DevTeach Vancouver: AngularJS 101

Presented by Shaun Luttin and Pat McGee

[Introduction 2](#_Toc372989084)

[Downloading Code Samples 2](#_Toc372989085)

[Advantages of the AngularJS Framework 2](#_Toc372989086)

[Terminology Introduction 2](#_Toc372989087)

[Sorting Filters 6](#_Toc372989088)

[Search Filters 8](#_Toc372989089)

[Modules, Routing and Views 10](#_Toc372989090)

[Services 14](#_Toc372989091)

[Route Parameters 15](#_Toc372989092)

[Managing Form Inputs 19](#_Toc372989093)

[Handling Button Clicks 19](#_Toc372989094)

[Hiding, Showing, Disabling, and Enabling Inputs 21](#_Toc372989095)

[Validating Form Inputs 22](#_Toc372989096)

[Applying Validation Styles 25](#_Toc372989097)

[Conclusion 28](#_Toc372989098)

# Introduction

This document presents a step-by-step approach to help you quickly ramp up on essential structures of AngularJS for professional, data-driven development.

## Downloading Code Samples

For the latest version of this document and code sample downloads please see <http://handsoncoding.net>

## Advantages of the AngularJS Framework

* Excellent separation of presentation and logic for data driven applications.
* Excellent support for one page application development.
* Convenient two way data binding and validation.
* Developed and supported by Google.
* Excellent browser compatibility.

## Terminology Introduction

To help start this discussion, here are some general definitions:

|  |  |
| --- | --- |
| **Model:** | * Is an object that defines the data attributes. * Defines the type of object(s) returned to the page. |
| **View:** | * Is a partial HTML template. |
| **Router:** | * Receives page requests and selects the appropriate view and controller. |
| **Controller:** | * Drives the logic to get/set the data and to call services. |
| **Scope:** | * Is an AngularJS service that manages data models for views and controller. * The scope retrieves the model; the controller gets/sets the model's data. |
| **Expression:** | * Is a Boolean condition that a directive uses to trigger behavior. |
| **Directives:** | * Are HTML attributes that invoke custom behavior. AngularJS provides many directives but you can also make your own. |
| **Services:** | * Are singleton objects or functions that carry out specific tasks common to web apps. For example, AngularJS defines an HTTP service to manage get, post, put, and delete web service operations. |
| **Module:** | * Is a collection of services, directives, filters, and configuration information. An AngularJS application can reference only one module. |
| **Dependency Injection:** | * Is a pattern of development that AngularJS uses; it provides modules and controllers with access to other AngularJS services. |

AngularJS directives are HTML attributes that trigger specific behavior. Table 1 shows some basic directives for loading the application and model data. We will need these to start our discussion.

### Table 1: Introductory AngularJS Directives

|  |  |
| --- | --- |
| ng-app | Marks the AngularJS application route; may also indicate which module to load. |
| ng-model | Is the data model that the scope retrieves. |
| ng-controller | Indicates which controller gets/sets the data model. |
| ng-repeat | Enables iteration through an array of objects to populate an HTML template.  <div ng-repeat="item in items">  <p>**{{** item **}}**</p>  </div> |

### Example 1: AngularJS Hello World! (tested 23 Nov ~ Shaun)

🞑 This example introduces basic AngularJS scope, controllers, views, and directives. Inside *controllers.js*, a dependency-injection pattern provides the controller with access to the scope service. The controller function defines a model called *title* and assigns data to it.

**js/controllers.js**

|  |
| --- |
| function AppCtrl($scope) {  'use strict';  $scope.title = "AngularJS Hello World!";  }  AppCtrl.$inject = ['$scope']; |

**index.html**

Inside the view, the **ng-app** directive marks the application root. The **ng-controller** directive tells AngularJS that the div tag will be scoped by the *AppCtrl* controller. The **ng-model** directive indicates that the *title* model is to be used in this section.

|  |
| --- |
| <!doctype html>  <html>  <head>  <title>Starting Angular</title>  </head>  <!-- The ng-app directive triggers load and setup of AngularJS after the DOM is loaded.  It sets the tag as the root of the AngularJS app. -->  <body ng-app>  <!-- Identify the controller and model to be used. -->  <div ng-controller="AppCtrl" ng-model="title">  <!-- The expression, '**{{**title**}}**' gets data from the model. -->  <h1>**{{**title**}}**</h1>  </div>  <script src="http://code.angularjs.org/angular-1.2.0-8336b3a/angular.min.js"></script>  <script src="js/controllers.js"></script>  </body>  </html> |

When the program runs the expression, {{title}}, displays the model data through the scope.



### Example 2: Adding a Repeater (tested 23 Nov ~ Shaun)

🞑 This example demonstrates how to implement the **ng-repeater** directive to display attributes of a model that is defined with JSON.

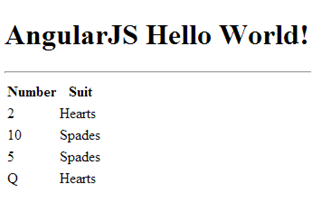
Starting with Example 1, replace the code inside *js/controller.js* with the following controller code. This code defines a new model called *cards* which stores a collection of JSON objects. Each object in the collection has a *number*, *suit*, and *numOrd* attribute.

|  |
| --- |
| function AppCtrl($scope) {  'use strict';  $scope.title = "AngularJS Hello World!";  // Define a new model called 'cards' with JSON.  $scope.cards = [  { "number": "2", "suit": "Hearts", "numOrd": 2 },  { "number": "10", "suit": "Spades", "numOrd": 10 },  { "number": "5", "suit": "Spades", "numOrd": 5 },  { "number": "Q", "suit": "Hearts", "numOrd": 12 }  ];  }  // Inject scope into the controller.  AppCtrl.$inject = ['$scope']; |

Next, to define a template that Angular will populate with each object in the *cards* collection, add the following tags and expressions after the <h1>{{title}}</h1> content in *index.html*. The *ng-repeat* directive allows us to define a *card* object for each item in the array. The card object is then used to populate *number* and *suit* attributes within an HTML template for a table row:

|  |
| --- |
| <table>  <tr><th>Number</th><th>Suit</th></tr>  <!-- Populate the template with each object in the cards model. -->  <tr ng-repeat="card in cards">  <td>**{{**card.number**}}**</td>  <td>**{{**card.suit**}}**</td>  </tr>  </table>  <br /> |

When running the application after these changes, the list of cards appears in the browser.



## Sorting Filters

AngularJS provides an **orderBy** filter that allows you to sort items that are displayed with a repeater. You first have to define a sorting model that includes the choices of sorting attributes:

<select ng-model="orderProp">

<option value="suit">Suit</option>

<option value="numOrd">Number</option>

</select>

Then, in the tag that contains your repeater, you set the orderBy filter with the designated sorting model:

<tr ng-repeat="card in cards | orderBy:orderProp ">

### Example 3: Ordering Collection Results (tested 23 Nov ~ Shaun)

🞑 This example shows how to order cards by either suit or number. To start, begin with the solution from Example 2. Then, add the following code just after the title header tags to input a user’s sort preference through a drop down selector.

|  |
| --- |
| Sort by:  <select ng-model="orderProp">  <option value="suit">Suit</option>  <option value="numOrd">Number</option>  </select> |

Notice that the *numOrd* property from the JSON model is used to sort the cards. The numOrd property is used for sorting since *orderBy* would not be able to determine the difference between the Queen, Q, its numeric equivalent of 12:

$scope.cards = [

{ "number": "2", "suit": "Hearts", "numOrd": 2 },

{ "number": "10", "suit": "Spades", "numOrd": 10 },

{ "number": "5", "suit": "Spades", "numOrd": 5 },

{ "number": "Q", "suit": "Hearts", "numOrd": 12 }

];

Next, inside index.html, replace the row tag and repeater directive with the following revision. This revision uses the *orderBy* filter to sort data by the selected property in the *orderProp* model:

|  |
| --- |
| <tr ng-repeat="card in cards | orderBy:orderProp "> |



## Search Filters

AngularJS provides a nice way to quickly filter selection results. The user must define a search term:

Search: <input type="text" ng-model="modelName">

Then, the model name is included in the repeater:

<ul ng-repeat="item in items | filter:modelName">

<li>**{{** item **}}**</li>

</ul>

### Example 4: Searching All Items in a Collection (tested 23 Nov ~ Shaun)

🞑 This example demonstrates an implementation of the AngularJS filter. Starting with Example 3, under the ordering options selector, add the following input for filtering terms:

|  |
| --- |
| Search: <input type="text" ng-model="searchTerm"> |

Then, replace the tag that has the repeater with the following version where a search filter is included:

|  |
| --- |
| <tr ng-model="card" ng-repeat="card in cards | orderBy:orderProp | filter:searchTerm "> |

With hardly any code, the filter allows a user to narrow collection results with search terms:



The effect is impressive. However, the filter in Example 4 filters on all attributes in the collection. Given the data defined with JSON:

$scope.cards = [ { "number": "2", "suit": "Hearts", "numOrd": 2 },

{ "number": "10", "suit": "Spades", "numOrd": 10 },

{ "number": "5", "suit": "Spades", "numOrd": 5 },

{ "number": "Q", "suit": "Hearts", "numOrd": 12 } ];

You will notice a search on ‘2’ would yield ‘Q Hearts’ and ‘2 Hearts’. ‘Q Hearts’ is selected because this item also contains a *numOrd* property of 12.



### Example 5: Refined Filtering (tested 23 Nov ~ Shaun)

🞑 Given the flawed filtering in Example 4, this example binds search filter to specific model attributes. To begin, replace the search input in index.html with the following inputs:

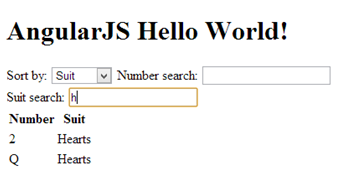
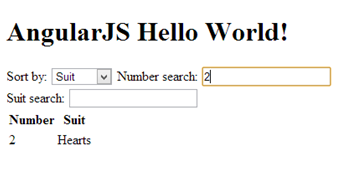
|  |
| --- |
| Number search: <input type="text" ng-model="searchTerm.card.number">  Suit search: <input type="text" ng-model="searchTerm.card.suit"><br/> |

These inputs allow you to specify the collection attributes that are used in a filter. For our example, the first input allows a user to define the search by card number. The second input allows a user to define the filter by suit.

To enable searching of the collection with both number and suit attributes, replace the existing row tag with the ng-repeat directive and ordering filter with a new one that also includes the improved search filter:

|  |
| --- |
| <tr ng-repeat="card in cards | orderBy:orderProp | filter:searchTerm.card "> |

Now the search results are reliable:



# Modules, Routing and Views

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 **layout template** which manages the general presentation for the site.
* **Partial templates**, or **views**, which are blocks of HTML 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 6: Modules, Routing, and Templates (tested 23 Nov ~ Shaun)

🞑 This example demonstrates how further to separate the logic and presentation using modules, routing and templates. This example is similar to the ones preceding it, but we will start from scratch to implement a more scalable development pattern for a one page application.

To create the first view, we will place the tags which filter and display the cards collection in a separate file called *list.html* in the **views** directory. To help later, some additional hyperlinks to detail and delete routes are included, but these new links are currently inactive.

**views/list.html**

|  |
| --- |
| Sort by:  <select ng-model="orderProp">  <option value="suit">Suit</option>  <option value="numOrd">Number</option>  </select>  Number search: <input type="text" ng-model="searchTerm.card.number">  Suit search: <input type="text" ng-model="searchTerm.card.suit"><br />  <table>  <tr><th><th><th>Number</th><th>Suit</th></tr>  <tr ng-repeat="card in cards | orderBy:orderProp | filter:searchTerm.card ">  <td><a href="#/home/number/{{card.number}}/suit/{{card.suit}}">View</a></td>  <td>  |  <a href="#/delete/number/{{card.number}}/suit/{{card.suit}}">Delete</a>  </td>  <td>**{{**card.number**}}**</td>  <td>**{{**card.suit**}}**</td>  </tr>  </table>  <br /> |

In this example, the controller assigns data to the model, the scope passes the populated model to the template, and the template renders the model data.

data 🡪 controller 🡪 model 🡪 scope 🡪 template 🡪 complete HTML

**js/controller.js**

|  |
| --- |
| /\*global angular \*/  var cardAppControllers = (function () {  'use strict';  var cardAppControllers = angular.module('cardAppControllers', []);  // Declare the application controller and inject the scope reference.  cardAppControllers.controller('AppCtrl', ['$scope', function ($scope) {  // Define the title model.  $scope.title = "AngularJS Tutorial";  }]);  // Declare the controller that populates 'list.html' with data and inject the scope.  cardAppControllers.controller('ListCtrl', ['$scope',  function ($scope) {  // Define the cards model.  $scope.cards = [  { "number": "2", "suit": "Hearts", "numOrd": 2 },  { "number": "10", "suit": "Spades", "numOrd": 10 },  { "number": "5", "suit": "Spades", "numOrd": 5 },  { "number": "Q", "suit": "Hearts", "numOrd": 12 }];  }]);  return cardAppControllers;  }()); |

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 ‘cardAppControllers’ 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**

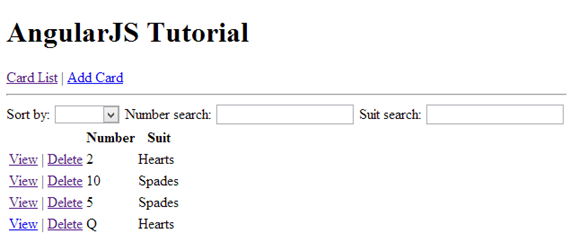
|  |
| --- |
| // Declare module that references our controllers.  var cardApp = angular.module('cardApp', ['ngRoute', 'cardAppControllers']).config(function ($routeProvider) {  /\*  Inject the AngularJS routing (ngRoute) service so we can  access the $routeProvider reference in our routing function.  Also inject the 'cardAppControllers' service which we will  define in 'controllers.js'.  \*/  'use strict';  $routeProvider.when("/home", {  /\* When 'home' route is selected  use the 'list.html' template and the 'ListCtrl' controller. \*/  templateUrl: 'views/list.html',  controller: 'ListCtrl'  }).  // If no route is selected then use the 'home' route.  otherwise({ redirectTo: '/home' });  }); |

Now we can consolidate the main layout template of our one page application. 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>  </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'. Routing for  #add does not exist yet so the default for '/home' will be chosen.-->  <a href="#home">Card List</a> | <a href="#add">Add Card</a>  <hr>  <!-- Our one page app works by switching views here. -->  <div ng-view></div><br />  </hr>  </div>  <!--angular-->  <script src="http://code.angularjs.org/angular-1.2.0-8336b3a/angular.min.js"></script>  <script src="http://code.angularjs.org/angular-1.2.0-8336b3a/angular-route.min.js"></script>  <!--site-->  <script src="js/controllers.js"></script>  <script src="js/app.js"></script>  </body>  </html> |

When we run our project the output similar to the output in Example 5. However, this current example is more scalable because we have separated the content and into layout, view, module, routing, and controller components.



# Services

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

### Example 7: Declaring a Custom Service (tested 23 Nov ~ Shaun)

🞑 This example shows how to declare a custom service for retrieving data for the card application. Start with the solution for Example 6. Next, define a service that will retrieve the card data. Note that we are referencing the AngularJS $http with inline injection. This service provides access to the browser’s *XMLHttpRequest* object for making requests to the server. In the *services.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/services.js**

|  |
| --- |
| cardApp.factory("cardService", function ($http) {  'use strict';  return {  getCards: function ($scope) {  **/\***  **// LIVE WEB SERVICE**  **$http.get('http://localhost:1353/api/Cards').success(function(data) {**  **$scope.cards = data;**  **});**  **\*/**  $scope.cards = [  { "number": "2", "suit": "Hearts", "numOrd": 2 },  { "number": "10", "suit": "Spades", "numOrd": 10 },  { "number": "5", "suit": "Spades", "numOrd": 5 },  { "number": "Q", "suit": "Hearts", "numOrd": 12 }];  },  }  }); |

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**

|  |
| --- |
| <script src="js/services.js"></script> |

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/controllers.js**

|  |
| --- |
| **// Inject the scope and new cardService reference into the controller.**  cardAppControllers.controller('ListCtrl', ['$scope', 'cardService',  function ($scope, cardService) {  **// Define the cards model.**  cardService.getCards($scope);  }]); |

The output after implementing these changes is identical to the output from Example 6. This time though, when the application runs, Angular uses the card service to retreive the JSON data.

# Route Parameters

When providing routing for detailed views, it is often necessary to capture parameter values from the URL. Routers use the following syntax to identify parameter names and values:

when('/home/parameter0Name/:parameter0Value', {

templateUrl: 'views/partialTemplate.html', controller: 'CtrlName' }).

The URL takes on the following format:

<a href="#/home/parameter0Name/**{{**parameter0Value**}}**">View</a>

The controller can then reference the parameter through the **routeParams** service:

**// Inject scope and routeParams services.**

appControllers.controller('CtrlName', ['$scope', '$routeParams',

function ($scope, $routeParams) {

alert('Parameter received: ' + $routeParams.parameter0Name);

}]);

### Example 8: Adding a Detail View (tested 23 Nov ~ Shaun)

🞑 To demonstrate how to implement route parameters we will add a detail view to the card application starting with the solution for Example 7. First, add a new partial template called *detail.html* which contains presentation tags for displaying a card’s number and suit data.

**views/detail.html**

|  |
| --- |
| <h3>Card Detail</h3>  <table>  <th></th><th>Number</th><th>Suit</th>  <tr ng-model= "card">  <td><a href="#/delete/number/**{{**card.number**}}**/suit/**{{**card.suit**}}**">Delete</a></td>  <td>**{{**card.number**}}**</td>  <td>**{{**card.suit**}}**</td>  </tr>  </table> |

Then, add a ‘getCard’ function (which is a service) to retrieve details for a single card. This service receives a reference to the scope, cardNum, and cardSuit. It will populate the data of scopes card object so that the scope can pass that data to the card object expressions in the view. The workflow is something like this:

**data store** 🡨scope.model 🡪 **service** 🡨scope.model 🡪 **controller** 🡨scope.model 🡪 **view**

**js/services.js**

|  |
| --- |
| getCard: function ($scope, cardNum, cardSuit) {  **/\***  **// LIVE WEB SERVICE**  **var url = "http://localhost:1353/api/cards?number=" + cardNum**  **+ "&suit=" + cardSuit;**  **$http.get(url).success(function(data) {**  **$scope.card = data;**  **}); \*/**  $scope.card = { "number": cardNum, "suit": cardSuit };  }, |

Next, add a controller to call the new service. Notice how we need to inject the $scope, $routeParams, and cardService references. The $routeParams service provides us access to the parameters that are passed to the controller. The cardService is the service module that we defined.

**js/controller.js**

|  |
| --- |
| **// Inject scope, $routeParams, and cardService**  cardAppControllers.controller('DetailCtrl', ['$scope', '$routeParams', 'cardService',  function ($scope, $routeParams, cardService) {  cardService.getCard($scope, $routeParams.numberID, $routeParams.suitID);  }]); |

Next, in the router, add a condition to select the detail view when the home address is provided along with number and suit parameters.

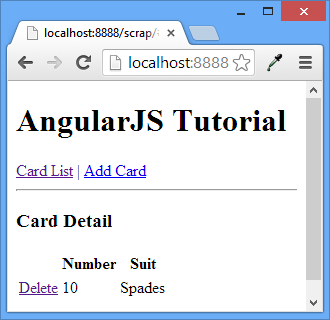
**js/app.js**

|  |
| --- |
| when('/home/number/:numberID/suit/:suitID', {  templateUrl: 'views/detail.html', controller: 'DetailCtrl' }). |

Note that back in Example 6, we added tags tags to the list view, to pass the card number and suit to the controller:

<td><a href="#/home/number/**{{**card.number**}}**/suit/**{{**card.suit**}}**">View</a></td>

When you run the program, it will function as before. This time though the detail view for each card is also enabled.



### Example 9: Adding a Delete Capability to the Application (tested 23 Nov ~ Shaun)

🞑 This example starts with Example 8 and adds the ability to delete a card. It offers a little more practice using the AngularJS structures we have discussed so far.

First, we need to define a service that allows deletion of data. Notice that instead of assigning card data to the scope, we use JSON to add a new model called ‘message;' it has a ‘Msg’ attribute that indicates whether the deletion is successful.

**js/services.js**

|  |
| --- |
| deleteCard: function($scope, cardNum, cardSuit) {  **/\***  **// LIVE WEB SERVICE**  **var url = "http://localhost:1353/api/cards?number=" + cardNum + "&suit=" + cardSuit;**  **$http.delete(url).success(function(data) {**  **$scope.message = data;**  **});**  **\*/**  $scope.message = { "Msg": cardNum.toUpperCase() + ' '  + cardSuit + ' has been deleted.'};  }, |

Next, add a controller that passes the card number and suit to the cardService for deletion.

**js/controllers.js**

|  |
| --- |
| cardAppControllers.controller('DeleteCtrl', ['$scope', '$routeParams', 'cardService',  function DeleteCtrl($scope, $routeParams, cardService) {  cardService.deleteCard($scope, $routeParams.numberID, $routeParams.suitID);  }]); |

Now, let’s add a confirmation view to display the deletion status message that the service returns.

**views/confirm.html**

|  |
| --- |
| <p ng-model = "message">  **{{**message.Msg**}}**  </p> |

Finally, add in this routing block to select the confirm.html template and deletion controller.

**js/app.js**

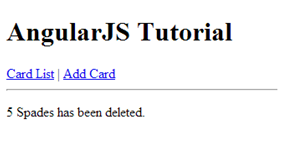
|  |
| --- |
| when('/delete/number/:numberID/suit/:suitID', {  templateUrl: 'views/confirm.html', controller: 'DeleteCtrl' }). |

The deletion is already enabled from the list and detail views since a deletion link already exists for each card:

<td> | <a href="#/delete/number/**{{**card.number**}}**/suit/**{{**card.suit**}}**">Delete</a></td>

When the program runs now you will have the ability to delete any card from the list or detail views when you click the delete link beside a specific card.

|  |
| --- |
| scope.model  url 🡪 route 🡪 controller 🡪 service 🡪 datastore 🡪 controller 🡪 view 🡪 directives |



# Managing Form Inputs

AngularJS helps to automate the management of user input. The functionality ranges from detecting changes of state to handling the automation of validation.

## Handling Button Clicks

Before we look more closely at how AngularJS helps with automating form inputs, we first need to examine how Angular submits forms. We can use the AngularJS **ng-click** directive to trigger the click event. The attribute of the ng-click directive specifies both the function to call on-click and any function parameters. The target function resides in the scope of the controller that we specify in the form’s scope. In the sample below, the ng-click directive is passing the *userAddress* model to the *update()* function that resides within the *Ctrl* controller's scope.

<form name="myForm" ng-controller="Ctrl">

City: <input name="city" ng-model="userAddress"><br/>

<button ng-click="update(userAddress)">Update</button>

</form>

### Example 10: Handling Button Click Events

🞑 Here is a more complete look at how to implement the *ng-click* directive which passes the *userAddress* model to the *update()* function within the *Ctrl* controller scope.

**index.html**

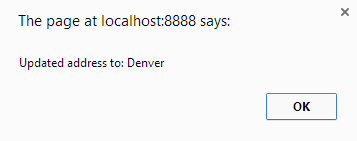
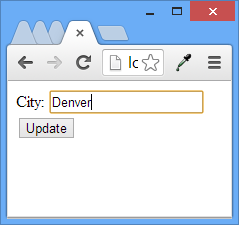
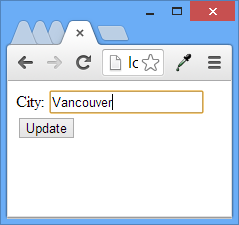
|  |
| --- |
| <!doctype html>  <html ng-app>  <head>  <title>Starting Angular</title>  </head>  <body>  <form name="myForm" ng-controller="Ctrl">  City: <input name="city" ng-model="userAddress"><br />  <button ng-click="update(userAddress)">Update</button>  </form>  <!--script at end for performance-->  <script src="http://code.angularjs.org/angular-1.2.0-8336b3a/angular.min.js"></script>  <script src="script.js"></script>  </body>  </html> |

In the controller, we add the function to the scope service.

**script.js**

|  |
| --- |
| function Ctrl($scope) {  $scope.userAddress = 'Vancouver';  $scope.update = function (input) {  alert("Updated address to: " + input);  };  } |

When the page loads, the **controller** accesses the **scope** and sets it's userAddress **model** with default **data**. The controller also sets the scope's update property with a **function**. The model data is what initially populates the form input. The user may change this value through the input box. Clicking the update button triggers the *update()* function that we added to the scope within the controller. An alert box displays.



## Hiding, Showing, Disabling, and Enabling Inputs

AngularJS directives can help to automate the hiding, showing, enabling, and disabling of inputs through pre-defined directives (see Table 2).

### Table 2: Hiding, Showing, Disabling, and Enabling Inputs

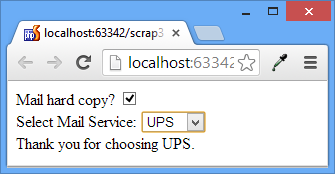
|  |  |
| --- | --- |
| **Directive** | **Function** |
| ng-hide | Hides an element when the expression is true. |
| ng-show | Shows an element when the expression is true. |
| ng-disabled | Disables an element when the expression is true. |

### Example 11: Showing, Hiding, Enabling, and Disabling

🞑 This example demonstrates how to show, hide, enable, and disable controls with AngularJS directives. The form prompts the user to select a checkbox if they wish to have a hard copy. A select input for a mail service is disabled if the checkbox is not checked. When the checkbox is checked, the select is enabled. Then, once the user selects a mail service, as long as the checkbox is selected, a thank you message is displayed which displays the name of the mail service with a reference to the model through an expression.

|  |
| --- |
| <!doctype html>  <html ng-app>  <head>  <script src="http://code.angularjs.org/angular-1.2.0-8336b3a/angular.min.js"></script>  </head>  <body>  <form name="myForm">  Mail hard copy?  <input type="checkbox" ng-model="checked"><br/>  Select Mail Service:  <select id="transferTo" ng-disabled="!checked" ng-model="mailService">  <option>UPS</option>  <option>Fedex</option>  </select>  <br/>  **<!-- Only show if checkbox is checked and mailService is selected -->**  <div ng-show="checked && mailService">Thank you for choosing **{{**mailService**}}**.</div>  </form>  </body>  </html> |

The form below shows the thank you message when both the checkbox is checked and a mail service has been selected. Notice that we do not have to write any JavaScript!



## Validating Form Inputs

Like any good data driven framework, AngularJS helps to automate input validation on the client.

### Validating Forms

It is possible to determine quickly if a form is valid with the expression **formName.$valid**. It is also possible to determine quickly if a control is valid with the expression **formName.inputName.$valid**.

### Validating Inputs

Most of the input controls implement a **required directive**. Table 3 lists several common **validation directives** that we can apply to a text input along with **validty** **expressions** that report their validity.

### Table 3: 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 12: 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. We complement these with expressions that relate to each type of validation. For each type of directive, this displays an error message when the invalid state is true. As well, the example displays the valid state for the control and the form. Here are the form tags:

**index.html**

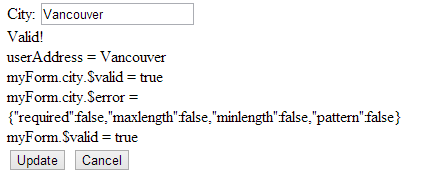
|  |
| --- |
| <!doctype html>  <html ng-app>  <head>  <script src="http://code.angularjs.org/angular-1.2.0-8336b3a/angular.min.js"></script>  <script src="script.js"></script>  </head>  <body>  <form name="myForm" ng-controller="Ctrl">  **<!-- Input with validation -->**  City: <input name="city" ng-model="userAddress"  ng-minlength="3"  ng-maxlength="10"  ng-pattern="/^[A-z]+$/"  required><br/>  **<!-- Show validation error messages -->**  <span ng-show="myForm.city.$error.required">Required!</span>  <span ng-show="myForm.city.$error.minlength">Minimum 3 characters required!</span>  <span ng-show="myForm.city.$error.maxlength">More than 10 characters not permitted!</span>  <span ng-show="myForm.city.$error.pattern">Only alphabetical characters are permitted!</span>  <span ng-show="myForm.city.$valid">Valid!</span>  <span ng-show="!myForm.city.$valid">Not valid!</span><br>  **<!-- Show input and form validation status -->**  userAddress = **{{**userAddress**}}**<br>  myForm.city.$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> |

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.

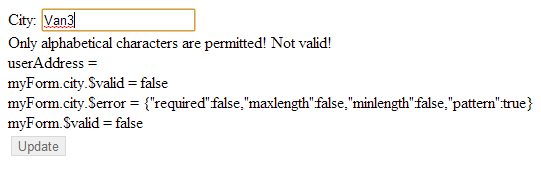
**js/controllers.js**

|  |
| --- |
| function Ctrl($scope) {  $scope.userAddress = 'Vancouver';  $scope.update = function (input) {  alert("Updated address to: " + input);  };  $scope.cancel = function (input) {  alert("Cancelled changes to: " + input);  };  } |

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.



## 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. Figure 1 displays several class names that Angular dynamically generates in the browser at run time when running Example 12.

Figure 1



### Example 13: Validation Styling

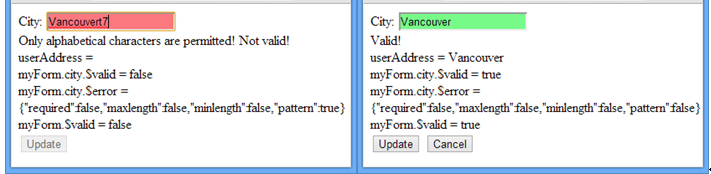
🞑 This example applies auto-generated css styles to the input box in Example 12. When the input is invalid the background color is set to red and when the input box is valid it is set to green. To begin, place the following styling block within the head tags of Example 12.

|  |
| --- |
| <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="Ctrl" class="css-form" > |

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



### Example 14: Form Inputs with Validation

🞑 Now that we have discussed several form automation techniques, let’s create a form that adds a playing card to the collection in our cards application. To start, begin with the card application solution from Example 9. Next, add the following service which adds a playing carding card.

**js/services.js**

|  |
| --- |
| addCard : function ($scope, cardNum, cardSuit) {  **/\***  **// LIVE WEB SERVICE**  **var url = "http://localhost:1353/api/cards?number=" + cardNum + "&suit=" + cardSuit;**  **$http.post(url).success(function(data) {**  **$scope.message = data;**  **});**  **\*/**  $scope.message = { "Msg": cardNum.toUpperCase() + ' '  + cardSuit + ' has been added.'};  }, |

Next, add the following controller code. Note that we define the two functions within the scope of the *AddCtrl* controller. The *add()* function redirects the application to the *AddConfirmCtrl* controller which then adds the card and displays a success or fail message in the confirmation view. The *reset()* function clears the object data in the form.

**js/controllers.js**

|  |
| --- |
| **// AddConfirmCtrl calls the service to add the card.**  cardAppControllers.controller('AddConfirmCtrl', ['$scope', '$routeParams', 'cardService',  function ($scope, $routeParams, cardService) {  cardService.addCard($scope, $routeParams.numberID, $routeParams.suitID);  }]);  **// The $location service is injected to enable the redirect.**  cardAppControllers.controller('AddCtrl', ['$scope', '$location', 'cardService',  function AddCtrl($scope, $location, cardService) {  **// Create an empty object.**  $scope.master = {};  **// When add is clicked, redirect to the confirm view and controller where**  **// the card is actually added added.**  $scope.add = function (card) {  $location.path('/addconfirm/number/' + card.number + '/suit/' + card.suit);  };  **// When reset is clicked clear the ‘card’ model defined within the scope to**  **// clear the form data.**  $scope.reset = function () {  $scope.card = angular.copy($scope.master);  };  }]); |

The *add.html* view allows us to input a card and suit combination. The *ng-pattern* directive ensures that the number ranges from A, 1, 2…J, Q, or K. When the number input is incorrect, Angular displays an error message using the *ng-show* directive inside a span tag. Clicking the reset button calls the *reset()* function in the *AddCtrl* controller to clear the data from the form. The save button is only enabled if the form is valid. Clicking the save button calls *add()* to store the new card.

**views/add.html**

|  |
| --- |
| <form name="form" class="css-form">  Number:  **<!-- Apply card number validation with the ng-pattern and required directives -->**  <input type="text" ng-model="card.number" name="number"  ng-pattern="/^([aAkKjJQq23456789]{1}|(10){1})$/" required/><br />  **<!-- When the number input is invalid show text explaining the error. -->**  <span ng-show="form.number.$error.pattern" class="invalidText">  Card numbers can only be A,a,2-10,j,J,q,Q,k,K!</span><br/>  Suit:  <input type="radio" ng-model="card.suit" value="Clubs" required/>Clubs  <input type="radio" ng-model="card.suit" value="Hearts" />Hearts  <input type="radio" ng-model="card.suit" value="Spades" />Spades  <input type="radio" ng-model="card.suit" value="Diamonds" />Diamonds  <br />  **<!-- Call reset() function within AddCtrl controller. -->**  <button ng-click="reset()">RESET</button>  **<!-- Call add() function within AddCtrl controller. -->**  <button ng-click="add(card)" ng-disabled="form.$invalid ">SAVE</button>  </form> |

Next, add two conditions to the router that select appropriate views and controllers for adding a playing card. When there is a request for the *add.html* view, Angular will use the *AddCtrl* controller both to manage data and to handle click events. The *add()* function will receive post data and then redirect to the *confirm.html* page. Angular will use the *AddConfirmCtrl* controller to show either a success or fail message.

**js/app.js**

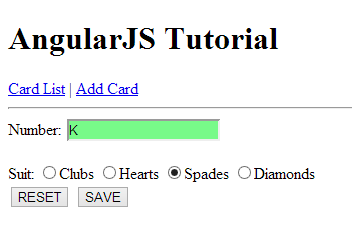
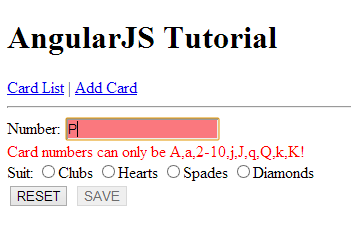
|  |
| --- |
| when('/add', {  templateUrl: 'views/add.html', controller: 'AddCtrl' }).  when('/addconfirm/number/:numberID/suit/:suitID', {  templateUrl: 'views/confirm.html', controller: 'AddConfirmCtrl' }).  **// If no route is selected then use the 'home' route.**  otherwise({ redirectTo: '/home' });  }); |

To apply styles that denote valid and invalid card additions, add the following css inside the head tag. The first two css classes apply colors to the background of the input box; this highlights the valid status. The third style applies red color to the error message when the number violates the required pattern.

**index.html**

|  |
| --- |
| <style type="text/css">  .css-form input.ng-invalid.ng-dirty { background-color: #FA787E; }  .css-form input.ng-valid.ng-dirty { background-color: #78FA89; }  span.invalidText { color: #ff0000; }  </style> |

An invalid form displays an invalid message and disables the save button. A valid form enables the save button. In either case, appropriate styling is applied.



# Conclusion

With such great support for two way data binding, automated form input and validation, styling, state and event management, AngularJS is a very attractive framework for any data driven web application.