JavaScript (http://www.sitepoint.com/javascript/)

Creating a CRUD App in Minutes with Angular’s $resource

(http://www.sitepoint.com/author/spanda/)

Sandeep Panda (http://www.sitepoint.com/author/spanda/)

Published June 11, 2014

Tweet (Https://twitter.com/share?text=Creating+a+CRUD+App+in+Minutes+with+Angular%E2%80%99s+%24resource&via=sitepointdotcom)

Subscribe (Https://confirmsubscription.com/h/y/1FD5B523FA48AA2B)

Most Single Page Applications involve CRUD operations. If you are building CRUD operations using AngularJS, then you can leverage the power of the `$resource` service. Built on the top of the `$http` service, Angular’s `$resource` is a factory that lets you interact with RESTful backends easily. So, let’s explore `$resource` and use it to implement CRUD operations in Angular.

Prerequisites

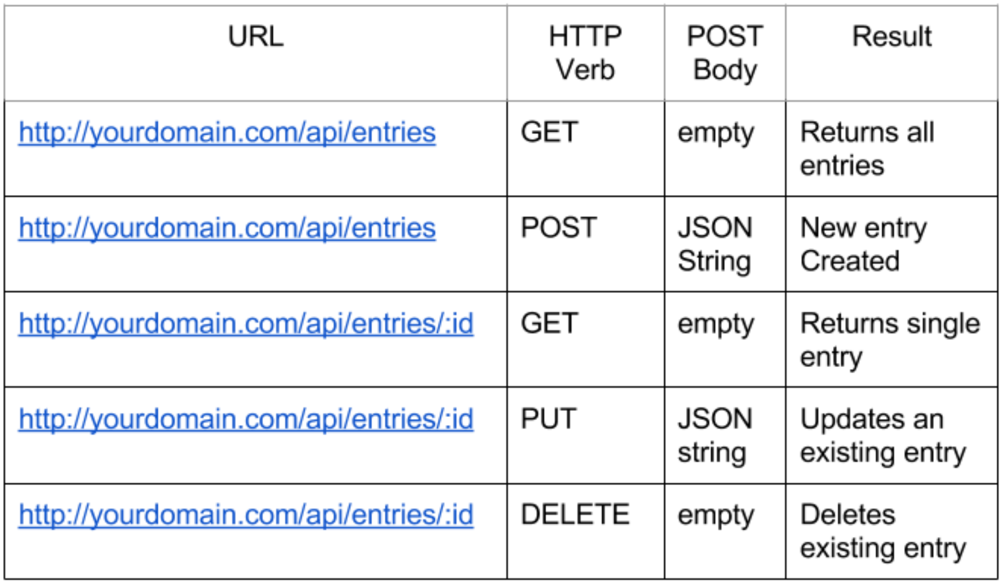
The `$resource` service doesn’t come bundled with the main Angular script. You need to download a separate file called `angular-resource.js` and include it in your HTML page. The script can be downloaded from http://cdnjs.cloudflare.com/ajax/libs/angular.js/1.2.16/angular-resource.min.js (http://cdnjs.cloudflare.com/ajax/libs/angular.js/1.2.16/angular-resource.min.js).

Also, your main app module should declare a dependency on the `ngResource` module in order to use `$resource`. The following example demonstrates how to do it:

```
angular.module('mainApp',['ngResource']); //mainApp is our main module
```

Getting Started

`$resource` expects a classic RESTful backend. This means you should have REST endpoints in the following format:



URL	HTTP Verb	POST Body	Result
http://yourdomain.com/api/entries	GET	empty	Returns all entries
http://yourdomain.com/api/entries	POST	JSON String	New entry Created
http://yourdomain.com/api/entries/:id	GET	empty	Returns single entry
http://yourdomain.com/api/entries/:id	PUT	JSON string	Updates an existing entry
http://yourdomain.com/api/entries/:id	DELETE	empty	Deletes existing entry

You can create the endpoints using the server side language of your choice. However, I have used Node + Express + MongoDB to design the RESTful API for the demo app. Once you have the URLs ready you can take help of `$resource` for interacting with these URLs. So, let's see how exactly `$resource` works.

How Does $resource Work?

To use `$resource` inside your controller/service you need to declare a dependency on `$resource`. The next step is calling the `$resource()` function with your REST endpoint, as shown in the following example. This function call returns a `$resource` class representation which can be used to interact with the REST backend.

```
angular.module('myApp.services').factory('Entry', function($resource) {
  return $resource('/api/entries/:id'); // Note the full endpoint address
});
```

The result of the function call is a resource class object which has the following five methods by default:

1. `get()`
2. `query()`
3. `save()`
4. `remove()`
5. `delete()`

Now, let’s see how we can use the `get()` , `query()` and `save()` methods in a controller:

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

angular.module('myApp.controllers').controller('ResourceController',function($scope, Entr
  var entry = Entry.get({ id: $scope.id }, function() {
    console.log(entry);
  }); // get() returns a single entry

  var entries = Entry.query(function() {
    console.log(entries);
  }); //query() returns all the entries

  $scope.entry = new Entry(); //You can instantiate resource class

  $scope.entry.data = 'some data';

  Entry.save($scope.entry, function() {
    //data saved. do something here.
  }); //saves an entry. Assuming $scope.entry is the Entry object
});
```

The `get()` function in the above snippet issues a GET request to `/api/entries/:id` . The parameter `:id` in the URL is replaced with `$scope.id` . You should also note that the function `get()` returns an empty object which is populated automatically when the actual data comes from server. The second argument to `get()` is a callback which is executed when the data arrives from server. This is a useful trick because you can set the empty object returned by `get()` to the `$scope` and refer to it in the view. When the actual data arrives and the object is populated, the data binding kicks in and your view is also updated.

The function `query()` issues a GET request to `/api/entries` (notice there is no :id) and returns an empty array. This array is populated when the data arrives from server. Again you can set this array as a `$scope` model and refer to it in the view using `ng-repeat` . You can also pass a callback to `query()` which is called once the data comes from server.

The `save()` function issues a POST request to `/api/entries` with the first argument as the post body. The second argument is a callback which is called when the data is saved. You might recall that the return value of the `$resource()` function is a resource class. So, in our case we can call `new Entry()` to instantiate an actual object out of this class, set various properties on it and finally save the object to backend.

Ideally, you will only use `get()` and `query()` on the resource class (`Entry` in our case). All the non GET methods like `save()` and `delete()` are also available in the instance obtained by calling `new Entry()` (call this a `$resource` instance). But the difference is that these methods are prefixed with a `$` . So, the methods available in the `$resource` instance (as opposed to `$resource` class) are:

1. `$save()`
2. `$delete()`
3. `$remove()`

For instance, the method `$save()` is used as following:

```
$scope.entry = new Entry(); //this object now has a $save() method
$scope.entry.$save(function() {
  //data saved. $scope.entry is sent as the post body.
});
```

We have explored the create, read and delete parts of CRUD. The only thing left is update. To support an update operation we need to modify our custom factory `Entity` as shown below.

```
angular.module('myApp.services').factory('Entry', function($resource) {
  return $resource('/api/entries/:id', { id: '@_id' }, {
    update: {
      method: 'PUT' // this method issues a PUT request
    }
  });
});
```

The second argument to `$resource()` is a hash indicating what should be the value of the parameter `:id` in the URL. Setting it to `@_id` means whenever we will call methods like `$update()` and `$delete()` on the resource instance, the value of `:id` will be set to the `_id` property of the instance. This is useful for PUT and DELETE requests. Also note the third argument. This is a hash that allows us to add any custom methods to the resource class. If the method issues a non-GET request it's made available to the `$resource` instance with a `$` prefix. So, let's see how to use our `$update` method. Assuming we are in a controller:

```
$scope.entry = Movie.get({ id: $scope.id }, function() {
  // $scope.entry is fetched from server and is an instance of Entry
  $scope.entry.data = 'something else';
  $scope.entry.$update(function() {
    //updated in the backend
  });
});
```

When the `$update()` function is called, it does the following:

1. AngularJS knows that `$update()` function will trigger a PUT request to the URL `/api/entries/:id`.
2. It reads the value of `$scope.entry._id` , assigns the value to `:id` and generates the URL.
3. Sends a PUT request to the URL with `$scope.entry` as the post body.

Similarly, if you want to delete an entry it can be done as following:

```
$scope.entry = Movie.get({ id: $scope.id }, function() {
  // $scope.entry is fetched from server and is an instance of Entry
  $scope.entry.data = 'something else';
  $scope.entry.$delete(function() {
    //gone forever!
  });
});
```

It follows the same steps as above, except the request type is DELETE instead of PUT.

We have covered all the operations in a CRUD, but left with a small thing. The `$resource` function also has an optional fourth parameter. This is a hash with custom settings. Currently, there is only one setting available which is `stripTrailingSlashes` . By default this is set to `true` , which means trailing slashes will be removed from the URL you pass to `$resource()` . If you want to turn this off you can do so like this:

```
angular.module('myApp.services').factory('Entry', function($resource) {
  return $resource('/api/entries/:id', { id: '@_id' }, {
    update: {
      method: 'PUT' // this method issues a PUT request
    }
  }, {
    stripTrailingSlashes: false
  });
});
```

By the way, I didn't cover each and every thing about `$resource`. What we covered here are the basics that will help you get started with CRUD apps easily. If you want to explore `$resource` in detail, you can go through the documentation (https://docs.angularjs.org/api/ngResource/service/$resource).

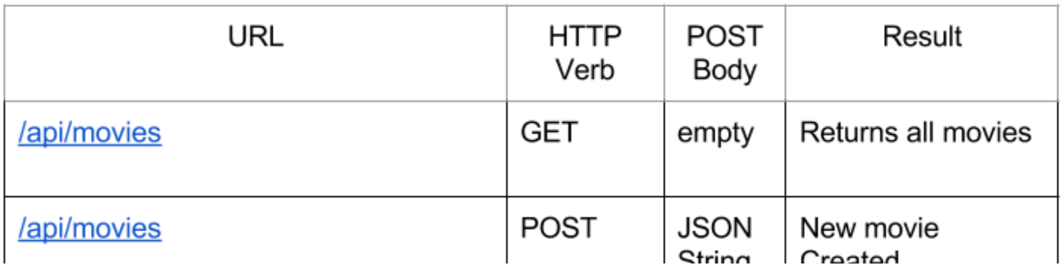
Building a Movie App

To reinforce the concepts of `$resource` let's build an app for movie lovers. This is going to be a SPA where users can add a new movie to our database, update an existing movie, and finally delete one. We will use `$resource` to interact with the REST API. You can check out a live demo of what we are going to build here (http://movieapp-sitepointdemos.rhcloud.com/).

Just note that the API I have built is CORS enabled, so it is possible for you to create an Angular app separately and use the URL `http://movieapp-sitepointdemos.rhcloud.com/` as the API. You can develop the Angular app and play around with it without worrying about the backend.

Our API

I have created a RESTful backend using Node and Express. Take a look at the following screenshot to get to know the API.



URL	HTTP Verb	POST Body	Result
/api/movies	GET	empty	Returns all movies
/api/movies	POST	JSON String	New movie Created

		string	created
/api/movies/:id	GET	empty	Returns single movie
/api/movies/:id	PUT	JSON string	Updates an existing movie
/api/movies/:id	DELETE	empty	Deletes existing movie

Directory Structure

Let’s start with the following directory structure for our AngularJS app:

```
movieApp
  /css
    bootstrap.css
    app.css
  /js
    app.js
    controllers.js
    services.js
  /lib
    angular.min.js
    angular-resource.min.js
    angular-ui-router.min.js
  /partials
    _form.html
    movie-add.html
    movie-edit.html
    movie-view.html
    movies.html
  index.html
```

Just note that we will be using Angular UI Router (https://github.com/angular-ui/ui-router) for routing.

Creating Our Service to Interact with REST Endpoints

As discussed in previous sections we will create a custom service that will use `$resource` internally to interact with the REST API. The service is defined in `js/services.js` .

`services.js:`

```
angular.module('movieApp.services', []).factory('Movie', function($resource) {
  return $resource('http://movieapp-sitepointdemos.rhcloud.com/api/movies/:id (http://movi
    update: {
      method: 'PUT'
    }
  });
});
```

The name of our factory is `Movie`. As we are using MongoDB, each movie instance has a property called `_id`. The rest is simple and straightforward.

Now that we have our service ready let's build views and controllers.

`index.html` : Building the App Entry Page

The `index.html` is our app entry point. To start we need to include all the required scripts and stylesheets in this page. We will use Bootstrap to quickly create the layout. Here is the content of `index.html`.

```html
<!DOCTYPE html>
  <html data-ng-app="movieApp">
  <head lang="en">
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <base href="/"/>
    <title>The Movie App</title>
    <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css"/>
    <link rel="stylesheet" type="text/css" href="css/app.css"/>
  </head>
  <body>
    <nav class="navbar navbar-default" role="navigation">
      <div class="container-fluid">
        <div class="navbar-header">
          <a class="navbar-brand" ui-sref="movies">The Movie App</a>
        </div>
        <div class="collapse navbar-collapse">
          <ul class="nav navbar-nav">
            <li class="active"><a ui-sref="movies">Home</a></li>
          </ul>
        </div>
      </div>
    </nav>
    <div class="container">
      <div class="row top-buffer">
        <div class="col-xs-8 col-xs-offset-2">
          <div ui-view></div> <!-- This is where our views will load -->
        </div>
      </div>
    </div>
    <script type="text/javascript" src="lib/angular.min.js"></script>
    <script type="text/javascript" src="js/app.js"></script>
    <script type="text/javascript" src="js/controllers.js"></script>
    <script type="text/javascript" src="js/services.js"></script>
    <script type="text/javascript" src="lib/angular-ui-router.min.js"></script>
    <script type="text/javascript" src="lib/angular-resource.min.js"></script>
  </body>
</html>
```

The markup is pretty self explanatory. Just pay special attention to `<div ui-view></div>`. The `ui-view` directive comes from UI Router module and acts as a container for our views.

Creating Main Module and States

Our main module and states are defined in `js/app.js` :

`app.js:`

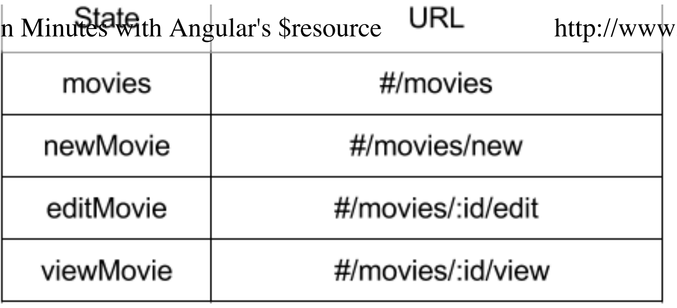
```
angular.module('movieApp', ['ui.router', 'ngResource', 'movieApp.controllers', 'movieApp.

angular.module('movieApp').config(function($stateProvider) {
  $stateProvider.state('movies', { // state for showing all movies
    url: '/movies',
    templateUrl: 'partials/movies.html',
    controller: 'MovieListController'
  }).state('viewMovie', { //state for showing single movie
    url: '/movies/:id/view',
    templateUrl: 'partials/movie-view.html',
    controller: 'MovieViewController'
  }).state('newMovie', { //state for adding a new movie
    url: '/movies/new',
    templateUrl: 'partials/movie-add.html',
    controller: 'MovieCreateController'
  }).state('editMovie', { //state for updating a movie
    url: '/movies/:id/edit',
    templateUrl: 'partials/movie-edit.html',
    controller: 'MovieEditController'
  });
}).run(function($state) {
  $state.go('movies'); //make a transition to movies state when app starts
});
```

So, our application has the following four states:

1. `movies`
2. `viewMovie`
3. `newMovie`
4. `editMovie`

Each state is composed of a `url` , `templateUrl` and `controller` . Also note that when our main module is loaded we make a transition to state `movies` showing all the movies in our system. Take a look at the following screenshot to know which state corresponds to what URL.



State	URL
movies	#/movies
newMovie	#/movies/new
editMovie	#/movies/:id/edit
viewMovie	#/movies/:id/view

Creating Templates

All of our templates are inside `partials` . Let's see what each of them does!

`_form.html:`

`_form.html` contains a simple form which allows users to enter data. Note that this form will be included by `movie-add.html` and `movie-edit.html` because both of them accept inputs from users.

Here is the content of `_form.html` :

```
<div class="form-group">
  <label for="title" class="col-sm-2 control-label">Title</label>
  <div class="col-sm-10">
    <input type="text" ng-model="movie.title" class="form-control" id="title" placeholder
  </div>
</div>
<div class="form-group">
  <label for="year" class="col-sm-2 control-label">Release Year</label>
  <div class="col-sm-10">
    <input type="text" ng-model="movie.releaseYear" class="form-control" id="year" placeh
  </div>
</div>
<div class="form-group">
  <label for="director" class="col-sm-2 control-label">Director</label>
  <div class="col-sm-10">
    <input type="text" ng-model="movie.director" class="form-control" id="director" place
  </div>
</div>
<div class="form-group">
  <label for="plot" class="col-sm-2 control-label">Movie Genre</label>
  <div class="col-sm-10">
    <input type="text" ng-model="movie.genre" class="form-control" id="plot" placeholder=
  </div>
</div>
<div class="form-group">
  <div class="col-sm-offset-2 col-sm-10">
    <input type="submit" class="btn btn-primary" value="Save"/>
  </div>
</div>
```

The template uses `ng-model` to bind various movie details to different properties of `scope` model `movie` .

`movie-add.html`

This template is used to accept user inputs and add a new movie to our system. Here is the content:

```
<form class="form-horizontal" role="form" ng-submit="addMovie()">
  <div ng-include="'partials/_form.html'"></div>
</form>
```

When the form is submitted the function `addMovie()` of the scope is called which in turn sends a POST request to server to create a new movie.

`movie-edit.html`:

This template is used to accept user inputs and update an existing movie in our system.

```
<form class="form-horizontal" role="form" ng-submit="updateMovie()">
  <div ng-include="'partials/_form.html'"></div>
</form>
```

Once the form is submitted the `scope` function `updateMovie()` is called which issues a PUT request to server to update a movie.

`movie-view.html`:

This template is used to show details about a single movie. The content looks like following:

```
<table class="table movietable">
  <tr>
    <td><h3>Details for {{movie.title}}</h3></td>
    <td></td>
  </tr>
  <tr>
    <td>Movie Title</td>
    <td>{{movie.title}}</td>
  </tr>
  <tr>
    <td>Director</td>
    <td>{{movie.director}}</td>
  </tr>
  <tr>
    <td>Release Year</td>
    <td>{{movie.releaseYear}}</td>
  </tr>
  <tr>
    <td>Movie Genre</td>
    <td>{{movie.genre}}</td>
  </tr>
</table>
<div>
  <a class="btn btn-primary" ui-sref="editMovie({id:movie._id})">Edit</a>
</div>
```

In the end there is an edit button. Once clicked it changes the state to `editMovie` with the movie `id` in the `$stateParams` .

movies.html

This template displays all the movies in the system.

```
<a ui-sref="newMovie" class="btn-primary btn-lg nodecoration">Add New Movie</a>

<table class="table movietable">
  <tr>
    <td><h3>All Movies</h3></td>
    <td></td>
  </tr>
  <tr ng-repeat="movie in movies">
    <td>{{movie.title}}</td>
    <td>
      <a class="btn btn-primary" ui-sref="viewMovie({id:movie._id})">View</a>
      <a class="btn btn-danger"  ng-click="deleteMovie(movie)">Delete</a>
    </td>
  </tr>
</table>
```

It loops through all the `movie` objects obtained from the API and displays the details. There is also a button `Add New Movie` which changes the state to `newMovie` . As a result a new route loads and we can create a new movie entry.

For each movie there are two buttons, `View` and `Delete` . `View` triggers a state transition so that the details for the movie are displayed. `Delete` button deletes the movie permanently.

Creating Controllers

Each state has a controller. So, in total we have four controllers for four states. All the controllers go into `js/controllers.js` . The controllers just utilize our custom service `Movie` and work the way we have discussed above. So, here is how our controllers look.

`controllers.js:`

```
angular.module('movieApp.controllers', []).controller('MovieListController', function($sc
  $scope.movies = Movie.query(); //fetch all movies. Issues a GET to /api/movies

  $scope.deleteMovie = function(movie) { // Delete a movie. Issues a DELETE to /api/movie
    if (popupService.showPopup('Really delete this?')) {
      movie.$delete(function() {
        $window.location.href = ''; //redirect to home
      });
    }
  };
}).controller('MovieViewController', function($scope, $stateParams, Movie) {
  $scope.movie = Movie.get({ id: $stateParams.id }); //Get a single movie.Issues a GET to
}).controller('MovieCreateController', function($scope, $state, $stateParams, Movie) {
  $scope.movie = new Movie();  //create new movie instance. Properties will be set via ng

  $scope.addMovie = function() { //create a new movie. Issues a POST to /api/movies
    $scope.movie.$save(function() {
      $state.go('movies'); // on success go back to home i.e. movies state.
    });
  };
}).controller('MovieEditController', function($scope, $state, $stateParams, Movie) {
  $scope.updateMovie = function() { //Update the edited movie. Issues a PUT to /api/movie
    $scope.movie.$update(function() {
      $state.go('movies'); // on success go back to home i.e. movies state.
    });
  };

  $scope.loadMovie = function() { //Issues a GET request to /api/movies/:id to get a movi
    $scope.movie = Movie.get({ id: $stateParams.id });
  };

  $scope.loadMovie(); // Load a movie which can be edited on UI
});
```

Conclusion

Assuming the app is deployed under `localhost/movieApp` , you can access it at `http://localhost/movieApp/index.html` . If you are a movie lover, you can start adding your favorite movies too! The source code for the app developd in this article is available on GitHub (https://github.com/jsprodotcom/source/blob/master/movieApp.zip) for download.

I hope you enjoyed this tutorial! Feel free to comment if you'd like to add something.

(http://www.sitepoint.com/author/spanda/)

Sandeep Panda (http://www.sitepoint.com/author/spanda/)

Sandeep is a web developer and writer with a passion for Java, JavaScript and HTML5. He has 4+ years of experience programming for the web. He loves experimenting with new technologies as they emerge and his current love interest is AngularJS. Sandeep always looks forward to working with new and exciting opportunities. While not programming, he can be found playing games and listening to music. You can also find him blogging about various web technologies on his own website HTMLxprs (http://www.htmlxprs.com).

(https://twitter.com/Sandeepg33k)
(http://facebook.com/sandeep.panda92)
(http://www.linkedin.com/pub/sandeep-panda/45/768/b23)
(https://plus.google.com/+SandeepPanda)

You might also like:

Single Page App with Laravel and EmberJS (http://www.sitepoint.com/single-page-app-laravel-emberjs/)

Book: Jump Start JavaScript (https://learnable.com/books/jump-start-javascript?utm_source=sitepoint&utm_medium=related-items&utm_content=js-javascript)

Best Practices REST API from Scratch – Implementation (http://www.sitepoint.com/best-practices-rest-api-scratch-implementation/)

JUMP START
Html5
BASICS

22 Comments SitePoint Login

Sort by Best Share Favorite ★

Join the discussion…

guest · 3 months ago

the popupService is missed in the tutorial but present in the demo code

4 ^ | v · Reply · Share ›

amitmojumder · 4 months ago

This demo code does nothing. I tried to run it from XAMPP. :(

3 ^ | v · Reply · Share ›

> **crime_master_gogo** → amitmojumder · 2 months ago
>
> use it as a starting point and write your own :)
>
> 1 ^ | v · Reply · Share ›

Wayou Liu · 5 months ago

nice tutorial!

2 ^ | v · Reply · Share ›

Wu Wenbin · 2 months ago

where's ur model layer? in most case, it's not good practice to invoke backend interface in the controller immediately. use the $resource like above, u will get sucked when u want to asyn model between views.

1 ^ | v · Reply · Share ›

Ravi Mone · 24 days ago

Hi Sandeep Panda,

It is good resource I found on $resource in AngularJS, Many thanks for this post, But when I tried to use this build, I got this error:

XMLHttpRequest cannot load http://movieapp-13434.onmodulu.... No 'Access-Control-Allow-Origin' header is present on the requested resource. ....

In the web console.
Was really feeling sad, for not seeing the resource feature in action, It is basically CORS issue, so please make that fix.

I am eagerly waiting to see this functionality in action, In the build.

^ | v · Reply · Share ›

rav · a month ago

am getting blank page while opening index.html. pls help me

^ | v · Reply · Share ›

Amir · a month ago

Great tutorial. Thanks Sandeep. Please keep bringing more tutorials.

^ | v · Reply · Share ›

About

About us (/about-us/)
Advertise (/advertising)
Legals (/legals)
Feedback (mailto:feedback@sitepoint.com)
Write for Us (/write-for-us)

Our Sites

Learnable (https://learnable.com)
Reference (http://reference.sitepoint.com)
Hosting Reviews (/hosting-reviews/)
Web Foundations (/web-foundations/)

Connect

(/feed) (/newsletter) (https://www.facebook.com/sitepoint) (http://twitter.com/sitepointdotcom) (https://plus.google.com/+sitepoint)