**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

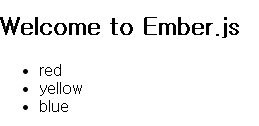
});

});

| 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:



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 that *App* 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-name*이 지정되지 않은 template을 Application template을 해석하고 바로 랜더링 한다.  
위 두 가지 템플릿이 모두 정의되어 있다면 이름없는 template 만 우선적으로 렌더링 된다.

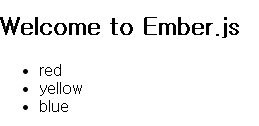
If you update *index.html*by adding this code:

<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* and *edit* 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' );

    });

  });

});

**Playing with Data**

App.IndexRoute = Ember.Route.extend({

 setupController: function(controller) {

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

 }

});

This allowed the controller to expose the data to the *index* template. That’s cute for a demo, but in real life, our data source will not be a hard-coded array.  
So, if we look at a result like this: ( simple array or data dynamically retrieved from a RESTful JSON API)

{

   "login": "rey",

   "id": 1,

   "age": 45,

   "gender": "male"

}

The attributes exposed in the model are:

* Login
* Id
* Age
* Gender

명시 적으로 controller 내에서 데이터를 설정할 수 있지만, 관심 대상, 코드 구조를 분리시키기 위해 model을 작성하는 것이 좋습니다.  
Alternatively, you could use a sister framework called [Ember Data](https://github.com/emberjs/data). It is an ORM-like API and persistence store, but I need to stress that it is in a state of flux as of this writing. It has a lot of potential, but using *Ember.Object* is much safer at this time.

**Defining your Models**

This data will be stored in my model and later used by a controller to fill a template.

{

  "nextId": null,

  "items": [{

          "title": "Docker, the Linux container runtime: now open-source",

          "url": "[http://docker.io](http://docker.io/)",

          "id": 5445387,

          "commentCount": 39,

          "points": 146,

          "postedAgo": "2 hours ago",

          "postedBy": "shykes"

      }, {

          "title": "What\u0027s Actually Wrong with Yahoo\u0027s Purchase of Summly",

          "url": "<http://hackingdistributed.com/2013/03/26/summly/>",

          "id": 5445159,

          "commentCount": 99,

          "points": 133,

          "postedAgo": "2 hours ago",

          "postedBy": "hoonose"

      },

  ],

  "version": "1.0",

  "cachedOnUTC": "\/Date(1364333188244)\/"

}

*Ember.Object*is the main base class for all Ember objects, and we’ll subclass it to create our model using its*extend()*method*.*we’ll add the following code to *js/app.js*immediately after the code that defines *App.IndexRoute:*

App.Item = Ember.Object.extend();

*App.Item* serves as the model class for data, but it has no methods to retrieve or manipulate that data. So, we’ll need to define those:

App.Item.reopenClass({

  all: function() {

      return $.getJSON("<http://api.ihackernews.com/page?format=jsonp&callback=?>").then(function(response) {

        var items = [];

        response.items.forEach( function (item) {

          items.push( App.Item.create(item) );

        });

         return items;

      });

  }

});

* Create your new model class by subclassing *Ember.Object* using *extend().*
* Add your model methods using *reopenClass().*
* Make an Ajax call to retrieve your data.
* Loop over your data, creating an *Item* object and pushing it into an array.
* Return the array when the method executes.

**Exposing Your Data**

Controllers는 proxy처럼 동작한다. Model의 속성값에 접근가능 하도록 하고, 동적으로 화면을 갱신시키기 위해 template에서도 model에 접근 가능하도록 해준다. 또한 서버에 저장하지 않고도 다른 application을 위해 데이터를 저장시킬 수도 있다.

현재 우리의 App은 다음과 같은 controller 를 가지고 있었는데  (the one that defines a static data set):

App.IndexRoute = Ember.Route.extend({

  setupController: function(controller) {

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

  }

});

다음과 같이 *App.IndexRoute* 에서 *model* method를 통해 model에 접근하도록 할수있다. (AKA the model hook):

App.IndexRoute = Ember.Route.extend({

  model: function() {

    return App.Item.all();

  }

});

Ember 는 별도로 정의하지 않으면 자동으로 controller를 생성한다. 따라서 위 경우에 Ember가 *Ember.ArrayController*의 인스턴스인 *IndexController* 를 생성하여 *model* method에 명시된 model을 자동으로 사용하게 된다. 우리가 할 일은 template에서 새로운 데이터를 사용하도록 업데이트하는 것이다.

we can see the following Handlebars template code:

{{#each item in model}}

    <li>{{item. title}}</li>

{{/each}}

controllers can also be used to define static attributes that need to persist throughout the life of your application. For example, I may want to persist certain static content, like this:

App.IndexController = Ember.ObjectController.extend({

  headerName: 'Welcome to the Hacker News App',

  appVersion:  2.1

});

Here, I subclass *Ember.ObjectController*to create a new controller for my *index* route and template to work with. I can now go to *index.html* and update my template to replace the following:

<h2>{{headerName}}</h2>

* By adhering to Ember’s naming conventions, I didn’t have to do any wiring to be able to use the controller with the index template.
* Even though I explicitly created an *IndexController*, Ember is smart enough not to overwrite the existing model that’s been associated via the route.

**Client-side Templates**

<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>

There's a slight difference with using it to define a template. First, we're specifying a *type* attribute of "text/x-handlebars". This *type* is ignored by the browser but leaves the text available for inspection and allows Ember to identify templates within the app.

 In addition, Ember uses a data attribute called "data-template-name" which Ember can use to associate specific parts of your application with a template. For example, the following declaration defines a template with a name "employee":

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

...

</script>

When your application starts, Ember scans the DOM for *type="text/x-handlebars*, compiles the templates it finds, and stores them in a property of the Ember object, called *Ember.TEMPLATES* which it uses to figure out what to render for a given route.

When you visit that URL, Ember knows it needs to load these resources and parse the template you've defined. It does this via its naming conventions, knowing that because you went to "/profile" it needs to load the resources defined in the *profile*, and render the template, named *data-template-name="profile".*

* **Route**: ProfileRoute
* **Controller**: ProfileController
* **Template**: profile (note that it's lowercase)

Also important to note is that, if you declare a template without a *data-template-name* attribute, Ember will assume that it is the Application-scoped template – the one typically used as a site-wide template for creating user interface elements, such as headers, footers and navigation. And if you don't explicitly define a template for an application or even a resource (e.g: for a URL), Ember does that automatically for you to ensure stability and consistency in your app.

**Expressions**

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

    <h2><strong>{{firstName}} {{lastName}}</strong></h2>

</script>

In this case, the *{{firstName}}* and *{{lastName}}* expressions will be parsed by Ember and replaced by actual data. In addition, Ember sets up observers so that as your data changes, your template is automatically updated and the updates reflected to the user of your app.

* Ember uses a special type attribute to define templates.
* Templates use standard markup along with delimited expressions, which are parsed on the client-side.
* These templates have the full feature set capabilities of Handlebars.
* Ember sets up observers to dynamically update your user interface data, as it changes.

If I wanted to ensure that the *title* data is available, I could add a conditional "if" statement by using the *#if* expression:

{{#if item.title}}

    <li>{{item.title}} - {{item.postedAgo}} by {{item.postedBy}}</li>

{{/if}}

Since this dataset contains more than one "record", it's safe to assume that we'd probably want to loop over each element of *item*. That's where the *{{#each}}* expression comes into play.

 we can use the *#each* expression to loop through every item available within our Ember model object. Remember that the Ember model is derived from the controller, which is associated to the template, via Ember's naming conventions. (controller에서 명명한 model이란 이름으로 순환한다)

<ul>

    {{#each item in model}}

    {{#if item.title}}

        <li>{{item.title}} - {{item.postedAgo}} by {{item.postedBy}}</li>

    {{/if}}

    {{/each}}

</ul>

If your conditional expression needs to be more complex, you'll want to create a [computed property](http://emberjs.com/guides/object-model/computed-properties/). This allows you to create a property based off of a method that can apply complex code conditions to your data.

* I need a computed property to scan each item and tell me if the title matches
* I need to create a controller that can be used by each item being enumerated over in the template
* I need to update the template so that it uses this controller for each item  
  The first thing I need to do is create the new controller that will wrap each item being looped over and create the computed property within it:

App.TitleController = Ember.ObjectController.extend({

    titleMatch: function() {

      return this.get(‘title’) === “Tearable Cloth Simulation in JavaScript”;

    }.property()

});

 we're subclassing *Ember.ObjectController* to create the controller. This is the controller that will wrap each item being looped over in our template. Next, we're creating a method, called *titleMatch* which uses the *get()* method to pull back the current title, compare it to the text I've defined, and return a boolean. Lastly, the Ember [*property()*](http://emberjs.com/api/classes/Function.html#method_property) method is called to define the *titleMatch* method as a computed property.

Once we have this in place, we update the template's *{{#each}}* expression to represent each item with the new controller we created. This is done by using the *itemController* directive.

 A key thing to understand is that *itemController* is a key phrase in Ember meant to associate a controller to items of a template. Don't confuse it for an actual controller name (as I did initially). The controller name is assigned to *itemController*, like this:

<ul>

   {{#each item in model itemController="title"}}

      {{#if titleMatch}}

        <li>{{foo.title}} - {{foo.postedAgo}} by {{foo.postedBy}}</li>

      {{/if}}

    {{/each}}

</ul>

Again, naming conventions dictate that, when assigning names in templates, we use lowercase. In this case, we're assigning *TitleController* to *itemController*.  
이름 명명규칙에 의해 위의 App.TitleController가 itemController="title"로 연결된다. 따라서 titleMatch에 접근 가능하게 된다.

**Binding Data to Elements**

Binding data to an element requires using special Ember helpers that assist in defining an attribute's context, as well as ensuring that the attributes are updated properly when data changes. For element attributes, the *{{bindAttr}}* helper is used to fill in the values for an attribute. If we needed to dynamically specify the URL of an image, we'd use the following syntax:

<img {{bindAttr src="logoUrl"}} alt="Logo">

The same can be done for attributes that don't receive a value, such as *disabled*:

<input type="checkbox" {{bindAttr disabled="isAdministrator"}}>

In this case, *isAdminstrator* could be a computed property based off a method in the controller, or just a normal object property giving you a lot of flexibility in defining the conditions for disabling the checkbox.

 If I wanted to use a conditional statement to define if a class should be applied to my element, I could use the following code:

<div {{bindAttr class="isUrgent"}}>Warning! </div>

Depending on the boolean state, my markup would either be:

<div {{bindAttr class="is-urgent"}}> Warning! </div>

for a *true* condition, or:

<div> Warning! </div>

for a *false* condition.

If you'd prefer to specify your own class based on the results, you can use a conditional expression similar to a ternary statement:

<div {{bindAttr class="isUrgent:urgent:normal"}}>

This will return *urgent* or *normal* for the class, based on the conditional value of *isUrgent*.

**An Introduction to Handlebars**

<table>

   {{#each Concerts}}

      <tr>

         <td>{{this}}</td>

      </tr>

   {{/each}}

</table>

You may also notice that we use the attribute name, *this*, to retrieve the current array element in the *each* loop.  
This example is good for an array of simple values, but how do you handle more complex data? Well, you essentially do the same thing. For example, we’re going to write a template for the following data:

[

   {

      Name : "Band",

      Date : "Aug 14th, 2012",

      Albums : [{Name : "Generic Name"},{Name : "Something Else!!"}]

   },

   {

      Name : "Other Guys",

      Date : "Aug 22nd, 2012"

      Albums : [{Name : "Album One"}]

   }

]

<table>

   {{#each Bands}}

      <tr>

         <td>{{Name}}</td>

         <td>{{Date}}</td>

         <td>{{Albums.0.Name}}</td>

      </tr>

   {{/each}}

</table>

In Handlebars, you can even access nested properties, like in the example above (*Albums.0.Name*), and of course, you could have used another *each* loop to iterate over a band’s albums. It’s worth noting that besides the dot notation to access nested properties, you can also use “../” to access a parent’s properties.

What if there aren’t any bands playing? You certainly don’t want an empty table, and Handlebars thankfully provides *if,else* and *unless* helpers. The *if* and *else* statements work like most programming languages

{{#if Bands}}

   <table>

      {{#each Bands}}

         <tr>

            <td>{{Name}}</td>

            <td>{{Date}}</td>

            <td>{{Albums.0.Name}}</td>

         </tr>

      {{/each}}

   </table>

{{else}}

   <h3>There are no concerts coming up.</h3>

{{/if}}

**Custom Helpers**

Handlebars gives you the ability to create your own custom helper. Simply register your function into Handlebars, and any template you compile afterwards can access your helper. There are two kinds of helpers that you can make:

* **Function helpers** are basically regular functions that, once registered, can be called anywhere in your template. Handlebars writes the function’s return value into the template.
* **Block helpers** are similar in nature to the *if, each*, etc. helpers. They allow you to change the context of what’s inside.

**Function helpers**

Let me show you a quick example of each. First, I’ll register a function helper with the following code:

Handlebars.registerHelper("Max", function(A, B){

   return (A > B) ? A : B;

});

The first argument passed to *registerHelper()* is the name of my customer helper; I’ll use this name in the template. The second argument is the function associated with this helper. Using this helper in a template is extremely simple:

{{Max 12 45}}

This template uses the *Max* helper, and passes the values 12 and 45 to the associated function. Handlebars function helpers support multiple parameters. You can directly insert numbers into the template itself, or you can use attributes from a JSON structure.

**Block helpers**

Block helpers allow you to set the context before running the code contained within the block. For example, consider the following object:

{

   Name: "Parent",

   Sub: {Name: "Child"}

}

In order to display both names, you can write a block helper that runs the template once with the parent’s context, and once with the child’s context. Here is the helper:

Handlebars.registerHelper("BothNames", function(context, options){

   return options.fn(context) + options.fn(context.Sub);

});

And the template looks like this:

{{#BothNames this}}

   <h2>{{Name}}</h2>

{{/BothName}}

The hash tag before the helper’s name tells Handlebars that this is a block helper, and you close the block not unlike you would an HTML tag. The *options.fn* function runs the section of template inside the block with whatever context you give it.