Contents

Chapter 1

Sample Calendar App

In this mini-book, we'll walk through the process of building a dynamic web app together and explain AngularJS concepts along the way.

We'll be building a complex TODO-style app. The final product will allow a user to log in and authorize the client-side app with Google. It will communicate with Google's calendar API to interact with the Google calendar.

Users will input (in English) the text of their schedule. The app will enable tagging, live search, grouping, and filtering, and it will include a custom calendar element.

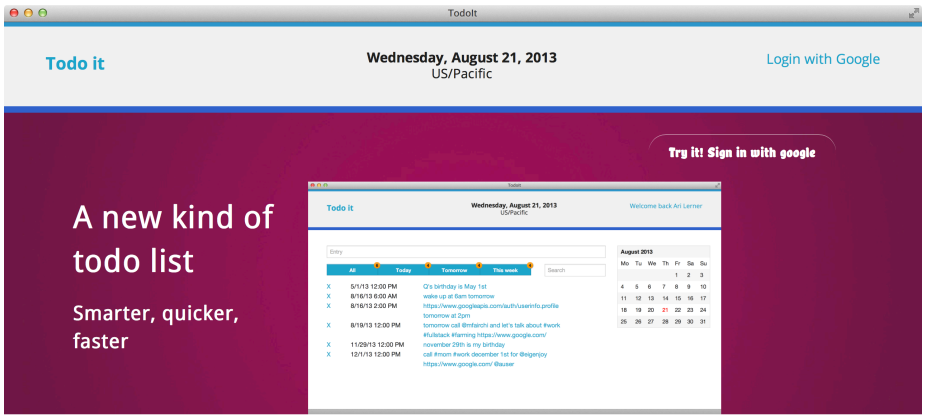

Figure 1.1: Finished app

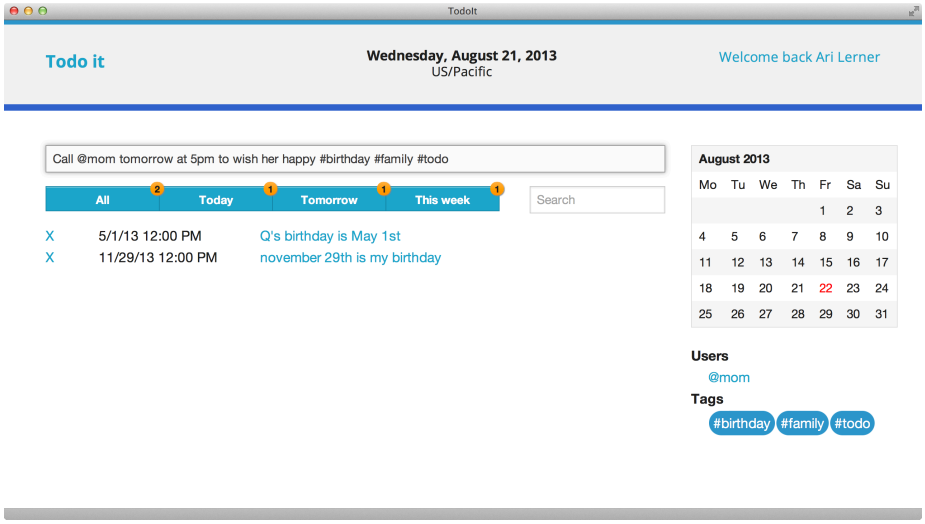

Figure 1.2: Finished app

> For the purposes of keeping this chapter focused on learning AngularJS, we will walk through how to build a raw AngularJS app without using any tools beyond AngularJS. We discuss production tools at length in our next steps chapter.

To build this app, we're going to start with a super simple server. In fact, we won't even need to write code for our server: We'll use the Python `SimpleHTTPServer` for the time being.

> Python's `SimpleHTTPServer` is a simple server written in Python that serves all the files in a single directory. It enables us to get started working quickly, rather than needing to write our own.

To start the app using the `SimpleHTTPServer`, we'll head to our terminal and change into the directory where we'll start building the app and type:

```
$ python -m SimpleHTTPServer 9000
```

Let's navigate to a directory where we'll keep the development of our code. We recommend creating a directory in a location that you navigate often. For instance, we recommend creating a directory in your Home folder (OSX/*nix) or creating a `Development` directory in `My Documents/` (Windows). Ultimately, the choice is up to you.

We'll refer to this top level directory as your app's root directory.

When we start building our app, we'll place an `index.html` in this directory and head to our browser at `http://localhost:9000`. In the browser, our `index.html` will be served through our local server.

To build the template for our app, you can do one of two things: Use `git` to clone the example repository that we've made available with the book or manually build the template.

Manually building our app

If instead, you'd prefer to build the app from scratch, the directory structure we will start off with will look like this:

```
/
  app/
    js/
    css/
    lib/
    img/
    views/
    index.html
```

Create these directories and files on your local machine. Open your terminal and type the following to create the initial structure:

```
$ mkdir -p app/{js,css,lib,img,views}
$ touch app/index.html
```

In building our app, we'll use the following directories to separate our different functions:

- **index.html**: our main application template
- **app/**: Our main app directory. This is where we'll serve our app from
- **app/js/**: where we'll store our application-specific JavaScript files that we write
- **app/css/**: where we'll store our CSS styles
- **app/lib/**: where we'll store any JavaScript library files that we download, like `angular.js`
- **app/views/**: where we'll store our AngularJS templates

Downloading angular

Once our directories are set, let's download angular.js and store the file in our `app/lib/` folder.

Additionally, make sure you grab the angular-route.js library and store that in the `app/lib/` folder.

The `angular-route.js` file is available at angularjs.org if you click on `extras` after clicking the download button.

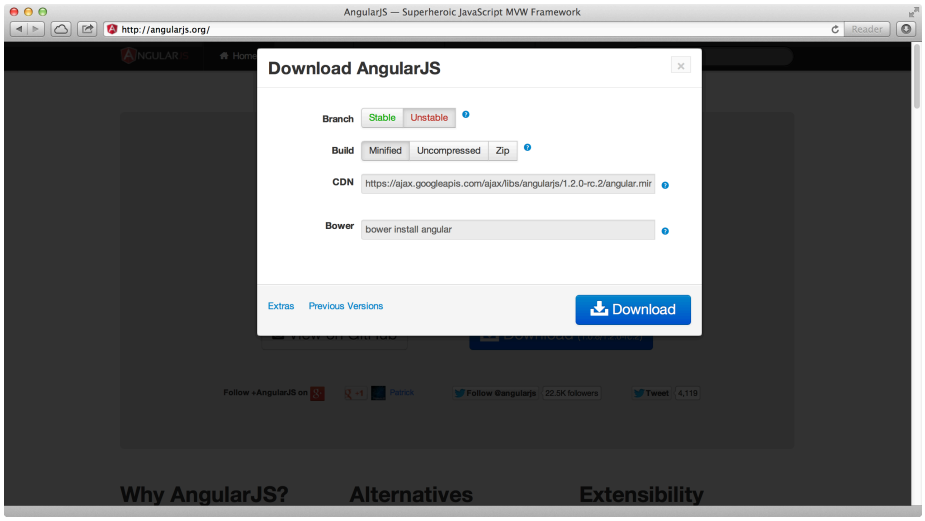

Figure 1.3: Download angular

Make sure you grab both `angular.js` and `angular-route.js` and store them in the `app/lib` folder.

In the demo application that comes with this book, we're using the foundation.zurb.com CSS framework. If you'd prefer to use another CSS framework, feel free to grab it and any JavaScript files that are associated with it and store them in the appropriate directories, but we recommend sticking with the book for the best experience.

Finally, when we're ready to go, let's fill out our `index.html` file like so:

```
<!doctype html>
<html lang="en" ng-app="myApp">
<head>
  <meta charset="utf-8">
  <title>TodoIt</title>
  <link rel="stylesheet" href="css/normalize.css"/>
  <link rel="stylesheet" href="css/app.css"/>
  <meta name="viewport" content="width=device-width">
</head>
<body>
  <!-- Library files -->
```

```
  <script src="lib/angular.js"></script>
  <script src="lib/angular-route.js"></script>
  <!-- App files -->
  <script src="js/filters.js"></script>
  <script src="js/controllers.js"></script>
  <script src="js/directives.js"></script>
  <script src="js/services.js"></script>
  <script src="js/app.js"></script>
</body>
</html>
```

Now that we have our `index.html` page filled, we need to create a few files that will contain our app.

Create these five JavaScript files in your `app/js/` folder:

- app/js/filters.js
- app/js/controllers.js
- app/js/directives.js
- app/js/services.js
- app/js/app.js

We can do so with another terminal command:

```
$ touch app/js/{filters.js,controllers.js,directives.js}
$ touch app/js/{services.js,app.js}
```

Each of these files will contain the different parts of our application that their titles suggest. We'll discuss what should go in each of these files as we continue through the book.

Booting it up

Finally, to see this app immediately in action, change into the `app/` folder in your terminal and start up the Python `SimpleHTTPServer`:

```
$ cd app/
$ python -m SimpleHTTPServer 9000
```

We'll head to the browser and navigate to `http://localhost:9000`. We should see an empty, but running app.

Chapter 2

Modules

As you may recall modules are how we define and organize our angularjs app. Let's create 5 different modules.

- A service module
- A directive module
- A filter modules
- A controller module
- A main app module

Our service module, `js/services.js` we'll make a services module. For the time being, we'll leave it relatively empty. We'll only declare an app version in the module.

js/services.js

```
angular.module('myApp.services', [])
  .constant('version', '0.0.1');
```

We'll contain our custom directives in the `js/directives.js` file. We'll cover directives in-depth in the directives chapter.

js/directives.js

```
angular.module('myApp.directives', []);
```

As we would expect, we'll contain our filters in the `js/filters.js` file. We'll cover filters in-depth in the filters chapter.

js/filters.js

```
angular.module('myApp.filters', []);
```

All of our controllers we will write will be placed in the controllers module in `js/controllers.js`. We will cover controllers in-depth in the controllers chapter.

js/controllers.js

```
angular.module('myApp.controllers', []);
```

Finally, we'll write our main app module that will depend upon all of the modules we just declared. When specifying a module the array `[]` is a list of all of the dependencies the app needs to be bootstrapped. Angular will not bootstrap the module until these dependencies have been found and met.

Our app depends on all of our modules to run, so we'll need to be sure to include them in the second parameter. We'll cover this *dependency injection* in the dependency injection chapter.

js/app.js

```
var app = angular.module('myApp', [
  'myApp.services',
  'myApp.directives',
  'myApp.filters',
  'myApp.controllers'
  ]);
```

If we navigate to our app, we'll notice that we haven't extended the functionality yet, but we're ready to start developing a robust app.

Starting our app

Now that we have a pretty good understanding of data-binding, `$scopes`, and controllers, let's start to add functionality to our application.

Provided we followed the steps in the introduction, we should have our development environment ready to go. If not, feel free to review the *introduction* chapter of *ng-book* for a step-by-step of setting up our development environment.

In our app, we have a header that will show the current date. To accomplish this, we'll create a variable on our `FrameController` called `today`. This `today` variable will hold the value of our date.

If our development environment is freshly created, we should have a `FrameController` controller in our `js/controller.js` file that looks like:

```
angular.module('myApp.controllers', [])
.controller('FrameController', ['$scope', function($scope) {
}]);
```

We'll use this `FrameController` as the parent controller to all of our other controllers. Let's add the variable called `today` on it and set it equal to today's date:

```
angular.module('myApp.controllers', [])
.controller('FrameController', ['$scope', function($scope) {
  $scope.today = new Date(); // Today's date
}]);
```

Now, in our view we can simply reference `today` and that value will be replaced with the value of `new Date();`. In our `index.html`, let's add a reference to today's date to see that change reflected in the view.

```
<!doctype html>
<html lang="en" ng-app="myApp">
<head>
  <meta charset="utf-8">
  <title>TodoIt</title>
  <link rel="stylesheet" href="css/vendor/normalize.css"/>
  <link rel="stylesheet" href="css/app.css"/>
  <meta name="viewport" content="width=device-width">
</head>
<body ng-controller="FrameController">
  {{ today }}
  <!-- Library files -->
  <script src="lib/angular.js"></script>
  <script src="lib/angular-route.js"></script>
  <!-- App files -->
  <script src="js/filters.js"></script>
  <script src="js/controllers.js"></script>
  <script src="js/directives.js"></script>
  <script src="js/services.js"></script>
  <script src="js/app.js"></script>
</body>
</html>
```

Now, when we can see we have a pretty ugly date in our view. We'll clean this up using `filters` later in this book.

Figure 2.1: First databinding

Next, we'll eventually allow a user to log in to our app using Google APIS, but for the time being we'll set our current user's name to a static name.

In our `FrameController` again, add the property of `name` to the scope:

```
angular.module('myApp.controllers', [])
.controller('FrameController', ['$scope', function($scope) {
  $scope.today = new Date(); // Today's date
  $scope.name = "Ari Lerner"; // Our logged-in user's name
}]);
```

And in our view, we can simply reference our user's name, like like before.

```
<!-- head and include files above -->
<body ng-controller="FrameController">
  <h1>{{ today }}</h1>
  <h2>Welcome back {{ name }}</h2>
<!-- include files -->
```

Now that our `FrameController` is holding on to some variables we'll use in multiple templates, we can create a child controller of the `FrameController` called the `DashboardController`.

First, let's edit our `js/controllers.js` file and add a new controller:

```
angular.module('myApp.controllers', [])
  .controller('FrameController',
```

```
    ['$scope', function($scope) {
      $scope.today = new Date();
      $scope.name = "Ari Lerner"; // Our logged-in user's name
    }])
  .controller('DashboardController',
    ['$scope', function($scope) {
    }]);
```

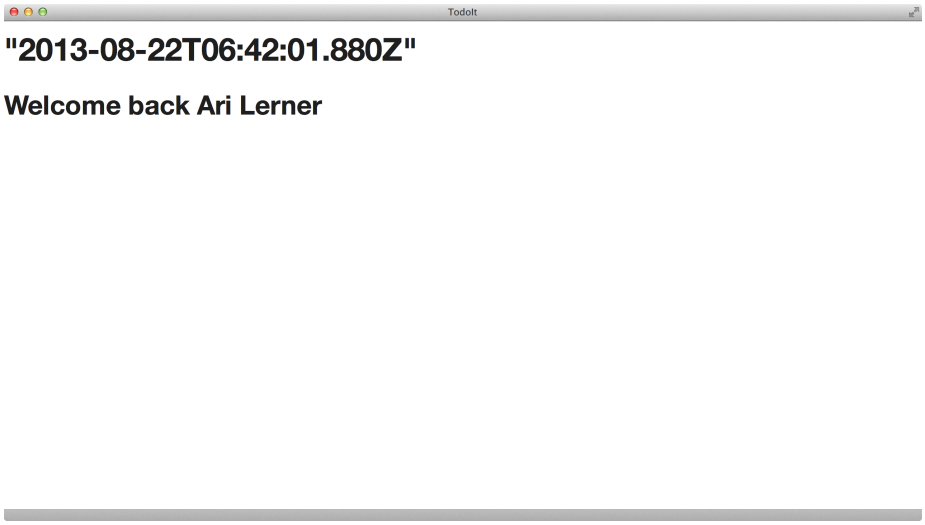

Figure 2.2: With two variables bound

Almost all of our application code up until this point will be nested in the `DashboardController`. First, we'll need to nest this controller in our view.

```
<body ng-controller="FrameController">
  <div ng-controller="DashboardController">
    <h1>{{ today }}</h1>
    <h2>Welcome back {{ name }}</h2>
  </div>
```

As we saw above, we can still reference the variables defined on our `FrameController` inside the DOM element of our `DashboardController`.

Now, let's create an input field where our users will type their todo items. To do this, we'll create a `entryInput` attribute on the `$scope` of the `DashboardController`.

This `entryInput` won't need to be initialized at the outset as it will be bound to the input field. If we were to initialize it, then the input field would be filled with the value of our `entryInput`.

First, in our `index.html` file, let's create an input field that we'll bind the `entryInput` variable:

```
<div ng-controller="DashboardController">
  <h1>{{ today }}</h1>
  <h2>Welcome back {{ name }}</h2>
  <input ng-model="entryInput" type="text" placeholder="New Entry" />
</div>
```

Anytime that the input box value is changed, the `$scope.entryInput` will change thanks to the bi-directional data-binding. Our updated controller looks like:

```
angular.module('myApp.controllers', [])
  .controller('FrameController',
    ['$scope', function($scope) {
      $scope.today = new Date();
      $scope.name = "Ari Lerner"; // Our logged-in user's name
    }])
  .controller('DashboardController',
    ['$scope', function($scope) {
      $scope.entryInput = undefined;
    }]);
```

> Because we want the input field to start out as undefined, we do **not** need to reference it in the `DashboardController`, as we did above for the `name` and `today` attributes. It is good practice, however to create variables inside the controller to be explicit.

In the following snippets, we'll be setting up the functionality of our app the will allow users to input their friends into the todo input box.

We'll *watch* the `entryInput` field and parse the the value entered for dates, #hashtags, links, and @users.

> Note: we'll move this functionality into a `$filter` to keep our controllers clean.

Looking at our current view, we haven't changed the html

```
<div ng-controller="DashboardController">
  <h1>{{ today }}</h1>
  <h2>Welcome back {{ name }}</h2>
  <input ng-model="entryInput"
         type="text"
         placeholder="New Entry" />
</div>
```

Also suppose that we've got a list of friends that are associated with the current user on the controller's scope:

```
$scope.users = {
  "ari": {
    "twitter": "@auser"
  },
  "nate": {
    "twitter": "@eigenjoy"
  }
};
```

We'll set up a `$watch` on the `entryInput` value that will track any changes that happen on it. When the value changes, we'll want to try to match any user that is set in the input box with tokens that start with a `@` symbol, such as: `@ari`.

As we previously set up, we'll only need to make sure we *inject* the `$parse` service into our controller:

```
.controller('DashboardController',
['$scope', '$parse', function($scope, $parse) {
  // Setting up a watch expression to watch the entryInput
  $scope.$watch('entryInput', function(newVal, oldVal, scope) {
    if (newVal !== oldVal) {
      // newVal now has the latest updated version
    }
  });
}]);
```

On change, let's look through the input text box and find any tokens that start with the `@` symbol using the built in regex feature of javascript called `match()`:

```
$scope.$watch('entryInput', function(newVal, oldVal, scope) {
  if (newVal !== oldVal) {
    // Look for any part of the string that starts with @
    var strUsers =
      newVal.match(/[@]+[A-Za-z0-9_]+/g),
      i;

    // If any part of the string starts
    // with @, then we
    // will have a list of those tokens
    // in strUsers
    if (strUsers) {
      // We'll loop through our users and parse the $scope
```

```
      // looking for user
      for (i = 0; i < strUsers.length; i++) {
        // Found user in the form @[user]
        var user = strUsers[i];
      }
    }
  }
});
```

Now that we are looping through any user that we may find and storing that user into a local variable of `user`. Let's modify the loop to use the `$parse` service to look through the current scope for the user:

```
for (i = 0; i < strUsers.length; i++) {
  // Found user in the form @[user]
  // Remove the @ symbol
  var user = strUsers[i],
  cleanUser = user.slice(1),
  // In here, we'll look up the cleanUser
  // in the local scope users object.
  // For instance, if the user
  // inputted "@ari" in the input, this
  // would try to find
  // users.ari in the local scope.
  parsedUser = $parse("users." + cleanUser)(scope);

  if (parsedUser) {
    // A user was found on our scope in
    // the "users" object
  } else {
    // No user was found on our scope in
    // the "users" object
  }
}
```

From here, we can assume we have the user in the parsedUser object if it's found on the local "users" object. If the `parsedUser` is undefined, it means it could not find the user on the `scope`. If it is found, then `parsedUser` will contain the object, if not, then it will be undefined.

> Note that this example is simply set up to demonstrate how to use the `$parse` service. A cleaner, more performant method of doing this would be to create a method on the scope that looks up the user, rather than appending a string as we've done above.

From here, we can choose to do what we want with the `parsedUser` object. In our case, we'll store it when we build the parse `$filter`.

In our app, we'll create several filters that will enable us to control the expected behavior of the app.

- blankIfNegative - a filter that ensures only the dates in the calendar's month appears
- excludeByDate - filters out any dates that do not belong in a date keep filter (such as today, tomorrow, next week, etc.)
- parseEntry - parses the `entryInput` field and pulls out any interesting tags, users, urls, and dates that the user writes into the entry field
- searchFilter - a search filter that enables us to *live search* the current list of events

In our app, we'll nest all of our filters in the `myApp.filters` module, as is best practice:

```
angular.module('myApp.filters', []);
```

We'll need to ensure that this module is set as a dependency for our app, we'll add this `myApp.filters` as a runtime dependency in our `myApp` module definition (in our `js/app.js` file):

```
angular.module('myApp', [
  'myApp.filters'
]);
```

blankIfNegative filter

We'll use the `blankIfNegative` filter in the calendar HTML that will prevent any negative values from showing up in the calendar table.

As we have not yet created our calendar's directive, this `filter` will be used in our HTML to show a space if the value passed in is non-positive. For instance:

```
<h2>{{ 15 | blankIfNegative }}</h2> <!-- 15 -->
<h2>{{ -15 | blankIfNegative }}</h2> <!-- -->
```

To create this filter, we'll use the `filter()` API:

```
angular.module('myApp.filters', [])
  .filter('blankIfNegative', function() {
    return function(input) {
```

```
      if (input <= 0) return ' ';
      else return input;
    }
  });
```

excludeByDate filter

In the main dashboard view of our app, we're repeating over the list of our events in our Google calendar. Since we have a few date filters as objects in our scope, we'll want to filter by our date.

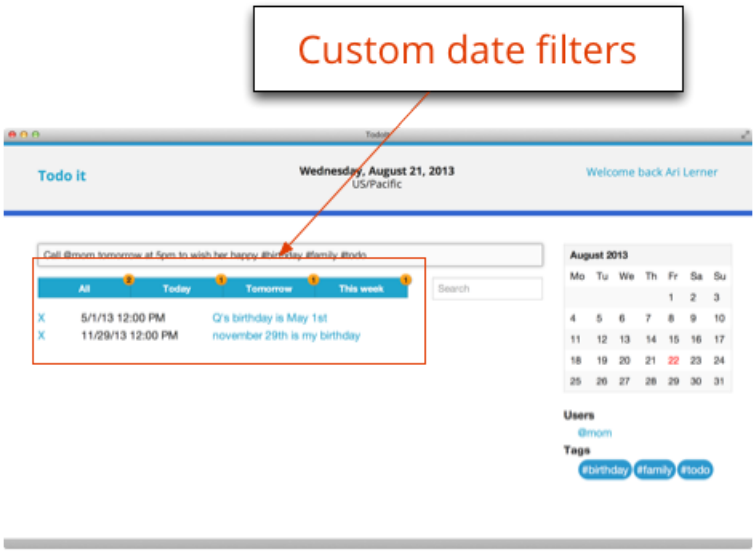

Figure 2.3: Custom date filters

In our HTML, we have buttons that are set to change a specific filter of a before-by date. Clicking on one of the filters will change the `keepDate` property on our `$scope` object.

Let's look at this process a bit deeper. The HTML we'll use looks similar to the following:

```
<ul>
  <li><a ng-click='keepDate=dateFilters["all"]'>
    All
  </a></li>
  <li><a ng-click='keepDate=dateFilters["today"]'>
    Today
```

```
  </a></li>
  <li><a ng-click='keepDate=dateFilters["tomorrow"]'>
    Tomorrow
  </a></li>
</ul>
```

Clicking on these links will update the `keepDate` property on our `DashboardController`. Obviously we need to set up our `dateFilters` property on our scope. Since these dates won't change often, we can set them as a single property value on our scope.

> To really be efficient, we can move the `dateFilters` into a service.

```
.controller('DashboardController',
  ['$scope',
  function($scope) {
    var date = new Date();
    $scope.dateFilters = {
      'all': 'all',
      'next week': new Date(date.setDate(date.getDate() + 7)),
      'tomorrow': chrono.parseDate('tomorrow at 11:59pm'),
      'today': chrono.parseDate('today at 11:59pm')
    }
}]);
```

> Note that for this function to work properly, we'll need to ensure we include the `chrono` library at the head of our HTML document.

With the backend set, we can implement this filter in the HTML by setting it to filter against the list of events in the standard lists filter manner:

```
<ul>
  <li ng-repeat="events in events |
                  excludeByDate:keepDate">
    {{ event.summary }}
  </li>
</ul>
```

With this set, we'll be using the `excludeByDate` filter in our view. The `excludeByDate` filter will receive the entire array of events as a parameter as well as a second parameter of the filter date.

The function itself will:

- check if the date passed in is 'all'. If so, it will return the entire set

- Return only the events who's `start.dateTime` is before the filterBy date

```
.filter('excludeByDate', function() {
  return function(arr, date) {
    if (date === 'all') return arr;
    var newArr = [];
    angular.forEach(arr, function(evt) {
      var evtDate = new Date(evt.start.dateTime);
      if (evtDate <= date)
        newArr.push(evt);
    });
    return newArr;
  }
});
```

One thing to note is that filters run a lot on almost all of the `$digest` loops. We can achieve this same effect with slightly better performance and more control if we move this filter function to the controller. Luckily, this is not difficult to do.

Instead of calling this custom filter in the view, we'll use the standard `filter` function in the view.

```
<ul>
  <li ng-repeat="events in events |
                 fitler:excludeByDate">
    {{ event.summary }}
  </li>
</ul>
```

Inside the `DashboardController` we can change the function as it will be called once for each element and return true if we want to keep the date in the list or false if we do not:

```
$scope.excludeByDate = function(input) {
  if ($scope.keepDate === 'all') {
    return true;
  } else {
    return new Date(input.start.dateTime).getTime() <
                $scope.keepDate.getTime();
  }
}
```

These two implementations of this filter process are effectively equivalent.

parseEntry filter

The last filter we'll need in our app is the `parseEntry` filter. This filter, unlike our other two will be called directly from our controller. Instead of returning a true or false condition for display purposes, we'll use this filter to parse our `entryInput` text.

The filter itself will take two parameters, a single value of the text inputed into the `newEntry` text and a javascript object of known users.

```
.filter('parseEntry', ['$parse', function($parse) {
  return function(val, users) {
    var i = 0;
    var data = { raw: val };
    if (val) {
      // Our parsing functions will go here
    }
}]);
```

Now inside our `parseEntry` filter, if we are given a value that makes sense, we can dissect the text. As the majority of this code is simple javascript, we won't cover it in detail, but the source is attached for completeness. Note that we are returning an object that potentially has the keys:

- raw - the raw entry
- tags - any text prepended with `#`
- users - any text prepended with `@`
- date - the chrono parsed date

```
.filter('parseEntry', ['$parse', function($parse) {
  return function(val, users) {
    var i = 0;
    var data = {
      raw: val
    };

    if (val) {
      // Find urls
  var strUrls =
    val.match(/[A-Za-z]+:\/\/[A-Za-z0-9-_]+\.[A-Za-z0-9-_:%&~\?\/.=]+/g),
          urls = [];

      if (strUrls) {
        for (i = 0; i < strUrls.length; i++) {
          urls.push(strUrls[i]);
```

```
      }
val = val
  .replace(/[A-Za-z]+:\/\/[A-Za-z0-9-_]+\.[A-Za-z0-9-_:%&~\?\/.=]+/g, '');
      data['urls'] = urls;
    }

    // Find tags
    var strTags = val.match(/[#]+[A-Za-z0-9_]+/g),
        tags = [];

    if (strTags) {
      for (i = 0; i < strTags.length; i++) {
        tags.push(strTags[i]);
      }
      val = val.replace(/[#]+[A-Za-z0-9_]+/g, '');
      data['tags'] = tags;
    }

    // Find users
    var strUsers = val.match(/[@]+[A-Za-z0-9_]+/g),
        users = [];

    if (strUsers) {
      for (i = 0; i < strUsers.length; i++) {
        var user = strUsers[i];

        if (users) {
          var parseVal = $parse("users." + user.slice(1))(users);
          if (typeof(parseVal) === 'undefined') parseVal = user;
        }
        else {
          var parseVal = user;
        }

        users.push(parseVal);
      }
      val = val.replace(/[@]+[A-Za-z0-9_]+/g, '');
      data['users'] = users;
    }

    // Finally, parse the date
    var date = chrono.parseDate(val);
    if (date) {
      data['date'] = date;
    }
  }
```

```
    return data;
  }
}])
```

Chapter 3

Creating our-calendar directive

```
app.directive('ourCalendar', function() {
  return {
    restrict: 'A',
    require: '^ngModel',
    scope: {
      ngModel: '='
    },
    template: '<table class="calendar"></table>'
  }
});
```

Going back to our calendar directive, we'll need to pass in a date with our directive from which to display. To do that, we could put in a date via text, but that will be pretty ugly. Instead, we'll *bind* it to the surrounding controller of 'date'.

```
<div our-calendar ng-model="date"></div>
```

Our directive now looks like this:

```
app.directive('ourCalendar', function() {
  return {
    restrict: 'EA',
    require: '^ngModel',
    scope: {
      ngModel: '='
```

```
    },
    template: '<table class="calendar"></table>'
  }
});
```

Drawing the calendar

To draw the calendar, we'll need to fill out out template. Luckily, the template is not that complex:

> We're using the \ at the end of each line to allow for us to be able to break up the template into multiple lines.

```
app.directive('ourCalendar', function() {
  return {
    restrict: 'EA',
    require: '^ngModel',
    scope: {
      ngModel: '='
    },
    template: '<table class="calendarTable">\
      <thead>\
        <tr>\
        <td class="monthHeader" colspan="7">\
          {{ monthName }} {{ year }}</td>\
        </tr>\
      </thead>\
      <tbody>\
        <tr>\
          <td ng-repeat="d in weekDays">{{ d }}</td>\
        </tr>\
        <tr ng-repeat="arr in month"> \
          <td ng-repeat="d in arr track by $index"\
              ng-class="{currentDay: d == day}">\
          {{ d }}\
          </td>\
        </tr>\
      </tbody>\
    </table>'
  }
});
```

Now, in order to actually render our template with the date, we'll have to set up the binding with the `link` function. We'll set up a `watch` on the `ngModel`

that will watch for changes on the date. That way, anytime the date changes in the containing controller, the calendar can reflect that change.

Controller option

In a directive, when we set the controller option, we are creating a controller for the directive. This controller is instantiated before the pre-linking phase.

The pre-linking and post-linking phases are executed by the compiler. The pre-link function is executed before the child elements are linked, while the post-link function is executed after. It is only safe to do DOM transformations after the post-link function.

We are defining a `controller` function in our directive, so we don't need to define either of these functions, but it is important to note that we cannot do DOM manipulations in our controller function because the controller is instantiated before the pre-link phase, thus we have no access to the DOM yet.

What does a controller function look like? Just like any other controller. Let's see what it looks like when we inject the `$http` service in our controller (using the `bracket` injection notation):

```
app.directive('ourCalendar', function() {
  return {
    restrict: 'EA',
    require: '^ngModel',
    scope: {
      ngModel: '='
    },
    template: '<table class="calendarTable">\
      <thead>\
        <tr>\
        <td class="monthHeader" colspan="7">\
          {{ monthName }} {{ year }}\
          </td>\
        </tr>\
      </thead>\
      <tbody>\
        <tr>\
          <td ng-repeat="d in weekDays">{{ d }}</td>\
        </tr>\
        <tr ng-repeat="arr in month"> \
          <td ng-repeat="d in arr track by $index"\
              ng-class="{currentDay: d == day}">\
          {{ d }}\
          </td>\
```

```
        </tr>\
      </tbody>\
    </table>',
    controller: ['$scope', '$http', function($scope, $http) {
      $scope.getHolidays = function() {}
    }]
  }
});
```

Note that in our link function, we have access to all of the attributes that were declared in the DOM element. This will become important in a minute when we go to customize the directive.

We're not making any ajax or api calls in this chapter, we won't actually create our `getHolidays` function, but it's good to note that this is how we can make `http` requests inside our directive.

```
<div our-calendar ng-model="date"></div>
```

Let's start building our `link` method that will actually build and draw the calendar on screen.

The `$watch` function will register a callback to be executed whenever the result of the expression changes. Inside the `$digest` loop, every time AngularJS detects a change, it will call this function. This has the side effect that we cannot depend on state inside this function. To counter this, we'll check for the value before we depend on it being in place.

Here is our new `$watch` function:

```
scope.$watch(attrs.ngModel, function(date) {
  if (date) {
    // we're ready to process the date
  }
});
```

Every time the *date* property on our scope changes, our function declared in the `watchExpression` will fire.

Here's what we have so far:

```
app.directive('ourCalendar', function() {
  return {
    restrict: 'EA',
    require: '^ngModel',
    scope: {
```

```
      ngModel: '='
    },
    template: '<table class="calendarTable">\
      <thead>\
        <tr>\
        <td class="monthHeader" colspan="7">\
          {{ monthName }} {{ year }}</td>\
        </tr>\
      </thead>\
      <tbody>\
        <tr>\
          <td ng-repeat="d in weekDays">{{ d }}</td>\
        </tr>\
        <tr ng-repeat="arr in month"> \
          <td ng-repeat="d in arr track by $index"
              ng-class="{currentDay: d == day}">\
          {{ d }}\
          </td>\
        </tr>\
      </tbody>\
    </table>',
    controller: ['$scope', '$http', function($scope, $http) {
      $scope.getHolidays = function() {}
    }],
    link: function(scope, ele, attrs, ctrl) {
      scope.$watch(attrs.ngModel, function(newDate, oldDate) {
        if (!newDate)
          newDate = new Date(); // If we don't specify
                                // a date, make the
                                // default date today
        });
    }
  }
});
```

Adding in our drawing code isn't particularly interesting or angular specific, so we'll include it here:

```
app.directive('ourCalendar', function() {
  // Array to store month names
  var months = new Array('January',' February', 'March', 'April', 'May',
    'June', 'July', 'August', 'September', 'October', 'November', 'December');
  // Array to store month days
  var monthDays = new Array(31, 28, 31, 30,
    31, 30, 31, 31, 30, 31, 30, 31);
```

```
// Array to store week names
var weekDay = new Array('Mo', 'Tu', 'We', 'Th', 'Fr', 'Sa', 'Su');
return {
  restrict: 'EA',
  require: '^ngModel',
  scope: {
    ngModel: '='
  },
  template: '<table class="calendarTable">\
    <thead>\
      <tr>\
      <td class="monthHeader" colspan="7">\
        {{ monthName }} {{ year }}\
      </td>\
      </tr>\
    </thead>\
    <tbody>\
      <tr>\
        <td ng-repeat="d in weekDays">{{ d }}</td>\
      </tr>\
      <tr ng-repeat="arr in month"> \
        <td ng-repeat="d in arr track by $index" \
          ng-class="{currentDay: d == day}">\
        {{ d }}\
        </td>\
      </tr>\
    </tbody>\
  </table>',
  controller: ['$scope', '$http', function($scope, $http) {
    $scope.getHolidays = function() {}
  }],
  link: function(scope, ele, attrs, c) {
    scope.$watch(attrs.ngModel, function(date) {
      if (!date) date = new Date();
      var week_day, counter, i, curr_week;

      var day = date.getDate(),
          month = date.getMonth(),
          year = date.getFullYear();
      scope.days_in_this_month = monthDays[month];
      scope.monthName = months[month];

      scope.currentWeek = 0;

      scope.month = {};
```

```
        var thisMonth = new Date(year, month, 1),
            firstDay = new Date(thisMonth.setDate(1)).getDay(),
            weeksOfMonth =
              Math.ceil((firstDay + scope.days_in_this_month)/ 7) + 1;

        scope.weekDays = weekDay;
        // First week
        curr_week = 0;

        scope.month[curr_week] = [];
        for (week_day = 0; week_day < thisMonth.getDay(); week_day++) {
          scope.month[curr_week][week_day] = week_day * -1;
        }
        week_day = thisMonth.getDay();
        for(counter=1; counter <= scope.days_in_this_month; counter++)
        {
          week_day %= 7;

          if (week_day == 0) {
            curr_week += 1;
            scope.month[curr_week] = [];
          }

          scope.month[curr_week].push(counter);

          week_day += 1;
        }

        while(scope.month[curr_week].length < 7) {
          scope.month[curr_week].push(counter * -1);
        }

        scope.day = day;
        scope.year = year;
      });
    }
  }
});
```

Now that we have our calendar being drawn, let's give our directive some customizable features.

For instance, our calendar right now features the full name of the calendar in the header. However, if we are using this calendar in a place that does not have a lot of room on the page, we might want to hide the full length and only show a shortened version of it instead.

We have access to the attrs (the normalized list of the attributes that we define in our HTML), we can simply look there first; otherwise we can set a default to see what how we want to show the header.

Let's change the setting of the `$scope.monthname` to:

```
if (attrs.showshortmonth) {
  scope.monthName = months[month].slice(0, 3);
} else {
  scope.monthName = months[month];
}
```

Now, we'll only show the first three characters of the name of the month if we include the showshortmonth as an attribute. When we invoke the directive, we'll use the attribute:

```
<div our-calendar showshortmonth='true'></div>
```

Now our calendar's header will change to show the shortened 3-character calendar header.

Transclude option

Although the name sounds complex, transclusion just refers to compiling the content of the element and making the source available to the directive. The transcluded function is pre-bound to the calling scope, so it has access to the current calling scope.

Looking at an example, let's change the invocation to:

```
<div our-calendar ng-model="date">
  <h3>On this day...</h3>
</div>
```

To get this to show up in our template, we'll need to use a special directive called `ngTransclude`. This is how the template knows where to place the custom HTML. Let's modify the template in our directive to include the custom content:

```
template: '<table class="calendarTable">\
  <thead>\
    <tr><td class="monthHeader" colspan="7">\
      {{ monthName }} {{ year }}</td>\
    </tr></thead>\
  <tbody>\
```

```
<tr><td ng-repeat="d in weekDays">{{ d }}</td></tr>\
<tr ng-repeat="arr in month"> \
  <td ng-repeat="d in arr track by $index"\
    ng-class="{currentDay: d == day}">\
  {{ d }}\
  </td>\
</tr>\
<tfoot><tr>\
<td colspan="7" ng-transclude></td>\
</tfood></tr></table>',
```

Additionally, we'll have to tell AngularJS that our directive will be using transclusion. To do that, we'll have to set the transclude option either to:

- true, which will take the content of the directive and place it in the template
- 'element', which will take the entire defined DOM element including the lower priority directives (see notes on directive priority order below)

Set the transclude option to true, as we only want the content of our directive to show up in our template:

```
transclude: true
```

Next steps

We've covered how to build a directive from a very basic level to a higher level of complexity. Hopefully you've gained some confidence and knowledge about how to move forward in the future and provide and build your own custom directives.

The entire source of our calendar directive is here:

```
angular.module('myApp.directives', [])
.directive('ourCalendar', [function() {
  // Array to store month names
  var months = new Array('January',' February', 'March', 'April', 'May',
    'June', 'July', 'August', 'September', 'October', 'November', 'December');

  // Array to store month days
  var monthDays = new Array(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);

  // Array to store week names
  var weekDay = new Array('Mo', 'Tu', 'We', 'Th', 'Fr', 'Sa', 'Su');
```

```
return {
  require: '^ngModel',
  restrict: 'EA',
  transclude: true,
  scope: {
    ngModel: '='
  },
  template: '<table class="calendarTable">\
    <thead>\
      <tr><td class="monthHeader" colspan="7">\
        {{ monthName }} {{ year }}</td>\
      </tr></thead>\
    <tbody>\
      <tr><td ng-repeat="d in weekDays">{{ d }}</td></tr>\
      <tr ng-repeat="arr in month"> \
        <td ng-repeat="d in arr track by $index"\
          ng-class="{currentDay: d == day}">\
        {{ d | blankIfNegative }}\
        </td>\
      </tr>\
      <tfoot><tr>\
      <td colspan="7" ng-transclude></td>\
      </tr></table>',
  link: function(scope, ele, attrs, c) {
      scope.$watch(attrs.ngModel, function(date) {
        if (!date) date = new Date();
        var week_day, counter, i, curr_week;

        var day = date.getDate(),
            month = date.getMonth(),
            year = date.getFullYear();
        scope.days_in_this_month = monthDays[month];

        if (attrs.showshortmonth) {
          scope.monthName = months[month].slice(0, 3);
        } else {
          scope.monthName = months[month];
        }

        scope.currentWeek = 0;

        scope.month = {};

        var thisMonth = new Date(year, month, 1),
            firstDay = new Date(thisMonth.setDate(1)).getDay(),
            weeksOfMonth =
```

```
            Math.ceil((firstDay + scope.days_in_this_month)/ 7) + 1;

        scope.weekDays = weekDay;
        // First week
        curr_week = 0;

        scope.month[curr_week] = [];
        for (week_day = 0; week_day < thisMonth.getDay(); week_day++) {
          scope.month[curr_week][week_day] = week_day * -1;
        }
        week_day = thisMonth.getDay();
        for (counter = 1; counter <= scope.days_in_this_month; counter++)
          {
          week_day %= 7;
          // We are at the start of a new week
          if (week_day == 0) {
            curr_week += 1;
            scope.month[curr_week] = [];
          }

          scope.month[curr_week].push(counter);
          week_day += 1;
        }

        while(scope.month[curr_week].length < 7) {
          scope.month[curr_week].push(counter * -1);
        }

        scope.day = day;
        scope.year = year;
      });
  }
}
```

Now to use this directive in-practice, we'll only have to tell the view where we want it placed and what to *bind* our `ngModel` to in the backend. To wrap all this back in the view, we'll set it as an attribute in the DOM element where we want it placed.

```
<div our-calendar showShortMonth="true" ng-model='newEntry.date'>
  <h3>Today</h3>
</div>
```

The `newEntry.date` in the above example is bound to the main input field where our users will enter their todo entry.

Now that we've got the handle of building a directive under our belt, we're going to use another directive to setup a custom form validation.

Integrating form validation in our app

In our app, we have a single input field. This input field that requires that a date is filled out. Without a date, we won't be able to place an event on the calendar. In order to add a custom validation, we'll need to build a directive just as we did above.

```
angular.module('myApp.directives')
.directive('ensureHasDate', [function() {
  // Our form validation for ensuring we have a date
  // will go in here
}]);
```

To parse dates from our input field, we'll use the fantastic `chrono` library. To include the chrono library into our app, we'll need to download it and store it in our `lib/` directory.

```
cd public/lib
curl -O https://raw.github.com/berryboy/chrono/master/chrono.min.js
```

Then we'll need to make sure we include this as a library in our `index.html` page. As we did previously with all of our library scripts, we'll just need to make sure that this is included at the bottom of our page:

```
<script src="lib/chrono.min.js"></script>
```

Now we can start to write our validation directive.

In our validation directive, we're going to use the chrono library to parse the text and find a date inside of it. If it finds a date, then we'll consider the form valid and set it as valid. If it doesn't not find a date, then we'll consider the input text to be invalid.

The first step in our directive is to *require* that the form has access to the `ngModel` and to create an *isolate* scope on that form validation with a binding to the parent's `ngModel`. In code, this looks like:

```
angular.module('myApp.directives')
.directive('ensureHasDate', [function() {
  return {
    require: '^ngModel',
```

```
    scope: {
      'ngModel': '='
    }
  }
}]);
```

Once our form gets loaded into the view, we'll want to attach a `$watch` to watch for the `ngModel` to be populated. Remember, this is because the directive will get loaded at compile-time and `ngModel` won't be available until the run-time.

```
angular.module('myApp.directives')
.directive('ensureHasDate', [function() {
  return {
    require: '^ngModel',
    scope: {
      'ngModel': '='
    },
    link: function(scope, ele, attrs, ctrl) {
      scope.$watch(attrs.ngModel, function(newVal) {
        if (newVal) {
          // We are in the runtime and ngModel has been populated
        } else {
          // At initialization stage
        }
      });
    }
  }
}]);
```

With this `$watch` in place, we can now create the validation that checks if there is a date or not in the string. The chrono library will return a `null` object if no date is found and a Date object if one is found. Using this fact, we can check if the date that's returned is a null object.

To set the validity on the form element, we'll use the `$setValidity` method on the directive's controller. This `$setValidity` method will set not only the validity of the input field, but it will set it on the form in general.

```
angular.module('myApp.directives')
.directive('ensureHasDate', [function() {
  return {
    require: '^ngModel',
    scope: {
      'ngModel': '='
    },
```

```
    link: function(scope, ele, attrs, ctrl) {
      scope.$watch(attrs.ngModel, function(newVal) {
        if (newVal) {
          // We are in the runtime and ngModel has been populated
          var date = chrono.parseDate(newVal);
          ctrl.$setValidity('hasDate', date !== null);
        } else {
          // Without a date, the form is always invalid
          ctrl.$setValidity('hasDate', false);
        }
      });
    }
  }
}]);
```

The `$setValidity` method will take care of telling AngularJS that the form is invalid. It will also let AngularJS set the proper classes for both the invalid form as well as the form itself.

Integrating with Google

Up until this moment, we've been working locally, without integration to Google. Although we'll have a fully-featured calendar application, we don't have any persistence, which is rather unimpressive.

Let's start building in Google calendar integration.

Including Google

In order to include Google in the source of our application, we'll need to tell our page to download the google JavaScript library and bootstrap it. We can simply include the `<script>` tag from Google itself, but then we won't be able to take advantage of the fantastic dependency injection from Angular. Additionally, we'll need to *wait* to bootstrap our angular app until the external Google script has been loaded on the page. This can definitely be a jarring experience for our users.

In order to take advantage of the dependency injection as well as bootstrap our app without waiting for external apps, we can wrap the loading of the Google API in a service that is specifically intended to *only* load the external service script. Luckily, we can do this easy using standard Angular mechanisms.

First, let's create an independent module we'll call `googleServices` where we'll write all of our google integration scripts. We'll also write the first api, which will wrap the `googleApi` script in a load tag.

In our `app/js/services.js` file, add the following:

```
angular.module('googleServices', [])
  .service('googleApi',
  function($window, $document, $q, $rootScope) {
    // Service definition function
});
```

Perfect. Now we can add the loading of the google script inside the service load. Since services themselves are singleton objects, we can guarantee that this function will run once and only once. Here we can load the script tag on the body of our page.

We'll use the promise api to load the script and resolve it after the script has loaded. Since Google gives us a method to call after the script has loaded, we'll need to call this function before we resolve the promise.

The full service looks like:

```
angular.module('googleServices', [])
  .service('googleApi',
    function($window, $document, $q, $rootScope) {
      // Create a defer to encapsulate the
      // loading of our Google API service.
      var d = $q.defer();

      // After the script loads in the browser,
      // we're going
      // to call this function, which in turn
      // will resolve
      // our global defer which enables the
      $window.bootGoogleApi = function(keys) {
        // We need to set our API key
        window.gapi.client.setApiKey(keys.apiKey);
        $rootScope.$apply(function() {
          d.resolve(keys);
        });
      };

    // Load client in the browser
    var scriptTag = $document[0].createElement('script');
    scriptTag.type = 'text/javascript';
    scriptTag.async = true;
    scriptTag.src='//apis.google.com/js/client:plusone.js?onload=onLoadCallback';
    var s = $document[0].getElementsByTagName('body')[0];
    s.appendChild(scriptTag);
```

```
    // Return a singleton object that
    // returns the promise
    return d.promise;
  });
```

Anywhere that we need to use the `googleApi` we can simply call it on the promise, rather than to the api itself. For instance:

```
googleApi.then(function(keys) {
  // The googleAPI has resolved and
  // we're ready to go.
});
```

Since we're going to need to build authorization in through the Google, we'll need to wrap in the authorization API. If we wrap this in a service, then we can simply call the authorization in a single factory.

Let's create a `googleAuthApi` factory that houses the authorization API. Since this service is fairly standard, we won't run through it line-by-line.

In our `api/js/service.js` file, add the following service:

```
.factory('googleAuthApi',
  function($window, $timeout, $rootScope, $q, googleApi) {
    var auth;

    // The authorize function will call
    // google auth.authorize which will
    // authorize this browser client to
    // the user's account along with the appropriate
    // scopes. It will then call
    // `handleAuthResult` with the promise
    // to resolve
    var authorize = function(keys, firstCheck) {
      var d = $q.defer();
      if (typeof(firstCheck) === "undefined")
        firstCheck = false;

      if (auth) {
        d.resolve(auth);
      } else {
        googleApi.then(function(keys) {
          gapi.auth.authorize({
            client_id: keys.clientId,
            scope: keys.scopes.join(" "),
```

```
            immediate: firstCheck
          }, handleAuthResult(d));
        });
      }
      return d.promise;
    };

    // handleAuthResult simple resolves the
    // deferred object if there are no errors.
    // If there are errors
    // then simply reject the promise
    var handleAuthResult = function(defer) {
      return function(authResult) {
        if (authResult && !authResult.error) {
          auth = authResult;
          $rootScope
            .$broadcast('user:authorized',authResult);
          defer.resolve(authResult);
        } else {
          defer.reject();
        }
      }
    };

    googleApi.then(function(keys) {
      authorize(keys, true);
    });

  // return a singleton object with the
  // single authorize api
  return {
    authorize: authorize
  }
})
```

A few notes about this service before we move on. We're using the `googleApi` service to resolve the loading of the google gapi client. Again, this allows us to load the google api without needing to wait, but safely allows us to describe the interactions that we'll have with the google API.

Additionally, we have the authorization check happening immediately on the load of the service (after the gapi client is made available). The *first* time we call authorize, we have to tell Google that we're calling the method the first time. This where the *firstCheck* variable comes into play.

Lastly, when Google responds we're sending an event to the rest of the app so that anywhere in the app. We'll use this later when we want to redirect the

user on a successful authorization when the page loads.

Building google Login

Now that we have our authorization api setup, we'll need to create the service that will actually make the login request.

Inside Google login, we'll need to wrap in the fetching of user info as well. This `googleLogin` service will need to include both handling login as well as handling fetching the user info. We'll also create an API method to check if our users are logged in.

Let's start building our `googleLoginApi` service.

```
.factory('googleLoginApi',
  function($q, $rootScope, googleAuthApi, googleApi) {
  // Create a load deferred object
  var loadedDefer = $q.defer(),
      loadedPromise = loadedDefer.promise,
      _loggedInUser = null,
      keys = null;
  // Our google Login API function definition
  // TODO: Define code here
  //
  // return our factory object
  return {
    login: login,
    getLoggedInStatus: function() {
      return loadedPromise;
    }
  }
})
```

Although we haven't written the functionality yet, the API is clear, we're providing a `login()` function and a `getLoggedInStatus()` function.

First off, we're going to need to tell the `gapi` to load the `oauth2` api. We can do this in side the factory function definition:

```
// call to load the oauth2 module on the
// gapi client immediately.
// When it's loaded, resolve the loadedDefer object
googleApi.then(function(_keys) {
  keys = _keys;
  gapi.client.load('oauth2', 'v2', function () {
    loadedDefer.resolve(keys);
```

```
  });
});
```

Once that's set, our `loadedDefer` will resolve which is simply a variable that we're holding onto inside the service to define the functionality on the service. This functionality is similar to defining the `googleApi` service in that we're simply defining a promise that resolves when our service is ready.

Once we set the `loadedDefer` to resolve when the library is ready, we can start defining the functionality of our service.

The `login()` function will be defined as a series of steps to attempt to resolve the currently logged in user. That is we will call to get the `userinfo` from the `gapi` client. If we succeed, then we'll simply resolve the user info and return the `loggedIn` user's info.

If the call to `userinfo` fails, then we'll use the `googleAuthApi()` that we just created to `authorize()` the Google service for the user. If this fails, then we'll simply need to reject the promise (which we can handle later in our app). If that succeeds, then we can re-request the `userinfo` from the `googleAuthApi` and proceed as normal.

The `login()` function in its entirety is as follows:

```
// Create a login function
// Inside this login function, we'll return a
// deferred object and then attempt to authorize
// and find user data
var login = function() {
  var d = $q.defer();

  // getUserInfo waits until the gapi login
  // service has loaded (using the loadedPromise)
  // and then immediately calls
  // gapi.client.oauth2.userinfo.get() to fetch
  // google data. It calls the `success` callback
  // if it's successful and the `fail` callback
  // if unsuccessful
  var getUserInfo = function(success, fail) {
    loadedPromise.then(function() {
      gapi.client.oauth2.userinfo.get()
      .execute(function(resp) {
        if (resp.email) success(resp);
        else fail(resp);
      });
    });
  };
```

```
  // resolveUserInfo resolves user data
  // from google and takes care of calling the
  // getUserInfo for us. It will also save and
  // cache the resolved user so we never call the
  // gapi session during the same browser load
  var resolveUserInfo = function(d) {
    getUserInfo(function success(resp) {
      // Resolve the response
      $rootScope.$apply(function() {
        d.resolve(resp);
      });
    },
    // Our failure function
    function fail(resp) {
      // If the response code is 401 (unauthorized)
      // then call authorize on the `googleAuthApi`
      // without being immediate (false)
      // and call resolveUserInfo to get the user's
      // info on load
      if (resp.code === 401) {
        googleAuthApi.authorize(keys, false)
          .then(function() {
            resolveUserInfo(d);
          });
      } else {
        d.reject(resp);
      }
    });
  };

  // Call resolve immediately
  resolveUserInfo(d);

  return d.promise;
}
```

Building a calendar API

Now that the basis for our google functionality is set, we can start to integrate other google services inside of our google api service. For us, we'll simply need to create an integration with the calendar API.

Considering that we've already created a pattern for how to integrate with Google, this process is quite easy. To recap, the pattern is:

1. Create a promise that gets resolved after the `googleApi` is resolved
2. Define our functionality atop the promise

First, let's create the `googleCalendarApi` serivce:

```
.factory('googleCalendarApi',
  function($timeout, $q, $rootScope,
  googleApi, googleAuthApi) {
    var calendarCache = [],
        loadedCalendar = $q.defer(),
        loadedPromise = loadedCalendar.promise;

    // Define our calendar functionality

    // First, call to load the calendar api
    // after the gapi client is loaded
    // and then resolve the loadedCalendar deferred object
    googleApi.then(function(keys) {
      gapi.client.load('calendar', 'v3', function() {
        // Loaded calendar after here
        loadedCalendar.resolve();
      });
    });

    return {
      // API
    }
})
```

Now all we need to do is interact with the google calendar API as expected on top of the `loadedPromise`.

In our service, we'll write a core function that will grab the entire list of calendars. In this function, we want to check to see if the user is actually logged in via Google. We obviously do not want an unregistered user to be able to use our Calendar API (although, we can handle this using our `googleLoginService`).

Other than that, we'll hold onto the calendar cache as a cache of the user's calendars and resolve the request:

```
var getCalendars = function(cb) {
  var d = $q.defer()
  googleAuthApi.authorize().then(function(auth) {
    if (calendarCache.length > 0) {
      d.resolve(calendarCache);
    } else {
```

```
      loadedPromise.then(function(keys) {
        gapi.client.calendar.calendarList
        .list({}).execute(function(resp) {
          // If the response is 401 (unauthorized)
          // then we know the user needs to login
          // so fire off the `user:login_required` event
          if (resp.code == 401) {
            console.log("resp", resp);
            $rootScope
              .$broadcast('user:login_required');
          } else {
            // otherwise, if we have calendar items in
            // the response, then we know the API call
            // was successful so we'll resolve the
            // calendar request
            if (resp && resp.items) {
              calendarCache = resp.items;
              $rootScope.$apply(function() {
                d.resolve(calendarCache);
              });
            } else {
              d.reject(resp);
            }
          }
        });
      });
    }
  });
  return d.promise;
};
```

In our service, it's likely more useful to to get a single calendar by it's `id`. We can use this calendar cache to sift through the list of calendars and pull out the calendar of the id we're interested in:

```
// Get a specific calendar by `id`.
// Since we are using the `getCalendars()` function
// from above, we know that we can depend on the
// gapi client and calendar service to be loaded.
// We'll simply walk through the list of calendars
// in our cache and find the one with the right "summary"
var getCalendar = function(id) {
  var d = $q.defer();

  var calendars = getCalendars()
```

```
  calendars.then(function(calendars) {
    var i = 0,
        c = null;
    for (i = 0; i < calendars.length; i++) {
      c = calendars[i];
      if (c.summary === id) {
        d.resolve(c);
      }
    }
    d.reject("Not found");
  });

  return d.promise;
}
```

Finally, we need to add two functions, one to get the list of events on the calendar and another to save an event to the calendar.

The functionality for `getEventsForCalendar()` is pretty much using the `gapi` api directly and it's functionality should be self explanatory:

```
var getEventsForCalendar = function(id, opts) {
  var d = $q.defer()

  var c = getCalendar(id);
  c.then(function(c) {
    gapi.client.calendar.events.list({
      'calendarId': c.id
    }).execute(function(resp) {
      $rootScope.$apply(function() {
        d.resolve(resp.items);
      });
    });
  });

  return d.promise;
}
```

Similarly, the `addEventToCalendar()` method is fairly self-explanatory as well. The only trick to the `addEventToCalendar()` method is that we'll need to define an event object that makes sense for our application.

```
var addEventToCalendar = function(evt, id) {
  var d = $q.defer();
```

```
  getCalendar(id).then(function(c) {
    var dateobj = {
      'calendarId': c.id,
      'resource': {
        'summary': evt.raw,
        'start': {
          'dateTime': evt.date
        },
        'end': {
          'dateTime': evt.date
        },
        'extendedProperties': {
          'shared': {
            'tags': evt.tags?evt.tags.join(',') : '',
            'urls': evt.urls?evt.urls.join(',') : '',
            'users': evt.users?evt.users.join(',') : ''
          }
        }
      }
    }
    gapi.client.calendar.events.insert(dateobj)
    .execute(function(resp) {
      $rootScope.$apply(function() {
        d.resolve(resp);
      });
    });
  });

  return d.promise;
}
```

Lastly, to remove an event from the Google calendar:

```
var deleteEventFromCalendar = function(evtId, id) {
  var d = $q.defer();
  getCalendar(id)
  .then(function(c) {
    gapi.client.calendar.events.delete({
      'calendarId': c.id,
      'eventId': evtId
    }).execute(function(r) {
      $rootScope.$apply(function(resp) {
        d.resolve(r);
      })
    });
```

```
  });

  return d.promise;
}
```

Providing login functionality in the app

Now that we have our services done, we'll need to integrate the `googleApi` into our app. There are two places that we'll handle authentication:

- The routes – we want only authorized users to be able to see pages that require authorization
- The events – we want to redirect on events that are thrown that say the user needs to be logged in

Setting up routes to require logged in users is easy. We'll use the `resolve key to resolve the user that's logged in. We can set this specifically on the`/dashboard‘ route:

```
// ...
.when('/dashboard', {
  templateUrl: 'views/dashboard.html',
  controller: 'DashboardController',
  resolve: {
    user: function(googleApi, googleLoginApi) {
      googleApi.then(function(keys) {
        return googleLoginApi.getLoggedInStatus();
      });
    }
  }
})
// ...
```

Inside of our `DashboardController`, we can inject the `user` to get a hold of the user, if we need it.

Secondly, we'll need to protect our app from unlogged in users. We'll simply use straight Angular idioms when looking for an event that gets fired on the `$rootScope`:

```
.run(['$rootScope', '$location', 'googleLoginApi',
  function($rootScope, $location, googleLoginApi) {
    $rootScope.$on('user:authorized', function(evt) {
      $location.path('/dashboard');
```

```
    });
    $rootScope.$on('user:login_required', function(evt) {
      $location.path('/');
    });
  }]);
```

With that, we now are completely integrated with the Google Calendar API.

The `app/js/app.js` in it's entirety is pasted below:

```
'use strict';

// App
function onLoadCallback() {
  var googleKeys = {
    clientId: 'YOUR_CLIENT_ID',
    apiKey: 'YOUR_API_KEY',
    scopes: [
      'https://www.googleapis.com/auth/calendar',
      'https://www.googleapis.com/auth/userinfo.profile',
      'https://www.googleapis.com/auth/userinfo.email'
    ]
  };
  window.bootGoogleApi(googleKeys);
}

angular.module('myApp', [
  'myApp.controllers',
  'myApp.filters',
  'myApp.services',
  'myApp.directives',
  'ngRoute',
  'googleServices'
])
.config(['$routeProvider', '$locationProvider',
  function($routeProvider, $locationProvider) {

    $routeProvider
      .when('/',
        {
          templateUrl: 'views/home.html',
          controller: 'HomeController'
        })
      .when('/login', {
        templateUrl: 'views/login.html',
        controller: 'LoginController'
```

```
      })
      .when('/dashboard', {
        templateUrl: 'views/dashboard.html',
        controller: 'DashboardController',
        resolve: {
          user: function(googleApi, googleLoginApi) {
            googleApi.then(function(keys) {
              return googleLoginApi.getLoggedInStatus();
            });
          }
        }
      })
      .otherwise({redirectTo: '/'});

}])
.run(['$rootScope', '$location', 'googleLoginApi',
  function($rootScope, $location, googleLoginApi) {
    $rootScope.$on('user:authorized', function(evt) {
      $location.path('/dashboard');
    });
    $rootScope.$on('user:login_required', function(evt) {
      $location.path('/');
    });
  }]);
```

There we go! The calendar is ready to go. Congrats on finishing the application. You've finished building the calendar app integrated with Google in natural language. For more information about Angular, head over to ng-newsletter.com!