**Getting Into Ember.js**

<http://dev.tutsplus.com/tutorials/getting-into-ember-js--net-30709>

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

Ember also relies on client-side templates... a **LOT**. It uses the [Handlebars templating library](http://www.handlebarsjs.com/) which provides expressions that allow you to create dynamic HTML-based templates.

<ul>

 {{#each people}}

   <li>Hello, {{name}}!</li>

 {{/each}}

</ul>

## Setting up Ember

The easiest way to get the files you need is to go to the Ember.js Github repo and pull down the [Starter Kit](https://github.com/emberjs/starter-kit).

<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>

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

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.

## 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.

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.

 The [Ember Data](https://github.com/emberjs/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.

## Views

Views in Ember.js are meant to manage events around user interaction and translate them into events that have meaning within your application.

## 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.

 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.

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.

위 한줄의 실행코드가 application의 router, controller, view, templete의 참조 객체를 생성한다. (model은 제외됨)

* **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.

Model에 대한 명명 규칙이 꼭 필요한 것은 아니지만 일관성을 위해 이 규칙을 따르는 것을 권한다.

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*](http://emberjs.com/guides/concepts/naming-conventions/)*are provided on the project's site and is a****must-read****.*

(<http://emberjs.com/guides/concepts/naming-conventions/>)

App.Router.map(function() {

this.resource('posts', function() { // the `posts` route

this.route('favorites'); // the `posts.favorites` route

this.resource('post'); // the `post` route

});

});

| **Route Name** | **Controller** | **Route** | **Template** |
| --- | --- | --- | --- |
| posts | PostsController | PostsRoute | posts |
| posts.favorites | PostsFavoritesController | PostsFavoritesRoute | posts/favorites |

## A Basic App

I noted previously that the easiest way to get the files you need is to go to the Ember.js Github repo and pull down the start kit

Open index.html in your browser, and you'll see the following:

**Welcome to Ember.js**

* red
* yellow
* blue

This is not very exciting, I know, but if you look at the code that rendered this, you’ll see that it was done with very little effort. If we look at "js/app.js", we see the following code:

App = Ember.Application.create({});

This code sets up an instance of the Ember application object, along with a default application template, event listeners and application router.

The next set of code sets up the behavior of a route, in this case, for the main index.html page:

App.IndexRoute = Ember.Route.extend({

  setupController: function(controller) {

    controller.set('content', ['red', 'yellow', 'blue']);

  }

});

Remember that routes are used to manage the resources associated with a specific URL within the application, and allows Ember to track the various states of individual pages. The URL is the key identifier that Ember uses to understand which application state needs to be presented to the user.

In this case, the root route is created by default in Ember. I could’ve also explicitly defined the route this way:

App.Router.map( function() {

    this.resource( 'index', { path: '/' } ); // Takes us to ‘/’

});

Going back to the following code:

App.IndexRoute = Ember.Route.extend({

  setupController: function(controller) {

    controller.set('content', ['red', 'yellow', 'blue']);

  }

});

In this case, when a user hits the site’s root, Ember will setup a controller that will load a sample set of data with a semantic name, called content. This data can later be used in the app, via this controller using that name. And that’s specifically what happens in index.html. Open the file and you’ll find the following:

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

<h2>Welcome to Ember.js</h2>

<ul>

{{#each item in model}}

    <li>{{item}}</li>

{{/each}}

 </ul>

</script>

This is a Handlebars client-side template. Remember that Handlebars is the templating library for Ember, and is vital to creating data-driven user interfaces for your app. Ember uses data attributes to link these templates to the controllers that manage your data, whether they’re specified via a route or as a standalone controller.

Handlebars는 데이터 중심으로 user Interface를 만드는데 매우 중요한 클라이언트측 템플릿입니다. Ember는 이런 (위의)방식으로 또는 독립적인 controller를 이용하여 데이터를 관리하고, 또 템플릿에 이를 링크시키기 위해 data 속성을 사용합니다.

위에서 controller는 datasource로 content라는 이름의 배열을 지정했습니다.

Basically, the array is your model, and the controller is used to expose that attributes of the model.

The naming conventions allow Ember to link this route’s resources (e.g.: the controller with data) to the template specified by the same name.

이름 명명규칙에 의해 Ember는 route의 resource(data를 가진 controller)를 같은 이름으로 작명된 templete에 연결시켜줍니다.

This gives the template access to the data exposed by the controller so it can render it using Handlebars’ directives.

Controller에 의해 데이터가 Template에 전달되고 Handlebars로 바로 렌더링 됩니다.

이때 배열은 model로 alias되고 각 요소가 loop 됩니다.

{{#each item in model}}

    <li>{{item}}</li>

{{/each}}



## Starting from the Ground Up

The [Ember guides](http://emberjs.com/guides/application/) do an excellent job of outlining specifically what instantiating an Ember application object does:

App = Ember.Application.create({});

* It sets your application’s namespace. All of the classes in your application will be defined as properties on this object (e.g. App.PostsView and App.PostsController). This helps to prevent polluting the global scope.  
  Application의 namespace를 정하고 이namespace안에 여러 객체들을 정의해 준다. 전역 객체 영역을 오염시키지 않도록 도와준다.
* It adds event listeners to the document and is responsible for sending events to your views.  
  document에 이벤트리스너을 추가하여 이를 view에 전달한다.
* It automatically renders the application template, the root-most template, into which your other templates will be rendered.  
  자동으로 templete를 랜더링한다.
* It automatically creates a router and begins routing, based on the current URL.  
  현재 URL정보를 기반으로 router를 생성하고 routing을 시작한다.

It’s important to note thatApp is not a keyword in Ember. It’s a normal global variable that you’re using to define the namespace and could be any valid variable name.

Taking the list above, what Ember does, via that one line, is essentially create this code for you automatically behind the scenes:

// Create the application namespace

App = Ember.Application.create({});

// Create the global router to manage page state via URLs

App.Router.map( function() {});

// Create the default application route to set application-level state properties

App.ApplicationRoute = Ember.Route.extend({});

// Create the default application template

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

    {{outlet}}

</script>

Defining the application template uses the same style syntax as any other template except with one small difference: the template name doesn’t need to be specified. So defining your template like this:

Application template은 템플릿 이름을 정하지 않아도 된다는 것 말고는 다른 template을 정의할 때와 같은 방법으로 정의한다.

<script type="text/x-handlebars">

    <h1>Application Template</h1>

</script>

또는

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

    <h1>Application Template</h1>

</script>

Ember는 data-template-name이 지정되지 않은 template을 Application template을 해석하고 바로 랜더링한다.

위 두가지 템플릿이 모두 정의되어 있다면 이름없는 template 만 우선적으로 렌더링된다.

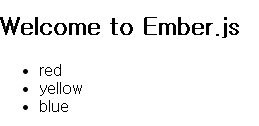
If you update index.html by adding this code:

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

    <h1>Application Template</h1>

    {{outlet}}

</script>



You’ll now see that the contents of the header tag appears on top of the content of the index template. The Handlebars *{{outlet}}* directive serves as a placeholder in the *application* template, allowing Ember to inject other templates into it (serving as a wrapper of sorts), and allowing you to have global UI features such as headers and footers that surround your content and functionality. By adding the *application* template toindex.html, you’ve instructed Ember to:

* Automatically render the *application* template
* Inject the index template into the *application* template via the Handlebars {{outlet}} directive
* Immediately process and render the index template

## Setting up Routes

Looking at js/app.js again, you’ll notice that a route has been created for the root page (index):

App.IndexRoute = Ember.Route.extend({

  setupController: function(controller) {

    controller.set('content', ['red', 'yellow', 'blue']);

  }

});

However, there’s no router instance. Remember that Ember will create a router by default if you don’t specify one. It will also create a default route entry for the root of the application similar to this:

App.Router.map( function() {

   this.resource( 'index', { path: '/' } );

});

This tells Ember that, when the root of the application is hit, it should load the resources of a route object instance called IndexRoute if it’s available

Ember internally knows that the root route should be named IndexRoute, will look for it, and load its resources, accordingly. In this case, it’s creating a controller that will contain data to be used in the index template.

Since URLs are the key identifiers that Ember uses to manage the state of your application, each one will generally have their own route handler specified if resources need to be loaded for that section of the app.

* Account: (URL: /account)
* Profile (URL: /profile)
* Gallery (URL: /gallery)

So you would create route handlers using the resource()-(e.g.: data or images).- method within Ember’s application router object instance like this:

App.Router.map( function() {

   this.resource( 'accounts' );

   this.resource( 'profiles' );

   this.resource( 'gallery' );

});

This allows Ember to understand the structure of the application and manage resources, accordingly. The routes definitions will correlate to individual route object instances which actually do the heavy-lifting like setting up or interfacing controllers:

App.GalleryRoute = Ember.Route.extend({

  setupController: function(controller) {

    controller.set('content', ['pic-1.png', 'pic-2.png', 'pic-3.png']);

  }

});

So in the example above, when a user visits "/gallery", Ember.js instantiate the GalleryRoute route object, setup a controller with data and render the gallery template.

Your application may also have nested URLs, like /account/new

you can define Ember resources that allow you to group routes together, like so:

App.Router.map( function() {

   this.resource( 'accounts',  function() {

     this.route( 'new' );

   });

});

In this example, we used the resource() method to group the routes together and the route() method to define the routes within the group. The general rule of thumb is to use resource() for nouns (Accounts and Account would both be resources even when nested) and route() for modifiers: (verbs like new andedit or adjectives like favorites and starred).

"/accounts":

* Controller: AccountsController
* Route: AccountsRoute
* Template: accounts (yes it’s lowercase)

"/accounts/new":

* Controller: AccountsNewController
* Route: AccountsNewRoute
* Template: accounts/new

When a user visits "/accounts/new" there’s a bit of a parent/child or master/detail scenario that occurs. Ember will first ensure that the resources for accounts are available and render the accounts template (this is the master part of it). Then, it will follow-up and do the same for "/accounts/new", setting up resources and rendering the accounts.new template.

Note that resources can also be nested for much deeper URL structures, like this:

App.Router.map( function() {

  this.resource( 'accounts', function() {

    this.route( 'new' );

    this.resource( 'pictures', function() {

      this.route( 'add' );

    });

  });

});