Contents

[Due June 18, 11:59PM 2](#_Toc484685490)

[Validation 2](#_Toc484685491)

[Validating Forms 2](#_Toc484685492)

[Validating Inputs 2](#_Toc484685493)

[Table 1: Input Directives 2](#_Toc484685494)

[Applying Validation Styles 5](#_Toc484685495)

[The Single Page Application (SPA) 6](#_Toc484685496)

[Child Views 6](#_Toc484685497)

[Controllers 7](#_Toc484685498)

[Routing 7](#_Toc484685499)

[Parent View 8](#_Toc484685500)

[Services 11](#_Toc484685501)

[The $q Service 14](#_Toc484685502)

# Due June 18, 11:59PM

Worth 8% of your mark.

20% will be deducted each day this is late.

# Validation

Like any good data driven framework, AngularJS helps to automate input validation on the client. When user inputs are not in the correct format, AngularJS makes it easy to notify users immediately so they can quickly get the information they need to continue. Validation not only ensures accuracy of data inputs, it also helps to keep your site secure by narrowing the range of attacks that can be leveraged against an application. Your application users will also appreciate any effort to implement client side validation to guide them while also keeping their data safe.

### Validating Forms

It is possible to determine quickly if a form is valid with the expression that uses the Boolean $valid property of the form; **formName.$valid**. It is also possible to determine quickly if the control of a form is valid with the expression **formName.inputName.$valid**.

### Validating Inputs

Most of the input controls implement a **required directive**. lists several common **validation directives** that we can apply to a text input along with expressions that report their validity status.

### Table : Input Directives

|  |  |
| --- | --- |
| **Directive** | **Boolean Error Expression (True when invalid)** |
| required | formName.inputName.$error.**required** |
| ng-minlength="3" | formName.inputName.$error.**minlength** |
| ng-maxlength="10" | formName.inputName.$error.**maxlength** |
| ng-pattern="/^[A-z]+$/" | formName.inputName.$error.**pattern** |

Example : Validation Directives and Expressions

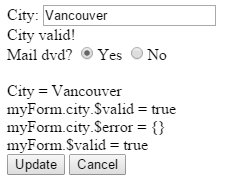
🞑 The following example shows how we can validate a text box field using directives for constraints such as minimum length, maximum length, regular expression and required. Each type of directive displays an error message when the invalid state is true. As well, the example displays the valid state for the control and the form.

The controller sets the default *userAddress* model to Vancouver. When the user clicks the update button, the *update()* function within the controller for the form launches an alert box with the update information. When the user clicks the cancel button the *cancel()* function within the controller displays an alert box with the cancelled information.

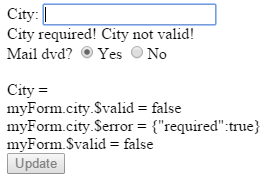
**index.html**

|  |
| --- |
| <!doctype html>  <html>  <head>  <title></title>  <script src="https://code.angularjs.org/1.5.8/angular.js"></script>  <script>  var myApp = angular.module('myApp', []);  myApp.controller('ExampleController', ['$scope', function ($scope) {  $scope.userAddress = 'Vancouver';  $scope.update = function (input) {  alert("Updated address to: " + input);  };  $scope.cancel = function (input) {  alert("Cancelled changes to: " + input);  };  }]);  </script>  </head>  <body ng-app="myApp">  <form name="myForm" ng-controller="ExampleController">  <!-- Input with validation -->  City: <input name="city" ng-model="userAddress"  ng-minlength="3"  ng-maxlength="10"  ng-pattern="/^[A-z]+$/"  required><br />  <!-- Show validation error messages -->  <span ng-show="myForm.city.$error.minlength">Minimum 3 characters required for city!</span>  <span ng-show="myForm.city.$error.maxlength">More than 10 characters not permitted for city!</span>  <span ng-show="myForm.city.$error.pattern">Only alphabetical characters allowed for city!</span>  <span ng-show="myForm.city.$error.required">City required!</span>  <span ng-show="myForm.city.$valid">City valid!</span>  <span ng-show="!myForm.city.$valid">City not valid!</span><br>  Mail dvd?  <input type="radio" name="mail" ng-model="dvd" value="yes" ng-required="!dvd">Yes  <input type="radio" name="mail" ng-model="dvd" value="no" ng-required="!dvd">No<br />  <!-- Validation message for mail -->  <span ng-show="!dvd">Mail DVD not selected<br></span>  <!-- Show input and form validation status -->  City = **{{**userAddress**}}**<br>  myForm.city.$valid = **{{**myForm.city.$valid**}}**<br>  myForm.city.$error = **{{**myForm.city.$error**}}**<br>  myForm.$valid = **{{**myForm.$valid**}}**<br>  <button ng-click="update(userAddress)" ng-disabled="myForm.$invalid">Update</button>  <button ng-click="cancel('Cancelled')" ng-hide="myForm.$invalid">Cancel</button>  </form>  </body>  </html> |

\When the form is valid, all invalidation expressions evaluate to false. The update button is enabled since the form must be valid. The cancel button is displayed since the form is valid.



When the form is invalid, the invalid expression evaluates to true. The update button is disabled in this case and the cancel button is hidden.



Exercise

🖍 Which directive and expression in enables the update button when the form is valid? (1 mark)

|  |
| --- |
|  |

Exercise

🖍 Starting with change the *required* attribute from the input text box to *ng-required*. Examine the use of *ng-required* with the radio options as an example of how to use the *ng-required* directive. (1 mark)

|  |
| --- |
|  |

Exercise

🖍 Adjust the code in Example 12 from day 1 to show the validation status for the mail option. If “Mail hard copy” is checked the form should only be valid if either UPS or Fedex is selected. None are selected by default. This validation message should declare whether the mail control is valid or invalid. Use the form’s $valid property value to show this message. Show the revised example with your adjustment. (2 marks)

|  |
| --- |
|  |

### Applying Validation Styles

We can apply validation styles by several means. To help with this effort, AngularJS actually automatically generates several CSS classes for elements that are using AngularJS validation directives.

* @property {boolean} $**untouched** True if control has not lost focus yet.
* @property {boolean} $**touched** True if control has lost focus.
* @property {boolean} $**pristine** True if user has not interacted with the control yet.
* @property {boolean} $**dirty** True if user has already interacted with the control.

displays several class names that Angular dynamically generates in the browser at run time:

Figure



Example : Validation Styling

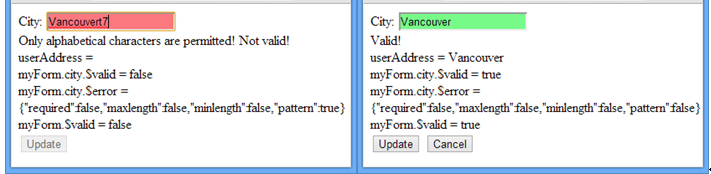
🞑 This example applies auto-generated css styles to the input box in . First, starting with , add the following code inside the head tag:

|  |
| --- |
| <style type="text/css">  .css-form input.ng-invalid.ng-dirty { background-color: #FA787E; }  .css-form input.ng-valid.ng-dirty { background-color: #78FA89; }  </style> |

Then, replace the form directive with this one to set the form’s class for styling:

|  |
| --- |
| <form name="myForm" ng-controller="ExampleController" class="css-form"> |

When running the application now, the input box background changes based on validation status:



Exercise

🖍 Modify to so the city shows a gray or slightly off white background to the city input if no modifications have been made. Show your revised HTML here.

|  |
| --- |
|  |

# The Single Page Application (SPA)

Now that we have examined some examples containing controllers, directives, and filters let’s see how to further separate our content and logic while improving the application scalability. In an AngularJS application it is common to separate the content and logic into the following structures:

* A **parent view** which manages the general presentation for the site.
* **Partial views**, or **child views**, where blocks of HTML are in separate files for specific presentations.
* An application **module** which contains a reference to the application and **routing** services.
* A series of **controllers** to populate the **models** with data.
* A **router** which receives page requests and selects the appropriate **template** and **controller**.

Example : Modules, Routing, and Templates

🞑 This example demonstrates how further to separate the logic and presentation using routing and partial views.

## Child Views

Child views can be swapped so only part of the page is refreshed within a one page application. For our example, here is the first child view which displays a listing of playing cards.

**views/list.html**

|  |
| --- |
| <table>  <tr ng-repeat="card **in** cards">  <td>  <a href="#/home/number/**{{**card.number**}}**/suit/**{{**card.suit**}}**">Detail</a>  </td>  <td>**{{**card.number**}}**</td>  <td>**{{**card.suit**}}**</td>  </tr>  </table> |

Here is our second child view which shows details about the playing card.

**views/detail.html**

|  |
| --- |
| Number: **{{**card.number**}}**<br />  Suit: **{{**card.suit**}}** <br />  Rank: **{{**card.numOrd**}}** <br /> |

## Controllers

Here is our controller file. It contains one parent controller and two child controllers. One child has parameters and the other does not.

**js/controller.js**

|  |
| --- |
| var myControllers = (function () {  'use strict';  var ctrls = angular.module('myControllers', []);  // Define this so it can be accessed globally for simplicity.  var deck = [{ "number": "Q", "suit": "Hearts", "numOrd": 12 },  { "number": "A", "suit": "Spades", "numOrd": 1 },  { "number": "J", "suit": "Spades", "numOrd": 11 }];  // Parent controller  ctrls.controller('AppCtrl', ['$scope', function ($scope) {  $scope.title = "AngularJS Tutorial";  }]);  // Child controller (no parameters) - Inject scope.  ctrls.controller('ListCtrl', ['$scope', function ($scope) {  $scope.cards = deck;  }]);  // Another child controller - Inject scope and $routeParams service.  ctrls.controller('DetailCtrl',  ['$scope', '$routeParams', function ($scope, $routeParams) {  // Define empty object.  $scope.card = {};  // Find card that matches the parameters and assign it to card.  for (var i = 0; i < deck.length; i++) {  if (deck[i].number == $routeParams.numberID  && deck[i].suit == $routeParams.suitID) {  $scope.card = deck[i];  break;  }  }  }]);  return ctrls;  }()); |

## Routing

Next, we will declare a module which references the services and routing structures in our application. To provide our application references to routing and controller services we will inject the AngularJS ‘ngRoute’ service and our own custom ‘myControllers’ service into the module declaration. The router selects a template and controller based on the page request with a series of **when** and **otherwise** statements.

**js/app.js**

|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* Declare application reference.  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  // Inject routing service and controller reference.  var cardApp = angular.module('cardApp', ['ngRoute', 'myControllers'])  .config(function ($routeProvider) {  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* Define routes.  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  // For the 'home' page.  $routeProvider.when("/home", {  templateUrl: 'views/list.html',  controller: 'ListCtrl'  })  // For the 'detail' page.  .when('/home/number/:numberID/suit/:suitID', {  templateUrl: 'views/detail.html',  controller: 'DetailCtrl'  })  // For the 'default' page.  .otherwise({ redirectTo: '/home' });  }); |

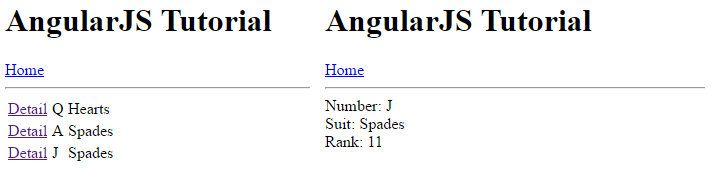
## Parent View

Now at the route of application is our parent view. The **ng-view** directive serves as a placeholder for the view that the controller selects.

**index.html**

|  |
| --- |
| <!doctype html>  <html>  <head>  <title>Starting Angular</title>  <script src="https://code.angularjs.org/1.5.8/angular.js"></script>  <script src="https://code.angularjs.org/1.5.8/angular-route.js"></script>  <script src="js/controller.js"></script>  <script src="js/app.js"></script>  </head>  <!-- Notice here that ng-app references our 'cardApp' module -->  <body ng-app='cardApp'>  <!-- Reference the application level controller for the title model. -->  <div ng-controller="AppCtrl" ng-model="title">  <h1>**{{**title**}}**</h1>  <!-- The #home link selects routing for '/home'. -->  <a href="#home">Home</a>  <hr>  <!-- Our one page app works by switching views here. -->  <div ng-view></div><br />  <hr>  </div>  </body>  </html> |

When we run our project, we can switch between parent and child views in our one page application thanks to routing.



Exercise (Can have more than one answer)

Starting with Example 3, what service was injected into the DetailCtrl controller to handle the card number and suit parameters?

1. $routeHelper
2. $routeParams
3. $scope
4. $router

Exercise

Select the best answer for the following statement.

The router helps us select:

1. The service.
2. The module and partial view.
3. The parent view and child controller.
4. The parent view and parent controller.
5. The partial view and child controller.

Exercise

🖍 The *AppCtrl* controller is referenced in the parent div tag within the index.html page. However, a reference to the *ListCtrl* controller cannot be found in index.html. How does the list.html template reference the controller than contains the array of cards? (1 mark)

|  |
| --- |
|  |

Exercise

This is the router for the application that runs at yourdomain.com:

|  |
| --- |
| var myApp = angular.module('myApp', ['ngRoute', 'myControllers'])  .config(function ($routeProvider) {  $routeProvider.when("/c", {  templateUrl: 'views/d.html',  controller: 'mCtrl'  })  .when('/a', {  templateUrl: 'views/o.html',  controller: 'rCtrl'  })  .when('/f', {  templateUrl: 'views/i.html',  controller: 'zCtrl'  })  .otherwise({ redirectTo: '/c' });  }); |

1. If somebody enters yourdomain.com/o in the browser, which html template is selected:
2. i.html
3. o.html
4. d.html
5. c.html
6. Which statement is true?
7. i.html is a parent controller and zCtrl is a parent view.
8. i.html is a child controller and zCtrl is a parent controller.
9. i.html is a parent view and zCtrl is a child controller.
10. i.html is a child view and zCtrl is a child controller.
11. List the name of the service that is injected to enable routing \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.
12. If somebody enters yourdomain.com/c in the browser, list the html template that is selected \_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Exercise

Starting with Example 3, add a third child view and store the tags in views/about.html. Display your name in this page. Provide an ‘About’ link in the parent view.

Hint: The link will take on the value: **#home/about** and the route will be **/home/about**

Show your partial view here: (1 mark)

|  |
| --- |
|  |

Show your updated controller here: (1 mark)

|  |
| --- |
|  |

Show your js/app.js file here with your updated router: (1 mark)

|  |
| --- |
|  |

# Services

Services allow us to divide the logic even further. When we package our logic as a service, we can use the AngularJS dependency injection pattern to pass a reference to it into our controllers.

Example : Declaring a Custom Service

🞑 This example shows how to declare a custom service for retrieving data for the card application. Start with the solution for . Next, define a service that will retrieve the card data. In the *service.js* file below, we have hard-coded the JSON data, but it is possible instead to use the commented-out commands to communicate with a live web service.

**js/service.js**

|  |
| --- |
| cardApp.factory("cardService", function () {  // Define this so it can be accessed globally for simplicity.  var deck = [{ "number": "Q", "suit": "Hearts", "numOrd": 12 },  { "number": "A", "suit": "Spades", "numOrd": 1 },  { "number": "J", "suit": "Spades", "numOrd": 11 }];  return {  getCards: function () {  return deck;  },  getCard: function (number, suit) {  for (var i = 0; i < deck.length; i++) {  if (deck[i].number == number && deck[i].suit == suit) {  return deck[i];  }  }  }  }  }); |

To use the new service from our list controller, we use dependency injection to provide a reference to the card service. Here is the list controller replacement code with the injected card service:

**js/controller.js**

|  |
| --- |
| var myControllers = (function () {  'use strict';  var ctrls = angular.module('myControllers', []);  // Parent controller  ctrls.controller('AppCtrl', ['$scope', 'cardService',  function ($scope, cardService) {  $scope.title = "AngularJS Tutorial";  $scope.cards = cardService.getCards();  $scope.getPlayingCard = function (cardNum, suit) {  $scope.card = cardService.getCard(cardNum, suit);  }  }]);  return ctrls;  }()); |

**js/app.js**

|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* Declare application reference.  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  // Inject routing service and controller reference.  var cardApp = angular.module('cardApp', ['myControllers']); |

To reference our new services file, add the following script reference to the index.html. Make sure that you add it after the reference to js/app.js.

**index.html**

|  |
| --- |
| <!doctype html>  <html>  <head>  <title>Starting Angular</title>  <script src="https://code.angularjs.org/1.5.8/angular.js"></script>  <!-- The loading order matters here -->  <script src="js/controller.js"></script>  <script src="js/app.js"></script>  <script src="js/service.js"></script>  </head>  <!-- Notice here that ng-app references our 'cardApp' module -->  <body ng-app='cardApp'>  <!-- Reference the application level controller for the title model. -->  <div ng-controller="AppCtrl">  <table>  <tr ng-repeat="card **in** cards">  <td>**{{**card.number**}}**</td>  <td>**{{**card.suit**}}**</td>  </tr>  </table>  Card Number: <input type="text" ng-model="cardNumber"><br />  Card Suit: <input type="text" ng-model="cardSuit"><br />  <input type="button" ng-click="getPlayingCard(cardNumber,cardSuit)" value="Get Ranking">  <label ng-show="card.numOrd">Card Ranking: **{{**card.numOrd**}}**</label>  </div>  </body>  </html> |

The output after implementing these changes is identical to the output from . This time though, when the application runs, Angular uses the card service to retreive the JSON data.

Exercise

Starting with Example 4, create new file named converterService.js inside the js folder and place the following code in it.

**js/converterService.js**

|  |
| --- |
| cardApp.factory("temperatureService", function () {  return {  getCelsius: function (f) {  return (f - 32) \* 5 / 9;  },  }  }); |

Next, reference your file from your html file with the following highlighted adjustment:

**index.html**

|  |
| --- |
| <!-- The order really matters here -->  <script src="js/controller.js"></script>  <script src="js/app.js"></script>  <script src="js/service.js"></script>  <script src="js/converterService.js"></script> |

Adjust your controller to receive the service reference. Then, create a function in your controller to receive a fahrenheit parameter. Call your getCelsius service function from your controller and store the returned value on the scope. In your HTML, add an input box that allows a user to input the temperature in Fahrenheit. Add a button which sends the Fahrenheit value to your new function in the controller. Show your Celsius value in a label once it is retrieved from the service. Show your adjusted controller here:

|  |
| --- |
|  |

Show your adjusted HTML file here:

|  |
| --- |
|  |

Exercise

Show a screen shot of your debugger while halted at a break point inside your service:

|  |
| --- |
|  |

# The $q Service

Asynchronous operations are operations that are executed simultaneously with others. For example, when retrieving a remote resource the application must wait. If you call for the resource asynchronously the application will continue operations while waiting for the resource. Then when the resource arrives it is able to process this response.

In AngularJS 1, the $q service helps you run functions asynchronously. You can use it to return values (or exceptions) when processing is complete.

The $q service allows you to create a deferred object:

var deferred = $q.defer();

The deferred object allows you to expose an associated **Promise** instance as well as APIs that can signal the successful or unsuccessful completion of a task. A deferred promise essentially promises a resource once it is available. The promise delivery is announced with a call to the *resolve()* function of the deferred object. When an error or issue occurs, the promise can be cancelled with the deferred object’s reject() function.

var deferred = $q.defer();

$http.get("http://ssdprogram.ca/tetJson.php")

.success(function (data) {

deferred.resolve(data);

})

.error(function () {

console.log('bob' + $q.reject + 'bob');

deferred.reject("\*\*\* Rejected! \*\*\*\*");

});;

return deferred.promise;

Example : Asynchronous calls with $q

🞑 This example begins with Example 4. It demonstrates how to use the q service to wait until cards are retrieved from a REST service. Data is returned when it is available. Starting with Example 4, replace the getCards() function in the service file with the following:

**js/service.js**

|  |
| --- |
| cardApp.factory("cardService", ["$http", "$q", function ($http, $q) {  var deck = [{ "number": "Q", "suit": "Hearts", "numOrd": 12 },  { "number": "A", "suit": "Spades", "numOrd": 1 },  { "number": "J", "suit": "Spades", "numOrd": 11 }];  return {  getCards: function () {  var deferred = $q.defer(); // Set up deferral.  $http.get("http://ssdprogram.ca/cards.php")  .success(function (data) {  deferred.resolve(data); // Return resource.  })  .error(function () {  deferred.reject("\*\*\* Rejected! \*\*\*\*"); // Return rejection.  });;  return deferred.promise; // Promise to return something once available.  },  getCard: function (number, suit) {  for (var i = 0; i < deck.length; i++) {  if (deck[i].number == number && deck[i].suit == suit) {  return deck[i];  }  }  }  }  }]); |

Then, replace the call to getCards() in the controller with the following:

**js/controller.js**

|  |
| --- |
| var promise = cardService.getCards();  promise.then(  function (data) {  $scope.cards = data;  },  function (errorReason) {  $scope.error = true;  $scope.msg = "An error occurred while fetching data.";  console.log(errorReason);  }); |

The application should run as before but this time it will display data from a remote source.

Exercise

Show a screenshot of your debugger while halted at a break point inside the controller when the remote data is returned to it:

|  |
| --- |
|  |

Exercise

Modify the URL so it is no longer valid. Show the error message in your HTML when the data is not returned to the application. Show your modified HTML file here:

|  |
| --- |
|  |

Exercise

🖍 Try and ensure it receives data successfully. In the blank spaces provided beside the lines of code, enter a number to indicate the order which each instruction is called. 1 is the first instruction and 6 is the last instruction. Step through your code or use logging to verify the sequence: (3 marks)

\_\_\_\_\_\_\_ var promise = cardService.getCards();

\_\_\_\_\_\_\_ promise.then(function (data) {

\_\_\_\_\_\_\_ $scope.cards = data;

},

\_\_\_\_\_\_\_ var deferred = $q.defer(); // Set up deferral.

$http.get("http://ssdprogram.ca/cards.php")

.success(function (data) {

\_\_\_\_\_\_\_ deferred.resolve(data); // Return resource.

})

.error(function () {

deferred.reject("\*\*\* Rejected! \*\*\*\*"); // Return rejection.

});;

\_\_\_\_\_\_\_ return deferred.promise; // Promise to return something once available

Exercise

Add a new function to your services file which retrieves numbers from the url <http://ssdprogram.ca/testJson.php>. Call this function from your controller. Show your data in the html.

Show your revised services file here:

|  |
| --- |
|  |

Show your revised controller file here:

|  |
| --- |
|  |

Show your revised html file here:

|  |
| --- |
|  |