

COEN 168/268

Mobile Web Application Development

Ember Components

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/components/>

Introduction

Why do we need Ember Components?

- HTML was designed in a time when the browser was a simple document viewer.
- Developers building great web apps need something more.
- Instead of trying to replace HTML, however, Ember.js embraces it
- Then adds powerful new features that modernize it for building web apps.

HTML tags are limiting

- Currently, you are limited to the tags that are created for you by the W3C.
- Ember gives you a way to define your own, application-specific HTML tag then implement their behavior using JavaScript
- The W3C is working on a draft for 'custom elements' right now (<http://www.w3.org/TR/components-intro/>)

Ember Components Adhere to the W3C Spec

- Components follows as closely to the Web Components specification as possible.
- Once Custom Elements are widely available in browsers, you should be able to easily migrate your Ember components to the W3C standard and have them be usable by other frameworks.
- Ember is very dedicated to make this standard work

A Short Example

- This lecture will create a `blog-post` custom element that you could use again and again in your application.
- This element can be reused in your code over and over.

**JS Bin**

Save

HTML**CSS****JavaScript**

Console

Output

Rails is omakase

There are lots of à la carte software environments in this world.

Edit title:

Broken Promises

James Coglan wrote a lengthy article about Promises in node.js.

Edit title:

Auto-run JS ☒

Run with JS

Defining a Component

Defining a Component

- To define a component, create a template whose name starts with `components/`.
- To define a new component, `{{blog-post}}` for example, create a `components/blog-post` template.
- Components **must** have a dash in their name to distinguish them from current or future HTML elements

If you are including your Handlebars templates inside an HTML file via `<script>` tags, it would look like this:

```
1 <script type="text/x-handlebars" id="components/blog-post">
2   <h1>Blog Post</h1>
3   <p>Lorem ipsum dolor sit amet.</p>
4 </script>
```

If you're using build tools, create a Handlebars file at `templates/components/blog-post.handlebars`.

Having a template whose name starts with `components/` creates a component of the same name. Given the above template, you can now use the `{{blog-post}}` custom element:

```
1 <h1>My Blog</h1>
2 {{#each}}
3   {{blog-post}}
4 {{/each}}
```

```
<!DOCTYPE html>
<html>
<head>
<script src="http://code.jquery.com/jquery.js">
</script>
<script
src="//cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.
0.0/handlebars.js"></script>
<script src="http://builds.emberjs.com/ember-
latest.js"></script>
<meta charset=utf-8 />
<title>JS Bin</title>
</head>
<body>
  <script type="text/x-handlebars" data-template-
name="index">
    {{#each}}
      {{blog-post}}
    {{/each}}
  </script>

  <script type="text/x-handlebars" data-template-
name="components/blog-post">
    <h1>Blog Post</h1>
    <p>Lorem ipsum dolor sit amet.</p>
  </script>
</body>
</html>
```

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

posts = [{
  title: "Rails is omakase",
  body: "There are lots of à la carte software
environments in this world."
}, {
  title: "Broken Promises",
  body: "James Coglán wrote a lengthy article about
Promises in node.js."
}];

App.IndexRoute = Ember.Route.extend({
  model: function() {
    return posts;
  }
});
```

Output


Run with JS Auto-run JS ☒

Blog Post

Lorem ipsum dolor sit amet.

Blog Post

Lorem ipsum dolor sit amet.

 tomdale

Under the hood

- Each component is backed by an element
- The default is to use a `<div>` element to contain your component's template
- However, you can customize this

Defining a Component Subclass

- Often times, your components will just encapsulate certain snippets of Handlebars templates that you find yourself using over and over.
- In those cases, you do not need to write any JavaScript at all.
- Just define the Handlebars template as described above and use the component that is created.

Defining a Component Subclass

- If you need to customize the behavior of the component you'll need to define a subclass of `Ember.Component`.
- You would need a custom subclass if you wanted to:
 - Change a component's element
 - Respond to actions from the component's template
 - Manually make changes to the component's element using JavaScript.

Defining a Component Subclass

- Knows which subclass powers a component based on name.
- For example, if you have a component called `blog-post`:
 - You would create a subclass called `App.BlogPostComponent`.
- If your component was called `audio-controls`, the class name would be `App.AudioControlsComponent`.

Ember looks for a class with the **camelized** name followed by `Component`.

Therefore, it will look like this:

Component Name		Component Class
blog-post		App.BlogPostComponent
audio-player-controls		App.AudioPlayerControlsComponent

Passing Properties to a Component

Passing Properties to a Component

By default a component does not have access to properties in the template scope in which it is used.

For example, imagine you have a `blog-post` component that is used to display a blog post:

```
1 <script type="text/x-handlebars" id="components/blog-post">
2   <h1>Component: {{title}}</h1>
3   <p>Lorem ipsum dolor sit amet.</p>
4 </script>
```

You can see that it has a `{{title}}` Handlebars expression to print the value of the `title` property inside the `<h1>`.

Now imagine we have the following template and route:

```
1 App.IndexRoute = Ember.Route.extend({
2   model: function() {
3     return {
4       title: "Rails is omakase"
5     };
6   }
7 });
```

```
1 {{! index.handlebars }}
2 <h1>Template: {{title}}</h1>
3 {{blog-post}}
```

The first `<h1>` (from the outer template) displays the `title` property, but the second `<h1>` (from inside the component) is empty.

HTML ▾

```
<!DOCTYPE html>
<html>
<head>
<script src="http://code.jquery.com/jquery.js">
</script>
<script
src="//cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.
0.0/handlebars.js"></script>
<script src="http://builds.emberjs.com/ember-
latest.js"></script>
<meta charset=utf-8 />
<title>JS Bin</title>
</head>
<body>
  <script type="text/x-handlebars"
id="components/blog-post">
    <h1>Component: {{title}}</h1>
    <p>Lorem ipsum dolor sit amet.</p>
  </script>

  <script type="text/x-handlebars" id="index">
    <h1>Template: {{title}}</h1>
    {{blog-post}}
  </script>
</body>
</html>
```

JavaScript ▾

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

App.IndexRoute = Ember.Route.extend({
  model: function() {
    return {
      title: "Rails is omakase"
    };
  }
});
```

Output

Run with JS

Auto-run JS ☒ ↗

Template: Rails is omakase

Component:

Lorem ipsum dolor sit amet.



tomdale

We can fix this by making the `title` property available to the component:

```
1 {{blog-post title=title}}
```

This will make the `title` property in the outer template scope available inside the component's template using the same name, `title`.

HTML ▾

```
<!DOCTYPE html>
<html>
<head>
<script src="http://code.jquery.com/jquery.js">
</script>
<script
src="//cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.
0.0/handlebars.js"></script>
<script src="http://builds.emberjs.com/ember-
latest.js"></script>
<meta charset=utf-8 />
<title>JS Bin</title>
</head>
<body>
  <script type="text/x-handlebars"
id="components/blog-post">
    <h1>Component: {{title}}</h1>
    <p>Lorem ipsum dolor sit amet.</p>
  </script>

  <script type="text/x-handlebars" id="index">
    <h1>Template: {{title}}</h1>
    {{blog-post title=title}}
  </script>
</body>
</html>
```

JavaScript ▾

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

App.IndexRoute = Ember.Route.extend({
  model: function() {
    return {
      title: "Rails is omakase"
    };
  }
});
```

Output

Run with JS

Auto-run JS ☒ ↗

Template: Rails is omakase

Component: Rails is omakase

Lorem ipsum dolor sit amet.



tomdale

If, in the above example, the model's `title` property was instead called `name`, we would change the component usage to:

```
1 {{blog-post title=name}}
```

In other words, you are binding a named property from the outer scope to a named property in the component scope, with the syntax `componentProperty=outerProperty`.

HTML ▾

```
<!DOCTYPE html>
<html>
<head>
<script src="http://code.jquery.com/jquery.js">
</script>
<script
src="//cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.
0.0/handlebars.js"></script>
<script src="http://builds.emberjs.com/ember-
latest.js"></script>
<meta charset=utf-8 />
<title>JS Bin</title>
</head>
<body>
  <script type="text/x-handlebars"
id="components/blog-post">
    <h1>Component: {{title}}</h1>
    <p>Lorem ipsum dolor sit amet.</p>
  </script>

  <script type="text/x-handlebars" id="index">
    <h1>Template: {{name}}</h1>
    {{blog-post title=name}}
  </script>
</body>
</html>
```

JavaScript ▾

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

App.IndexRoute = Ember.Route.extend({
  model: function() {
    return {
      name: "Rails is omakase"
    };
  }
});
```

Output

Run with JS

Auto-run JS ☒ ↗

Template: Rails is omakase

Component: Rails is omakase

Lorem ipsum dolor sit amet.



tomdale

These properties are bound

- It is important to note that the value of these properties is bound.
- Whether you change the value on the model or inside the component, the values stay in sync.

```
<head>
<script src="http://code.jquery.com/jquery.js">
</script>
<script
src="//cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.
0.0/handlebars.js"></script>
<script src="http://builds.emberjs.com/ember-
latest.js"></script>
<meta charset=utf-8 />
<title>JS Bin</title>
</head>
<body>
  <script type="text/x-handlebars"
id="components/blog-post">
    <h1>Component: {{title}}</h1>
    <p>Lorem ipsum dolor sit amet.</p>
    <p>Edit title in component: {{input type="text"
value=title}}</p>
  </script>

  <script type="text/x-handlebars" id="index">
    <h1>Template: {{name}}</h1>
    {{blog-post title=name}}
    <p>Edit title in outer template: {{input
type="text" value=name}}</p>
  </script>
</body>
</html>
```

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

App.IndexRoute = Ember.Route.extend({
  model: function() {
    return {
      name: "Rails is omakase"
    };
  }
});
```

Output

Run with JS

Auto-run JS ☒



Template: Rails is omakase

Component: Rails is omakase

Lorem ipsum dolor sit amet.

Edit title in component:

Edit title in outer template:



tomdale

You can also bind properties from inside an `{{#each}}` loop. This will create a component for each item and bind it to each model in the loop.

```
1 {{#each}}  
2   {{blog-post title=title}}  
3 {{/each}}
```

```
<head>
<script src="http://code.jquery.com/jquery.js">
</script>
<script
src="//cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.
0.0/handlebars.js"></script>
<script src="http://builds.emberjs.com/ember-
latest.js"></script>
<meta charset=utf-8 />
<title>JS Bin</title>
</head>
<body>
  <script type="text/x-handlebars"
id="components/blog-post">
    <h1>Component: {{title}}</h1>
    <p>Lorem ipsum dolor sit amet.</p>
    <p>Edit title in component: {{input type="text"
value=title}}</p>
  </script>

  <script type="text/x-handlebars" id="index">
    <h1>Template: {{name}}</h1>
    {{blog-post title=name}}
    <p>Edit title in outer template: {{input
type="text" value=name}}</p>
  </script>
</body>
</html>
```

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

App.IndexRoute = Ember.Route.extend({
  model: function() {
    return {
      name: "Rails is omakase"
    };
  }
});
```

Output Run with JS Auto-run JS


Template: Rails is omakase

Component: Rails is omakase

Lorem ipsum dolor sit amet.

Edit title in component: Rails is omakase

Edit title in outer template: Rails is omakase

tomdale

Wrapping Content in a Component

Sometimes, you may want to define a component that wraps content provided by other templates.

For example, imagine we are building a `blog-post` component that we can use in our application to display a blog post:

```
1 <script type="text/x-handlebars" id="components/blog-post">
2   <h1>{{title}}</h1>
3   <div class="body">{{body}}</div>
4 </script>
```

Now, we can use the `{{blog-post}}` component and pass it properties in another template:

```
1 {{blog-post title=title body=body}}
```

HTML ▾

```
<!DOCTYPE html>
<html>
<head>
<script src="http://code.jquery.com/jquery.js">
</script>
<script
src="//cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.
0.0/handlebars.js"></script>
<script src="http://builds.emberjs.com/ember-
latest.js"></script>
<meta charset=utf-8 />
<title>JS Bin</title>
</head>
<body>
  <script type="text/x-handlebars" data-template-
name="index">
    {{blog-post title=title body=body}}
  </script>

<script type="text/x-handlebars" id="components/blog-
post">
  <h1>{{title}}</h1>
  <div class="body">{{body}}</div>
</script>

</body>
</html>
```

JavaScript ▾

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

App.IndexRoute = Ember.Route.extend({
  model: function() {
    return {
      title: "Top 2 Paula Cole Songs",
      body: "1. Where Have All the Cowboys Gone? 2. I
Don't Want to Wait"
    };
  }
});
```

Output

Run with JS

Auto-run JS ☒ ↗

Top 2 Paula Cole Songs

1. Where Have All the Cowboys Gone? 2. I Don't Want to Wait



tomdale

Block Form

- In previous case, the content came from the model.
- But what if we want the developer using our component to be able to provide custom HTML content?
- Components support being used in **block form**

Block Form

- In block form, components can be passed a Handlebars template that is rendered inside the component's template wherever the `{{yield}}` expression appears.
- To use the block form, add a `#` character to the beginning of the component name, then make sure to add a closing tag.
- In that case, we can use the `{{blog-post}}` component in **block form** and tell Ember where the block content should be rendered using the `{{yield}}` helper.

To update the example above, we'll first change the component's template:

```
1 <script type="text/x-handlebars" id="components/blog-post">
2   <h1>{{title}}</h1>
3   <div class="body">{{yield}}</div>
4 </script>
```

You can see that we've replaced `{{body}}` with `{{yield}}`. This tells Ember that this content will be provided when the component is used.

Next, we'll update the template using the component to use the block form:

```
1 {{#blog-post title=title}}  
2     <p class="author">by {{author}}</p>  
3     {{body}}  
4 {{/blog-post}}
```

HTML ▾

```
<!DOCTYPE html>
<html>
<head>
<script src="http://code.jquery.com/jquery.js">
</script>
<script
src="//cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.
0.0/handlebars.js"></script>
<script src="http://builds.emberjs.com/ember-
latest.js"></script>
<meta charset=utf-8 />
<title>JS Bin</title>
</head>
<body>
  <script type="text/x-handlebars" data-template-
name="index">
    {{#blog-post title=title}}
      <p class="author">by {{author}}</p>
      {{body}}
    {{/blog-post}}
  </script>

  <script type="text/x-handlebars" id="components/blog-
post">
    <h1>{{title}}</h1>
    <div class="body">{{yield}}</div>
  </script>
```

JavaScript ▾

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

App.IndexRoute = Ember.Route.extend({
  model: function() {
    return {
      title: "Top 2 Paula Cole Songs",
      author: "Tom Dale",
      body: "1. Where Have All the Cowboys Gone? 2. I
Don't Want to Wait"
    };
  }
});
```

Output

Run with JS

Auto-run JS ☒ ↗

Top 2 Paula Cole Songs

by Tom Dale

1. Where Have All the Cowboys Gone? 2. I Don't Want to Wait



tomdale

A Note About Scope

- It's important to note that the template scope inside the component block is the same as outside.
- If a property is available in the template outside the component, it is also available inside the component block.
- The next slides shows the concept


```
<script
src="//cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.
0.0/handlebars.js"></script>
<script src="http://builds.emberjs.com/ember-
latest.js"></script>
<meta charset=utf-8 />
<title>JS Bin</title>
</head>
<body>
  <script type="text/x-handlebars" data-template-
name="index">
    <p>The <code>name</code> property outside the
component's template: {{name}}</p>
    {{#my-component}}
      The <code>name</code> property inside the
component's block template: {{name}}
    {{/my-component}}
  </script>

  <script type="text/x-handlebars" data-template-
name="components/my-component">
    <p>{{yield}}</p>
    <p>The <code>name</code> property in the
component's template: {{name}}</p>
  </script>
</body>
</html>
```

JavaScript ▾

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

App.IndexRoute = Ember.Route.extend({
  model: function() {
    return { name: "Girl Talk" }
  }
});
```

▶ 1 error

Output

Run with JS

Auto-run JS ☒



The name property outside the component's template: Girl Talk

The name property inside the component's block template: Girl Talk

The name property in the component's template:



tomdale

Customizing A Component's Element

By Default, Components are `<div>` elements

- By default, each component is backed by a `<div>` element.
- If you were to look at a rendered component in your developer tools, you would see a DOM representation that looked something like:

```
1 <div id="ember180" class="ember-view">  
2   <h1>My Component</h1>  
3 </div>
```

Customizing the Element

To use a tag other than `div`, subclass `Ember.Component` and assign it a `tagName` property. This property can be any valid HTML5 tag name as a string.

```
1 App.NavigationBarComponent = Ember.Component.extend({  
2   tagName: 'nav'  
3 });
```

```
1 {{! templates/components/navigation-bar }}  
2 <ul>  
3   <li>{{#link-to 'home'}}Home{{/link-to}}</li>  
4   <li>{{#link-to 'about'}}About{{/link-to}}</li>  
5 </ul>
```

Customizing Class Names

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

```
1 App.NavigationBarComponent = Ember.Component.extend({  
2   classNames: ['primary']  
3 });
```

Customizing Class Names Via Bindings

If you bind to a Boolean property, the class name will be added or removed depending on the value:

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

This component would render the following:

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

If `isUrgent` is changed to `false`, then the `is-urgent` class name will be removed. By default the boolean properties will be dasherized.

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.TodoItemComponent = Ember.Component.extend({  
2   classNameBindings: ['isUrgent:urgent'],  
3   isUrgent: true  
4 });
```

This would render this HTML:

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


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.TodoItemComponent = Ember.Component.extend({  
2   classNameBindings: ['isEnabled:enabled:disabled'],  
3   isEnabled: false  
4 });
```

This would render this HTML:

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

You can also specify a class which should only be added when the property is `false` by declaring `classNameBindings` like this:

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

This would render this HTML:

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

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.TodoItemComponent = Ember.Component.extend({  
2   classNameBindings: ['priority'],  
3   priority: 'highestPriority'  
4 });
```

This would render this HTML:

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

Customizing Attributes

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

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

You can also bind these attributes to differently named properties:

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

Example

Here is an example todo application that shows completed todos with a red background...

```
src="http://builds.emberjs.com/ember-
latest.js"></script>
<meta charset=utf-8 />
<title>JS Bin</title>
</head>
<body>
  <script type="text/x-handlebars"
data-template-name="application">
    <h1>Todos</h1>
    <ul>
      {{#each}}
        {{todo-item title=title
isDone=isDone}}
      {{/each}}
    </ul>
  </script>

  <script type="text/x-handlebars"
data-template-name="components/todo-
item">
    <label>{{input type="checkbox"
checked=isDone}} {{title}}</label>
  </script>
</body>
</html>
```

CSS

```
li.is-done {
  background-color: red;
}
```

JavaScript

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

todos = [{
  title: "Learn Ember.js",
  isDone: false
}, {
  title: "Make awesome web apps",
  isDone: true
}];

App.ApplicationRoute =
Ember.Route.extend({
  model: function() {
    return todos;
  }
});


App.TODOItemComponent =
Ember.Component.extend({
  tagName: 'li',
  classNameBindings: ['isDone']
});
```

Output

Run with JS Auto-run JS

Todos

- ☐ Learn Ember.js
- ☒ Make awesome web apps

 tomdale

Handling User Interaction With Actions

Handling User Interaction With Actions

- Components allow you to define controls that you can reuse throughout your application.
- If they're generic enough, they can also be shared with others and used in multiple applications.
- To make a reusable control useful, however, you first need to allow users of your application to interact with it.

Use the `{{action}}` helper

- You can make elements in your component interactive by using the `{{action}}` helper.
- This is the same `{{action}}` helper you use in application templates, but it has an important difference when used inside a component.
- Instead of sending an action to the template's controller, then bubbling up the route hierarchy, actions sent from inside a component are sent directly to the component's `Ember.Component` instance, and do not bubble.

For example, imagine the following component that shows a post's title. When the title is clicked, the entire post body is shown:

```
1 <script type="text/x-handlebars" id="components/post-summary">
2   <h3 {{action "toggleBody"}}>{{title}}</h3>
3   {{#if isShowingBody}}
4     <p>{{{body}}}</p>
5   {{/if}}
6 </script>
```

```
1 App.PostSummaryComponent = Ember.Component.extend({
2   actions: {
3     toggleBody: function() {
4       this.toggleProperty('isShowingBody');
5     }
6   }
7 });
```

```
0.0/handlebars.js"></script>
<script src="http://builds.emberjs.com/ember-
latest.js"></script>
<meta charset=utf-8 />
<title>JS Bin</title>
</head>
<body>
  <script type="text/x-handlebars" data-template-
name="index">
    {{#each}}
      {{post-summary title=title body=body}}
    {{/each}}
  </script>

  <script type="text/x-handlebars"
id="components/post-summary">
    <h3 {{action "toggleBody"}}>{{title}}</h3>
    {{#if isShowingBody}}
      <p>{{{body}}}</p>
    {{/if}}
  </script>
</body>
</html>
```

JavaScript ▾

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

posts = [{
  title: "Rails is omakase",
  body: "There are lots of à la carte software
environments in this world."
}, {
  title: "Broken Promises",
  body: "James Coglán wrote a lengthy article about
Promises in node.js."
}];

App.IndexRoute = Ember.Route.extend({
  model: function() {
    return posts;
  }
});

App.PostSummaryComponent = Ember.Component.extend({
  actions: {
    toggleBody: function() {
      this.toggleProperty('isShowingBody');
    }
  }
});
```

Output

Run with JS

Auto-run JS ☒ ↗

Rails is omakase

There are lots of à la carte software environments in this world.

Broken Promises



pwagenet

The `{{action}}` helper can accept arguments, listen for different event types, control how action bubbling occurs, and more.

For details about using the `{{action}}` helper, see the [Actions section](#) of the Templates chapter.

Sending Actions From Components To Your App

Sending Actions From Components To Your App

- When a component is used inside a template, it has the ability to send actions to that template's controller and routes.
- These allow the component to inform the app when important events, such as the user clicking a particular element in a component, occur.
- Like the `{{action}}` Handlebars helper, actions sent from components first go to the template's controller
- If not handled there, it will bubble up the route's hierarchy

Actions Need to be Specified

- Components are designed to be reusable across different parts of your application.
- In order to be reusable, it's important that the actions that your components send be specified when the component is used in a template.
- Instead of sending a generic `click` action, you want to specify which click action it should be.
- Luckily, components have a `sendAction()` method that allows them to send actions specified when used in a template.

Sending a Primary Action

- Many components only send one kind of action.
- For example, a button component might send an action when it is clicked on; this is the *primary action*.

To set a component's primary action, set its `action` attribute in Handlebars:

```
1 {{my-button action="showUser"}}
```

This tells the `my-button` component that it should send the `showUser` action when it triggers its primary action.

So how do you trigger sending a component's primary action? After the relevant event occurs, you can call the `sendAction()` method without arguments:

```
1 App.MyButtonComponent = Ember.Component.extend({  
2   click: function() {  
3     this.sendAction();  
4   }  
5 });
```

In the above example, the `my-button` component will send the `showUser` action when the component is clicked.

Sending Parameters with an Action

- You may want to provide additional context to the route or controller handling an action.
- For example, a button component may want to tell a controller not only that *an* item was deleted, but also *which* item.

To send parameters with the primary action, call `sendAction()` with the string `'action'` as the first argument and any additional parameters following it:

```
1 this.sendAction('action', param1, param2);
```

For example, imagine we're building a todo list that allows the user to delete a todo:

```
1 App.IndexRoute = Ember.Route.extend({
2   model: function() {
3     return {
4       todos: [{
5         title: "Learn Ember.js"
6       }, {
7         title: "Walk the dog"
8       }]
9     };
10  },
11
12  actions: {
13    deleteTodo: function(todo) {
14      var todos = this.modelFor('index').todos;
15      todos.removeObject(todo);
16    }
17  }
18 });
```

```
1 {{! index.handlebars }}
2
3 {{#each todo in todos}}
4   <p>{{todo.title}} <button {{action "deleteTodo" todo}}>Delete</button></p>
5 {{/each}}
```

We want to update this app so that, before actually deleting a todo, the user must confirm that this is what they intended.

In the component, when triggering the primary action, we'll pass an additional argument that the component user can specify:

```
1 App.ConfirmButtonComponent = Ember.Component.extend({
2   actions: {
3     showConfirmation: function() {
4       this.toggleProperty('isShowingConfirmation');
5     },
6
7     confirm: function() {
8       this.toggleProperty('isShowingConfirmation');
9       this.sendAction('action', this.get('param'));
10    }
11  }
12 });
```

```
1 {{#if isShowingConfirmation}}
2   <button {{action "confirm"}}>Click again to confirm</button>
3 {{else}}
4   <button {{action "showConfirmation"}}>{{title}}</button>
5 {{/if}}
```

Now we can update our initial template and replace the `{{action}}` helper with our new component:

```
1   {{#each todo in todos}}
2     <p>{{todo.title}} {{confirm-button title="Delete" action="deleteTodo" param=todo}}</p>
3   {{/each}}
```

```

<title>JS Bin</title>
</head>
<body>
  <script type="text/x-handlebars" data-template-
name="index">
    {{#each todo in todos}}
      <p>{{todo.title}} {{confirm-button
title="Delete" action="deleteTodo" param=todo}}</p>
    {{/each}}
  </script>

  <script type="text/x-handlebars" data-template-
name="components/confirm-button">
    {{#if isShowingConfirmation}}
      <button {{action "confirm"}}>Click again to
confirm</button>
    {{else}}
      <button {{action "showConfirmation"}}>{{title}}
</button>
    {{/if}}
  </script>
</body>
</html>

```

```

App.IndexRoute = Ember.Route.extend({
  model: function() {
    return {
      todos: [{
        title: "Learn Ember.js"
      }, {
        title: "Walk the dog"
      }]
    };
  },

  actions: {
    deleteTodo: function(todo) {
      var todos = this.modelFor('index').todos;
      todos.removeObject(todo);
    }
  }
});

App.ConfirmButtonComponent = Ember.Component.extend({
  actions: {
    showConfirmation: function() {
      this.toggleProperty('isShowingConfirmation');
    },

    confirm: function() {
      this.toggleProperty('isShowingConfirmation');
      this.sendAction('action', this.get('param'));
    }
  }
});

```

Output

Run with JS

Auto-run JS ☒ ↗

Learn Ember.js

Click again to confirm

Walk the dog

Delete



tomdale

Sending Multiple Actions

- Depending on the complexity of your component, you may need to let users specify multiple different actions for different events that your component can generate.
- For example, imagine that you're writing a form component that the user can either submit or cancel. Depending on which button the user clicks, you want to send a different action to your controller or route.

You can specify *which* action to send by passing the name of the event as the first argument to `sendAction()`. For example, you can specify two actions when using the form component:

```
1 {{user-form submit="createUser" cancel="cancelUserCreation"}}
```

In this case, you can send the `createUser` action by calling `this.sendAction('submit')`, or send the `cancelUserCreation` action by calling `this.sendAction('cancel')`.

```

<meta charset=utf-8 />
<title>JS Bin</title>
</head>
<body>
  <script type="text/x-handlebars" data-template-
name="index">
    <h1>Create New User</h1>
    {{user-form submit="createUser"
cancel="cancelUserCreation" submitTitle="Create
User"}}
  </script>

  <script type="text/x-handlebars" data-template-
name="components/user-form">
    <form {{action "submit" on="submit"}}>
      <p><label>Name {{input type="text" value=name}}
</label></p>
      <p><label>Bio {{textarea value=bio}}</label></p>

      <button {{action "cancel"}}>Cancel</button>
      <input type="submit" {{bindAttr
value=submitTitle}}>
    </form>
  </script>
</body>
</html>

```

```

App = Ember.Application.create();

App.IndexController = Ember.ObjectController.extend({
  actions: {
    createUser: function(user) {
      alert("Created user " + user.name + " with bio "
+ user.bio + ".");
    },
    cancelUserCreation: function() {
      alert("Canceled user creation.");
    }
  }
});

App.UserFormComponent = Ember.Component.extend({
  actions: {
    submit: function() {
      this.sendAction('submit', {
        name: this.get('name'),
        bio: this.get('bio')
      });
    },
    cancel: function() {
      this.sendAction('cancel');
    }
  }
});

```

Output

Run with JS

Auto-run JS ☒ ↗

Create New User

Name

Bio

Cancel

Create User



tomdale

Actions That Aren't Specified

If someone using your component does not specify an action for a particular event, calling `sendAction()` has no effect.

For example, if you define a component that triggers the primary action on click:

```
1 App.MyButtonComponent = Ember.Component.extend({  
2   click: function() {  
3     this.sendAction();  
4   }  
5 });
```


Using this component without assigning a primary action will have no effect if the user clicks it:

```
1 {{my-button}}
```

Thinking About Component Actions

- In general, you should think of component actions as translating a *primitive event* (like a mouse click or an `<audio>` element's pause event) into actions that have meaning within your application.
- This allows your routes and controllers to implement action handlers with names like `deleteTodo` or `songDidPause` instead of vague names like `click` or `pause` that may be ambiguous to other developers when read out of context.

Thinking About Component Actions, Cont'd

- Another way to think of component actions is as the *public API* of your component.
- Thinking about which events in your component can trigger actions in their application is the primary way other developers will use your component.
- In general, keeping these events as generic as possible will lead to components that are more flexible and reusable.

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

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

COEN 168/268

Mobile Web Application Development

Ember Components

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University