

COEN 168/268

# Mobile Web Application Development

## **Ember Views**

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University

The lecture contents is adapted from the Ember Guides available  
under the MIT license

<http://emberjs.com/guides/views/>

# Ember Views

# Ember Views

- Because Handlebars templates in Ember.js are so powerful, the majority of your application's user interface will be described using them.
- If you are coming from other JavaScript libraries, you may be surprised at how few views you have to create.




# Views in Ember.js are typically only created for the following reasons:

- When you need sophisticated handling of user events
- When you want to create a re-usable component

Often, both of these requirements will be present at the same time.

# Event Handling

- The role of the view in an Ember.js application is to translate primitive browser events into events that have meaning to your application.
- For example, imagine you have a list of todo items. Next to each todo is a button to delete that item:

- Buy milk 
- Mow the lawn 
- Read Nietzsche 

# The View Handles the Event

- The view is responsible for turning a *primitive event* (a click) into a *semantic event*: delete this todo!
- These semantic events are first sent up to the controller
- If no method is defined there, your application's router, will try to handle it



click

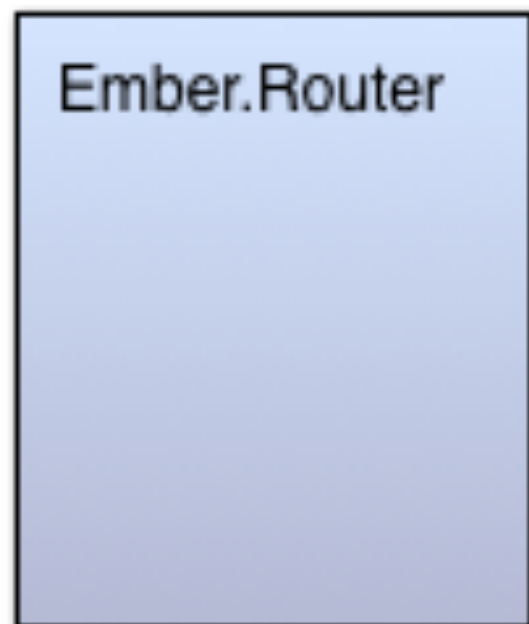


Delete

deleteTodo



Ember.Router



# Defining a View

You can use `Ember.View` to render a Handlebars template and insert it into the DOM.

To tell the view which template to use, set its `templateName` property. For example, if I had a `<script>` tag like this:

```
1 <html>
2   <head>
3     <script type="text/x-handlebars" data-template-name="say-hello">
4       Hello, <b>{{view.name}}</b>
5     </script>
6   </head>
7 </html>
```

I would set the `templateName` property to `"say-hello"`.

```
1 var view = Ember.View.create({
2   templateName: 'say-hello',
3   name: "Bob"
4 });
```

# Note

- For the remainder of the guide, the `templateName` property will be omitted from most examples.
- You can assume that if we show a code sample that includes an `Ember.View` and a `Handlebars` template, the view has been configured to display that template via the `templateName` property

# Adding and Removing Views

You can append views to the document by calling `appendTo`:

```
view.appendTo( '#container' );
```

As a shorthand, you can append a view to the document body by calling `append`:

```
view.append();
```

To remove a view from the document, call `remove`:

```
view.remove();
```

# Handling Events

# Handling Events

Instead of having to register event listeners on elements you'd like to respond to, simply implement the name of the event you want to respond to as a method on your view.

For example, imagine we have a template like this:

```
1 {{#view App.ClickableView}}
2   This is a clickable area!
3 {{/view}}
```

Let's implement `App.ClickableView` such that when it is clicked, an alert is displayed:

```
1 App.ClickableView = Ember.View.extend({
2   click: function(evt) {
3     alert("ClickableView was clicked!");
4   }
5 });
```

# What Happens?

- Events bubble up from the target view to each parent view in succession, until the root view.
- These values are read-only.
- If you want to manually manage views in JavaScript (instead of creating them using the `{{view}}` helper in Handlebars), see the `Ember.ContainerView` documentation below.

# Sending Events

To have the click event from `App.ClickableView` affect the state of your application, simply send an event to the view's controller:

```
1 App.ClickableView = Ember.View.extend({  
2   click: function(evt) {  
3     this.get('controller').send('turnItUp', 11);  
4   }  
5 });
```

# Sending Events, Cont'd

If the controller has an action handler called `turnItUp`, it will be called:

```
1 App.PlaybackController = Ember.ObjectController.extend({  
2   actions: {  
3     turnItUp: function(level){  
4       //Do your thing  
5     }  
6   }  
7 });
```

# Sending Events, Cont'd

If it doesn't, the message will be passed to the current route:

```
1 App.PlaybackRoute = Ember.Route.extend({  
2   actions: {  
3     turnItUp: function(level){  
4       //This won't be called if it's defined on App.PlaybackController  
5     }  
6   }  
7 });
```

# Inserting Views In Templates

# Inserting Views In Templates

- So far, we've discussed writing templates for a single view.
- However, as your application grows, you will often want to create a hierarchy of views to encapsulate different areas on the page.
- Each view is responsible for handling events and maintaining the properties needed to display it.

`{{view}}`

To add a child view to a parent, use the `{{view}}` helper, which takes a path to a view class.

# {{view}} Continued

```
1 // Define parent view
2 App.UserView = Ember.View.extend({
3   templateName: 'user',
4
5   firstName: "Albert",
6   lastName: "Hofmann"
7 });
8
9 // Define child view
10 App.InfoView = Ember.View.extend({
11   templateName: 'info',
12
13   posts: 25,
14   hobbies: "Riding bicycles"
15 });
```

# {{view}} Continued

```
1 <script type="text/x-handlebars" data-template-name="user">
2   User: {{view.firstName}} {{view.lastName}}
3   {{view App.InfoView}}
4 </script>
```

```
1 <script type="text/x-handlebars" data-template-name="info">
2   <b>Posts:</b> {{view.posts}}
3   <br>
4   <b>Hobbies:</b> {{view.hobbies}}
5 </script>
```

# {{view}} Continued

If we were to create an instance of `App.UserView` and render it, we would get a DOM representation like this:

```
1 User: Albert Hofmann
2 <div>
3   <b>Posts:</b> 25
4   <br>
5   <b>Hobbies:</b> Riding bicycles
6 </div>
```

# Relative Paths

Instead of specifying an absolute path, you can also specify which view class to use relative to the parent view.

For example, we could nest the above view hierarchy.

# Relative Paths, Continued

```
1 App.UserView = Ember.View.extend({
2   templateName: 'user',
3
4   firstName: "Albert",
5   lastName: "Hofmann",
6
7   infoView: Ember.View.extend({
8     templateName: 'info',
9
10    posts: 25,
11    hobbies: "Riding bicycles"
12  })
13 });
```

# Relative Paths, Continued

```
1 User: {{view.firstName}} {{view.lastName}}  
2 {{view view.infoView}}
```

When nesting a view class like this, make sure to use a lowercase letter, as Ember will interpret a property with a capital letter as a global property.

# Setting Child View Templates

If you'd like to specify the template your child views use inline in the main template, you can use the block form of the `{{view}}` helper.

We might rewrite the above example...

# Setting Child View Templates, Continued

```
1 App.UserView = Ember.View.extend({
2   templateName: 'user',
3
4   firstName: "Albert",
5   lastName: "Hofmann"
6 });
7
8 App.InfoView = Ember.View.extend({
9   posts: 25,
10  hobbies: "Riding bicycles"
11 });
```

# Setting Child View Templates, Continued

```
1 User: {{view.firstName}} {{view.lastName}}
2 {{#view App.InfoView}}
3   <b>Posts:</b> {{view.posts}}
4   <br>
5   <b>Hobbies:</b> {{view.hobbies}}
6 {{/view}}
```

When you do this, it may be helpful to think of it as assigning views to portions of the page. This allows you to encapsulate event handling for just that part of the page.

# Adding Layouts to Views

# Adding Layouts to Views

- Views can have a secondary template that wraps their main template.
- Like templates, layouts are Handlebars templates that will be inserted inside the view's tag.
- To tell a view which layout template to use, set its `layoutName` property.

# Adding Layouts to Views, Continued

- To tell the layout template where to insert the main template, use the Handlebars `{{yield}}` helper.
- The HTML contents of a view's rendered template will be inserted where the `{{yield}}` helper is.

# Adding Layouts to Views, Continued

First, you define the following layout template:

```
1 <script type="text/x-handlebars" data-template-name="my_layout">
2   <div class="content-wrapper">
3     {{yield}}
4   </div>
5 </script>
```

And then the following main template:

```
1 <script type="text/x-handlebars" data-template-name="my_content">
2   Hello, <b>{{view.name}}</b>!
3 </script>
```

# Adding Layouts to Views, Continued

Then, define a view & tell it to wrap the template with the layout:

```
1 AViewWithLayout = Ember.View.extend({  
2   name: 'Teddy',  
3   layoutName: 'my_layout',  
4   templateName: 'my_content'  
5 });
```

This will result in view instances containing the following HTML

```
1 <div class="content-wrapper">  
2   Hello, <b>Teddy</b>!  
3 </div>
```

# Applying Layouts in Practice

- Layouts are extremely useful when you have a view with a common wrapper and behavior, but its main template might change.
- One possible scenario is a Popup View.

You can define your popup layout template:

```
1 <script type="text/x-handlebars" data-template-name="popup">
2   <div class="popup">
3     <button class="popup-dismiss">x</button>
4     <div class="popup-content">
5       {{yield}}
6     </div>
7   </div>
8 </script>
```

Then define your popup view:

```
1 App.PopupView = Ember.View.extend({
2   layoutName: 'popup'
3 });
```

Now you can re-use your popup with different templates:

```
1 {{#view App.PopupView}}
2   <form>
3     <label for="name">Name:</label>
4     <input id="name" type="text" />
5   </form>
6 {{/view}}
7
8 {{#view App.PopupView}}
9   <p> Thank you for signing up! </p>
10 {{/view}}
```

# Customizing a View's Element

# Changing the HTML Tag

- A view is represented by a single DOM element on the page.
- You can change what kind of element is created by changing the tagName property.

```
1 App.MyView = Ember.View.extend({  
2   tagName: 'span'  
3 });
```

# Specifying the CSS Class Names

You can also specify which class names are applied to the view by setting its `classNames` property to an array of strings:

```
1 App.MyView = Ember.View.extend({  
2   classNames: ['my-view']  
3 });
```

# Changing the CSS Class Names Via Bindings

- If you want class names to be determined by the state of properties on the view, you can use class name bindings.
- If you bind to a Boolean property, the class name will be added or removed depending on the value:

```
1 App.MyView = Ember.View.extend({  
2   classNameBindings: ['isUrgent'],  
3   isUrgent: true  
4 });
```

# Boolean Class Names Are Dasherized

This would render a view like this:

```
<div class="ember-view is-urgent">
```

If `isUrgent` is changed to `false`, then the `is-urgent` class name will be removed.

# Boolean Class Names Can Be Customized

By default, the name of the Boolean property is dasherized. You can customize the class name applied by delimiting it with a colon:

```
1 App.MyView = Ember.View.extend({  
2   classNameBindings: ['isUrgent:urgent'],  
3   isUrgent: true  
4 });
```

This would render this HTML:

```
<div class="ember-view urgent">
```

# Boolean Class Names Can Be Customized Both When True and False

Besides the custom class name for the value being `true`, you can also specify a class name which is used when the value is `false`:

```
1 App.MyView = Ember.View.extend({  
2   classNameBindings: ['isEnabled:enabled:disabled'],  
3   isEnabled: false  
4 });
```

# Boolean Class Names Can Be Customized Both When True and False, Continued

This would render this HTML:

```
<div class="ember-view disabled">
```

# You Can Add Class Names Only When **False**

You can also specify to only add a class when the property is `false` by declaring `classNameBindings` like this:

```
1 App.MyView = Ember.View.extend({  
2   classNameBindings: ['isEnabled::disabled'],  
3   isEnabled: false  
4 });
```

This would render this HTML:

```
<div class="ember-view disabled">
```

# You Can Add Class Names Only When **False**

If the `isEnabled` property is set to `true`, no class name is added:

```
<div class="ember-view">
```

If the bound value is a string, that value will be added as a class name without modification:

```
1 App.MyView = Ember.View.extend({  
2   classNameBindings: ['priority'],  
3   priority: 'highestPriority'  
4 });
```

This would render this HTML:

```
1 <div class="ember-view highestPriority">
```

# Attribute Bindings on a View

You can bind attributes to the DOM element that represents a view by using `attributeBindings`:

```
1 App.MyView = Ember.View.extend({  
2   tagName: 'a',  
3   attributeBindings: ['href'],  
4   href: "http://emberjs.com"  
5 });
```

# Attribute Bindings on a View, Cont'd

You can also bind these attributes to differently named properties:

```
1 App.MyView = Ember.View.extend({  
2   tagName: 'a',  
3   attributeBindings: ['customHref:href'],  
4   customHref: "http://emberjs.com"  
5 });
```

# Customizing a View's Element from Handlebars

- When you append a view, it creates a new HTML element that holds its content.
- If your view has any child views, they will also be displayed as child nodes of the parent's HTML element.

# Changing the tag using tagName

By default, new instances of `Ember.View` create a `<div>` element. You can override this by passing a `tagName` parameter:

```
{{view App.InfoView tagName="span"}}
```

# Adding an id attribute using id

You can also assign an ID attribute to the view's HTML element by passing an `id` parameter:

```
{{view App.InfoView id="info-view"}}
```

This makes it easy to style using CSS ID selectors:

```
1 /** Give the view a red background. */  
2 #info-view {  
3     background-color: red;  
4 }
```

# Adding an class names:

You can assign class names similarly:

```
{{view App.InfoView class="info urgent"}}
```

# Binding class names

You can bind class names to a property of the view by using `classBinding` instead of `class`.

```
1 App.AlertView = Ember.View.extend({  
2   priority: "p4",  
3   isUrgent: true  
4 });
```

```
{{view App.AlertView classBinding="isUrgent priority"}}
```

This yields a view wrapper that will look something like this:

```
<div id="ember420" class="ember-view is-urgent p4"></div>
```

# Built-In Views

# Built-In Views

Ember comes pre-packaged with a set of views for building a few basic controls like text inputs, check boxes, and select lists.

They are:

# Ember.Checkbox

```
1 <label>
2   {{view Ember.Checkbox checked=model.isDone}}
3   {{model.title}}
4 </label>
```

# Ember.TextField

```
1 App.MyText = Ember.TextField.extend({  
2   formBlurred: null, // passed to the view helper as formBlurred=controllerPropertyName  
3   change: function(evt) {  
4     this.set('formBlurred', true);  
5   }  
6 });
```

# Ember.Select

```
1 {{view Ember.Select viewName="select"  
2      content=people  
3      optionLabelPath="content.fullName"  
4      optionValuePath="content.id"  
5      prompt="Pick a person:"  
6      selection=selectedPerson}}}
```

# Ember.TextArea

```
1 var textArea = Ember.TextArea.create({  
2   valueBinding: 'TestObject.value'  
3 });
```

# Manually Managing View Hierarchy

# Ember.ContainerView

- As you probably know by now, views usually create their child views by using the `{{view}}` helper.
- However, it is sometimes useful to *manually* manage a view's child views. - `Ember.ContainerView` is the way to do just that.

# Ember.ContainerView, Continued

As you programmatically add or remove views to a `ContainerView`, those views' rendered HTML are added or removed from the DOM to match.

```
1 var container = Ember.ContainerView.create();
2 container.append();
3
4 var firstView = App.FirstView.create(),
5     secondView = App.SecondView.create();
6
7 container.pushObject(firstView);
8 container.pushObject(secondView);
9
10 // When the rendering completes, the DOM will contain a `div` for the ContainerView
11 // and nested inside of it, a `div` for each of firstView and secondView.
```

# Defining the Initial Views of a Container View

There are a few ways to specify which initial child views a `ContainerView` should render.

The most straight-forward way is to add them in `init`:

```
1 var container = Ember.ContainerView.create({
2   init: function() {
3     this._super();
4     this.pushObject(App.FirstView.create());
5     this.pushObject(App.SecondView.create());
6   }
7 });
8
9 container.objectAt(0).toString(); //=> '<App.FirstView:ember123>'
10 container.objectAt(1).toString(); //=> '<App.SecondView:ember124>'
```

## Using the `childViews` property

As a shorthand, you can specify a `childViews` property that will be consulted on instantiation of the `ContainerView` also.

# This example is equivalent to the one above:

```
1 var container = Ember.ContainerView.extend({  
2   childViews: [App.FirstView, App.SecondView]  
3 });  
4  
5 container.objectAt(0).toString(); //=> '<App.FirstView:ember123>'  
6 container.objectAt(1).toString(); //=> '<App.SecondView:ember124>'
```

Another bit of syntactic sugar is available as an option as well:

- Specifying string names in the `childViews` property that correspond to properties on the `ContainerView`.
- This style is less intuitive at first but has the added bonus that each named property will be updated to reference its instantiated child view.

```
1 var container = Ember.ContainerView.create({
2   childViews: ['firstView', 'secondView'],
3   firstView: App.FirstView,
4   secondView: App.SecondView
5 });
6
7 container.objectAt(0).toString(); //=> '<App.FirstView:ember123>'
8 container.objectAt(1).toString(); //=> '<App.SecondView:ember124>'
9
10 container.get('firstView').toString(); //=> '<App.FirstView:ember123>'
11 container.get('secondView').toString(); //=> '<App.SecondView:ember124>'
```

# It Feels Like an Array Because it *is* an Array

- You may have noticed that some of these examples use `pushObject` to add a child view, just like you would interact with an Ember array.
- `Ember.ContainerView` gains its collection-like behavior by mixing in `Ember.MutableArray`.
- That means that you can manipulate the collection of views very expressively, using methods like `pushObject`, `popObject`, `shiftObject`, `unshiftObject`, `insertAt`, `removeAt`, or any other method you would use to interact with an Ember array.

The lecture contents is adapted from the Ember Guides available  
under the MIT license

<http://emberjs.com/guides/views/>

COEN 168/268

# Mobile Web Application Development

## **Ember Views**

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University