COEN 168/268

Mobile Web Application Development

Ember Templates

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University

Starting at: http://emberjs.com/guides/templates/the-application-template/

The `Application` template

The `Application` template

The `application` template is the default template that is rendered when your application starts.

You should put your header, footer, and any other decorative content here. Additionally, you should have at least one `{{outlet}}`:

a placeholder that the router will fill in with the appropriate template, based on the current URL.

Here's an example template

```
<header>
  <h1>Igor's Blog</h1>
</header>

<div>
  {{outlet}}
</div>

<footer>
  &copy;2013 Igor's Publishing, Inc.
</footer>
```

The header and footer will always be displayed on screen, but the contents of the `<div>` will change depending on if the user is currently at `/posts` or `/posts/15`, for example.

Defining The `Application` Template

- If you are keeping your templates in HTML, create a `<script>` tag without a template name.
- Ember will use the template without a name as the application template.
- It will automatically be compiled and appended to the screen.

An Example `application` Template

```
<script type="text/x-handlebars">
  <div>
    {{outlet}}
  </div>
</script>
```

Also, if you're using build tools to load your templates, make sure you name the template `application`.

`Handlebars` Basics

`Handlebars` Templating Language

- Ember.js uses the Handlebars templating library to power your app's user interface.
- `Handlebars` templates are **just like regular HTML**, but gives you the ability to embed expressions that alter what is displayed.
- Think of your Handlebars templates as an HTML-like DSL for describing the user interface of your app.
- Once you've told Ember.js to render a given template on the screen, you don't need to write any additional code to make sure it keeps up-to-date.

Defining The `Application` Template

If you're **not** using build tools, you can define your application's main template inside your HTML by putting it inside a `<script>` tag, without a **name**:

```html
<html>
  <body>
    <script type="text/x-handlebars">
      Hello, <strong>{{firstName}} {{lastName}}</strong>!
    </script>
  </body>
</html>
```

Defining Other `Handlebar Templates`

- You can also define templates by name that can be used later.
- To tell Ember.js to save the template for later, instead of displaying it immediately, you can add the `data-template-name` attribute:

```
<html>
  <head>
    <script type="text/x-handlebars" data-template-name="say-hello">
      <div class="my-cool-control">{{name}}</div>
    </script>
  </head>
</html>
```

Handlebars Expressions

- Each template has an associated *controller*: this is where the template finds the properties that it displays.
- You can display a property from your controller by wrapping the property name in curly {{braces}}
- Looks up the `firstName` and `lastName` properties from the controller, insert them into the HTML described in the template

```
Hello, <strong>{{firstName}} {{lastName}}</strong>!
```

The ApplicationController

By default, your top-most application template is bound to your ApplicationController:

```
App.ApplicationController = Ember.Controller.extend({
  firstName: "Trek",
  lastName: "Glowacki"
});
```

The above template and controller would combine to display the following rendered HTML:

```
Hello, <strong>Trek Glowacki</strong>!
```

`Handlebars` Expressions Are Bindings Aware

- That means that if the values used by your templates ever change, your HTML will be updated automatically.
- As your application grows in size, it will have many templates, each bound to different controllers.

`Handlebars` Conditionals

`Handlebars` Conditionals

Sometimes you may only want to display part of your template if a property exists. We can use the `{{#if}}` helper to conditionally render a block:

```
{{#if person}}
  Welcome back, <b>{{person.firstName}} {{person.lastName}}</b>!
{{/if}}
```

Handlebars will not render the block if the argument passed evaluates to `false`, `undefined`, `null` or `[]` (i.e., any "falsy" value).

The {{#if}} and {{else}} conditionals

If the expression evaluates to falsy, we can also display an alternate template using {{else}}:

```
{{#if person}}
  Welcome back, <b>{{person.firstName}} {{person.lastName}}</b>!
{{else}}
  Please log in.
{{/if}}
```

Also, `{{#unless}}`

To only render a block if a value is falsy, use `{{#unless}}`:

```
{{#unless hasPaid}}
  You owe: ${{total}}
{{/unless}}
```

Block Expressions

`{{#if}}` and `{{#unless}}` are examples of block expressions.

- These allow you to invoke a helper with a portion of your template.
- Block expressions look like normal expressions except that they contain a hash (#) before the helper name, and require a closing expression.

Displaying a List of Items

Looping Using {{#each}}

If you need to enumerate over a list of objects, use Handlebars' `{{#each}}` helper:

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

The template inside of the `{{#each}}` block will be repeated once for each item in the array, with the context of the template set to the current item.

Looping Using {{#each}}, Cont'd

The above example will print a list like this:

```html
<ul>
  <li>Hello, Yehuda!</li>
  <li>Hello, Tom!</li>
  <li>Hello, Trek!</li>
</ul>
```

`{{#each}}` Is Bindings Aware

- Like everything in Handlebars, the `{{#each}}` helper is bindings-aware.
- If your application adds a new item to the array, or removes an item, the DOM will be updated without having to write any code.

An Alternative Way To Use {{#each}}

- Does not change the scope of its inner template.
- This is useful for cases where you need to access a property from the outer scope within the loop.

```
{{name}}'s Friends

<ul>
  {{#each friend in friends}}
    <li>{{name}}'s friend {{friend.name}}</li>
  {{/each}}
</ul>
```

An Alternative Way To Use `{{#each}}`, Cont'd

This would print a list like this:

```
Trek's Friends

<ul>
  <li>Trek's friend Yehuda</li>
  <li>Trek's friend Tom!</li>
</ul>
```

The `{{#each}}` helper can have a matching `{{else}}`.

The contents of this block will render if the collection is empty:

```
{{#each people}}
  Hello, {{name}}!
{{else}}
  Sorry, nobody is here.
{{/each}}
```

Changing Scope

Sometimes you may want to invoke a section of your template with a different context.

For example, instead of repeating a long path, like in this example:

```
Welcome back, <b>{{person.firstName}} {{person.lastName}}</b>!
```

We can use the `{{#with}}` helper to clean it up:

```
{{#with person}}
  Welcome back, <b>{{firstName}} {{lastName}}</b>!
{{/with}}
```

`{{#with}}` changes the *context* of the block

- The context, by default, is the template's controller.
- By using the `{{#with}}` helper, you can change the context of all of the Handlebars expressions contained inside the block.

Note: it's possible to store the context within a variable for nested usage using the "as" keyword:

```
{{#with person as user}}
  {{#each book in books}}
    {{user.firstName}} has read {{book.name}}!
  {{/each}}
{{/with}}
```

Binding HTML Element Attributes Using `{{bind-attr}}`

You Can Bind to HTML Element Attributes

For example, imagine your controller has a property that contains a URL to an image:

```
<div id="logo">
  <img {{bind-attr src=logoUrl}} alt="Logo">
</div>
```

This generates the following HTML:

```
<div id="logo">
 <img src="http://www.example.com/images/logo.png" alt="Logo">
</div>
```

Using `{{bind-attr}}` With Booleans

If you use `{{bind-attr}}` with a Boolean value, it will add or remove the specified attribute.

For example, given this template:

```
<input type="checkbox" {{bind-attr disabled=isAdministrator}}>
```

Using `{{bind-attr}}` With Booleans

If `isAdministrator` is `true`:

```
<input type="checkbox" disabled>
```

If `isAdministrator` is `false`:

```
<input type="checkbox">
```

Adding data attributes

By default, view helpers do not accept *data attributes*.

```
{{#link-to "photos" data-toggle="dropdown"}}Photos{{/link-to}}

{{input type="text" data-toggle="tooltip" data-placement="bottom" title="Name"}}
```

renders the following HTML:

```
<a id="ember239" class="ember-view" href="#/photos">Photos</a>

<input id="ember257" class="ember-view ember-text-field" type="text" title="Name">
```

How to Enable *data attributes*

Two ways:

1. Add an attribute binding on the view, e.g. `Ember.LinkView` or `Ember.TextField` for the specific attribute

2. Automatically bind data attributes on the base view

Adding An Attribute Binding

```
Ember.LinkView.reopen({
  attributeBindings: ['data-toggle']
});
Ember.TextField.reopen({
  attributeBindings: ['data-toggle', 'data-placement']
});
```

```
<a id="ember240" class="ember-view" href="#/photos" data-toggle="dropdown">Photos</a>

<input id="ember259" class="ember-view ember-text-field"
       type="text" data-toggle="tooltip" data-placement="bottom" title="Name">
```

Bind Data Attributes To Base View

```
Ember.View.reopen({
  init: function() {
    this._super();
    var self = this;

    // bind attributes beginning with 'data-'
    Em.keys(this).forEach(function(key) {
      if (key.substr(0, 5) === 'data-') {
        self.get('attributeBindings').pushObject(key);
      }
    });
  }
});
```

Now you can add as many `data-attributes` as you want without having to specify them by name.

Binding Element Class Names

An HTML element's `class` attribute can be bound like any other attribute

```
<div {{bind-attr class="priority"}}>
  Warning!
</div>
```

If the controller's `priority` property is `"p4"`, this template will emit the following HTML:

```
<div class="p4">
  Warning!
</div>
```

Binding to Boolean Values

If the value to which you bind is a Boolean, Ember.js will apply the *dasherized* version of the property name as a class:

```
<div {{bind-attr class="isUrgent"}}>
  Warning!
</div>
```

Binding to Boolean Values, Cont'd

If `isUrgent` is true, this emits the following HTML:

```html
<div class="is-urgent">
  Warning!
</div>
```

If `isUrgent` is false, no class name is added:

```html
<div>
  Warning!
</div>
```

If you want to explicitly provide a class name (instead of Ember.js dasherizing the property name), use the following syntax:

```
<div {{bind-attr class="isUrgent:urgent"}}>
  Warning!
</div>
```

Instead of the dasherized name, this will produce:

```
<div class="urgent">
  Warning!
</div>
```

You can also specify a class name to add when the property is `false`:

```
<div {{bind-attr class="isEnabled:enabled:disabled"}}>
  Warning!
</div>
```

In this case, if the `isEnabled` property is `true`, the `enabled` class will be added. If the property is `false`, the class `disabled` will be added.

This syntax can also be used to add a class if a property is `false` and remove it if the property is `true`, so this:

```
<div {{bind-attr class="isEnabled::disabled"}}>
  Warning!
</div>
```

Will add the class `disabled` when `isEnabled` is `false` and add no class if `isEnabled` is `true`.

Static Classes

If you need an element to have a combination of static and bound classes, you should include the static class in the list of bound properties, prefixed by a colon:

```
<div {{bind-attr class=":high-priority isUrgent"}}>
  Warning!
</div>
```

This will add the literal `high-priority` class to the element:

```
<div class="high-priority is-urgent">
  Warning!
</div>
```

However, bound class names and static class names cannot be combined!

The following example **will not work**:

```
<div class="high-priority" {{bind-attr class="isUrgent"}}>
  Warning!
</div>
```

You have been warned

Binding Multiple Classes

Unlike other element attributes, you can bind multiple classes:

```
<div {{bind-attr class="isUrgent priority"}}>
  Warning!
</div>
```

This works how you would expect, applying the rules described above in order:

```
<div class="is-urgent p4">
  Warning!
</div>
```

Links

The {{link-to}} Helper

You create a link to a route using the {{link-to}} helper.

```
App.Router.map(function() {
  this.resource("photos", function(){
    this.route("edit", { path: "/:photo_id" });
  });
});
```

```
<!-- photos.handlebars -->
<ul>
{{#each photo in photos}}
  <li>{{#link-to 'photos.edit' photo}}{{photo.title}}{{/link-to}}</li>
{{/each}}
</ul>
```

If the model for the `photos` template is a list of three photos, the rendered HTML would look something like this:

```
<ul>
  <li><a href="/photos/1">Happy Kittens</a></li>
  <li><a href="/photos/2">Puppy Running</a></li>
  <li><a href="/photos/3">Mountain Landscape</a></li>
</ul>
```

One Nice Thing:

- When the rendered link matches the current route
- And the same object instance is passed into the helper
- Then the link is given `class="active"`.

This makes your selected CSS styling a lot easier

The `{{link-to}}` helper takes:

- The name of a route. In this example, it would be `index`, `photos`, or `photos.edit`.
- At most one model for each dynamic segment.
- An optional title which will be bound to the `a` title attribute

```
{{! photos.handlebars }}

{{#link-to 'photo.edit' 1}}
  First Photo Ever
{{/link-to}}
```

Example for Multiple Segments

If the route is nested, you can supply a model or an identifier for each dynamic segment.

```
App.Router.map(function() {
  this.resource("photos", function(){
    this.resource("photo", { path: "/:photo_id" }, function(){
      this.route("comments");
      this.route("comment", { path: "/comments/:comment_id" });
    });
  });
});
```

Example for Multiple Segments, Cont'd

```
<!-- photoIndex.handlebars -->

<div class="photo">
  {{body}}
</div>

<p>{{#link-to 'photo.comment' primaryComment}}Main Comment{{/link-to}}</p>
```

If you specify only one model, it will represent the innermost dynamic segment `:comment_id`.
The `:photo_id` segment will use the current photo.

Alternatively, you could pass both a photo and a comment to the helper:

```
<p>
  {{#link-to 'photo.comment' 5 primaryComment}}
    Main Comment for the Next Photo
  {{/link-to}}
</p>
```

Alternatively, you could pass both a photo and a comment to the helper, cont'd

- The model hook for `PhotoRoute` will run with `params.photo_id = 5`.
- The `model` hook for `CommentRoute` *won't* run since you supplied a model object for the `comment` segment.
- The comment's id will populate the url according to `CommentRoute`'s `serialize` hook.

Adding additional attributes on a link

- When generating a link you might want to set additional attributes for it.
- You can do this with additional arguments to the `link-to` helper:

```
<p>
  {{link-to 'Edit this photo' 'photo.edit' photo class="btn btn-primary"}}
</p>
```

Adding additional attributes on a link, cont'd

- Many of the common HTML properties you would want to use like `class`, and `rel` will work.
- When adding class names, Ember will also apply the standard `ember-view` and possibly `active` class names.

Replacing history entries

- The default behavior for `link-to` is to add entries to the browser's history when transitioning between the routes.
- However, to replace the current entry in the browser's history you can use the `replace=true` option:

```
<p>
  {{#link-to 'photo.comment' 5 primaryComment replace=true}}
    Main Comment for the Next Photo
  {{/link-to}}
</p>
```

Actions!

The `{{action}}` Helper

- Your app will often need a way to let users interact with controls that change application state.
- You can use the `{{action}}` helper to make an HTML element clickable.
- When a user clicks the element, the named event will be sent to your application.

An {{action}} Helper Example

```
<!-- post.handlebars -->

<div class='intro'>
  {{intro}}
</div>

{{#if isExpanded}}
  <div class='body'>{{body}}</div>
  <button {{action 'contract'}}>Contract</button>
{{else}}
 <button {{action 'expand'}}>Show More...</button>
{{/if}}
```

An {{action}} Helper Example, Cont'd

```
App.PostController = Ember.ObjectController.extend({
  // initial value
  isExpanded: false,

  actions: {
    expand: function() {
      this.set('isExpanded', true);
    },

    contract: function() {
      this.set('isExpanded', false);
    }
  }
});
```

Action Bubbling

- By default, the `{{action}}` helper triggers a method on the template's controller.
- If the controller does not implement a method with the same name as the action in its actions object, the action will be sent to the router.
- There, the currently active leaf route will be given a chance to handle the action.

`actions` hash

- Routes and controllers that handle actions **must place action handlers inside an `actions` hash**.
- Even if a route has a method with the same name as the actions, it will not be triggered unless it is inside an `actions` hash.

`actions` hash, continued

```
App.PostRoute = Ember.Route.extend({
  actions: {
    expand: function() {
      this.controller.set('isExpanded', true);
    },

    contract: function() {
      this.controller.set('isExpanded', false);
    }
  }
});
```

As you can see in this example, the action handlers are called such that when executed, `this` is the route, not the `actions` hash.

Continuing Bubbling

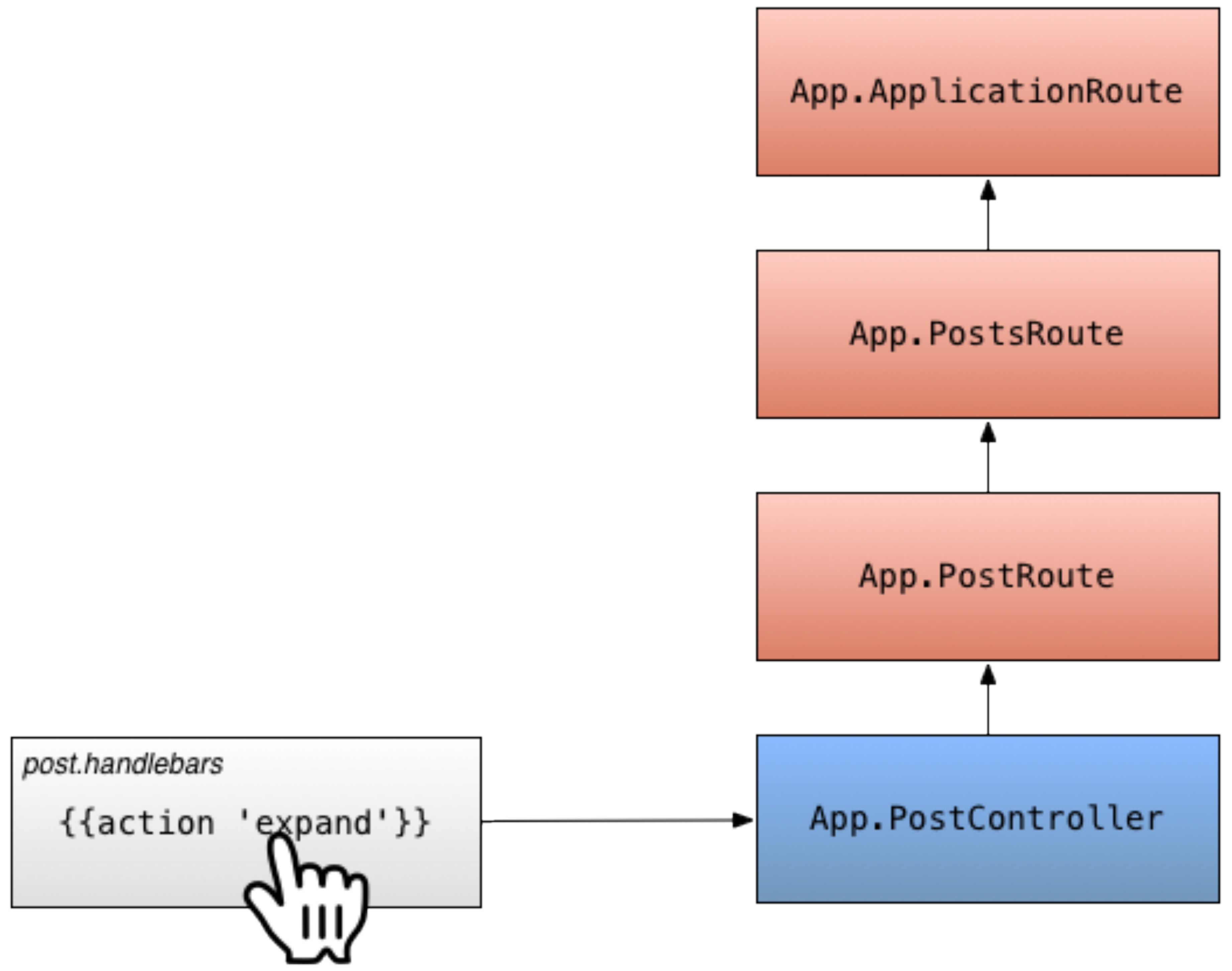
To continue bubbling the action, you must return true from the handler:

```
App.PostRoute = Ember.Route.extend({
  actions: {
    expand: function() {
      this.controller.set('isExpanded', true);
    },

    contract: function() {
      // ...
      if (actionShouldAlsoBeTriggeredOnParentRoute) {
        return true;
      }
    }
  }
});
```

Bubbling Upwards...

- If neither the template's controller nor the currently active route implements a handler, the action will continue to bubble to any parent routes.
- Ultimately, if an `ApplicationRoute` is defined, it will have an opportunity to handle the action.
- When an action is triggered, but no matching action handler is implemented on the controller, the current route, or any of the current route's ancestors, an error will be thrown.


App.ApplicationRoute
App.PostsRoute
App.PostRoute
post.handlebars
{{action 'expand'}}
App.PostController

Actions are based on where you are in the app

- Due to bubbling upwards in the controller hierarchy you can have different behavior based on where you are in the app
- For example, you might want to have a button in a sidebar that does one thing if you are somewhere inside of the `/posts` route, and another thing if you are inside of the `/about` route.

Action Parameters

- You can optionally pass arguments to the action handler.
- Any values passed to the `{{action}}` helper after the action name will be passed to the handler as arguments.

An Action Parameter Example

```
<p><button {{action "select" post}}>✓</button> {{post.title}}</p>
```

The controller's `select` action handler would be called with a single argument containing the post model:

```
App.PostController = Ember.ObjectController.extend({
  actions: {
    select: function(post) {
      console.log(post.get('title'));
    }
  }
});
```

Specifying the Type of Event

By default, the `{{action}}` helper listens for click events and triggers the action when the user clicks on the element.

You can specify an alternative event by using the on option.

```
<p>
  <button {{action "select" post on="mouseUp"}}>✓</button>
  {{post.title}}
</p>
```

Specifying Whitelisted Modifier Keys

- By default the `{{action}}` helper will ignore click events with pressed modifier keys.
- You can supply an `allowedKeys` option to specify which keys should not be ignored.

```
<script type="text/x-handlebars" data-template-name='a-template'>
  <div {{action 'anActionName' allowedKeys="alt"}}>
    click me
  </div>
</script>
```

This way the `{{action}}` will fire when clicking with the alt key pressed down.

Stopping Event Propagation

- By default, the `{{action}}` helper allows events it handles to bubble up to parent DOM nodes.
- If you want to stop propagation, you can disable propagation to the parent node.

```
{{#link-to 'post'}}
 Post
 <button {{action 'close' bubbles=false}}>✗</button>
{{/link-to}}
```

Stopping Event Propagation, Cont'd

Without `bubbles=false`, if the user clicked on the button, Ember.js will trigger the action, and then the browser will propagate the click to the link.

With `bubbles=false`, Ember.js will stop the browser from propagating the event.

Specifying a Target

- By default, the `{{action}}` helper will send the action to the view's target, which is generally the view's controller.
- You can specify an alternative target by using the `target` option.
- This is most commonly used to send actions to a view instead of a controller.

```
<p>
  <button {{action "select" post target="view"}}>✓</button>
  {{post.title}}
</p>
```

Specifying a Target, Cont'd

You would handle this in an `actions` hash on your view.

```
App.PostsIndexView = Ember.View.extend({
  actions: {
    select: function(post) {
      // do your business.
    }
  }
});
```

Input Helpers

- The `{{input}}` and `{{textarea}}` helpers in Ember.js are the easiest way to create common form controls.
- The `{{input}}` helper wraps the built-in [Ember.TextField][1] and [Ember.Checkbox][2] views
- The `{{textarea}}` wraps [Ember.TextArea][3].
- Using these helpers, you can create these views with declarations almost identical to how you'd create a traditional `<input>` or `<textarea>` element.

Text fields

```
{{input value="http://www.facebook.com"}}
```

Will become:

```
<input type="text" value="http://www.facebook.com"/>
```

You can pass these `<input>` attributes within the input helper:

- `value`
- `size`
- `name`
- `pattern`
- `placeholder`
- `disabled`
- `maxlength`

- If these attributes are set to a quoted string, their values will be set directly on the element.
- When left unquoted, these values will be bound to a property on the template's current rendering context:

```
{{input type="text" value=firstName disabled=entryNotAllowed size="50"}}
```

Will bind the `disabled` attribute to the value of `entryNotAllowed` in the current context.

Checkboxes

You can also use the `{{input}}` helper to create a checkbox by setting its `type`:

```
{{input type="checkbox" name="isAdmin" checked=isAdmin}}
```

Checkboxes support the following properties:

- `checked`
- `disabled`
- `tabindex`
- `indeterminate`
- `name`

Which can be bound or set as described in the previous section.

Text Areas

```
{{textarea value=name cols="80" rows="6"}}
```

Will bind the value of the text area to `name` on the current context.

`{{textarea}}` supports binding and/or setting the following properties:

- `rows`
- `cols`
- `placeholder`
- `disabled`
- `maxlength`
- `tabindex`

Development Helpers

Development Helpers

- Handlebars and Ember come with a few helpers that can make developing your templates a bit easier.
- These helpers make it simple to output variables into your browser's console, or activate the debugger from your templates.

Logging

The `{{log}}` helper makes it easy to output variables or expressions in the current rendering context into your browser's console:

```
{{log 'Name is:' name}}
```

The `{{log}}` helper also accepts primitive types such as strings or numbers.

Adding a breakpoint

- The `{{debugger}}` helper provides a handlebars equivalent to JavaScript's `debugger` keyword.
- It will halt execution inside the debugger helper and give you the ability to inspect the current rendering context:

```
{{debugger}}
```

Just before the helper is invoked two useful variables are defined:

- `templateContext` The current context that variables are fetched from. This is likely a controller.
- `typeOfTemplateContext` A string describing what the templateContext is.

- For example, if you are wondering why a specific variable isn't displaying in your template, you could use the `{{debugger}}` helper.
- When the breakpoint is hit, you can use the `templateContext` in your console to lookup properties:

```
> templateContext.get('name')
"Bruce Lee"
```

Rendering With Helpers

Rendering With Helpers

Ember.js provides several helpers that allow you to render other views and templates in different ways.

The {{partial}} Helper

- {{partial}} takes the template to be rendered as an argument, and renders that template in place.
- {{partial}} does not change context or scope. It simply drops the given template into place with the current scope.

```
<script type="text/x-handlebars" data-template-name='_author'>
  Written by {{author.firstName}} {{author.lastName}}
</script>

<script type="text/x-handlebars" data-template-name='post'>
  <h1>{{title}}</h1>
  <div>{{body}}</div>
  {{partial "author"}}
</script>
```

The {{partial}} Helper, Cont'd

```
<div>
  <h1>Why You Should Use Ember.JS</h1>
  <div>Because it's awesome!</div>
  Written by Yehuda Katz
</div>
```

The partial's `data-template-name` must start with an underscore (e.g. `data-template-name='_author'` or `data-template-name='foo/_bar'`)

The {{view}} Helper

This helper works like the partial helper, except instead of providing a template to be rendered within the current template, you provide a view class. The view controls what template is rendered.

```
App.AuthorView = Ember.View.extend({
  // We are setting templateName manually here to the default value
  templateName: "author",

  // A fullName property should probably go on App.Author,
  // but we're doing it here for the example
  fullName: (function() {
    return this.get("author").get("firstName") + " " + this.get("author").get("lastName");
  }).property("firstName","lastName")
})
```

The {{view}} Helper, Cont'd

```
<script type="text/x-handlebars" data-template-name='author'>
  Written by {{view.fullName}}
</script>

<script type="text/x-handlebars" data-template-name='post'>
  <h1>{{title}}</h1>
  <div>{{body}}</div>
  {{view "author"}}
</script>
```

```
<div>
  <h1>Why You Should Use Ember.JS</h1>
  <div>Because it's awesome!</div>
  Written by Yehuda Katz
</div>
```

When using `{{partial "author"}}`:

- No instance of App.AuthorView will be created
- The given template will be rendered

When using `{{view "author"}}`:

- An instance of App.AuthorView will be created
- It will be rendered here, using the template associated with that view (the default template being "author")

The `{{render}}` Helper

`{{render}}` takes two parameters:

- The first parameter describes the context to be setup
- The optional second parameter is a model, which will be passed to the controller if provided

The `{{render}}` Helper

`{{render}}` does several things:

- When no model is provided it gets the singleton instance of the corresponding controller
- When a model is provided it gets a unique instance of the corresponding controller
- Renders the named template using this controller
- Sets the model of the corresponding controller

Modifying the post / author example slightly:

```
<script type="text/x-handlebars" data-template-name='author'>
  Written by {{firstName}} {{lastName}}.
  Total Posts: {{postCount}}
</script>

<script type="text/x-handlebars" data-template-name='post'>
  <h1>{{title}}</h1>
  <div>{{body}}</div>
  {{render "author" author}}
</script>
```

```
App.AuthorController = Ember.ObjectController.extend({
  postCount: function() {
    return this.get("model.posts.length");
  }.property("model.posts.[]")
})
```

In this example, render will:

- Get an instance of App.AuthorView if that class exists, otherwise uses a default generated view
- Use the corresponding template (in this case the default of "author")
- Get (or generate) the singleton instance of AuthorController
- Set the AuthorController's model to the 2nd argument passed to render, here the author field on the post
- Render the template in place, with the context created in the previous steps.

`{{render}}` does not require the presence of a matching route.

`{{render}}` is similar to `{{outlet}}`. Both tell Ember.js to devote this portion of the page to something.

`{{outlet}}`: The router determines the route and sets up the appropriate controllers/views/models.
`{{render}}`: You specify (directly and indirectly) the appropriate controllers/views/models.

Note: `{{render}}` cannot be called multiple times for the same route when not specifying a model.

Writing Helpers

Writing Helpers

- Sometimes, you may use the same HTML in your application multiple times.
- In those cases, you can register a custom helper that can be invoked from any Handlebars template.

A Helper Example

For example, imagine you are frequently wrapping certain values in a <span> tag with a custom class. You can register a helper from your JavaScript like this:

```
Ember.Handlebars.helper('highlight', function(value, options) {
  var escaped = Handlebars.Utils.escapeExpression(value);
  return new Ember.Handlebars.SafeString('<span class="highlight">' + escaped + '</span>');
});
```

If you return HTML from a helper, and you don't want it to be escaped, make sure to return a new `SafeString`. Make sure you first escape any user data!

A Helper Example, Continued

In your Handlebars templates, you can now invoke this helper:

```
{{highlight name}}
```

and it will output the following:

```
<span class="highlight">Peter</span>
```

If the `name` property on the current context changes, Ember.js will automatically execute the helper again and update the DOM with the new value.

Dependencies

- Imagine you want to render the full name of an `App.Person`.
- In this case, you will want to update the output if the person itself changes, or if the `firstName` or `lastName` properties change.

```
Ember.Handlebars.helper('fullName', function(person) {
  return person.get('firstName') + ' ' + person.get('lastName');
}, 'firstName', 'lastName');
```

You would use the helper like this:

```
{{fullName person}}
```

Now, whenever the context's person changes, or when any of the *dependent keys* change, the output will automatically update.

Both the path passed to the `fullName` helper and its dependent keys may be full *property paths* (e.g. `person.address.country`).

Custom View Helpers

You may also find yourself rendering your view classes in multiple places using the `{{view}}` helper. In this case, you can save yourself some typing by registering a custom view helper.

For example, let's say you have a view called `App.CalendarView`.

You can register a helper like this:

```
Ember.Handlebars.helper('calendar', App.CalendarView);
```

Using `App.CalendarView` in a template then becomes as simple as:

```
{{calendar}}
```

Which is functionally equivalent to, and accepts all the same arguments as:

```
{{view App.CalendarView}}
```

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

Starting at: http://emberjs.com/guides/templates/the-application-template/

COEN 168/268

Mobile Web Application Development

Ember Templates

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University