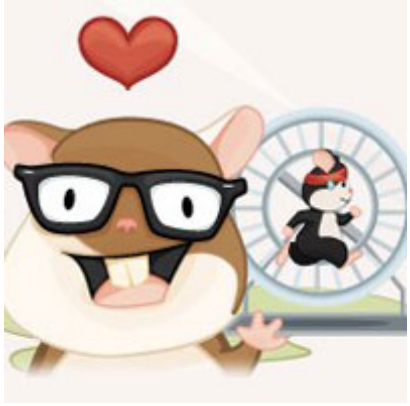

Getting Into Ember.js

Rey Bango on Mar 14th 2013 with 32 Comments

Tutorial Details

-
- **Difficulty:** Intermediate
- **Completion Time:** 15 Minutes

There are a lot of JavaScript libraries available, and most are really good at providing the traditional DOM-centric interactions that your typical websites need. But when it's time to build a manageable code base for a single-page app, that's where a whole suite of new frameworks come in to smooth things out.

> The old saying is true: "Use the best tool for the task."

It's not that traditional libraries like jQuery can't help you build desktop-like experiences, it's just not the use-case for it and is missing things like data-binding, event routing and state management. Sure, you can probably cobble together a bunch of plugins to achieve some of that functionality, but starting with a framework that's been specifically built from the ground up to tackle these specific problems, in my opinion, makes more sense. The old saying is true: "Use the best tool for the

task."

> I recently did an interview with the Ember.js team; it was motivated by my desire to get to know what I've come to call "the new hotness": Ember.js.

Ember fits the bill for what I've described above, and does so in a fashion that reminds of a lot of how jQuery allows developers to get up and running quickly. The team has purposely taken steps to abstract a lot of the complexities inherent in designing and building Model/View/Controller based applications using years of expertise and knowledge gained from building large-scale apps.

What I'd like to do is help get you up to speed with Ember.js, via a multi-part article series which will gradually introduce you to the concepts of the framework. We'll start off with the usual intro (which happens to be this post), and then gradually ramp up to building a full application. The great part is that this will also help me reinforce the concepts that I've already learned, and maybe pick up some new techniques along the way! I'll do my best to have the Ember.js team review this material for accuracy and perhaps even contribute some nuggets to it.

Before we continue, a heads up: Ember.js does a lot of magic for you. There are times when you'll look at the code and say "Huh? How'd it do that?" I've been there and I'll do my best to distill things, but I'm not going to dive into the bowels of Ember's framework code. Instead, I'll discuss how you can leverage its tools and API to build your app.

So let's kick this off.

Core Concepts

> Ember.js is not a framework for building traditional websites.

The first thing to keep in mind is that Ember.js is not a framework for building traditional websites. Libraries like jQuery and MooTools are a great fit for that. If you're considering Ember.js, then the assumption is that you're looking to build

desktop–like experiences – especially scalable ones. In fact, the slogan for the framework is "a framework for developing ambitious web applications" which tells you it's clearly not your daddy's JavaScript library.

I mentioned previously that Ember leverages the MVC pattern for promoting proper code management and organization. If you've never done MVC–based development, you should definitely read up on it. Nettuts+ has a great article on the topic here. For those of you familiar with the concepts, you should feel at home. The one thing I've heard consistently is that making the shift from Backbone to Ember.js is actually easy because Ember does a lot of the heavy lifting for you, while still maintaining the code organization patterns that those developers are accustomed to.

Ember also relies on client–side templates... a **LOT**. It uses the Handlebars templating library which provides expressions that allow you to create dynamic HTML–based templates. An Ember developer can bind data to these embeddable expressions and dynamically change the display of their app on the fly. For example, I can create a template that can receive an array of people and display them in an unordered list:

```
<ul>
 {{#each people}}
   <li>Hello, {{name}}!</li>
 {{/each}}
</ul>
```

Notice the "#each" expression that works as a loop directive, enumerating over each element of the "people" array and replacing the "{{name}}" expression with an actual value. It's important to note that the double brackets are tokens used by Handlebars to identify expressions. This is a small example, and we'll dive into more detail later.

Handlebars is an incredibly powerful client–side templating engine and I would recommend reviewing not only the Ember guides, but the Handlebars website itself to get a full grasp of the options available. You'll be using it quite a bit.

Setting up Ember

Ember.js relies on additional libraries, so you'll need to go grab a copy of jQuery and Handlebars. But, wait, didn't I say that jQuery and Ember play in different spaces? Well, yeah I did, but here's the thing: the Ember team is all about not reinventing the wheel. They chose jQuery to do what it does best: work with the DOM. And that's a good thing, since jQuery is really good at that. It's also why they went with Handlebars, which is an excellent templating library that happened to be written by Yehuda Katz, who is an Ember core team member.

The easiest way to get the files you need is to go to the Ember.js Github repo and pull down the Starter Kit. It's a boilerplate for you to start off with. At the time of this writing, it contains:

- Ember 1.0 RC1
- Handlerbars 1.0 RC3
- jQuery 1.9.1

There's also a basic html template that is coded to include all of the associated libraries (jQuery, Ember, etc.) and along with an example of a Handlebars template and "app.js", which includes code for kicking off a basic Ember app.

```
<script src="js/libs/jquery-1.9.1.js"></script>
<script src="js/libs/handlebars-1.0.0-rc.3.js"></script>
<script src="js/libs/ember-1.0.0-rc.1.js"></script>
<script src="js/app.js"></script>
```

Just note that app.js isn't part of the framework. It's a plain ole JavaScript file; you can name it anything you want. And, while we'll use it for the purposes of this tutorial series, down the road, you'll likely end up splitting your JavaScript into multiple files just like you would for any other site or app. Also, Ember doesn't expect a specific directory structure for your framework files.

When you look at the Starter Kit code, it may look like your typical website code. In some respects, you're right! Once we start organizing things, though, you'll see how building an Ember app is different.

The Lay of Ember Land

Before you start hacking at code, it's important to understand how Ember.js works and that you grok the moving parts that make up an Ember app. Let's take a look at those parts and how they relate to each other.

Templates

Templates are a key part of defining your user interface. As I mentioned previously, Handlebars is the client–side library used in Ember and the expressions provided by the library are used extensively when creating the UI for your application. Here's a simple example:

```
<script type="text/x-handlebars">
     <h2><strong>{{firstName}} {{lastName}}</strong></h2>
</script>
```

Notice that the expressions are mixed into your HTML markup and, via Ember, will dynamically change the content displayed on the page. In this case, the {{firstName}} and {{lastName}} placeholders will be replaced by data retrieved from the app.

Handlebars offers a lot of power, via a flexible API. It will be important for you to understand what it offers.

Routing

An application's Router helps to manage the state of the application.

An application's Router helps to manage the state of the application and the resources needed as a user navigates the app. This can include tasks such as requesting data from a model, hooking up controllers to views, or displaying templates.

You do this by creating a route for specific locations within your application. Routes specify parts of the application and the URLs associated to them. The URL is the key identifier that Ember uses to understand which application state needs to be presented to the user.

```
App.Router.map( function() {
   this.route( 'about' ); // Takes us to "/about"
});
```

The behaviors of a route (e.g.: requesting data from a model) are managed via instances of the Ember route object and are fired when a user navigates to a specific URL. An example would be requesting data from a model, like this:

```
App.EmployeesRoute = Ember.Route.extend({
   model: function() {
       return App.Employee.find();
   }
});
```

In this case, when a user navigates to the "/employees" section of the application, the route makes a request to the model for a list of all employees.

Models

An object representation of the data.

Models are an object representation of the data your application will use. It could be a simple array or data dynamically retrieved from a RESTful JSON API, via an Ajax

request. The Ember Data library offers the API for loading, mapping and updating data to models within your application.

Controllers

Controllers are typically used to store and represent model data and attributes. They act like a proxy, giving you access to the model's attributes and allowing templates to access them to dynamically render the display. This is why a template will always be connected to a controller.

The main thing to remember is that, while a model retrieves data, a controller is what you'll use to programmatically expose that data to different parts of your application. While it may seem that models and controllers are tightly coupled, in fact, models, themselves, have no knowledge of the controllers that will use them later on.

You can also store other application properties that need to persist but don't need to be saved to a server.

Views

Views in Ember.js are meant to manage events around user interaction and translate them into events that have meaning within your application. So, if a user clicks on a button to delete an employee, the view is responsible for interpreting that native browser click event and processing it appropriately within the context your application's current state.

Naming Conventions

One of the ways that Ember.js helps to minimize the amount of code needed and handle things for you behind the scenes is through naming conventions. The way that you define and name your routes (and resources) impacts the naming of your

controllers, models, views and templates. For example, if I create a route, called "employees":

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

I would then name my components, like this:

- **Route object:** *App.EmployeesRoute*
- **Controller:** *App.EmployeesController*
- **Model:** *App.Employee*
- **View:** *App.EmployeesView*
- **Template:** *employees*

Using this naming convention serves a dual purpose. First, it gives you a semantic relationship between like components. Secondly, Ember can automatically create the necessary objects that may not exist (e.g.: a route object or a controller) and wire them up for use in your application. This is the "magic" that I mentioned earlier. In fact, this is specifically what Ember does at the global Application level, when you instantiate the Application object:

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

That single line creates the default references to the application's router, controller, view and template.

- **Route object:** *App.ApplicationRoute*
- **Controller:** *App.ApplicationController*
- **View:** *App.ApplicationView*
- **Template:** *application*

Going back to the "employees" route that I created above, what will happen is that, when a user navigates to "/employees" in your application, Ember will look for the following objects:

- *App.EmployeesRoute*

- *App.EmployeesController*
- the *employees* template

If it doesn't find them, it will create an instance of each but simply won't render anything, since you haven't specified a model to derive data from or a template to display the data with. This is why the naming convention is so important. It allows Ember to know how to handle the tasks associated with a specific route, without you having to wire things up manually.

Notice that, in the first example, I used the singular name, "Employee," to define the model. That's on purpose. The very nature of the name "Employees" dictates that I may be working with 0 to many employees, so it's important to build a model that could provide the flexibility to return one employee or all employees. The singular naming convention of this model is not a requirement of Ember, as models themselves have no knowledge of the controllers that will use them later on. So you do have flexibility in naming them, but for consistency, sticking with this convention will make managing your code substantially easier.

Also, I chose to use the *resource()* method to define my route, because in this scenario, I would most likely have nested routes to manage detail pages for specific employee information. We'll talk about nesting later in the series.

The key takeaway is that by using a consistent naming scheme, Ember can easily manage the hooks that bind these components together without your needing to explicitly define the relationships via a ton of code.

> Full details of Ember's naming conventions are provided on the project's site and is a **must–read**.

Next Up: Building an App

> In the next part of the series, we'll dive into the code to create the basis for our application.

We've gone over the core concepts of Ember and discussed the key high-level aspects of the framework. In the next part of the series, we'll dive into the code to create the basis for our application. In the interim, I want to again recommend that you begin looking at the documentation for Handlebars to get a feel for the expressions syntax. Also, if you're really chomping at the bit to get into Ember, stay tuned to Tuts+ Premium, which will soon offer a full course that walks you through building an Ember-based application!

As I noted at the beginning of this article, Ember.js Core Team leads Yehuda Katz and Tom Dale reviewed this for accuracy, and gave it the thumbs up. Ember team approved! See you in a bit!

Tags: ember

By Rey Bango

Rey Bango is developer evangelist at Microsoft focused on meeting the needs of the web development community. He's an advocate for cross-browser, standards-based development using JavaScript & HTML5 and a former member of the jQuery Project Team.

Note: Want to add some source code? Type <pre><code> before it and </code></pre> after it. Find out more