 1);
    }
    itemIndex = this.taskModel.addedTasks.indexOf(task);
    if (itemIndex >= 0) {
        this.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.

Now, look at the **scheduleTask()** method:

```
scheduleTask(task:TaskVO): void {
    this.schedulerError = '';
    this.taskService.scheduleTask(task).subscribe(
        result => {
        }, error => {
        }
    );
};
```

For the purposes of deleting a task, we can assume that the task’s **dateScheduled** property will already be updated to the most current. No additional processing of it is needed before calling the service method to schedule the task. This service call is setup similar to how we’ve done so in the past, with a **subscribe()** function that contains the result and error handlers.

The error function is always easier:

```
this.schedulerError = 'We had an error scheduling the tasks.';
```

It just sets the **schedulerError** value to be displayed to the user.

The **success()** method is more complex. It is probably the most complicated success handler we've created to date. Let's look at it in pieces. First, check the error:

```
if (result.error) {
  this.schedulerError = 'We could not remove the task from the
  schedule.';
  return;
}
```

This is standard. If there was an error, let the user know and then stop processing with the **return** statement.

Then, we want to replace the modified task with the item in the model:

```
this.taskModel.replaceTask(result.resultObject[0]);
```

I encapsulated the **replaceTask()** function into a method in the **TaskModel**. Look for it in **TaskModel.ts** in the **com/dotComIt/LearnWith/model** directory:

```
replaceTask (task:TaskVO) : void {
  for (let index :number = 0; index < this.tasks.length; ++index)
  {
    if (this.tasks[index].taskID === task.taskID) {
      this.tasks[index] = task;
      break;
    }
  }
}
```

This accepts a task as an argument. It loops over the **tasks** array stored in the **TaskModel**. When it finds the item, based on a **taskID** check, it will replace it, then break out of the loop. If the item isn't found, no replacement occurs. I encapsulated this method so it can be easily reused in the next chapter when completing tasks.

Back to the success method from the service call:

```
for (let index : number = 0; index <
this.taskModel.scheduledTasks.length;
  ++index) {
  if (this.taskModel.scheduledTasks[index].taskID ===
```

```

    result.resultObject[0].taskID) {
  this.deleteTaskFromSchedule(this.taskModel.scheduledTasks[index]);
}
}

```

It looks at 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()**:

```

onTaskListSchedule() {
  let localDate : Date = new Date(this.schedulerDate.month + '/' +
    this.schedulerDate.day + '/' +
    this.schedulerDate.year);

  this.taskService.scheduleTaskList(this.taskModel.scheduledTasks,
    localDate)

    .subscribe(
      result => {
      }, error => {
      }
    );
}

```

First, we create a local date instance using the **schedulerDate** property selected from the **DatePicker**. Remember the **DatePicker** does not use a **Date** object as the selected date, but just a wrapper with the month, year, and day. Then the **scheduleTaskList()** method is called in the **TaskService**. It passes in the list of **scheduledTasks** and the date. The service returns a promise object; which is used to queue the success and error handler functions.

The error handler is always simpler:

```

this.schedulerError = 'We had an error scheduling all the tasks.';

```

We've seen methods like this all throughout the app. The success handler code is more interesting. It will check for an error. If there is one, it will display a message to the user:

```
if (result.error) {
  this.schedulerError = 'We had an error scheduling all the
tasks.';
  return;
}
```

We've seen that before. If there is no error, it compares the **tasks** array displayed in the **TaskGrid** with the **scheduledTasks** array from the scheduler template to find the same item. When the task is found, 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:

```
for (let scheduledTaskIndex : number = 0;
  scheduledTaskIndex < this.taskModel.scheduledTasks.length;
  scheduledTaskIndex++) {
  for (let masterTaskIndex : number = 0;
    masterTaskIndex < this.taskModel.tasks.length;
    masterTaskIndex++) {
    if (this.taskModel.tasks[masterTaskIndex].taskID ===
this.taskModel.scheduledTasks[scheduledTaskIndex].taskID) {
      this.taskModel.tasks[masterTaskIndex].dateScheduled =
localDate;
      break;
    }
  }
}
```

The outer loop is over the **scheduledTasks** in the **TaskModel**, while the inner loop is over the **tasks** array from the **TaskModel**. The two respective tasks are then compared; looking for items where the **taskID** matches. 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**, **hidden**, and **ngModelChange**, while also introducing multiple 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, and the grid does not like being resized on the fly.

I hope you are having a great experience with this book. There are two chapters remaining. 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 created 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	
<input type="checkbox"/>	Finish Chapter 2	Business	3/28/2017	4/17/2017	Edit Task
<input type="checkbox"/>	Write Code for Chapter	Business	3/29/2017	3/29/2017	Edit Task
<input type="checkbox"/>	Plan Chapter 5	Business	3/28/2017	3/20/2017	Edit Task
<input type="checkbox"/>	Create React Proof of C	Business	3/31/2017		Edit Task

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 **taskgrid.component.html** file in the **com/dotComIt/learnWith/views/tasks** directory. The checkbox column is defined as an in-line column as part of the **ngx-datatable**:

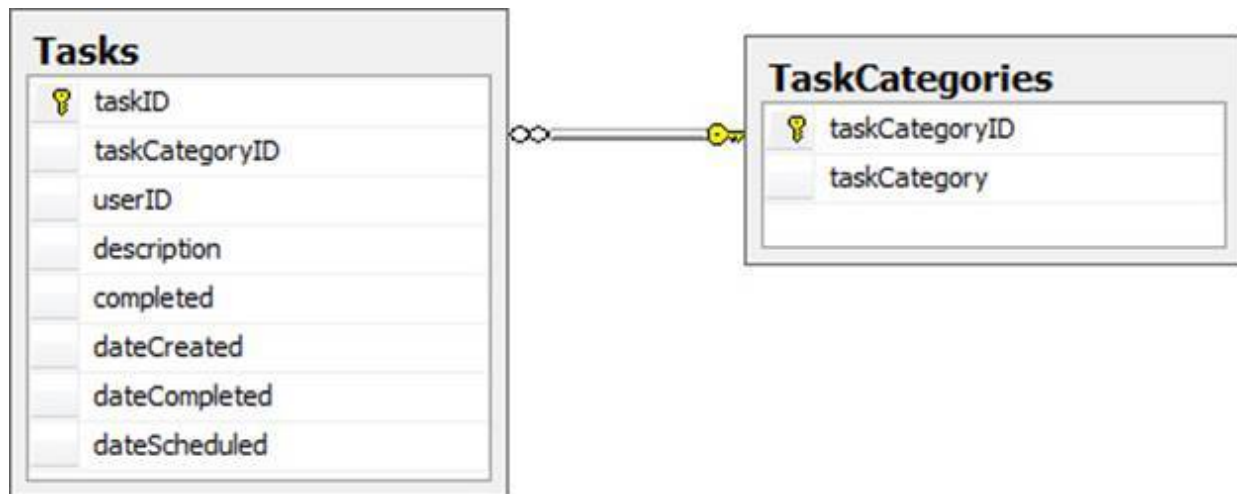
```
<ngx-datatable-column name="Completed" >
  <ng-template let-row="row" let-value="value" ngx-datatable-cell-
template>
    <input type="checkbox" [checked]="value"
      (click)="onCompletedCheckBoxChange (row)" />
  </ng-template>
</ngx-datatable-column>
```

This is a simple template inside the **ngx-datatable-column**. It adds a few new things. First, the **let-row** property is put on the template. This allows us to access the current row inside of the template. The **click** event handler

was added to the checkbox. When the checkbox is clicked, the **onCompletedCheckBoxChange()** method is executed inside the **TaskGrid** component, and the current row is passed as an argument. Notice that the **checked** property uses one-way binding with square brackets instead of two-way binding with a mix of square brackets and parentheses. The checked property is not open to two-way binding the same way an **ngModel** would be. Besides, we wouldn't want to change the task data in memory until we receive confirmation from the service that the data was updated. 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.

Creating the Service

This section will show you the NodeJS code required for marking a task “completed”.

The `completeTask()` Service Method

Open up the **TaskService** NodeJS package in the **com/dotComIt/learnWith/services** directory. Create the method signature of our new handler:

```
function completeTask(response, queryString) {  
}  
exports.completeTask = completeTask;
```

The **completeTask()** method is a new request handler and uses the same method signature of other request handlers we created. The two arguments in it are the **response** object and the **queryString** variable. I also included the line that exports the **completeTask()** method so it can be referenced in the **ResponseHandler** package. Next up comes some boilerplate code to set up the **resultObject** and a **callback** string:

```
var resultObject = {};  
var callback = '';  
if(queryString.callback != undefined) {  
    callback = queryString.callback;  
}
```

There should be two arguments defined in the query string; the **taskID** and a **completed** value. The **completed** value should be a Boolean value that determines whether the task—represented by the **taskID**—should become marked “completed”, or “not completed”. Perform this check:

```
if((queryString.taskID == undefined) ||  
    (queryString.completed == undefined)){  
    resultObject.error = 1;  
    responseHandler.execute(response, resultObject, callback);  
} else {  
    // Query processing code here  
}
```

If either the **taskID** or the **completed** property is undefined, then an **error** is sent back to the client.

If both values are defined, we create the query:

```

query = "update tasks set ";
if(queryString.completed == 'true'){
    query = query + " completed = 1, ";
    query = query + " dateCompleted = GETDATE() ";
} else {
    query = query + " completed = 0, ";
    query = query + " dateCompleted = null ";
}
query = query + " where taskID = " + queryString.taskID + " ";

```

The query updates the tasks table, and sets the **completed** property and the **dateCompleted** value. The value of the **completed** property and **dateCompleted** columns depend upon the “completed” argument in the **queryString**. If the **completed** value is “true”, then the task is marked as “completed” and the **dateCompleted** is set to the current date. If the **completed** value is “false”—or anything other than “true”—then the property is set to “0”, marking the task “not completed”, and the **dateCompleted** value is set to “null”.

Next, execute the query and process the results:

```

var dataQuery = databaseConnection.executeQuery(query,
    function(result) {
        var queryStringData = {};
        queryStringData.taskID = queryString.taskID;
        queryStringData = JSON.stringify(queryStringData);
        mockQueryString = {};
        mockQueryString.filter = queryStringData;
        mockQueryString.callback = callback;
        getFilteredTasks(response, mockQueryString)
    },
    function(err) {
        responseObject.error = 1;
        responseHandler.execute(response, responseObject, callback);
    }
);

```

The query is executed using our **DatabaseConnection** module. We now send a string representing the query to execute a result handler, and an error handler, into the **executeQuery** function of that module. The result handler function is an anonymous function defined in-line. The result expected from this service call is an updated task object representing the modified task. The **getFilteredTasks()** method is used to load the new task

and send the results back to the user. The error handler sends an error response back to the user.

The last step to implement this method is to add the handler into our **handler** object. Open up the **ResponseHandler** module in the **com/dotComIt/learnWith/server** directory. Add this line:

```
handlers["/taskService/completeTask"] = taskService.completeTask;
```

Restart your NodeJS app server, and you should be good to go.

Testing the completeTask() service

To test the **completeTask()** method you just need to load a URL in the browser and review the results. This URL can be used to mark a task “completed”:

```
http://127.0.0.1:8080/taskService/completeTask?  
taskID=1&completed=true
```

Run this and you should see results similar to this:

```
{  
  "responseObject":  
    [  
      {  
        "taskcategoryID":2,  
        "description":"Get Milk",  
        "taskCategory":"Personal",  
        "dateScheduled":"03\29\2013",  
        "dateCompleted":"05\26\2013",  
        "taskID":1,  
        "dateCreated":"03\27\2013",  
        "completed":true,  
        "userID":1  
      }  
    ],  
  "error":0.0  
}
```

You can change the value of the **completed** property in the URL to set the same task to “not complete”:

```
http://127.0.0.1:8080/taskService/completeTask?  
taskID=1&completed=false
```

This should give you these results:

```
{
  "responseObject":
  [
    {
      "taskcategoryID":2,
      "description":"Get Milk",
      "taskCategory":"Personal",
      "dateScheduled":"03\29\2013",
      "dateCompleted":"",
      "taskID":1,
      "dateCreated":"03\27\2013",
      "completed":false,
      "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.

Complete Tasks from Angular

The last step for this book is to create the JSONP service that can be used to call the NodeJS **completedTask()** method. Open up the **TaskService** file from the **com/dotComIt/learnWith/services/nodejs** directory.

Create a method named **completeTask()**:

```
completeTask(task:TaskVO) : Observable<ResultObjectVO> {
    let parameters = "taskID" + "=" + task.taskID + '&';
    parameters += "completed" + "=" + !task.completed + '&';
    parameters += "callback" + "=" + "JSONP_CALLBACK" ;
    let url = SERVER + 'taskService/completeTask?' + parameters;
    return this.jsonp.request(url)
        .map((result) => result.json() as ResultObjectVO);
}
```

The method accepts one argument; the **task** object which is being completed. This returns an **Observable** instance of a **ResultObjectVO**.

The first thing this method does is to create a URL query string. Two arguments go into the service method; the **taskID**, and the **completed** value. The **completed** value is reversed using the “not” operator, as the UI did not change the task object yet. So, if the **completed** value is “false”, then “true” should be sent to the service, completing the task. Conversely, if the **completed** value is “true”, then “false” should be sent to the service, opening the task. Required in the final URL argument is the **callback** argument. It is necessary in order to access the results from the remote service call. The last line of the code makes the service call using the **jsonp** service. Finally, the **map()** function is executed on the service result to convert the returned JSON into a **ResultObjectVO** before the invoking code executes its own result function.

Wire Up the UI

This section will show you how to integrate the service into the UI code to mark the task as 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:

```
onCompletedCheckBoxChange (task:TaskVO):void {
  this.taskLoadError = '';
  this.taskService.completeTask(task).subscribe(
    result => {
      }, error => {
      }
  );
}
```

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 **completeTask()** method returns a promise object, and a success or failure method will execute based on the **Observable** result.

The error handler just shows the user an error message:

```
this.taskLoadError = 'Error completing the task.';
```

The success method will check to see if an error is flagged in the **ResultObjectVO**:

```
if ( result.error ) {
  this.taskLoadError = 'Error completing the task.';
  return;
}
```

If an error is found, the user is notified via the **taskLoadError** value and the processing stops. Conversely, if there is no error, then we call the **replaceTask()** method we created in the previous chapter:

```
this.taskModel.replaceTask(result.resultObject[0]);
```

The **replaceTask()** method will take the updated task object and modify the underlying tasks array in the **taskModel**. This time, we're using it to update

the task object after its completed property has been changed.

If you experiment with this long enough as is, you'll notice that the grid object's is not properly updating immediately after the **taskModel** is updated. So, if you double click the completed checkbox, it will mark it completed both times instead of going from completed to incomplete, and back. The solution is to force the grid to recalculate, similar to what we did after resizing it when opening the scheduler:

```
setTimeout(() => this.taskGrid.recalculate(), 10);
```

I used a very short timeout for this because we already updated the value. I lost quite a few hours on this, looking for a fringe bug in the services before discovering the real issue.

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 has signed in. There are no new services to cover for this chapter, so the structure of it will be a bit different than previous ones. The bulk of it relates to conditionally modifying the UI. A new Angular directive is introduced: **disabled**.

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 you 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 **isUserRole()** 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.ts** file from the **com/dotComIt/learnWith/model** directory. Create the **isUserRole()** function:

```
isUserRole(roleToCompare:number):boolean {
    if (!this.user) {
        return false;
    }
    if (this.user.role === roleToCompare) {
        return true;
    }
    return false;
}
```

This function will be used throughout our implementation of the new functionality. First, it checks to make sure the user object is defined. If not, return “false”. This condition will occur if the app is reloaded from the internal screen. Then, it compares the user’s role value with the **roleToCompare**. If they are equal, return “true”. Otherwise, return “false”.

There are two separate roles in this app. For encapsulation sake, I defined them both as part of the **UserModel** class:

```
readonly TASKER_ROLE = 1;
readonly 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. Normally, I’d want to create these as constants. However, TypeScript does not support constants as class members, so I defined them as **readonly** values. By using

the convention of all caps, this should hopefully distinguish our constant variables from regular variables which use camel case.

Disabling the Completed Checkbox

The “completed” checkbox column was implemented as part of the **TaskGrid** component using a custom template. Open up the **taskgrid.component.ts** file from the **com/dotComIt/learnWith/views/tasks** directory. We need to inject the **UserModel** so it can be used from the view. First import it:

```
import {UserModel} from "../../model/usermodel";
```

Then, add it to the constructor:

```
constructor(private taskModel :TaskModel,  
            private taskService : TaskService,  
            private userModel :UserModel) {  
}
```

Now we can use the **userModel** instance inside the view template. Open up **taskgrid.component.html** and find the checkbox template:

```
<ngx-datatable-column name="Completed" >  
  <ng-template let-row="row" let-value="value" ngx-datatable-cell-  
template>  
    <input type="checkbox"  
  
[disabled]="userModel.isUserInRole(userModel.CREATOR_ROLE) "  
          [checked]="value"  
          (click)="onCompletedCheckBoxChange (row) " />  
  </ng-template>  
</ngx-datatable-column>
```

The only new attribute on the checkbox is the **disabled** 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, Angular 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
<input type="checkbox"/>	Copy edit Chapter 1
<input type="checkbox"/>	Finish Chapter 2
<input type="checkbox"/>	Write Code for Chapter

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
<input type="checkbox"/>	Copy edit Chapter 1
<input type="checkbox"/>	Finish Chapter 2
<input type="checkbox"/>	Write Code for Chapter

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 ***ngIf** directive in the **tasks.component.html** file, right on the **div** that contains the button:

```
<div class="horizontal-layout-4"
  *ngIf="userModel.isUserInRole(userModel.TASKER_ROLE)">
  <button class="height100" (click)="onToggleScheduler()" >
    {{schedulerShowButtonLabel}}
  </button>
</div>
```

We've used the ***ngIf** directive in the past to show or hide various error alerts. Here, the value of the component acts similar to the **disabled**

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%;
  height:100%;
}
```

The style on the task grid is set using the **ngClass** directive and the **TasksComponent’s gridContainerStyle** variable. You’ll need to set the **gridContainerStyle** to something other than the default. You can do so in the **ngOnInit()** method of the **tasks.component.ts** file:

```
if (this.userModel.isUserInRole(this.userModel.CREATOR_ROLE)) {
  this.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 use another ***ngIf** condition on the button template. Open up the **taskgrid.component.html** file from the **com/dotComIt/learnWith/views/tasks** directory. Find the last column in the **ngx-datatable**:

```
<ngx-datatable-column
*ngIf="userModel.isUserInRole(userModel.TASKER_ROLE)">
  <ng-template let-row="row" ngx-datatable-cell-template >
    <button (click)="onEditTask(row)" [hidden]="schedulerState">
      Edit Task
    </button>
    <button (click)="onScheduleTaskRequest(row)"
      [hidden]="!schedulerState">
```

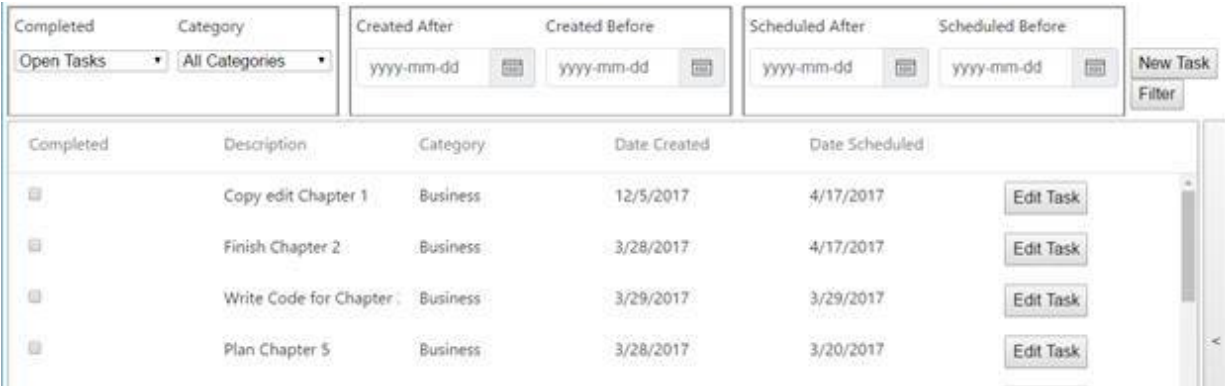
```

+
  </button>
</ng-template>
</ngx-datatable-column>

```

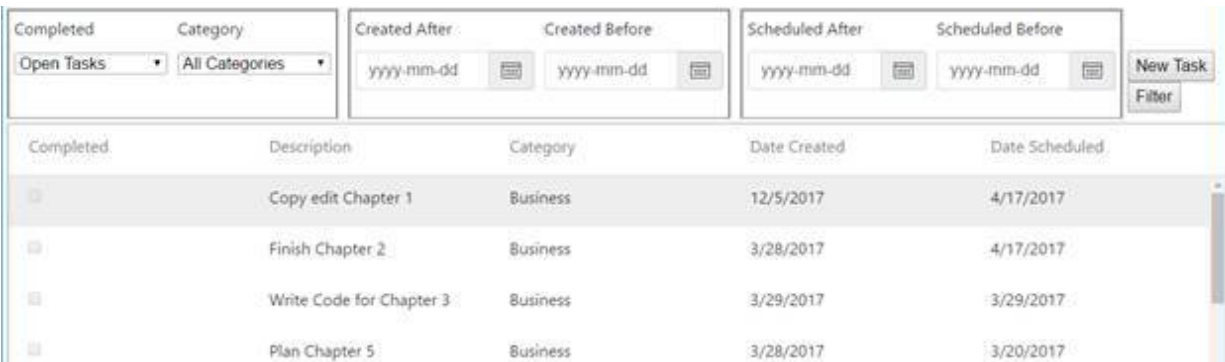
The ***ngIf** property tells Angular to show the item if the user is in the tasker role, and hide it if the user is not.

If the user logs in as a tasker, this is what they'll see:



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:



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 Angular. You should be ready to tackle your first Angular 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 continues the LearnWith series, focusing on Angular 4 with some hints of TypeScript. 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.

When you're there, be sure to sign up to get a free monthly tutorial from us.

If you need personal mentoring or have a custom consulting project, we'd love to help out. Please don't hesitate to reach out.

Send me an email at jeffry@dot-com-it.com and tell me how this book helped you become a success.