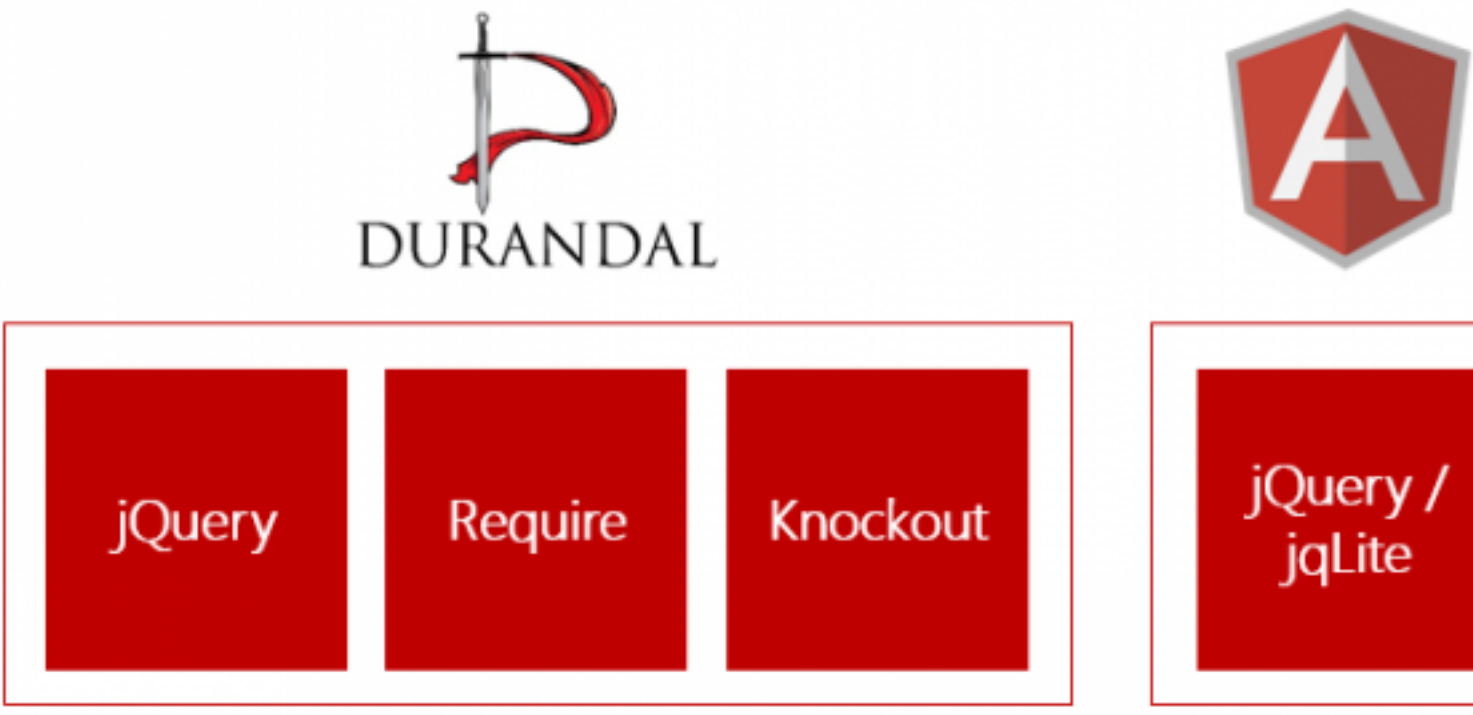
عنوان: **تفاوت AngularJS با KnockoutJS**
نویسنده: شاهین کیاست
تاریخ: ۱۳۹۲/۱۰/۱۰ ۱۹:۳۵
آدرس: www.dotnettips.info
گروه‌ها: KnockoutJS, AngularJS, DurandalJS, SPA

با پیشرفت HTML 5 و پدید آمدن چارچوب‌های مختلف JavaScript توسعه‌ی نرم افزارهای تک صفحه ای تحت وب (Single Page Applications) محبوب شده است. اخیرا مطالب خوبی در رابطه با AngularJS در وبسایت جاری منتشر شده است.
KnockoutJS توسط Microsoft معرفی شد و در قالب پیشفرض پروژه‌های SPA قرار گرفت ، بنابراین احتمالا این سوال برای افرادی مطرح شده است که تفاوت بین KnockoutJS و AngularJS چیست ؟
می توان پاسخ داد این مقایسه ممکن نیست.

KnockoutJS : یک پیاده سازی مستقل JavaScript از الگوی MVVM با امکانات Databinding می‌باشد. Knokcout یک کتابخانه‌ی Databinding است نه یک کتاب خانه‌ی SPA
AngularJS : طبق معرفی در این مطلب AngularJS فریم ورکی متن باز و نوشته شده به زبان جاوا اسکریپت است. هدف از به وجود آمدن این فریم ورک، توسعه هر چه ساده‌تر SPAها با الگوی طراحی MVC و تست پذیری هر چه آسان‌تر آن‌ها است. این فریم ورک توسط یکی از محققان Google در سال 2009 به وجود آمد. بعدها این فریم ورک تحت مجوز MIT به صورت متن باز در آمد و اکنون گوگل آن را حمایت می‌کند و توسط هزاران توسعه دهنده در سرتاسر دنیا، توسعه داده می‌شود.

بنابراین شاید بهتر باشد ذکر شود AngularJS یک Presentation Framework مخصوص برنامه‌های وب تک صفحه ای می‌باشد در حالی که KnockoutJS کتاب خانه ای با تمرکز بر Databinding می‌باشد ، بنابراین مقایسه‌ی این‌ها چندان صحیح نیست.

اگر قصد بر بررسی گزینه‌های دیگر در کنار Angular باشد ، می‌توان از Durandal نام برد. Durandal یک چارچوب SPA می‌باشد ، این چارچوب بر فراز RequireJS ، jQuery و Knockout توسعه پیدا کرده است. (سابقا برای routing از SammyJS استفاده می‌کرد که در نسخه‌های اخیر از موتور خودش استفاده می‌کند.)



Durandal از Knockout جهت Databinding و از RequireJS برای مدیریت وابستگی‌ها استفاده می‌کند.
Angular همه‌ی امکانات بالا را مستقل پیاده سازی کرده و حتی نیازی به jQuery ندارد. اگر jQuery وجود داشته باشد Angular از آن استفاده می‌کند در غیر این صورت از jQuery Lite یا jqLite استفاده می‌کند. jqLite پیاده سازی توابع متداول jQuery برای دستکاری DOM می‌باشد. اطلاعات بیشتر در اینجا

بنابراین با استفاده تنها از KnockoutJS نمی‌توان یک برنامه‌ی کامل SPA توسعه داد ، در کنار آن نیاز به کتابخانه‌های دیگری مثل jQuery برای مدیریت درخواست‌های AJAX و استفاده از دیگر APIها ، Sammy برای routing و RequireJS برای مدیریت وابستگی‌ها می‌باشد.

در Knockout و در نتیجه Durandal عمل Databinding به این صورت است :

```
// JavaScript
var vm = {
    firstName = ko.observable('John')
};
ko.applyBindings(vm);
```

```
<!-- HTML -->
<input data-bind="value:firstName"/>
```

در Angular :

```
// JavaScript
// Inside of a personController
this.firstName = 'John';
```

در Angular همچنین از یک روش Controller As استفاده می‌شود :

```
<!-- HTML -->
<div ng-controller="personController as vm">
    <input ng-model="vm.firstName"/>
</div>
```

اگر تنها نیاز به یک کتابخانه‌ی Databinding باشد ، Knockout گزینه‌ی مناسبی است ، به خوبی از عمل مقید سازی داده‌ها پشتیبانی می‌کند و Syntax خوش دستی دارد اما اگر نیاز به چارچوبی برای توسعه‌ی پروژه‌های SPA می‌باشد می‌توان از Angular یا Durandal استفاده کرد.

مقایسه‌ی Knockout با Angular همانند مقایسه‌ی موتور بنز با ماشین پورشه می‌باشد.

مطالعه‌ی بیشتر

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۰/۱۱ ۱:۱۱

برای مطالعه بیشتر: سری 8 قسمتی AngularJS vs Knockout

نویسنده: mohammad sepahvand
تاریخ: ۱۳۹۲/۱۱/۲۱ ۱۰:۲۹

به نظر من مقایسه angular و knockout آنقدر هم احمقانه نیست. اگر بخواهیم فقط هم از data binding استفاده کنیم angular خیلی از knockout خوش دست‌تر و ساده‌تر است. تازگی angular بیشتر modular شده و بنابراین مقایسه این دو مانند مقایسه موتور بنز با خود پورشه نیست، چون اگر تنها نیازمان data-binding است لزومی ندارد از moduleهای دیگر angular مانند ng-animate, ng-route..... استفاده کنیم و حتی نیازی نیست آن اسکریپتهارا در پروژه خود include کنیم.

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۲/۱۱/۲۱ ۱۰:۴۳

در واقع زمانی که تنها از ماژول Data binding استفاده می‌شود یعنی به عنوان مثال تنها از موتور بنز استفاده شده .

نویسنده: خیام
تاریخ: ۱۳۹۲/۱۱/۲۱ ۱۳:۴۳

حالا که زحمت مقایسه AngularJs و knockout رو انجام دادین ، بهتر بود Angular رو با یک فریم ورک قویتری مثل Ember مقایسه کنید و از این دو سخن بگید ؟ نظر شما در مورد این دو چی هست ؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۱/۲۱ ۱۷:۳

فاکتورهایی را که باید حین انتخاب یک فریم‌ورک JavaScript MVC در نظر داشت

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۲/۱۱/۲۱ ۱۸:۲۱

مقایسه از این قبیل زیاد است
اگر نگاهی به جامعه کاربری استفاده کننده کنیم به طور مثال در Stackoverflow با تگ Angular حدود 25 هزار سوال پرسیده شده در حالی که با تگ Backbone حدود 14 هزار سوال پرسیده شده.
Angular امکانات کاملی برای توسعه‌ی SPA در بر دارد.

نویسنده: سعید رضایی
تاریخ: ۱۳۹۲/۱۲/۲۰ ۱۶:۴۳

با عرض سلام.
angularjs با مرورگر ie 11 به پایین مشکل داره..

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۲/۲۰ ۱۷:۰

خیر؛ با IE 9 به بعد مشکلی ندارد. با IE8 هم کار می‌کند ولی یک سری نکات خاص خودش را دارد. اطلاعات بیشتر را در مستندات رسمی آن در مورد IE مطالعه کنید.

عنوان: **شروع به کار با Ember.js**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۱۴ ۸:۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, jQuery, SPA, EmberJS

Ember.js کتابخانه‌ای است جهت ساده سازی تولید برنامه‌های تک صفحه‌ای وب. برنامه‌هایی که شبیه به برنامه‌های دسکتاپ در مرورگر کاربر عمل می‌کنند. دو برنامه نویس اصلی آن Yehuda Katz که عضو اصلی تیم‌های jQuery و Ruby on Rails است و Tom Dale که ابتدا SproutCore را به وجود آورد و بعدها به Ember.js تغییر نام یافت، هستند.

منابع اصلی Ember.js

پیش از شروع به بحث نیاز است با تعدادی از سایت‌های اصلی مرتبط با Ember.js آشنا شد:
سایت اصلی: http://emberjs.com
مخزن کدهای آن: https://github.com/emberjs
انجمن اختصاصی پرسش و پاسخ: http://discuss.emberjs.com
موتور قالب‌های آن: http://handlebarsjs.com
لیست منابع مطالعاتی مرتبط مانند ویدیوهای آموزشی و لیست مقالات موجود: http://emberwatch.com
و بسته‌ی نیوگت آن: https://www.nuget.org/packages/EmberJS

مفاهیم پایه‌ای Ember.js

شیء Application

```
App = Ember.Application.create();
```

یک برنامه‌ی Ember.js با تعریف وهله‌ای از شیء Application آن آغاز می‌شود. با اینکار به صورت خودکار رویدادگردان‌هایی به صفحه اضافه می‌شوند. کامپونت‌های پیش فرض آن ایجاد شده و همچنین قالب اصلی برنامه رندر می‌شود.

مسیر یابی

با مرور قسمت‌های مختلف برنامه توسط کاربر، نیاز است حالات برنامه را مدیریت کرد؛ اینجا است که کار قسمت مسیریابی شروع می‌شود. مسیریابی، منابع مورد نیاز جهت آدرس‌های مشخصی را تامین می‌کند.

```
App.Router.map(function() {
    this.resource('accounts'); // takes us to /accounts
    this.resource('gallery'); // takes us to /gallery
});
```

در اینجا نحوه‌ی تعریف آغازین مسیریابی Ember.js را مشاهده می‌کنید که توسط متد resource آن مسیرهای قابل ارائه توسط برنامه مشخص می‌شوند.
به این ترتیب مسیرهای accounts/ و gallery/ قابل پردازش خواهند شد.

این مسیرها، تو در تو نیز می‌توانند باشند. برای مثال:

```
App.Router.map(function() {
    this.resource('news', function() {
        this.resource('images', function () { // takes us to /news/images
            this.route('add');// takes us to /news/images/add
        });
    });
});
```

به این ترتیب نحوه‌ی تعریف مسیریابی آدرس news/images/add را مشاهده می‌کنید. همچنین در این مثال از دو متد resource و route استفاده شده‌است. از متد resource برای حالت تعریف اسامی استفاده کنید و از متد route برای تعریف افعال و تغییر دهنده‌ها. برای نمونه در اینجا فعل افزودن تصاویر با متد route مشخص شده‌است.

مدل‌ها

مدل‌ها همان اشیایی هستند که برنامه مورد استفاده قرار می‌دهد و می‌توانند یک آرایه‌ی ساده و یا اشیاء JSON دریافتی از وب سرور باشند.

حداقل به دو روش می‌توان مدل‌ها را تعریف کرد:

الف) با استفاده از افزونه‌ی Ember Data

ب) با کمک شیء Ember.Object

```
App.SiteLink = Ember.Object.extend({});
App.SiteLink.reopenClass({
    findAll: function() {
        var links = [];
        //… $.getJSON …

        return links;
    }
});
```

ابتدا یک زیرکلاس از Ember.Object به کمک متد extend ایجاد خواهد شد. سپس از متد توکار reopenClass برای توسعه‌ی API کمک خواهیم گرفت.

در ادامه متد دلخواهی را ایجاد کرده و برای مثال آرایه‌ای از اشیاء دلخواه جاوا اسکریپتی را بازگشت خواهیم داد.

پس از تعریف مدل، نیاز است آن‌را به سیستم مسیریابی معرفی کرد:

```
App.GalleryRoute = Ember.Route.extend({
    model: function() {
        return App.SiteLink.findAll();
    }
});
```

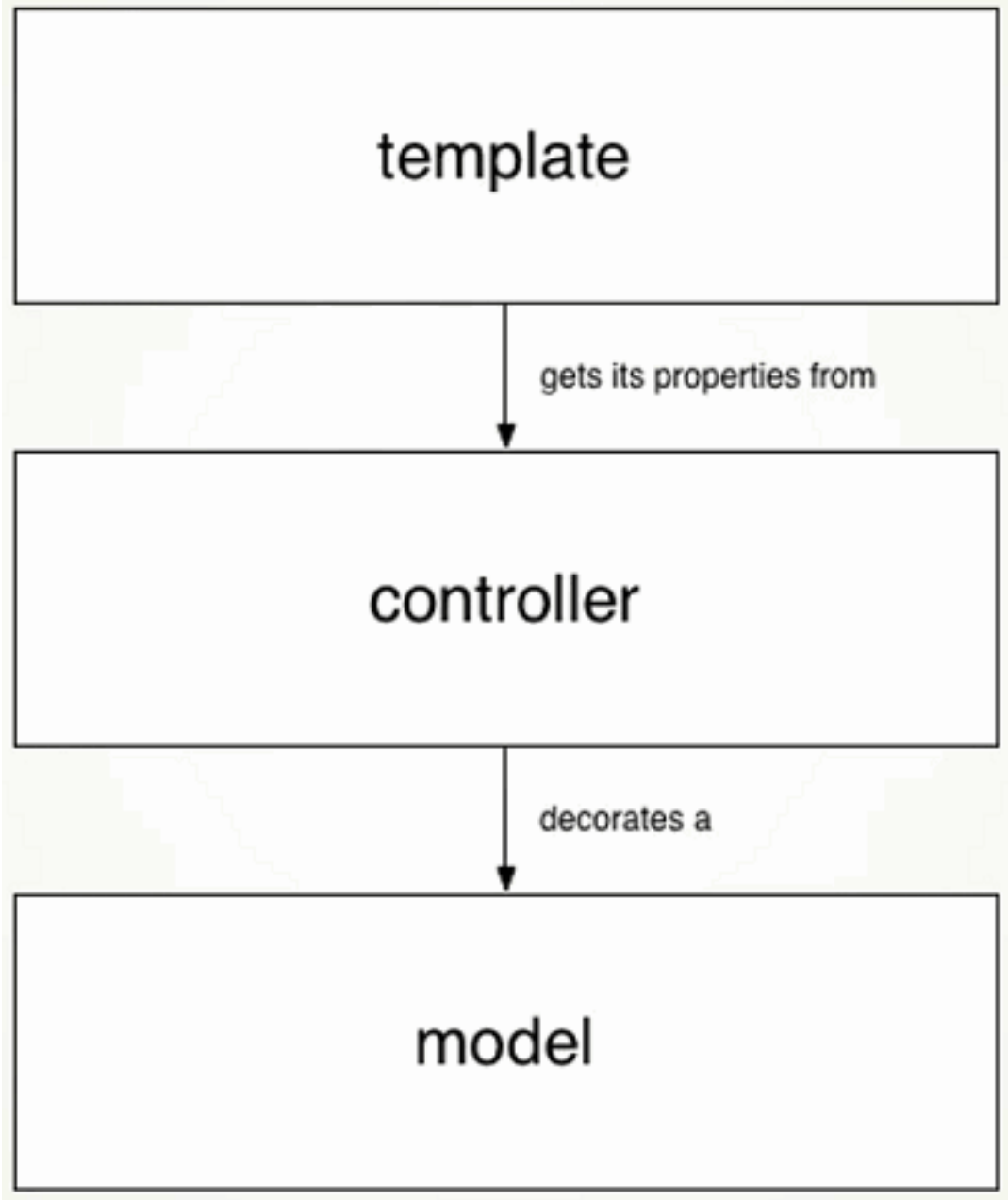
به این ترتیب زمانیکه کاربر به آدرس gallery/ مراجعه می‌کند، دسترسی به model وجود خواهد داشت. در اینجا model یک واژه‌ی کلیدی است.

کنترلرها

کنترلرها جهت ارائه‌ی اطلاعات مدل‌ها به View و قالب برنامه تعریف می‌شوند. در اینجا همیشه باید بخاطر داشت که model تامین کننده‌ی اطلاعات است. کنترلر جهت در معرض دید قرار دادن این اطلاعات، به View برنامه کاربرد دارد و مدل‌ها هیچ اطلاعی از وجود کنترلرها ندارند.

کنترلرها علاوه بر اطلاعات model، می‌توانند حاوی یک سری خواص و اشیاء صرفا نمایشی که قرار نیست در بانک اطلاعاتی ذخیره شوند نیز باشند.

در Ember.js قالب‌ها (templates) اطلاعات خود را از کنترلر دریافت می‌کنند. کنترلرها اطلاعات مدل را به همراه سایر خواص نمایشی مورد نیاز در اختیار View و قالب‌های برنامه قرار می‌دهند.



برای تعریف یک کنترلر می‌توان درون شیء مسیریابی، با تعریف متد setupController شروع کرد:

```
App.GalleryRoute = Ember.Route.extend({
    setupController: function(controller) {
        controller.set('content', ['red', 'yellow', 'blue']);
    }
});
```

در این مثال یک خاصیت دلخواه به نام content تعریف و سپس آرایه‌ای به آن انتساب داده شده‌است.

روش دوم تعریف کنترلرها با ایجاد یک زیر کلاس از شیء Ember.Controller انجام می‌شود:

```
App.GalleryController = Ember.Controller.extend({
    search: '',
    content: ['red', 'yellow', 'blue'],
    query: function() {
        var data = this.get('search');
        this.transitionToRoute('search', { query: data });
```

```
    }
});
```

قالب‌ها یا templates

قالب‌ها قسمت‌های اصلی رابط کاربری را تشکیل خواهند داد. در اینجا از کتابخانه‌ای به نام handlebars برای تهیه قالب‌های سمت کاربر کمک گرفته می‌شود.

```
<script type="text/x-handlebars" data-template-name="sayhello">
    Hello,
    <strong>{{firstName}} {{lastName}}</strong>!
</script>
```

این قالب‌ها توسط تگ اسکریپت تعریف شده و نوع آن‌ها text/x-handlebars مشخص می‌شود. به این ترتیب Ember.js، این قسمت از صفحه را یافته و عبارات داخل {{}} را با مقادیر دریافتی از کنترلر جایگزین می‌کند.

```
<script type="text/x-handlebars" data-template-name="sayhello">
    Hello,
    <strong>{{firstName}} {{lastName}}</strong>!

    {{#if person}}
    Welcome back,
    <strong>{{person.firstName}} {{person.lastName}}</strong>!
    {{/if}}

    <ul>
        {{#each friend in friends}}
        <li>
            {{friend.name}}
        </li>
        {{/each}}
    </ul>

    <img {{bindAttr src="link.url" }} />
    {{#linkTo ''about}}About{{/linkTo}}
</script>
```

در این مثال نحوه‌ی تعریف عبارات شرطی و یا یک حلقه را نیز مشاهده می‌کنید. همچنین امکان اتصال به ویژگی‌هایی مانند src یک تصویر و یا ایجاد لینک‌ها را نیز دارا است.
بهترین مرجع آشنایی با ریز جزئیات کتابخانه‌ی handlebars، مراجعه به سایت اصلی آن است.

قواعد پیش فرض نامگذاری در Ember.js

اگر به مثال‌های فوق دقت کرده باشید، خواصی مانند GalleryController و یا GalleryRoute به شیء App اضافه شده‌اند. این نوع نامگذاری‌ها در ember.js بر اساس روش convention over configuration کار می‌کنند. برای نمونه اگر مسیریابی خاصی را به نحو ذیل تعریف کردید:

```
this.resource('employees');
```

شیء مسیریابی آن App.EmployeesRoute
کنترلر آن App.EmployeesController
مدل آن App.Employee
View آن App.EmployeesView
و قالب آن employees
بهتر است تعریف شوند. به عبارتی اگر اینگونه تعریف شوند، به صورت خودکار توسط Ember.js یافت شده و هر کدام با مسئولیت‌های خاص مرتبط با آن‌ها پردازش می‌شوند و همچنین ارتباطات بین آن‌ها به صورت خودکار برقرار خواهد شد. به این ترتیب برنامه نظم بهتری خواهد یافت. با یک نگاه می‌توان قسمت‌های مختلف را تشخیص داد و همچنین کدنویسی پردازش و

اتصال قسمت‌های مختلف برنامه نیز به شدت کاهش می‌یابد.

تهیه‌ی اولین برنامه‌ی Ember.js

تا اینجا نگاهی مقدماتی داشتیم به اجزای تشکیل دهنده‌ی هسته‌ی Ember.js. در ادامه مثال ساده‌ای را جهت نمایش ساختار ابتدایی یک برنامه‌ی Ember.js، بررسی خواهیم کرد.

بسته‌ی Ember.js را همانطور که در قسمت منابع اصلی آن در ابتدای بحث عنوان شد، می‌توانید از سایت و یا مخزن کد آن دریافت کنید و یا اگر از VS.NET استفاده می‌کنید، تنها کافی است دستور ذیل را صادر نمائید:

```
PM> Install-Package EmberJS
```

پس از اضافه شدن فایل‌های js آن به پوشه‌ی Scripts برنامه، در همان پوشه، فایل جدید Scripts\app.js را نیز اضافه کنید. از آن برای افزودن تعاریف کدهای Ember.js استفاده خواهیم کرد.

در این حالت ترتیب تعریف اسکریپت‌های مورد نیاز صفحه به صورت ذیل خواهند بود:

```
<script src="Scripts/jquery-2.1.1.js" type="text/javascript"></script>
<script src="Scripts/handlebars.js" type="text/javascript"></script>
<script src="Scripts/ember.js" type="text/javascript"></script>
<script src="Scripts/app.js" type="text/javascript"></script>
```

کدهای ابتدایی فایل app.js جهت وهله سازی شیء Application و سپس تعریف مسیریابی صفحه‌ی index بر اساس روش convention over configuration به همراه تعریف یک کنترلر و افزودن متغیری به نام content به آن که با یک آرایه مقدار دهی شده‌است:

```
App = Ember.Application.create();
App.IndexRoute = Ember.Route.extend({
    setupController:function(controller) {
        controller.set('content', ['red', 'yellow', 'blue']);
    }
});
```

باید دقت داشت که تعریف مقدماتی Ember.Application.create به همراه یک سری تنظیمات پیش فرض نیز هست. برای مثال مسیریابی index به صورت خودکار به نحو ذیل توسط آن تعریف خواهد شد و نیازی به تعریف مجدد آن نیست:

```
App.Router.map(function() {
    this.resource('application');
    this.resource('index');
});
```

سپس برای اتصال این کنترلر به یک template خواهیم داشت:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <script src="Scripts/jquery-2.1.1.js" type="text/javascript"></script>
    <script src="Scripts/handlebars.js" type="text/javascript"></script>
    <script src="Scripts/ember.js" type="text/javascript"></script>
    <script src="Scripts/app.js" type="text/javascript"></script>
</head>
<body>
    <script type="text/x-handlebars" data-template-name="index">
        Hello,
        <strong>Welcome to Ember.js</strong>!
        <ul>
            {{#each item in content}}
            <li>
                {{item}}
            </li>
            {{/each}}
        </ul>
    </script>
```

```
</body>
</html>
```

توسط اسکریپتی از نوع text/x-handlebars، اطلاعات آرایه content دریافت و در طی یک حلقه در صفحه نمایش داده خواهد شد.
مقدار data-template-name در اینجا مهم است. اگر آن‌را به هر نام دیگری بجز index تنظیم کنید، منبع دریافت اطلاعات آن مشخص نخواهد بود. نام index در اینجا به معنای اتصال این قالب به اطلاعات ارائه شده توسط کنترلر index است.

تا همینجا اگر برنامه را اجرا کنید، به خوبی کار خواهد کرد. نکته‌ی دیگری که در مورد قالب‌های Ember.js قابل توجه هستند، قالب پیش فرض application است. با تعریف Ember.Application.create یک چنین قالبی نیز به ابتدای هر صفحه به صورت خودکار اضافه خواهد شد:

```
<body>
    <script type="text/x-handlebars" data-template-name="application">
        <h1>Header</h1>
        {{outlet}}
    </script>
```

outlet واژه‌ای است کلیدی که سبب رندر سایر قالب‌های تعریف شده در صفحه می‌گردد. مقدار data-template-name آن نیز به application تنظیم شده‌است (اگر این مقدار ذکر نگردد نیز به صورت خودکار از application استفاده می‌شود). برای مثال اگر بخواهید به تمام قالب‌های رندر شده در صفحات مختلف، مقدار ثابتی را اضافه کنید (مانند هدر یا منو)، می‌توان قالب application را به صورت دستی به نحو فوق اضافه کرد و آن‌را سفارشی سازی نمود.

کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

EmberJS01.zip

نظرات خوانندگان

نویسنده: نصیری
تاریخ: ۱۳۹۳/۰۹/۱۸ ۱۷:۴۹

با سلام؛ با توجه به قدرت خیلی زیاد AngularJs و نیز اینکه پشتیبانی اون داره از طرف تیم قدرتمندی در google اداره میشه و دلیل بعدی اینکه تعداد خیلی زیادی از برنامه نویسهای دنیا به سمت این فریمورک رفتن بهتر نیست یادگرفتن و دنبال AngularJs رفتن را به EmberJs ترجیح داد
با تشکر

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۱۸ ۱۸:۱۸

- صرف نوشتن مطلبی در مورد موضوعی خاص، به معنای «بهترین بودن آن» و «برترین حالت ممکن» نیست. امروز راجع به ember.js مطلب نوشتم. شاید هفته‌های بعد در مورد meteor.js مطلب نوشتم.
- مقایسه‌ای در اینجا « AngularJS vs Ember »
- مقایسه‌ای کامل‌تر در اینجا « AngularJs vs EmberJs »
- « Rails JS frameworks: Ember.js vs. AngularJS »
- « AngularJS vs. Backbone.js vs. Ember.js »
- یک نمونه‌ی دیگر « The Top 10 Javascript MVC Frameworks Reviewed »
- این نظرسنجی را هم دنبال کنید: « آیا به یادگیری یا ادامه‌ی استفاده از AngularJS خواهید پرداخت؟ »

عنوان: **افزودن یک صفحه‌ی جدید و دریافت و نمایش اطلاعات از سرور به کمک Ember.js**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۱۵ ۸:۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, ASP.NET Web API, jQuery, SPA, EmberJS

در قسمت قبل با مقدمات برپایی یک برنامه‌ی تک صفحه‌ای وب مبتنی بر Ember.js آشنا شدیم. مثال انتهای بحث آن نیز یک لیست ساده را نمایش می‌دهد. در ادامه همین برنامه را جهت نمایش لیستی از اشیاء JSON دریافتی از سرور تغییر خواهیم داد. همچنین یک صفحه‌ی about را نیز به آن اضافه خواهیم کرد.

پیشنیازهای سمت سرور

- ابتدا یک پروژه‌ی خالی ASP.NET را ایجاد کنید. نوع آن مهم نیست که Web Forms باشد یا MVC.
- سپس قصد داریم مدل کاربران سیستم را توسط یک ASP.NET Web API Controller در اختیار Ember.js قرار دهیم. مباحث پایه‌ای Web API نیز در وب فرم‌ها و MVC یکی است.

مدل سمت سرور برنامه چنین شکلی را دارد:

```
namespace EmberJS02.Models
{
    public class User
    {
        public int Id { set; get; }
        public string UserName { set; get; }
        public string Email { set; get; }
    }
}
```

کنترلر Web API ایی که این اطلاعات را در ختیار کلاینت‌ها قرار می‌دهد، به نحو ذیل تعریف می‌شود:

```
using System.Collections.Generic;
using System.Web.Http;
using EmberJS02.Models;

namespace EmberJS02.Controllers
{
    public class UsersController : ApiController
    {
        // GET api/<controller>
        public IEnumerable<User> Get()
        {
            return UsersDataSource.UsersList;
        }
    }
}
```

در اینجا UsersDataSource.UsersList صرفا یک لیست جنریک ساده از کلاس User است و کدهای کامل آن‌را می‌توانید از فایل پیوست انتهای بحث دریافت کنید.

همچنین فرض بر این است که مسیریابی سمت سرور ذیل را نیز به فایل global.asax.cs، جهت فعال سازی دسترسی به متدهای کنترلر UsersController تعریف کرده‌اید:

```
using System;
using System.Web.Http;
using System.Web.Routing;

namespace EmberJS02
{
    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 }
                );
        }
    }
}
```

پیشنیازهای سمت کاربر

پیشنیازهای سمت کاربر این قسمت با قسمت « تهیه‌ی اولین برنامه‌ی Ember.js » دقیقا یکی است.
ابتدا فایل‌های مورد نیاز Ember.js به برنامه اضافه شده‌اند:

```
PM> Install-Package EmberJS
```

سپس یک فایل app.js با محتوای ذیل به پوشه‌ی Scripts اضافه شده‌است:

```
App = Ember.Application.create();
App.IndexRoute = Ember.Route.extend({
    setupController:function(controller) {
        controller.set('content', ['red', 'yellow', 'blue']);
    }
});
```

و در آخر یک فایل index.html با محتوای ذیل کار برپایی اولیه‌ی یک برنامه‌ی مبتنی بر Ember.js را انجام می‌دهد:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <script src="Scripts/jquery-2.1.1.js" type="text/javascript"></script>
    <script src="Scripts/handlebars.js" type="text/javascript"></script>
    <script src="Scripts/ember.js" type="text/javascript"></script>
    <script src="Scripts/app.js" type="text/javascript"></script>
</head>
<body>
    <script type="text/x-handlebars" data-template-name="application">
        <h1>Header</h1>
        {{outlet}}
    </script>
    <script type="text/x-handlebars" data-template-name="index">
        Hello,
        <strong>Welcome to Ember.js</strong>!
        <ul>
            {{#each item in content}}
            <li>
                {{item}}
            </li>
            {{/each}}
        </ul>
    </script>
</body>
</html>
```

تا اینجا را در قسمت قبل مطالعه کرده بودید.
در ادامه قصد داریم به هدر صفحه، دو لینک Home و About را اضافه کنیم؛ به نحوی که لینک Home به مسیریابی index و لینک About به مسیریابی about که صفحه‌ی جدید «درباره‌ی برنامه» را نمایش می‌دهد، اشاره کنند.

تعریف صفحه‌ی جدید About

برنامه‌های Ember.js، برنامه‌های تک صفحه‌ای وب هستند و صفحات جدید در آن‌ها به صورت یک template جدید تعریف می‌شوند که نهایتا متناظر با یک مسیریابی مشخص خواهند بود.
به همین جهت ابتدا در فایل app.js مسیریابی about را اضافه خواهیم کرد:

```
App.Router.map(function() {
    this.resource('about');
});
```

به این ترتیب با فراخوانی آدرس /about در مرورگر توسط کاربر، منابع مرتبط با این آدرس و قالب مخصوص آن، توسط Ember.js پردازش خواهند شد.
بنابراین به صفحه‌ی index.html برنامه مراجعه کرده و صفحه‌ی about را توسط یک قالب جدید تعریف می‌کنیم:

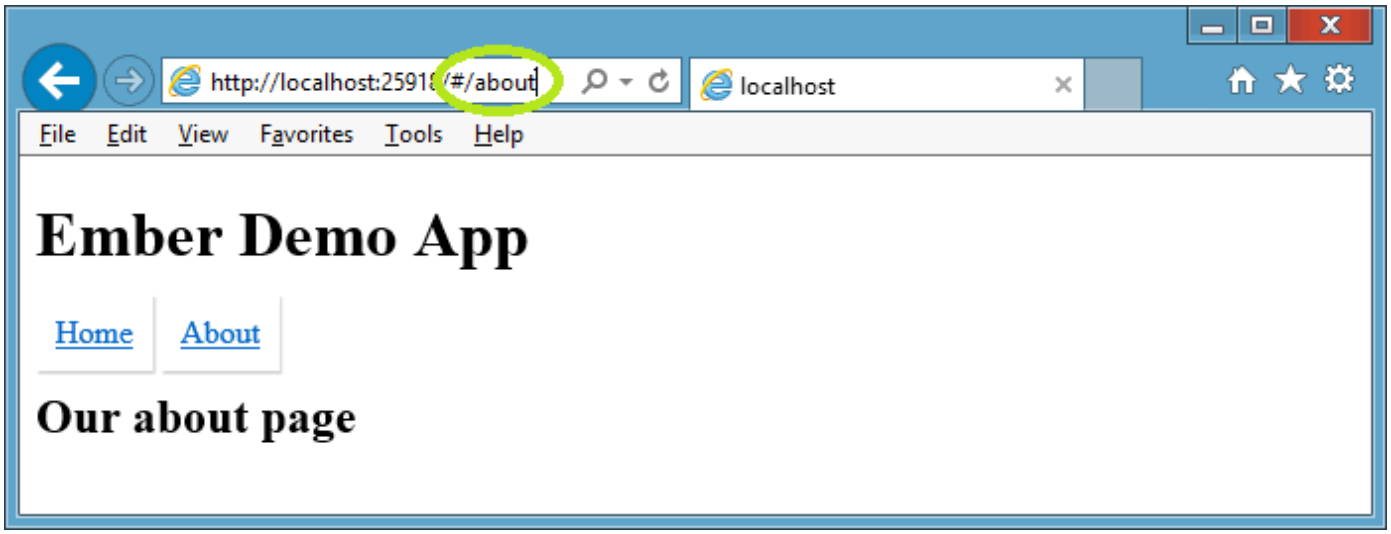
```
<script type="text/x-handlebars" data-template-name="about">
    <h2>Our about page</h2>
</script>
```

تنها نکته‌ی مهم در اینجا مقدار data-template-name است که سبب خواهد شد تا به مسیریابی about، به صورت خودکار متصل و مرتبط شود.

در این حالت اگر برنامه را در حالت معمولی اجرا کنید، خروجی خاصی را مشاهده نخواهید کرد. بنابراین نیاز است تا لینکی را جهت اشاره به این مسیر جدید به صفحه اضافه کنیم:

```
<script type="text/x-handlebars" data-template-name="application">
    <h1>Ember Demo App</h1>
    <ul class="nav">
        <li>{{#linkTo 'index'}}Home{{/linkTo}}</li>
        <li>{{#linkTo 'about'}}About{{/linkTo}}</li>
    </ul>
    {{outlet}}
</script>
```

اگر از قسمت قبل به خاطر داشته باشید، عنوان شد که قالب ویژه‌ی application به صورت خودکار با وهله سازی Ember.Application.create به صفحه اضافه می‌شود. اگر نیاز به سفارشی سازی آن وجود داشت، خصوصا جهت تعریف عناصری که باید در تمام صفحات حضور داشته باشند (مانند منوها)، می‌توان آن‌را به نحو فوق سفارشی سازی کرد.
در اینجا با استفاده از امکان یا directive ویژه‌ای به نام linkTo، لینک‌هایی به مسیریابی‌های index و about اضافه شده‌اند. به این ترتیب اگر کاربری برای مثال بر روی لینک About کلیک کند، کتابخانه‌ی Ember.js او را به صورت خودکار به مسیریابی about و سپس نمایش قالب مرتبط با آن (قالب about ایی که پیشتر تعریف کردیم) هدایت خواهد کرد؛ مانند تصویر ذیل:



همانطور که در آدرس صفحه نیز مشخص است، هرچند صفحه‌ی about نمایش داده شده‌است، اما هنوز نیز در همان صفحه‌ی اصلی برنامه قرار داریم. به علاوه در این قسمت جدید، همچنان منوی بالای صفحه نمایان است؛ از این جهت که تعاریف آن به قالب application اضافه شده‌اند.

دریافت و نمایش اطلاعات از سرور

اکنون که با نحوه‌ی تعریف یک صفحه‌ی جدید و برپایی سیم کشی‌های مرتبط با آن آشنا شدیم، می‌خواهیم صفحه‌ی دیگری را به نام Users به برنامه اضافه کنیم و در آن لیست کاربران ارائه شده توسط کنترلر Web API سمت سرور ابتدای بحث را نمایش دهیم.
بنابراین ابتدا مسیریابی جدید users را به صفحه اضافه می‌کنیم تا لیست کاربران، در آدرس users/ قابل دسترسی شود:

```
App.Router.map(function() {
    this.resource('about');
    this.resource('users');
});
```

سپس نیاز است مدلی را توسط فراخوانی Ember.Object.extend ایجاد کرده و به کمک متد reopenClass آن‌را توسعه دهیم:

```
App.UsersLink = Ember.Object.extend({});
App.UsersLink.reopenClass({
    findAll: function () {
        var users = [];
        $.getJSON('/api/users').then(function(response) {
            response.forEach(function(item) {
                users.pushObject(App.UsersLink.create(item));
            });
        });
        return users;
    }
});
```

در اینجا متد دلخواهی را به نام findAll اضافه کرده‌ایم که توسط متد getJSON جی‌کوئری، به مسیر /api/users سمت سرور متصل شده و لیست کاربران را از سرور به صورت JSON دریافت می‌کند. در اینجا خروجی دریافتی از سرور به کمک متد pushObject به آرایه کاربران اضافه خواهد شد. همچنین نحوه‌ی فراخوانی متد create مدل UsersLink را نیز در اینجا مشاهده می‌کنید (App.UsersLink.create).

پس از اینکه نحوه‌ی دریافت اطلاعات از سرور مشخص شد، باید اطلاعات این مدل را در اختیار مسیریابی Users قرار داد:

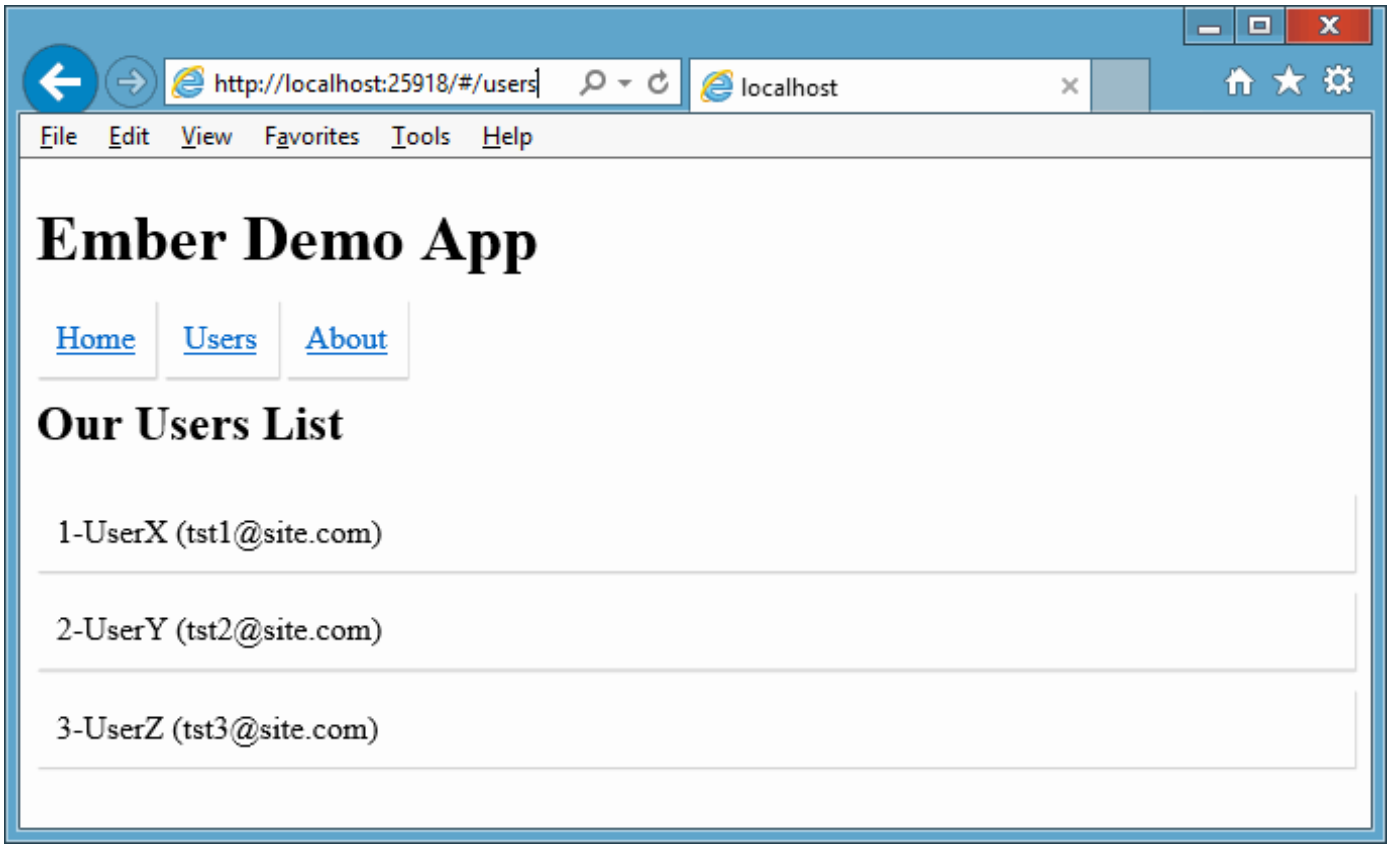
```
App.UsersRoute = Ember.Route.extend({
    model: function() {
        return App.UsersLink.findAll();
    }
});

App.UsersController = Ember.ObjectController.extend({
    customHeader : 'Our Users List'
});
```

به این ترتیب زمانیکه کاربر به مسیر users/ مراجعه می‌کند، سیستم مسیریابی می‌داند که اطلاعات مدل خود را باید از کجا تهیه نماید.
همچنین در کنترلری که تعریف شده، صرفا یک خاصیت سفارشی و دلخواه جدید، به نام customHeader برای نمایش در ابتدای صفحه تعریف و مقدار دهی گردیده‌است.
اکنون قالبی که قرار است اطلاعات مدل را نمایش دهد، چنین شکلی را خواهد داشت:

```
<script type="text/x-handlebars" data-template-name="users">
    <h2>{{customHeader}}</h2>
    <ul>
        {{#each item in model}}
        <li>
            {{item.Id}}-{{item.UserName}} ({{item.Email}})
```

```
        </li>
        {{/each}}
    </ul>
</script>
```

با تنظیم data-template-name به users سبب خواهیم شد تا این قالب اطلاعات خودش را از مسیریابی users دریافت کند. سپس یک حلقه نوشته‌ایم تا کلیه عناصر موجود در مدل را خوانده و در صفحه نمایش دهد. همچنین در عنوان قالب نیز از خاصیت سفارشی customHeader استفاده شده‌است:



کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

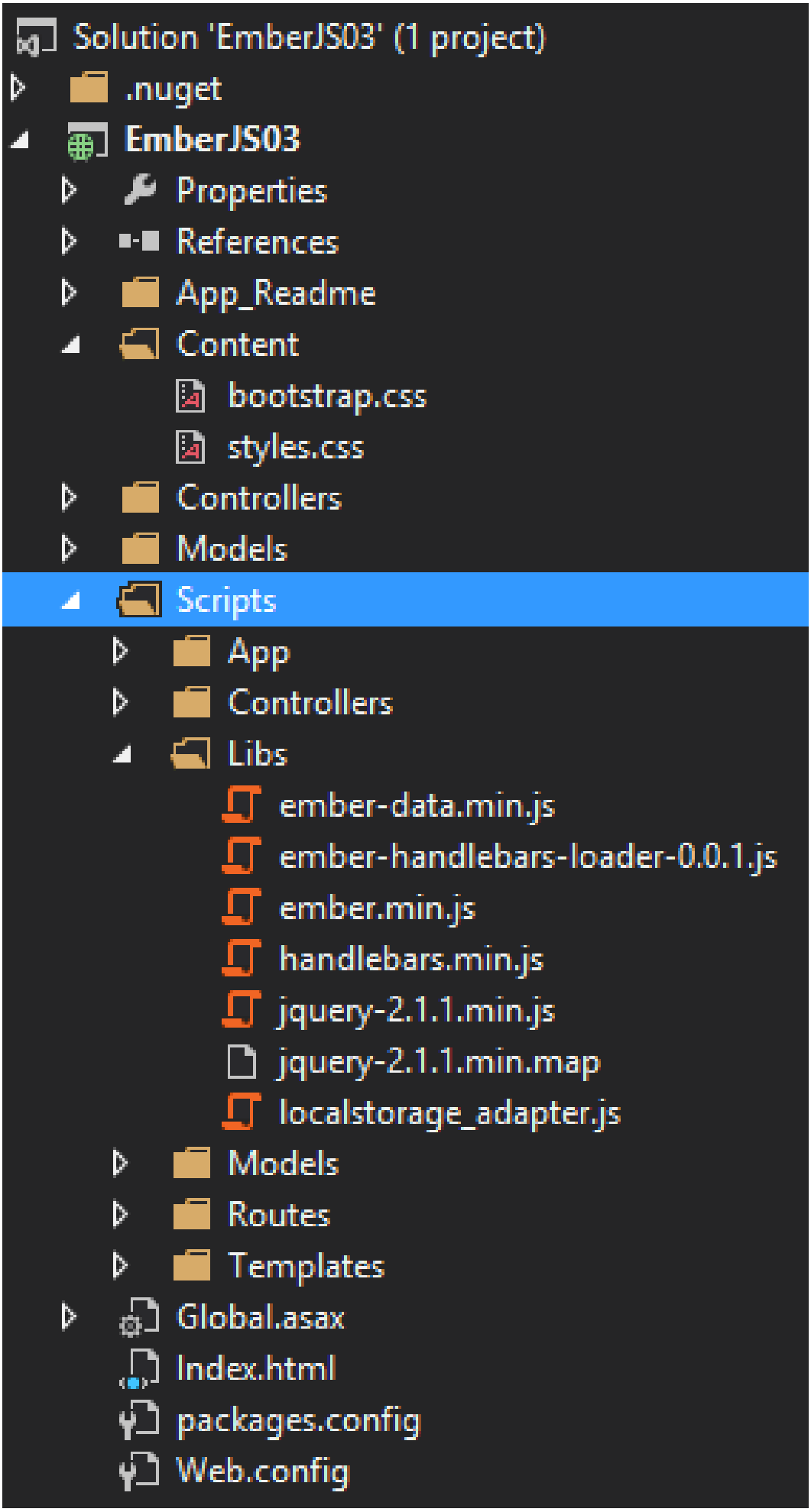
EmberJS02.zip

عنوان: **ساخت یک بلاگ ساده با Ember.js، قسمت اول**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۱۶ ۸:۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, jQuery, SPA, EmberJS

پس از آشنایی مقدماتی با اجزای مهم تشکیل دهنده‌ی Ember.js (^ و ^)، بهتر است این دانسته‌ها را جهت تکمیل یک پروژه‌ی ساده‌ی تک صفحه‌ای بلاگ، بکار بگیریم.
در این بلاگ می‌توان:
- یک مطلب جدید را ارسال کرد.
- مطالب قابل ویرایش و یا حذف هستند.
- مطالب بلاگ قسمت ارسال نظرات دارند.
- امکان گزارشگیری از آخرین نظرات ارسالی وجود دارد.
- سایت صفحات درباره و تماس با ما را نیز دارا است.

ساختار پوشه‌های برنامه

در تصویر ذیل، ساختار پوشه‌های برنامه بلاگ را ملاحظه می‌کنید. چون قسمت سمت کلاینت این برنامه کاملا جاوا اسکریپتی است، پوشه‌های Routes، Models، Libs، Controllers، App و Templates آن در پوشه‌ی Scripts تعریف شده‌اند و به این ترتیب می‌توان تفکیک بهتری را بین اجزای تشکیل دهنده‌ی یک برنامه‌ی تک صفحه‌ای وب Emeber.js پدید آورد.


Solution 'EmberJS03' (1 project)
.nuget
EmberJS03
Properties
References
App_Readme
Content
bootstrap.css
styles.css
Controllers
Models
Scripts
App
Controllers
Libs
ember-data.min.js
ember-handlebars-loader-0.0.1.js
ember.min.js
handlebars.min.js
jquery-2.1.1.min.js
jquery-2.1.1.min.map
localstorage_adapter.js
Models
Routes
Templates
Global.asax
Index.html
packages.config
Web.config

فایل CSS بوت استرپ نیز به پوشه‌ی Content اضافه شده‌است.

دریافت پیشنیازهای سمت کاربر برنامه

در ساختار پوشه‌های فوق، از پوشه‌ی Libs برای قرار دادن کتابخانه‌های پایه برنامه مانند jQuery و Ember.js استفاده خواهیم کرد. به این ترتیب:
- نیاز به آخرین نگارش‌های Ember.js و همچنین افزونه‌ی Ember-Data آن برای کار ساده‌تر با داده‌ها و سرور وجود دارد. این فایل‌ها را از آدرس ذیل می‌توانید دریافت کنید (نسخه‌های نیوگت به دلیل قدیمی بودن و به روز نشدن مداوم آن‌ها توصیه نمی‌شوند):

http://emberjs.com/builds/#/beta

برای حالت آزمایش برنامه، استفاده از فایل‌های دیباگ آن توصیه می‌شوند (فایل‌هایی با نام اصلی و بدون پسوند prod یا min). زیرا این فایل‌ها خطاها و اطلاعات بسیار مفصلی را از اشکالات رخ داده، در کنسول وب مرورگرها، فایرباگ و یا Developer tools آن‌ها نمایش می‌دهند. نسخه‌ی min برای حالت ارائه‌ی نهایی برنامه است. نسخه‌ی prod همان نسخه‌ی دیباگ است با حذف اطلاعات دیباگ (نسخه‌ی production فشرده نشده). نسخه‌ی فشرده شده‌ی prod آن، فایل min نهایی را تشکیل می‌دهد.
- دریافت جی‌کوئری
- آخرین نگارش handlebars.js را از سایت رسمی آن دریافت کنید: http://handlebarsjs.com
- Ember Handlebars Loader: https://github.com/michaelrkn/ember-handlebars-loader
- Ember Data Local Storage Adapter: https://github.com/kurko/ember-localstorage-adapter

ترتیب تعریف و قرارگیری این فایل‌ها را پس از دریافت، در فایل index.html قرار گرفته در ریشه‌ی سایت، در کدهای ذیل مشاهده می‌کنید:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Ember Blog</title>

  <link href="Content/bootstrap.css" rel="stylesheet" />
  <link href="Content/bootstrap-theme.css" rel="stylesheet" />
  <link href="Content/styles.css" rel="stylesheet" />

  <script src="Scripts/Libs/jquery-2.1.1.min.js" type="text/javascript"></script>
  <script src="Scripts/Libs/bootstrap.min.js" type="text/javascript"></script>
  <script src="Scripts/Libs/handlebars-v2.0.0.js" type="text/javascript"></script>
  <script src="Scripts/Libs/ember.js" type="text/javascript"></script>
  <script src="Scripts/Libs/ember-handlebars-loader-0.0.1.js" type="text/javascript"></script>
  <script src="Scripts/Libs/ember-data.js" type="text/javascript"></script>
  <script src="Scripts/Libs/localstorage_adapter.js" type="text/javascript"></script>
</head>
<body>

</body>
</html>
```

اصلاح فایل ember-handlebars-loader-0.0.1.js

اگر به فایل ember-handlebars-loader-0.0.1.js مراجعه کنید، مسیر فایل‌های قالب handlebars قسمت‌های مختلف برنامه را از پوشه‌ی templates واقع در ریشه‌ی سایت می‌خواند. با توجه به تصویر ساختار پوشه‌ی پروژه‌ی جاری، پوشه‌ی template به داخل پوشه‌ی Scripts منتقل شده‌است و نیاز به یک چنین اصلاحی دارد:

```
url: "Scripts/Templates/" + name + ".hbs",
```

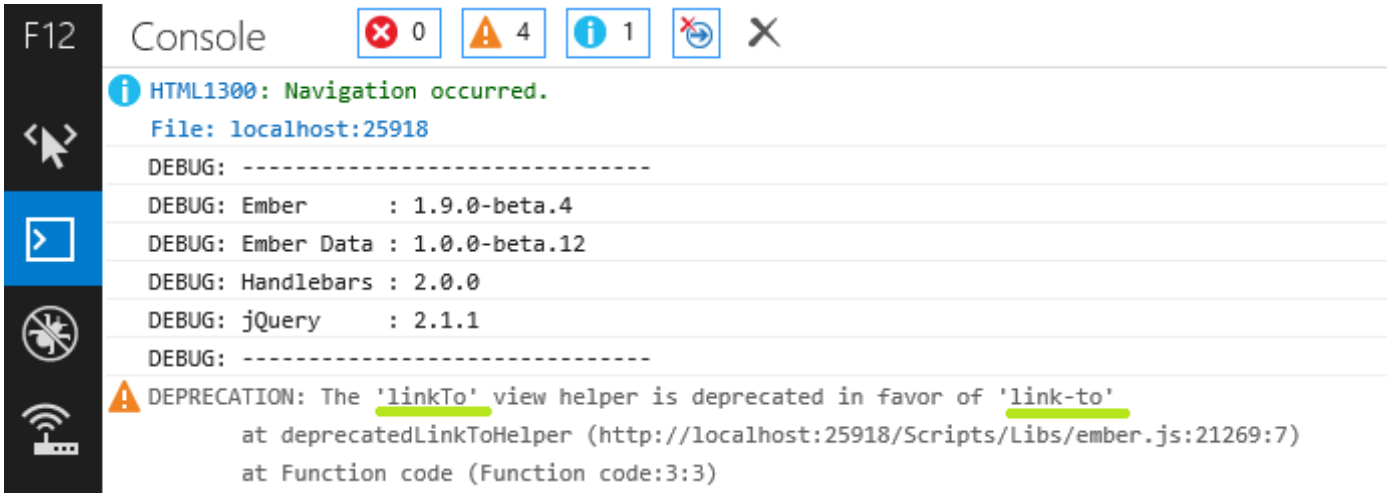
کار اسکریپت ember-handlebars-loader-0.0.1.js بارگذاری خودکار فایل‌های hbs یا handlebars از پوشه‌ی قالب‌های سفارشی

برنامه است. در این حالت اگر برنامه را اجرا کنید، خطای 404 را دریافت خواهید کرد. از این جهت که mime-type پسوند hbs در IIS تعریف نشده‌است. اضافه کردن آن نیز ساده‌است. به فایل web.config برنامه مراجعه کرده و تغییر ذیل را اعمال کنید:

```
<system.webServer>
  <staticContent>
        <mimeMap fileExtension=".hbs" mimeType="text/x-handlebars-template" />
  </staticContent>
 </system.webServer>
```

مزیت استفاده از نسخه‌ی دیباگ ember.js در حین توسعه‌ی برنامه

نسخه‌ی دیباگ ember.js علاوه بر به همراه داشتن خطاهای بسیار جامع و توضیح علل مشکلات، مواردی را که در آینده منسوخ خواهند شد نیز توضیح می‌دهد:



برای مثال در اینجا عنوان شده‌است که دیگر از linkTo استفاده نکنید و آن‌را به link-to تغییر دهید.
همچنین اگر از مرورگر کروم استفاده می‌کنید، افزونه‌ی Ember Inspector را نیز می‌توانید نصب کنید تا اطلاعات بیشتری را از جزئیات مسیریابی‌های تعریف شده و قالب‌های Ember.js بتوان مشاهده کرد. این افزونه به صورت یک برگه‌ی جدید در Developer tools آن ظاهر می‌شود.

ایجاد شیء Application

همانطور که در قسمت‌های پیشین نیز عنوان شد (^ و ^)، یک برنامه‌ی Ember.js با تعریف وهله‌ای از شیء Application آن آغاز می‌شود. برای این منظور به پوشه‌ی App مراجعه کرده و فایل جدید Scripts\App\blogger.js را اضافه کنید؛ با این محتوا:

```
Blogger = Ember.Application.create();
```

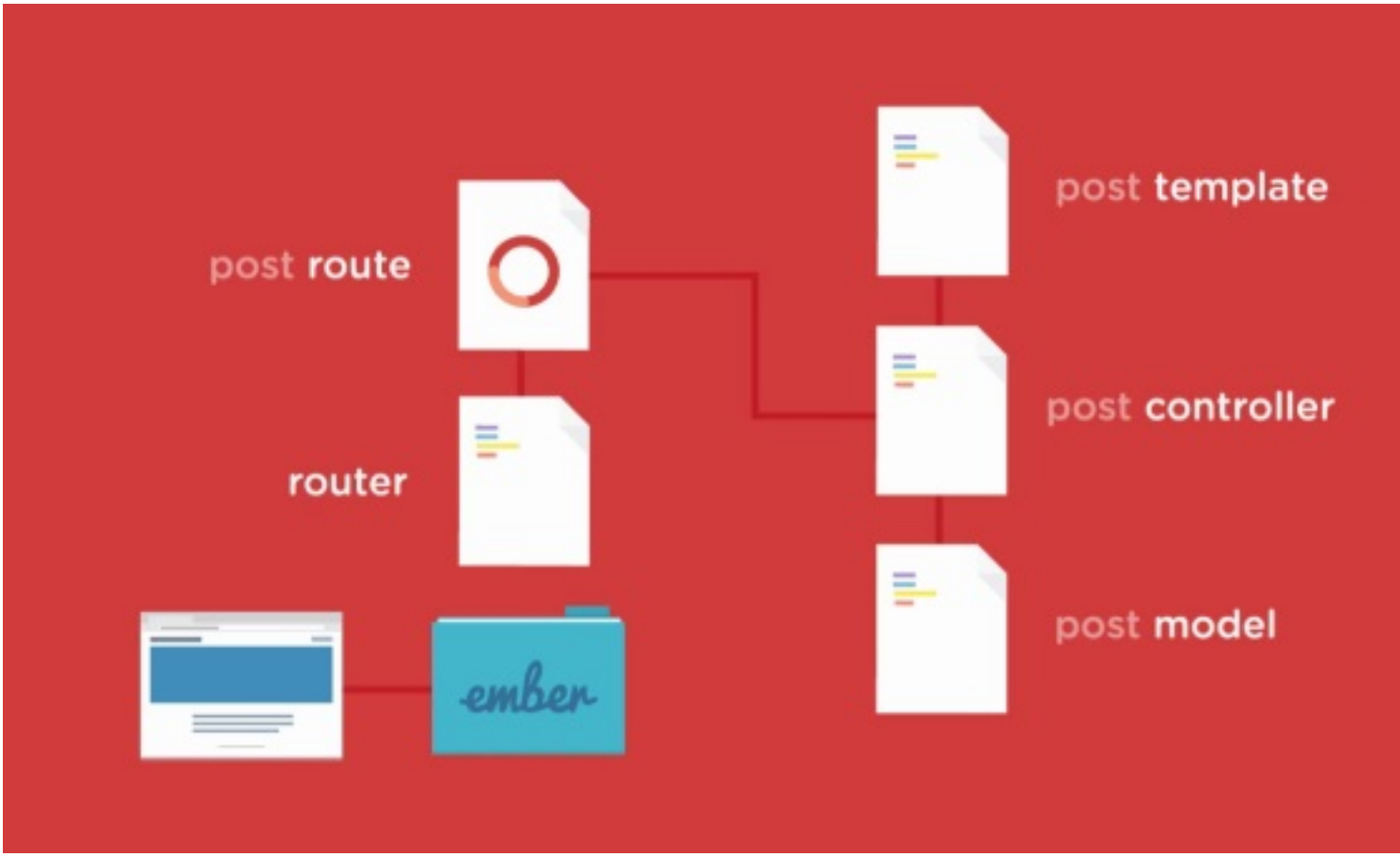
مدخل این فایل را نیز پس از تعاریف وابستگی‌های اصلی برنامه، به فایل index.html اضافه خواهیم کرد:

```
<script src="Scripts/App/blogger.js" type="text/javascript"></script>
```

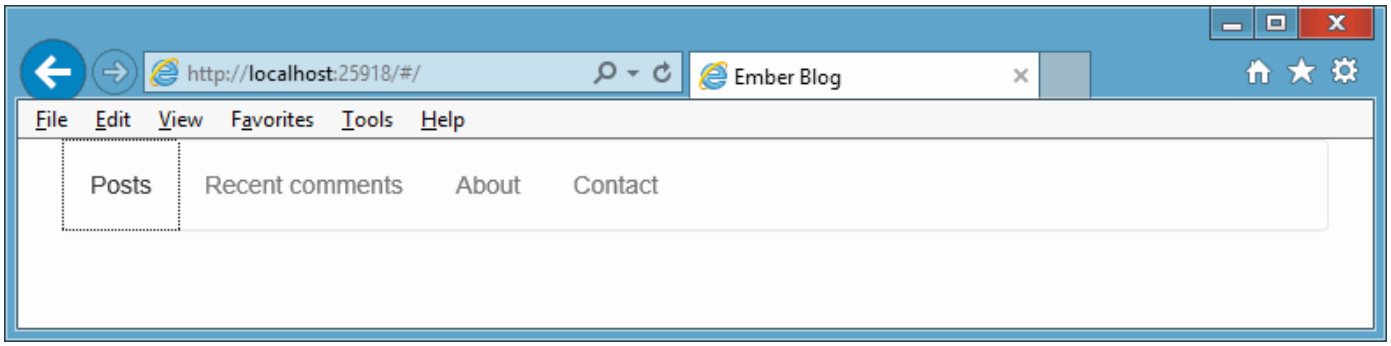
تا اینجا برپایی اولیه‌ی برنامه‌ی تک صفحه‌ای وب مبتنی بر Ember.js ما به پایان می‌رسد.

تعاریف مسیریابی و قالب‌ها

اکنون در ادامه قصد داریم لیستی از عناوین مطالب ارسالی را نمایش دهیم. در ابتدا این عناوین را از یک آرایه‌ی ثابت جاوا اسکریپتی دریافت خواهیم کرد و پس از آن از یک Web API کنترلر، جهت دریافت اطلاعات از سرور کمک خواهیم گرفت.



کار router در Ember.js، نگاشت آدرس درخواستی توسط کاربر، به یک route یا مسیریابی تعریف شده‌است. به این ترتیب مدل، کنترلر و قالب آن route به صورت خودکار بارگذاری و پردازش خواهند.
با مراجعه به ریشه‌ی سایت، فایل index.html بارگذاری می‌شود.



در اینجا تصویری از صفحه‌ی آغازین بلاگ را مشاهده می‌کنید. در آن صفحه‌ی posts همان ریشه‌ی سایت نیز می‌باشد. بنابراین نیاز است ابتدا مسیریابی آن‌را تعریف کرد. برای این منظور، فایل جدید Scripts\App\router.js را به پوشه‌ی App اضافه کنید؛ با این محتوا:

```
Blogger.Router.map(function () {
   this.resource('posts', { path: '/' });
});
```

علت تعریف قسمت path این است که ریشه‌ی سایت (/) نیز به مسیریابی posts ختم شود. در غیر اینصورت کاربر با مراجعه به ریشه‌ی سایت، یک صفحه‌ی خالی را مشاهده خواهد کرد؛ زیرا به صورت پیش فرض، آدرس قابل ترجمه‌ی یک صفحه، با آدرس و نام مسیریابی آن یکی است.

همچنین مدخل آن‌را نیز در فایل index.html تعریف نمائید:

```
<script src="Scripts/App/blogger.js" type="text/javascript"></script>
<script src="Scripts/App/router.js" type="text/javascript"></script>
```

در اینجا Blogger همان شیء Application برنامه است که پیشتر در فایل Scripts\App\blogger.js تعریف کردیم. سپس به کمک متد Blogger.Router.map، اولین مسیریابی برنامه را افزوده‌ایم.

افزودن مسیریابی و قالب posts

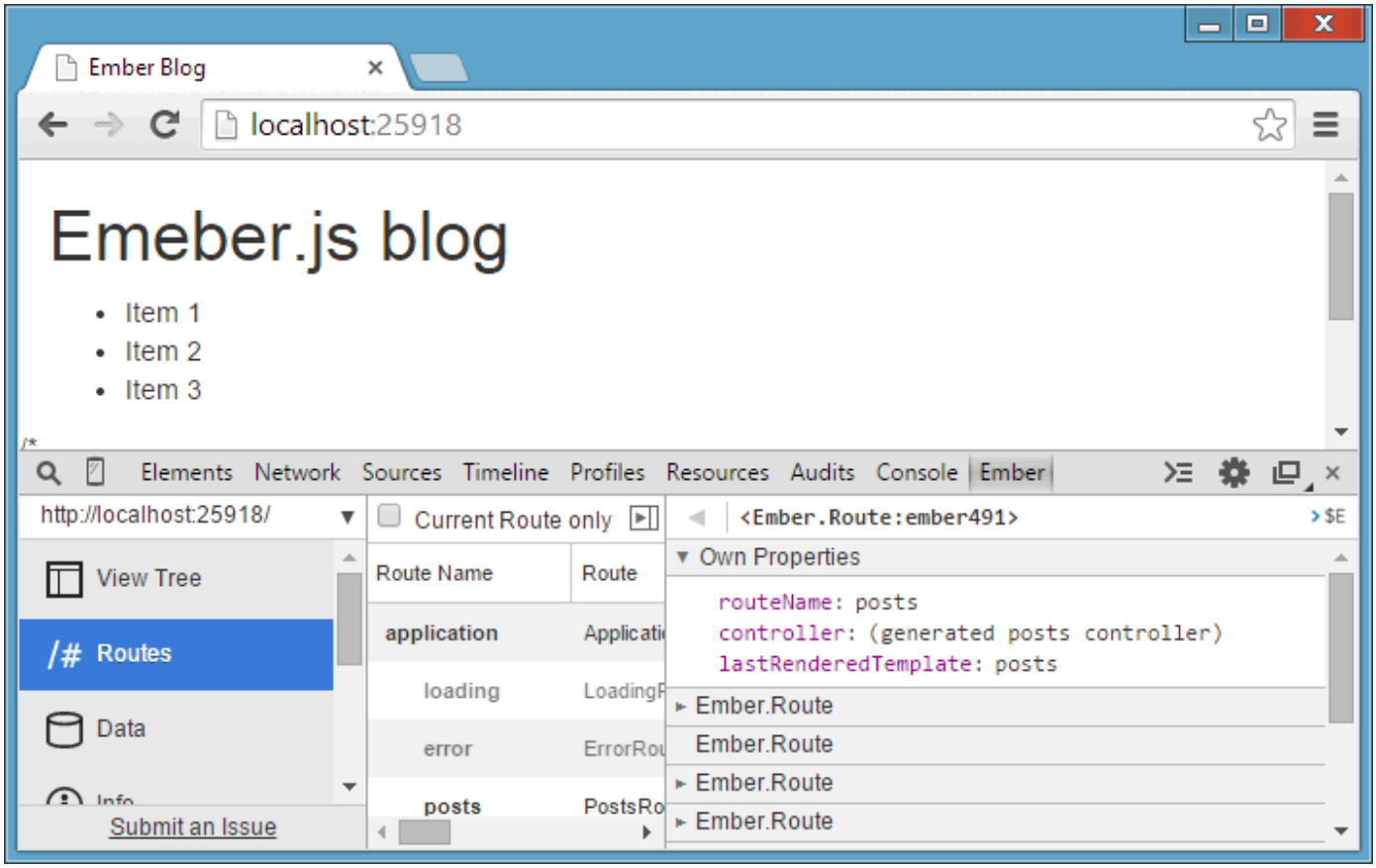
در ادامه فایل جدید Scripts\Templates\posts.hbs را اضافه کنید. به این ترتیب قالب خالی مطالب به پوشه‌ی templates اضافه می‌شود. محتوای این فایل را به نحو ذیل تنظیم کنید:

```
<div class="container">
  <h1>Emeber.js blog</h1>
  <ul>
   <li>Item 1</li>
   <li>Item 2</li>
   <li>Item 3</li>
  </ul>
</div>
```

در اینجا دیگر نیازی به ذکر تگ script از نوع text/x-handlebars نیست.
برای بارگذاری این قالب نیاز است آن‌را به template loader توضیح داده شده در ابتدای بحث، در فایل index.html اضافه کنیم:

```
<script>
 EmberHandlebarsLoader.loadTemplates([
 'posts'
 ]);
</script>
```

اکنون برنامه را اجرا کنید. به تصویر ذیل خواهید رسید که در آن افزونه‌ی Ember Inspector نیز نمایش داده شده‌است:



افزودن مسیریابی و قالب about

در ادامه قصد داریم صفحه‌ی about را اضافه کنیم. مجددا با افزودن مسیریابی آن به فایل Scripts\App\router.js شروع می‌کنیم:

```
Blogger.Router.map(function () {
  this.resource('posts', { path: '/' });
  this.resource('about');
});
```

سپس فایل قالب آن‌را در مسیر Scripts\Templates\about.hbs ایجاد خواهیم کرد؛ برای مثال با این محتوای فرضی:

```
<h1>About Ember Blog</h1>
<p>Bla bla bla!</p>
```

اکنون نام این فایل را به template loader فایل index.html اضافه می‌کنیم.

```
<script>
 EmberHandlebarsLoader.loadTemplates([
 'posts', 'about'
 ]);
</script>
```

اگر از قسمت قبل به خاطر داشته باشید، ساختار کلی قالب‌های ember.js یک چنین شکلی را دارد:

```
<script type="text/x-handlebars" data-template-name="about">

</script>
```

اسکریپت template loader، این تعاریف را به صورت خودکار اضافه می‌کند. مقدار data-template-name را نیز به نام فایل متناظر با آن تنظیم خواهد کرد و چون ember.js بر اساس ایده‌ی convention over configuration کار می‌کند، مسیریابی about با کنترلری به همین نام و قالبی هم نام پردازش خواهد شد. بنابراین نام فایل‌های قالب را باید بر اساس مسیریابی‌های متناظر با آن‌ها تعیین کرد.
برای آزمایش این مسیر و قالب جدید آن، آدرس http://localhost/#/about را بررسی کنید.

اضافه کردن منوی ثابت بالای سایت

روش اول این است که به ابتدای هر دو قالب about.hbs و posts.hbs، تعاریف منو را اضافه کنیم. مشکل این‌کار، تکرار کدها و پایین آمدن قابلیت نگهداری برنامه است. روش بهتر، افزودن کدهای مشترک بین صفحات، در قالب application برنامه است. نمونه‌ی آن‌را در مثال قسمت قبل مشاهده کرده‌اید. در اینجا چون قصد نداریم به صورت دستی قالب‌ها را به صفحه اضافه کنیم و همچنین برای ساده شدن نگهداری برنامه، قالب‌ها را در فایل‌های مجزایی قرار داده‌ایم، تنها کافی است فایل جدید Scripts\Templates\application.hbs را به پوشه‌ی قالب‌های برنامه اضافه کنیم؛ با این محتوا:

```
<div class='container'>
 <nav class='navbar navbar-default' role='navigation'>
  <ul class='nav navbar-nav'>
  <li>{{#link-to 'posts'}}Posts{{/link-to}}</li>
  <li>{{#link-to 'about'}}About{{/link-to}}</li>
  </ul>
 </nav>

  {{outlet}}
</div>
```

و سپس همانند قبل، نام فایل قالب اضافه شده را به template loader فایل index.html اضافه می‌کنیم:

```
<script>
 EmberHandlebarsLoader.loadTemplates([
 'posts', 'about', 'application'
 ]);
</script>
```

افزودن مسیریابی و قالب contact

برای افزودن صفحه‌ی تماس با مای سایت، ابتدا مسیریابی آن‌را در فایل Scripts\App\router.js تعریف می‌کنیم:

```
Blogger.Router.map(function () {
  this.resource('posts', { path: '/' });
  this.resource('about');
  this.resource('contact');
});
```

سپس قالب متناظر با آن‌را به نام Scripts\Templates\contact.hbs اضافه خواهیم کرد؛ فعلا با این محتوای اولیه:

```
<h1>Contact</h1>
<ul>
  <li>Phone: ...</li>
  <li>Email: ...</li>
</ul>
```

و بعد template loader فایل index.html را از وجود آن مطلع خواهیم کرد:

```
<script>
 EmberHandlebarsLoader.loadTemplates([
 'posts', 'about', 'application', 'contact' ]);
</script>
```

همچنین لینکی به مسیریابی آن‌را به فایل Scripts\Templates\application.hbs که منوی سایت در آن تعریف شده‌است، اضافه می‌کنیم:

```
<div class='container'>
 <nav class='navbar navbar-default' role='navigation'>
  <ul class='nav navbar-nav'>
  <li>{{#link-to 'posts'}}Posts{{/link-to}}</li>
  <li>{{#link-to 'about'}}About{{/link-to}}</li>
  <li>{{#link-to 'contact'}}Contact{{/link-to}}</li>
  </ul>
 </nav>

  {{outlet}}
</div>
```

تعریف مسیریابی تو در تو در صفحه‌ی contact

در حال حاضر صفحه‌ی Contact، ایمیل و شماره تماس را در همان بار اول نمایش می‌دهد. می‌خواهیم این دو را تبدیل به دو لینک جدید کنیم که با کلیک بر روی هر کدام، محتوای مرتبط، در قسمتی از همان صفحه بارگذاری شده و نمایش داده شود.
برای اینکار نیاز است مسیریابی را تو در تو تعریف کنیم:

```
Blogger.Router.map(function () {
  this.resource('posts', { path: '/' });
  this.resource('about');
  this.resource('contact', function () {
   this.resource('email');
   this.resource('phone');
  });
});
```

اگر مسیریابی‌های email و یا phone را به صورت مستقل مانند about و یا posts تعریف کنیم، با کلیک کاربر بر روی لینک متناظر با هر کدام، یک صفحه‌ی کاملا جدید نمایش داده می‌شود. اما در اینجا قصد داریم تنها قسمت کوچکی از همان صفحه‌ی contact را با محتوای ایمیل و یا شماره تماس جایگزین نمائیم. به همین جهت مسیریابی‌های متناظر را در اینجا به صورت تو در تو و ذیل مسیریابی contact تعریف کرده‌ایم.

پس از آن دو فایل قالب جدید Scripts\Templates\email.hbs را با محتوای:

```
<h2>Email</h2>
<p>
<span></span> Email name@site.com.
</p>
```

و فایل قالب Scripts\Templates\phone.hbs را با محتوای:

```
<h2>Phone</h2>
<p>
<span></span> Call 12345678.
</p>
```

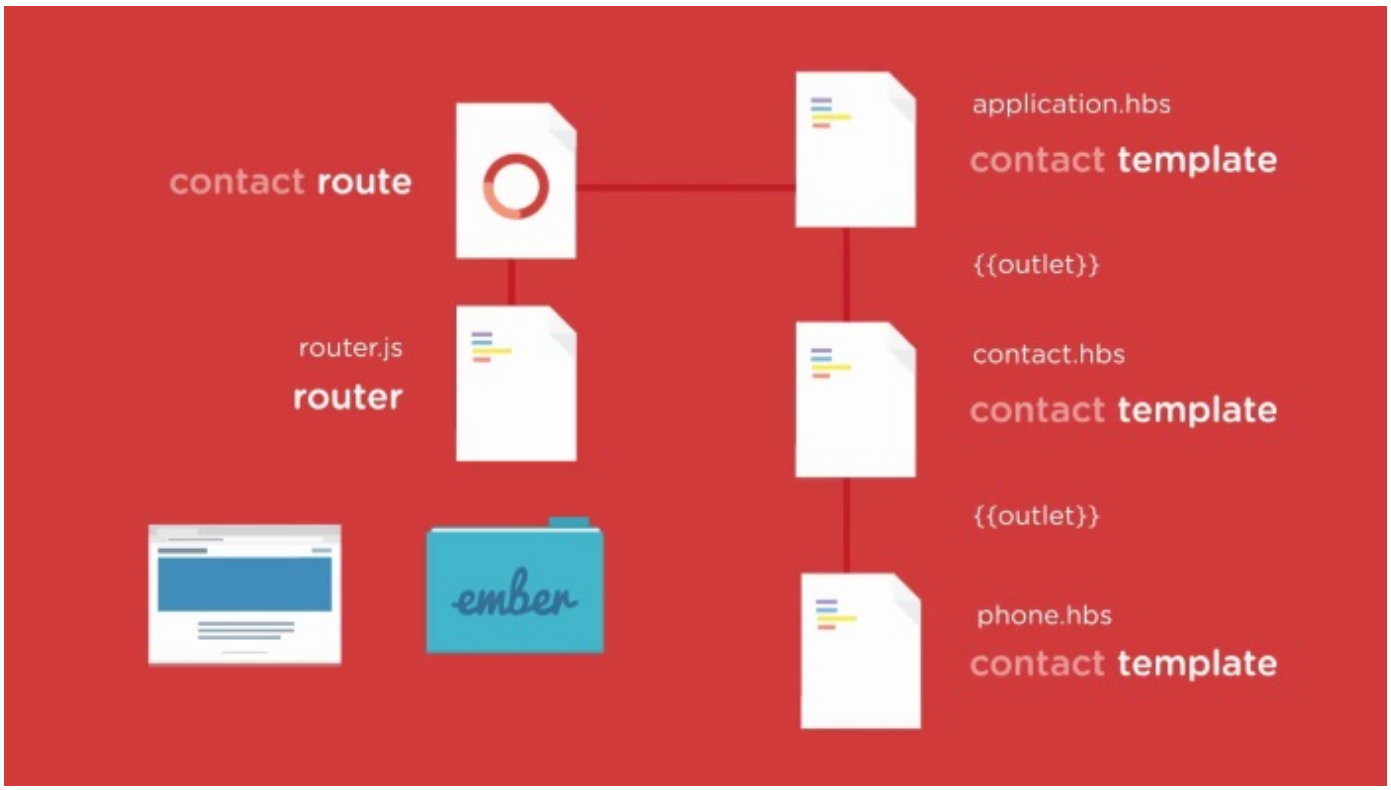
به پوشه‌ی قالب‌ها اضافه نمائید به همراه معرفی نام آن‌ها به template loader برنامه در صفحه‌ی index.html :

```
<script>
 EmberHandlebarsLoader.loadTemplates([
 'posts', 'about', 'application', 'contact', 'email', 'phone' ]);
</script>
```

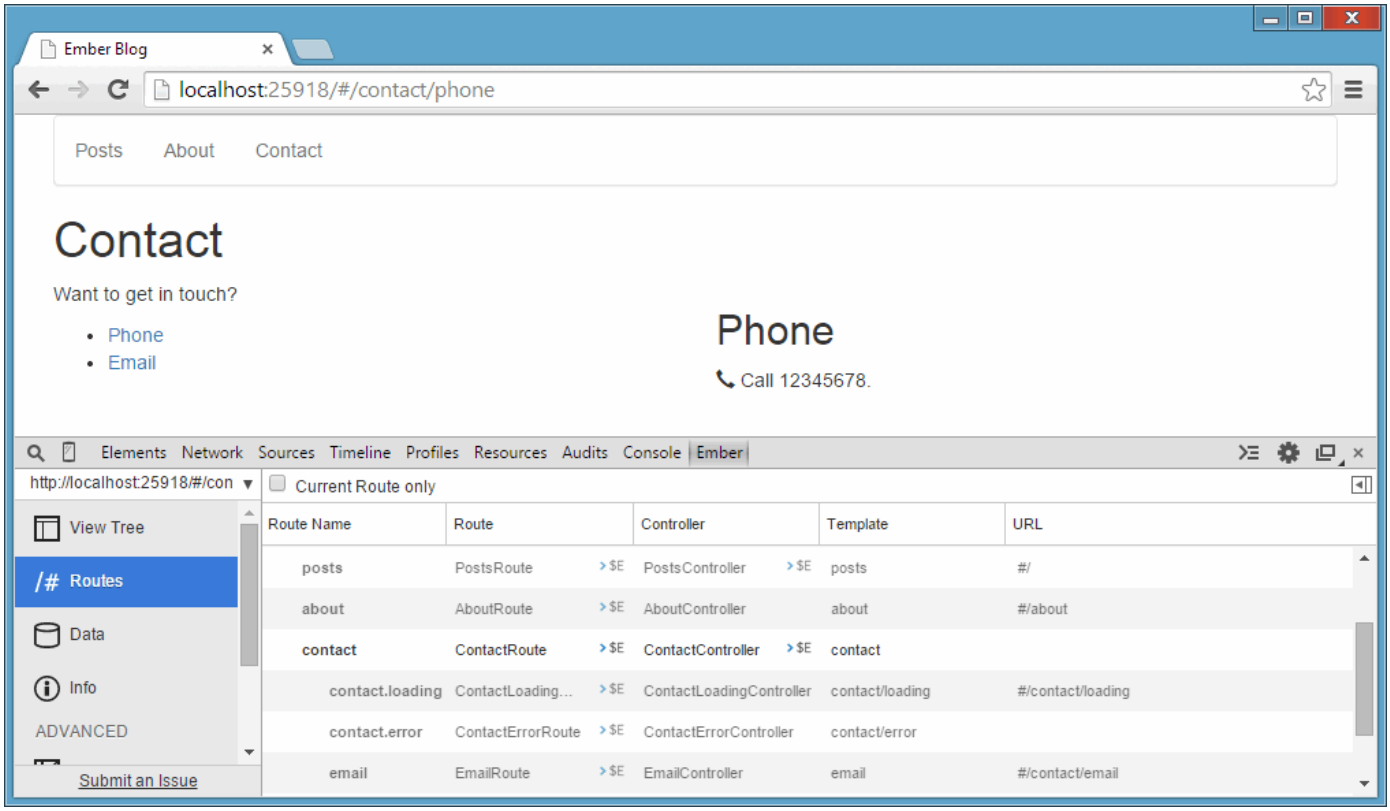
اکنون به قالب contact.hbs مجددا مراجعه کرده و تعاریف آن را به نحو ذیل تغییر دهید:

```
<h1>Contact</h1>
<div class="row">
  <div class="col-md-6">
   <p>
    Want to get in touch?
    <ul>
      <li>{{#link-to 'phone'}}Phone{{/link-to}}</li>
      <li>{{#link-to 'email'}}Email{{/link-to}}</li>
    </ul>
   </p>
  </div>
  <div class="col-md-6">
   {{outlet}}
  </div>
</div>
```

در اینجا دو لینک به مسیریابی‌های Phone و Email تعریف شده‌اند. همچنین {{outlet}} نیز قابل مشاهده است. با کلیک بر روی لینک Phone، مسیریابی آن فعال شده و سپس قالب متناظر با آن در قسمت {{outlet}} رندر می‌شود. در مورد لینک Email نیز به همین نحو رفتار خواهد شد.



در اینجا نحوه‌ی پردازش مسیریابی contact را ملاحظه می‌کنید. ابتدا درخواستی جهت مشاهده‌ی آدرس http://localhost/#/contact دریافت می‌شود. سپس router این درخواست را به مسیریابی همنامی منتقل می‌کند. این مسیریابی ابتدا قالب عمومی application را رندر کرده و سپس قالب اصلی و همنام مسیریابی جاری یا همان contact.hbs را رندر می‌کند. در این صفحه چون مسیریابی تو در تویی تعریف شده‌است، اگر درخواستی برای مشاهده‌ی http://localhost/#/contact/phone دریافت شود، محتوای آن‌را در {{outlet}} قالب contact.hbs جاری رندر می‌کند.



کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

EmberJS03_01.zip

عنوان: **ساخت یک بلاگ ساده با Ember.js، قسمت دوم**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۱۷ ۸:۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, jQuery, SPA, EmberJS

پس از تهیه ساختار اولیه‌ی بلاگی مبتنی بر ember.js در قسمت قبل ، در ادامه قصد داریم امکانات تعاملی را به آن اضافه کنیم. بنابراین کار را با تعریف کنترلرها که تعیین کننده‌ی رفتار برنامه هستند، ادامه می‌دهیم.

اضافه کردن دکمه‌ی More info به صفحه‌ی About و مدیریت کلیک بر روی آن

فایل Scripts\Templates\about.hbs را گشوده و سپس محتوای فعلی آن را به نحو ذیل تکمیل کنید:

```
<h2>About Ember Blog</h2>

<p>Bla bla bla!</p>

<button class="btn btn-primary" {{action 'showRealName' }}>more info</button>
```

در ember.js اگر قصد مدیریت عملی را که قرار است توسط کلیک بر روی المانی رخ دهد، داشته باشیم، می‌توان از handlebar helper ایی به نام action استفاده کرد. سپس برای تهیه کدهای مرتبط با آن، این اکشن را باید در کنترلر متناظر با route جاری (مسیریابی about) اضافه کنیم.
به همین جهت فایل جدید Scripts\Controllers\about.js را در پوشه‌ی کنترلرهای سمت کاربر اضافه کنید (نام آن با نام مسیریابی یکی است)؛ با این محتوا:

```
Blogger.AboutController = Ember.Controller.extend({
  actions: {
   showRealName: function () {
    alert("You clicked at showRealName of AboutController.");
   }
  }
});
```

کنترلرها به صورت یک خاصیت جدید به شیء Application برنامه اضافه می‌شوند. مطابق اصول نامگذاری ember.js، نام خاصیت کنترلر با حروف بزرگ متناظر با route آن شروع می‌شود و به نام Controller ختم خواهد شد. به این ترتیب ember.js هرگاه قصد پردازش مسیریابی about را داشته باشد، می‌داند که باید از کدام شیء جهت پردازش اعمال کاربر استفاده کند.
در ادامه این خاصیت را با تهیه یک زیرکلاس از کلاس پایه Controller تهیه شده توسط ember.js مقدار دهی می‌کنیم. به این ترتیب به کلیه امکانات این کلاس پایه دسترسی خواهیم داشت؛ به علاوه می‌توان ویژگی‌های سفارشی را نیز به آن افزود. برای مثال در اینجا در قسمت actions آن، دقیقا مطابق نام اکشنی که در فایل about.hbs تعریف کرده‌ایم، یک متد جدید اضافه شده‌است.

پس از تعریف کنترلر about.js نیاز است مدخل متناظر با آن‌را به فایل index.html برنامه نیز در انتهای تعاریف موجود، اضافه کرد:

```
<script src="Scripts/Controllers/about.js" type="text/javascript"></script>
```

اکنون یکبار برنامه را اجرا کرده و در صفحه‌ی about بر روی دکمه‌ی more info کلیک کنید.

اضافه کردن دکمه‌ی ارسال پیام خصوصی به صفحه‌ی Contact و مدیریت کلیک بر روی آن

در ادامه به قالب فعلی Scripts\Templates\contact.hbs یک دکمه را جهت ارسال پیام خصوصی اضافه می‌کنیم.

```
<h1>Contact</h1>
<div class="row">
  <div class="col-md-6">
   <p>
    Want to get in touch?
    <ul>
      <li>{{#link-to 'phone'}}Phone{{/link-to}}</li>
      <li>{{#link-to 'email'}}Email{{/link-to}}</li>
    </ul>
   </p>

   <p>
    Or, click here to send a secret message:
   </p>
   <button class="btn btn-primary" {{action 'sendMessage' }}>Send message</button>
  </div>
  <div class="col-md-6">
   {{outlet}}
  </div>
</div>
```

سپس برای مدیریت اکشن جدید sendMessage نیاز است کنترلر آن‌را نیز تعریف کنیم. با توجه به نام مسیریابی جاری، نام این کنترلر نیز contact خواهد بود. برای این منظور ابتدا فایل جدید Scripts\Controllers\contact.js را اضافه نمائید؛ با این محتوا:

```
Blogger.ContactController = Ember.Controller.extend({
  actions: {
   sendMessage: function () {
    var message = prompt('Type your message here:');
   }
  }
});
```

همچنین مدخل متناظر با فایل contact.js نیز باید به صفحه‌ی index.html اضافه شود:

```
<script src="Scripts/Controllers/contact.js" type="text/javascript"></script>
```

نمایش تصویری تعاملی در صفحه‌ی about

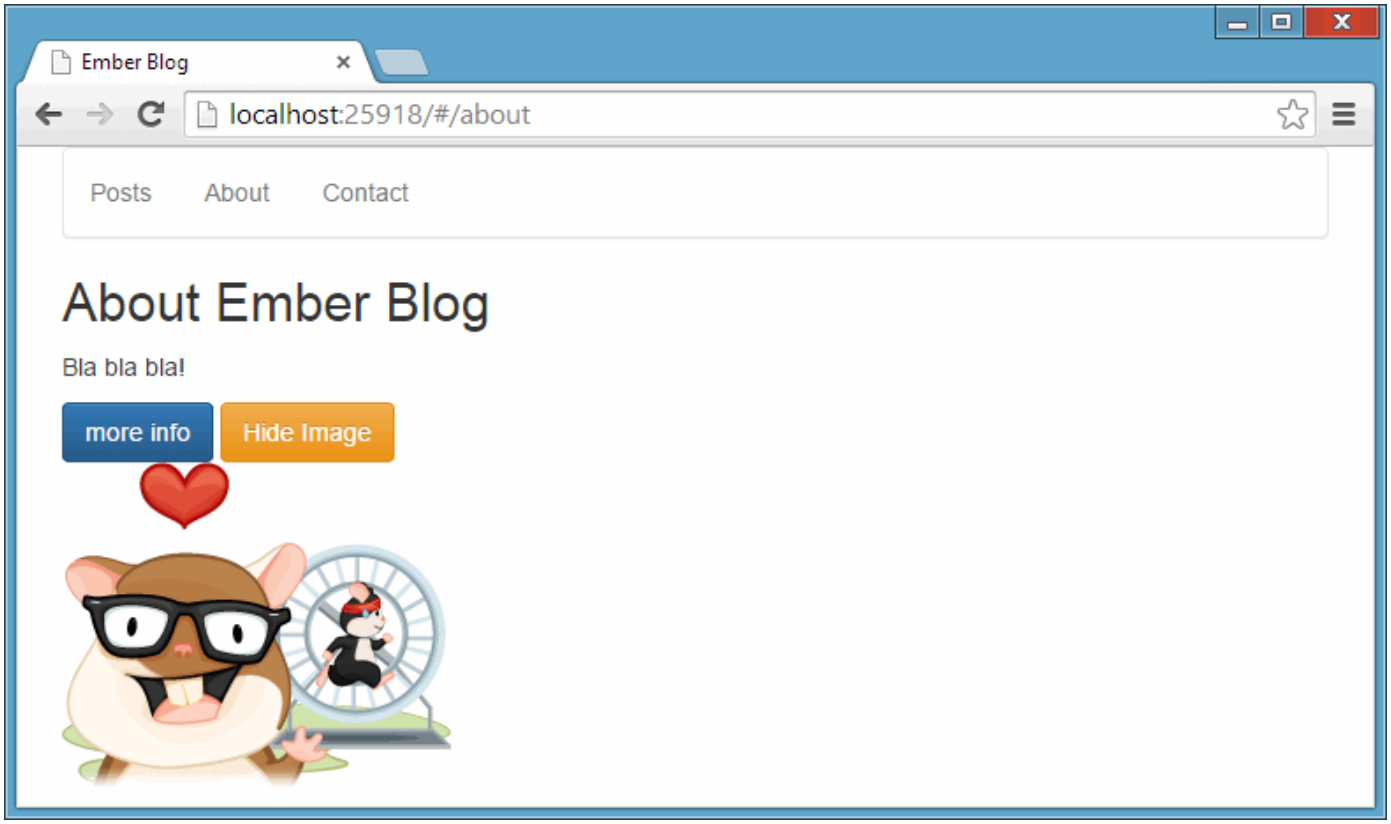
تا اینجا با نحوه‌ی تعریف اکشن‌ها در قالب‌ها و مدیریت آن‌ها توسط کنترلرهای متناظر آشنا شدیم. در ادامه قصد داریم با اصول binding اطلاعات در ember.js آشنا شویم. برای مثال فرض کنید می‌خواهیم دکمه‌ای را در صفحه‌ی about قرار داده و با کلیک بر روی آن، لوگوی ember.js را که به صورت یک تصویر مخفی در صفحه قرار دارد، نمایان کنیم. برای اینکار نیاز است خاصیتی را در کنترلر متناظر، تعریف کرده و سپس آن‌را به template جاری bind کرد.
برای این منظور فایل Scripts\Templates\about.hbs را گشوده و تعاریف موجود آن‌را به نحو ذیل تکمیل کنید:

```
<h2>About Ember Blog</h2>
<p>Bla bla bla!</p>
<button class="btn btn-primary" {{action 'showRealName' }}>more info</button>

{{#if isAuthorShowing}}
<button class="btn btn-warning" {{action 'hideAuthor' }}>Hide Image</button>
<p><img src="Content/images/ember-productivity-sm.png"></p>
{{else}}
<button class="btn btn-info" {{action 'showAuthor' }}>Show Image</button>
{{/if}}
```

در اینجا بر اساس مقدار خاصیت isAuthorShowing تصمیم گیری خواهد شد که آیا تصویر لوگوی ember.js نمایش داده شود یا خیر. همچنین دو اکشن نمایش و مخفی کردن تصویر نیز اضافه شده‌اند که با کلیک بر روی هر کدام، سبب تغییر وضعیت خاصیت isAuthorShowing خواهیم شد.
کنترلر about (فایل Scripts\Controllers\about.js) جهت مدیریت این خاصیت جدید، به همراه دو اکشن تعریف شده، اینبار به نحو ذیل تغییر خواهد یافت:

```
Blogger.AboutController = Ember.Controller.extend({
  isAuthorShowing: false,
  actions: {
   showRealName: function () {
    alert("You clicked at showRealName of AboutController.");
   },
   showAuthor: function () {
    this.set('isAuthorShowing', true);
   },
   hideAuthor: function () {
    this.set('isAuthorShowing', false);
   }
  }
});
```

ابتدا خاصیت isAuthorShowing به کنترلر اضافه شده‌است. از این خاصیت بار اولی که مسیر http://localhost:25918/#/about توسط کاربر درخواست می‌شود، استفاده خواهد شد.
سپس در دو متد showAuthor و hideAuthor که به اکشن‌های دو دکمه‌ی جدید تعریف شده در قالب about متصل خواهند شد، نحوه‌ی تغییر مقدار خاصیت isAuthorShowing را توسط متد set ملاحظه می‌کنید.
این قسمت مهم‌ترین تفاوت ember.js با jQuery است. در jQuery مستقیما المان‌های صفحه در همانجا تغییر داده می‌شوند. در ember.js منطق مدیریت کننده‌ی رابط کاربری و کدهای قالب متناظر با آن از هم جدا شده‌اند تا بتوان یک برنامه‌ی بزرگ را بهتر مدیریت کرد. همچنین در اینجا مشخص است که هر قسمت و هر فایل، چه ارتباطی با سایر اجزای تعریف شده دارد و چگونه به هم متصل شده‌اند و اینبار شاهد انبوهی از کدهای جاوا اسکریپتی مخلوط بین المان‌های HTML صفحه نیستیم.

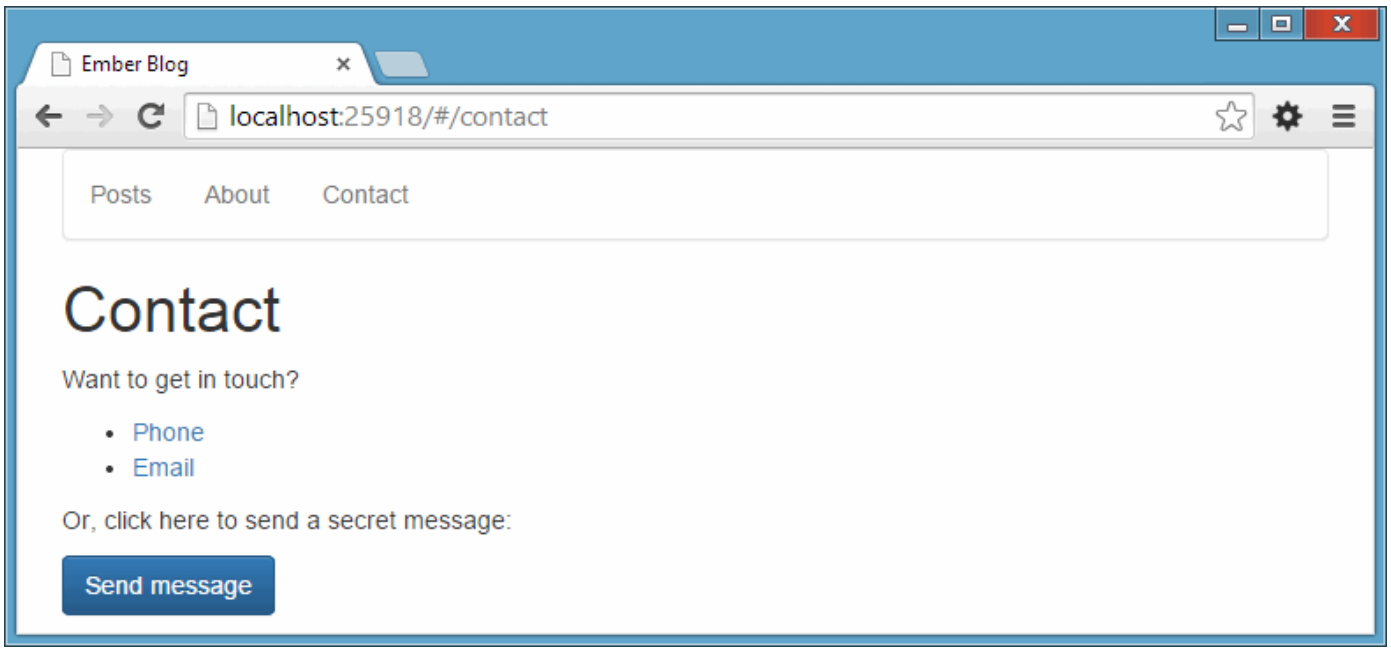


نمایش پیامی به کاربر پس از ارسال پیام خصوصی در صفحه‌ی تماس با ما

قصد داریم ویژگی مشابهی را به صفحه‌ی contact نیز اضافه کنیم. اگر کاربر بر روی دکمه‌ی ارسال پیام کلیک کرد، پیام تشکری به همراه عددی ویژه به او نمایش خواهیم داد.
برای این‌کار قالب Scripts\Templates\contact.hbs را به نحو ذیل تکمیل کنید:

```
<h1>Contact</h1>
<div class="row">
  <div class="col-md-6">
   <p>
    Want to get in touch?
    <ul>
      <li>{{#link-to 'phone'}}Phone{{/link-to}}</li>
      <li>{{#link-to 'email'}}Email{{/link-to}}</li>
    </ul>
   </p>
   {{#if messageSent}}
   <p>
    Thank you. Your message has been sent.
    Your confirmation number is {{confirmationNumber}}.
   </p>
   {{else}}
   <p>
    Or, click here to send a secret message:
   </p>
   <button class="btn btn-primary" {{action 'sendMessage' }}>Send message</button>
   {{/if}}
  </div>
  <div class="col-md-6">
   {{outlet}}
  </div>
</div>
```

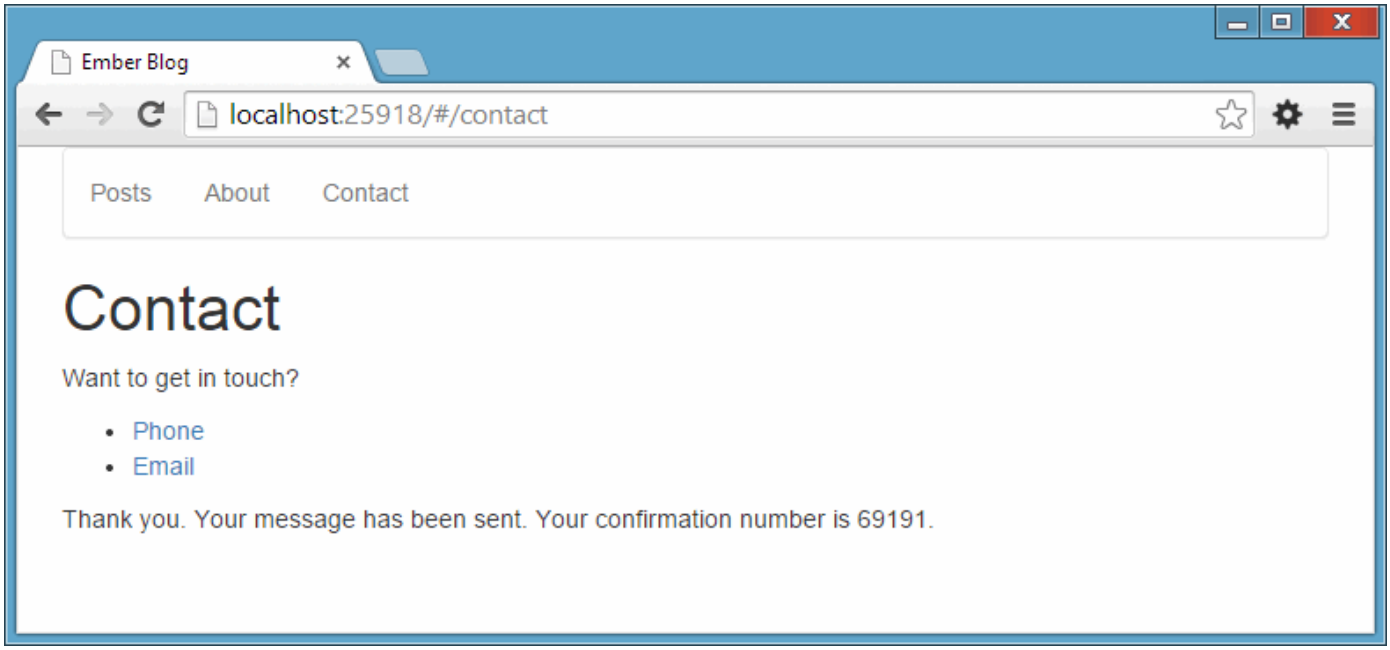
در آن شرط بررسی if messageSent اضافه شده‌است؛ به همراه نمایش confirmationNumber در انتهای پیام تشکر.



برای تعریف منطق مرتبط با این خواص، به کنترلر contact واقع در فایل Scripts\Controllers\contact.js مراجعه کرده و آن‌را به نحو ذیل تغییر می‌دهیم:

```
Blogger.ContactController = Ember.Controller.extend({
  messageSent: false,
  actions: {
   sendMessage: function () {
    var message = prompt('Type your message here:');
    if (message) {
      this.set('confirmationNumber', Math.round(Math.random() * 100000));
      this.set('messageSent', true);
    }
   }
  }
});
```

همانطور که مشاهده می‌کنید، مقدار اولیه خاصیت messageSent مساوی false است. بنابراین در قالب contact.hbs قسمت else شرط نمایش داده می‌شود. اگر کاربر پیامی را وارد کند، خاصیت confirmationNumber به یک عدد اتفاقی و خاصیت messageSent به true تنظیم خواهد شد. به این ترتیب اینبار به صورت خودکار پیام تشکر به همراه عددی اتفاقی، به کاربر نمایش داده می‌شود.



بنابراین به صورت خلاصه، کار کنترلر، مدیریت منطق نمایشی برنامه است و برای اینکار حداقل دو مکانیزم را ارائه می‌دهد: اکشن‌ها و خواص. اکشن‌ها بیانگر نوعی رفتار هستند؛ برای مثال نمایش یک popup و یا تغییر مقدار یک خاصیت. مقدار خواص را می‌توان مستقیما در صفحه نمایش داد و یا از آن‌ها جهت پردازش عبارات شرطی و نمایش قسمت خاصی از قالب جاری نیز می‌توان کمک گرفت.

کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

EmberJS03_02.zip

عنوان: **ساخت یک بلاگ ساده با Ember.js، قسمت سوم**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۱۸ ۸:۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, jQuery, SPA, EmberJS

پس از ایجاد کنترلرها ، در این قسمت سعی خواهیم کرد تا آرایه‌ای ثابت از مطالب و نظرات را در سایت نمایش دهیم. همچنین امکان ویرایش اطلاعات را نیز به این آرایه‌های جاوا اسکریپتی مدل، اضافه خواهیم کرد.

تعریف مدل سمت کاربر برنامه

فایل جدید Scripts\App\store.js را اضافه کرده و محتوای آن‌را به نحو ذیل تغییر دهید:

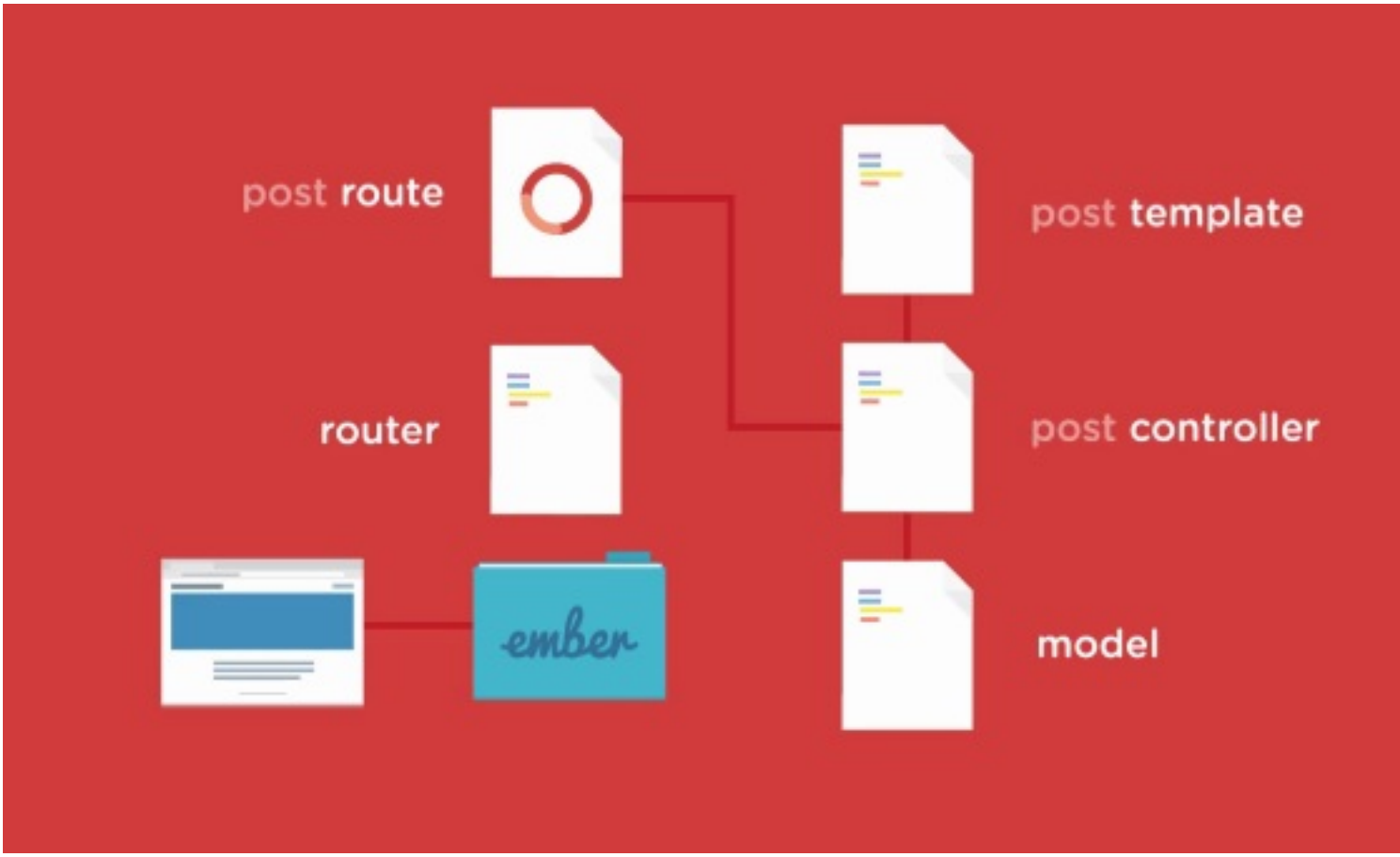
```
var posts = [
  {
      id: '1',
      title: "Getting Started with Ember.js",
      body: "Bla bla bla 1."
  },
  {
      id: '2',
      title: "Routes and Templates",
      body: "Bla bla bla 2."
  },
  {
      id: '3',
      title: "Controllers",
      body: "Bla bla bla 3."
  }
];

var comments = [
    {
        id: '1',
        postId: '3',
        text: 'Thanks!'
    },
    {
        id: '2',
        postId: '3',
        text: 'Good to know that!'
    },
    {
        id: '3',
        postId: '1',
        text: 'Great!'
    }
];
```

در اینجا دو آرایه ثابت از اشیاء مطالب و نظرات را مشاهده می‌کنید.
سپس جهت استفاده از آن، تعریف مدخل آن‌را به فایل index.html، پیش از تعاریف کنترلرها اضافه خواهیم کرد:

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

ویرایش قالب مطالب برای نمایش لیستی از عناوین ارسالی

قالب فعلی Scripts\Templates\posts.hbs صرفا دارای یک سری عنوان درج شده به صورت مستقیم در صفحه است. اکنون قصد داریم آن‌را جهت نمایش لیستی از آرایه مطالب تغییر دهیم.



همانطور که در تصویر ملاحظه می‌کنید، با درخواست آدرس صفحه‌ی مطالب، router آن مسیریابی متناظری را یافته و سپس بر این اساس، template، کنترلر و مدلی را انتخاب می‌کند. به صورت پیش فرض، قالب و کنترلر انتخاب شده، مواردی هستند همنام با مسیریابی جاری. اما مقدار پیش فرضی برای model وجود ندارد و باید آن‌را به صورت دستی مشخص کرد.
برای این منظور فایل Scripts\Routes\posts.js را به پوشه‌ی routes با محتوای ذیل اضافه کنید:

```
Blogger.PostsRoute = Ember.Route.extend({
    controllerName: 'posts',
    renderTemplare: function () {
        this.render('posts');
    },
    model: function () {
        return posts;
    }
});
```

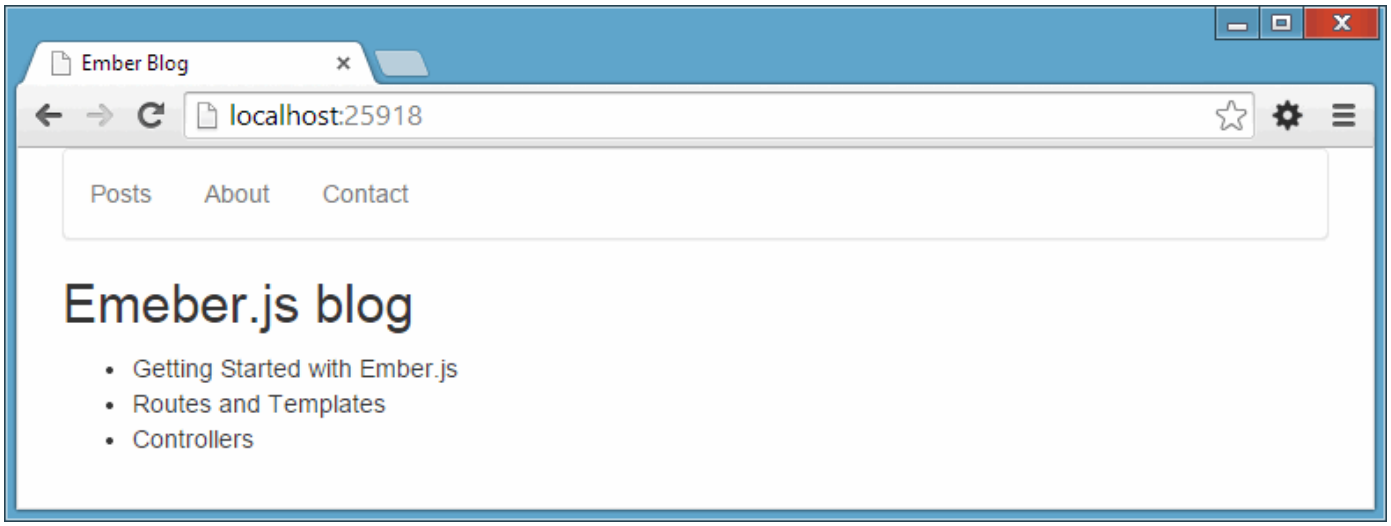
در اینجا صرفا جهت نمایش پیش فرض‌ها و نحوه‌ی کار یک route، دو خاصیت controllerName و renderTemplare آن نیز مقدار دهی شده‌اند. این دو خاصیت به صورت پیش فرض، همنام مسیریابی جاری مقدار دهی می‌شوند و نیازی به ذکر صریح آن‌ها نیست. اما خاصیت model یک مسیریابی است که باید دقیقا مشخص شود. در اینجا مقدار آن‌را به آرایه posts تعریف شده در فایل Scripts\App\store.js تنظیم کرده‌ایم. به این ترتیب مدل تعریف شده در اینجا، به صورت خودکار در کنترلر posts و قالب متناظر با آن، قابل استفاده خواهد بود.
همچنین اگر به خاطر داشته باشید، در پوشه‌ی کنترلرها فایل posts.js تعریف نشده‌است. اگر اینکار صورت نگیرد، ember.js به صورت خودکار کنترلر پیش فرضی را ایجاد خواهد کرد. در کل، یک قالب هیچگاه به صورت مستقیم با مدل کار نمی‌کند. این کنترلر است که مدل را در اختیار یک قالب قرار می‌دهد.
سپس مدخل تعریف این فایل را به فایل index.html، پس از تعاریف کنترلرها اضافه نمائید:

```
<script src="Scripts/Routes/posts.js" type="text/javascript"></script>
```

اکنون فایل Scripts\Templates\posts.hbs را گشوده و به نحو ذیل، جهت نمایش عناوین مطالب، ویرایش کنید:

```
<h2>Emeber.js blog</h2>
<ul>
    {{#each post in model}}
    <li>{{post.title}}</li>
    {{/each}}
</ul>
```

در این قالب، حلقه‌ای بر روی عناصر model تشکیل شده و سپس خاصیت title هر عضو نمایش داده می‌شود.



نمایش لیست آخرین نظرات ارسالی

در ادامه قصد داریم تا آرایه comments ابتدای بحث را در صفحه‌ای جدید نمایش دهیم. بنابراین نیاز است تا ابتدا مسیریابی آن تعریف شود. بنابراین فایل Scripts\App\router.js را گشوده و مسیریابی جدید recent-comments را به آن اضافه کنید:

```
Blogger.Router.map(function () {
    this.resource('posts', { path: '/' });
    this.resource('about');
    this.resource('contact', function () {
        this.resource('email');
        this.resource('phone');
    });
    this.resource('recent-comments');
});
```

سپس جهت تعیین مدل این مسیریابی جدید نیاز است تا فایل Scripts\Routes\recent-comments.js را در پوشه‌ی routes با محتوای ذیل اضافه کرد:

```
Blogger.RecentCommentsRoute = Ember.Route.extend({
    model: function () {
        return comments;
    }
});
```

در اینجا آرایه comments بازگشتی، همان آرایه‌ای است که در ابتدای بحث در فایل Scripts\App\store.js تعریف کردیم.
همچنین نیاز است تا تعریف مدخل این فایل جدید را نیز به انتهای تعاریف مداخل فایل index.html اضافه کنیم:

```
<script src="Scripts/Routes/recent-comments.js" type="text/javascript"></script>
```

اکنون قالب application واقع در فایل Scripts\Templates\application.hbs را جهت افزودن منوی مرتبط با این مسیریابی جدید، به نحو ذیل ویرایش خواهیم کرد:

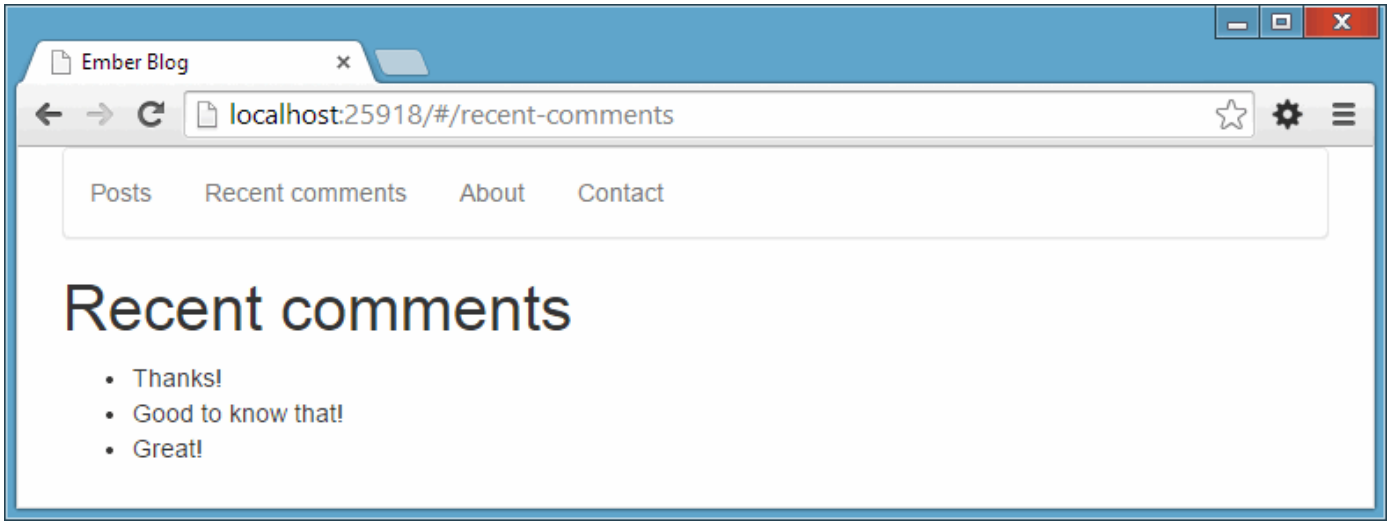
```
<div class='container'>
    <nav class='navbar navbar-default' role='navigation'>
        <ul class='nav navbar-nav'>
            <li>{{#link-to 'posts'}}Posts{{/link-to}}</li>
            <li>{{#link-to 'recent-comments'}}Recent comments{{/link-to}}</li>
            <li>{{#link-to 'about'}}About{{/link-to}}</li>
            <li>{{#link-to 'contact'}}Contact{{/link-to}}</li>
        </ul>
    </nav>
    {{outlet}}
</div>
```

و در آخر قالب جدید Scripts\Templates\recent-comments.hbs را برای نمایش لیست آخرین نظرات، با محتوای زیر اضافه می‌کنیم:

```
<h1>Recent comments</h1>
<ul>
  {{#each comment in model}}
    <li>{{comment.text}}</li>
  {{/each}}
</ul>
```

برای فعال شدن آن نیاز است تا تعریف این قالب جدید را به template loader برنامه، در فایل index.html اضافه کنیم:

```
<script type="text/javascript">
    EmberHandlebarsLoader.loadTemplates([
       'posts', 'about', 'application', 'contact', 'email', 'phone',
       'recent-comments'
    ]);
</script>
```



نمایش مجزای هر مطلب در یک صفحه‌ی جدید

تا اینجا در صفحه‌ی اول سایت، لیست عناوین مطالب را نمایش دادیم. در ادامه نیاز است تا بتوان هر عنوان را به صفحه‌ی متناظر و اختصاصی آن لینک کرد؛ برای مثال لینکی مانند http://localhost:25918/#/posts/3 به سومین مطلب ارسالی اشاره می‌کند. Ember.js به عدد 3 در اینجا، یک dynamic segment می‌گوید. از این جهت که مقدار آن بر اساس شماره مطلب درخواستی، متفاوت خواهد بود. برای پردازش این نوع آدرس‌ها نیاز است مسیریابی ویژه‌ای را تعریف کرد. فایل Scripts\App\router.js را گشوده و سپس مسیریابی post را به نحو ذیل به آن اضافه نمائید:

```
Blogger.Router.map(function () {
    this.resource('posts', { path: '/' });
    this.resource('about');
    this.resource('contact', function () {
        this.resource('email');
        this.resource('phone');
    });
    this.resource('recent-comments');
    this.resource('post', { path: 'posts/:post_id' });
});
```

قسمت پویای مسیریابی با یک : مشخص می‌شود.
با توجه به اینکه این مسیریابی جدید post نام گرفت (جهت نمایش یک مطلب)، به صورت خودکار، کنترلر و قالبی به همین نام را بارگذاری می‌کند. همچنین مدل خود را نیز باید از مسیریابی خاص خود دریافت کند. بنابراین فایل جدید Scripts\Routes\post.js را در پوشه‌ی routes با محتوای ذیل اضافه کنید:

```
Blogger.PostRoute = Ember.Route.extend({
    model: function (params) {
        return posts.findBy('id', params.post_id);
    }
});
```

در اینجا مدل مسیریابی post بر اساس پارامتری به نام params تعیین می‌شود. این پارامتر حاوی مقدار متغیر پویای post_id که در مسیریابی جدید post مشخص کردیم می‌باشد. در ادامه از آرایه posts تعریف شده در ابتدای بحث، توسط متد findBy که توسط Ember.js اضافه شده‌است، عنصری را که خاصیت id آن مساوی post_id دریافتی است، انتخاب کرده و به عنوان مقدار مدل بازگشت می‌دهیم.
برای مثال، جهت آدرس http://localhost:25918/#/posts/3، مقدار post_id به صورت خودکار به عدد 3 تنظیم می‌شود.

پس از آن نیاز است مدخل این فایل جدید را در صفحه‌ی index.html نیز اضافه کنیم:

```
<script src="Scripts/Routes/post.js" type="text/javascript"></script>
```

در ادامه برای نمایش اطلاعات مدل نیاز است قالب جدید Scripts\Templates\post.hbs را با محتوای زیر اضافه کنیم:

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

و template loader صفحه‌ی index.html را نیز باید از وجود آن باخبر کرد:

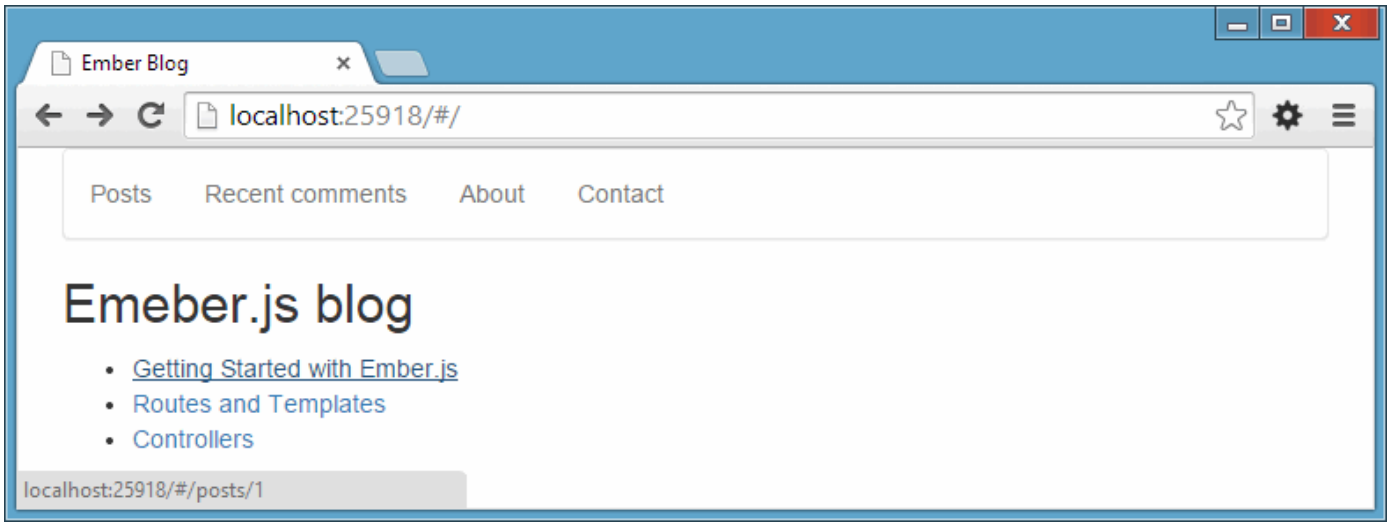
```
<script type="text/javascript">
    EmberHandlebarsLoader.loadTemplates([
       'posts', 'about', 'application', 'contact', 'email', 'phone',
       'recent-comments', 'post'
    ]);
</script>
```

اکنون به قالب Scripts\Templates\posts.hbs مراجعه کرده و هر عنوان را به مطلب متناظر با آن لینک می‌کنیم:

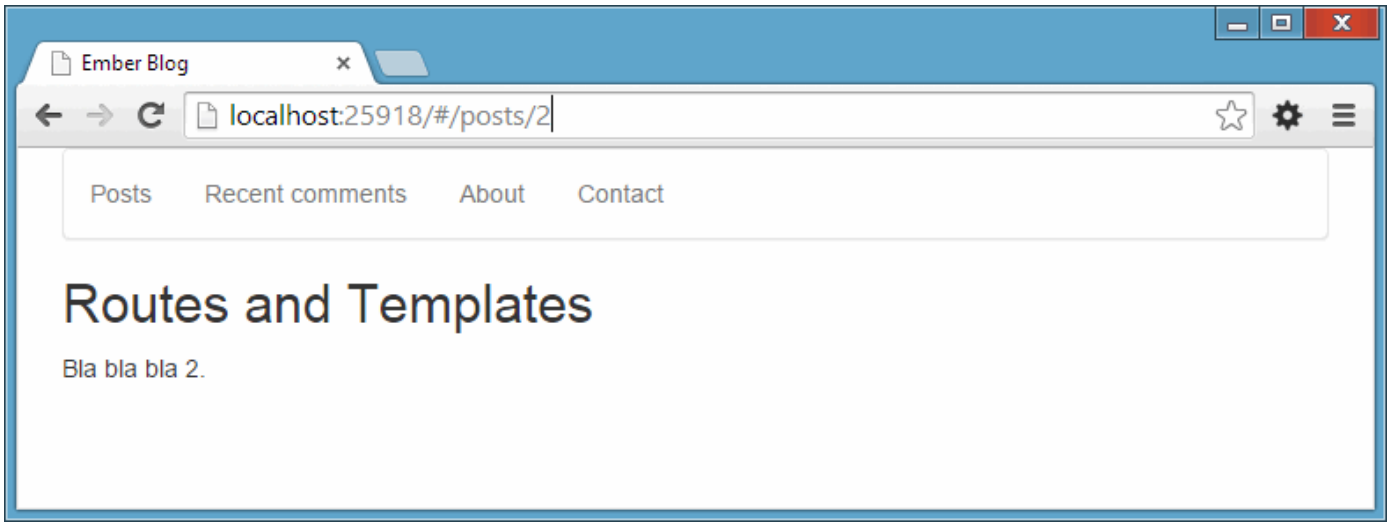
```
<h2>Emeber.js blog</h2>
<ul>
```

```
    {{#each post in model}}
    <li>{{#link-to 'post' post.id}}{{post.title}}{{/link-to}}</li>
    {{/each}}
</ul>
```

همانطور که ملاحظه می‌کنید، link-to امکان پذیرش id یک مطلب را به صورت متغیر نیز دارا است که سبب خواهد شد تا عناوین، به مطالب متناظر لینک شوند:



همچنین با کلیک بر روی هر عنوان نیز مطلب مرتبط نمایش داده خواهد شد:



افزودن امکان ویرایش مطالب

می‌خواهیم در صفحه‌ی نمایش جزئیات یک مطلب، امکان ویرایش آن‌را نیز فراهم کنیم. بنابراین فایل Scripts\Templates\post.hbs را گشوده و محتوای آن‌را به نحو ذیل ویرایش کنید:

```
<h2>{{title}}</h2>
{{#if isEditing}}
<form>
    <div class="form-group">
        <label for="title">Title</label>
        {{input value=title id="title" class="form-control"}}
    </div>
    <div class="form-group">
        <label for="body">Body</label>
        {{textarea value=body id="body" class="form-control" rows="5"}}
    </div>
    <button class="btn btn-primary" {{action 'save' }}>Save</button>
</form>
{{else}}
<p>{{body}}</p>
<button class="btn btn-primary" {{action 'edit' }}>Edit</button>
{{/if}}
```

شبیه به این if و else را در قسمت قبل حین ایجاد صفحات about و یا contact نیز مشاهده کرده‌اید. در اینجا اگر خاصیت isEditing مساوی true باشد، فرم ویرایش اطلاعات ظاهر می‌شود و اگر خیر، محتوای مطلب جاری نمایش داده خواهد شد.
در فرم تعریف شده، المان‌های ورودی اطلاعات از handlebar helperهای ویژه‌ی input و textarea استفاده می‌کنند؛ بجای المان‌های متداول HTML. همچنین value یکی به title و دیگری به body تنظیم شده‌است (خواص مدل ارائه شده توسط کنترلر متصل به قالب). این مقادیر نیز داخل '' قرار ندارند؛ به عبارتی در یک handlebar helper به عنوان متغیر در نظر گرفته می‌شوند.
به این ترتیب اطلاعات کنترلر جاری، به این المان‌های ورودی اطلاعات به صورت خودکار bind می‌شوند و برعکس. اگر کاربر مقادیر آن‌ها را تغییر دهد، تغییرات نهایی به صورت خودکار به خواص متناظری در کنترلر جاری منعکس خواهند شد (two-way data binding).
دو دکمه نیز تعریف شده‌اند که به اکشن‌های save و edit متصل هستند.
بنابراین نیاز به یک کنترلر جدید، به نام post داریم تا بتوان رفتار قالب post را کنترل کرد. برای این منظور فایل جدید Scripts\Controllers\post.js را با محتوای ذیل ایجاد کنید:

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

همچنین مدخل تعریف آن‌را نیز به فایل index.html اضافه نمائید (پس از تعاریف کنترلرهای موجود):

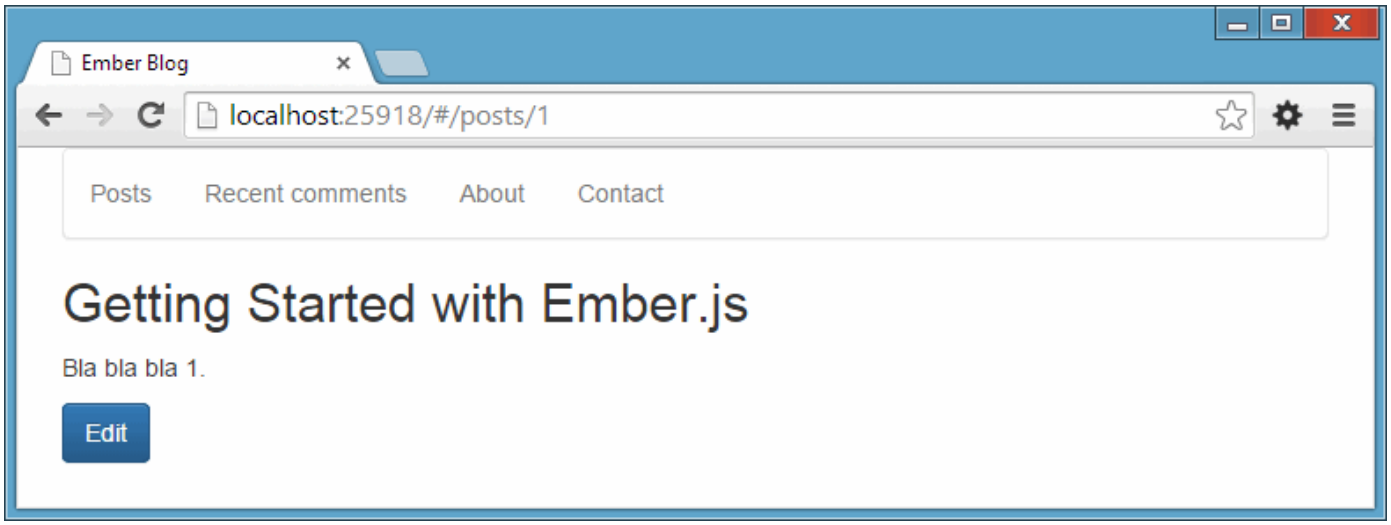
```
<script src="Scripts/Controllers/post.js" type="text/javascript"></script>
```

اگر به کدهای این کنترلر دقت کرده باشید، اینبار زیرکلاسی از ObjectController ایجاد شده‌است و نه Controller، مانند مثال‌های قبل . ObjectController تغییرات رخ داده بر روی خواص مدل را که توسط کنترلر در معرض دید قالب قرار داده‌است، به صورت خودکار به مدل مرتبط نیز منعکس می‌کند (Ember.ObjectController.extend)؛ اما Controller خیر (Ember.Controller.extend). در اینجا مدل کنترلر، تنها «یک» شیء است که بر اساس id آن انتخاب شده‌است. به همین جهت از ObjectController برای ارائه two-way data binding کمک گرفته شد.
در ember.js، یک قالب تنها با کنترلر خودش دارای تبادل اطلاعات است. اگر این کنترلر از نوع ObjectController باشد، تغییرات خاصیتی در یک قالب، ابتدا به کنترلر آن منعکس می‌شود و سپس این کنترلر، در صورت یافتن معادلی از این خاصیت در مدل، آن‌را به روز خواهد کرد. در حالت استفاده از Controller معمولی، صرفا تبادل اطلاعات بین قالب و کنترلر را شاهد خواهیم بود و نه بیشتر.

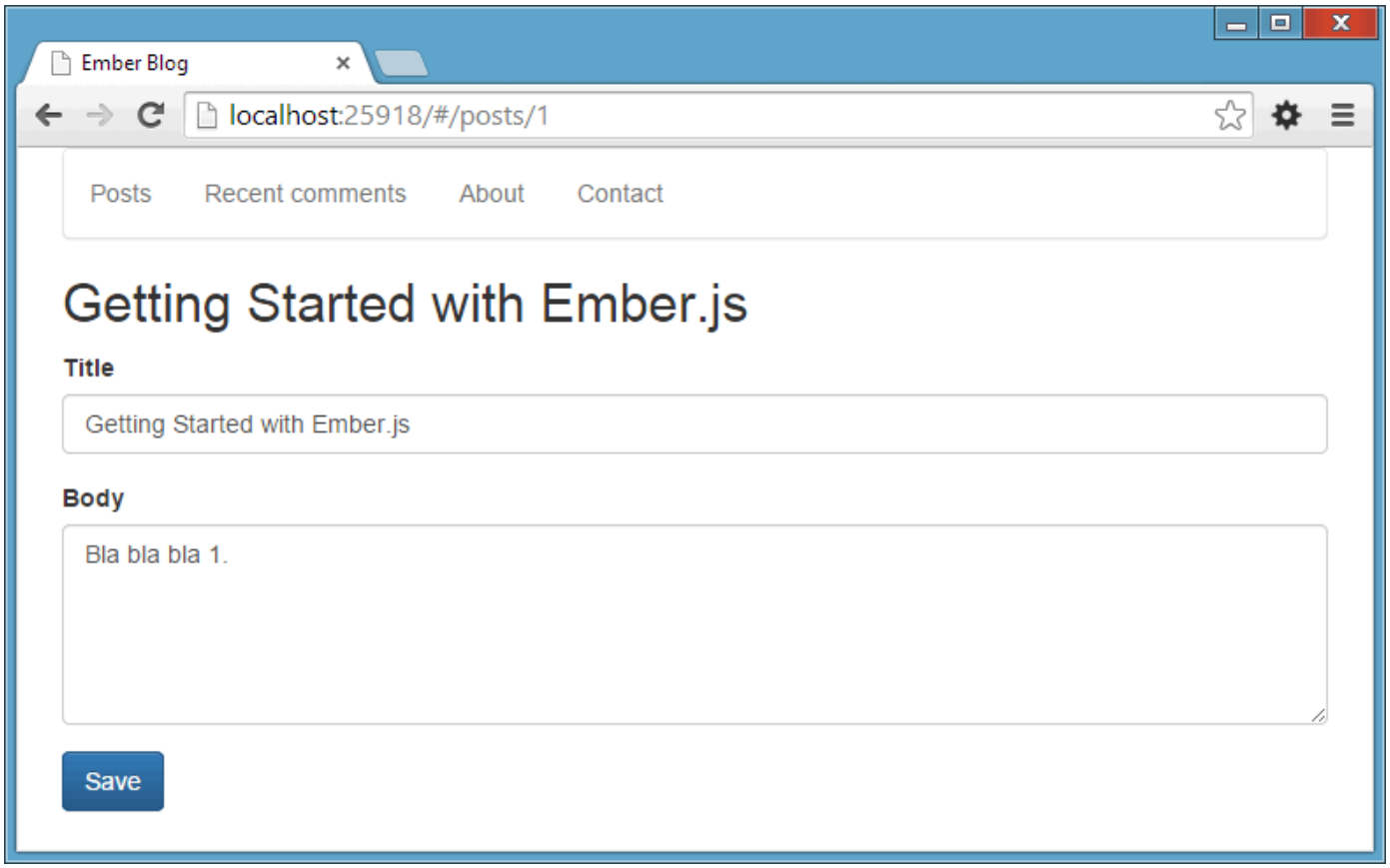
در ابتدای کار مقدار خاصیت isEditing مساوی false است. این مورد سبب می‌شود تا در بار اول بارگذاری اطلاعات یک مطلب

انتخابی، صرفا عنوان و محتوای مطلب نمایش داده شوند؛ به همراه یک دکمه‌ی edit. با کلیک بر روی دکمه‌ی edit، مطابق کدهای کنترلر فوق، تنها خاصیت isEditing به true تنظیم می‌شود و در این حالت، بدنه‌ی اصلی شرط if isEditing در قالب post، رندر خواهد شد.

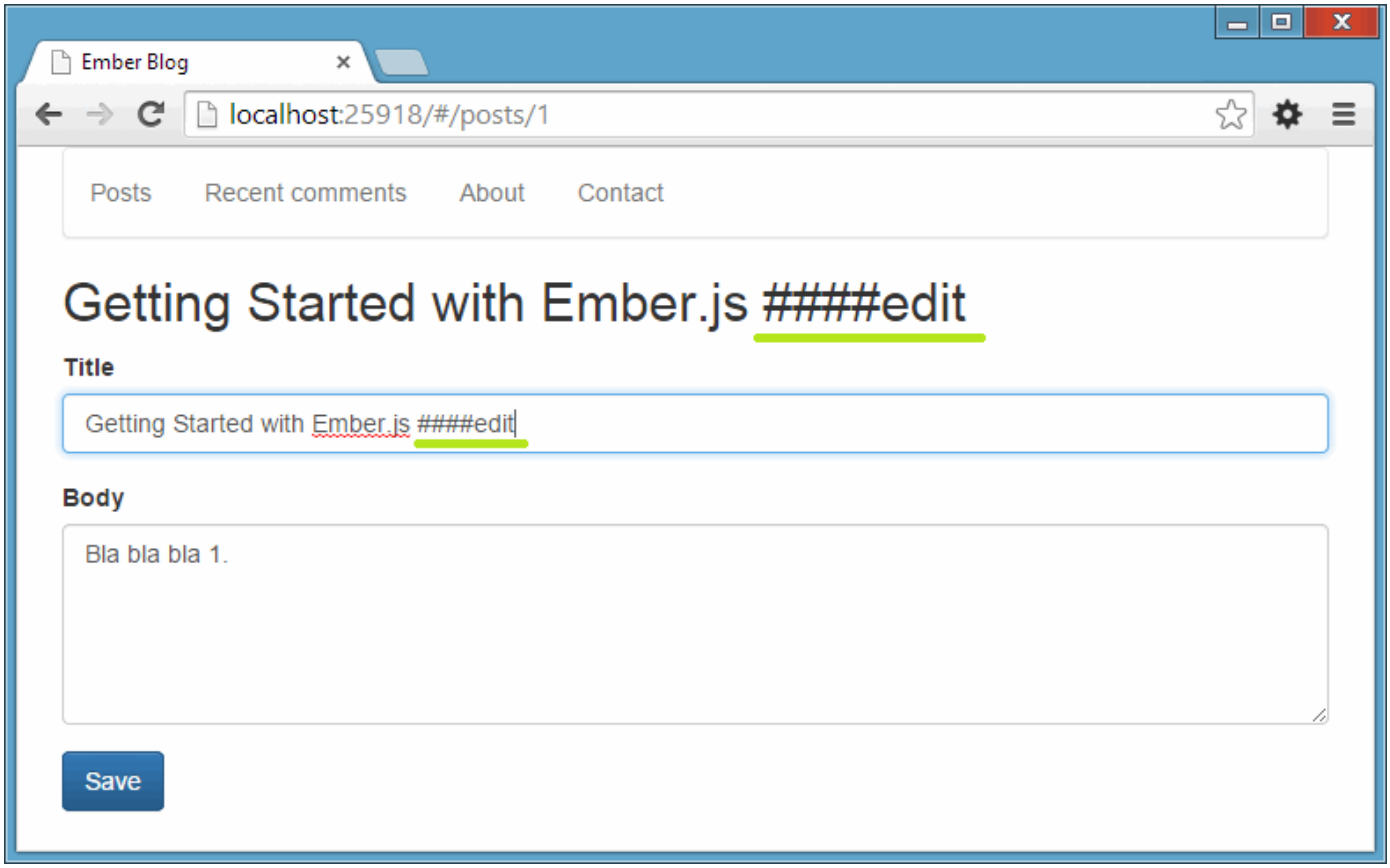
برای مثال در ابتدا مطلب شماره یک را انتخاب می‌کنیم:



با کلیک بر روی دکمه‌ی edit، فرم ویرایش ظاهر خواهد شد:



نکته‌ی جالب آن، مقدار دهی خودکار المان‌های ویرایش اطلاعات است. در این حالت سعی کنید، عنوان مطلب جاری را اندکی ویرایش کنید:



با ویرایش عنوان، می‌توان بلافاصله مقدار تغییر یافته را در برچسب عنوان مطلب نیز مشاهده کرد. این مورد دقیقا مفهوم two-way data binding و اتصال مقادیر value هر کدام از handlebar helperهای ویژه‌ی input و textarea را به عناصر مدل ارائه شده توسط کنترلر post، بیان می‌کند.
در این حالت در کدهای متد save، تنها کافی است که خاصیت isEditing را به false تنظیم کنیم. زیرا کلیه مقادیر ویرایش شده توسط کاربر، در همان لحظه در برنامه منتشر شده‌اند و نیاز به کار بیشتری برای اعمال تغییرات نیست.

اضافه کردن دکمه‌ی مرتب سازی بر اساس عناوین، در صفحه‌ی اول سایت

Ember.ObjectController.extend برای data bindg یک شیء کاربرد دارد. اگر قصد داشته باشیم با آرایه‌ای از اشیاء کار کنیم می‌توان از ArrayController استفاده کرد. فرض کنید در صفحه‌ی اول سایت می‌خواهیم امکان مرتب سازی مطالب را بر اساس عنوان آن‌ها اضافه کنیم. فایل Scripts\Templates\posts.hbs را گشوده و لینک Sort by title را به انتهای آن اضافه کنید:

```
<h2>Emeber.js blog</h2>
<ul>
    {{#each post in model}}
    <li>{{#link-to 'post' post.id}}{{post.title}}{{/link-to}}</li>
    {{/each}}
</ul>

<a href="#" class="btn btn-primary" {{action 'sortByTitle'}}>Sort by title</a>
```

در اینجا چون قصد تغییر رفتار قالب posts را توسط اکشن جدید sortByTitle داریم، نیاز است کنترلر متناظر با آن‌را نیز اضافه کنیم. برای این منظور فایل جدید Scripts\Controllers\posts.js را به پوشه‌ی کنترلرها اضافه کنید؛ با محتوای ذیل:

```
Blogger.PostsController = Ember.ArrayController.extend({
    sortProperties: ['id'],// مقادیر پیش فرض مرتب سازی
    sortAscending: false,
    actions: {
        sortByTitle: function () {
            this.set('sortProperties', ['title']);
            this.set('sortAscending', !this.get('sortAscending'));
        }
    }
});
```

sortProperties جزو خواص کلاس پایه ArrayController است. اگر مانند سطر اول به صورت مستقیم مقدار دهی شود، خاصیت یا خواص پیش فرض مرتب سازی را مشخص می‌کند. اگر مانند اکشن sortByTitle توسط متد set مقدار دهی شود، امکان مرتب سازی تعاملی و با فرمان کاربر را فراهم خواهد کرد.

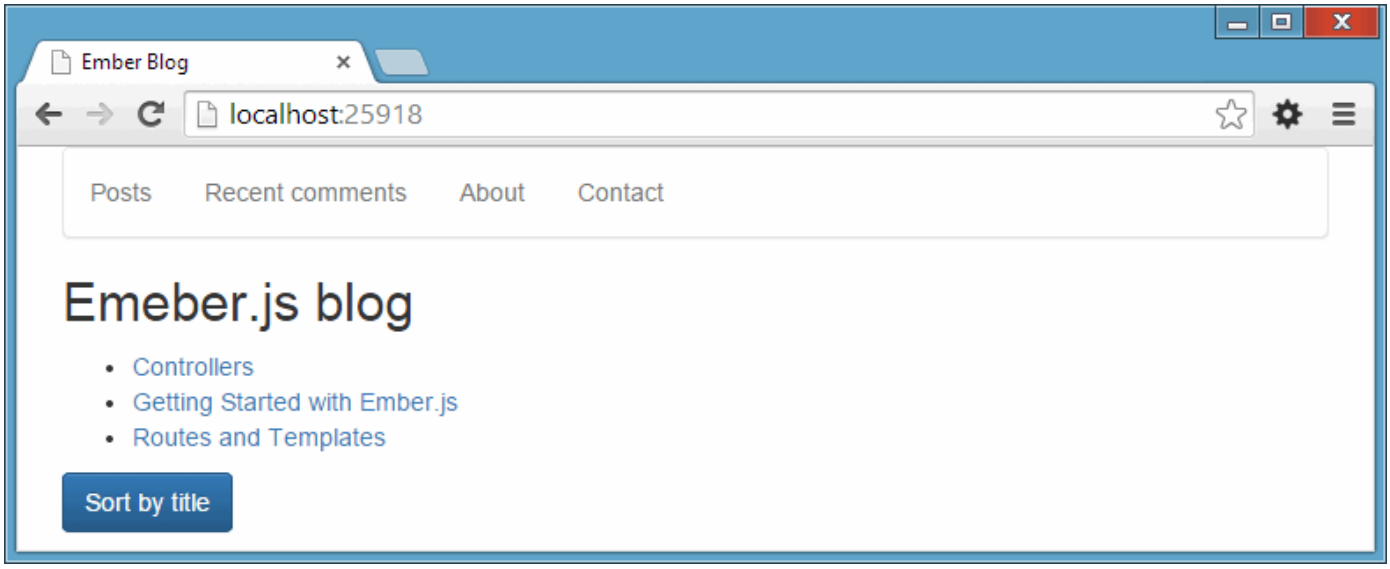
در ادامه، تعریف مدخل این کنترلر جدید را نیز باید به فایل index.html، اضافه کرد:

```
<script src="Scripts/Controllers/posts.js" type="text/javascript"></script>
```

اگر برنامه را در این حالت اجرا کرده و بر روی دکمه‌ی Sort by title کلیک کنید، اتفاقی رخ نمی‌دهد. علت اینجا است که ArrayController خروجی تغییر یافته خودش را توسط خاصیتی به نام arrangedContent در اختیار قالب خود قرار می‌دهد. بنابراین نیاز است فایل قالب Scripts\Templates\posts.hbs را به نحو ذیل ویرایش کرد:

```
<h2>Emeber.js blog</h2>
<ul>
    {{#each post in arrangedContent}}
    <li>{{#link-to 'post' post.id}}{{post.title}}{{/link-to}}</li>
    {{/each}}
</ul>

<a href="#" class="btn btn-primary" {{action 'sortByTitle'}}>Sort by title</a>
```

اینبار کلیک بر روی دکمه‌ی مرتب سازی بر اساس عناوین، هربار لیست موجود را به صورت صعودی و یا نزولی مرتب می‌کند.



یک نکته: حلقه‌ی ویژه‌ای به نام each

اگر قالب Scripts\Templates\posts.hbs را به نحو ذیل، با یک حلقه‌ی each ساده بازنویسی کنید:

```
<h2>Ember.js blog</h2>
<ul>
    {{#each}}
    <li>{{#link-to 'post' id}}{{title}}{{/link-to}}</li>
    {{/each}}
</ul>

<a href="#" class="btn btn-primary" {{action 'sortByTitle'}}>Sort by title</a>
```

هم در حالت نمایش معمولی و هم در حالت استفاده از ArrayController برای نمایش اطلاعات مرتب شده، بدون مشکل کار می‌کند و نیازی به تغییر نخواهد داشت.

کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

EmberJS03_03.zip

عنوان: **ساخت یک بلاگ ساده با Ember.js، قسمت چهارم**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۱۹ ۸:۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, jQuery, SPA, EmberJS

در قسمت قبل ، اطلاعات نمایش داده شده، از یک سری آرایه ثابت جاوا اسکریپتی تامین شدند. در یک برنامه‌ی واقعی نیاز است داده‌ها را یا از local storage HTML 5 تامین کرد و یا از سرور به کمک Ajax. برای اینگونه اعمال، ember.js به همراه افزونه‌ای است به نام Ember Data که جزئیات کار با آن‌را در این قسمت بررسی خواهیم کرد.

استفاده از Ember Data با Local Storage

برای کار با local storage HTML 5 نیاز به Ember Data Local Storage Adapter نیز هست که در قسمت اول این سری، آدرس دریافت آن معرفی شد. این فایل‌ها نیز در پوشه‌ی Scripts\Libs برنامه کپی خواهند شد.
در ادامه به فایل Scripts\App\store.js که در قسمت قبل جهت تعریف دو آرایه ثابت مطالب و نظرات اضافه شد، مراجعه کرده و محتوای فعلی آن‌را با کدهای زیر جایگزین کنید:

```
Blogger.ApplicationSerializer = DS.LSSerializer.extend();
Blogger.ApplicationAdapter = DS.LSAdapter.extend();
```

این تعاریف سبب خواهند شد تا Ember Data از Local Storage Adapter استفاده کند.
در ادامه با توجه به حذف دو آرایه‌ی posts و comments که پیشتر در فایل store.js تعریف شده بودند، نیاز است مدل‌های متناظری را جهت تعریف خواص آن‌ها، به برنامه اضافه کنیم. این‌کار را با افزودن دو فایل جدید comment.js و post.js به پوشه‌ی Scripts\Models انجام خواهیم داد.
محتوای فایل Scripts\Models\post.js :

```
Blogger.Post = DS.Model.extend({
  title: DS.attr(),
  body: DS.attr()
});
```

محتوای فایل Scripts\Models\comment.js :

```
Blogger.Comment = DS.Model.extend({
  text: DS.attr()
});
```

سپس مداخل تعریف آن‌ها را به فایل index.html نیز اضافه خواهیم کرد:

```
<script src="Scripts/Models/post.js" type="text/javascript"></script>
<script src="Scripts/Models/comment.js" type="text/javascript"></script>
```

برای تعاریف مدل‌ها در Ember data مرسوم است که نام مدل‌ها، اسامی جمع نباشند. سپس با ایجاد وهله‌ای از DS.Model.extend یک مدل ember data را تعریف خواهیم کرد. در این مدل، خواص هر شیء را مشخص کرده و مقدار آن‌ها همیشه DS.attr() خواهد بود. این نکته را در دو مدل Post و Comment مشاهده می‌کنید. اگر دقت کنید به هر دو مدل، خاصیت id اضافه نشده‌است. این خاصیت به صورت خودکار توسط Ember data تنظیم می‌شود.

اکنون نیاز است برنامه را جهت استفاده از این مدل‌های جدید به روز کرد. برای این منظور فایل Scripts\Routes\posts.js را گشوده و مدل آن‌را به نحو ذیل ویرایش کنید:

```
Blogger.PostsRoute = Ember.Route.extend({
    //controllerName: 'posts', // مقدار پیش فرض است و نیازی به ذکر آن نیست
```

```
    //renderTemplare: function () {
    //    this.render('posts'); // مقدار پیش فرض است و نیازی به ذکر آن نیست
    //},
    model: function () {
        return this.store.find('post');
    }
});
```

در اینجا this.store معادل data store برنامه است که مطابق تنظیمات برنامه، همان ember data می‌باشد. سپس متد find را به همراه نام مدل، به صورت رشته‌ای در اینجا مشخص می‌کنیم.

به همین ترتیب فایل Scripts\Routes\recent-comments.js را نیز جهت استفاده از data store ویرایش خواهیم کرد:

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

و فایل Scripts\Routes\post.js که در آن منطق یافتن یک مطلب بر اساس آدرس مختص به آن قرار دارد، به صورت ذیل بازنویسی می‌شود:

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

اگر متد find بدون پارامتر ذکر شود، به معنای بازگشت تمامی عناصر موجود در آن مدل خواهد بود و اگر پارامتر دوم آن مانند این مثال تنظیم شود، تنها همان وهله‌ی درخواستی را بازگشت می‌دهد.

افزودن امکان ثبت یک مطلب جدید

تا اینجا اگر برنامه را اجرا کنید، برنامه بدون خطا بارگذاری خواهد شد اما فعلا رکوردی را برای نمایش ندارد. در ادامه، برنامه را جهت افزودن مطالب جدید توسعه خواهیم داد. برای اینکار ابتدا به فایل Scripts\App\router.js مراجعه کرده و سپس مسیریابی جدید new-post را تعریف خواهیم کرد:

```
Blogger.Router.map(function () {
    this.resource('posts', { path: '/' });
    this.resource('about');
    this.resource('contact', function () {
        this.resource('email');
        this.resource('phone');
    });
    this.resource('recent-comments');
    this.resource('post', { path: 'posts/:post_id' });
    this.resource('new-post');
});
```

اکنون در صفحه‌ی اول سایت، توسط قالب Scripts\Templates\posts.hbs، دکمه‌ای را جهت ایجاد یک مطلب جدید اضافه خواهیم کرد:

```
<h2>Ember.js blog</h2>
<ul>
    {{#each post in arrangedContent}}
    <li>{{#link-to 'post' post.id}}{{post.title}}{{/link-to}}</li>
    {{/each}}
</ul>

<a href="#" class="btn btn-primary" {{action 'sortByTitle' }}>Sort by title</a>
{{#link-to 'new-post' classNames="btn btn-success"}}New Post{{/link-to}}
```

در اینجا دکمه‌ی New Post به مسیریابی جدید new-post اشاره می‌کند.
برای تعریف عناصر نمایشی این مسیریابی، فایل جدید قالب Scripts\Templates\new-post.hbs را با محتوای زیر اضافه کنید:

```
<h1>New post</h1>
<form>
  <div class="form-group">
    <label for="title">Title</label>
    {{input value=title id="title" class="form-control"}}
  </div>

  <div class="form-group">
    <label for="body">Body</label>
    {{textarea value=body id="body" class="form-control" rows="5"}}
  </div>


  <button class="btn btn-primary" {{action 'save'}}>Save</button>
</form>
```

با نمونه‌ی این فرم در قسمت قبل در حین ویرایش یک مطلب، آشنا شدیم. دو المان دریافت اطلاعات در آن قرار دارند که هر کدام به خواص مدل برنامه bind شده‌اند. همچنین یک دکمه‌ی save، با اکشنی به همین نام در اینجا تعریف شده‌است.
پس از آن نیاز است نام فایل قالب new-post را به template loader برنامه در فایل index.html اضافه کرد:

```
<script type="text/javascript">
    EmberHandlebarsLoader.loadTemplates([
       'posts', 'about', 'application', 'contact', 'email', 'phone',
       'recent-comments', 'post', 'new-post'
    ]);
</script>
```

برای مدیریت دکمه‌ی save این قالب جدید نیاز است کنترلر جدیدی را در فایل جدید Scripts\Controllers\new-post.js تعریف کنیم؛ با این محتوا:

```
Blogger.NewPostController = Ember.Controller.extend({
    actions: {
        save: function () {
            var newPost = this.store.createRecord('post', {
                title: this.get('title'),
                body: this.get('body')
            });
            newPost.save();
            this.transitionToRoute('posts');
        }
    }
});
```

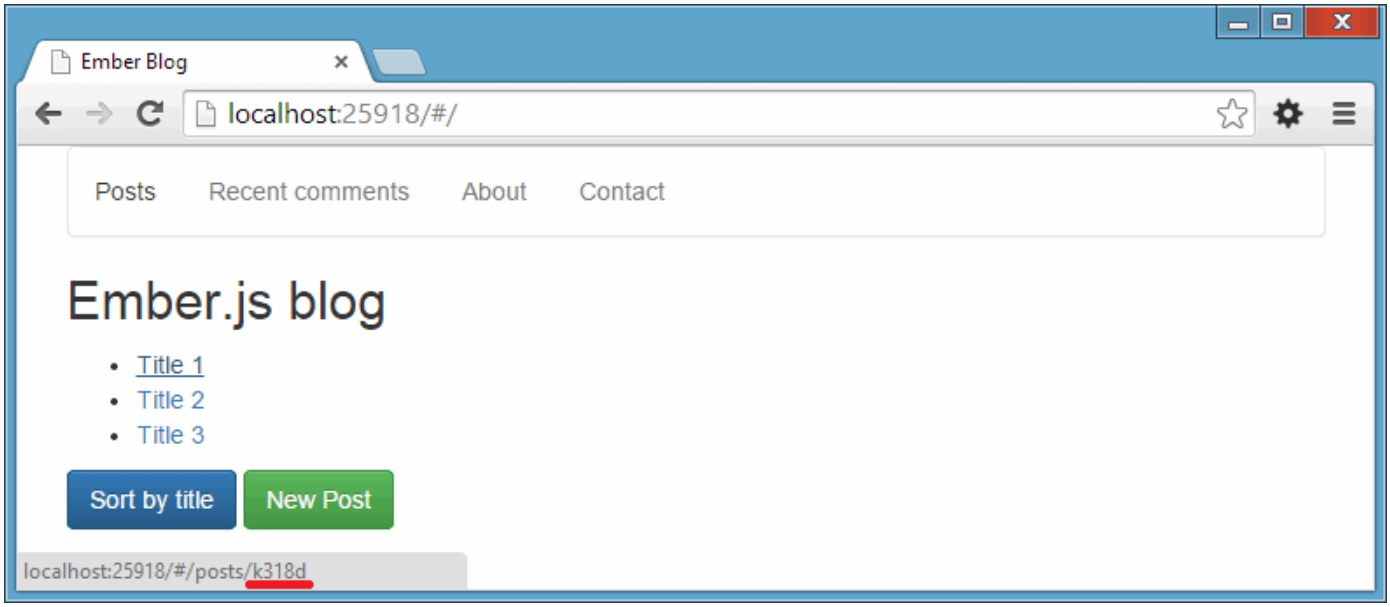
به همراه افزودن مدخلی از آن به فایل index.html برنامه:

```
<script src="Scripts/Controllers/new-post.js" type="text/javascript"></script>
```

در اینجا کنترلر جدید NewPostController را مشاهده می‌کنید. از این جهت که برای دسترسی به خواص مدل تغییر کرده، از متد this.get استفاده شده‌است، نیازی نیست حتما از یک ObjectController مانند قسمت قبل استفاده کرد و Controller معمولی نیز برای اینکار کافی است.
آرگومان اول this.store.createRecord نام مدل است و آرگومان دوم آن، وهله‌ای که قرار است به آن اضافه شود. همچنین باید دقت داشت که برای تنظیم یک خاصیت، از متد this.set و برای دریافت مقدار یک خاصیت تغییر کرده از this.get به همراه نام خاصیت مورد نظر استفاده می‌شود و نباید مستقیما برای مثال از this.title استفاده کرد.
this.store.createRecord صرفا یک شیء جدید (ember data object) را ایجاد می‌کند. برای ذخیره سازی نهایی آن باید متد save آن‌را فراخوانی کرد (پیاده سازی الگوی active record است). به این ترتیب این شیء در local storage ذخیره خواهد شد.
پس از ذخیره‌ی مطلب جدید، از متد this.transitionToRoute استفاده شده‌است. این متد، برنامه را به صورت خودکار به

صفحه‌ی متناظر با مسیریابی posts هدایت می‌کند.

اکنون برنامه را اجرا کنید. بر روی دکمه‌ی سبز رنگ new post در صفحه‌ی اول کلیک کرده و یک مطلب جدید را تعریف کنید. بلافاصله عنوان و لینک متناظر با این مطلب را در صفحه‌ی اول سایت مشاهده خواهید کرد.
همچنین اگر برنامه را مجددا بارگذاری کنید، این مطالب هنوز قابل مشاهده هستند؛ زیرا در local storage مرورگر ذخیره شده‌اند.

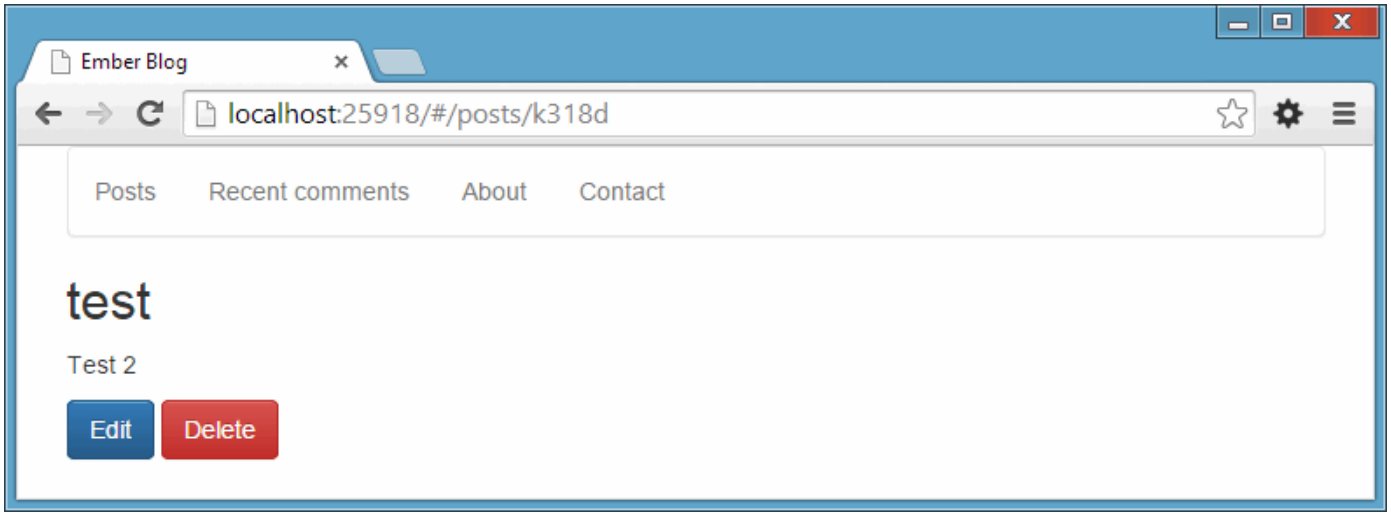


در اینجا اگر به لینک‌های تولید شده دقت کنید، id آن‌ها عددی نیست. این روشی است که local storage با آن کار می‌کند.

افزودن امکان حذف یک مطلب به سایت

برای حذف یک مطلب، دکمه‌ی حذف را به انتهای قالب Scripts\Templates\post.hbs اضافه خواهیم کرد:

```
<h2>{{title}}</h2>
{{#if isEditing}}
<form>
    <div class="form-group">
        <label for="title">Title</label>
        {{input value=title id="title" class="form-control"}}
    </div>
    <div class="form-group">
        <label for="body">Body</label>
        {{textarea value=body id="body" class="form-control" rows="5"}}
    </div>
    <button class="btn btn-primary" {{action 'save' }}>Save</button>
</form>
{{else}}
<p>{{body}}</p>
<button class="btn btn-primary" {{action 'edit' }}>Edit</button>
<button class="btn btn-danger" {{action 'delete' }}>Delete</button>
{{/if}}
```



سپس کنترلر Scripts\Controllers\post.js را جهت مدیریت اکشن جدید delete به نحو ذیل تکمیل می‌کنیم:

```
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?')) {
                this.get('model').destroyRecord();
                this.transitionToRoute('posts');
            }
        }
    }
});
```

متد destroyRecord، مدل انتخابی را هم از حافظه و هم از data store حذف می‌کند. سپس کاربر را به صفحه‌ی اصلی سایت هدایت خواهیم کرد.
متد save نیز در اینجا بهبود یافته‌است. ابتدا مدل جاری دریافت شده و سپس متد save بر روی آن فراخوانی می‌شود. به این ترتیب اطلاعات از حافظه به local storage نیز منتقل خواهند شد.

ثبت و نمایش نظرات به همراه تنظیمات روابط اشیاء در Ember Data

در ادامه قصد داریم امکان افزودن نظرات را به مطالب، به همراه نمایش آن‌ها در ذیل هر مطلب، پیاده سازی کنیم. برای اینکار نیاز است رابطه‌ی بین یک مطلب و نظرات مرتبط با آن‌را در مدل ember data مشخص کنیم. به همین جهت فایل Scripts\Models\post.js را گشوده و تغییرات ذیل را به آن اعمال کنید:

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

در اینجا خاصیت جدیدی به نام comments به مدل مطلب اضافه شده‌است و توسط آن می‌توان به تمامی نظرات یک مطلب دسترسی یافت؛ تعریف رابطه‌ی یک به چند، به کمک متد DS.hasMany که پارامتر اول آن نام مدل مرتبط است. تعریف async: true

برای کار با local storage اجباری است و در نگارش‌های آتی ember data حالت پیش فرض خواهد بود.
همچنین نیاز است یک سر دیگر رابطه را نیز مشخص کرد. برای این منظور فایل Scripts\Models\comment.js را گشوده و به نحو ذیل تکمیل کنید:

```
Blogger.Comment = DS.Model.extend({
    text: DS.attr(),
    post: DS.belongsTo('post', { async: true })
});
```

در اینجا خاصیت جدید post به مدل نظر اضافه شده‌است و مقدار آن از طریق متد DS.belongsTo که مدل post را به یک نظر، مرتبط می‌کند، تامین خواهد شد. بنابراین در این حالت اگر به شیء comment مراجعه کنیم، خاصیت جدید post.id آن، به id مطلب متناظر اشاره می‌کند.

در ادامه نیاز است بتوان تعدادی نظر را ثبت کرد. به همین جهت با تعریف مسیریابی آن شروع می‌کنیم. این مسیریابی تعریف شده در فایل Scripts\App\router.js نیز باید تو در تو باشد؛ زیرا قسمت ثبت نظر (new-comment) دقیقا داخل همان صفحه‌ی نمایش یک مطلب ظاهر می‌شود:

```
Blogger.Router.map(function () {
    this.resource('posts', { path: '/' });
    this.resource('about');
    this.resource('contact', function () {
        this.resource('email');
        this.resource('phone');
    });
    this.resource('recent-comments');
    this.resource('post', { path: 'posts/:post_id' }, function () {
        this.resource('new-comment');
    });
    this.resource('new-post');
});
```

لینک آن‌را نیز به انتهای فایل Scripts\Templates\post.hbs اضافه می‌کنیم. از این جهت که این لینک به مدل جاری اشاره می‌کند، با استفاده از متغیر this، مدل جاری را به عنوان مدل مورد استفاده مشخص خواهیم کرد:

```
<h2>{{title}}</h2>
{{#if isEditing}}
<form>
    <div class="form-group">
        <label for="title">Title</label>
        {{input value=title id="title" class="form-control"}}
    </div>
    <div class="form-group">
        <label for="body">Body</label>
        {{textarea value=body id="body" class="form-control" rows="5"}}
    </div>
    <button class="btn btn-primary" {{action 'save' }}>Save</button>
</form>
{{else}}
<p>{{body}}</p>
<button class="btn btn-primary" {{action 'edit' }}>Edit</button>
<button class="btn btn-danger" {{action 'delete' }}>Delete</button>
{{/if}}

<h2>Comments</h2>
{{#each comment in comments}}
<p>
    {{comment.text}}
</p>
{{/each}}

<p>{{#link-to 'new-comment' this class="btn btn-success"}}New comment{{/link-to}}</p>
{{outlet}}
```

پس از تکمیل روابط مدل‌ها، قالب Scripts\Templates\post.hbs را جهت استفاده از این خواص به روز خواهیم کرد. در تغییرات جدید، قسمت <h2>Comments</h2> به انتهای صفحه اضافه شده‌است. سپس حلقه‌ای بر روی خاصیت جدید comments تشکیل

شده و مقدار خاصیت text هر آیتم نمایش داده می‌شود.
در انتهای قالب نیز یک {{outlet}} اضافه شده‌است. کار آن نمایش قالب ارسال یک نظر جدید، پس از کلیک بر روی لینک New Comment می‌باشد. این قالب را با افزودن فایل Scripts\Templates\new-comment.hbs با محتوای ذیل ایجاد خواهیم کرد:

```
<h2>New comment</h2>

<form>
  <div class="form-group">
    <label for="text">Your thoughts:</label>
    {{textarea value=text id="text" class="form-control" rows="5"}}
  </div>

  <button class="btn btn-primary" {{action "save"}}>Add your comment</button>
</form>
```

سپس نام این قالب را به template loader فایل index.html نیز اضافه می‌کنیم؛ تا در ابتدای بارگذاری برنامه شناسایی شده و استفاده شود:

```
<script type="text/javascript">
    EmberHandlebarsLoader.loadTemplates([
       'posts', 'about', 'application', 'contact', 'email', 'phone',
       'recent-comments', 'post', 'new-post', 'new-comment'
    ]);
</script>
```

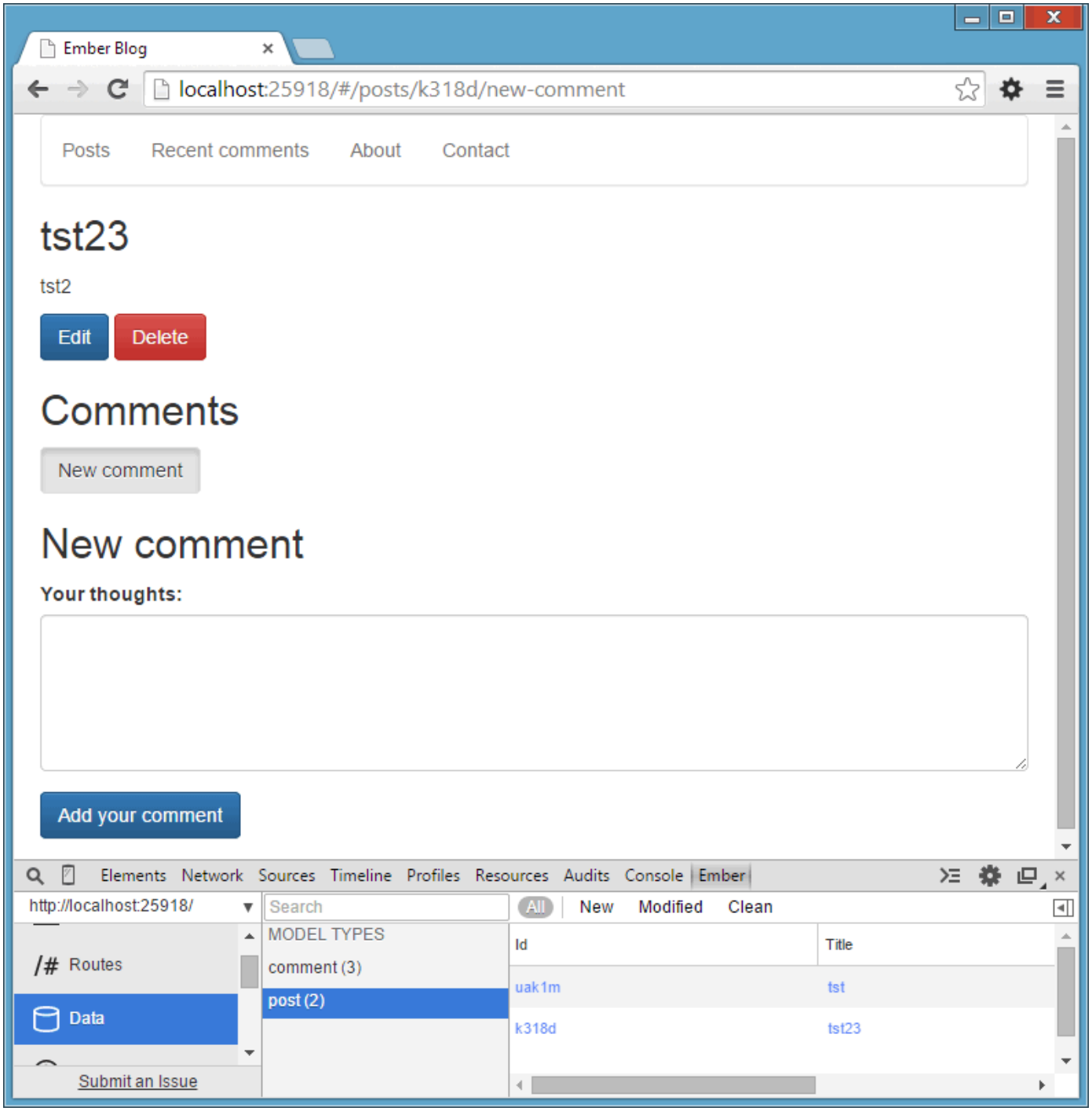
این قالب به خاصیت text یک comment متصل بوده و همچنین اکشن جدیدی به نام save دارد. بنابراین برای مدیریت اکشن save، نیاز به کنترلری متناظر خواهد بود. به همین جهت فایل جدید Scripts\Controllers\new-comment.js را با محتوای ذیل ایجاد کنید:

```
Blogger.NewCommentController = Ember.ObjectController.extend({
    needs: ['post'],
    actions: {
        save: function () {
            var comment = this.store.createRecord('comment', {
                text: this.get('text')
            });
            comment.save();

            var post = this.get('controllers.post.model');
            post.get('comments').pushObject(comment);
            post.save();

            this.transitionToRoute('post', post.id);
        }
    }
});
```

و مدخل تعریف آن‌را نیز به صفحه‌ی index.html اضافه می‌کنیم:

```
<script src="Scripts/Controllers/new-comment.js" type="text/javascript"></script>
```

قسمت ذخیره سازی comment جدید با ذخیره سازی یک post جدید که پیشتر بررسی کردیم، تفاوتی ندارد. از متد this.store.createRecord جهت معرفی وهله‌ای جدید از comment استفاده و سپس متد save آن، برای ثبت نهایی فراخوانی شده‌است.
در ادامه باید این نظر جدید را به post متناظر با آن مرتبط کنیم. برای اینکار نیاز است تا به مدل کنترلر post دسترسی داشته باشیم. به همین جهت خاصیت needs را به تعاریف کنترلر جاری به همراه نام کنترلر مورد نیاز، اضافه کرده‌ایم. به این ترتیب می‌توان توسط متد this.get و پارامتر controllers.post.model در کنترلر NewComment به اطلاعات کنترلر post دسترسی یافت. سپس خاصیت comments شیء post جاری را یافته و مقدار آن‌را به comment جدیدی که ثبت کردیم، تنظیم می‌کنیم. در ادامه با فراخوانی متد save، کار تنظیم ارتباطات یک مطلب و نظرهای جدید آن به پایان می‌رسد.
در آخر با فراخوانی متد transitionToRoute به مطلبی که نظر جدیدی برای آن ارسال شده‌است باز می‌گردیم.



همانطور که در این تصویر نیز مشاهده می‌کنید، اطلاعات ذخیره شده در local storage را توسط افزونه‌ی Ember Inspector نیز می‌توان مشاهده کرد.

افزودن دکمه‌ی حذف به لیست نظرات ارسالی

برای افزودن دکمه‌ی حذف، به قالب Scripts\Templates\post.hbs مراجعه کرده و قسمتی را که لیست نظرات را نمایش می‌دهد، به نحو ذیل تکمیل می‌کنیم:

```
{{#each comment in comments}}
```

```
<p>
    {{comment.text}}
    <button class="btn btn-xs btn-danger" {{action 'delete' }}>delete</button>
</p>
{{/each}}
```

همچنین برای مدیریت اکشن جدید delete، کنترلر جدید comment را در فایل Scripts\Controllers\comment.js اضافه خواهیم کرد.

```
Blogger.CommentController = Ember.ObjectController.extend({
    needs: ['post'],
    actions: {
        delete: function () {
            if (confirm('Do you want to delete this comment?')) {
                var comment = this.get('model');
                comment.deleteRecord();
                comment.save();

                var post = this.get('controllers.post.model');
                post.get('comments').removeObject(comment);
                post.save();
            }
        }
    }
});
```

به همراه تعریف مدخل آن در فایل index.html :

```
<script src="Scripts/Controllers/comment.js" type="text/javascript"></script>
```

در این حالت اگر برنامه را اجرا کنید، پیام «Do you want to delete this **post**» را مشاهده خواهید کرد بجای پیام «Do you want to delete this **comment**». علت اینجا است که قالب post به صورت پیش فرض به کنترلر post متصل است و نه کنترلر comment. برای رفع این مشکل تنها کافی است از itemController به نحو ذیل استفاده کنیم:

```
{{#each comment in comments  itemController="comment"}}
<p>
    {{comment.text}}
    <button class="btn btn-xs btn-danger" {{action 'delete' }}>delete</button>
</p>
{{/each}}
```

به این ترتیب اکشن delete به کنترلر comment ارسال خواهد شد و نه کنترلر پیش فرض post جاری.
در کنترلر Comment روش دیگری را برای حذف یک رکورد مشاهده می‌کنید. می‌توان ابتدا متد deleteRecord را بر روی مدل فراخوانی کرد و سپس آن‌را save نمود تا نهایی شود. همچنین در اینجا نیاز است نظر حذف شده را از سر دیگر رابطه نیز حذف کرد. روش دسترسی به post جاری در این حالت، همانند توضیحات NewCommentController است که پیشتر بحث شد.

کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

EmberJS03_04.zip

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

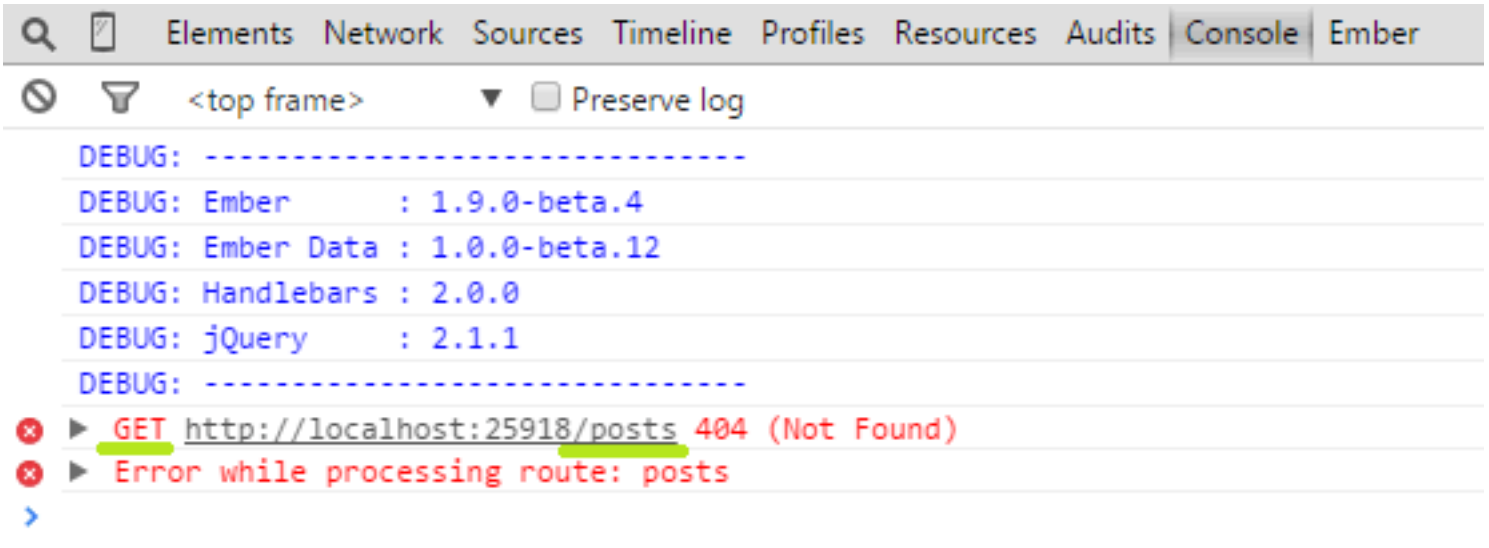
Ember data به صورت پیش فرض و در پشت صحنه با استفاده از Ajax برای کار با یک REST Web Service طراحی شده‌است و کلیه تبادلات آن نیز با فرمت JSON انجام می‌شود. بنابراین تمام کدهای سمت کاربر قسمت قبل نیز در این حالت کار خواهند کرد. تنها کاری که باید انجام شود، حذف تنظیمات ابتدایی آن برای کار با HTML 5 local 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