عنوان: **ساخت یک بلاگ ساده با Ember.js، قسمت پنجم**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۲۰ ۸:۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, ASP.NET Web API, jQuery, SPA, EmberJS

مقدمات ساخت بلاگ مبتنی بر ember.js در قسمت قبل به پایان رسید. در این قسمت صرفا قصد داریم بجای استفاده از HTML 5 local storage از یک REST web service مانند یک ASP.NET Web API Controller و یا یک ASP.NET MVC Controller استفاده کنیم و اطلاعات نهایی را به سرور ارسال و یا از آن دریافت کنیم.

تنظیم Ember data برای کار با سرور

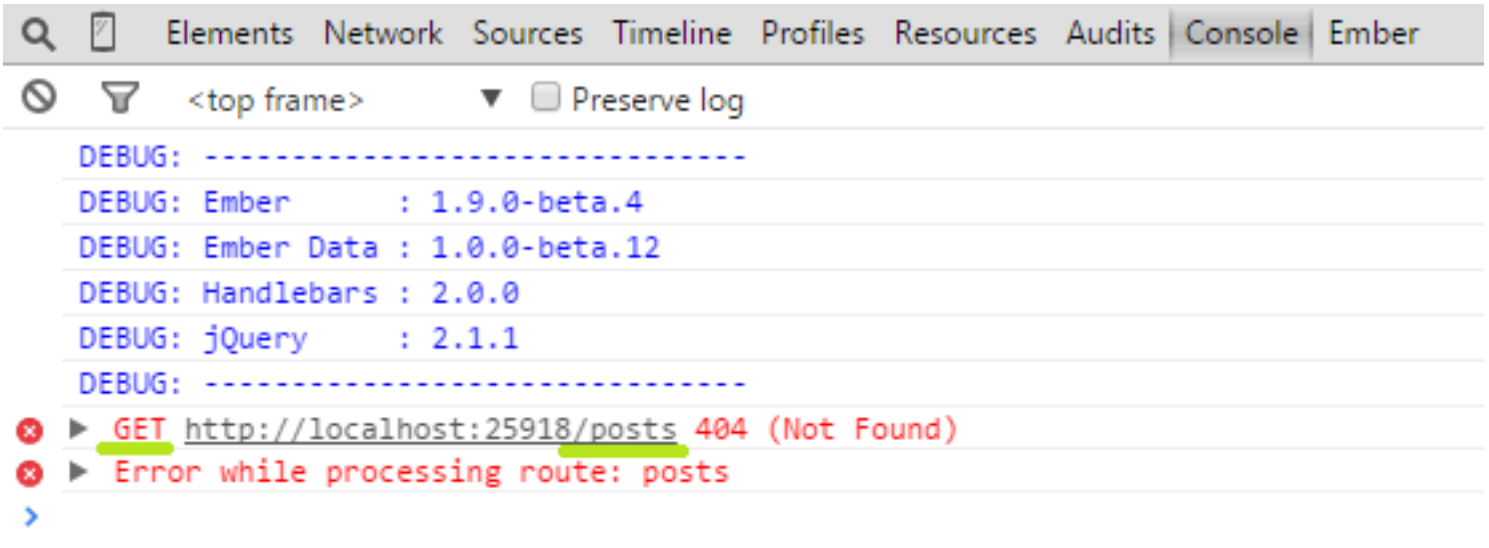
Ember data به صورت پیش فرض و در پشت صحنه با استفاده از Ajax برای کار با یک REST Web Service طراحی شده‌است و کلیه تبادلات آن نیز با فرمت JSON انجام می‌شود. بنابراین تمام کدهای سمت کاربر قسمت قبل نیز در این حالت کار خواهند کرد. تنها کاری که باید انجام شود، حذف تنظیمات ابتدایی آن برای کار با HTML 5 local storage است.
برای این منظور ابتدا فایل index.html را گشوده و سپس مدخل localstorage_adapter.js را از آن حذف کنید:

```
<!--<script src="Scripts/Libs/localstorage_adapter.js" type="text/javascript"></script>-->
```

همچنین دیگر نیازی به store.js نیز نمی‌باشد:

```
<!--<script src="Scripts/App/store.js" type="text/javascript"></script>-->
```

اکنون برنامه را اجرا کنید، چنین پیام خطایی را مشاهده خواهید کرد:



همانطور که عنوان شد، ember data به صورت پیش فرض با سرور کار می‌کند و در اینجا به صورت خودکار، یک درخواست Get را به آدرس http://localhost:25918/posts جهت دریافت آخرین مطالب ثبت شده، ارسال کرده‌است و چون هنوز وب سرویسی در برنامه تعریف نشده، با خطای 404 و یا یافت نشد، مواجه شده‌است.
این درخواست نیز بر اساس تعاریف موجود در فایل Scripts\Routes\posts.js، به سرور ارسال شده‌است:

```
Blogger.PostsRoute = Ember.Route.extend({
    model: function () {
        return this.store.find('post');
    }
```

```
});
```

Ember data شبیه به یک ORM عمل می‌کند. تنظیمات ابتدایی آن‌را تغییر دهید، بدون نیازی به تغییر در کدهای اصلی برنامه، می‌تواند با یک منبع داده جدید کار کند.

تغییر تنظیمات پیش فرض آغازین Ember data

آدرس درخواستی http://localhost:25918/posts به این معنا است که کلیه درخواست‌ها، به همان آدرس و پورت ریشه‌ی اصلی سایت ارسال می‌شوند. اما اگر یک ASP.NET Web API Controller را تعریف کنیم، نیاز است این درخواست‌ها، برای مثال به آدرس api/posts ارسال شوند؛ بجای /posts.
برای این منظور پوشه‌ی جدید Scripts\Adapters را ایجاد کرده و فایل web_api_adapter.js را با این محتوا به آن اضافه کنید:

```
DS.RESTAdapter.reopen({
      namespace: 'api'
});
```

سپس تعریف مدخل آن‌را نیز به فایل index.html اضافه نمائید:

```
<script src="Scripts/Adapters/web_api_adapter.js" type="text/javascript"></script>
```

تعریف فضای نام در اینجا سبب خواهد شد تا درخواست‌های جدید به آدرس api/posts ارسال شوند.

تغییر تنظیمات پیش فرض ASP.NET Web API

در سمت سرور، بنابر اصول نامگذاری خواص، نام‌ها با حروف بزرگ شروع می‌شوند:

```
namespace EmberJS03.Models
{
    public class Post
    {
        public int Id { set; get; }
        public string Title { set; get; }
        public string Body { set; get; }
    }
}
```

اما در سمت کاربر و کدهای اسکریپتی، عکس آن صادق است. به همین جهت نیاز است که CamelCasePropertyNamesContractResolver را در JSON.NET تنظیم کرد تا به صورت خودکار اطلاعات ارسالی به کلاینت‌ها را به صورت camel case تولید کند:

```
using System;
using System.Web.Http;
using System.Web.Routing;
using Newtonsoft.Json.Serialization;

namespace EmberJS03
{
    public class Global : System.Web.HttpApplication
    {

        protected void Application_Start(object sender, EventArgs e)
        {
            RouteTable.Routes.MapHttpRoute(
               name: "DefaultApi",
               routeTemplate: "api/{controller}/{id}",
               defaults: new { id = RouteParameter.Optional }
               );

            var settings =
GlobalConfiguration.Configuration.Formatters.JsonFormatter.SerializerSettings;
            settings.ContractResolver = new CamelCasePropertyNamesContractResolver();
```

```
        }
    }
}
```

نحوه‌ی صحیح بازگشت اطلاعات از یک ASP.NET Web API جهت استفاده در Ember data

با تنظیمات فوق، اگر کنترلر جدیدی را به صورت ذیل جهت بازگشت لیست مطالب تهیه کنیم:

```
namespace EmberJS03.Controllers
{
    public class PostsController : ApiController
    {
        public IEnumerable<Post> Get()
        {
            return DataSource.PostsList;
        }
    }
}
```

با یک چنین خطایی در سمت کاربر مواجه خواهیم شد:

```
WARNING: Encountered "0" in payload, but no model was found for model name "0" (resolved model name
using DS.RESTSerializer.typeForRoot("0"))
```

این خطا از آنجا ناشی می‌شود که Ember data، اطلاعات دریافتی از سرور را بر اساس قرارداد JSON API دریافت می‌کند. برای حل این مشکل راه‌حل‌های زیادی مطرح شده‌اند که تعدادی از آن‌ها را در لینک‌های زیر می‌توانید مطالعه کنید:

http://jsonapi.codeplex.com

https://github.com/xqiu/MVCSPAWithEmberjs

https://github.com/rmichela/EmberDataAdapter

https://github.com/MilkyWayJoe/Ember-WebAPI-Adapter

http://blog.yodersolutions.com/using-ember-data-with-asp-net-web-api

http://emadibrahim.com/2014/04/09/emberjs-and-asp-net-web-api-and-json-serialization

و خلاصه‌ی آن‌ها به این صورت است:

خروجی JSON تولیدی توسط ASP.NET Web API چنین شکلی را دارد:

```
[
  {
    Id: 1,
    Title: 'First Post'
  }, {
    Id: 2,
    Title: 'Second Post'
  }
]
```

اما Ember data نیاز به یک چنین خروجی دارد:

```
{
  posts: [{
    id: 1,
    title: 'First Post'
  }, {
    id: 2,
    title: 'Second Post'
  }]
}
```

به عبارتی آرایه‌ی مطالب را از ریشه‌ی posts باید دریافت کند (مطابق فرمت JSON API). برای انجام اینکار یا از لینک‌های معرفی شده استفاده کنید و یا راه حل ساده‌ی ذیل هم پاسخگو است:

```
using System.Web.Http;
using EmberJS03.Models;

namespace EmberJS03.Controllers
{
    public class PostsController : ApiController
    {
        public object Get()
        {
            return new { posts = DataSource.PostsList };
        }
    }
}
```

در اینجا ریشه‌ی posts را توسط یک anonymous object ایجاد کرده‌ایم.
اکنون اگر برنامه را اجرا کنید، در صفحه‌ی اول آن، لیست عناوین مطالب را مشاهده خواهید کرد.

تاثیر قرارداد JSON API در حین ارسال اطلاعات به سرور توسط Ember data

در تکمیل کنترلرهای Web API مورد نیاز (کنترلرهای مطالب و نظرات)، نیاز به متدهای Post، Update و Delete هم خواهد بود.
دقیقا فرامین ارسالی توسط Ember data توسط همین HTTP Verbs به سمت سرور ارسال می‌شوند. در این حالت اگر متد Post کنترلر نظرات را به این شکل طراحی کنیم:

```
public HttpResponseMessage Post(Comment comment)
```

کار نخواهد کرد؛ چون مطابق فرمت JSON API ارسالی توسط Ember data، یک چنین شیء JSON ایی را دریافت خواهیم کرد:

```
{"comment":{"text":"data...","post":"3"}}
```

بنابراین Ember data چه در حین دریافت اطلاعات از سرور و چه در زمان ارسال اطلاعات به آن، اشیاء جاوا اسکریپتی را در یک ریشه‌ی هم نام آن شیء قرار می‌دهد.
برای پردازش آن، یا باید از راه حل‌های ثالث مطرح شده در ابتدای بحث استفاده کنید و یا می‌توان مطابق کدهای ذیل، کل اطلاعات JSON ارسالی را توسط کتابخانه‌ی JSON.NET نیز پردازش کرد:

```
namespace EmberJS03.Controllers
{
    public class CommentsController : ApiController
    {
        public HttpResponseMessage Post(HttpRequestMessage requestMessage)
        {
            var jsonContent = requestMessage.Content.ReadAsStringAsync().Result;
            // {"comment":{"text":"data...","post":"3"}}
            var jObj = JObject.Parse(jsonContent);
            var comment = jObj.SelectToken("comment", false).ToObject<Comment>();


            var id = 1;
            var lastItem = DataSource.CommentsList.LastOrDefault();
            if (lastItem != null)
            {
                id = lastItem.Id + 1;
            }
            comment.Id = id;
            DataSource.CommentsList.Add(comment);

            // ارسال آی دی با فرمت خاص مهم است
            return Request.CreateResponse(HttpStatusCode.Created, new { comment = comment });
        }
    }
```

```
}
```

در اینجا توسط requestMessage به محتوای ارسال شده‌ی به سرور که همان شیء JSON ارسالی است، دسترسی خواهیم داشت. سپس متد JObject.Parse، آن‌را به صورت عمومی تبدیل به یک شیء JSON می‌کند و نهایتا با استفاده از متد SelectToken آن می‌توان ریشه‌ی comment و یا در کنترلر مطالب، ریشه‌ی post را انتخاب و سپس تبدیل به شیء Comment و یا Post کرد.
همچنین فرمت return نهایی هم مهم است. در این حالت خروجی ارسالی به سمت کاربر، باید مجددا با فرمت JSON API باشد؛ یعنی باید comment اصلاح شده را به همراه ریشه‌ی comment ارسال کرد. در اینجا نیز anonymous object تهیه شده، چنین کاری را انجام می‌دهد.

Lazy loading در Ember data

تا اینجا اگر برنامه را اجرا کنید، لیست مطالب صفحه‌ی اول را مشاهده خواهید کرد، اما لیست نظرات آن‌ها را خیر؛ از این جهت که ضرورتی نداشت تا در بار اول ارسال لیست مطالب به سمت کاربر، تمام نظرات متناظر با آن‌ها را هم ارسال کرد. بهتر است زمانیکه کاربر یک مطلب خاص را مشاهده می‌کند، نظرات خاص آن‌را به سمت کاربر ارسال کنیم.
در تعاریف سمت کاربر Ember data، پارامتر دوم رابطه‌ی hasMany که با async:true مشخص شده‌است، دقیقا معنای lazy loading را دارد.

```
Blogger.Post = DS.Model.extend({
   title: DS.attr(),
   body: DS.attr(),
   comments: DS.hasMany('comment', { async: true } /* lazy loading */)
});
```

در سمت سرور، دو راه برای فعال سازی این lazy loading تعریف شده در سمت کاربر وجود دارد:
الف) Idهای نظرات هر مطلب را به صورت یک آرایه، در بار اول ارسال لیست نظرات به سمت کاربر، تهیه و ارسال کنیم:

```
namespace EmberJS03.Models
{
    public class Post
    {
        public int Id { set; get; }
        public string Title { set; get; }
        public string Body { set; get; }

        // lazy loading via an array of IDs
        public int[] Comments { set; get; }
    }
}
```

در اینجا خاصیت Comments، تنها کافی است لیستی از Idهای نظرات مرتبط با مطلب جاری باشد. در این حالت در سمت کاربر اگر مطلب خاصی جهت مشاهده‌ی جزئیات آن انتخاب شود، به ازای هر Id ذکر شده، یکبار دستور Get صادر خواهد شد.
ب) این روش به علت تعداد رفت و برگشت بیش از حد به سرور، کارآیی آنچنانی ندارد. بهتر است جهت مشاهده‌ی جزئیات یک مطلب، تنها یکبار درخواست Get کلیه نظرات آن صادر شود.
برای اینکار باید مدل برنامه را به شکل زیر تغییر دهیم:

```
namespace EmberJS03.Models
{
    public class Post
    {
        public int Id { set; get; }
        public string Title { set; get; }
        public string Body { set; get; }

        // load related models via URLs instead of an array of IDs
        // ref. https://github.com/emberjs/data/pull/1371
        public object Links { set; get; }

        public Post()
        {
```

```
            Links = new { comments = "comments" }; // api/posts/id/comments
        }
    }
}
```

در اینجا یک خاصیت جدید به نام Links ارائه شده‌است. نام Links در Ember data استاندارد است و از آن برای دریافت کلیه اطلاعات لینک شده‌ی به یک مطلب استفاده می‌شود. با تعریف این خاصیت به نحوی که ملاحظه می‌کنید، اینبار Ember data تنها یکبار درخواست ویژه‌ای را با فرمت api/posts/id/comments، به سمت سرور ارسال می‌کند. برای مدیریت آن، قالب مسیریابی پیش فرض api/{controller}/{id} را می‌توان به صورت api/{controller}/{id}/{name} اصلاح کرد:

```
namespace EmberJS03
{
    public class Global : System.Web.HttpApplication
    {

        protected void Application_Start(object sender, EventArgs e)
        {
            RouteTable.Routes.MapHttpRoute(
               name: "DefaultApi",
               routeTemplate: "api/{controller}/{id}/{name}",
               defaults: new { id = RouteParameter.Optional, name = RouteParameter.Optional }
               );

            var settings =
GlobalConfiguration.Configuration.Formatters.JsonFormatter.SerializerSettings;
            settings.ContractResolver = new CamelCasePropertyNamesContractResolver();
        }
    }
}
```

اکنون دیگر درخواست جدید api/posts/3/comments با پیام 404 یا یافت نشد مواجه نمی‌شود.
در این حالت در طی یک درخواست می‌توان کلیه نظرات را به سمت کاربر ارسال کرد. در اینجا نیز ذکر ریشه‌ی comments همانند ریشه posts، الزامی است:

```
namespace EmberJS03.Controllers
{
    public class PostsController : ApiController
    {
        //GET api/posts/id
        public object Get(int id)
        {
            return
                new
                {
                    posts = DataSource.PostsList.FirstOrDefault(post => post.Id == id),
                    comments = DataSource.CommentsList.Where(comment => comment.Post == id).ToList()
                };
        }
    }
}
```

پردازش‌های async و متد transitionToRoute در Ember.js

اگر متد حذف مطالب را نیز به کنترلر Posts اضافه کنیم:

```
namespace EmberJS03.Controllers
{
    public class PostsController : ApiController
    {
        public HttpResponseMessage Delete(int id)
        {
            var item = DataSource.PostsList.FirstOrDefault(x => x.Id == id);
            if (item == null)
                return Request.CreateResponse(HttpStatusCode.NotFound);
```

```
            DataSource.PostsList.Remove(item);

            //حذف کامنت‌های مرتبط
            var relatedComments = DataSource.CommentsList.Where(comment => comment.Post ==
id).ToList();
            relatedComments.ForEach(comment => DataSource.CommentsList.Remove(comment));

            return Request.CreateResponse(HttpStatusCode.OK, new { post = item });
        }
    }
}
```

قسمت سمت سرور کار تکمیل شده‌است. اما در سمت کاربر، چنین خطایی را دریافت خواهیم کرد:

```
Attempted to handle event `pushedData` on  while in state root.deleted.inFlight.
```

منظور از حالت inFlight در اینجا این است که هنوز کار حذف سمت سرور تمام نشده‌است که متد transitionToRoute را صادر کرده‌اید. برای اصلاح آن، فایل Scripts\Controllers\post.js را باز کرده و پس از متد destroyRecord، متد then را قرار دهید:

```
Blogger.PostController = Ember.ObjectController.extend({
    isEditing: false,
    actions: {
        edit: function () {
            this.set('isEditing', true);
        },
        save: function () {
            var post = this.get('model');
            post.save();

            this.set('isEditing', false);
        },
        delete: function () {
            if (confirm('Do you want to delete this post?')) {
                var thisController = this;
                var post = this.get('model');
                post.destroyRecord().then(function () {
                    thisController.transitionToRoute('posts');
                });
            }
        }
    }
});
```

به این ترتیب پس از پایان عملیات حذف سمت سرور، قسمت then اجرا خواهد شد . همچنین باید دقت داشت که this اشاره کننده به کنترلر جاری را باید پیش از فراخوانی then ذخیره و استفاده کرد.

کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

EmberJS03_05.zip