The AngularJS UI-Router

By Jeffry Houser

A DotComIt Whitepaper

Table of Contents

The AngularJS UI-Router

Part of the AngularJS library is a router, named ngRoute. The ngRoute library allowed you to show different views of your Angular application based on the URL in the browser. This works great for simple applications; however it has some distinct limitations. Only a single view can be displayed per URL and nested views are not supported. Applications often have multiple views, such as a header, footer, and some main content. Main content is often split up into multiple sections. The ngRoute directive can only bring us so far.

The AngularUI team has created an alternate router, uiRouter, to addresses some of the limitations of the ngRoute. The ngRoute directive approaches an application as a collection of URLs, each one displaying a different view. The uiRouter looks at an application as a collection of states, and it allows multiple, nested states. This is a white paper about using the uiRouter within an AngularJS application.

Switching Between Two States

The first sample will show you how to switch between two separate states.

Create the Application Skeleton

First, import the Angular library and the ui-router library:

```
<script src="//code.angularjs.org/1.5.8/angular.min.js"></script>
<script
       src="//cdnjs.cloudflare.com/ajax/libs/angular-ui-router/0.3.1/angular-ui-router.min.js">
</script>
```

I reference both libraries from hosted CDN instead of copying them locally as part of my application.

Next, create an Angular app in a script block:

```
<script>
    angular.module('routerTestApp', ['ui.router']);
</script>
```

Configure the UI Router

A single configuration argument is passed into the Angular module, ui.router. This tells the Angular Module—routerTestApp—to load the uiRouter code and make it available to the application.

This sample will create two UI States, state1 and state2. The router configuration is done in an Angular config block.

```
angular.module('routerTestApp').config(['$stateProvider',
  function($stateProvider){
    $stateProvider
            .state('state1', {
                url: '/state1',
                templateUrl : 'states/state1.html'
            })
            .state('state2', {
                url: '/state2',
```

```
                templateUrl : 'states/state2.html'
            })
 }
]);
```

A service named $stateProvider is passed into the config block using the Angular dependency injection syntax. The $stateProvider service is part of the UI-router and is analogous to the $routeProvider used as part of ngRoute. The $stateProvider is used to define the two states of the application. In this case, a URL is defined. This is the value that will display in the URL when the state is displayed. A templateURL is used to refer to an HTML page that contains the view code that will be displayed when the state is active.

Create the HTML

First, we'll create the two HTML templates. The first is state1.html:

```
<h1>State 1</h1>
<a ui-sref="state2">Go to State 2</a>
```

This template is very simple. It includes a header with simple text stating 'State 1'. The anchor link is a bit more interesting. It includes an Angular directive named uiSref. This directive tells the uiRouter library to load a new state and change the URL whenever that link is clicked. The value of the uiSref directive is the name of the state that should be loaded. If the state doesn't exist, the anchor is immediately disabled and does not present the user with a link.

The state2.html template is almost identical to state1.html:

```
<h1>State 2</h1>
<a ui-sref="state1">Go to State 1</a>
```

The header says 'State 2' instead of 'State 1'. The link goes back to state1 instead of state2.

The main index needs some HTML to make this work too. First, add the ngApp directive to the body tag:

```
<body ng-app="routerTestApp">
```

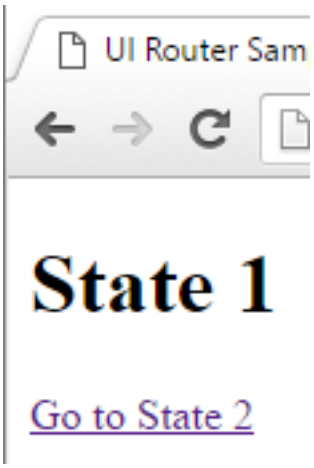
Then, add the ui-view:

```
<div ui-view></div>
```

The uiView directive tells the uiRouter directive to put the view here.

Test the App

Load the app in a browser, be sure to specify the current state with a URL like '/Index.html#/state1'. You should see the first state:



Click the link to open state 2. State2 should open:

The URL should change and state2 is displayed on the screen.

Play with the code here. The Plunker is a bit tricky to work with given the current codebase. No default state is specified and the initial load of this code will be blank if a state is not specified in the URL. Let's address that next.

Setting a Default State with the uiRouter

The biggest problem with the previous sample was that no default state was set, so the app would load a blank screen. That is something that needs to be addressed. I could use two different ways to set up a default state, and I'm going to explain both of them here.

Create a state with no URL

The first approach to creating a default state is to create a state with an empty URL attribute. Do this in the config that defines the states with the $stateProvider:

```
angular.module('routerTestApp').config(['$stateProvider',
  function($stateProvider){
    $stateProvider
```

```
            .state('state1', {
                url: '/state1',
                templateUrl : 'states/state1.html'
            })
            .state('state2', {
                url: '/state2',
                templateUrl : 'states/state2.html'
            })
            .state('default', {
                url: '',
                templateUrl : 'states/state1.html'
            })
  }
]);
```

The new state is added at the end. The name of the state is default, and the url parameter is an empty string. This will make it so that upon initial load the application—with no URL variables—the state1.hml template will load.

Now upon initial load the app, you'll see the correct state loaded even if you don't specify it in the URL:

Play with the app here.

As I congratulated myself on the creativity of this approach, I realized a limitation of this very quickly. If I want state1 to equal the default state, then every property of the state1 object will need to be defined as part of the state2 object, so it feels like some duplication of code.

Set a default state with the $urlRouteProvider

The uiRouter provides a standard way to set the default state. Into the config you can pass a uiRouter service named $urlRouteProvider. This service can be used to specify the default state:

```
angular.module('routerTestApp')
  .config(['$stateProvider','$urlRouterProvider',
    function($stateProvider,$urlRouterProvider){
      $urlRouterProvider.otherwise("/state1");
```

```
      // other state setup code here
    }
]);
```

The otherwise() function is called on the urlRouteProvider and it accepts a single string. If no state is loaded, then the otherwise state will be specified.

Load the new app to see the initial state load:

Play with the code here. I prefer using the $urlRouteProvider to set a default state on the application.

Adding a Controller to a State

When building Angular applications, the views are not very useful without an associated controller. An Angular controller provides the business logic behind the view. Thankfully, uiRouter provides a way to associate the state with a controller.

Create the Controllers

The first thing we're going to do is create the controllers. We can create a controller for each of the two views in the application:

```
angular.module('routerTestApp').controller('state1Controller',
  ['$scope',function($scope){
    $scope.title = "State 1 Controller"
  }]
);
angular.module('routerTestApp').controller('state2Controller',
  ['$scope',function($scope){
    $scope.title = "State 2 Controller"
  }]
);
```

These are two simple controllers. Each once contains a $scope variable, title, which will be used to replace the hard coded title used in previous examples.

Next, edit the config block to tell the $stateProvider which controller should go with which view:

```
$stateProvider
  .state('state1', {
      url: '/state1',
      templateUrl : 'states/state1.html',
      controller : 'state1Controller'
  })
  .state('state2', {
      url: '/state2',
      templateUrl : 'states/state2.html',
      controller : 'state2Controller'
  })
```

The object which defines the state has a new property, named controller. This controller's value is a string that refers to the name of the controller. When the view is setup, the controller will be associated with the view.

Create new HTML Templates

For this sample, we'll need two new state templates. Instead of using hard coded titles in the template page, it will use a title whose text is populated by the title $scope variable. This is the template for state1:

```
<h1>{{title}}</h1>
<a ui-sref="state2">Go to State 2</a>
```

This is the template for state 2:

```
<h1>{{title}}</h1>
<a ui-sref="state1">Go to State 1</a>
```

These two templates will show the dynamic title variable as the header, and also still include the links to switch between states.

Test The App

Now you can load the app and see the changes. This is the initial load:

This is what the app should look like after changing to state 2:

Play with the full code here.

In a real world application, you'll see lots more code in the controllers and interaction in the views.

Adding a Nested View

The samples we've seen so far have not demonstrated anything that couldn't be done using the ngRoute directive. Now is the time to advance. We'll add two nested views in the first state.

Define the Nested State

Creating a nested state is just like creating a non-nested state. It supports the sample object properties, such as the name, templateUrl, or controller. There are two ways to tell the uiRouter that this is a nested state. The first is to define the parent state, using a property named parent. I like this because it is a very overt. The second, more common, approach lies in the naming of the state. If a state includes a master state, followed by a period, followed by the state name, then that state is pegged as a substate.

Modify the config block to define two nested views:

```
$stateProvider
     .state('state1', {
         url: '/state1',
         templateUrl : 'states/state1.html',
         controller : 'state1Controller'
     })
     .state('state1.subview1', {
         url: '/state1.subview1',
         templateUrl : 'states/state1subview1.html',
     })
     .state('state1.subview2', {
         url: '/state1.subview2',
         templateUrl : 'states/state1subview2.html',
     })
     .state('state2', {
         url: '/state2',
         templateUrl : 'states/state2.html',
          controller : 'state2Controller'
     })
```

I repeated all states for comparison sakes. State1 and state2 are defined with the name 'state1' and few properties. The first substate under state1 is named 'state1.subview1'. The second is named 'state1.subview2'. Each view is defined with a url, and a template, but for the purposes of this example I Did not give them a custom controller. They will inherit from state1's controller in this case.

The period in the state name defines it as a substate, not the url property in the state object. The url could be anything you desire and is independent of the state name. For the purposes of these samples, I kept the state name and the url identical.

Create the Nested Views

Let's look at the HTML Behind the two subviews. This is subview 1:

```
Sub view 1
```

This is subview 2:

```
Sub view 2
```

Both are simplistic for demonstration purposes.

Activate the Nested Views

Let's look at the state1.html that will be used to activate the nested views:

```
<h1>{{title}}</h1>

<a ui-sref="state1">Hide Subview</a>
<a ui-sref="state1.subview1">Show Nested View 1</a>
<a ui-sref="state1.subview2">Show Nested View 2</a>
<div ui-view></div>
```

```
<a ui-sref="state2">Go to State 2</a>
```

The title display at the top and the link at the bottom to launch state 2 are the same as was used in previous samples. The middle part is new. It includes three hyperlinks, all using the uiSref directive to display and hide states. The first link goes directly to the state1; which is the default state. That will effectively hide all visible substates. The second link will open the first subview, and the third link will open the second subview. After that we see a div with the uiView directive on it. This is where the subview is displayed in the context of the view. It is similar to how we put the main view on the main index page.

Test the Nested Views

Load this in an app to test the nested views:

The initial state displays all the links, but does not display any of the nested view content. Click the Show Nested View 1 link:

Then click the Nested View 2 Link:

Test out the code here.

Deeper Nesting with Subviews

In the previous sample, we showed a sample of nesting a view inside another. This can be done multiple levels deep. Here is a sample.

Define the Deeper Subviews

First, we need to define a subview that goes multiple levels deep. For completeness, here are all the defined states of state1:

```
$stateProvider
       .state('state1', {
            url: '/state1',
            templateUrl : 'states/state1.html',
            controller : 'state1Controller'
       })
       .state('state1.subview1', {
            url: '/state1subview1',
            templateUrl : 'states/state1subview1.html',
       })
       .state('state1.subview1.deeper', {
            url: '/state1subview2deeper',
            templateUrl : 'states/state1subview1deeper.html',
       })
       .state('state1.subview2', {
            url: '/state1subview2',
            templateUrl : 'states/state1subview2.html',
       })
}]);
```

The default state is state1. Under state1, there are two states, state1.subview and state1.subview2. This code adds a brand new state, state1.subview.deeper. This is defined as proof of principle that the states can nest multiple levels deep. I left out state2 from the code here as it doesn't apply to this sample.

Modify the Subview HTML

Open up state1subview1.html file:

```
Sub view 1

<a ui-sref="state1.subview1">Hide Deeper Subview</a>
<a ui-sref="state1.subview1.deeper">Show Deeper Nested View 1</a>

<div ui-view></div>
```

The top is just text specifying the file being displayed. Next come two links. The first one loads the subview1, essentially hiding the second nested subview. The second links loads the second nested subview. Finally, the uiView directive is used on a div so uiRouter knows where to put the view that is three levels deep.

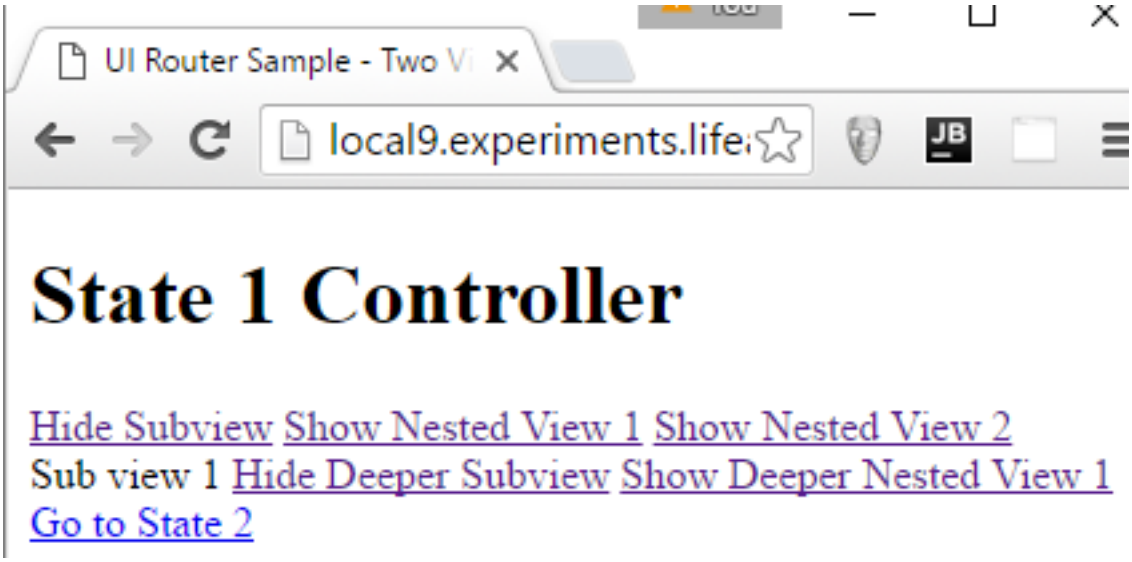
This is the template for the deeper nested view:

```
Deeper Even
```

It is simple with no additional functionality.

Test the Deeply Nested Views

Load the app and click the "Show Nested View 1" link:



Click the Show Deeper Nested View link, and the view that is two levels deep should display:

Play with the code here. You can use this approach to build complicated structures with dynamic content.

Add Multiple Views to a Single App

One aspect of the uiRouter is that you can add multiple views to a single Angular module. In this sample we're going to create an angular app that has a header, footer, and a main content area.

Create the Multiple Views Config

In previous samples, we had a lot of nested views. While nested views will work perfectly fine even if you have multiple views, I pulled them out of this sample for simplicity sake. To define the router, go back to the config block:

```
$stateProvider
 .state('state1', {
     url: '/state1',
     views : {
        "header"      : {templateUrl : 'states/header.html'},
        "mainContent" : {
                         templateUrl : 'states/state1.html',
                         controller  : 'state1Controller'
                        },
         "footer"     : {templateUrl : 'states/footer.html'}
     }
 })
 .state('state2', {
     url: '/state2',
     views : {
        "header"      : {templateUrl : 'states/header.html'},
        "mainContent" : {
                         templateUrl : 'states/state2.html',
```

```
                        controller  : 'state2Controller'
                        },
        "footer"      : {templateUrl : 'states/footer.html'}
    }
})
```

Each state is defined with a name and an object defining that state's properties. The state has two properties. The first defines the state's URL. This is the URL that will show in the browser bar when the state is loaded. The second piece is an object defining the views. Each element of the view object refers to an active view. It tells the router that in the current state, which template each view will display. All view properties are valid here including the templateUrl and controller. I specify the controller on the mainContent views, but for the sake of this sample the header and footer do not have a custom controller.

Create the Template HTML

Let's create the HTML templates. First, the header.html:

```
<a ui-sref="state1">State 1</a> |  <a ui-sref="state2">State 2</a>
```

I created a simple navigation bar, which includes link to switch between the two states.

Now, the footer.html:

```
Copyright notice, contact link, and other footer stuff here!
```

I just put in some placeholder text for common things you might find in the footer of a public web site. Both footer.html and header.html are reused between the two states.

Finally, take a look at the state1.html, which will be the main content for the state1 view:

```
<h1>{{title}}</h1>
Other State 1 content
```

The state1.html template references the title from the controller; just as it did in previous samples. It also displays some static placeholder text for state 1 content.

This is state2.html which will be displayed when the app is displaying state 2:

```
<h1>{{title}}</h1>
Other State 2 Content
```

The template for state2 is almost identical to state1. The main different is that the description text for fake content refers to State2 instead of state1.

Create the Views in the Main Template

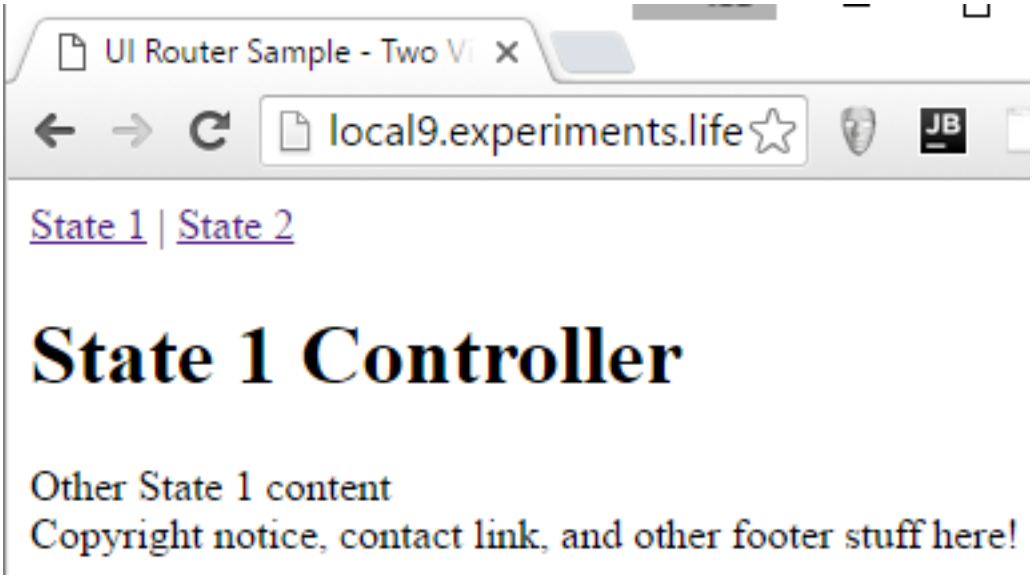
Finally, you'll have to add all three views to the main template of your HTML application.

```
<div ui-view="header"></div>
<div ui-view="mainContent"></div>
<div ui-view="footer"></div>
```

Previously, we just added the uiView directive with no value. However, when you have multiple views, you must name them so the uiRouter knows which one to direct at which app.

Test the App

Load the app, and you should see this:



The app loaded into the default state of State 1. You can see the navigation bar up top and the footer at the bottom, with the main content in the center. Click the State 2 link:

The main content changed, but the footer and header stayed the same.

Play with the code here.

Passing Parameters to a State

I'm going to modify the sample from the previous section to show you how to pass parameters to a state. We'll pass parameters to a headerController and use it to hide the link to the active state.

Resolve the Parameter

First, we'll need to modify the state definitions in the $stateProvider to pass a value to the views controller. In this case, we'll pass a hard coded string representing the name of the state. Here is the state definition for state1:

```
.state('state1', {
    url: '/state1',
    views : {
        "header" : {templateUrl : 'states/08header.html',
                    controller:'headerController',
                    resolve : {
                        selectedNav : function(){
                            return "state1"
                        }
            }
        },
        "mainContent" : {
                          templateUrl : 'states/07state1.html',
                          controller : 'state1Controller'
        },
        "footer" : {templateUrl : 'states/07footer.html'}
    }
})
```

The header view definition is the one that changed the most. First, a controller, headerController, was added. We'll look at headerController shortly. Then a resolve property was added. The resolve property defines services, on demand, that will be passed to the view's controller. The new services are defined by a function, which returns the value of the service. Whenever the state is activated, the service functions are executed, and once all the service functions are activated the controller is initialized.

The definition for state2 is almost identical to state1:

```
.state('state2', {
    url: '/state2',
    views : {
        "header" : {templateUrl : 'states/08header.html',
            controller:'headerController',
            resolve : {
                selectedNav : function(){
                    return "state2"
                }
```

```
            }
        },
        "mainContent" : {
                        templateUrl : 'states/07state2.html',
                        controller : 'state2Controller'
        },
        "footer" : {templateUrl : 'states/07footer.html'}
    }
})
```

The header view of state2 uses the same controller. It also passes in the same selectedNav service. The main difference is the value of that service. For the state1, the value of selectedNav is state1. For state2 the value of the selectedNav is state2.

Let's look at the headerController:

```
angular.module('routerTestApp').controller('headerController',
  ['$scope','selectedNav',function($scope,selectedNav){
    $scope.selectedNav = selectedNav;
  }]
);
```

The headerController is a simple controller. It accepts two arguments, the $scope and the selectedNav. All the code does is copy the selectedNav simple value into a $scope variable so it can be accessed inside the view template.

Remove the link to the Active State

To remove the link to the active state, open up the header.html file. You should see two links, similar to this:

```
<a ui-sref="state1" >State 1</a>
<a ui-sref="state2" >State 2</a>
```

To remove the states, the ng-hide will be used. We'll just put together a condition that compares the selectedNav controller variable to the actual selected state:

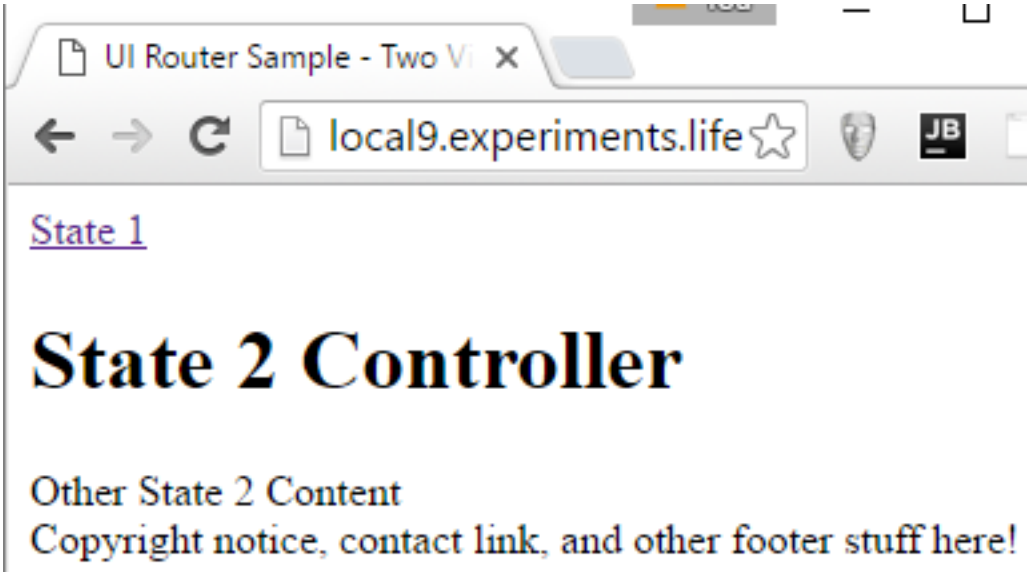
```
<a ui-sref="state1" ng-hide="selectedNav == 'state1'">State 1</a>
<a ui-sref="state2" ng-hide="selectedNav == 'state2'">State 2</a>
```

This will make things work.

Test the App

Run the app:

The app loads in the default state, state1. You'll see that the link to open state1 is hidden while state1 is displayed. Then click the State 2 link:



Once the app enters State2 the State2 link is hidden, and there is only a link to move back to state1.

Play with the App here!

Resolve can be a very powerful tool and is not limited to simple values. Objects, functions, promise objects, or other services can all be passed as values to a view controller using resolve.

Final Thoughts

The uiRouter is a great tool for Angular developers who want flexibility when piecing their applications together. As this white paper has shown, the complexity easily scales up and down depending upon your needs. I wish I had discovered it earlier than I did.