Ember.js Guides

Ember.js Guides

The guide for building Ambitious Web Applications

Precious Jahlom Agboado

This book is for sale at http://leanpub.com/emberjsguides

This version was published on 2016-09-09

Contents

Ember.js Guides

Welcome to the Ember.js guides! This documentation will take you from total beginner to Ember expert. It is designed to start from the basics, and slowly increase to more sophisticated concepts until you know everything there is to know about building awesome web applications.

To help you get started, we've also made a 30-minute screencast that will guide you through building a full-featured Ember.js application:

https://www.youtube.com/watch?v=1QHrlFlaXdI[1]

Source code for the app we build in the video is available at https://github.com/tildeio/bloggr-client">https://github.com/tildeio/bloggr-client[2]

Most of these guides are designed to help you start building apps right away. If you'd like to know more about the thinking behind Ember.js, you'll find what you're looking for in the Understanding Ember.js[3] section.

These guides are written in Markdown and are available on GitHub[4], inside the `source/guides` directory. If there is something missing, or you find a typo or mistake, please help us by filing an issue or submitting a pull request. Thanks!

We're excited for all of the great apps you're going to build with Ember.js. To get started, select a topic from the left. They are presented in the order that we think will be most useful to you as you're learning Ember.js, but you can also jump to whatever seems most interesting.

Good luck!

[1] https://www.youtube.com/watch?v=1QHrlFlaXdI

[2] https://github.com/tildeio/bloggr-client">https://github.com/tildeio/bloggr-client

[3] http://emberjs.com/guides/understanding-ember/the-view-layer

[4] https://github.com/emberjs/website/

Getting Started

Welcome to Ember.js! This guide will take you through creating a simple application using Ember.js and briefly explain the core concepts behind the framework. This guide assumes you are already familiar with basic web technologies like JavaScript, HTML, and CSS and development technologies like your browser's web inspector[5].

In this guide we will walk through the steps of building the popular TodoMVC demo application[6].

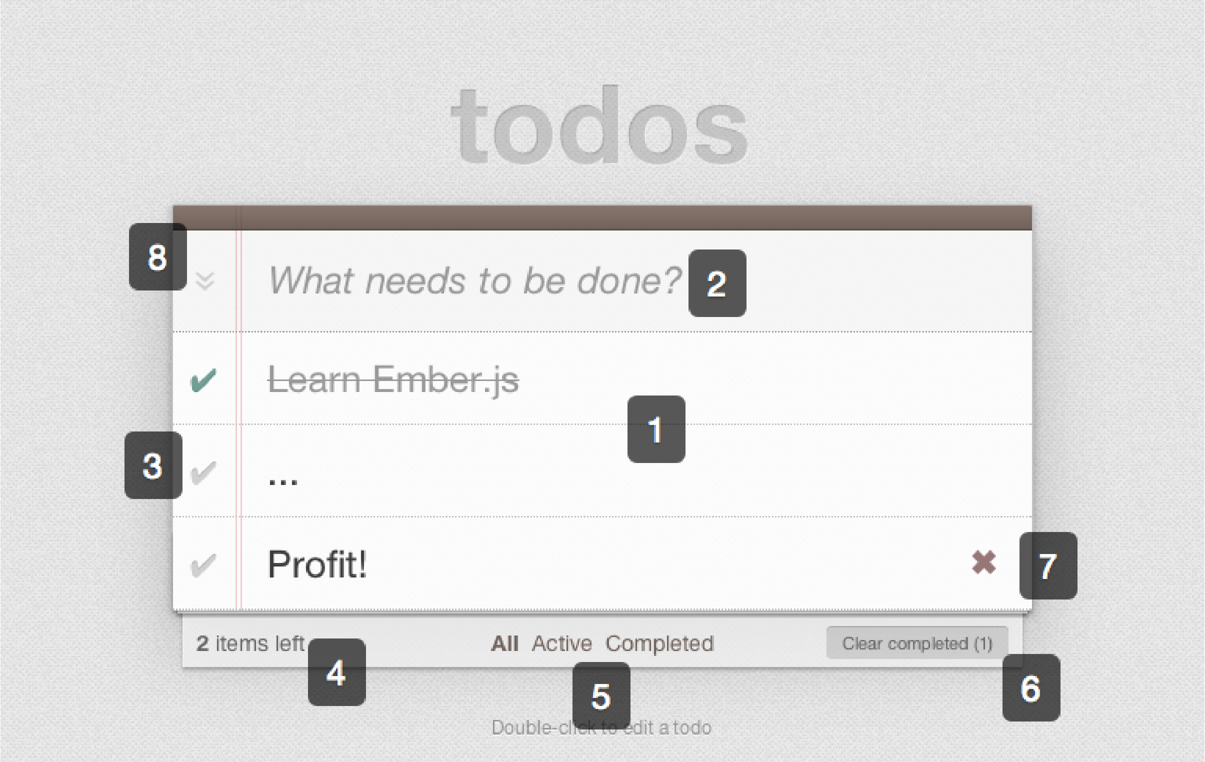
Planning The Application

TodoMVC, despite its small size, contains most of the behaviors typical in modern single page applications. Before continuing, take a moment to understand how TodoMVC works from the user's perspective.

TodoMVC has the following main features:

[5] https://developers.google.com/chrome-developer-tools/

[6] http://todomvc.com



TODO MVC

1. It displays a list of todos for a user to see. This list will grow and shrink as the user adds and removes todos.
2. It accepts text in an `<input>` for entry of new todos. Hitting the `<enter>` key creates the new item and displays it in the list below.
3. It provides a checkbox to toggle between complete and incomplete states for each todo. New todos start as incomplete.
4. It displays the number of incomplete todos and keeps this count updated as new todos are added and existing todos are completed.
5. It provides links for the user to navigate between lists showing all, incomplete, and completed todos.
6. It provides a button to remove all completed todos and informs the user of the number of completed todos. This button will not be visible if there are no completed todos.
7. It provides a button to remove a single specific todo. This button displays as a user hovers over a todo and takes the form of a red X.
8. It provides a checkbox to toggle all existing todos between complete and incomplete states. Further, when all todos are completed this checkbox becomes checked without user interaction.
9. It allows a user to double click to show a textfield for editing a single todo. Hitting the `<enter>` key or moving focus outside of this textfield will persist the changed text.

10. It retains a user's todos between application loads by using the browser's `localstorage` mechanism.

You can interact with a completed version of the application by visiting the TodoMVC site[7].

Creating a Static Mockup

Before adding any code, we can roughly sketch out the layout of our application. In your text editor, create a new file and name it `index.html`. This file will contain the HTML templates of our completed application and trigger requests for the additional image, stylesheet, and JavaScript resources.

To start, add the following text to `index.html`:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Ember.js • TodoMVC</title>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <section id="todoapp">
      <header id="header">
        <h1>todos</h1>
        <input type="text" id="new-todo" placeholder="What needs to be done?" />
      </header>

      <section id="main">
        <ul id="todo-list">
          <li class="completed">
            <input type="checkbox" class="toggle">
            <label>Learn Ember.js</label><button class="destroy"></button>
          </li>
          <li>
            <input type="checkbox" class="toggle">
            <label>...</label><button class="destroy"></button>
          </li>
          <li>
            <input type="checkbox" class="toggle">
            <label>Profit!</label><button class="destroy"></button>
```

[7] http://todomvc.com/architecture-examples/emberjs/

```
          </li>
        </ul>

        <input type="checkbox" id="toggle-all">
      </section>

      <footer id="footer">
        <span id="todo-count">
          <strong>2</strong> todos left
        </span>
        <ul id="filters">
          <li>
            <a href="all" class="selected">All</a>
          </li>
          <li>
            <a href="active">Active</a>
          </li>
          <li>
            <a href="completed">Completed</a>
          </li>
        </ul>

        <button id="clear-completed">
          Clear completed (1)
        </button>
      </footer>
    </section>

    <footer id="info">
      <p>Double-click to edit a todo</p>
    </footer>
  </body>
</html>
```

The associated stylesheet[8] and background image[9] for this project should be downloaded and placed in the same directory as `index.html`

Open `index.html` in your web browser to ensure that all assets are loading correctly. You should see the TodoMVC application with three hard-coded `<li>` elements where the text of each todo will appear.

[8] http://emberjs.com.s3.amazonaws.com/getting-started/style.css

[9] http://emberjs.com.s3.amazonaws.com/getting-started/bg.png

Live Preview

Ember.js • TodoMVC[10]

Additional Resources

- Changes in this step in `diff` format[11]
- TodoMVC stylesheet[12]
- TodoMVC background image[13]

Obtaining Ember.Js And Dependencies

TodoMVC has a few dependencies:

- jQuery[14]
- Handlebars[15]
- Ember.js 1.3[16]
- Ember Data 1.0 beta[17]

For this example, all of these resources should be stored in the folder `js/libs` located in the same location as `index.html`. Update your `index.html` to load these files by placing `<script>` tags just before your closing `</body>` tag in the following order:

```
<!-- ... additional lines truncated for brevity ... -->
  <script src="js/libs/jquery-1.10.2.min.js"></script>
  <script src="js/libs/handlebars-1.0.0.js"></script>
  <script src="js/libs/ember.js"></script>
  <script src="js/libs/ember-data.js"></script>
</body>
<!-- ... additional lines truncated for brevity ... -->
```

Reload your web browser to ensure that all files have been referenced correctly and no errors occur.

If you are using a package manager, such as bower[18], make sure to checkout the Getting Ember[19] guide for info on other ways to get Ember.js (this guide is dependant on ember-data v1.0 or greater so please be sure to use the latest beta).

[10] http://jsbin.com/uduyip

[11] https://github.com/emberjs/quickstart-code-sample/commit/4d91f9fa1f6be4f4675b54babd3074550095c930

[12] http://emberjs.com.s3.amazonaws.com/getting-started/style.css

[13] http://emberjs.com.s3.amazonaws.com/getting-started/bg.png

[14] http://code.jquery.com/jquery-1.10.2.min.js

[15] http://builds.handlebarsjs.com.s3.amazonaws.com/handlebars-1.0.0.js

[16] http://builds.emberjs.com/tags/v1.3.0/ember.js

[17] http://builds.emberjs.com/tags/v1.0.0-beta.5/ember-data.js

[18] http://bower.io

[19] http://emberjs.com/guides/getting-ember

Live Preview

Ember.js • TodoMVC[20]

Additional Resources

- Changes in this step in `diff` format[21]

Next, we will create an Ember.js application, a route (‘/’), and convert our static mockup into a Handlebars template.

Inside your `js` directory, add a file for the application at `js/application.js` and a file for the router at `js/router.js`. You may place these files anywhere you like (even just putting all code into the same file), but this guide will assume you have separated them into their own files and named them as indicated.

Inside `js/application.js` add the following code:

```
window.Todos = Ember.Application.create();
```

This will create a new instance of `Ember.Application` and make it available as a variable named `Todos` within your browser's JavaScript environment.

Inside `js/router.js` add the following code:

```
Todos.Router.map(function() {
  this.resource('todos', { path: '/' });
});
```

This will tell Ember.js to detect when the application's URL matches `'/'` and to render the `todos` template.

Next, update your `index.html` to wrap the inner contents of `<body>` in a Handlebars script tag and include `js/application.js` and `js/router.js` after Ember.js and other javascript dependencies:

[20] http://jsbin.com/ijefig

[21] https://github.com/emberjs/quickstart-code-sample/commit/0880d6e21b83d916a02fd17163f58686a37b5b2c

```
<!-- ... additional lines truncated for brevity ... -->
<body>
  <script type="text/x-handlebars" data-template-name="todos">

    <section id="todoapp">
      {{! ... additional lines truncated for brevity ... }}
    </section>

    <footer id="info">
      <p>Double-click to edit a todo</p>
    </footer>

  </script>

  <!-- ... Ember.js and other javascript dependencies ... -->
  <script src="js/application.js"></script>
  <script src="js/router.js"></script>
</body>
<!-- ... additional lines truncated for brevity ... -->
```

Reload your web browser to ensure that all files have been referenced correctly and no errors occur.

Live Preview

Ember.js • TodoMVC[22]

Additional Resources

- Changes in this step in `diff` format[23]
- Handlebars Guide[24]
- Ember.Application Guide[25]
- Ember.Application API Documentation[26]

Modeling Data

Next we will create a model class to describe todo items.

Create a file at `js/models/todo.js` and put the following code inside:

[22] http://jsbin.com/OKEMIJi

[23] https://github.com/emberjs/quickstart-code-sample/commit/8775d1bf4c05eb82adf178be4429e5b868ac145b

[24] http://emberjs.com/guides/templates/handlebars-basics

[25] http://emberjs.com/guides/application

[26] http://emberjs.com/api/classes/Ember.Application.html

```
Todos.Todo = DS.Model.extend({
  title: DS.attr('string'),
  isCompleted: DS.attr('boolean')
});
```

This code creates a new class `Todo` and places it within your application's namespace. Each todo will have two attributes: `title` and `isCompleted`.

You may place this file anywhere you like (even just putting all code into the same file), but this guide will assume you have created a file and named it as indicated.

Finally, update your `index.html` to include a reference to this new file:

```
<!-- ... additional lines truncated for brevity ... -->
  <script src="js/models/todo.js"></script>
</body>
<!-- ... additional lines truncated for brevity ... -->
```

Reload your web browser to ensure that all files have been referenced correctly and no errors occur.

Live Preview

Ember.js • TodoMVC[27]

Additional Resources

- Changes in this step in `diff` format[28]
- Models Guide[29]

Using Fixtures

Now we'll add fixture data. Fixtures are a way to put sample data into an application before connecting the application to long-term persistence.

First, update `js/application.js` to indicate that your application's `ApplicationAdapter` is an extension of the `DS.FixtureAdapter`. Adapters are responsible for communicating with a source of data for your application. Typically this will be a web service API, but in this case we are using an adapter designed to load fixture data:

[27] http://jsbin.com/AJoyOGo

[28] https://github.com/emberjs/quickstart-code-sample/commit/a1ccdb43df29d316a7729321764c00b8d850fcd1

[29] http://emberjs.com/guides/models

```
window.Todos = Ember.Application.create();

Todos.ApplicationAdapter = DS.FixtureAdapter.extend();
```

Next, update the file at `js/models/todo.js` to include the following fixture data:

```
// ... additional lines truncated for brevity ...
Todos.Todo.FIXTURES = [
 {
   id: 1,
   title: 'Learn Ember.js',
   isCompleted: true
 },
 {
   id: 2,
   title: '...',
   isCompleted: false
 },
 {
   id: 3,
   title: 'Profit!',
   isCompleted: false
 }
];
```

Reload your web browser to ensure that all files have been referenced correctly and no errors occur.

Live Preview

Ember.js • TodoMVC[30]

Additional Resources

- Changes in this step in `diff` format[31]

Displaying Model Data

Next we'll update our application to display dynamic todos, replacing our hard coded section in the `todos` template.

Inside the file `js/router.js` implement a `TodosRoute` class with a `model` function that returns all the existing todos:

[30] http://jsbin.com/Ovuw

[31] https://github.com/emberjs/quickstart-code-sample/commit/a586fc9de92cad626ea816e9bb29445525678098

```
// ... additional lines truncated for brevity ...
Todos.TodosRoute = Ember.Route.extend({
  model: function() {
    return this.store.find('todo');
  }
});
```

Because we hadn't implemented this class before, Ember.js provided a `Route` for us with the default behavior of rendering a matching template named `todos` using its naming conventions for object creation[32].

Now that we need custom behavior (returning a specific set of models), we implement the class and add the desired behavior.

Update `index.html` to replace the static `<li>` elements with a Handlebars `{{each}}` helper and a dynamic `{{title}}` for each item.

```
{{! ... additional lines truncated for brevity ... }}
<ul id="todo-list">
  {{#each}}
    <li>
      <input type="checkbox" class="toggle">
      <label>{{title}}</label><button class="destroy"></button>
    </li>
  {{/each}}
</ul>
{{! ... additional lines truncated for brevity ... }}
```

The template loops over the content of its controller. This controller is an instance of `ArrayController` that Ember.js has provided for us as the container for our models. Because we don't need custom behavior for this object yet, we can use the default object provided by the framework.

Reload your web browser to ensure that all files have been referenced correctly and no errors occur.

Live Preview

Ember.js • TodoMVC[33]

[32] http://emberjs.com/guides/concepts/naming-conventions/

[33] http://jsbin.com/EJISAne

Additional Resources

- Changes in this step in `diff` format[34]
- Templates Guide[35]
- Controllers Guide[36]
- Naming Conventions Guide[37]

Displaying A Model's Complete State

TodoMVC strikes through completed todos by applying a CSS class `completed` to the `<li>` element. Update `index.html` to apply a CSS class to this element when a todo's `isCompleted` property is true:

```
{{! ... additional lines truncated for brevity ... }}
<li {{bind-attr class="isCompleted:completed"}}>
  <input type="checkbox" class="toggle">
  <label>{{title}}</label><button class="destroy"></button>
</li>
{{! ... additional lines truncated for brevity ... }}
```

This code will apply the CSS class `completed` when the todo's `isCompleted` property is `true` and remove it when the property becomes `false`.

The first fixture todo in our application has an `isCompleted` property of `true`. Reload the application to see the first todo is now decorated with a strike-through to visually indicate it has been completed.

Live Preview

Ember.js • TodoMVC[38]

Additional Resources

- Changes in this step in `diff` format[39]
- bind-attr API documentation[40]
- bind and bind-attr article by Peter Wagenet[41]

[34] https://github.com/emberjs/quickstart-code-sample/commit/87bd57700110d9dd0b351c4d4855edf90baac3a8

[35] http://emberjs.com/guides/templates/handlebars-basics

[36] http://emberjs.com/guides/controllers

[37] http://emberjs.com/guides/concepts/naming-conventions

[38] http://jsbin.com/oKuwomo

[39] https://github.com/emberjs/quickstart-code-sample/commit/b15e5deffc41cf5ba4161808c7f46a283dc2277f

[40] http://emberjs.com/api/classes/Ember.Handlebars.helpers.html#method_bind-attr

[41] http://www.emberist.com/2012/04/06/bind-and-bindattr.html

Creating A New Model Instance

Next we'll update our static HTML `<input>` to an Ember view that can expose more complex behaviors. Update `index.html` to replace the new todo `<input>` with an `{{input}}` helper:

```
{{! ... additional lines truncated for brevity ... }}
<h1>todos</h1>
{{input type="text" id="new-todo" placeholder="What needs to be done?"
              value=newTitle action="createTodo"}}
{{! ... additional lines truncated for brevity ... }}
```

This will render an `<input>` element at this location with the same `id` and `placeholder` attributes applied. It will also connect the `newTitle` property of this template's controller to the `value` attribute of the `<input>`. When one changes, the other will automatically update to remain synchronized.

Additionally, we connect user interaction (pressing the `<enter>` key) to a method `createTodo` on this template's controller.

Because we have not needed a custom controller behavior until this point, Ember.js provided a default controller object for this template. To handle our new behavior, we can implement the controller class Ember.js expects to find according to its naming conventions[42] and add our custom behavior. This new controller class will automatically be associated with this template for us.

Add a `js/controllers/todos_controller.js` file. You may place this file anywhere you like (even just putting all code into the same file), but this guide will assume you have created the file and named it as indicated.

Inside `js/controllers/todos_controller.js` implement the controller Ember.js expects to find according to its naming conventions[43]:

```
Todos.TodosController = Ember.ArrayController.extend({
  actions: {
    createTodo: function() {
      // Get the todo title set by the "New Todo" text field
      var title = this.get('newTitle');
      if (!title) { return false; }
      if (!title.trim()) { return; }

      // Create the new Todo model
      var todo = this.store.createRecord('todo', {
        title: title,
        isCompleted: false
```

[42] http://emberjs.com/guides/concepts/naming-conventions

[43] http://emberjs.com/guides/concepts/naming-conventions

```
      });

      // Clear the "New Todo" text field
      this.set('newTitle', '');

      // Save the new model
      todo.save();
    }
  }
});
```

This controller will now respond to user action by using its `newTitle` property as the title of a new todo whose `isCompleted` property is false. Then it will clear its `newTitle` property which will synchronize to the template and reset the textfield. Finally, it persists any unsaved changes on the todo.

In `index.html` include `js/controllers/todos_controller.js` as a dependency:

```
<!--- ... additional lines truncated for brevity ... -->
   <script src="js/models/todo.js"></script>
   <script src="js/controllers/todos_controller.js"></script>
 </body>
 <!--- ... additional lines truncated for brevity ... -->
```

Reload your web browser to ensure that all files have been referenced correctly and no errors occur. You should now be able to add additional todos by entering a title in the `<input>` and hitting the `<enter>` key.

Live Preview

Ember.js • TodoMVC[44]

Additional Resources

- Changes in this step in `diff` format[45]
- Ember.TextField API documention[46]
- Ember Controller Guide[47]
- Naming Conventions Guide[48]

[44] href=”http://jsbin.com/ImukUZO

[45] https://github.com/emberjs/quickstart-code-sample/commit/60feb5f369c8eecd9df3f561fbd01595353ce803

[46] http://emberjs.com/api/classes/Ember.TextField.html

[47] http://emberjs.com/guides/controllers

[48] http://emberjs.com/guides/concepts/naming-conventions

Marking a Model as Complete or Incomplete

In this step we'll update our application to allow a user to mark a todo as complete or incomplete and persist the updated information.

In `index.html` update your template to wrap each todo in its own controller by adding an `itemController` argument to the `{{each}}` Handlebars helper. Then convert our static `<input type="checkbox">` into a `{{input}}` helper:

```
{{! ... additional lines truncated for brevity ... }}
{{#each itemController="todo"}}
  <li {{bind-attr class="isCompleted:completed"}}>
    {{input type="checkbox" checked=isCompleted class="toggle"}}
    <label>{{title}}</label><button class="destroy"></button>
  </li>
{{/each}}
{{! ... additional lines truncated for brevity ... }}
```

When this `{{input}}` is rendered it will ask for the current value of the controller's `isCompleted` property. When a user clicks this input, it will set the value of the controller's `isCompleted` property to either `true` or `false` depending on the new checked value of the input.

Implement the controller for each todo by matching the name used as the `itemController` value to a class in your application `Todos.TodoController`. Create a new file at `js/controllers/todo_controller.js` for this code. You may place this file anywhere you like (even just putting all code into the same file), but this guide will assume you have created the file and named it as indicated.

Inside `js/controllers/todo_controller.js` add code for `Todos.TodoController` and its `isCompleted` property:

```
Todos.TodoController = Ember.ObjectController.extend({
  isCompleted: function(key, value){
    var model = this.get('model');

    if (value === undefined) {
      // property being used as a getter
      return model.get('isCompleted');
    } else {
      // property being used as a setter
      model.set('isCompleted', value);
      model.save();
      return value;
    }
  }.property('model.isCompleted')
});
```

When called from the template to display the current `isCompleted` state of the todo, this property will proxy that question[49] to its underlying `model`. When called with a value because a user has toggled the checkbox in the template, this property will set the `isCompleted` property of its `model` to the passed value (`true` or `false`), persist the model update, and return the passed value so the checkbox will display correctly.

The `isCompleted` function is marked a computed property[50] whose value is dependent on the value of `model.isCompleted`.

In `index.html` include `js/controllers/todo_controller.js` as a dependency:

```
<!--- ... additional lines truncated for brevity ... -->
   <script src="js/models/todo.js"></script>
   <script src="js/controllers/todos_controller.js"></script>
   <script src="js/controllers/todo_controller.js"></script>
 </body>
 <!--- ... additional lines truncated for brevity ... -->
```

Reload your web browser to ensure that all files have been referenced correctly and no errors occur. You should now be able to change the `isCompleted` property of a todo.

Live Preview

Ember.js • TodoMVC[51]

Additional Resources

- Changes in this step in `diff` format[52]
- Ember.Checkbox API documentation[53]
- Ember Controller Guide[54]
- Computed Properties Guide[55]
- Naming Conventions Guide[56]

[49] http://emberjs.com/api/classes/Ember.ObjectController.html
[50] http://emberjs.com/guides/object-model/computed-properties/
[51] http://jsbin.com/UDoPajA
[52] https://github.com/emberjs/quickstart-code-sample/commit/8d469c04c237f39a58903a3856409a2592cc18a9
[53] /api/classes/Ember.Checkbox.html
[54] http://emberjs.com/guides/controllers
[55] http://emberjs.com/guides/object-model/computed-properties/
[56] http://emberjs.com/guides/concepts/naming-conventions

Displaying the Number of Incomplete Todos

Next we'll update our template's hard-coded count of completed todos to reflect the actual number of completed todos. Update `index.html` to use two properties:

```
{{! ... additional lines truncated for brevity ... }}
<span id="todo-count">
  <strong>{{remaining}}</strong> {{inflection}} left
</span>
{{! ... additional lines truncated for brevity ... }}
```

Implement these properties as part of this template's controller, the `Todos.TodosController`:

```
// ... additional lines truncated for brevity ...
actions: {
  // ... additional lines truncated for brevity ...
},

remaining: function() {
  return this.filterBy('isCompleted', false).get('length');
}.property('@each.isCompleted'),

inflection: function() {
  var remaining = this.get('remaining');
  return remaining === 1 ? 'item' : 'items';
}.property('remaining')
// ... additional lines truncated for brevity ...
```

The `remaining` property will return the number of todos whose `isCompleted` property is false. If the `isCompleted` value of any todo changes, this property will be recomputed. If the value has changed, the section of the template displaying the count will be automatically updated to reflect the new value.

The `inflection` property will return either a plural or singular version of the word "item" depending on how many todos are currently in the list. The section of the template displaying the count will be automatically updated to reflect the new value.

Reload your web browser to ensure that no errors occur. You should now see an accurate number for remaining todos.

Live Preview

Ember.js • TodoMVC[57]

[57] http://jsbin.com/onOCIrA

Additional Resources

- Changes in this step in `diff` format[58]
- Computed Properties Guide[59]

Toggling Between Showing and Editing States

TodoMVC allows users to double click each todo to display a text `<input>` element where the todo's title can be updated. Additionally the `<li>` element for each todo obtains the CSS class `editing` for style and positioning.

We'll update the application to allow users to toggle into this editing state for a todo. In `index.html` update the contents of the `{{each}}` Handlebars helper to:

```
 {{! ... additional lines truncated for brevity ... }}
{{#each itemController="todo"}}
  <li {{bind-attr class="isCompleted:completed isEditing:editing"}}>
    {{#if isEditing}}
      <input class="edit">
    {{else}}
      {{input type="checkbox" checked=isCompleted class="toggle"}}
      <label {{action "editTodo" on="doubleClick"}}>{{title}}</label><button cla\
ss="destroy"></button>
    {{/if}}
  </li>
{{/each}}
 {{! ... additional lines truncated for brevity ... }}
```

The above code applies three new behaviors to our application: it applies the CSS class `editing` when the controller's `isEditing` property is true and removes it when the `isEditing` property is false. We add a new `{{action}}` helper to the `<label>` so double-clicks will call `editTodo` on this todo's controller. Finally, we wrap our todo in a Handlebars `{{if}}` helper so a text `<input>` will display when we are editing and the todos title will display when we are not editing.

Inside `js/controllers/todo_controller.js` we'll implement the matching logic for this template behavior:

[58] https://github.com/emberjs/quickstart-code-sample/commit/b418407ed9666714c82d894d6b70f785674f7a45

[59] http://emberjs.com/guides/object-model/computed-properties/

```
Todos.TodoController = Ember.ObjectController.extend({
  actions: {
    editTodo: function() {
      this.set('isEditing', true);
    }
  },

  isEditing: false,

// ... additional lines truncated for brevity ...
```

Above we defined an initial `isEditing` value of `false` for controllers of this type and said that when the `editTodo` action is called it should set the `isEditing` property of this controller to `true`. This will automatically trigger the sections of template that use `isEditing` to update their rendered content.

Reload your web browser to ensure that no errors occur. You can now double-click a todo to edit it.

Live Preview

Ember.js • TodoMVC[60]

Additional Resources

- Changes in this step in `diff` format[61]
- Handlebars Conditionals Guide[62]
- bind-attr API documentation[63]
- action API documentation[64]
- bind and bindAttr article by Peter Wagenet[65]

Accepting Edits

In the previous step we updated TodoMVC to allow a user to toggle the display of a text `<input>` for editing a todo's title. Next, we'll add the behavior that immediately focuses the `<input>` when it appears, accepts user input and, when the user presses the `<enter>` key or moves focus away from the editing `<input>` element, persists these changes, then redisplays the todo with its newly updated text.

[60] http://jsbin.com/usiXemu
[61] https://github.com/emberjs/quickstart-code-sample/commit/616bc4f22900bbaa2bf9bdb8de53ba41209d8cc0
[62] http://emberjs.com/guides/templates/conditionals
[63] http://emberjs.com/api/classes/Ember.Handlebars.helpers.html#method_bind-attr
[64] http://emberjs.com/api/classes/Ember.Handlebars.helpers.html#method_action
[65] http://www.emberist.com/2012/04/06/bind-and-bindattr.html

To accomplish this, we'll create a new custom component and register it with Handlebars to make it available to our templates.

Create a new file `js/views/edit_todo_view.js`. You may place this file anywhere you like (even just putting all code into the same file), but this guide will assume you have created the file and named it as indicated.

In `js/views/edit_todo_view.js` create an extension of `Ember.TextField` and register it as a helper[66]:

```
Todos.EditTodoView = Ember.TextField.extend({
  didInsertElement: function() {
    this.$().focus();
  }
});

Ember.Handlebars.helper('edit-todo', Todos.EditTodoView);
```

In `index.html` require this new file:

```
<!--- ... additional lines truncated for brevity ... -->
  <script src="js/controllers/todo_controller.js"></script>
  <script src="js/views/edit_todo_view.js"></script>
</body>
<!--- ... additional lines truncated for brevity ... -->
```

In `index.html` replace the static `<input>` element with our custom `{{edit-todo}}` component, connecting the `value` property, and actions:

```
{{! ... additional lines truncated for brevity ... }}
{{#if isEditing}}
  {{edit-todo class="edit" value=title focus-out="acceptChanges"
                           insert-newline="acceptChanges"}}
{{else}}
{{! ... additional lines truncated for brevity ... }}
```

Pressing the `<enter>` key will trigger the `acceptChanges` event on the instance of `TodoController`. Moving focus away from the `<input>` will trigger the `focus-out` event, calling a method `acceptChanges` on this view's instance of `TodoController`.

[66] http://emberjs.com/api/classes/Ember.Handlebars.html#method_helper

Additionally, we connect the `value` property of this `<input>` to the `title` property of this instance of `TodoController`. We will not implement a `title` property on the controller so it will retain the default behavior of proxying all requests[67] to its `model`.

A CSS class `edit` is applied for styling.

In `js/controllers/todo_controller.js`, add the method `acceptChanges` that we called from `EditTodoView`:

```
// ... additional lines truncated for brevity ...
actions: {
  editTodo: function() {
    this.set('isEditing', true);
  },
  acceptChanges: function() {
    this.set('isEditing', false);

    if (Ember.isEmpty(this.get('model.title'))) {
      this.send('removeTodo');
    } else {
      this.get('model').save();
    }
  },
  removeTodo: function () {
    var todo = this.get('model');
    todo.deleteRecord();
    todo.save();
  }
},
// ... additional lines truncated for brevity ...
```

This method will set the controller's `isEditing` property to false and commit all changes made to the todo.

Live Preview

Ember.js • TodoMVC[68]

Additional Resources

- Changes in this step in `diff` format[69]

[67] http://emberjs.com/guides/controllers/#toc_representing-models

[68] http://jsbin.com/USOlAna

[69] https://github.com/emberjs/quickstart-code-sample/commit/a7e2f40da4d75342358acdfcbda7a05ccc90f348

- Controller Guide[70]
- Ember.TextField API documentation[71]

Deleting a Model

TodoMVC displays a button for removing todos next to each todo when its `<li>` is hovered. Clicking this button will remove the todo and update the display of remaining incomplete todos and remaining completed todos appropriately.

In `index.html` update the static `<button>` element to include an `{{action}}` Handlebars helper:

```
{{! ... additional lines truncated for brevity ... }}
<button {{action "removeTodo"}} class="destroy"></button>
{{! ... additional lines truncated for brevity ... }}
```

This will call the `removeTodo` action defined in the previous chapter and will delete the todo locally and then persist this data change.

Because the todo is no longer part of the collection of all todos, its `<li>` element in the page will be automatically removed for us. If the deleted todo was incomplete, the count of remaining todos will be decreased by one and the display of this number will be automatically re-rendered. If the new count results in an inflection change between "todo" and "todos" this area of the page will be automatically re-rendered.

Reload your web browser to ensure that there are no errors and the behaviors described above occurs.

<aside> Note: The current action may be invoked twice (via `acceptChanges`) leading to an exception. For details on how to handle this situation, please see the latest version of the TodoMVC source[72]. </aside>

Live Preview

Ember.js • TodoMVC[73]

Additional Resources

- Changes in this step in `diff` format[74]
- action API documentation[75]

[70] http://emberjs.com/guides/controllers

[71] http://emberjs.com/api/classes/Ember.TextField.html

[72] https://github.com/tastejs/todomvc/blob/gh-pages/architecture-examples/emberjs/js/controllers/todo_controller.js

[73] http://jsbin.com/eREkanA

[74] https://github.com/emberjs/quickstart-code-sample/commit/14e1f129f76bae8f8ea6a73de1e24d810678a8fe

[75] http://emberjs.com/api/classes/Ember.Handlebars.helpers.html#method_action

Adding Child Routes

Next we will split our single template into a set of nested templates so we can transition between different lists of todos in reaction to user interaction.

In `index.html` move the entire `<ul>` of todos into a new template named `todos/index` by adding a new Handlebars template `<script>` tag inside the `<body>` of the document:

```
<!--- ... additional lines truncated for brevity ... -->
<body>
<script type="text/x-handlebars" data-template-name="todos/index">
  <ul id="todo-list">
    {{#each itemController="todo"}}
      <li {{bind-attr class="isCompleted:completed isEditing:editing"}}>
        {{#if isEditing}}
          {{edit-todo class="edit" value=title focus-out="acceptChanges" insert-\
newline="acceptChanges"}}
        {{else}}
          {{input type="checkbox" checked=isCompleted class="toggle"}}
          <label {{action "editTodo" on="doubleClick"}}>{{title}}</label><button\
 {{action "removeTodo"}} class="destroy"></button>
        {{/if}}
      </li>
    {{/each}}
  </ul>
</script>
<!--- ... additional lines truncated for brevity ... -->
```

Still within `index.html` place a Handlebars `{{outlet}}` helper where the `<ul>` was previously:

```
{{! ... additional lines truncated for brevity ... }}
<section id="main">
  {{outlet}}

  <input type="checkbox" id="toggle-all">
</section>
{{! ... additional lines truncated for brevity ... }}
```

The `{{outlet}}` Handlebars helper designates an area of a template that will dynamically update as we transition between routes. Our first new child route will fill this area with the list of all todos in the application.

In `js/router.js` update the router to change the `todos` mapping, with an additional empty function parameter so it can accept child routes, and add this first `index` route:

```
Todos.Router.map(function () {
  this.resource('todos', { path: '/' }, function () {
    // additional child routes will go here later
  });
});

// ... additional lines truncated for brevity ...

Todos.TodosIndexRoute = Ember.Route.extend({
  model: function() {
    return this.modelFor('todos');
  }
});
```

When the application loads at the url `'/'` Ember.js will enter the `todos` route and render the `todos` template as before. It will also transition into the `todos.index` route and fill the `{{outlet}}` in the `todos` template with the `todos/index` template. The model data for this template is the result of the `model` method of `TodosIndexRoute`, which indicates that the model for this route is the same model as for the `TodosRoute`.

This mapping is described in more detail in the Naming Conventions Guide[76].

Live Preview

Ember.js • TodoMVC[77]

Additional Resources

- Changes in this step in `diff` format[78]
- Ember Router Guide[79]
- Ember Controller Guide[80]
- outlet API documentation[81]

Transitioning to Show Only Incomplete Todos

Next we'll update the application so a user can navigate to a url where only todos that are not complete are displayed.

[76] http://emberjs.com/guides/concepts/naming-conventions
[77] http://jsbin.com/oweNovo
[78] https://github.com/emberjs/quickstart-code-sample/commit/3bab8f1519ffc1ca2d5a12d1de35e4c764c91f05
[79] http://emberjs.com/guides/routing
[80] http://emberjs.com/guides/controllers
[81] http://emberjs.com/api/classes/Ember.Handlebars.helpers.html#method_outlet

In `index.html` convert the `<a>` tag for 'Active' todos into a Handlebars `{{link-to}}` helper and remove the active class from the `<a>` tag for 'All':

```
{{! ... additional lines truncated for brevity ... }}
<li>
  <a href="all">All</a>
</li>
<li>
  {{#link-to "todos.active" activeClass="selected"}}Active{{/link-to}}
</li>
<li>
  <a href="completed">Completed</a>
</li>
{{! ... additional lines truncated for brevity ... }}
```

In `js/router.js` update the router to recognize this new path and implement a matching route:

```
Todos.Router.map(function() {
  this.resource('todos', { path: '/' }, function() {
    // additional child routes
    this.route('active');
  });
});

// ... additional lines truncated for brevity ...
Todos.TodosActiveRoute = Ember.Route.extend({
  model: function(){
    return this.store.filter('todo', function(todo) {
      return !todo.get('isCompleted');
    });
  },
  renderTemplate: function(controller) {
    this.render('todos/index', {controller: controller});
  }
});
```

The model data for this route is the collection of todos whose `isCompleted` property is `false`. When a todo's `isCompleted` property changes this collection will automatically update to add or remove the todo appropriately.

Normally transitioning into a new route changes the template rendered into the parent `{{outlet}}`, but in this case we'd like to reuse the existing `todos/index` template. We can accomplish this by implementing the `renderTemplate` method and calling `render` ourselves with the specific template and controller options.

Reload your web browser to ensure that there are no errors and the behavior described above occurs.

Live Preview

[Ember.js • TodoMVC[(http://jsbin.com/arITiZu)

Additional Resources

- Changes in this step in `diff` format[82]
- link-to API documentation[83]
- Route#renderTemplate API documentation[84]
- Route#render API documentation[85]
- Ember Router Guide[86]

Transitioning to Show Only Complete Todos

Next we'll update the application so a user can navigate to a url where only todos that have already been completed are displayed.

In `index.html` convert the `<a>` tag for 'Completed' todos into a Handlebars `{{link-to}}` helper:

```
{{! ... additional lines truncated for brevity ... }}
<li>
  <a href="all">All</a>
</li>
<li>
  {{#link-to "todos.active" activeClass="selected"}}Active{{/link-to}}
</li>
<li>
  {{#link-to "todos.completed" activeClass="selected"}}Completed{{/link-to}}
</li>
{{! ... additional lines truncated for brevity ... }}
```

In `js/router.js` update the router to recognize this new path and implement a matching route:

[82] https://github.com/emberjs/quickstart-code-sample/commit/2a1d35293a52e40d0125f552a1a8b2c01f759313

[83] http://emberjs.com/api/classes/Ember.Handlebars.helpers.html#method_link-to

[84] http://emberjs.com/api/classes/Ember.Route.html#method_renderTemplate

[85] http://emberjs.com/api/classes/Ember.Route.html#method_render

[86] http://emberjs.com/guides/routing

```
Todos.Router.map(function() {
  this.resource('todos', { path: '/' }, function() {
    // additional child routes
    this.route('active');
    this.route('completed');
  });
});

// ... additional lines truncated for brevity ...

Todos.TodosCompletedRoute = Ember.Route.extend({
  model: function() {
    return this.store.filter('todo', function(todo) {
      return todo.get('isCompleted');
    });
  },
  renderTemplate: function(controller) {
    this.render('todos/index', {controller: controller});
  }
});
```

The model data for this route is the collection of todos whose `isCompleted` property is `true`. Just like we recently saw with the similar function for the active todos, changes to a todo's `isCompleted` property will automatically cause this collection to refresh, updating the UI accordingly.

`TodosCompletedRoute` has a similar purpose to the active todos - to reuse the existing `todos/index` template, rather than having to create a new template.

Reload your web browser to ensure that there are no errors and the behavior described above occurs.

Live Preview

Ember.js • TodoMVC[87]

Additional Resources

- Changes in this step in `diff` format[88]
- link-to API documentation[89]
- Route#renderTemplate API documentation[90]

[87] http://jsbin.com/OzUvuPu

[88] https://github.com/emberjs/quickstart-code-sample/commit/bba939a11197552e3a927bcb3a3adb9430e4f331

[89] http://emberjs.com/api/classes/Ember.Handlebars.helpers.html#method_link-to

[90] http://emberjs.com/api/classes/Ember.Route.html#method_renderTemplate

- Route#render API documentation[91]
- Ember Router Guide[92]

Transitioning back to Show All Todos

Next we can update the application to allow navigating back to the list of all todos.

In `index.html` convert the `<a>` tag for 'All' todos into a Handlebars `{{link-to}}` helper:

```
{{! ... additional lines truncated for brevity ... }}
<li>
  {{#link-to "todos.index" activeClass="selected"}}All{{/link-to}}
</li>
<li>
  {{#link-to "todos.active" activeClass="selected"}}Active{{/link-to}}
</li>
<li>
  {{#link-to "todos.completed" activeClass="selected"}}Completed{{/link-to}}
</li>
{{! ... additional lines truncated for brevity ... }}
```

Reload your web browser to ensure that there are no errors. You should be able to navigate between urls for all, active, and completed todos.

Live Preview

Ember.js • TodoMVC[93]

Additional Resources

- Changes in this step in `diff` format[94]
- link-to API documentation[95]

[91] http://emberjs.com/api/classes/Ember.Route.html#method_render

[92] http://emberjs.com/guides/routing

[93] http://jsbin.com/uYuGA

[94] https://github.com/emberjs/quickstart-code-sample/commit/843ff914873081560e4ba97df0237b8595b6ae51

[95] http://emberjs.com/api/classes/Ember.Handlebars.helpers.html#method_link-to

Displaying a Button to Remove All Completed Todos

TodoMVC allows users to delete all completed todos at once by clicking a button. This button is visible only when there are any completed todos, displays the number of completed todos, and removes all completed todos from the application when clicked.

In this step, we'll implement that behavior. In `index.html` update the static `<button>` for clearing all completed todos:

```
{{! ... additional lines truncated for brevity ... }}
{{#if hasCompleted}}
  <button id="clear-completed" {{action "clearCompleted"}}>
    Clear completed ({{completed}})
  </button>
{{/if}}
{{! ... additional lines truncated for brevity ... }}
```

In `js/controllers/todos_controller.js` implement the matching properties and a method that will clear completed todos and persist these changes when the button is clicked:

```
// ... additional lines truncated for brevity ...
actions: {
  clearCompleted: function() {
    var completed = this.filterBy('isCompleted', true);
    completed.invoke('deleteRecord');
    completed.invoke('save');
  },
  // ... additional lines truncated for brevity ...
},
hasCompleted: function() {
  return this.get('completed') > 0;
}.property('completed'),

completed: function() {
  return this.filterBy('isCompleted', true).get('length');
}.property('@each.isCompleted'),
// ... additional lines truncated for brevity ...
```

The `completed` and `clearCompleted` methods both invoke the `filterBy` method, which is part of the ArrayController[96] API and returns an instance of EmberArray[97] which contains only the items

[96] http://emberjs.com/api/classes/Ember.ArrayController.html#method_filterProperty

[97] http://emberjs.com/api/classes/Ember.Array.html

for which the callback returns true. The `clearCompleted` method also invokes the `invoke` method which is part of the EmberArray[98] API. `invoke` will execute a method on each object in the Array if the method exists on that object.

Reload your web browser to ensure that there are no errors and the behavior described above occurs.

Live Preview

Ember.js • TodoMVC[99]

Additional Resources

- Changes in this step in `diff` format[100]
- Handlebars Conditionals Guide[101]
- Enumerables Guide[102]

Indicating When All Todos Are Complete

Next we'll update our template to indicate when all todos have been completed. In `index.html` replace the static checkbox `<input>` with an `{{input}}`:

```
{{! ... additional lines truncated for brevity ... }}
<section id="main">
  {{outlet}}
  {{input type="checkbox" id="toggle-all" checked=allAreDone}}
</section>
{{! ... additional lines truncated for brevity ... }}
```

This checkbox will be checked when the controller property `allAreDone` is `true` and unchecked when the property `allAreDone` is `false`.

In `js/controllers/todos_controller.js` implement the matching `allAreDone` property:

[98] http://emberjs.com/api/classes/Ember.Array.html#method_invoke
[99] http://jsbin.com/ULovoJI
[100] https://github.com/emberjs/quickstart-code-sample/commit/1da450a8d693f083873a086d0d21e031ee3c129e
[101] http://emberjs.com/guides/templates/conditionals
[102] http://emberjs.com/guides/enumerables

```
// ... additional lines truncated for brevity ...
allAreDone: function(key, value) {
  return !!this.get('length') && this.isEvery('isCompleted');
}.property('@each.isCompleted')
// ... additional lines truncated for brevity ...
```

This property will be `true` if the controller has any todos and every todo's `isCompleted` property is true. If the `isCompleted` property of any todo changes, this property will be recomputed. If the return value has changed, sections of the template that need to update will be automatically updated for us.

Reload your web browser to ensure that there are no errors and the behavior described above occurs.

Live Preview

Ember.js • TodoMVC[103]

Additional Resources

- Changes in this step in `diff` format[104]
- Ember.Checkbox API documentation[105]

Toggling All Todos Between Complete and Incomplete

TodoMVC allows users to toggle all existing todos into either a complete or incomplete state. It uses the same checkbox that becomes checked when all todos are completed and unchecked when one or more todos remain incomplete.

To implement this behavior update the `allAreDone` property in `js/controllers/todos_controller.js` to handle both getting and setting behavior:

[103] http://jsbin.com/IcItARE

[104] https://github.com/emberjs/quickstart-code-sample/commit/9bf8a430bc4afb06f31be55f63f1d9806e6ab01c

[105] http://emberjs.com/api/classes/Ember.Checkbox.html

```
// ... additional lines truncated for brevity ...
allAreDone: function(key, value) {
  if (value === undefined) {
    return !!this.get('length') && this.isEvery('isCompleted', true);
  } else {
    this.setEach('isCompleted', value);
    this.invoke('save');
    return value;
  }
}.property('@each.isCompleted')
// ... additional lines truncated for brevity ...
```

If no `value` argument is passed this property is being used to populate the current value of the checkbox. If a `value` is passed it indicates the checkbox was used by a user and we should set the `isCompleted` property of each todo to this new value.

The count of remaining todos and completed todos used elsewhere in the template automatically re-render for us if necessary.

Reload your web browser to ensure that there are no errors and the behavior described above occurs.

Live Preview

Ember.js • TodoMVC[106]

Additional Resources

- Changes in this step in `diff` format[107]
- Ember.Checkbox API documentation[108]
- Computed Properties Guide[109]

Replacing the Fixture Adapter with Another Adapter

Finally we'll replace our fixture data with real persistence so todos will remain between application loads by replacing the fixture adapter with a `localstorage`-aware adapter instead.

Change `js/application.js` to:

[106] http://jsbin.com/AViZATE
[107] https://github.com/emberjs/quickstart-code-sample/commit/47b289bb9f669edaa39abd971f5e884142988663
[108] http://emberjs.com/api/classes/Ember.Checkbox.html
[109] http://emberjs.com/guides/object-model/computed-properties/

```
window.Todos = Ember.Application.create();

Todos.ApplicationAdapter = DS.LSAdapter.extend({
  namespace: 'todos-emberjs'
});
```

The local storage adapter, written by Ryan Florence, can be downloaded from its source[110]. Add it to your project as `js/libs/localstorage_adapter.js`. You may place this file anywhere you like (even just putting all code into the same file), but this guide will assume you have created the file and named it as indicated.

In `index.html` include `js/libs/localstorage_adapter.js` as a dependency:

```
<!--- ... additional lines truncated for brevity ... -->
<script src="js/libs/ember-data.js"></script>
<script src="js/libs/localstorage_adapter.js"></script>
<script src="js/application.js"></script>
 <!--- ... additional lines truncated for brevity ... -->
```

Reload your application. Todos you manage will now persist after the application has been closed.

Live Preview

Ember.js • TodoMVC[111]

Additional Resources

- Changes in this step in `diff` format[112]
- LocalStorage Adapter on GitHub[113]

[110] https://raw.github.com/rpflorence/ember-localstorage-adapter/master/localstorage_adapter.js

[111] http://jsbin.com/aZIXaYo

[112] https://github.com/emberjs/quickstart-code-sample/commit/81801d87da42d0c83685ff946c46de68589ce38f

[113] https://github.com/rpflorence/ember-localstorage-adapter

Getting Ember

Ember Builds

The Ember Release Management Team maintains a variety of ways to get Ember and Ember Data builds.

Channels

The latest Release[114], Beta[115], and Canary[116] builds of Ember and Ember data can be found here[117]. For each channel a development, minified, and production version is available. For more on the different channels read the Post 1.0 Release Cycle[118] blog post.

Tagged Releases

Past release and beta builds of Ember and Ember Data are available at Tagged Releases[119]. These builds can be useful to track down regressions in your application, but it is recommended to use the latest stable release in production.

Bower

Bower is a package manager for the web. Bower makes it easy to manage dependencies in your application including Ember and Ember Data. To learn more about Bower visit http://bower.io/[120].

Adding Ember to your application with Bower is easy; simply run `bower install ember --save`. For Ember Data, run `bower install ember-data --save`. You can also add `ember` or `ember-data` to your `bower.json` file as follows.

[114] http://emberjs.com/builds#/release

[115] http://emberjs.com/builds#/beta

[116] http://emberjs.com/builds#/canary

[117] http://emberjs.com/builds

[118] http://emberjs.com/blog/2013/09/06/new-ember-release-process.html

[119] http://emberjs.com/builds#/tagged

[120] http://bower.io/

```
{
        "name": "your-app",
        "dependencies": {
                "ember": "~1.6",
                "ember-data": "~1.0.0-beta.8"
        }
}
```

RubyGems

If your application uses a Ruby based build system, you can use the ember-source[121] and ember-data-source[122] RubyGems to access ember and ember data sources from Ruby.

If your application is built in Rails, the ember-rails[123] RubyGem makes it easy to integrate Ember into your Ruby on Rails application.

[121] http://rubygems.org/gems/ember-source

[122] http://rubygems.org/gems/ember-data-source

[123] http://rubygems.org/gems/ember-rails

Concepts

Core Concept

To get started with Ember.js, there are a few core concepts you should understand.

Ember.js is designed to help developers build ambitiously large web applications that are competitive with native apps. Doing so requires both new tools and a new vocabulary of concepts. We've spent a lot of time borrowing ideas pioneered by native application frameworks like Cocoa and Smalltalk.

However, it's important to remember what makes the web special. Many people think that something is a web application because it uses technologies like HTML, CSS and JavaScript. In reality, these are just implementation details.

Instead, **the web derives its power from the ability to bookmark and share URLs**. URLs are the key feature that give web applications superior shareability and collaboration. Today, most JavaScript frameworks treat the URL as an afterthought, instead of the primary reason for the web's success.

Ember.js, therefore, marries the tools and concepts of native GUI frameworks with support for the feature that makes the web so powerful: the URL.

Concepts

Templates

A **template**, written in the Handlebars templating language, describes the user interface of your application. Each template is backed by a model, and the template automatically updates itself if the model changes.

In addition to plain HTML, templates can contain:

- **Expressions**, like `{{firstName}}`, which take information from the template's model and put it into HTML.
- **Outlets**, which are placeholders for other templates. As users move around your app, different templates can be plugged into the outlet by the router. You can put outlets into your template using the `{{outlet}}` helper.
- **Components**, custom HTML elements that you can use to clean up repetitive templates or create reusable controls.

Router

The **router** translates a URL into a series of nested templates, each backed by a model. As the templates or models being shown to the user change, Ember automatically keeps the URL in the browser's address bar up-to-date.

This means that, at any point, users are able to share the URL of your app. When someone clicks the link, they reliably see the same content as the original user.

Components

A **component** is a custom HTML tag whose behavior you implement using JavaScript and whose appearance you describe using Handlebars templates. They allow you to create reusable controls that can simplify your application's templates.

Models

A **model** is an object that stores *persistent state*. Templates are responsible for displaying the model to the user by turning it into HTML. In many applications, models are loaded via an HTTP JSON API, although Ember is agnostic to the backend that you choose.

Route

A **route** is an object that tells the template which model it should display.

Controllers

A **controller** is an object that stores *application state*. A template can optionally have a controller in addition to a model, and can retrieve properties from both.

These are the core concepts you'll need to understand as you develop your Ember.js app. They are designed to scale up in complexity, so that adding new functionality doesn't force you to go back and refactor major parts of your app.

Now that you understand the roles of these objects, you're equipped to dive deep into Ember.js and learn the details of how each of these individual pieces work.

Naming Conventions

Ember.js uses naming conventions to wire up your objects without a lot of boilerplate. You will want to use these conventional names for your routes, controllers and templates.

You can usually guess the names, but this guide outlines, in one place, all of the naming conventions. In the following examples 'App' is a name that we chose to namespace or represent our Ember application when it was created, but you can choose any name you want for your application. We will show you later how to create an Ember application, but for now we will focus on conventions.

The Application

When your application boots, Ember will look for these objects:

- `App.ApplicationRoute`
- `App.ApplicationController`
- the `application` template

Ember.js will render the `application` template as the main template. If `App.ApplicationController` is provided, Ember.js will set an instance of `App.ApplicationController` as the controller for the template. This means that the template will get its properties from the controller.

If your app provides an `App.ApplicationRoute`, Ember.js will invoke the[124] router's[125] hooks[126] first, before rendering the `application` template. Hooks are implemented as methods and provide you access points within an Ember object's lifecycle to intercept and execute code to modify the default behavior at these points to meet your needs. Ember provides several hooks for you to utilize for various purposes (e.g. `model`, `setupController`, etc). In the example below `App.ApplicationRoute`, which is a `Ember.Route` object, implements the `setupController` hook.

Here's a simple example that uses a route, controller, and template:

[124] http://emberjs.com/guides/routing/specifying-a-routes-model

[125] http://emberjs.com/guides/routing/setting-up-a-controller

[126] http://emberjs.com/guides/routing/rendering-a-template

```
App.ApplicationRoute = Ember.Route.extend({
  setupController: function(controller) {
    // `controller` is the instance of ApplicationController
    controller.set('title', "Hello world!");
  }
});

App.ApplicationController = Ember.Controller.extend({
  appName: 'My First Example'
});
```

```
<!-- application template -->
<h1>{{appName}}</h1>

<h2>{{title}}</h2>
```

In Ember.js applications, you will always specify your controllers as **classes**, and the framework is responsible for instantiating them and providing them to your templates.

This makes it super-simple to test your controllers, and ensures that your entire application shares a single instance of each controller.

Simple Routes

Each of your routes will have a controller, and a template with the same name as the route.

Let’s start with a simple router:

```
App.Router.map(function() {
  this.route('favorites');
});
```

If your user navigates to `/favorites`, Ember.js will look for these objects:

- `App.FavoritesRoute`
- `App.FavoritesController`
- the `favorites` template

Ember.js will render the `favorites` template into the `{{outlet}}` in the `application` template. It will set an instance of the `App.FavoritesController` as the controller for the template.

If your app provides an `App.FavoritesRoute`, the framework will invoke it before rendering the template. Yes, this is a bit repetitive.

For a route like `App.FavoritesRoute`, you will probably implement the `model` hook to specify what model your controller will present to the template.

Here's an example:

```
App.FavoritesRoute = Ember.Route.extend({
  model: function() {
    // the model is an Array of all of the posts
    return this.store.find('post');
  }
});
```

In this example, we didn't provide a `FavoritesController`. Because the model is an Array, Ember.js will automatically supply an instance of `Ember.ArrayController`, which will present the backing Array as its model.

You can treat the `ArrayController` as if it was the model itself. This has two major benefits:

- You can replace the controller's model at any time without having to directly notify the view of the change.
- The controller can provide additional computed properties or view-specific state that do not belong in the model layer. This allows a clean separation of concerns between the view, the controller and the model.

The template can iterate over the elements of the controller:

```
<ul>
{{#each post}}
  <li>{{title}}</li>
{{/each}}
</ul>
```

Dynamic Segments

If a route uses a dynamic segment, the route's model will be based on the value of that segment provided by the user.

Consider this router definition:

```
App.Router.map(function() {
  this.resource('post', { path: '/posts/:post_id' });
});
```

In this case, the route's name is `post`, so Ember.js will look for these objects:

- `App.PostRoute`
- `App.PostController`
- the `post` template

Your route handler's `model` hook converts the dynamic `:post_id` parameter into a model. The `serialize` hook converts a model object back into the URL parameters for this route (for example, when generating a link for a model object).

```
App.PostRoute = Ember.Route.extend({
  model: function(params) {
    return this.store.find('post', params.post_id);
  },

  serialize: function(post) {
    return { post_id: post.get('id') };
  }
});
```

Because this pattern is so common, it is the default for route handlers.

- If your dynamic segment ends in `_id`, the default `model` hook will convert the first part into a model class on the application's namespace (`post` becomes `App.Post`). It will then call `find` on that class with the value of the dynamic segment.
- The default `serialize` hook will pull the dynamic segment with the `id` property of the model object.

Route, Controller and Template Defaults

If you don't specify a route handler for the `post` route (`App.PostRoute`), Ember.js will still render the `post` template with the app's instance of `App.PostController`.

If you don't specify the controller (`App.PostController`), Ember will automatically make one for you based on the return value of the route's `model` hook. If the model is an Array, you get an `ArrayController`. Otherwise, you get an `ObjectController`.

If you don't specify a `post` template, Ember.js won't render anything!

Nesting

You can nest routes under a `resource`.

```
App.Router.map(function() {
  this.resource('posts', function() { // the `posts` route
    this.route('favorites');          // the `posts.favorites` route
    this.resource('post');            // the `post` route
  });
});
```

A **resource** is the beginning of a route, controller, or template name. Even though the `post` resource is nested, its route is named `App.PostRoute`, its controller is named `App.PostController` and its template is `post`.

When you nest a **route** inside a resource, the route name is added to the resource name, after a `.`.

Here are the naming conventions for each of the routes defined in this router:

<table> <thead> <tr> <th>Route Name</th> <th>Controller</th> <th>Route</th> <th>Template</th>
</tr> </thead> <tr> <td><code>posts</code></td> <td><code>PostsController</code></td> <td><code>PostsRoute<
<td><code>posts</code></td> </tr> <tr> <td><code>posts.favorites</code></td> <td><code>PostsFavoritesControll
<td><code>PostsFavoritesRoute</code></td> <td><code>posts/favorites</code></td> </tr> <tr> <td><code>post</c
<td><code>PostController</code></td> <td><code>PostRoute</code></td> <td><code>post</code></td>
</tr> </table>

The rule of thumb is to use resources for nouns, and routes for adjectives (`favorites`) or verbs (`edit`). This ensures that nesting does not create ridiculously long names, but avoids collisions with common adjectives and verbs.

The Index Route

At every level of nesting (including the top level), Ember.js automatically provides a route for the `/` path named `index`.

For example, if you write a simple router like this:

```
App.Router.map(function() {
  this.route('favorites');
});
```

It is the equivalent of:

```
App.Router.map(function() {
  this.route('index', { path: '/' });
  this.route('favorites');
});
```

If the user visits `/`, Ember.js will look for these objects:

- `App.IndexRoute`
- `App.IndexController`
- the `index` template

The `index` template will be rendered into the `{{outlet}}` in the `application` template. If the user navigates to `/favorites`, Ember.js will replace the `index` template with the `favorites` template.

A nested router like this:

```
App.Router.map(function() {
  this.resource('posts', function() {
    this.route('favorites');
  });
});
```

Is the equivalent of:

```
App.Router.map(function() {
  this.route('index', { path: '/' });
  this.resource('posts', function() {
    this.route('index', { path: '/' });
    this.route('favorites');
  });
});
```

If the user navigates to `/posts`, the current route will be `posts.index`. Ember.js will look for objects named:

- `App.PostsIndexRoute`
- `App.PostsIndexController`
- The `posts/index` template

First, the `posts` template will be rendered into the `{{outlet}}` in the `application` template. Then, the `posts/index` template will be rendered into the `{{outlet}}` in the `posts` template.

If the user then navigates to `/posts/favorites`, Ember.js will replace the `{{outlet}}` in the `posts` template with the `posts/favorites` template.

The Object Model

Classes and Instances

To define a new Ember *class*, call the `extend()` method on `Ember.Object`:

```
App.Person = Ember.Object.extend({
  say: function(thing) {
    alert(thing);
  }
});
```

This defines a new `App.Person` class with a `say()` method.

You can also create a *subclass* from any existing class by calling its `extend()` method. For example, you might want to create a subclass of Ember's built-in `Ember.View` class:

```
App.PersonView = Ember.View.extend({
  tagName: 'li',
  classNameBindings: ['isAdministrator']
});
```

When defining a subclass, you can override methods but still access the implementation of your parent class by calling the special `_super()` method:

```
App.Person = Ember.Object.extend({
  say: function(thing) {
    var name = this.get('name');
    alert(name + " says: " + thing);
  }
});

App.Soldier = App.Person.extend({
  say: function(thing) {
    this._super(thing + ", sir!");
  }
});

```

```
var yehuda = App.Soldier.create({
  name: "Yehuda Katz"
});

yehuda.say("Yes"); // alerts "Yehuda Katz says: Yes, sir!"
```

Creating Instances

Once you have defined a class, you can create new *instances* of that class by calling its `create()` method. Any methods, properties and computed properties you defined on the class will be available to instances:

```
var person = App.Person.create();
person.say("Hello"); // alerts " says: Hello"
```

When creating an instance, you can initialize the value of its properties by passing an optional hash to the `create()` method:

```
App.Person = Ember.Object.extend({
  helloWorld: function() {
    alert("Hi, my name is " + this.get('name'));
  }
});

var tom = App.Person.create({
  name: "Tom Dale"
});

tom.helloWorld(); // alerts "Hi, my name is Tom Dale"
```

For performance reasons, note that you cannot redefine an instance's computed properties or methods when calling `create()`, nor can you define new ones. You should only set simple properties when calling `create()`. If you need to define or redefine methods or computed properties, create a new subclass and instantiate that.

By convention, properties or variables that hold classes are PascalCased, while instances are not. So, for example, the variable `App.Person` would point to a class, while `person` would point to an instance (usually of the `App.Person` class). You should stick to these naming conventions in your Ember applications.

Initializing Instances

When a new instance is created, its `init` method is invoked automatically. This is the ideal place to do setup required on new instances:

```
App.Person = Ember.Object.extend({
  init: function() {
    var name = this.get('name');
    alert(name + ", reporting for duty!");
  }
});

App.Person.create({
  name: "Stefan Penner"
});

// alerts "Stefan Penner, reporting for duty!"
```

If you are subclassing a framework class, like `Ember.View` or `Ember.ArrayController`, and you override the `init` method, make sure you call `this._super()`! If you don't, the system may not have an opportunity to do important setup work, and you'll see strange behavior in your application.

When accessing the properties of an object, use the `get` and `set` accessor methods:

```
var person = App.Person.create();

var name = person.get('name');
person.set('name', "Tobias Fünke");
```

Make sure to use these accessor methods; otherwise, computed properties won't recalculate, observers won't fire, and templates won't update.

Computed Properties

What are Computed Properties?

In a nutshell, computed properties let you declare functions as properties. You create one by defining a computed property as a function, which Ember will automatically call when you ask for the property. You can then use it the same way you would any normal, static property.

It's super handy for taking one or more normal properties and transforming or manipulating their data to create a new value.

Computed properties in action

We'll start with a simple example:

```
App.Person = Ember.Object.extend({
  // these will be supplied by `create`
  firstName: null,
  lastName: null,

  fullName: function() {
    return this.get('firstName') + ' ' + this.get('lastName');
  }.property('firstName', 'lastName')
});

var ironMan = App.Person.create({
  firstName: "Tony",
  lastName:  "Stark"
});

ironMan.get('fullName'); // "Tony Stark"
```

Notice that the `fullName` function calls `property`. This declares the function to be a computed property, and the arguments tell Ember that it depends on the `firstName` and `lastName` attributes.

Whenever you access the `fullName` property, this function gets called, and it returns the value of the function, which simply calls `firstName` + `lastName`.

Alternate invocation

At this point, you might be wondering how you are able to call the `.property` function on a function. This is possible because Ember extends the `function` prototype. More information about extending native prototypes is available in the disabling prototype extensions guide[127]. If you'd like to replicate the declaration from above without using these extensions you could do so with the following:

```
  fullName: Ember.computed('firstName', 'lastName', function() {
    return this.get('firstName') + ' ' + this.get('lastName');
  })
```

Chaining computed properties

You can use computed properties as values to create new computed properties. Let's add a `description` computed property to the previous example, and use the existing `fullName` property and add in some other properties:

[127] http://emberjs.com/guides/configuring-ember/disabling-prototype-extensions/

```
App.Person = Ember.Object.extend({
  firstName: null,
  lastName: null,
  age: null,
  country: null,

  fullName: function() {
    return this.get('firstName') + ' ' + this.get('lastName');
  }.property('firstName', 'lastName'),

  description: function() {
    return this.get('fullName') + '; Age: ' + this.get('age') + '; Country: ' + \
this.get('country');
  }.property('fullName', 'age', 'country')
});

var captainAmerica = App.Person.create({
  firstName: 'Steve',
  lastName: 'Rogers',
  age: 80,
  country: 'USA'
});

captainAmerica.get('description'); // "Steve Rogers; Age: 80; Country: USA"
```

Dynamic updating

Computed properties, by default, observe any changes made to the properties they depend on and are dynamically updated when they're called. Let's use computed properties to dynamically update.

```
captainAmerica.set('firstName', 'William');

captainAmerica.get('description'); // "William Rogers; Age: 80; Country: USA"
```

So this change to `firstName` was observed by `fullName` computed property, which was itself observed by the `description` property.

Setting any dependent property will propagate changes through any computed properties that depend on them, all the way down the chain of computed properties you've created.

Setting Computed Properties

You can also define what Ember should do when setting a computed property. If you try to set a computed property, it will be invoked with the key (property name), the value you want to set it to, and the previous value.

```
App.Person = Ember.Object.extend({
  firstName: null,
  lastName: null,

  fullName: function(key, value, previousValue) {
    // setter
    if (arguments.length > 1) {
      var nameParts = value.split(/\s+/);
      this.set('firstName', nameParts[0]);
      this.set('lastName',  nameParts[1]);
    }

    // getter
    return this.get('firstName') + ' ' + this.get('lastName');
  }.property('firstName', 'lastName')
});


var captainAmerica = App.Person.create();
captainAmerica.set('fullName', "William Burnside");
captainAmerica.get('firstName'); // William
captainAmerica.get('lastName'); // Burnside
```

Ember will call the computed property for both setters and getters, so if you want to use a computed property as a setter, you'll need to check the number of arguments to determine whether it is being called as a getter or a setter. Note that if a value is returned from the setter, it will be cached as the property's value.

Computed Properties and Aggregate Data with @each

Often, you may have a computed property that relies on all of the items in an array to determine its value. For example, you may want to count all of the todo items in a controller to determine how many of them are completed.

Here's what that computed property might look like:

```
App.TodosController = Ember.Controller.extend({
  todos: [
    Ember.Object.create({ isDone: true }),
    Ember.Object.create({ isDone: false }),
    Ember.Object.create({ isDone: true })
  ],

  remaining: function() {
    var todos = this.get('todos');
    return todos.filterBy('isDone', false).get('length');
  }.property('todos.@each.isDone')
});
```

Note here that the dependent key (`todos.@each.isDone`) contains the special key `@each`. This instructs Ember.js to update bindings and fire observers for this computed property when one of the following four events occurs:

1. The `isDone` property of any of the objects in the `todos` array changes.
2. An item is added to the `todos` array.
3. An item is removed from the `todos` array.
4. The `todos` property of the controller is changed to a different array.

In the example above, the `remaining` count is `1`:

```
App.todosController = App.TodosController.create();
App.todosController.get('remaining');
// 1
```

If we change the todo's `isDone` property, the `remaining` property is updated automatically:

```
var todos = App.todosController.get('todos');
var todo = todos.objectAt(1);
todo.set('isDone', true);

App.todosController.get('remaining');
// 0

todo = Ember.Object.create({ isDone: false });
todos.pushObject(todo);

App.todosController.get('remaining');
// 1
```

Note that @each only works one level deep. You cannot use nested forms like todos.@each.owner.name or todos.@each.owner.@each.name.

Observers

Ember supports observing any property, including computed properties. You can set up an observer on an object by using the observes method on a function:

```
Person = Ember.Object.extend({
  // these will be supplied by `create`
  firstName: null,
  lastName: null,

  fullName: function() {
    var firstName = this.get('firstName');
    var lastName = this.get('lastName');

    return firstName + ' ' + lastName;
  }.property('firstName', 'lastName'),

  fullNameChanged: function() {
    // deal with the change
  }.observes('fullName').on('init')
});

var person = Person.create({
  firstName: 'Yehuda',
  lastName: 'Katz'
});

person.set('firstName', 'Brohuda'); // observer will fire
```

Because the fullName computed property depends on firstName, updating firstName will fire observers on fullName as well.

Observers and asynchrony

Observers in Ember are currently synchronous. This means that they will fire as soon as one of the properties they observe changes. Because of this, it is easy to introduce bugs where properties are not yet synchronized:

```
Person.reopen({
  lastNameChanged: function() {
    // The observer depends on lastName and so does fullName. Because observers
    // are synchronous, when this function is called the value of fullName is
    // not updated yet so this will log the old value of fullName
    console.log(this.get('fullName'));
  }.observes('lastName')
});
```

This synchronous behaviour can also lead to observers being fired multiple times when observing multiple properties:

```
Person.reopen({
  partOfNameChanged: function() {
    // Because both firstName and lastName were set, this observer will fire twi\
ce.
  }.observes('firstName', 'lastName')
});

person.set('firstName', 'John');
person.set('lastName', 'Smith');
```

To get around these problems, you should make use of `Ember.run.once`. This will ensure that any processing you need to do only happens once, and happens in the next run loop once all bindings are synchronized:

```
Person.reopen({
  partOfNameChanged: function() {
    Ember.run.once(this, 'processFullName');
  }.observes('firstName', 'lastName'),

  processFullName: function() {
    // This will only fire once if you set two properties at the same time, and
    // will also happen in the next run loop once all properties are synchronized
    console.log(this.get('fullName'));
  }
});

person.set('firstName', 'John');
person.set('lastName', 'Smith');
```

Observers and object initialization

Observers never fire until after the initialization of an object is complete.

If you need an observer to fire as part of the initialization process, you cannot rely on the side effect of set. Instead, specify that the observer should also run after init by using `.on('init')`:

```
App.Person = Ember.Object.extend({
  init: function() {
    this.set('salutation', "Mr/Ms");
  },

  salutationDidChange: function() {
    // some side effect of salutation changing
  }.observes('salutation').on('init')
});
```

Unconsumed Computed Properties Do Not Trigger Observers

If you never `get` a computed property, its observers will not fire even if its dependent keys change. You can think of the value changing from one unknown value to another.

This doesn't usually affect application code because computed properties are almost always observed at the same time as they are fetched. For example, you get the value of a computed property, put it in DOM (or draw it with D3), and then observe it so you can update the DOM once the property changes.

If you need to observe a computed property but aren't currently retrieving it, just get it in your init method.

Without prototype extensions

You can define inline observers by using the `Ember.observer` method if you are using Ember without prototype extensions:

```
Person.reopen({
  fullNameChanged: Ember.observer('fullName', function() {
    // deal with the change
  })
});
```

Outside of class definitions

You can also add observers to an object outside of a class definition using addObserver:

```
person.addObserver('fullName', function() {
  // deal with the change
});
```

Bindings

A binding creates a link between two properties such that when one changes, the other one is updated to the new value automatically. Bindings can connect properties on the same object, or across two different objects. Unlike most other frameworks that include some sort of binding implementation, bindings in Ember.js can be used with any object, not just between views and models.

The easiest way to create a two-way binding is to use a computed alias, that specifies the path to another object.

```
wife = Ember.Object.create({
  householdIncome: 80000
});

husband = Ember.Object.create({
  wife: wife,
  householdIncome: Ember.computed.alias('wife.householdIncome')
});

husband.get('householdIncome'); // 80000

// Someone gets raise.
husband.set('householdIncome', 90000);
wife.get('householdIncome'); // 90000
```

Note that bindings don't update immediately. Ember waits until all of your application code has finished running before synchronizing changes, so you can change a bound property as many times as you'd like without worrying about the overhead of syncing bindings when values are transient.

One-Way Bindings

A one-way binding only propagates changes in one direction. Often, one-way bindings are just a performance optimization and you can safely use a two-way binding (as, of course, two-way bindings are de facto one-way bindings if you only ever change one side). Sometimes one-way bindings are useful to achieve specific behaviour such as a default that is the same as another property but can be overriden (e.g. a shipping address that starts the same as a billing address but can later be changed)

```
user = Ember.Object.create({
  fullName: "Kara Gates"
});

userView = Ember.View.create({
  user: user,
  userName: Ember.computed.oneWay('user.fullName')
});

// Changing the name of the user object changes
// the value on the view.
user.set('fullName', "Krang Gates");
// userView.userName will become "Krang Gates"

// ...but changes to the view don't make it back to
// the object.
userView.set('userName', "Truckasaurus Gates");
user.get('fullName'); // "Krang Gates"
```

Reopening Classes and Instances

You don't need to define a class all at once. You can reopen a class and define new properties using the `reopen` method.

```
Person.reopen({
  isPerson: true
});

Person.create().get('isPerson') // true
```

When using `reopen`, you can also override existing methods and call `this._super`.

```
Person.reopen({
  // override `say` to add an ! at the end
  say: function(thing) {
    this._super(thing + "!");
  }
});
```

`reopen` is used to add instance methods and properties that are shared across all instances of a class. It does not add methods and properties to a particular instance of a class as in vanilla JavaScript (without using prototype).

But when you need to create class methods or add properties to the class itself you can use `reopenClass`.

```
Person.reopenClass({
  createMan: function() {
    return Person.create({isMan: true})
  }
});

Person.createMan().get('isMan') // true
```

Bindings, Observers, Computed Properties: What Do I Use When?

Sometimes new users are confused about when to use computed properties, bindings and observers. Here are some guidelines to help:

1. Use *computed properties* to build a new property by synthesizing other properties. Computed properties should not contain application behavior, and should generally not cause any side-effects when called. Except in rare cases, multiple calls to the same computed property should always return the same value (unless the properties it depends on have changed, of course.)
2. *Observers* should contain behavior that reacts to changes in another property. Observers are especially useful when you need to perform some behavior after a binding has finished synchronizing.
3. *Bindings* are most often used to ensure objects in two different layers are always in sync. For example, you bind your views to your controller using Handlebars.

Application

Creating an Application

The first step to creating an Ember.js application is to make an instance of `Ember.Application` and assign it to a global variable.

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

Most people call their application `App`, but you can call it whatever makes the most sense to you. Just make sure it starts with a capital letter.

What does creating an `Ember.Application` instance get you?

1. It is 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.
2. It adds event listeners to the document and is responsible for delegating events to your views. (See The View Layer[128] for a detailed description.)
3. It automatically renders the application template[129].
4. It automatically creates a router and begins routing, choosing which template and model to display based on the current URL.

[128] http://emberjs.com/guides/understanding-ember/the-view-layer

[129] http://emberjs.com/guides/templates/the-application-template

Templates

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.

For more information about how outlets are filled in by the router, see Routing[130].

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 and it will automatically be compiled and appended to the screen.

[130] http://emberjs.com/guides/routing

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

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

Handlebars Basics

Ember.js uses the Handlebars templating library[131] to power your app's user interface. Handlebars templates are just like regular HTML, but also give you the ability to embed expressions that change what is displayed.

We take Handlebars and extend it with many powerful features. It may help to think of your Handlebars templates as an HTML-like DSL for describing the user interface of your app. And, 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.

If you'd prefer an indentation-based alternative to Handlebars syntax, try Emblem.js[132], but make sure you're comfortable with Handlebars first!

Defining Templates

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, like so:

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

This template will be compiled automatically and become your application template[133], which will be displayed on the page when your app loads.

You can also define templates by name that can be used later. For example, you may want to define a reusable control that is used in many different places in your user interface. To tell Ember.js to save the template for later, instead of displaying it immediately, you can add the `data-template-name` attribute:

[131] http://www.handlebarsjs.com

[132] http://www.emblemjs.com

[133] http://emberjs.com/guides/templates/the-application-template

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

If you are using build tools to manage your application's assets, most will know how to precompile Handlebars templates and make them available to Ember.js.

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, like this:

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

This would look up the `firstName` and `lastName` properties from the controller, insert them into the HTML described in the template, then put them into the DOM.

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>!
```

These expressions (and the other Handlebars features you will learn about next) 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.

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).

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}}
```

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

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

`{{#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

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.

The above example will print a list like this:

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

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.

There is an alternative form of `{{#each}}` that 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>
```

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 you pass to it. 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 Element Attributes

In addition to normal text, you may also want to have your templates contain HTML elements whose attributes are bound to the controller.

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>
```

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}}>
```

If `isAdministrator` is `true`, Handlebars will produce the following HTML element:

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

If `isAdministrator` is `false`, Handlebars will produce the following:

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

Adding data attributes

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

```
{{#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\
">
```

There are two ways to enable support for data attributes. One way would be to add an attribute binding on the view, e.g. `Ember.LinkView` or `Ember.TextField` for the specific attribute:

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

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

Now the same handlebars code above renders the following HTML:

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

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

You can also automatically bind data attributes on the base view with the following:

```
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>
```

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

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

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

```
<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>
```

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>
```

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>
```

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"`.

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[134]. By default, Ember.js will replace each segment with the value of the corresponding object's `id` property. If there is no model to pass to the helper, you can provide an explicit identifier value instead. The value will be filled into the dynamic segment[135] of the route, and will make sure that the `model` hook is triggered.
- 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" });
    });
  });
});
```

```
<!-- 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:

[134] http://emberjs.com/guides/routing/defining-your-routes/#toc_dynamic-segments

[135] http://emberjs.com/guides/routing/defining-your-routes/#toc_dynamic-segments

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

In the above example, 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.

Using link-to as an inline helper

In addition to being used as a block expression, the `link-to` helper can also be used in inline form by specifying the link text as the first argument to the helper:

```
A link in {{#link-to 'index'}}Block Expression Form{{/link-to}},
and a link in {{link-to 'Inline Form' 'index'}}.
```

The output of the above would be:

```
A link in <a href='/'>Block Expression Form</a>,
and a link in <a href='/'>Inline Form</a>.
```

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>
```

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. For example, imagine that you have a template that shows a blog post, and supports expanding the post with additional information.

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.

```
<!-- 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}}
```

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

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

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

Note that actions may be attached to any element of the DOM, but not all respond to the `click` event. For example, if an action is attached to an `a` link without an `href` attribute, or to a `div`, some browsers won't execute the associated function. If it's really needed to define actions over such elements, a CSS workaround exists to make them clickable, `cursor: pointer`. For example:

```
[data-ember-action] {
  cursor: pointer;
}
```

Action Bubbling

By default, the `{{action}}` helper triggers a method on the template's controller, as illustrated above.

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, where the currently active leaf route will be given a chance to handle the action.

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. In the case of a controller, while there is deprecated support for triggering a method directly on the controller, it is strongly recommended that you put your action handling methods inside an `actions` hash for forward compatibility.

```
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.

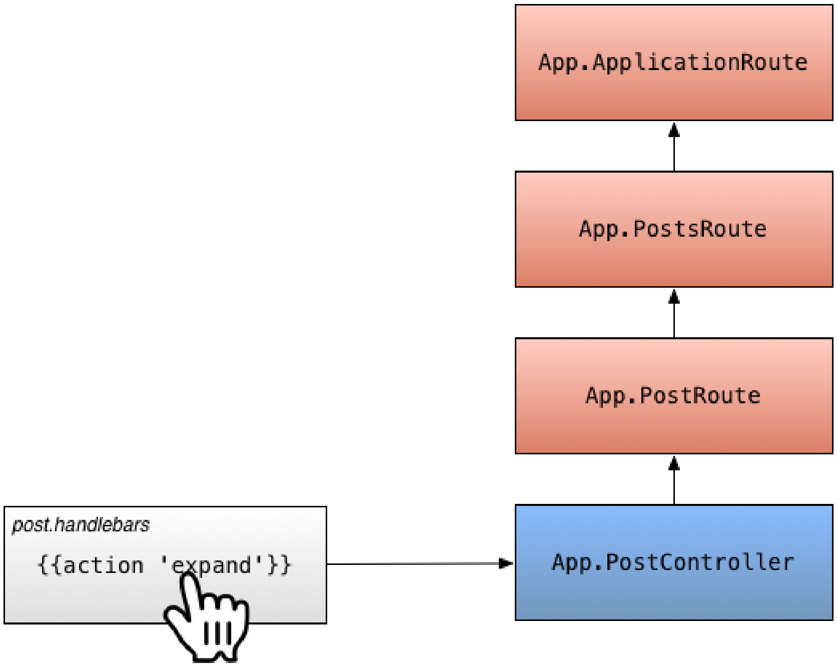
To continue bubbling the action, you must return true from the handler:

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

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

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.



```
{{action 'expand'}}
```

Action Bubbling

This allows you to create a button that has different behavior based on where you are in the application. 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.

For example, if the `post` argument was passed:

```
<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>
```

You should use the normalized event names listed in the View guide[136]. In general, two-word event names (like `keypress`) become `keyPress`.

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.

[136] http://emberjs.com/api/classes/Ember.TextField.html

```
<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.

For example, if you have a ⊠ button inside of a link, you will want to ensure that if the user clicks on the ⊠, that the link is not clicked.

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

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. (Note: in the case of an Ember.Component, the default target is the component itself.)

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>
```

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

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

Note that actions sent to views in this way do not bubble up the currently rendered view hierarchy. If you want to handle the action in a parent view, use the following:

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

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[137] and Ember.Checkbox[138] views, while `{{textarea}}` wraps Ember.TextArea[139]. 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 the following standard `<input>` attributes within the input helper:

<table> <tr><td>readonly</td><td>required</td><td>autofocus</td></tr> <tr><td>value</td><td>placeholder<
<tr><td>size</td><td>tabindex</td><td>maxlength</td></tr> <tr><td>name</td><td>min</td><td>max</td></tr>
<tr><td>pattern</td><td>accept</td><td>autocomplete</td></tr> <tr><td>autosave</td><td>formaction</td><
<tr><td>formmethod</td><td>formnovalidate</td><td>formtarget</td></tr> <tr><td>height</td><td>inputmode

[137] http://emberjs.com/api/classes/Ember.TextField.html
[138] http://emberjs.com/api/classes/Ember.Checkbox.html
[139] http://emberjs.com/api/classes/Ember.TextArea.html

<tr><td>`step`</td><td>`width`</td><td>`form`</td></tr> <tr><td>`selectionDirection`</td><td>`spellcheck`</td><td
</table>

If these attributes are set to a quoted string, their values will be set directly on the element, as in the previous example. However, when left unquoted, these values will be bound to a property on the template's current rendering context. For example:

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

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

Actions

To dispatch an action on specific events, such as `enter` or `key-press`, use the following

```
{{input value=firstName action="updateFirstName" on="key-press"}}
```

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`
- `autofocus`
- `form`

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:

- `value`
- `name`
- `rows`
- `cols`
- `placeholder`
- `disabled`
- `maxlength`
- `tabindex`
- `selectionEnd`
- `selectionStart`
- `selectionDirection`
- `wrap`
- `readonly`
- `autofocus`
- `form`
- `spellcheck`
- `required`

Extending Built-In Controls

See the Built-in Views[140] section of these guides to learn how to further extend these views.

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:

[140] http://emberjs.com/guides/views/built-in-views

```
{{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

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>
```

```
<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("l\
astName");
  }).property("firstName","lastName")
})
```

```
<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")

For more information, see Inserting Views in Templates[141]

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

`{{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:

[141] http://emberjs.com/guides/views/inserting-views-in-templates

```
<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.

Comparison Table

General

<table> <thead> <tr> <th>Helper</th> <th>Template</th> <th>Model</th> <th>View</th> <th>Controller</th> </tr> </thead> <tbody> <tr> <td><code>{{partial}}</code></td> <td>Specified Template</td> <td>Current

Model</td> <td>Current View</td> <td>Current Controller</td> </tr> <tr> <td><code>{{view}}</code></td> <td>View’s Template</td> <td>Current Model</td> <td>Specified View</td> <td>Current Controller</td> </tr> <tr> <td><code>{{render}}</code></td> <td>View’s Template</td> <td>Specified Model</td> <td>Specified View</td> <td>Specified Controller</td> </tr> </tbody> </table>

Specific

<table> <thead> <tr> <th>Helper</th> <th>Template</th> <th>Model</th> <th>View</th> <th>Controller</th> </tr> </thead> <tbody> <tr> <td><code>{{partial “author”}}</code></td> <td><code>author.hbs</code></td> <td>Post</td> <td><code>App.PostView</code></td> <td><code>App.PostController</code></td> </tr> <tr> <td><code>{{view “author”}}</code></td> <td><code>author.hbs</code></td> <td>Post</td> <td><code>App.AuthorView</code></td> <td><code>App.PostController</code></td> </tr> <tr> <td><code>{{rend
“author” author}}</code></td> <td><code>author.hbs</code></td> <td>Author</td> <td><code>App.AuthorView</
<td><code>App.AuthorController</code></td> </tr> </tbody> </table>

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.

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!

Anywhere 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 "calendar"}}
```

Routing

Introduction

Routing

As users interact with your application, it moves through many different states. Ember.js gives you helpful tools for managing that state in a way that scales with your application.

To understand why this is important, imagine we are writing a web app for managing a blog. At any given time, we should be able to answer questions like: *Is the user currently logged in? Are they an admin user? What post are they looking at? Is the settings screen open? Are they editing the current post?*

In Ember.js, each of the possible states in your application is represented by a URL. Because all of the questions we asked above— *Are we logged in? What post are we looking at?* —are encapsulated by route handlers for the URLs, answering them is both simple and accurate.

At any given time, your application has one or more *active route handlers*. The active handlers can change for one of two reasons:

1. The user interacted with a view, which generated an event that caused the URL to change.
2. The user changed the URL manually (e.g., via the back button), or the page was loaded for the first time.

When the current URL changes, the newly active route handlers may do one or more of the following:

1. Conditionally redirect to a new URL.
2. Update a controller so that it represents a particular model.
3. Change the template on screen, or place a new template into an existing outlet.

Logging Route Changes

As your application increases in complexity, it can be helpful to see exactly what is going on with the router. To have Ember write out transition events to the log, simply modify your `Ember.Application`:

```
App = Ember.Application.create({
  LOG_TRANSITIONS: true
});
```

Specifying a Root URL

If your Ember application is one of multiple web applications served from the same domain, it may be necessary to indicate to the router what the root URL for your Ember application is. By default, Ember will assume it is served from the root of your domain.

If for example, you wanted to serve your blogging application from www.emberjs.com/blog/, it would be necessary to specify a root URL of `/blog/`.

This can be achieved by setting the rootURL on the router:

```
App.Router.reopen({
  rootURL: '/blog/'
});
```

Defining Your Routes

When your application starts, the router is responsible for displaying templates, loading data, and otherwise setting up application state. It does so by matching the current URL to the *routes* that you've defined.

The map[142] method of your Ember application's router can be invoked to define URL mappings. When calling `map`, you should pass a function that will be invoked with the value `this` set to an object which you can use to create routes[143] and resources[144].

```
App.Router.map(function() {
  this.route("about", { path: "/about" });
  this.route("favorites", { path: "/favs" });
});
```

Now, when the user visits `/about`, Ember.js will render the `about` template. Visiting `/favs` will render the `favorites` template.

<aside> **Heads up!** You get a few routes for free: the `ApplicationRoute` and the `IndexRoute` (corresponding to the `/` path). See below for more details. </aside>

Note that you can leave off the path if it is the same as the route name. In this case, the following is equivalent to the above example:

[142] http://emberjs.com/api/classes/Ember.Router.html#method_map

[143] http://emberjs.com/guides/routing/defining-your-routes/

[144] http://emberjs.com/guides/routing/defining-your-routes/#toc_resources

```
App.Router.map(function() {
  this.route("about");
  this.route("favorites", { path: "/favs" });
});
```

Inside your templates, you can use `{{link-to}}` to navigate between routes, using the name that you provided to the `route` method (or, in the case of `/`, the name `index`).

```
{{#link-to 'index'}}<img class="logo">{{/link-to}}

<nav>
  {{#link-to 'about'}}About{{/link-to}}
  {{#link-to 'favorites'}}Favorites{{/link-to}}
</nav>
```

The `{{link-to}}` helper will also add an `active` class to the link that points to the currently active route.

You can customize the behavior of a route by creating an `Ember.Route` subclass. For example, to customize what happens when your user visits `/`, create an `App.IndexRoute`:

```
App.IndexRoute = Ember.Route.extend({
  setupController: function(controller) {
    // Set the IndexController's `title`
    controller.set('title', "My App");
  }
});
```

The `IndexController` is the starting context for the `index` template. Now that you've set `title`, you can use it in the template:

```
<!-- get the title from the IndexController -->
<h1>{{title}}</h1>
```

(If you don't explicitly define an `App.IndexController`, Ember.js will automatically generate one for you.)

Ember.js automatically figures out the names of the routes and controllers based on the name you pass to `this.route`.

<table> <thead> <tr> <th>URL</th> <th>Route Name</th> <th>Controller</th> <th>Route</th> <th>Template</th> </tr> </thead> <tr> <td><code>/</code></td> <td><code>index</code></td> <td><code>IndexController</code></td> <td><code>IndexRoute</code></td> <td><code>index</code></td>

</tr> <tr> <td><code>/about</code></td> <td><code>about</code></td> <td><code>AboutController</code></td> <td><code>AboutRoute</code></td> <td><code>about</code></td> </tr> <tr> <td><code>/favs</code></td> <td><code>favorites</code></td> <td><code>FavoritesController</code></td> <td><code>FavoritesRoute</code>< <td><code>favorites</code></td> </tr> </table>

Resources

You can define groups of routes that work with a resource:

```
App.Router.map(function() {
  this.resource('posts', { path: '/posts' }, function() {
    this.route('new');
  });
});
```

As with `this.route`, you can leave off the path if it's the same as the name of the route, so the following router is equivalent:

```
App.Router.map(function() {
  this.resource('posts', function() {
    this.route('new');
  });
});
```

This router creates three routes:

<table> <thead> <tr> <th>URL</th> <th>Route Name</th> <th>Controller</th> <th>Route</th> <th>Template</th> </tr> </thead> <tr> <td><code>/</code></td> <td><code>index</code></td> <td><code>IndexController</code></td> <td><code>IndexRoute</code></td> <td><code>index</code></td> </tr> <tr> <td>N/A</td> <td><code>posts</code><sup>1</sup></td> <td><code>PostsController</code></td> <td><code>PostsRoute</code></td> <td><code>posts</code></td> </tr> <tr> <td><code>/posts</code></td> <td><code>posts.index</code></code></td> <td><code>PostsController</code><br>⊠<code>PostsIndexController< <td><code>PostsRoute</code><br>⊠<code>PostsIndexRoute</code></td> <td><code>posts</code><br>⊠<code>pos </tr> <tr> <td><code>/posts/new</code></td> <td><code>posts.new</code></td> <td><code>PostsController</cod <td><code>PostsRoute</code><br>⊠<code>PostsNewRoute</code></td> <td><code>posts</code><br>⊠<code>post </tr> </table>

<small><sup>1</sup> Transitioning to `posts` or creating a link to `posts` is equivalent to transitioning to `posts.index` or linking to `posts.index`</small>

NOTE: If you define a resource using `this.resource` and **do not** supply a function, then the implicit `resource.index` route is **not** created. In that case, `/resource` will only use the `ResourceRoute`, `ResourceController`, and `resource` template.

Routes nested under a resource take the name of the resource plus their name as their route name. If you want to transition to a route (either via `transitionTo` or `{{#link-to}}`), make sure to use the full route name (`posts.new`, not `new`).

Visiting `/` renders the `index` template, as you would expect.

Visiting `/posts` is slightly different. It will first render the `posts` template. Then, it will render the `posts/index` template into the `posts` template's outlet.

Finally, visiting `/posts/new` will first render the `posts` template, then render the `posts/new` template into its outlet.

NOTE: You should use `this.resource` for URLs that represent a **noun**, and `this.route` for URLs that represent **adjectives** or **verbs** modifying those nouns. For example, in the code sample above, when specifying URLs for posts (a noun), the route was defined with `this.resource('posts')`. However, when defining the `new` action (a verb), the route was defined with `this.route('new')`.

Dynamic Segments

One of the responsibilities of a resource's route handler is to convert a URL into a model.

For example, if we have the resource `this.resource('posts');`, our route handler might look like this:

```
App.PostsRoute = Ember.Route.extend({
  model: function() {
    return this.store.find('posts');
  }
});
```

The `posts` template will then receive a list of all available posts as its context.

Because `/posts` represents a fixed model, we don't need any additional information to know what to retrieve. However, if we want a route to represent a single post, we would not want to have to hardcode every possible post into the router.

Enter *dynamic segments*.

A dynamic segment is a portion of a URL that starts with a `:` and is followed by an identifier.

```
App.Router.map(function() {
  this.resource('posts');
  this.resource('post', { path: '/post/:post_id' });
});

App.PostRoute = Ember.Route.extend({
  model: function(params) {
    return this.store.find('post', params.post_id);
  }
});
```

Because this pattern is so common, the above `model` hook is the default behavior.

For example, if the dynamic segment is `:post_id`, Ember.js is smart enough to know that it should use the model `App.Post` (with the ID provided in the URL). Specifically, unless you override `model`, the route will return `this.store.find('post', params.post_id)` automatically.

Not coincidentally, this is exactly what Ember Data expects. So if you use the Ember router with Ember Data, your dynamic segments will work as expected out of the box.

If your model does not use the `id` property in the URL, you should define a serialize method on your route:

```
App.Router.map(function() {
  this.resource('post', {path: '/posts/:post_slug'});
});

App.PostRoute = Ember.Route.extend({
  model: function(params) {
    // the server returns `{ slug: 'foo-post' }`
    return jQuery.getJSON("/posts/" + params.post_slug);
  },

  serialize: function(model) {
    // this will make the URL `/posts/foo-post`
    return { post_slug: model.get('slug') };
  }
});
```

The default `serialize` method inserts the model's `id` into the route's dynamic segment (in this case, `:post_id`).

Nested Resources

You can nest both routes and resources:

```
App.Router.map(function() {
  this.resource('post', { path: '/post/:post_id' }, function() {
    this.route('edit');
    this.resource('comments', function() {
      this.route('new');
    });
  });
});
```

This router creates five routes:

<div style=”overflow: auto”> <table> <thead> <tr> <th>URL</th> <th>Route Name</th> <th>Controller</th>
<th>Route</th> <th>Template</th> </tr> </thead> <tr> <td><code>/</code></td> <td><code>index</code></td>
<td><code>App.IndexController</code></td> <td><code>App.IndexRoute</code></td> <td><code>index</code></
</tr> <tr> <td>N/A</td> <td><code>post</code></td> <td><code>App.PostController</code></td>
<td><code>App.PostRoute</code></td> <td><code>post</code></td> </tr> <tr> <td><code>/post/:post_id<sup>2</sup></code></td> <td><code>post.index</code></td> <td><code>App.PostIndexController</code></td>
<td><code>App.PostIndexRoute</code></td> <td><code>post/index</code></td> </tr> <tr> <td><code>/post/:post_
id/edit</code></td> <td><code>post.edit</code></td> <td><code>App.PostEditController</code></td>
<td><code>App.PostEditRoute</code></td> <td><code>post/edit</code></td> </tr> <tr> <td>N/A</td>
<td><code>comments</code></td> <td><code>App.CommentsController</code></td> <td><code>App.Comments
<td><code>comments</code></td> </tr> <tr> <td><code>/post/:post_id/comments</code></td> <td><code>comme
<td><code>App.CommentsIndexController</code></td> <td><code>App.CommentsIndexRoute</code></td>
<td><code>comments/index</code></td> </tr> <tr> <td><code>/post/:post_id/comments/new</code></td>
<td><code>comments.new</code></td> <td><code>App.CommentsNewController</code></td> <td><code>App.C
<td><code>comments/new</code></td> </tr> </table> </div>

<small><sup>2</sup> :post_id is the post’s id. For a post with id = 1, the route will be: /post/1</small>

The comments template will be rendered in the post outlet. All templates under comments (comments/index and comments/new) will be rendered in the comments outlet.

The route, controller, and view class names for the comments resource are not prefixed with Post. Resources always reset the namespace, ensuring that the classes can be re-used between multiple parent resources and that class names don’t get longer the deeper nested the resources are.

You are also able to create deeply nested resources in order to preserve the namespace on your routes:

```
App.Router.map(function() {
  this.resource('foo', function() {
    this.resource('foo.bar', { path: '/bar' }, function() {
      this.route('baz'); // This will be foo.bar.baz
    });
  });
});
```

This router creates the following routes:

<div style=”overflow: auto”> <table> <thead> <tr> <th>URL</th> <th>Route Name</th> <th>Controller</th>
<th>Route</th> <th>Template</th> </tr> </thead> <tr> <td><code>/</code></td> <td><code>index</code></td>
<td><code>App.IndexController</code></td> <td><code>App.IndexRoute</code></td> <td><code>index</code></
</tr> <tr> <td><code>/foo</code></td> <td><code>foo.index</code></td> <td><code>App.FooIndexController</co
<td><code>App.FooIndexRoute</code></td> <td><code>foo/index</code></td> </tr> <tr> <td><code>/foo/bar</co
<td><code>foo.bar.index</code></td> <td><code>App.FooBarIndexController</code></td> <td><code>App.FooBar
<td><code>foo/bar/index</code></td> </tr> <tr> <td><code>/foo/bar/baz</code></td> <td><code>foo.bar.baz</cod
<td><code>App.FooBarBazController</code></td> <td><code>App.FooBarBazRoute</code></td> <td><code>foo/b
</tr> </table> </div>

Initial routes

A few routes are immediately available within your application:

- `App.ApplicationRoute` is entered when your app first boots up. It renders the `application` template.
- `App.IndexRoute` is the default route, and will render the `index` template when the user visits `/` (unless `/` has been overridden by your own custom route).

Remember, these routes are part of every application, so you don’t need to specify them in `App.Router.map`.

Wildcard / globbing routes

You can define wildcard routes that will match multiple routes. This could be used, for example, if you’d like a catchall route which is useful when the user enters an incorrect URL not managed by your app.

```
App.Router.map(function() {
  this.route('catchall', {path: '/*wildcard'});
});
```

Like all routes with a dynamic segment, you must provide a context when using a `{{link-to}}` or `transitionTo` to programatically enter this route.

```
App.ApplicationRoute = Ember.Route.extend({
  actions: {
    error: function () {
      this.transitionTo('catchall', "application-error");
    }
  }
});
```

With this code, if an error bubbles up to the Application route, your application will enter the `catchall` route and display `/application-error` in the URL.

Generated Objects

As explained in the routing guide[145], whenever you define a new route, Ember.js attempts to find corresponding Route, Controller, View, and Template classes named according to naming conventions. If an implementation of any of these objects is not found, appropriate objects will be generated in memory for you.

Generated routes

Given you have the following route:

```
App.Router.map(function() {
  this.resource('posts');
});
```

When you navigate to `/posts`, Ember.js looks for `App.PostsRoute`. If it doesn't find it, it will automatically generate an `App.PostsRoute` for you.

Custom Generated Routes

You can have all your generated routes extend a custom route. If you define `App.Route`, all generated routes will be instances of that route.

[145] http://emberjs.com/guides/routing/specifying-a-routes-model

Generated Controllers

If you navigate to route `posts`, Ember.js looks for a controller called `App.PostsController`. If you did not define it, one will be generated for you.

Ember.js can generate three types of controllers: `Ember.ObjectController`, `Ember.ArrayController`, and `Ember.Controller`.

The type of controller Ember.js chooses to generate for you depends on your route's `model` hook:

- If it returns an object (such as a single record), an ObjectController[146] will be generated.
- If it returns an array, an ArrayController[147] will be generated.
- If it does not return anything, an instance of `Ember.Controller` will be generated.

Custom Generated Controllers

If you want to customize generated controllers, you can define your own `App.Controller`, `App.ObjectController` and `App.ArrayController`. Generated controllers will extend one of these three (depending on the conditions above).

Generated Views and Templates

A route also expects a view and a template. If you don't define a view, a view will be generated for you.

A generated template is empty. If it's a resource template, the template will simply act as an `outlet` so that nested routes can be seamlessly inserted. It is equivalent to:

```
{{outlet}}
```

Specifying A Routes Model

Templates in your application are backed by models. But how do templates know which model they should display?

For example, if you have a `photos` template, how does it know which model to render?

This is one of the jobs of an `Ember.Route`. You can tell a template which model it should render by defining a route with the same name as the template, and implementing its `model` hook.

For example, to provide some model data to the `photos` template, we would define an `App.PhotosRoute` object:

[146] http://emberjs.com/guides/templates/links
[147] http://emberjs.com/guides/controllers/representing-multiple-models-with-arraycontroller

```
App.PhotosRoute = Ember.Route.extend({
  model: function() {
    return [{
      title: "Tomster",
      url: "http://emberjs.com/images/about/ember-productivity-sm.png"
    }, {
      title: "Eiffel Tower",
      url: "http://emberjs.com/images/about/ember-structure-sm.png"
    }];
  }
});
```

JS Bin[148]

Asynchronously Loading Models

In the above example, the model data was returned synchronously from the `model` hook. This means that the data was available immediately and your application did not need to wait for it to load, in this case because we immediately returned an array of hardcoded data.

Of course, this is not always realistic. Usually, the data will not be available synchronously, but instead must be loaded asynchronously over the network. For example, we may want to retrieve the list of photos from a JSON API available on our server.

In cases where data is available asynchronously, you can just return a promise from the `model` hook, and Ember will wait until that promise is resolved before rendering the template.

If you're unfamiliar with promises, the basic idea is that they are objects that represent eventual values. For example, if you use jQuery's `getJSON()` method, it will return a promise for the JSON that is eventually returned over the network. Ember uses this promise object to know when it has enough data to continue rendering.

For more about promises, see A Word on Promises[149] in the Asynchronous Routing guide.

Let's look at an example in action. Here's a route that loads the most recent pull requests sent to Ember.js on GitHub:

[148] http://jsbin.com/oLUTEd

[149] http://emberjs.com/guides/routing/asynchronous-routing/#toc_a-word-on-promises

```
App.PullRequestsRoute = Ember.Route.extend({
  model: function() {
    return Ember.$.getJSON('https://api.github.com/repos/emberjs/ember.js/pulls'\
);
  }
});
```

While this example looks like it's synchronous, making it easy to read and reason about, it's actually completely asynchronous. That's because jQuery's `getJSON()` method returns a promise. Ember will detect the fact that you've returned a promise from the `model` hook, and wait until that promise resolves to render the `pullRequests` template.

(For more information on jQuery's XHR functionality, see jQuery.ajax[150] in the jQuery documentation.)

Because Ember supports promises, it can work with any persistence library that uses them as part of its public API. You can also use many of the conveniences built in to promises to make your code even nicer.

For example, imagine if we wanted to modify the above example so that the template only displayed the three most recent pull requests. We can rely on promise chaining to modify the data returned from the JSON request before it gets passed to the template:

```
App.PullRequestsRoute = Ember.Route.extend({
  model: function() {
    var url = 'https://api.github.com/repos/emberjs/ember.js/pulls';
    return Ember.$.getJSON(url).then(function(data) {
      return data.splice(0, 3);
    });
  }
});
```

Setting Up Controllers with the Model

So what actually happens with the value you return from the `model` hook?

By default, the value returned from your `model` hook will be assigned to the `model` property of the associated controller. For example, if your `App.PostsRoute` returns an object from its `model` hook, that object will be set as the `model` property of the `App.PostsController`.

(This, under the hood, is how templates know which model to render: they look at their associated controller's `model` property. For example, the `photos` template will render whatever the `App.PhotosController`'s `model` property is set to.)

[150] http://api.jquery.com/jQuery.ajax/

See the Setting Up a Controller guide[151] to learn how to change this default behavior. Note that if you override the default behavior and do not set the model property on a controller, your template will not have any data to render!

Dynamic Models

Some routes always display the same model. For example, the /photos route will always display the same list of photos available in the application. If your user leaves this route and comes back later, the model does not change.

However, you will often have a route whose model will change depending on user interaction. For example, imagine a photo viewer app. The /photos route will render the photos template with the list of photos as the model, which never changes. But when the user clicks on a particular photo, we want to display that model with the photo template. If the user goes back and clicks on a different photo, we want to display the photo template again, this time with a different model.

In cases like this, it's important that we include some information in the URL about not only which template to display, but also which model.

In Ember, this is accomplished by defining routes with *dynamic segments*.

A dynamic segment is a part of the URL that is filled in by the current model's ID. Dynamic segments always start with a colon (:). Our photo example might have its photo route defined like this:

```
App.Router.map(function() {
  this.resource('photo', { path: '/photos/:photo_id' });
});
```

In this example, the photo route has a dynamic segment :photo_id. When the user goes to the photo route to display a particular photo model (usually via the {{link-to}} helper), that model's ID will be placed into the URL automatically.

See Links[152] for more information about linking to a route with a model using the {{link-to}} helper.

For example, if you transitioned to the photo route with a model whose id property was 47, the URL in the user's browser would be updated to:

```
/photos/47
```

What happens if the user visits your application directly with a URL that contains a dynamic segment? For example, they might reload the page, or send the link to a friend, who clicks on it.

[151] http://emberjs.com/guides/routing/specifying-a-routes-model

[152] http://emberjs.com/guides/templates/links

At that point, because we are starting the application up from scratch, the actual JavaScript model object to display has been lost; all we have is the ID from the URL.

Luckily, Ember will extract any dynamic segments from the URL for you and pass them as a hash to the `model` hook as the first argument:

```
App.Router.map(function() {
  this.resource('photo', { path: '/photos/:photo_id' });
});

App.PhotoRoute = Ember.Route.extend({
  model: function(params) {
    return Ember.$.getJSON('/photos/'+params.photo_id);
  }
});
```

In the `model` hook for routes with dynamic segments, it's your job to turn the ID (something like `47` or `post-slug`) into a model that can be rendered by the route's template. In the above example, we use the photo's ID (`params.photo_id`) to construct a URL for the JSON representation of that photo. Once we have the URL, we use jQuery to return a promise for the JSON model data.

Note: A route with a dynamic segment will only have its `model` hook called when it is entered via the URL. If the route is entered through a transition (e.g. when using the link-to[153] Handlebars helper), then a model context is already provided and the hook is not executed. Routes without dynamic segments will always execute the model hook.

Refreshing your model

If your data represented by your model is being updated frequently, you may want to refresh it periodically:

JS Bin[154]

The controller can send an action to the Route; in this example above, the IndexController exposes an action `getLatest` which sends the route an action called `invalidateModel`. Calling the route's `refresh` method will force Ember to execute the model hook again.

Ember Data

Many Ember developers use a model library to make finding and saving records easier than manually managing Ajax calls. In particular, using a model library allows you to cache records that have been loaded, significantly improving the performance of your application.

[153] http://emberjs.com/guides/templates/links

[154] http://jsbin.com/sefuv

One popular model library built for Ember is Ember Data. To learn more about using Ember Data to manage your models, see the Models[155] guide.

Setting Up A Controller

Changing the URL may also change which template is displayed on screen. Templates, however, are usually only useful if they have some source of information to display.

In Ember.js, a template retrieves information to display from a controller.

Two built-in controllers—`Ember.ObjectController` and `Ember.ArrayController`—make it easy for a controller to present a model's properties to a template, along with any additional display-specific properties.

To tell one of these controllers which model to present, set its `model` property in the route handler's `setupController` hook.

```
App.Router.map(function() {
  this.resource('post', { path: '/posts/:post_id' });
});

App.PostRoute = Ember.Route.extend({
  // The code below is the default behavior, so if this is all you
  // need, you do not need to provide a setupController implementation
  // at all.
  setupController: function(controller, model) {
    controller.set('model', model);
  }
});
```

The `setupController` hook receives the route handler's associated controller as its first argument. In this case, the `PostRoute`'s `setupController` receives the application's instance of `App.PostController`.

To specify a controller other than the default, set the route's `controllerName` property:

```
App.SpecialPostRoute = Ember.Route.extend({
  controllerName: 'post'
});
```

As a second argument, it receives the route handler's model. For more information, see Specifying a Route's Model[156].

[155] http://emberjs.com/guides/models

[156] http://emberjs.com/guides/routing/specifying-a-routes-model

The default `setupController` hook sets the `model` property of the associated controller to the route handler's model.

If you want to configure a controller other than the controller associated with the route handler, use the `controllerFor` method:

```
App.PostRoute = Ember.Route.extend({
  setupController: function(controller, model) {
    this.controllerFor('topPost').set('model', model);
  }
});
```

Rendering A Template

One of the most important jobs of a route handler is rendering the appropriate template to the screen.

By default, a route handler will render the template into the closest parent with a template.

```
App.Router.map(function() {
  this.resource('posts');
});

App.PostsRoute = Ember.Route.extend();
```

If you want to render a template other than the one associated with the route handler, implement the `renderTemplate` hook:

```
App.PostsRoute = Ember.Route.extend({
  renderTemplate: function() {
    this.render('favoritePost');
  }
});
```

If you want to use a different controller than the route handler's controller, pass the controller's name in the `controller` option:

```
App.PostsRoute = Ember.Route.extend({
  renderTemplate: function() {
    this.render({ controller: 'favoritePost' });
  }
});
```

Ember allows you to name your outlets. For instance, this code allows you to specify two outlets with distinct names:

```
<div class="toolbar">{{outlet toolbar}}</div>
<div class="sidebar">{{outlet sidebar}}</div>
```

So, if you want to render your posts into the `sidebar` outlet, use code like this:

```
App.PostsRoute = Ember.Route.extend({
  renderTemplate: function() {
    this.render({ outlet: 'sidebar' });
  }
});
```

All of the options described above can be used together in whatever combination you'd like:

```
App.PostsRoute = Ember.Route.extend({
  renderTemplate: function() {
    var controller = this.controllerFor('favoritePost');

    // Render the `favoritePost` template into
    // the outlet `posts`, and use the `favoritePost`
    // controller.
    this.render('favoritePost', {
      outlet: 'posts',
      controller: controller
    });
  }
});
```

If you want to render two different templates into outlets of two different rendered templates of a route:

```
App.PostRoute = App.Route.extend({
  renderTemplate: function() {
    this.render('favoritePost', {   // the template to render
      into: 'posts',                // the template to render into
      outlet: 'posts',              // the name of the outlet in that template
      controller: 'blogPost'        // the controller to use for the template
    });
    this.render('comments', {
      into: 'favoritePost',
      outlet: 'comment',
      controller: 'blogPost'
    });
  }
});
```

Redirecting

Transitioning and Redirecting

Calling `transitionTo` from a route or `transitionToRoute` from a controller will stop any transition currently in progress and start a new one, functioning as a redirect. `transitionTo` takes parameters and behaves exactly like the link-to[157] helper:

- If you transition into a route without dynamic segments that route's `model` hook will always run.
- If the new route has dynamic segments, you need to pass either a *model* or an *identifier* for each segment. Passing a model will skip that segment's `model` hook. Passing an identifier will run the `model` hook and you'll be able to access the identifier in the params. See Links[158] for more detail.

Before the model is known

If you want to redirect from one route to another, you can do the transition in the `beforeModel` hook of your route handler.

[157] http://emberjs.com/guides/templates/links

[158] http://emberjs.com/guides/templates/links

```
App.Router.map(function() {
  this.resource('posts');
});

App.IndexRoute = Ember.Route.extend({
  beforeModel: function() {
    this.transitionTo('posts');
  }
});
```

After the model is known

If you need some information about the current model in order to decide about the redirection, you should either use the `afterModel` or the `redirect` hook. They receive the resolved model as the first parameter and the transition as the second one, and thus function as aliases. (In fact, the default implementation of `afterModel` just calls `redirect`.)

```
App.Router.map(function() {
  this.resource('posts');
  this.resource('post', { path: '/post/:post_id' });
});

App.PostsRoute = Ember.Route.extend({
  afterModel: function(posts, transition) {
    if (posts.get('length') === 1) {
      this.transitionTo('post', posts.get('firstObject'));
    }
  }
});
```

When transitioning to the `PostsRoute` if it turns out that there is only one post, the current transition will be aborted in favor of redirecting to the `PostRoute` with the single post object being its model.

Based on other application state

You can conditionally transition based on some other application state.

```
App.Router.map(function() {
  this.resource('topCharts', function() {
    this.route('choose', { path: '/' });
    this.route('albums');
    this.route('songs');
    this.route('artists');
    this.route('playlists');
  });
});

App.TopChartsChooseRoute = Ember.Route.extend({
  beforeModel: function() {
    var lastFilter = this.controllerFor('application').get('lastFilter');
    this.transitionTo('topCharts.' + (lastFilter || 'songs'));
  }
});

// Superclass to be used by all of the filter routes below
App.FilterRoute = Ember.Route.extend({
  activate: function() {
    var controller = this.controllerFor('application');
    controller.set('lastFilter', this.templateName);
  }
});

App.TopChartsSongsRoute = App.FilterRoute.extend();
App.TopChartsAlbumsRoute = App.FilterRoute.extend();
App.TopChartsArtistsRoute = App.FilterRoute.extend();
App.TopChartsPlaylistsRoute = App.FilterRoute.extend();
```

In this example, navigating to the / URL immediately transitions into the last filter URL that the user was at. The first time, it transitions to the /songs URL.

Your route can also choose to transition only in some cases. If the `beforeModel` hook does not abort or transition to a new route, the remaining hooks (`model`, `afterModel`, `setupController`, `renderTemplate`) will execute as usual.

Specifying The URL Type

By default the Router uses the browser's hash to load the starting state of your application and will keep it in sync as you move around. At present, this relies on a hashchange[159] event existing in the browser.

[159] http://caniuse.com/hashchange

Given the following router, entering `/#/posts/new` will take you to the `posts.new` route.

```
App.Router.map(function() {
  this.resource('posts', function() {
    this.route('new');
  });
});
```

If you want `/posts/new` to work instead, you can tell the Router to use the browser's history[160] API.

Keep in mind that your server must serve the Ember app at all the routes defined here.

```
App.Router.reopen({
  location: 'history'
});
```

Finally, if you don't want the browser's URL to interact with your application at all, you can disable the location API entirely. This is useful for testing, or when you need to manage state with your Router, but temporarily don't want it to muck with the URL (for example when you embed your application in a larger page).

```
App.Router.reopen({
  location: 'none'
});
```

Query Parameters

Query parameters are optional key-value pairs that appear to the right of the `?` in a URL. For example, the following URL has two query params, `sort` and `page`, with respective values `ASC` and `2`:

```
http://example.com/articles?sort=ASC&page=2
```

Query params allow for additional application state to be serialized into the URL that can't otherwise fit into the *path* of the URL (i.e. everything to the left of the `?`). Common use cases for query params include representing the current page, filter criteria, or sorting criteria.

[160] http://caniuse.com/history

Specifying Query Parameters

Query params can be declared on route-driven controllers, e.g. to configure query params that are active within the `articles` route, they must be declared on `ArticlesController`.

<aside> **Note**: The controller associated with a given route can be changed by specifying the `controllerName` property on that route. </aside>

Let's say we'd like to add a `category` query parameter that will filter out all the articles that haven't been categorized as popular. To do this, we specify `'category'` as one of `ArticlesController`'s `queryParams`:

```
App.ArticlesController = Ember.ArrayController.extend({
  queryParams: ['category'],
  category: null
});
```

This sets up a binding between the `category` query param in the URL, and the `category` property on `ArticlesController`. In other words, once the `articles` route has been entered, any changes to the `category` query param in the URL will update the `category` property on `ArticlesController`, and vice versa.

Now we just need to define a computed property of our category-filtered array that the `articles` template will render:

```
App.ArticlesController = Ember.ArrayController.extend({
  queryParams: ['category'],
  category: null,

  filteredArticles: function() {
    var category = this.get('category');
    var articles = this.get('model');

    if (category) {
      return articles.filterBy('category', category);
    } else {
      return articles;
    }
  }.property('category', 'model')
});
```

With this code, we have established the following behaviors:

1. If the user navigates to `/articles`, `category` will be `null`, so the articles won't be filtered.

2. If the user navigates to `/articles?category=recent`, `category` will be set to `"recent"`, so articles will be filtered.
3. Once inside the `articles` route, any changes to the `category` property on `ArticlesController` will cause the URL to update the query param. By default, a query param property change won't cause a full router transition (i.e. it won't call `model` hooks and `setupController`, etc.); it will only update the URL.

link-to Helper

The `link-to` helper supports specifying query params by way of the `query-params` subexpression helper.

```
// Explicitly set target query params
{{#link-to 'posts' (query-params direction="asc")}}Sort{{/link-to}}

// Binding is also supported
{{#link-to 'posts' (query-params direction=otherDirection)}}Sort{{/link-to}}
```

In the above examples, `direction` is presumably a query param property on the `PostsController`, but it could also refer to a `direction` property on any of the controllers associated with the `posts` route hierarchy, matching the leaf-most controller with the supplied property name.

<aside> **Note**: Subexpressions are only available in Handlebars 1.3 or later. </aside>

The link-to helper takes into account query parameters when determining its "active" state, and will set the class appropriately. The active state is determined by calculating whether the query params end up the same after clicking a link. You don't have to supply all of the current, active query params for this to be true.

transitionTo

`Route#transitionTo` (and `Controller#transitionToRoute`) now accepts a final argument, which is an object with the key `queryParams`.

```
this.transitionTo('post', object, {queryParams: {showDetails: true}});
this.transitionTo('posts', {queryParams: {sort: 'title'}});

// if you just want to transition the query parameters without changing the route
this.transitionTo({queryParams: {direction: 'asc'}});
```

You can also add query params to URL transitions:

```
this.transitionTo("/posts/1?sort=date&showDetails=true");
```

Opting into a full transition

Keep in mind that if the arguments provided to `transitionTo` or `link-to` only correspond to a change in query param values, and not a change in the route hierarchy, it is not considered a full transition, which means that hooks like `model` and `setupController` won't fire by default, but rather only controller properties will be updated with new query param values, as will the URL.

But some query param changes necessitate loading data from the server, in which case it is desirable to opt into a full-on transition. To opt into a full transition when a controller query param property changes, you can use the optional `queryParams` configuration hash on the `Route` associated with that controller, and set that query param's `refreshModel` config property to `true`:

```
App.ArticlesRoute = Ember.Route.extend({
  queryParams: {
    category: {
      refreshModel: true
    }
  },
  model: function(params) {
    // This gets called upon entering 'articles' route
    // for the first time, and we opt into refiring it upon
    // query param changes by setting `refreshModel:true` above.

    // params has format of { category: "someValueOrJustNull" },
    // which we can just forward to the server.
    return this.store.findQuery('articles', params);
  }
});

App.ArticlesController = Ember.ArrayController.extend({
  queryParams: ['category'],
  category: null
});
```

Update URL with `replaceState` instead

By default, Ember will use `pushState` to update the URL in the address bar in response to a controller query param property change, but if you would like to use `replaceState` instead (which prevents an additional item from being added to your browser's history), you can specify this on the `Route`'s `queryParams` config hash, e.g. (continued from the example above):

```
App.ArticlesRoute = Ember.Route.extend({
  queryParams: {
    category: {
      replace: true
    }
  }
});
```

Note that the name of this config property and its default value of `false` is similar to the `link-to` helper's, which also lets you opt into a `replaceState` transition via `replace=true`.

Map a controller's property to a different query param key

By default, specifying `foo` as a controller query param property will bind to a query param whose key is `foo`, e.g. `?foo=123`. You can also map a controller property to a different query param key using the following configuration syntax:

```
App.ArticlesController = Ember.ArrayController.extend({
  queryParams: {
    category: "articles_category"
  },
  category: null
});
```

This will cause changes to the `ArticlesController`'s `category` property to update the `articles_category` query param, and vice versa.

Note that query params that require additional customization can be provided along with strings in the `queryParams` array.

```
App.ArticlesController = Ember.ArrayController.extend({
  queryParams: [ "page", "filter", {
    category: "articles_category"
  }],
  category: null,
  page: 1,
  filter: "recent"
});
```

Default values and deserialization

In the following example, the controller query param property `page` is considered to have a default value of `1`.

```
App.ArticlesController = Ember.ArrayController.extend({
  queryParams: 'page',
  page: 1
});
```

This affects query param behavior in two ways:

1. Query param values are cast to the same datatype as the default value, e.g. a URL change from `/?page=3` to `/?page=2` will set `ArticlesController`'s `page` property to the number `2`, rather than the string `"2"`. The same also applies to boolean default values.
2. When a controller's query param property is currently set to its default value, this value won't be serialized into the URL. So in the above example, if `page` is `1`, the URL might look like `/articles`, but once someone sets the controller's `page` value to `2`, the URL will become `/articles?page=2`.

Sticky Query Param Values

By default, query param values in Ember are "sticky", in that if you make changes to a query param and then leave and re-enter the route, the new value of that query param will be preserved (rather than reset to its default). This is a particularly handy default for preserving sort/filter parameters as you navigate back and forth between routes.

Furthermore, these sticky query param values are remembered/restored according to the model loaded into the route. So, given a `team` route with dynamic segment `/:team_name` and controller query param "filter", if you navigate to `/badgers` and filter by `"rookies"`, then navigate to `/bears` and filter by `"best"`, and then navigate to `/potatoes` and filter by `"lamest"`, then given the following nav bar links,

```
{{#link-to 'team' 'badgers '}}Badgers{{/link-to}}
{{#link-to 'team' 'bears'    }}Bears{{/link-to}}
{{#link-to 'team' 'potatoes'}}Potatoes{{/link-to}}
```

the generated links would be

```
<a href="/badgers?filter=rookies">Badgers</a>
<a href="/bears?filter=best">Bears</a>
<a href="/potatoes?filter=lamest">Potatoes</a>
```

This illustrates that once you change a query param, it is stored and tied to the model loaded into the route.

If you wish to reset a query param, you have two options:

1. explicitly pass in the default value for that query param into `link-to` or `transitionTo`
2. use the `Route.resetController` hook to set query param values back to their defaults before exiting the route or changing the route's model

In the following example, the controller's `page` query param is reset to `1`, *while still scoped to the pre-transition `ArticlesRoute` model*. The result of this is that all links pointing back into the exited route will use the newly reset value `1` as the value for the `page` query param.

```
App.ArticlesRoute = Ember.Route.extend({
  resetController: function (controller, isExiting, transition) {
    if (isExiting) {
      // isExiting would be false if only the route's model was changing
      controller.set('page', 1);
    }
  }
});
```

In some cases, you might not want the sticky query param value to be scoped to the route's model but would rather reuse a query param's value even as a route's model changes. This can be accomplished by setting the `scope` option to `"controller"` within the controller's `queryParams` config hash:

```
App.ArticlesRoute = Ember.Route.extend({
  queryParams: {
    showMagnifyingGlass: {
      scope: "controller"
    }
  }
});
```

The following demonstrates how you can override both the scope and the query param URL key of a single controller query param property:

```
App.ArticlesController = Ember.Controller.extend({
  queryParams: [ "page", "filter",
    {
      showMagnifyingGlass: {
        scope: "controller",
        as: "glass",
      }
    }
  ]
});
```

Examples

- Search queries[161]
- Sort: client-side, no refiring of model hook[162]
- Sort: server-side, refire model hook[163]
- Pagination + Sorting[164]
- Boolean values. False value removes QP from URL[165]
- Global query params on app route[166]
- Opt-in to full transition via refresh()[167]
- update query params by changing controller QP property[168]
- update query params with replaceState by changing controller QP property[169]
- w/ {{partial}} helper for easy tabbing[170]
- link-to with no route name, only QP change[171]
- Complex: serializing textarea content into URL (and subexpressions))[172]
- Arrays[173]

Asynchronous Routing

This section covers some more advanced features of the router and its capability for handling complex async logic within your app.

A Word on Promises…

Ember's approach to handling asynchronous logic in the router makes heavy use of the concept of Promises. In short, promises are objects that represent an eventual value. A promise can either *fulfill* (successfully resolve the value) or *reject* (fail to resolve the value). The way to retrieve this eventual value, or handle the cases when the promise rejects, is via the promise's `then` method, which accepts two optional callbacks, one for fulfillment and one for rejection. If the promise fulfills, the fulfillment handler gets called with the fulfilled value as its sole argument, and if the promise rejects, the rejection handler gets called with a reason for the rejection as its sole argument. For example:

[161] http://emberjs.jsbin.com/ucanam/6046
[162] http://emberjs.jsbin.com/ucanam/6048
[163] http://emberjs.jsbin.com/ucanam/6049
[164] http://emberjs.jsbin.com/ucanam/6050
[165] http://emberjs.jsbin.com/ucanam/6051
[166] http://emberjs.jsbin.com/ucanam/6052
[167] http://emberjs.jsbin.com/ucanam/6054
[168] http://emberjs.jsbin.com/ucanam/6055
[169] http://emberjs.jsbin.com/ucanam/6056/edit
[170] http://emberjs.jsbin.com/ucanam/6058
[171] http://emberjs.jsbin.com/ucanam/6060
[172] http://emberjs.jsbin.com/ucanam/6062/edit
[173] http://emberjs.jsbin.com/ucanam/6064

```
var promise = fetchTheAnswer();

promise.then(fulfill, reject);

function fulfill(answer) {
  console.log("The answer is " + answer);
}

function reject(reason) {
  console.log("Couldn't get the answer! Reason: " + reason);
}
```

Much of the power of promises comes from the fact that they can be chained together to perform sequential asynchronous operations:

```
// Note: jQuery AJAX methods return promises
var usernamesPromise = Ember.$.getJSON('/usernames.json');

usernamesPromise.then(fetchPhotosOfUsers)
                .then(applyInstagramFilters)
                .then(uploadTrendyPhotoAlbum)
                .then(displaySuccessMessage, handleErrors);
```

In the above example, if any of the methods `fetchPhotosOfUsers`, `applyInstagramFilters`, or `uploadTrendyPhotoAlbum` returns a promise that rejects, `handleErrors` will be called with the reason for the failure. In this manner, promises approximate an asynchronous form of try-catch statements that prevent the rightward flow of nested callback after nested callback and facilitate a saner approach to managing complex asynchronous logic in your applications.

This guide doesn't intend to fully delve into all the different ways promises can be used, but if you'd like a more thorough introduction, take a look at the readme for RSVP[174], the promise library that Ember uses.

The Router Pauses for Promises

When transitioning between routes, the Ember router collects all of the models (via the `model` hook) that will be passed to the route's controllers at the end of the transition. If the `model` hook (or the related `beforeModel` or `afterModel` hooks) return normal (non-promise) objects or arrays, the transition will complete immediately. But if the `model` hook (or the related `beforeModel` or `afterModel` hooks) returns a promise (or if a promise was provided as an argument to `transitionTo`), the transition will pause until that promise fulfills or rejects.

[174] https://github.com/tildeio/rsvp.js

<aside> **Note**: The router considers any object with a `then` method defined on it to be a promise. </aside>

If the promise fulfills, the transition will pick up where it left off and begin resolving the next (child) route's model, pausing if it too is a promise, and so on, until all destination route models have been resolved. The values passed to the `setupController` hook for each route will be the fulfilled values from the promises.

A basic example:

```
App.TardyRoute = Ember.Route.extend({
  model: function() {
    return new Ember.RSVP.Promise(function(resolve) {
      Ember.run.later(function() {
        resolve({ msg: "Hold Your Horses" });
      }, 3000);
    });
  },

  setupController: function(controller, model) {
    console.log(model.msg); // "Hold Your Horses"
  }
});
```

When transitioning into `TardyRoute`, the `model` hook will be called and return a promise that won't resolve until 3 seconds later, during which time the router will be paused in mid-transition. When the promise eventually fulfills, the router will continue transitioning and eventually call `TardyRoute`'s `setupController` hook with the resolved object.

This pause-on-promise behavior is extremely valuable for when you need to guarantee that a route's data has fully loaded before displaying a new template.

When Promises Reject…

We've covered the case when a model promise fulfills, but what if it rejects?

By default, if a model promise rejects during a transition, the transition is aborted, no new destination route templates are rendered, and an error is logged to the console.

You can configure this error-handling logic via the `error` handler on the route's `actions` hash. When a promise rejects, an `error` event will be fired on that route and bubble up to `ApplicationRoute`'s default error handler unless it is handled by a custom error handler along the way, e.g.:

```
App.GoodForNothingRoute = Ember.Route.extend({
  model: function() {
    return Ember.RSVP.reject("FAIL");
  },

  actions: {
    error: function(reason) {
      alert(reason); // "FAIL"

      // Can transition to another route here, e.g.
      // this.transitionTo('index');

      // Uncomment the line below to bubble this error event:
      // return true;
    }
  }
});
```

In the above example, the error event would stop right at `GoodForNothingRoute`'s error handler and not continue to bubble. To make the event continue bubbling up to `ApplicationRoute`, you can return true from the error handler.

Recovering from Rejection

Rejected model promises halt transitions, but because promises are chainable, you can catch promise rejects within the `model` hook itself and convert them into fulfills that won't halt the transition.

```
App.FunkyRoute = Ember.Route.extend({
  model: function() {
    return iHopeThisWorks().then(null, function() {
      // Promise rejected, fulfill with some default value to
      // use as the route's model and continue on with the transition
      return { msg: "Recovered from rejected promise" };
    });
  }
});
```

beforeModel and afterModel

The `model` hook covers many use cases for pause-on-promise transitions, but sometimes you'll need the help of the related hooks `beforeModel` and `afterModel`. The most common reason for this is that if you're transitioning into a route with a dynamic URL segment via `{{link-to}}` or `transitionTo` (as opposed to a transition caused by a URL change), the model for the route you're transitioning into will have already been specified (e.g. `{{#link-to 'article' article}}` or `this.transitionTo('article', article)`), in which case the `model` hook won't get called. In these cases, you'll need to make use of either the `beforeModel` or `afterModel` hook to house any logic while the router is still gathering all of the route's models to perform a transition.

`beforeModel`

Easily the more useful of the two, the `beforeModel` hook is called before the router attempts to resolve the model for the given route. In other words, it is called before the `model` hook gets called, or, if `model` doesn't get called, it is called before the router attempts to resolve any model promises passed in for that route.

Like `model`, returning a promise from `beforeModel` will pause the transition until it resolves, or will fire an `error` if it rejects.

The following is a far-from-exhaustive list of use cases in which `beforeModel` is very handy:

- Deciding whether to redirect to another route before performing a potentially wasteful server query in `model`
- Ensuring that the user has an authentication token before proceeding onward to `model`
- Loading application code required by this route

```
App.SecretArticlesRoute  = Ember.Route.extend({
  beforeModel: function() {
    if (!this.controllerFor('auth').get('isLoggedIn')) {
      this.transitionTo('login');
    }
  }
});
```

See the API Docs for `beforeModel`[175]

`afterModel`

The `afterModel` hook is called after a route's model (which might be a promise) is resolved, and follows the same pause-on-promise semantics as `model` and `beforeModel`. It is passed the already-resolved model and can therefore perform any additional logic that depends on the fully resolved value of a model.

[175] http://emberjs.com/api/classes/Ember.Route.html#method_beforeModel

```
App.ArticlesRoute = Ember.Route.extend({
  model: function() {
    // App.Article.find() returns a promise-like object
    // (it has a `then` method that can be used like a promise)
    return App.Article.find();
  },
  afterModel: function(articles) {
    if (articles.get('length') === 1) {
      this.transitionTo('article.show', articles.get('firstObject'));
    }
  }
});
```

You might be wondering why we can't just put the `afterModel` logic into the fulfill handler of the promise returned from `model`; the reason, as mentioned above, is that transitions initiated via `{{link-to}}` or `transitionTo` likely already provided the model for this route, so `model` wouldn't be called in these cases.

See the API Docs for `afterModel`[176]

More Resources

- Embercasts: Client-side Authentication Part 2[177]
- RC6 Blog Post describing these new features[178]

Loading / Error Substates

In addition to the techniques described in the Asynchronous Routing Guide[179], the Ember Router provides powerful yet overridable conventions for customizing asynchronous transitions between routes by making use of `error` and `loading` substates.

`loading` substates

The Ember Router allows you to return promises from the various `beforeModel`/`model`/`afterModel` hooks in the course of a transition (described here[180]). These promises pause the transition until they fulfill, at which point the transition will resume.

Consider the following:

[176] http://emberjs.com/api/classes/Ember.Route.html#method_afterModel

[177] http://www.embercasts.com/episodes/client-side-authentication-part-2

[178] http://emberjs.com/blog/2013/06/23/ember-1-0-rc6.html

[179] http://emberjs.com/guides/routing/asynchronous-routing/

[180] http://emberjs.com/guides/routing/asynchronous-routing/

```
App.Router.map(function() {
  this.resource('foo', function() { // -> FooRoute
    this.route('slowModel');        // -> FooSlowModelRoute
  });
});

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

If you navigate to `foo/slow_model`, and in `FooSlowModelRoute#model`, you return an AJAX query promise that takes a long time to complete. During this time, your UI isn't really giving you any feedback as to what's happening; if you're entering this route after a full page refresh, your UI will be entirely blank, as you have not actually finished fully entering any route and haven't yet displayed any templates; if you're navigating to `foo/slow_model` from another route, you'll continue to see the templates from the previous route until the model finish loading, and then, boom, suddenly all the templates for `foo/slow_model` load.

So, how can we provide some visual feedback during the transition?

Ember provides a default implementation of the `loading` process that implements the following loading substate behavior.

```
App.Router.map(function() {
  this.resource('foo', function() {       // -> FooRoute
    this.resource('foo.bar', function() { // -> FooBarRoute
      this.route('baz');                  // -> FooBarBazRoute
    });
  });
});
```

If a route with the path `foo.bar.baz` returns a promise that doesn't immediately resolve, Ember will try to find a `loading` route in the hierarchy above `foo.bar.baz` that it can transition into, starting with `foo.bar.baz`'s sibling:

1. `foo.bar.loading`
2. `foo.loading`
3. `loading`

Ember will find a loading route at the above location if either a) a Route subclass has been defined for such a route, e.g.

1. `App.FooBarLoadingRoute`
2. `App.FooLoadingRoute`
3. `App.LoadingRoute`

or b) a properly-named loading template has been found, e.g.

1. `foo/bar/loading`
2. `foo/loading`
3. `loading`

During a slow asynchronous transition, Ember will transition into the first loading sub-state/route that it finds, if one exists. The intermediate transition into the loading substate happens immediately (synchronously), the URL won't be updated, and, unlike other transitions that happen while another asynchronous transition is active, the currently active async transition won't be aborted.

After transitioning into a loading substate, the corresponding template for that substate, if present, will be rendered into the main outlet of the parent route, e.g. `foo.bar.loading`'s template would render into `foo.bar`'s outlet. (This isn't particular to loading routes; all routes behave this way by default.)

Once the main async transition into `foo.bar.baz` completes, the loading substate will be exited, its template torn down, `foo.bar.baz` will be entered, and its templates rendered.

Eager vs. Lazy Async Transitions

Loading substates are optional, but if you provide one, you are essentially telling Ember that you want this async transition to be "eager"; in the absence of destination route loading substates, the router will "lazily" remain on the pre-transition route while all of the destination routes' promises resolve, and only fully transition to the destination route (and renders its templates, etc.) once the transition is complete. But once you provide a destination route loading substate, you are opting into an "eager" transition, which is to say that, unlike the "lazy" default, you will eagerly exit the source routes (and tear down their templates, etc) in order to transition into this substate. URLs always update immediately unless the transition was aborted or redirected within the same run loop.

This has implications on error handling, i.e. when a transition into another route fails, a lazy transition will (by default) just remain on the previous route, whereas an eager transition will have already left the pre-transition route to enter a loading substate.

The `loading` event

If you return a promise from the various `beforeModel`/`model`/`afterModel` hooks, and it doesn't immediately resolve, a `loading` event will be fired on that route and bubble upward to `ApplicationRoute`.

If the `loading` handler is not defined at the specific route, the event will continue to bubble above a transition's pivot route, providing the `ApplicationRoute` the opportunity to manage it.

```
App.FooSlowModelRoute = Ember.Route.extend({
  model: function() {
    return somePromiseThatTakesAWhileToResolve();
  },
  actions: {
    loading: function(transition, originRoute) {
      //displayLoadingSpinner();

      // Return true to bubble this event to `FooRoute`
      // or `ApplicationRoute`.
      return true;
    }
  }
});
```

The `loading` handler provides the ability to decide what to do during the loading process. If the last loading handler is not defined or returns `true`, Ember will perform the loading substate behavior.

```
App.ApplicationRoute = Ember.Route.extend({
  actions: {
    loading: function(transition, originRoute) {
      displayLoadingSpinner();

      // substate implementation when returning `true`
      return true;
    }
  }
});
```

`error` substates

Ember provides an analogous approach to `loading` substates in the case of errors encountered during a transition.

Similar to how the default `loading` event handlers are implemented, the default `error` handlers will look for an appropriate error substate to enter, if one can be found.

```
App.Router.map(function() {
  this.resource('articles', function() { // -> ArticlesRoute
    this.route('overview');              // -> ArticlesOverviewRoute
  });
});
```

For instance, an error thrown or rejecting promise returned from `ArticlesOverviewRoute#model` (or `beforeModel` or `afterModel`) will look for:

1. Either `ArticlesErrorRoute` or `articles/error` template
2. Either `ErrorRoute` or `error` template

If one of the above is found, the router will immediately transition into that substate (without updating the URL). The "reason" for the error (i.e. the exception thrown or the promise reject value) will be passed to that error state as its `model`.

If no viable error substates can be found, an error message will be logged.

`error` substates with dynamic segments

Routes with dynamic segments are often mapped to a mental model of "two separate levels." Take for example:

```
App.Router.map(function() {
  this.resource('foo', {path: '/foo/:id'}, function() {
    this.route('baz');
  });
});

App.FooRoute = Ember.Route.extend({
  model: function(params) {
    return new Ember.RSVP.Promise(function(resolve, reject) {
       reject("Error");
    });
  }
});
```

In the URL hierarchy you would visit `/foo/12` which would result in rendering the `foo` template into the `application` template's `outlet`. In the event of an error while attempting to load the `foo` route you would also render the top-level `error` template into the `application` template's `outlet`. This is intentionally parallel behavior as the `foo` route is never successfully entered. In order to create a `foo` scope for errors and render `foo/error` into `foo`'s `outlet` you would need to split the dynamic segment:

```
App.Router.map(function() {
  this.resource('foo', {path: '/foo'}, function() {
    this.resource('elem', {path: ':id'}, function() {
      this.route('baz');
    });
  });
});
```

Example JSBin[181]

The `error` event

If `ArticlesOverviewRoute#model` returns a promise that rejects (because, for instance, the server returned an error, or the user isn't logged in, etc.), an `error` event will fire on `ArticlesOverviewRoute` and bubble upward. This `error` event can be handled and used to display an error message, redirect to a login page, etc.

```
App.ArticlesOverviewRoute = Ember.Route.extend({
  model: function(params) {
    return new Ember.RSVP.Promise(function(resolve, reject) {
       reject("Error");
    });
  },
  actions: {
    error: function(error, transition) {

      if (error && error.status === 400) {
        // error substate and parent routes do not handle this error
        return this.transitionTo('modelNotFound');
      }

      // Return true to bubble this event to any parent route.
      return true;
    }
  }
});
```

In analogy with the `loading` event, you could manage the `error` event at the Application level to perform any app logic and based on the result of the last `error` handler, Ember will decide if substate behavior must be performed or not.

[181] http://emberjs.jsbin.com/ucanam/4279

```
App.ApplicationRoute = Ember.Route.extend({
  actions: {
    error: function(error, transition) {

      // Manage your errors
      Ember.onerror(error);

      // substate implementation when returning `true`
      return true;

    }
  }
});
```

Legacy `LoadingRoute`

Previous versions of Ember (somewhat inadvertently) allowed you to define a global `LoadingRoute` which would be activated whenever a slow promise was encountered during a transition and exited upon completion of the transition. Because the `loading` template rendered as a top-level view and not within an outlet, it could be used for little more than displaying a loading spinner during slow transitions. Loading events/substates give you far more control, but if you'd like to emulate something similar to the legacy `LoadingRoute` behavior, you could do as follows:

```
App.LoadingView = Ember.View.extend({
  templateName: 'global-loading',
  elementId: 'global-loading'
});

App.ApplicationRoute = Ember.Route.extend({
  actions: {
    loading: function() {
      var view = this.container.lookup('view:loading').append();
      this.router.one('didTransition', view, 'destroy');
    }
  }
});
```

Example JSBin[182]

This will, like the legacy `LoadingRoute`, append a top-level view when the router goes into a loading state, and tear down the view once the transition finishes.

[182] http://emberjs.jsbin.com/ucanam/3307

Preventing And Retrying Transitions

During a route transition, the Ember Router passes a transition object to the various hooks on the routes involved in the transition. Any hook that has access to this transition object has the ability to immediately abort the transition by calling `transition.abort()`, and if the transition object is stored, it can be re-attempted at a later time by calling `transition.retry()`.

Preventing Transitions via `willTransition`

When a transition is attempted, whether via `{{link-to}}`, `transitionTo`, or a URL change, a `willTransition` action is fired on the currently active routes. This gives each active route, starting with the leaf-most route, the opportunity to decide whether or not the transition should occur.

Imagine your app is in a route that's displaying a complex form for the user to fill out and the user accidentally navigates backwards. Unless the transition is prevented, the user might lose all of the progress they made on the form, which can make for a pretty frustrating user experience.

Here's one way this situation could be handled:

```
App.FormRoute = Ember.Route.extend({
  actions: {
    willTransition: function(transition) {
      if (this.controller.get('userHasEnteredData') &&
          !confirm("Are you sure you want to abandon progress?")) {
        transition.abort();
      } else {
        // Bubble the `willTransition` action so that
        // parent routes can decide whether or not to abort.
        return true;
      }
    }
  }
});
```

When the user clicks on a `{{link-to}}` helper, or when the app initiates a transition by using `transitionTo`, the transition will be aborted and the URL will remain unchanged. However, if the browser back button is used to navigate away from `FormRoute`, or if the user manually changes the URL, the new URL will be navigated to before the `willTransition` action is called. This will result in the browser displaying the new URL, even if `willTransition` calls `transition.abort()`.

Aborting Transitions Within `model`, `beforeModel`, `afterModel`

The `model`, `beforeModel`, and `afterModel` hooks described in Asynchronous Routing[183] each get called with a transition object. This makes it possible for destination routes to abort attempted transitions.

```
App.DiscoRoute = Ember.Route.extend({
  beforeModel: function(transition) {
    if (new Date() < new Date("January 1, 1980")) {
      alert("Sorry, you need a time machine to enter this route.");
      transition.abort();
    }
  }
});
```

Storing and Retrying a Transition

Aborted transitions can be retried at a later time. A common use case for this is having an authenticated route redirect the user to a login page, and then redirecting them back to the authenticated route once they've logged in.

```
App.SomeAuthenticatedRoute = Ember.Route.extend({
  beforeModel: function(transition) {
    if (!this.controllerFor('auth').get('userIsLoggedIn')) {
      var loginController = this.controllerFor('login');
      loginController.set('previousTransition', transition);
      this.transitionTo('login');
    }
  }
});

App.LoginController = Ember.Controller.extend({
  actions: {
    login: function() {
      // Log the user in, then reattempt previous transition if it exists.
      var previousTransition = this.get('previousTransition');
      if (previousTransition) {
        this.set('previousTransition', null);
        previousTransition.retry();
      } else {
```

[183] http://emberjs.com/guides/routing/asynchronous-routing

```
        // Default back to homepage
        this.transitionToRoute('index');
      }
    }
  }
});
```

Components

Introduction

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.

Currently, you are limited to the tags that are created for you by the W3C. Wouldn't it be great if you could define your own, application-specific HTML tags, then implement their behavior using JavaScript?

That's exactly what components let you do. In fact, it's such a good idea that the W3C is currently working on the Custom Elements[184] spec.

Ember's implementation of components hews as closely to the Web Components specification[185] 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.

This is so important to us that we are working closely with the standards bodies to ensure our implementation of components matches the roadmap of the web platform.

To highlight the power of components, here is a short example of turning a blog post into a reusable `blog-post` custom element that you could use again and again in your application. Keep reading this section for more details on building components.

JS Bin[186]

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.

<aside> **Note**: Components must have a dash in their name. So `blog-post` is an acceptable name, but `post` is not. This prevents clashes with current or future HTML element names, and ensures Ember picks up the components automatically. </aside>

[184] https://dvcs.w3.org/hg/webcomponents/raw-file/tip/spec/custom/index.html

[185] http://www.w3.org/TR/components-intro/

[186] http://jsbin.com/ifuxey

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

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

```
<h1>My Blog</h1>
{{#each}}
  {{blog-post}}
{{/each}}
```

JS Bin[187]

Each component, under the hood, is backed by an element. By default Ember will use a `<div>` element to contain your component's template. To learn how to change the element Ember uses for your component, see Customizing a Component's Element[188].

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.

If you need to customize the behavior of the component you'll need to define a subclass of `Ember.Component`. For example, you would need a custom subclass if you wanted to change a component's element, respond to actions from the component's template, or manually make changes to the component's element using JavaScript.

Ember knows which subclass powers a component based on its 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-player-controls`, the class name would be `App.AudioPlayerControlsComponent`.

In other words, Ember will look for a class with the camelized name of the component, followed by `Component`.

<table> <thead> <tr> <th>Component Name</th> <th>Component Class</th> </tr> </thead> <tr> <td><code>blog-post</code></td> <td><code>App.BlogPostComponent</code></td> </tr> <tr> <td><code>audio-player-controls</code></td> <td><code>App.AudioPlayerControlsComponent</code></td> </tr> </table>

[187] http://jsbin.com/ifuxey

[188] http://emberjs.com/guides/components/customizing-a-components-element

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:

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

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

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

Running this code, you will see that the first `<h1>` (from the outer template) displays the `title` property, but the second `<h1>` (from inside the component) is empty.

JS Bin[189]

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

```
{{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`.

JS Bin[190]

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

[189] http://jsbin.com/ufedet

[190] http://jsbin.com/ufedet

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

JS Bin[191]

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`.

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. In the following example, type some text in the text field either in the outer template or inside the component and note how they stay in sync.

JS Bin[192]

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.

```
{{#each}}
  {{blog-post title=title}}
{{/each}}
```

JS Bin[193]

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:

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

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

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

[191] http://jsbin.com/ufedet

[192] http://jsbin.com/ufedet

[193] http://jsbin.com/ifuxey

JS Bin[194]

(See Passing Properties to a Component[195] for more.)

In this case, the content we wanted to display came from the model. But what if we want the developer using our component to be able to provide custom HTML content?

In addition to the simple form you've learned so far, components also support being used in **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. (See the Handlebars documentation on block expressions[196] for more.)

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:

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

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

JS Bin[197]

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.

This JSBin illustrates the concept:

JS Bin[198]

[194] http://jsbin.com/obogub

[195] http://emberjs.com/guides/components/passing-properties-to-a-component/

[196] http://handlebarsjs.com/#block-expressions

[197] http://jsbin.com/osulic

[198] http://jsbin.com/iqocuf

Customizing A Component's Element

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:

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

You can customize what type of element Ember generates for your component, including its attributes and class names, by creating a subclass of `Ember.Component` in your JavaScript.

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.

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

```
{{! templates/components/navigation-bar }}
<ul>
  <li>{{#link-to 'home'}}Home{{/link-to}}</li>
  <li>{{#link-to 'about'}}About{{/link-to}}</li>
</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:

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

If you want class names to be determined by properties of the component, 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:

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

This component would render the following:

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

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

This would render this HTML:

```
<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`:

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

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:

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

This would render this HTML:

```
<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 property's value is a string, that value will be added as a class name without modification:

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

This would render this HTML:

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

Customizing Attributes

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

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

You can also bind these attributes to differently named properties:

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

Example

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

JS Bin[199]

<aside> **Note**: The binding functionality in this very simple example could also be implemented without the use of `Ember.Component` but by simply binding element attributes[200] or binding element class names[201]. </aside>

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.

You can make elements in your component interactive by using the `{{action}}` helper. This is the same `{{action}}` helper you use in application templates[202], 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:

[199] http://jsbin.com/utonef

[200] http://emberjs.com/guides/templates/binding-element-attributes

[201] http://emberjs.com/guides/templates/binding-element-class-names

[202] http://emberjs.com/guides/templates/actions

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

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

JS Bin[203]

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[204] of the Templates chapter.

Sending Actions from Components to Your Application

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 application 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 the controller does not implement a handler for that action, it will bubble to the template's route, and then up the route hierarchy. For more information about this bubbling behavior, see Action Bubbling[205].

Components are designed to be reusable across different parts of your application. In order to achieve this reusability, it's important that the actions that your components send can be specified when the component is used in a template.

In other words, if you were writing a button component, you would not want to send a `click` action, because it is ambiguous and likely to conflict with other components on the page. Instead, you would want to allow the person using the component to specify which action to send when the button was clicked.

[203] http://jsbin.com/EWEQeKO

[204] http://emberjs.com/guides/templates/actions

[205] http://emberjs.com/guides/templates/actions/#toc_action-bubbling

Luckily, components have a `sendAction()` method that allows them to send actions specified when the component is 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:

```
{{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:

```
App.MyButtonComponent = Ember.Component.extend({
  click: function() {
    this.sendAction();
  }
});
```

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:

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

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

```
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);
    }
  }
});
```

```
{{! index.handlebars }}

{{#each todo in todos}}
  <p>{{todo.title}} <button {{action "deleteTodo" todo}}>Delete</button></p>
{{/each}}
```

We want to update this app so that, before actually deleting a todo, the user must confirm that this is what they intended. We'll implement a component that first double-checks with the user before completing the action.

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

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

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

```
{{! templates/components/confirm-button.handlebars }}

{{#if isShowingConfirmation}}
  <button {{action "confirm"}}>Click again to confirm</button>
{{else}}
  <button {{action "showConfirmation"}}>{{title}}</button>
{{/if}}
```

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

```
{{! index.handlebars }}

    {{#each todo in todos}}
      <p>{{todo.title}} {{confirm-button title="Delete" action="deleteTodo" para\
m=todo}}</p>
    {{/each}}
```

Note that we've specified the action to send by setting the component's `action` attribute, and we've specified which argument should be sent as a parameter by setting the component's `param` attribute.

[JS Bin]http://jsbin.com/atIgUSi)

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:

```
{{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')`.

JS Bin[206]

[206] http://jsbin.com/OpebEFO

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:

```
App.MyButtonComponent = Ember.Component.extend({
  click: function() {
    this.sendAction();
  }
});
```

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

```
{{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.

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.

Controllers

Introduction

Controllers

In Ember.js, controllers allow you to decorate your models with display logic. In general, your models will have properties that are saved to the server, while controllers will have properties that your app does not need to save to the server.

For example, if you were building a blog, you would have a `BlogPost` model that you would present in a `blog_post` template.

Your `BlogPost` model would have properties like:

- `title`
- `intro`
- `body`
- `author`

Your template would bind to these properties in the `blog_post` template:

```
<h1>{{title}}</h1>
<h2>by {{author}}</h2>

<div class='intro'>
  {{intro}}
</div>
<hr>
<div class='body'>
  {{body}}
</div>
```

In this simple example, we don't have any display-specific properties or actions just yet. For now, our controller just acts as a pass-through (or "proxy") for the model properties. (Remember that a controller gets the model it represents from its route handler.)

Let's say we wanted to add a feature that would allow the user to toggle the display of the body section. To implement this, we would first modify our template to show the body only if the value of a new `isExpanded` property is true.

```
<h1>{{title}}</h1>
<h2>by {{author}}</h2>

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

{{#if isExpanded}}
  <button {{action 'toggleProperty' 'isExpanded'}}>Hide Body</button>
  <div class='body'>
    {{body}}
  </div>
{{else}}
  <button {{action 'toggleProperty' 'isExpanded'}}>Show Body</button>
{{/if}}
```

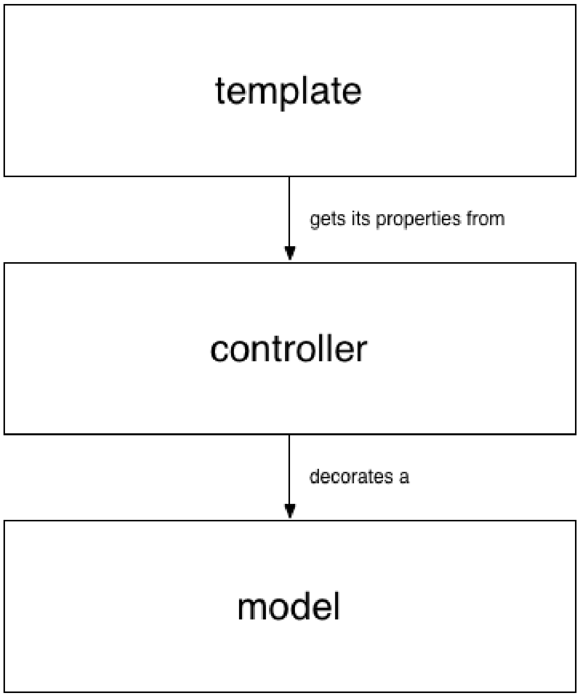
You might think you should put this property on the model, but whether the body is expanded or not is strictly a display concern.

Putting this property on the controller cleanly separates logic related to your data model from logic related to what you display on the screen. This makes it easy to unit-test your model without having to worry about logic related to your display creeping into your test setup.

A Note on Coupling

In Ember.js, templates get their properties from controllers, which decorate a model.

This means that templates *know about* controllers and controllers *know about* models, but the reverse is not true. A model knows nothing about which (if any) controllers are decorating it, and controller does not know which views are presenting its properties.



Objects

This also means that as far as a template is concerned, all of its properties come from its controller, and it doesn't need to know about the model directly.

In practice, Ember.js will create a template's controller once for the entire application, but the controller's model may change throughout the lifetime of the application without requiring that the view knows anything about those mechanics.

<aside> For example, if the user navigates from `/posts/1` to `/posts/2`, the `PostController` will change its model from `Post.find(1)` to `Post.find(2)`. The template will update its representations of any properties on the model, as well as any computed properties on the controller that depend on the model. </aside>

This makes it easy to test a template in isolation by rendering it with a controller object that contains the properties the template expects. From the template's perspective, a **controller** is simply an object that provides its data.

Representing Models

Templates are always connected to controllers, not models. This makes it easy to separate display-specific properties from model specific properties, and to swap out the controller's model as the user navigates around the page.

For convenience, Ember.js provides controllers that *proxy* properties from their models so that you can say `{{name}}` in your template rather than `{{model.name}}`. An `Ember.ArrayController` proxies properties from an Array, and an `Ember.ObjectController` proxies properties from an object.

If your controller is an `ArrayController`, you can iterate directly over the controller using `{{#each controller}}`. This keeps the template from having to know about how the controller is implemented and makes isolation testing and refactoring easier.

Storing Application Properties

Not all properties in your application need to be saved to the server. Any time you need to store information only for the lifetime of this application run, you should store it on a controller.

For example, imagine your application has a search field that is always present. You could store a `search` property on your `ApplicationController`, and bind the search field in the ‘ application‘ template to that property, like this:

```
<!-- application.handlebars -->
<header>
  {{input type="text" value=search action="query"}}
</header>

{{outlet}}
```

```
App.ApplicationController = Ember.Controller.extend({
  // the initial value of the `search` property
  search: '',

  actions: {
    query: function() {
      // the current value of the text field
      var query = this.get('search');
      this.transitionToRoute('search', { query: query });
    }
  }
});
```

The `application` template stores its properties and sends its actions to the `ApplicationController`. In this case, when the user hits enter, the application will transition to the `search` route, passing the query as a parameter.

Representing A Single Model With ObjectController

Use `Ember.ObjectController` to represent a single model. To tell an `ObjectController` which model to represent, set its `model` property in your route's `setupController` method.

When a template asks an `ObjectController` for the value of a property, the controller looks for a property with the same name on itself first before checking the model.

For example, imagine you are writing a music player. You have defined your `SongController` to represent the currently playing song.

```
App.SongController = Ember.ObjectController.extend({
  soundVolume: 1
});
```

In the Song route, you set the `model` of the controller to the currently playing song:

```
App.SongRoute = Ember.Route.extend({
  setupController: function(controller, song) {
    controller.set('model', song);
  }
});
```

In your template, you want to display the name of the currently playing song, as well as the volume at which it is playing.

```
<p>
  <strong>Song</strong>: {{name}} by {{artist}}
</p>
<p>
  <strong>Current Volume</strong>: {{soundVolume}}
</p>
```

Because `name` and `artist` are persisted information, and thus stored on the model, the controller looks them up there and provides them to the template.

`soundVolume`, however, is specific to the current user's session, and thus stored on the controller. The controller can return its own value without consulting the model.

The advantage of this architecture is that it is easy to get started by accessing the properties of the model via the object controller. If, however, you need to transform a model property for a template, there is a well-defined place to do so without adding view-specific concerns to the model.

For example, imagine we want to display the duration of the song:

```
<p>
  <strong>Song</strong>: {{name}} by {{artist}}
</p>
<p>
  <strong>Duration</strong>: {{duration}}
</p>
```

This is saved on the server as an integer representing the number of seconds, so our first attempt looks like this:

```
<p>
  <strong>Song</strong>: 4 Minute Warning by Radiohead
</p>
<p>
  <strong>Duration</strong>: 257
</p>
```

Since our users are humans and not robots, however, we'd like to display the duration as a formatted string.

This is very easy to do by defining a computed property on the controller which transforms the model's value into a human-readable format for the template:

```
App.SongController = Ember.ObjectController.extend({
  duration: function() {
    var duration = this.get('model.duration'),
         minutes = Math.floor(duration / 60),
         seconds = duration % 60;

    return [minutes, seconds].join(':');
  }.property('model.duration')
});
```

Now, the output of our template is a lot friendlier:

```
<p>
  <strong>Song</strong>: 4 Minute Warning by Radiohead
</p>
<p>
  <strong>Duration</strong>: 4:17
</p>
```

Representing Multiple Models With ArrayController

You can use Ember.ArrayController[207] to represent an array of models. To tell an `ArrayController` which models to represent, set its `model` property in your route's `setupController` method.

You can treat an `ArrayController` just like its underlying array. For example, imagine we want to display the current playlist. In our route, we setup our `SongsController` to represent the songs in the playlist:

```
App.SongsRoute = Ember.Route.extend({
  setupController: function(controller, playlist) {
    controller.set('model', playlist.get('songs'));
  }
});
```

In the `songs` template, we can use the `{{#each}}` helper to display each song:

```
<h1>Playlist</h1>

<ul>
  {{#each}}
    <li>{{name}} by {{artist}}</li>
  {{/each}}
</ul>
```

You can use the `ArrayController` to collect aggregate information about the models it represents. For example, imagine we want to display the number of songs that are over 30 seconds long. We can add a new computed property called `longSongCount` to the controller:

[207] http://emberjs.com/api/classes/Ember.ArrayController.html

```
App.SongsController = Ember.ArrayController.extend({
  longSongCount: function() {
    var longSongs = this.filter(function(song) {
      return song.get('duration') > 30;
    });
    return longSongs.get('length');
  }.property('@each.duration')
});
```

Now we can use this property in our template:

```
<ul>
  {{#each}}
    <li>{{name}} by {{artist}}</li>
  {{/each}}
</ul>

{{longSongCount}} songs over 30 seconds.
```

Sorting

The `Ember.ArrayController` uses the Ember.SortableMixin[208] to allow sorting of content. There are two properties that can be set in order to set up sorting:

```
App.SongsController = Ember.ArrayController.extend({
  sortProperties: ['name', 'artist'],
  sortAscending: true // false for descending
});
```

Item Controller

It is often useful to specify a controller to decorate individual items in the `ArrayController` while iterating over them. This can be done by creating an `ObjectController`:

[208] http://emberjs.com/api/classes/Ember.SortableMixin.html

```
App.SongController = Ember.ObjectController.extend({
  fullName: function() {

    return this.get('name') + ' by ' + this.get('artist');

  }.property('name', 'artist')
});
```

Then, the `ArrayController` `itemController` property must be set to the decorating controller.

```
App.SongsController = Ember.ArrayController.extend({
  itemController: 'song'
});
```

```
{{#each controller}}
  <li>{{fullName}}</li>
{{/each}}
```

or you could setup the `itemController` directly in the template:

```
App.SongsController = Ember.ArrayController.extend({
});
```

```
{{#each controller itemController="song"}}
  <li>{{fullName}}</li>
{{/each}}
```

Managing Dependencies Between Controllers

Sometimes, especially when nesting resources, we find ourselves needing to have some kind of connection between two controllers. Let's take this router as an example:

```
App.Router.map(function() {
  this.resource("post", { path: "/posts/:post_id" }, function() {
    this.resource("comments", { path: "/comments" });
  });
});
```

If we visit a `/posts/1/comments` URL, our `Post` model will get loaded into a `PostController`'s model, which means it is not directly accessible in the `CommentsController`. We might however want to display some information about it in the `comments` template.

To be able to do this we define our `CommentsController` to `need` the `PostController` which has our desired `Post` model.

```
App.CommentsController = Ember.ArrayController.extend({
  needs: "post"
});
```

This tells Ember that our `CommentsController` should be able to access its parent `PostController`, which can be done via `controllers.post` (either in the template or in the controller itself).

```
<h1>Comments for {{controllers.post.title}}</h1>

<ul>
  {{#each comments}}
    <li>{{text}}</li>
  {{/each}}
</ul>
```

We can also create an aliased property to give ourselves a shorter way to access the `PostController` (since it is an `ObjectController`, we don't need or want the `Post` instance directly).

```
App.CommentsController = Ember.ArrayController.extend({
  needs: "post",
  post: Ember.computed.alias("controllers.post")
});
```

If you want to connect multiple controllers together, you can specify an array of controller names:

```
App.AnotherController = Ember.Controller.extend({
  needs: ['post', 'comments']
});
```

For more information about dependecy injection and `needs` in Ember.js, see the dependency injection guide[209]. For more information about aliases, see the API docs for aliased properties[210].

[209] http://emberjs.com/guides/understanding-ember/dependency-injection-and-service-lookup

[210] http://emberjs.com/api/#method_computed_alias

Models

Introduction

Models

In Ember, every route has an associated model. This model is set by implementing a route's `model` hook, by passing the model as an argument to `{{link-to}}`, or by calling a route's `transitionTo()` method.

See Specifying a Route's Model[211] for more information on setting a route's model.

For simple applications, you can get by using jQuery to load JSON data from a server, then use those JSON objects as models.

However, using a model library that manages finding models, making changes, and saving them back to the server can dramatically simplify your code while improving the robustness and performance of your application.

Many Ember apps use Ember Data[212] to handle this. Ember Data is a library that integrates tightly with Ember.js to make it easy to retrieve records from a server, cache them for performance, save updates back to the server, and create new records on the client.

Without any configuration, Ember Data can load and save records and their relationships served via a RESTful JSON API, provided it follows certain conventions.

If you need to integrate your Ember.js app with existing JSON APIs that do not follow strong conventions, Ember Data is designed to be easily configurable to work with whatever data your server returns.

Ember Data is also designed to work with streaming APIs like socket.io, Firebase, or WebSockets. You can open a socket to your server and push changes to records into the store whenever they occur.

Currently, Ember Data ships as a separate library from Ember.js. Until Ember Data is included as part of the standard distribution, you can get a copy of the latest passing build from emberjs.com/builds[213]:

- Development[214]
- Minified[215]

[211] http://emberjs.com/guides/routing/specifying-a-routes-model

[212] https://github.com/emberjs/data

[213] http://emberjs.com/builds

[214] http://builds.emberjs.com/canary/ember-data.js

[215] http://builds.emberjs.com/canary/ember-data.min.js

Core Concepts

Learning to use Ember Data is easiest once you understand some of the concepts that underpin its design.

Store

The **store** is the central repository of records in your application. You can think of the store as a cache of all of the records available in your app. Both your application's controllers and routes have access to this shared store; when they need to display or modify a record, they will first ask the store for it.

This instance of `DS.Store` is created for you automatically and is shared among all of the objects in your application.

You will use the store to retrieve records, as well to create new ones. For example, we might want to find an `App.Person` model with the ID of `1` from our route's `model` hook:

```
App.IndexRoute = Ember.Route.extend({
  model: function() {
    return this.store.find('person', 1);
  }
});
```

Models

A **model** is a class that defines the properties and behavior of the data that you present to the user. Anything that the user expects to see if they leave your app and come back later (or if they refresh the page) should be represented by a model.

For example, if you were writing a web application for placing orders at a restaurant, you might have models like `Order`, `LineItem`, and `MenuItem`.

Fetching orders becomes very easy:

```
this.store.find('order');
```

Models define the type of data that will be provided by your server. For example, a `Person` model might have a `firstName` attribute that is a string, and a `birthday` attribute that is a date:

```
App.Person = DS.Model.extend({
  firstName: DS.attr('string'),
  birthday:  DS.attr('date')
});
```

A model also describes its relationships with other objects. For example, an `Order` may have many `LineItems`, and a `LineItem` may belong to a particular `Order`.

```
App.Order = DS.Model.extend({
  lineItems: DS.hasMany('lineItem')
});

App.LineItem = DS.Model.extend({
  order: DS.belongsTo('order')
});
```

Models don't have any data themselves; they just define the properties and behavior of specific instances, which are called *records*.

Records

A **record** is an instance of a model that contains data loaded from a server. Your application can also create new records and save them back to the server.

A record is uniquely identified by its model type and id.

For example, if you were writing a contact management app, you might have a model called `Person`. An individual record in your app might have a type of `Person` and an ID of `1` or `steve-buscemi`.

```
this.store.find('person', 1); // => { id: 1, name: 'steve-buscemi' }
```

IDs are usually assigned by the server when you save them for the first time, but you can also generate IDs client-side.

Adapter

An **adapter** is an object that knows about your particular server backend and is responsible for translating requests for and changes to records into the appropriate calls to your server.

For example, if your application asks for a `person` record with an ID of `1`, how should Ember Data load it? Is it over HTTP or a WebSocket? If it's HTTP, is the URL `/person/1` or `/resources/people/1`?

The adapter is responsible for answering all of these questions. Whenever your app asks the store for a record that it doesn't have cached, it will ask the adapter for it. If you change a record and save it, the store will hand the record to the adapter to send the appropriate data to your server and confirm that the save was successful.

Serializer

A **serializer** is responsible for turning a raw JSON payload returned from your server into a record object.

JSON APIs may represent attributes and relationships in many different ways. For example, some attribute names may be `camelCased` and others may be `under_scored`. Representing relationships is even more diverse: they may be encoded as an array of IDs, an array of embedded objects, or as foreign keys.

When the adapter gets a payload back for a particular record, it will give that payload to the serializer to normalize into the form that Ember Data is expecting.

While most people will use a serializer for normalizing JSON, because Ember Data treats these payloads as opaque objects, there's no reason they couldn't be binary data stored in a `Blob` or ArrayBuffer[216].

Automatic Caching

The store will automatically cache records for you. If a record had already been loaded, asking for it a second time will always return the same object instance. This minimizes the number of round-trips to the server, and allows your application to render its UI to the user as fast as possible.

For example, the first time your application asks the store for a `person` record with an ID of `1`, it will fetch that information from your server.

However, the next time your app asks for a `person` with ID `1`, the store will notice that it had already retrieved and cached that information from the server. Instead of sending another request for the same information, it will give your application the same record it had provided it the first time. This feature—always returning the same record object, no matter how many times you look it up—is sometimes called an *identity map*.

Using an identity map is important because it ensures that changes you make in one part of your UI are propagated to other parts of the UI. It also means that you don't have to manually keep records in sync—you can ask for a record by ID and not have to worry about whether other parts of your application have already asked for and loaded it.

Architecture Overview

The first time your application asks the store for a record, the store sees that it doesn't have a local copy and requests it from your adapter. Your adapter will go and retrieve the record from your persistence layer; typically, this will be a JSON representation of the record served from an HTTP server.

[216] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays/ArrayBuffer

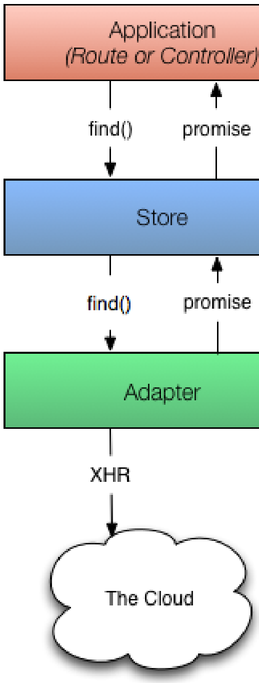


Diagram showing process for finding an unloaded record

As illustrated in the diagram above, the adapter cannot always return the requested record immediately. In this case, the adapter must make an *asynchronous* request to the server, and only when that request finishes loading can the record be created with its backing data.

Because of this asynchronicity, the store immediately returns a *promise* from the `find()` method. Similarly, any requests that the store makes to the adapter also return promises.

Once the request to the server returns with a JSON payload for the requested record, the adapter resolves the promise it returned to the store with the JSON.

The store then takes that JSON, initializes the record with the JSON data, and resolves the promise returned to your application with the newly-loaded record.

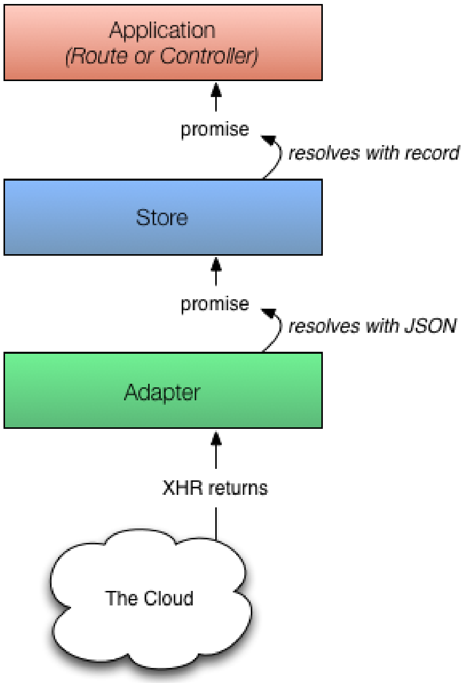


Diagram showing process for finding an unloaded record after the payload has returned from the server

Let's look at what happens if you request a record that the store already has in its cache.

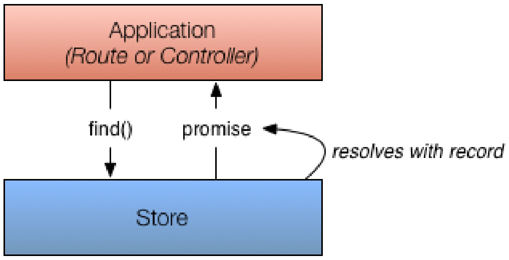


Diagram showing process for finding an unloaded record after the payload has returned from the server

In this case, because the store already knew about the record, it returns a promise that it resolves with the record immediately. It does not need to ask the adapter (and, therefore, the server) for a copy since it already has it saved locally.

These are the core concepts you should understand to get the most out of Ember Data. The following sections go into more depth about each of these concepts, and how to use them together.

Defining Models

A model is a class that defines the properties and behavior of the data that you present to the user. Anything that the user expects to see if they leave your app and come back later (or if they refresh the page) should be represented by a model.

Make sure to include `ember-data.js` after `ember.js`

```
<script type="text/javascript" src="ember.js"></script>
<script type="text/javascript" src="ember-data.js"></script>
```

For every model in your application, create a subclass of `DS.Model`:

```
App.Person = DS.Model.extend();
```

After you have defined a model class, you can start finding and creating records of that type. When interacting with the store, you will need to specify a record's type using the model name. For example, the store's `find()` method expects a string as the first argument to tell it what type of record to find:

```
store.find('person', 1);
```

The table below shows how model names map to model classes.

<table> <thead> <tr> <th>Model Name</th> <th>Model Class</th> </tr> </thead> <tr> <td><code>photo</code></td
<td><code>App.Photo</code></td> </tr> <tr> <td><code>adminUserProfile</code></td> <td><code>App.AdminUs
</tr> </table>

Defining Attributes

You can specify which attributes a model has by using `DS.attr`.

```
var attr = DS.attr;

App.Person = DS.Model.extend({
  firstName: attr(),
  lastName: attr(),
  birthday: attr()
});
```

Attributes are used when turning the JSON payload returned from your server into a record, and when serializing a record to save back to the server after it has been modified.

You can use attributes just like any other property, including as part of a computed property. Frequently, you will want to define computed properties that combine or transform primitive attributes.

```
var attr = DS.attr;

App.Person = DS.Model.extend({
  firstName: attr(),
  lastName: attr(),

  fullName: function() {
    return this.get('firstName') + ' ' + this.get('lastName');
  }.property('firstName', 'lastName')
});
```

For more about adding computed properties to your classes, see Computed Properties[217].

If you don't specify the type of the attribute, it will be whatever was provided by the server. You can make sure that an attribute is always coerced into a particular type by passing a `type` to `attr`:

```
App.Person = DS.Model.extend({
  birthday: DS.attr('date')
});
```

The default adapter supports attribute types of `string`, `number`, `boolean`, and `date`. Custom adapters may offer additional attribute types, and new types can be registered as transforms. See the documentation section on the REST Adapter[218].

Please note: Ember Data serializes and deserializes dates according to ISO 8601[219]. For example: `2014-05-27T12:54:01`

Options

`DS.attr` takes an optional hash as a second parameter:

- `defaultValue`: Pass a string or a function to be called to set the attribute to a default value if none is supplied.

  Example

[217] http://emberjs.com/guides/object-model/computed-properties

[218] http://emberjs.com/guides/models/the-rest-adapter

[219] http://en.wikipedia.org/wiki/ISO_8601

```
var attr = DS.attr;

App.User = DS.Model.extend({
    username: attr('string'),
    email: attr('string'),
    verified: attr('boolean', {defaultValue: false}),
    createdAt: DS.attr('string', {
        defaultValue: function() { return new Date(); }
    })
});
```

Defining Relationships

Ember Data includes several built-in relationship types to help you define how your models relate to each other.

One-to-One

To declare a one-to-one relationship between two models, use `DS.belongsTo`:

```
App.User = DS.Model.extend({
  profile: DS.belongsTo('profile')
});

App.Profile = DS.Model.extend({
  user: DS.belongsTo('user')
});
```

One-to-Many

To declare a one-to-many relationship between two models, use `DS.belongsTo` in combination with `DS.hasMany`, like this:

```
App.Post = DS.Model.extend({
  comments: DS.hasMany('comment')
});

App.Comment = DS.Model.extend({
  post: DS.belongsTo('post')
});
```

Many-to-Many

To declare a many-to-many relationship between two models, use `DS.hasMany`:

```
App.Post = DS.Model.extend({
  tags: DS.hasMany('tag')
});

App.Tag = DS.Model.extend({
  posts: DS.hasMany('post')
});
```

Explicit Inverses

Ember Data will do its best to discover which relationships map to one another. In the one-to-many code above, for example, Ember Data can figure out that changing the `comments` relationship should update the `post` relationship on the inverse because `post` is the only relationship to that model.

However, sometimes you may have multiple `belongsTo`/`hasMany`s for the same type. You can specify which property on the related model is the inverse using `DS.hasMany`'s `inverse` option:

```
var belongsTo = DS.belongsTo,
    hasMany = DS.hasMany;

App.Comment = DS.Model.extend({
  onePost: belongsTo('post'),
  twoPost: belongsTo('post'),
  redPost: belongsTo('post'),
  bluePost: belongsTo('post')
});


App.Post = DS.Model.extend({
  comments: hasMany('comment', {
    inverse: 'redPost'
  })
});
```

You can also specify an inverse on a `belongsTo`, which works how you'd expect.

Reflexive relation

When you want to define a reflexive relation, you must either explicitly define the other side, and set the explicit inverse accordingly, and if you don't need the other side, set the inverse to null.

```
var belongsTo = DS.belongsTo,
    hasMany = DS.hasMany;

App.Folder = DS.Model.extend({
  children: hasMany('folder', {inverse: 'parent'}),
  parent: belongsTo('folder', {inverse: 'children'})
});
```

or

```
var belongsTo = DS.belongsTo,

App.Folder = DS.Model.extend({
  parent: belongsTo('folder', {inverse: null})
});
```

Creating Deleting Records

You can create records by calling the `createRecord` method on the store.

```
store.createRecord('post', {
  title: 'Rails is Omakase',
  body: 'Lorem ipsum'
});
```

The store object is available in controllers and routes using `this.store`.

Although `createRecord` is fairly straightforward, the only thing to watch out for is that you cannot assign a promise as a relationship, currently.

For example, if you want to set the `author` property of a post, this would **not** work if the `user` with id isn't already loaded into the store:

```
var store = this.store;

store.createRecord('post', {
  title: 'Rails is Omakase',
  body: 'Lorem ipsum',
  author: store.find('user', 1)
});
```

However, you can easily set the relationship after the promise has fulfilled:

```
var store = this.store;

var post = store.createRecord('post', {
  title: 'Rails is Omakase',
  body: 'Lorem ipsum'
});

store.find('user', 1).then(function(user) {
  post.set('author', user);
});
```

Deleting Records

Deleting records is just as straightforward as creating records. Just call `deleteRecord()` on any instance of `DS.Model`. This flags the record as `isDeleted` and thus removes it from `all()` queries on the `store`. The deletion can then be persisted using `save()`. Alternatively, you can use the `destroyRecord` method to delete and persist at the same time.

```
store.find('post', 1).then(function (post) {
  post.deleteRecord();
  post.get('isDeleted'); // => true
  post.save(); // => DELETE to /posts/1
});

// OR
store.find('post', 2).then(function (post) {
  post.destroyRecord(); // => DELETE to /posts/2
});
```

Pushing Records Into The Store

One way to think about the store is as a cache of all of the records that have been loaded by your application. If a route or a controller in your app asks for a record, the store can return it immediately if it is in the cache. Otherwise, the store must ask the adapter to load it, which usually means a trip over the network to retrieve it from the server.

Instead of waiting for the app to request a record, however, you can push records into the store's cache ahead of time.

This is useful if you have a good sense of what records the user will need next. When they click on a link, instead of waiting for a network request to finish, Ember.js can render the new template immediately. It feels instantaneous.

Another use case for pushing in records is if your application has a streaming connection to a backend. If a record is created or modified, you want to update the UI immediately.

Pushing Records

To push a record into the store, call the store's `push()` method.

For example, imagine we want to preload some data into the store when the application boots for the first time.

We can use the `ApplicationRoute` to do so. The `ApplicationRoute` is the top-most route in the route hierarchy, and its `model` hook gets called once when the app starts up.

```
var attr = DS.attr;

App.Album = DS.Model.extend({
  title: attr(),
  artist: attr(),
  songCount: attr()
});

App.ApplicationRoute = Ember.Route.extend({
  model: function() {
    this.store.push('album', {
      id: 1,
      title: "Fewer Moving Parts",
      artist: "David Bazan",
      songCount: 10
    });

    this.store.push('album', {
      id: 2,
      title: "Calgary b/w I Can't Make You Love Me/Nick Of Time",
      artist: "Bon Iver",
      songCount: 2
    });
  }
});
```

Persisting Records

Records in Ember Data are persisted on a per-instance basis. Call `save()` on any instance of `DS.Model` and it will make a network request.

Here are a few examples:

```
var post = store.createRecord('post', {
  title: 'Rails is Omakase',
  body: 'Lorem ipsum'
});

post.save(); // => POST to '/posts'
```

```
store.find('post', 1).then(function (post) {
  post.get('title'); // => "Rails is Omakase"

  post.set('title', 'A new post');

  post.save(); // => PUT to '/posts/1'
});
```

Promises

`save()` returns a promise, so it is extremely easy to handle success and failure scenarios. Here's a common pattern:

```
var post = store.createRecord('post', {
  title: 'Rails is Omakase',
  body: 'Lorem ipsum'
});

var self = this;

function transitionToPost(post) {
  self.transitionToRoute('posts.show', post);
}

function failure(reason) {
  // handle the error
}

post.save().then(transitionToPost).catch(failure);

// => POST to '/posts'
// => transitioning to posts.show route
```

Promises even make it easy to work with failed network requests:

```
var post = store.createRecord('post', {
  title: 'Rails is Omakase',
  body: 'Lorem ipsum'
});

var self = this;

var onSuccess = function(post) {
  self.transitionToRoute('posts.show', post);
};

var onFail = function(post) {
  // deal with the failure here
};

post.save().then(onSuccess, onFail);

// => POST to '/posts'
// => transitioning to posts.show route
```

You can read more about promises here[220], but here is another example showing how to retry persisting:

```
function retry(callback, nTimes) {
  // if the promise fails
  return callback().catch(function(reason) {
    // if we haven't hit the retry limit
    if (nTimes-- > 0) {
      // retry again with the result of calling the retry callback
      // and the new retry limit
      return retry(callback, nTimes);
    }

    // otherwise, if we hit the retry limit, rethrow the error
    throw reason;
  });
}

// try to save the post up to 5 times
retry(function() {
  return post.save();
}, 5);
```

[220] https://github.com/tildeio/rsvp.js

Finding Records

The Ember Data store provides a simple interface for finding records of a single type through the store object's find method. Internally, the store uses find, findAll, and findQuery based on the supplied arguments.

The first argument to store.find() is always the record type. The optional second argument determines if a request is made for all records, a single record, or a query.

Finding All Records of a Type

```
var posts = this.store.find('post'); // => GET /posts
```

To get a list of records already loaded into the store, without making another network request, use all instead.

```
var posts = this.store.all('post'); // => no network request
```

find returns a DS.PromiseArray that fulfills to a DS.RecordArray and all directly returns a DS.RecordArray.

It's important to note that DS.RecordArray is not a JavaScript array. It is an object that implements Ember.Enumerable[221]. This is important because, for example, if you want to retrieve records by index, the [] notation will not work–you'll have to use objectAt(index) instead.

Finding a Single Record

If you provide a number or string as the second argument to store.find(), Ember Data will assume that you are passing in an ID and attempt to retrieve a record of the type passed in as the first argument with that ID. This will return a promise that fulfills with the requested record:

```
var aSinglePost = this.store.find('post', 1); // => GET /posts/1
```

Querying For Records

If you provide a plain object as the second argument to find, Ember Data will make a GET request with the object serialized as query params. This method returns DS.PromiseArray in the same way as find with no second argument.

For example, we could search for all person models who have the name of Peter:

[221] http://emberjs.com/guides/models/defining-models

```
var peters = this.store.find('person', { name: "Peter" }); // => GET to /persons\
?name='Peter'
```

Integrating with the Route's Model Hook

As discussed in Specifying a Route's Model[222], routes are responsible for telling their template which model to render.

`Ember.Route`'s `model` hook supports asynchronous values out-of-the-box. If you return a promise from the `model` hook, the router will wait until the promise has fulfilled to render the template.

This makes it easy to write apps with asynchronous data using Ember Data. Just return the requested record from the `model` hook, and let Ember deal with figuring out whether a network request is needed or not.

```
App.Router.map(function() {
  this.resource('posts');
  this.resource('post', { path: ':post_id' });
});

App.PostsRoute = Ember.Route.extend({
  model: function() {
    return this.store.find('post');
  }
});

App.PostRoute = Ember.Route.extend({
  model: function(params) {
    return this.store.find('post', params.post_id);
  }
})
```

Working With Records

Modifying Attributes

Once a record has been loaded, you can begin making changes to its attributes. Attributes behave just like normal properties in Ember.js objects. Making changes is as simple as setting the attribute you want to change:

[222] http://emberjs.com/guides/models/the-rest-adapter

```
var tyrion = this.store.find('person', 1);
// ...after the record has loaded
tyrion.set('firstName', "Yollo");
```

All of the Ember.js conveniences are available for modifying attributes. For example, you can use `Ember.Object`'s `incrementProperty` helper:

```
person.incrementProperty('age'); // Happy birthday!
```

You can tell if a record has outstanding changes that have not yet been saved by checking its `isDirty` property. You can also see what parts of the record were changed and what the original value was using the `changedAttributes` function. `changedAttributes` returns an object, whose keys are the changed properties and values are an array of values `[oldValue, newValue]`.

```
person.get('isAdmin');      //=> false
person.get('isDirty');      //=> false
person.set('isAdmin', true);
person.get('isDirty');      //=> true
person.changedAttributes(); //=> { isAdmin: [false, true] }
```

At this point, you can either persist your changes via `save()` or you can rollback your changes. Calling `rollback()` reverts all the `changedAttributes` to their original value.

```
person.get('isDirty');      //=> true
person.changedAttributes(); //=> { isAdmin: [false, true] }

person.rollback();

person.get('isDirty');      //=> false
person.get('isAdmin');      //=> false
person.changedAttributes(); //=> {}
```

Using Fixtures

When developing client-side applications, your server may not have an API ready to develop against. The FixtureAdapter allows you to begin developing Ember.js apps now, and switch to another adapter when your API is ready to consume without any changes to your application code.

Getting Started

Using the fixture adapter entails three very simple setup steps:

1. Create a new store using the fixture adapter and attach it to your app.
2. Define your model using `DS.Model.extend`.
3. Attach fixtures(also known as sample data) to the model's class.

Creating a Fixture Adapter

Simply attach it as the `ApplicationAdapter` property on your instance of `Ember.Application`:

```
var App = Ember.Application.create();
App.ApplicationAdapter = DS.FixtureAdapter;
```

Define Your Model

You should refer to Defining a Model[223] for a more in-depth guide on using Ember Data Models, but for the purposes of demonstration we'll use an example modeling people who document Ember.js.

```
App.Documenter = DS.Model.extend({
  firstName: DS.attr( 'string' ),
  lastName: DS.attr( 'string' )
});
```

Attach Fixtures to the Model Class

Attaching fixtures couldn't be simpler. Just attach a collection of plain JavaScript objects to your Model's class under the `FIXTURES` property:

```
App.Documenter.FIXTURES = [
  { id: 1, firstName: 'Trek', lastName: 'Glowacki' },
  { id: 2, firstName: 'Tom' , lastName: 'Dale'     }
];
```

That's it! You can now use all of methods for Finding Records[224] in your application. For example:

[223] http://emberjs.com/guides/models/defining-models
[224] http://emberjs.com/guides/models/finding-records

```
App.DocumenterRoute = Ember.Route.extend({
  model: function() {
    return this.store.find('documenter', 1); // returns a promise that will reso\
lve
                                             // with the record representing Tre\
k Glowacki
  }
});
```

Naming Conventions

Unlike the REST Adapter[225], the Fixture Adapter does not make any assumptions about the naming conventions of your model. As you saw in the example above, if you declare the attribute as `firstName` during `DS.Model.extend`, you use `firstName` to represent the same field in your fixture data.

Importantly, you should make sure that each record in your fixture data has a uniquely identifiable field. By default, Ember Data assumes this key is called `id`. Should you not provide an `id` field in your fixtures, or not override the primary key, the Fixture Adapter will throw an error.

The REST Adapter

By default, your store will use DS.RESTAdapter[226] to load and save records. The RESTAdapter assumes that the URLs and JSON associated with each model are conventional; this means that, if you follow the rules, you will not need to configure the adapter or write any code in order to get started.

URL Conventions

The REST adapter is smart enough to determine the URLs it communicates with based on the name of the model. For example, if you ask for a `Post` by ID:

```
store.find('post', 1).then(function(post) {
});
```

The REST adapter will automatically send a `GET` request to `/posts/1`.

The actions you can take on a record map onto the following URLs in the REST adapter:

<table> <thead> <tr><th>Action</th><th>HTTP Verb</th><th>URL</th></tr> </thead> <tbody> <tr><th>Find</th><td>GET</td><td>/people/123</td></tr> <tr><th>Find All</th><td>GET</td><td>/people</td><
<tr><th>Update</th><td>PUT</td><td>/people/123</td></tr> <tr><th>Create</th><td>POST</td><td>/people</td
<tr><th>Delete</th><td>DELETE</td><td>/people/123</td></tr> </tbody> </table>

[225] http://emberjs.com/guides/models/the-rest-adapter

[226] http://emberjs.com/api/data/classes/DS.RESTAdapter.html

Pluralization Customization

Irregular or uncountable pluralizations can be specified via `Ember.Inflector.inflector`:

```
var inflector = Ember.Inflector.inflector;

inflector.irregular('formula', 'formulae');
inflector.uncountable('advice');
```

This will tell the REST adapter that requests for `App.Formula` requests should go to `/formulae/1` instead of `/formulas/1`.

Endpoint Path Customization

Endpoint paths can be prefixed with a namespace by setting the `namespace` property on the adapter:

```
App.ApplicationAdapter = DS.RESTAdapter.extend({
  namespace: 'api/1'
});
```

Requests for `App.Person` would now target `/api/1/people/1`.

Host Customization

An adapter can target other hosts by setting the `host` property.

```
App.ApplicationAdapter = DS.RESTAdapter.extend({
  host: 'https://api.example.com'
});
```

Requests for `App.Person` would now target `https://api.example.com/people/1`.

JSON Conventions

When requesting a record, the REST adapter expects your server to return a JSON representation of the record that conforms to the following conventions.

JSON Root

The primary record being returned should be in a named root. For example, if you request a record from `/people/123`, the response should be nested inside a property called `person`:

```
{
  "person": {
    "firstName": "Jeff",
    "lastName": "Atwood"
  }
}
```

Note: Although after `destroyRecord` or `deleteRecord`/`save` the adapter expects an empty object e.g. `{}` to be returned from the server after destroying a record.

If you don't have the option to change the data that the server responds with, you can override the DS.JSONSerializer#extractDeleteRecord[227], like so:

```
extractDeleteRecord: function(store, type, payload) {
  // payload is {delete: true} and then ember data wants to go ahead and set
  // the new properties, return null so it doesn't try to do that
  return null;
}
```

Attribute Names

Attribute names should be camelized. For example, if you have a model like this:

```
App.Person = DS.Model.extend({
  firstName: DS.attr('string'),
  lastName:  DS.attr('string'),

  isPersonOfTheYear: DS.attr('boolean')
});
```

The JSON returned from your server should look like this:

```
{
  "person": {
    "firstName": "Barack",
    "lastName": "Obama",
    "isPersonOfTheYear": true
  }
}
```

Irregular keys can be mapped with a custom serializer. If the JSON for the `Person` model has a key of `lastNameOfPerson`, and the desired attribute name is simply `lastName`, then create a custom Serializer for the model and override the `normalizeHash` property.

[227] http://emberjs.com/api/data/classes/DS.JSONSerializer.html#method_extractDeleteRecord

```
App.Person = DS.Model.extend({
  lastName: DS.attr('string')
});

App.PersonSerializer = DS.RESTSerializer.extend({
  normalizeHash: {
    lastNameOfPerson: function(hash) {
      hash.lastName = hash.lastNameOfPerson;
      delete hash.lastNameOfPerson;

      return hash;
    }
  }
});
```

Relationships

References to other records should be done by ID. For example, if you have a model with a `hasMany` relationship:

```
App.Post = DS.Model.extend({
  comments: DS.hasMany('comment', {async: true})
});
```

The JSON should encode the relationship as an array of IDs:

```
{
  "post": {
    "comments": [1, 2, 3]
  }
}
```

`Comments` for a `post` can be loaded by `post.get('comments')`. The REST adapter will send a `GET` request to `/comments?ids[]=1&ids[]=2&ids[]=3`.

Any `belongsTo` relationships in the JSON representation should be the camelized version of the Ember Data model's name, with the string `Id` appended. For example, if you have a model:

```
App.Comment = DS.Model.extend({
  post: DS.belongsTo('post')
});
```

The JSON should encode the relationship as an ID to another record:

```
{
  "comment": {
    "post": 1
  }
}
```

If needed these naming conventions can be overwritten by implementing the `keyForRelationship` method.

```
 App.ApplicationSerializer = DS.RESTSerializer.extend({
   keyForRelationship: function(key, relationship) {
      return key + 'Ids';
   }
 });
```

Sideloaded Relationships

To reduce the number of HTTP requests necessary, you can sideload additional records in your JSON response. Sideloaded records live outside the JSON root, and are represented as an array of hashes:

```
{
  "post": {
    "id": 1,
    "title": "Node is not omakase",
    "comments": [1, 2, 3]
  },

  "comments": [{
    "id": 1,
    "body": "But is it _lightweight_ omakase?"
  },
  {
    "id": 2,
    "body": "I for one welcome our new omakase overlords"
  },
  {
    "id": 3,
    "body": "Put me on the fast track to a delicious dinner"
  }]
}
```

Creating Custom Transformations

In some circumstances, the built in attribute types of `string`, `number`, `boolean`, and `date` may be inadequate. For example, a server may return a non-standard date format.

Ember Data can have new JSON transforms registered for use as attributes:

```
App.CoordinatePointTransform = DS.Transform.extend({
  serialize: function(value) {
    return [value.get('x'), value.get('y')];
  },
  deserialize: function(value) {
    return Ember.create({ x: value[0], y: value[1] });
  }
});

App.Cursor = DS.Model.extend({
  position: DS.attr('coordinatePoint')
});
```

When `coordinatePoint` is received from the API, it is expected to be an array:

```
{
  cursor: {
    position: [4,9]
  }
}
```

But once loaded on a model instance, it will behave as an object:

```
var cursor = App.Cursor.find(1);
cursor.get('position.x'); //=> 4
cursor.get('position.y'); //=> 9
```

If `position` is modified and saved, it will pass through the `serialize` function in the transform and again be presented as an array in JSON.

Connecting to an HTTP Server

If your Ember application needs to load JSON data from an HTTP server, this guide will walk you through the process of configuring Ember Data to load records in whatever format your server returns.

The store uses an object called an *adapter* to know how to communicate over the network. By default, the store will use `DS.RESTAdapter`, an adapter that communicates with an HTTP server by transmitting JSON via XHR.

This guide is divided into two sections. The first section covers what the default behavior of the adapter is, including what URLs it will request records from and what format it expects the JSON to be in.

The second section covers how to override these default settings to customize things like which URLs data is requested from and how the JSON data is structured.

URL Conventions

The REST adapter uses the name of the model to determine what URL to send JSON to.

For example, if you ask for an `App.Photo` record by ID:

```
App.PhotoRoute = Ember.Route.extend({
  model: function(params) {
    return this.store.find('photo', params.photo_id);
  }
});
```

The REST adapter will automatically send a `GET` request to `/photos/1`.

The actions you can take on a record map onto the following URLs in the REST adapter:

<table> <thead> <tr><th>Action</th><th>HTTP Verb</th><th>URL</th></tr> </thead> <tbody> <tr><th>Find</th><td>GET</td><td>/photos/123</td></tr> <tr><th>Find All</th><td>GET</td><td>/photos</td>< <tr><th>Update</th><td>PUT</td><td>/photos/123</td></tr> <tr><th>Create</th><td>POST</td><td>/photos</td <tr><th>Delete</th><td>DELETE</td><td>/photos/123</td></tr> </tbody> </table>

JSON Conventions

Given the following models:

```
App.Post = DS.Model.extend({
  title:    DS.attr(),
  comments: DS.hasMany('comment'),
  user:     DS.belongsTo('user')
});

App.Comment = DS.Model.extend({
  body: DS.attr()
});
```

Ember Data expects that a `GET` request to `/posts/1` would return the JSON in the following format:

```
{
  "post": {
    "id": 1,
    "title": "Rails is omakase",
    "comments": ["1", "2"],
    "user" : "dhh"
  },

  "comments": [{
    "id": "1",
    "body": "Rails is unagi"
  }, {
    "id": "2",
    "body": "Omakase O_o"
  }]
}
```

To quickly prototype a model and see the expected JSON, try using the Ember Data Model Maker[228] by Andy Crum.

Customizing the Adapter

To customize the REST adapter, define a subclass of `DS.RESTAdapter` and name it `App.ApplicationAdapter`. You can then override its properties and methods to customize how records are retrieved and saved.

Customizing a Specific Model

It's entirely possible that you need to define options for just one model instead of an application-wide customization. In that case, you can create an adapter named after the model you are specifying:

[228] http://andycrum.github.io/ember-data-model-maker/

```
App.PostAdapter = DS.RESTAdapter.extend({
  namespace: 'api/v2',
  host: 'https://api.example2.com'
});

App.PhotoAdapter = DS.RESTAdapter.extend({
  namespace: 'api/v1',
  host: 'https://api.example.com'
});
```

This allows you to easily connect to multiple API versions simultaneously or interact with different domains on a per model basis.

Customizing URLs

URL Prefix

If your JSON API lives somewhere other than on the host root, you can set a prefix that will be added to all requests.

For example, if you are using a versioned JSON API, a request for a particular person might go to `/api/v1/people/1`.

In that case, set `namespace` property to `api/v1`.

```
App.ApplicationAdapter = DS.RESTAdapter.extend({
  namespace: 'api/v1'
});
```

Requests for a `person` with ID `1` would now go to `/api/v1/people/1`.

URL Host

If your JSON API runs on a different domain than the one serving your Ember app, you can change the host used to make HTTP requests.

Note that in order for this to work, you will need to be using a browser that supports CORS[229], and your server will need to be configured to send the correct CORS headers.

To change the host that requests are sent to, set the `host` property:

[229] http://www.html5rocks.com/en/tutorials/cors/

```
App.ApplicationAdapter = DS.RESTAdapter.extend({
  host: 'https://api.example.com'
});
```

Requests for a `person` with ID `1` would now target `https://api.example.com/people/1`.

Custom HTTP Headers

Some APIs require HTTP headers, e.g. to provide an API key. Arbitrary headers can be set as key/value pairs on the `RESTAdapter`'s `headers` property and Ember Data will send them along with each ajax request.

For Example

```
App.ApplicationAdapter = DS.RESTAdapter.extend({
  headers: {
    'API_KEY': 'secret key',
    'ANOTHER_HEADER': 'Some header value'
  }
});
```

Requests for any resource will include the following HTTP headers.

```
ANOTHER_HEADER: Some header value
API_KEY: secret key
```

Handling Metadata

Along with the records returned from your store, you'll likely need to handle some kind of metadata. *Metadata* is data that goes along with a specific *model* or *type* instead of a record.

Pagination is a common example of using metadata. Imagine a blog with far more posts than you can display at once. You might query it like so:

```
var result = this.store.find("post", {
  limit: 10,
  offset: 0
});
```

To get different *pages* of data, you'd simply change your offset in increments of 10. So far, so good. But how do you know how many pages of data you have? Your server would need to return the total number of records as a piece of metadata.

By default, Ember Data's JSON deserializer looks for a `meta` key:

```
{
  "post": {
    "id": 1,
    "title": "Progressive Enhancement is Dead",
    "comments": ["1", "2"],
    "links": {
      "user": "/people/tomdale"
    },
    // ...
  },

  "meta": {
    "total": 100
  }
}
```

The metadata for a specific type is then set to the contents of `meta`. You can access it either with `store.metadataFor`, which is updated any time any query is made against the same type:

```
var meta = this.store.metadataFor("post");
```

Or you can access the metadata just for this query:

```
var meta = result.get("content.meta");
```

Now, `meta.total` can be used to calculate how many pages of posts you'll have.

You can also customize metadata extraction by overriding the `extractMeta` method. For example, if instead of a `meta` object, your server simply returned:

```
{
  "post": [
    // ...
  ],
  "total": 100
}
```

You could extract it like so:

```
App.ApplicationSerializer = DS.RESTSerializer.extend({
  extractMeta: function(store, type, payload) {
    if (payload && payload.total) {
      store.metaForType(type, { total: payload.total });  // sets the metadata f\
or "post"
      delete payload.total;  // keeps ember data from trying to parse "total" as\
 a record
    }
  }
});
```

Customizing Adpters

In Ember Data, the logic for communicating with a backend data store lives in the `Adapter`. Ember Data's Adapter has some built-in assumptions of how a REST API[230] should look. If your backend conventions differ from these assumptions Ember Data makes it easy to change its functionality by swapping out or extending the default Adapter.

Some reasons for customizing an Adapter include using `underscores_case` in your urls, using a medium other than REST to communicate with your backend API or even using a local backend[231].

Extending Adapters is a natural process in Ember Data. Ember takes the position that you should extend an adapter to add different functionality instead of adding a flag. This results in code that is more testable, easier to understand and reduces bloat for people who may want to subclass your adapter.

If your backend has some consistent rules you can define an `ApplicationAdapter`. The `ApplicationAdapter` will get priority over the default Adapter, however it will still be superseded by model specific Adapters.

```
App.ApplicationAdapter = DS.RESTAdapter.extend({
  // Application specific overrides go here
});
```

If you have one model that has exceptional rules for communicating with its backend than the others you can create a Model specific Adapter by naming an adapter "ModelName" + "Adapter".

[230] http://jsonapi.org/

[231] https://github.com/rpflorence/ember-localstorage-adapter

```
App.PostAdapter = DS.RESTAdapter.extend({
  namespace: 'api/v1'
});
```

By default Ember Data comes with several builtin adapters. Feel free to use these adapters as a starting point for creating your own custom adapter.

- DS.Adapter[232] is the basic adapter with no functionality. It is generally a good starting point if you want to create an adapter that is radically different from the other Ember adapters.
- DS.FixtureAdapter[233] is an adapter that loads records from memory. Its primarily used for development and testing.
- DS.RESTAdapter[234] is the most commonly extended adapter. The `RESTAdapter` allows your store to communicate with an HTTP server by transmitting JSON via XHR. Most Ember.js apps that consume a JSON API should use the REST adapter.
- DS.ActiveModelAdapter[235] is a specialized version of the `RESTAdapter` that is set up to work out of the box with Rails-style REST APIs.

Customizing the RESTAdapter

The DS.RESTAdapter[236] is the most commonly extended adapter that ships with Ember Data. It has a handful of hooks that are commonly used to extend it to work with non-standard backends.

Endpoint Path Customization

The `namespace` property can be used to prefix requests with a specific url namespace.

```
App.ApplicationAdapter = DS.RESTAdapter.extend({
  namespace: 'api/1'
});
```

Requests for `App.Person` would now target `/api/1/people/1`.

Host Customization

By default the adapter will target the current domain. If you would like to specify a new domain you can do so by setting the `host` property on the adapter.

[232] http://emberjs.com/api/data/classes/DS.Adapter.html
[233] http://emberjs.com/api/data/classes/DS.FixtureAdapter.html
[234] http://emberjs.com/api/data/classes/DS.RESTAdapter.html
[235] http://emberjs.com/api/data/classes/DS.ActiveModelAdapter.html
[236] http://emberjs.com/api/data/classes/DS.RESTAdapter.html

```
App.ApplicationAdapter = DS.RESTAdapter.extend({
  host: 'https://api.example.com'
});
```

Requests for `App.Person` would now target `https://api.example.com/people/1`.

Path Customization

By default the `RESTAdapter` will attempt to pluralize and camelCase the model name to generate the path name. If this convention does not conform to your backend you can override the `pathForType` method.

For example, if you did not want to pluralize model names and needed underscore_case instead of camelCase you could override the `pathForType` method like this:

```
App.ApplicationAdapter = DS.RESTAdapter.extend({
  pathForType: function(type) {
    return Ember.String.underscore(type);
  }
});
```

Requests for `App.Person` would now target `/person/1`. Requests for `App.UserProfile` would now target `/user_profile/1`.

Authoring Adapters

The `defaultSerializer` property can be used to specify the serializer that will be used by this adapter. This is only used when a model specific serializer or ApplicationSerializer are not defined.

In an application, it is often easier to specify an `ApplicationSerializer`. However, if you are the author of a community adapter it is important to remember to set this property to ensure Ember does the right thing in the case a user of your adapter does not specify an `ApplicationSerializer`.

```
MyCustomAdapterAdapter = DS.RESTAdapter.extend({
  defaultSerializer: '-default'
});
```

Community Adapters

If none of the builtin Ember Data Adapters work for your backend, be sure to check out some of the community maintained Ember Data Adapters. Some good places to look for Ember Data Adapters include:

- GitHub[237]
- Bower[238]

[237] https://github.com/search?q=ember+data+adapter&ref=cmdform

[238] http://bower.io/search/?q=ember-data-

Frequently Asked Questions

Should I use a query or a filter to search records?

It depends on how many records you want to search and whether they have been loaded into the store.

Queries are useful for doing searches of hundreds, thousands, or even millions of records. You just hand the search options to your server, and it is responsible for handing you back the list of records that match. Because the response from the server includes the ID of all of the records that matched, it doesn't matter if the store hadn't loaded them previously; it sees that they are not in the cache and can request the records by ID if necessary.

The downside of queries is that they do not live update, they are slower, and they require that your server support the kind of queries that you wish to perform.

Because the server decides which records match the query, not the store, queries do not live update. If you want to update them, you must manually call `reload()` and wait for the server to respond. If you create a new record on the client, it will not show up in the results until you both save the new record to the server and reload the query results.

Because the store must confer with your server to determine the results of a query, it necessitates a network request. This can feel slow to users, especially if they are on a slow connection or your server is slow to respond. The typical speed of JavaScript web applications can heighten the perceived slowness when the server must be consulted.

Lastly, performing queries requires collaboration between the store and your server. By default, Ember Data will send the search options that you pass as the body of an HTTP request to your server. If your server does not support requests in this format, you will need to either change your server to do so, or customize how queries are sent by creating a custom adapter.

Filters, on the other hand, perform a live search of all of the records in the store's cache. As soon as a new record is loaded into the store, the filter will check to see if the record matches, and if so, add it to the array of search results. If that array is displayed in a template, it will update automatically.

Filters also take into account newly created records that have not been saved, and records that have been modified but not yet saved. If you want records to show up in search results as soon as they are created or modified on the client, you should use a filter.

Keep in mind that records will not show up in a filter if the store doesn't know about them. You can ensure that a record is in the store by using the store's `push()` method.

There is also a limit to how many records you can reasonably keep in memory and search before you start hitting performance issues.

Finally, keep in mind that you can combine queries and filters to take advantage of their respective strengths and weaknesses. Remember that records returned by a query to the server are cached in the store. You can use this fact to perform a filter, passing it a query that starts matching records into the store, and a filter function that matches the same records.

This will offload searching all of the possible records to the server, while still creating a live updating list that includes records created and modified on the client.

```
App.PostsFavoritedRoute = Ember.Route.extend({
  model: function() {
    var store = this.store;

    // Create a filter for all favorited posts that will be displayed in
    // the template. Any favorited posts that are already in the store
    // will be displayed immediately;
    // Kick off a query to the server for all posts that
    // the user has favorited. As results from the query are
    // returned from the server, they will also begin to appear.
    return store.filter('post', { favorited: true }, function(post) {
      return post.get('isFavorited');
    });
  }
});
```

How do I inform Ember Data about new records created on the backend?

When you request a record using Ember Data's `store.find` method, Ember will automatically load the data into the store. This allows Ember to avoid the latency of making a round trip to the backend next time that record is requested. Additionally, loading a record into the store will update any `RecordArrays` (e.g. the result of `store.filter` or `store.all`) that should include that record. This means any data bindings or computed properties that depend on the `RecordArray` will automatically be synced to include the new or updated record values.

Some applications may want to add or update records in the store without requesting the record via `store.find`. To accomplish this you can use the `DS.Store`'s `push`, `pushPayload`, or `update` methods. This is useful for web applications that have a channel (such as SSE[239] or Web Sockets[240]) to notify it of new or updated records on the backend.

push[241] is the simplest way to load records to Ember Data's store. When using `push` it is important to remember to deserialize the JSON object before pushing it into the store. `push` only accepts one record at a time. If you would like to load an array of records to the store you can call pushMany[242].

[239] http://dev.w3.org/html5/eventsource/

[240] http://www.w3.org/TR/2009/WD-websockets-20091222/

[241] http://emberjs.com/api/data/classes/DS.Store.html#method_push

[242] http://emberjs.com/api/data/classes/DS.Store.html#method_pushMany

```
socket.on('message', function (message) {
  var type = store.modelFor(message.model);
  var serializer = store.serializerFor(type.typeKey);
  var record = serializer.extractSingle(store, type, message.data);
  store.push(message.model, record);
});
```

pushPayload[243] is a convenience wrapper for `store#push` that will deserialize payloads if the model's Serializer implements a `pushPayload` method. It is important to note this method will not work with the `JSONSerializer` because it does not implement a `pushPayload` method.

```
socket.on('message', function (message) {
  store.pushPayload(message.model, message.data);
});
```

update[244] works like a `push` except it can handle partial attributes without overwriting the existing record properties. This method is useful if your web application only receives notifications of the changed attributes on a model. Like `push` it is important to remember to deserialize the JSON object before calling `update`.

```
socket.on('message', function (message) {
  var hash = message.data;
  var type = store.modelFor(message.model);
  var fields = Ember.get(type, 'fields');
  fields.forEach(function(field) {
    var payloadField = Ember.String.underscore(field);
    if (field === payloadField) { return; }
      hash[field] = hash[payloadField];
      delete hash[payloadField];
  });
  store.push(message.model, hash);
});
```

[243] http://emberjs.com/api/data/classes/DS.Store.html#method_pushPayload
[244] http://emberjs.com/api/data/classes/DS.Store.html#method_update

Views

Introduction

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:

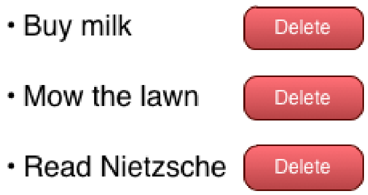
- When you need sophisticated handling of user events
- When you want to create a re-usable component

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.

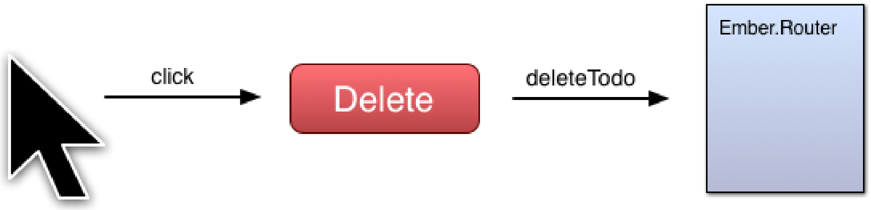
For example, imagine you have a list of todo items. Next to each todo is a button to delete that item:



- Buy milk
- Mow the lawn
- Read Nietzsche

Todo List

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, or if no method is defined there, your application's router, which is responsible for reacting to the event based on the current state of the application.



Todo List

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:

```
<html>
  <head>
    <script type="text/x-handlebars" data-template-name="say-hello">
      Hello, <b>{{view.name}}</b>
    </script>
  </head>
</html>
```

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

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

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.

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

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:

```
{{#view "clickable"}}
This is a clickable area!
{{/view}}
```

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

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

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:

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

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


````javascript
App.PlaybackController = Ember.ObjectController.extend({
  actions: {
    turnItUp: function(level){
      //Do your thing
    }
  }
});
````

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

````javascript
App.PlaybackRoute = Ember.Route.extend({
  actions: {
    turnItUp: function(level){
      //This won't be called if it's defined on App.PlaybackController
    }
  }
});
````

To see a full listing of the `Ember.View` built-in events, see the
documentation section on [Event Names](http://emberjs.com/api/classes/Ember.View\
.html#toc_event-names).

## Adding Layouts to Views

Views can have a secondary template that wraps their main template. Like templat\
es,
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.

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

First, you define the following layout template:

```html
<script type="text/x-handlebars" data-template-name="my_layout">
  <div class="content-wrapper">
    {{yield}}
  </div>
</script>
~~~

And then the following main template:

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

Finally, you define a view, and instruct it to wrap the template with the defined layout:

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

This will result in view instances containing the following HTML

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

```
<script type="text/x-handlebars" data-template-name="popup">
  <div class="popup">
    <button class="popup-dismiss">x</button>
    <div class="popup-content">
    {{yield}}
    </div>
  </div>
</script>
```

Then define your popup view:

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

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

```
{{#view "popup"}}
  <form>
    <label for="name">Name:</label>
    <input id="name" type="text" />
  </form>
{{/view}}

{{#view "popup"}}
  <p> Thank you for signing up! </p>
{{/view}}
```

Customizing A Views Element

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.

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

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

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

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:

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

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.

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

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

This would render this HTML:

```
<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`:

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

This would render this HTML:

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

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

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

This would render this HTML:

```
<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:

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

This would render this HTML:

```
<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`:

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

You can also bind these attributes to differently named properties:

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

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.

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

```
{{view "info" tagName="span"}}
```

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

```
{{view "info" id="info-view"}}
```

This makes it easy to style using CSS ID selectors:

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

You can assign class names similarly:

```
{{view "info" class="info urgent"}}
```

You can bind class names to a property of the view by using `classBinding` instead of `class`. The same behavior as described in `bind-attr` applies:

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

```
{{view "alert" 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

Ember comes pre-packaged with a set of views for building a basic controls like text inputs, check boxes, and select lists. Usually, these views will be used via the input helpers[245]. However, the base views may be helpful in creating custom form behaviors.

- Ember.Checkbox[246]
- Ember.TextField[247]
- Ember.TextArea[248]

For example, here we have created a custom text field that toggles a dirty property:

```
// {{view "myText" value=name inputDidChange=nameDidChange}}
App.MyText = Ember.TextField.extend({
  inputDidChange: false,
  change: function() {
    this.set('inputDidChange', true);
  }
});
```

Ember itself provides one additional view not covered by the input helpers, and this is the select box view.

- Ember.Select[249]

This class can also be customized by extending it. To use the select view bundled with Ember, call it via the view helper:

[245] http://emberjs.com/guides/templates/input-helpers/
[246] http://emberjs.com/api/classes/Ember.Checkbox.html
[247] http://emberjs.com/api/classes/Ember.TextField.html
[248] http://emberjs.com/api/classes/Ember.TextArea.html
[249] http://emberjs.com/api/classes/Ember.Select.html

```
{{view Ember.Select content=people
                    optionLabelPath="content.fullName"
                    optionValuePath="content.id"
                    prompt="Pick a person:"
                    selection=selectedPerson}}
```

The select view is extremely feature-rich, and may perform badly when rendering many items. Due to this, it has not yet been converted into an component or helper like other inputs.

Manually Managing View Hierachy

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`[250] is the way to do just that.

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

```
var container = Ember.ContainerView.create();
container.append();

var firstView = App.FirstView.create(),
    secondView = App.SecondView.create();

container.pushObject(firstView);
container.pushObject(secondView);

// When the rendering completes, the DOM
// will contain a `div` for the ContainerView
// 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`:

[250] /api/classes/Ember.ContainerView.html

```
var container = Ember.ContainerView.create({
  init: function() {
    this._super();
    this.pushObject(App.FirstView.create());
    this.pushObject(App.SecondView.create());
  }
});

container.objectAt(0).toString(); //=> '<App.FirstView:ember123>'
container.objectAt(1).toString(); //=> '<App.SecondView:ember124>'
```

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:

```
var container = Ember.ContainerView.extend({
  childViews: [App.FirstView, App.SecondView]
});

container.objectAt(0).toString(); //=> '<App.FirstView:ember123>'
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:

```
var container = Ember.ContainerView.create({
  childViews: ['firstView', 'secondView'],
  firstView: App.FirstView,
  secondView: App.SecondView
});

container.objectAt(0).toString(); //=> '<App.FirstView:ember123>'
container.objectAt(1).toString(); //=> '<App.SecondView:ember124>'

container.get('firstView').toString(); //=> '<App.FirstView:ember123>'
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`[251] gains its collection-like behavior by

[251] /api/classes/Ember.ContainerView.html

mixing in `Ember.MutableArray`[252]. 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.

[252] /api/classes/Ember.MutableArray.html

Enumerables

Enumerables

In Ember.js, an Enumerable is any object that contains a number of child objects, and which allows you to work with those children using the Ember.Enumerable[253] API. The most common Enumerable in the majority of apps is the native JavaScript array, which Ember.js extends to conform to the Enumerable interface.

By providing a standardized interface for dealing with enumerables, Ember.js allows you to completely change the way your underlying data is stored without having to modify the other parts of your application that access it.

For example, you might display a list of items from fixture data during development. If you switch the underlying data from synchronous fixtures to an array that fetches data from the server lazily, your view, template and controller code do not change at all.

The Enumerable API follows ECMAScript specifications as much as possible. This minimizes incompatibility with other libraries, and allows Ember.js to use the native browser implementations in arrays where available.

For instance, all Enumerables support the standard `forEach` method:

```
[1,2,3].forEach(function(item) {
  console.log(item);
});

//=> 1
//=> 2
//=> 3
```

In general, Enumerable methods, like `forEach`, take an optional second parameter, which will become the value of `this` in the callback function:

[253] http://emberjs.com/api/classes/Ember.Enumerable.html

```
var array = [1,2,3];

array.forEach(function(item) {
  console.log(item, this.indexOf(item));
}, array)

//=> 1 0
//=> 2 1
//=> 3 2
```

Enumerables in Ember.js

Usually, objects that represent lists implement the Enumerable interface. Some examples:

- **Array** - Ember extends the native JavaScript `Array` with the Enumerable interface (unless you disable prototype extensions.[254])
- **Ember.ArrayController** - A controller that wraps an underlying array and adds additional functionality for the view layer.
- **Ember.Set** - A data structure that can efficiently answer whether it includes an object.

API Overview

In this guide, we'll explore some of the most common Enumerable conveniences. For the full list, please see the Ember.Enumerable API reference documentation.[255]

Iterating Over an Enumerable

To enumerate all the values of an enumerable object, use the `forEach` method:

```
var food = ["Poi", "Ono", "Adobo Chicken"];

food.forEach(function(item, index) {
  console.log('Menu Item %@: %@'.fmt(index+1, item));
});

// Menu Item 1: Poi
// Menu Item 2: Ono
// Menu Item 3: Adobo Chicken
```

Making an Array Copy

You can make a native array copy of any object that implements `Ember.Enumerable` by calling the `toArray()` method:

[254] http://emberjs.com/guides/configuring-ember/disabling-prototype-extensions/

[255] http://emberjs.com/api/classes/Ember.Enumerable.html

```
var states = Ember.Set.create();

states.add("Hawaii");
states.add("California")

states.toArray()
//=> ["Hawaii", "California"]
```

Note that in many enumerables, such as the `Ember.Set` used in this example, the order of the resulting array is not guaranteed.

First and Last Objects

All Enumerables expose `firstObject` and `lastObject` properties that you can bind to.

```
var animals = ["rooster", "pig"];

animals.get('lastObject');
//=> "pig"

animals.pushObject("peacock");

animals.get('lastObject');
//=> "peacock"
```

Map

You can easily transform each item in an enumerable using the `map()` method, which creates a new array with results of calling a function on each item in the enumerable.

```
var words = ["goodbye", "cruel", "world"];

var emphaticWords = words.map(function(item) {
  return item + "!";
});
// ["goodbye!", "cruel!", "world!"]
```

If your enumerable is composed of objects, there is a `mapBy()` method that will extract the named property from each of those objects in turn and return a new array:

```
var hawaii = Ember.Object.create({
  capital: "Honolulu"
});

var california = Ember.Object.create({
  capital: "Sacramento"
});

var states = [hawaii, california];

states.mapBy('capital');
//=> ["Honolulu", "Sacramento"]
```

Filtering

Another common task to perform on an Enumerable is to take the Enumerable as input, and return an Array after filtering it based on some criteria.

For arbitrary filtering, use the `filter` method. The filter method expects the callback to return `true` if Ember should include it in the final Array, and `false` or `undefined` if Ember should not.

```
var arr = [1,2,3,4,5];

arr.filter(function(item, index, self) {
  if (item < 4) { return true; }
})

// returns [1,2,3]
```

When working with a collection of Ember objects, you will often want to filter a set of objects based upon the value of some property. The `filterBy` method provides a shortcut.

```
Todo = Ember.Object.extend({
  title: null,
  isDone: false
});

todos = [
  Todo.create({ title: 'Write code', isDone: true }),
  Todo.create({ title: 'Go to sleep' })
];

```

```
todos.filterBy('isDone', true);

// returns an Array containing only items with `isDone == true`
```

If you want to return just the first matched value, rather than an Array containing all of the matched values, you can use `find` and `findBy`, which work just like `filter` and `filterBy`, but return only one item.

Aggregate Information (All or Any)

If you want to find out whether every item in an Enumerable matches some condition, you can use the `every` method:

```
Person = Ember.Object.extend({
  name: null,
  isHappy: false
});

var people = [
  Person.create({ name: 'Yehuda', isHappy: true }),
  Person.create({ name: 'Majd', isHappy: false })
];

people.every(function(person, index, self) {
  if(person.get('isHappy')) { return true; }
});

// returns false
```

If you want to find out whether at least one item in an Enumerable matches some conditions, you can use the `some` method:

```
people.some(function(person, index, self) {
  if(person.get('isHappy')) { return true; }
});

// returns true
```

Just like the filtering methods, the `every` and `some` methods have analogous `isEvery` and `isAny` methods.

```
people.isEvery('isHappy', true) // false
people.isAny('isHappy', true)  // true
```

Testing

Introduction

Testing is a core part of the Ember framework and its development cycle.

Let’s assume you are writing an Ember application which will serve as a blog. This application would likely include models such as `user` and `post`. It would also include interactions such as *login* and *create post*. Let’s finally assume that you would like to have automated tests[256] in place for your application.

There are two different classifications of tests that you will need: **Integration** and **Unit**.

Integration Tests

Integration tests are used to test user interaction and application flow. With the example scenario above, some integration tests you might write are:

- A user is able to log in via the login form.
- A user is able to create a blog post.
- A visitor does not have access to the admin panel.

Unit Tests

Unit tests are used to test isolated chunks of functionality, or “units” without worrying about their dependencies. Some examples of unit tests for the scenario above might be:

- A user has a role
- A user has a username
- A user has a fullname attribute which is the aggregate of its first and last names with a space between
- A post has a title
- A post’s title must be no longer than 50 characters

[256] http://en.wikipedia.org/wiki/Test_automation

Testing Frameworks

QUnit[257] is the default testing framework for this guide, but others are supported through third-party adapters.

Contributing

The Ember testing guide provides best practices and examples on how to test your Ember applications. If you find any errors or believe the documentation can be improved, please feel free to contribute[258].

Integration Test

Integration tests are generally used to test important workflows within your application. They emulate user interaction and confirm expected results.

Setup

In order to integration test the Ember application, you need to run the app within your test framework. Set the root element of the application to an arbitrary element you know will exist. It is useful, as an aid to test-driven development, if the root element is visible while the tests run. You can potentially use #qunit-fixture, which is typically used to contain fixture html for use in tests, but you will need to override css to make it visible.

```
App.rootElement = '#arbitrary-element-to-contain-ember-application';
```

This hook defers the readiness of the application, so that you can start the app when your tests are ready to run. It also sets the router's location to 'none', so that the window's location will not be modified (preventing both accidental leaking of state between tests and interference with your testing framework).

```
App.setupForTesting();
```

This injects the test helpers into the window's scope.

```
App.injectTestHelpers();
```

With QUnit, `setup` and `teardown` functions can be defined in each test module's configuration. These functions are called for each test in the module. If you are using a framework other than QUnit, use the hook that is called before each individual test.

After each test, reset the application: `App.reset()` completely resets the state of the application.

[257] http://qunitjs.com/

[258] https://github.com/emberjs/website

```
module('Integration Tests', {
  teardown: function() {
    App.reset();
  }
});
```

Test adapters for other libraries

If you use a library other than QUnit, your test adapter will need to provide methods for `asyncStart` and `asyncEnd`. To facilitate asynchronous testing, the default test adapter for QUnit uses methods that QUnit provides: (globals) `stop()` and `start()`.

Please note:

The `ember-testing` package is not included in the production builds, only development builds of Ember include the testing package. The package can be loaded in your dev or qa builds to facilitate testing your application. By not including the ember-testing package in production, your tests will not be executable in a production environment.

Test Helpers

One of the major issues in testing web applications is that all code is event-driven, therefore has the potential to be asynchronous (ie output can happen out of sequence from input). This has the ramification that code can be executed in any order.

An example may help here: Let's say a user clicks two buttons, one after another and both load data from different servers. They take different times to respond.

When writing your tests, you need to be keenly aware of the fact that you cannot be sure that the response will return immediately after you make your requests, therefore your assertion code (the "tester") needs to wait for the thing being tested (the "testee") to be in a synchronized state. In the example above, that would be when both servers have responded and the test code can go about its business checking the data (whether it is mock data, or real data).

This is why all Ember's test helpers are wrapped in code that ensures Ember is back in a synchronized state when it makes its assertions. It saves you from having to wrap everything in code that does that, and it makes it easier to read your tests because there's less boilerplate in them.

Ember includes several helpers to facilitate integration testing. There are two types of helpers: **asynchronous** and **synchronous**.

Asynchronous Helpers

Asynchronous helpers are "aware" of (and wait for) asynchronous behavior within your application, making it much easier to write deterministic tests.

Also, these helpers register themselves in the order that you call them and will be run in a chain; each one is only called after the previous one finishes, in a chain. You can rest assured, therefore, that the order you call them in will also be their execution order, and that the previous helper has finished before the next one starts.

- `visit(url)`
  - Visits the given route and returns a promise that fulfills when all resulting async behavior is complete.
- `fillIn(selector, text)`
  - Fills in the selected input with the given text and returns a promise that fulfills when all resulting async behavior is complete.
- `click(selector)`
  - Clicks an element and triggers any actions triggered by the element's `click` event and returns a promise that fulfills when all resulting async behavior is complete.
- `keyEvent(selector, type, keyCode)`
  - Simulates a key event type, e.g. `keypress`, `keydown`, `keyup` with the desired keyCode on element found by the selector.
- `triggerEvent(selector, type, options)`
  - Triggers the given event, e.g. `blur`, `dblclick` on the element identified by the provided selector.

Synchronous Helpers

Synchronous helpers are performed immediately when triggered.

- `find(selector, context)`
  - Finds an element within the app's root element and within the context (optional). Scoping to the root element is especially useful to avoid conflicts with the test framework's reporter, and this is done by default if the context is not specified.
- `currentPath()`
  - Returns the current path.
- `currentRouteName()`
  - Returns the currently active route name.
- `currentURL()`
  - Returns the current URL.

Wait Helpers

The `andThen` helper will wait for all preceding asynchronous helpers to complete prior to progressing forward. Let's take a look at the following example.

```
test('simple test', function() {
  expect(1); // Ensure that we will perform one assertion

  visit('/posts/new');
  fillIn('input.title', 'My new post');
  click('button.submit');

  // Wait for asynchronous helpers above to complete
  andThen(function() {
    equal(find('ul.posts li:last').text(), 'My new post');
  });
});
```

First we tell qunit that this test should have one assertion made by the end of the test by calling `expect` with an argument of `1`. We then visit the new posts URL "/posts/new", enter the text "My new post" into an input control with the CSS class "title", and click on a button whose class is "submit".

We then make a call to the `andThen` helper which will wait for the preceding asynchronous test helpers to complete (specifically, `andThen` will only be called **after** the new posts URL was visited, the text filled in and the submit button was clicked, **and** the browser has returned from doing whatever those actions required). Note `andThen` has a single argument of the function that contains the code to execute after the other test helpers have finished.

In the `andThen` helper, we finally make our call to equal which makes an assertion that the text found in the last li of the ul whose class is "posts" is equal to "My new post".

Custom Test Helpers

`Ember.Test.registerHelper` and `Ember.Test.registerAsyncHelper` are used to register test helpers that will be injected when `App.injectTestHelpers` is called. The difference between `Ember.Test.registerHelper` and `Ember.Test.registerAsyncHelper` is that the latter will not run until any previous async helper has completed and any subsequent async helper will wait for it to finish before running.

The helper method will always be called with the current Application as the first parameter. Helpers need to be registered prior to calling `App.injectTestHelpers()`.

Here is an example of a non-async helper:

```
Ember.Test.registerHelper('shouldHaveElementWithCount',
  function(app, selector, n, context) {
    var el = findWithAssert(selector, context);
    var count = el.length;
    equal(n, count, 'found ' + count + ' times');
  }
);

// shouldHaveElementWithCount("ul li", 3);
```

Here is an example of an async helper:

```
Ember.Test.registerAsyncHelper('dblclick',
  function(app, selector, context) {
    var $el = findWithAssert(selector, context);
    Ember.run(function() {
      $el.dblclick();
    });
  }
);

// dblclick("#person-1")
```

Async helpers also come in handy when you want to group interaction into one helper. For example:

```
Ember.Test.registerAsyncHelper('addContact',
  function(app, name, context) {
    fillIn('#name', name);
    click('button.create');
  }
);

// addContact("Bob");
// addContact("Dan");
```

Example

Here is an example using both `registerHelper` and `registerAsyncHelper`.

Custom Test Helpers[259]

[259] http://jsbin.com/jesuyeri

Testing User Interaction

Almost every test has a pattern of visiting a route, interacting with the page (using the helpers), and checking for expected changes in the DOM.

Examples:

```
test('root lists first page of posts', function(){
  visit('/posts');
  andThen(function() {
    equal(find('ul.posts li').length, 3, 'The first page should have 3 posts');
  });
});
```

The helpers that perform actions use a global promise object and automatically chain onto that promise object if it exists. This allows you to write your tests without worrying about async behaviour your helper might trigger.

```
module('Integration: Transitions', {
  teardown: function() {
    App.reset();
  }
});

test('add new post', function() {
  visit('/posts/new');
  fillIn('input.title', 'My new post');
  click('button.submit');

  andThen(function() {
    equal(find('ul.posts li:last').text(), 'My new post');
  });
});
```

Live Example

<a class=”jsbin-embed” href=”http://jsbin.com/gokor/embed?output”>Testing User Interaction</a>

Testing Transitions

Suppose we have an application which requires authentication. When a visitor visits a certain URL as an unauthenticated user, we expect them to be transitioned to a login page.

```
App.ProfileRoute = Ember.Route.extend({
  beforeModel: function() {
    var user = this.modelFor('application');
    if (Em.isEmpty(user)) {
      this.transitionTo('login');
    }
  }
});
```

We could use the route helpers to ensure that the user would be redirected to the login page when the restricted URL is visited.

```
module('Integration: Transitions', {
  teardown: function() {
    App.reset();
  }
});

test('redirect to login if not authenticated', function() {
  visit('/');
  click('.profile');

  andThen(function() {
    equal(currentRouteName(), 'login');
    equal(currentPath(), 'login');
    equal(currentURL(), '/login');
  });
});
```

Live Example

Testing Transitions[260]

Unit Testing Basics

Unit tests are generally used to test a small piece of code and ensure that it is doing what was intended. Unlike integration tests, they are narrow in scope and do not require the Ember application to be running.

As it is the basic object type in Ember, being able to test a simple `Ember.Object` sets the foundation for testing more specific parts of your Ember application such as controllers, components, etc. Testing an `Ember.Object` is as simple as creating an instance of the object, setting its state, and running assertions against the object. By way of example lets look at a few common cases.

[260] http://jsbin.com/nulif

Testing Computed Properties

Let's start by looking at an object that has a `computedFoo` computed property based on a `foo` property.

```
App.SomeThing = Ember.Object.extend({
  foo: 'bar',
  computedFoo: function(){
    return 'computed ' + this.get('foo');
  }.property('foo')
});
```

Within the test we'll create an instance, update the `foo` property (which should trigger the computed property), and assert that the logic in our computed property is working correctly.

```
module('Unit: SomeThing');

test('computedFoo correctly concats foo', function() {
  var someThing = App.SomeThing.create();
  someThing.set('foo', 'baz');
  equal(someThing.get('computedFoo'), 'computed baz');
});
```

Live Example

Unit Testing Basics: Computed Properties[261]

Testing Object Methods

Next let's look at testing logic found within an object's method. In this case the `testMethod` method alters some internal state of the object (by updating the `foo` property).

```
App.SomeThing = Ember.Object.extend({
  foo: 'bar',
  testMethod: function() {
    this.set('foo', 'baz');
  }
});
```

To test it, we create an instance of our class `SomeThing` as defined above, call the `testMethod` method and assert that the internal state is correct as a result of the method call.

[261] http://jsbin.com/miziz

```
module('Unit: SomeThing');

test('calling testMethod updates foo', function() {
  var someThing = App.SomeThing.create();
  someThing.testMethod();
  equal(someThing.get('foo'), 'baz');
});
```

Live Example

Unit Testing Basics: Method Side Effects[262]

In the event the object's method returns a value you can simply assert that the return value is calculated correctly. Suppose our object has a `calc` method that returns a value based on some internal state.

```
App.SomeThing = Ember.Object.extend({
  count: 0,
  calc: function() {
    this.incrementProperty('count');
    return 'count: ' + this.get('count');
  }
});
```

The test would call the `calc` method and assert it gets back the correct value.

```
module('Unit: SomeThing');

test('testMethod returns incremented count', function() {
  var someThing = App.SomeThing.create();
  equal(someThing.calc(), 'count: 1');
  equal(someThing.calc(), 'count: 2');
});
```

Live Example

Unit Testing Basics: Method Side Effects[263]

Testing Observers

Suppose we have an object that has an observable method based on the `foo` property.

[262] http://jsbin.com/weroh
[263] http://jsbin.com/qutar

```
App.SomeThing = Ember.Object.extend({
  foo: 'bar',
  other: 'no',
  doSomething: function(){
    this.set('other', 'yes');
  }.observes('foo')
});
```

In order to test the `doSomething` method we create an instance of `SomeThing`, update the observed property (`foo`), and assert that the expected effects are present.

```
module('Unit: SomeThing');

test('doSomething observer sets other prop', function() {
  var someThing = App.SomeThing.create();
  someThing.set('foo', 'baz');
  equal(someThing.get('other'), 'yes');
});
```

Live Example

Unit Testing Basics: Observers[264]

Unit Test Helpers

Globals vs Modules

In the past, it has been difficult to test portions of your Ember application without loading the entire app as a global. By having your application written using modules (CommonJS[265], AMD[266], etc), you are able to require just code that is to be tested without having to pluck the pieces out of your global application.

Unit Testing Helpers

Ember-QUnit[267] is the default *unit* testing helper suite for Ember. It can and should be used as a template for other test framework helpers. It uses your application's resolver to find and automatically create test subjects for you using the `moduleFor` and `test` helpers.

[264] http://jsbin.com/daxok

[265] http://wiki.commonjs.org/wiki/CommonJS

[266] http://requirejs.org/docs/whyamd.html

[267] https://github.com/rpflorence/ember-qunit

A test subject is simply an instance of the object that a particular test is making assertions about. Usually test subjects are manually created by the writer of the test.

The unit testing section of this guide will use the Ember-QUnit library, but the concepts and examples should translate easily to other frameworks.

Available Helpers

By including Ember-QUnit[268], you will have access to a number of test helpers.

- `moduleFor(fullName [, description [, callbacks]])`
- **fullName**: The full name of the unit, (ie. `controller:application`, `route:index`, etc.)
- **description**: the description of the module
- **callbacks**: normal QUnit callbacks (setup and teardown), with addition to needs, which allows you specify the other units the tests will need.
- `moduleForComponent(name [, description [, callbacks]])`
- **name**: the short name of the component that you'd use in a template, (ie. `x-foo`, `ic-tabs`, etc.)
- **description**: the description of the module
- **callbacks**: normal QUnit callbacks (setup and teardown), with addition to needs, which allows you specify the other units the tests will need.
- `moduleForModel(name [, description [, callbacks]])`
- **name**: the short name of the model you'd use in store operations (ie. `user`, `assignmentGroup`, etc.)
- **description**: the description of the module
- **callbacks**: normal QUnit callbacks (setup and teardown), with addition to needs, which allows you specify the other units the tests will need.
- `test`
- Same as QUnit `test` except it includes the `subject` function which is used to create the test subject.
- `setResolver`
- Sets the resolver which will be used to lookup objects from the application container.

Unit Testing Setup

In order to unit test your Ember application, you need to let Ember know it is in test mode. To do so, you must call `Ember.setupForTesting()`.

[268] https://github.com/rpflorence/ember-qunit

```
Ember.setupForTesting();
```

The `setupForTesting()` function call makes ember turn off its automatic run loop execution. This gives us an ability to control the flow of the run loop ourselves, to a degree. Its default behaviour of resolving all promises and completing all async behaviour are suspended to give you a chance to set up state and make assertions in a known state. In other words, you know that if you run "visit" to get to a particular URL, you can be sure the URL has been visited and that's the only behaviour that has transpired. If we didn't use this mode, our assertions would most likely be executed before the async behaviour had taken place, so our assertion results would be unpredictable.

With a module-based application, you have access to the unit test helpers simply by requiring the exports of the module. However, if you are testing a global Ember application, you are still able to use the unit test helpers. Instead of importing the `ember-qunit` module, you need to make the unit test helpers global with `emq.globalize()`:

```
emq.globalize();
```

This will make the above helpers available globally.

The Resolver

The Ember resolver plays a huge role when unit testing your application. It provides the lookup functionality based on name, such as `route:index` or `model:post`.

If you do not have a custom resolver or are testing a global Ember application, the resolver should be set like this:

Make sure to replace "App" with your application's namespace in the following line

```
setResolver(Ember.DefaultResolver.create({ namespace: App }))
```

Otherwise, you would require the custom resolver and pass it to `setResolver` like this *(ES6 example)*:

```
import Resolver from './path/to/resolver';
import { setResolver } from 'ember-qunit';
setResolver(Resolver.create());
```

Unit Test Components

Unit testing methods and computed properties follows previous patterns shown in Unit Testing Basics[269] because Ember.Component extends Ember.Object.

Setup

Before testing components, be sure to add testing application div to your testing html file:

[269] http://emberjs.com/guides/testing/unit-testing-basics

```
<!-- as of time writing, ID attribute needs to be named exactly ember-testing -->
<div id="ember-testing"></div>
```

and then you'll also need to tell Ember to use this element for rendering the application in

```
App.rootElement = '#ember-testing'
```

Components can be tested using the `moduleForComponent` helper. Here is a simple Ember component:

```
App.PrettyColorComponent = Ember.Component.extend({
  classNames: ['pretty-color'],
  attributeBindings: ['style'],
  style: function() {
    return 'color: ' + this.get('name') + ';';
  }.property('name')
});
```

with an accompanying Handlebars template:

```
Pretty Color: {{name}}
```

Unit testing this component can be done using the `moduleForComponent` helper. This helper will find the component by name (pretty-color) and it's template (if available).

```
moduleForComponent('pretty-color');
```

Now each of our tests has a function `subject()` which aliases the create method on the component factory.

Here's how we would test to make sure rendered HTML changes when changing the color on the component:

```
test('changing colors', function(){

  // this.subject() is available because we used moduleForComponent
  var component = this.subject();

  // we wrap this with Ember.run because it is an async function
  Ember.run(function(){
    component.set('name','red');
  });

  // first call to $() renders the component.
  equal(this.$().attr('style'), 'color: red;');

  // another async function, so we need to wrap it with Ember.run
  Ember.run(function(){
    component.set('name', 'green');
  });

  equal(this.$().attr('style'), 'color: green;');
});
```

Another test that we might perform on this component would be to ensure the template is being rendered properly.

```
test('template is rendered with the color name', function(){

  // this.subject() is available because we used moduleForComponent
  var component = this.subject();

  // first call to $() renders the component.
  equal($.trim(this.$().text()), 'Pretty Color:');

  // we wrap this with Ember.run because it is an async function
  Ember.run(function(){
    component.set('name', 'green');
  });

  equal($.trim(this.$().text()), 'Pretty Color: green');
});
```

Live Example

Unit Testing Components[270]

Interacting with Components in the DOM

Ember Components are a great way to create powerful, interactive, self-contained custom HTML elements. Because of this, it is important to not only test the methods on the component itself, but also the user's interaction with the component.

Let's look at a very simple component which does nothing more than set it's own title when clicked:

```
App.MyFooComponent = Em.Component.extend({
  title:'Hello World',

  actions:{
    updateTitle: function(){
      this.set('title', 'Hello Ember World');
    }
  }
});
```

We would use Integration Test Helpers[271] to interact with the rendered component:

```
moduleForComponent('my-foo', 'MyFooComponent');

test('clicking link updates the title', function() {
  var component = this.subject();

  // append the component to the DOM
  this.append();

  // assert default state
  equal(find('h2').text(), 'Hello World');

  // perform click action
  click('button');

  andThen(function() { // wait for async helpers to complete
    equal(find('h2').text(), 'Hello Ember World');
  });
});
```

[270] http://jsbin.com/hihef

[271] http://emberjs.com/guides/testing/test-helpers

Live Example

Unit Testing Components[272]

Components with built in layout

Some components do not use a separate template. The template can be embedded into the component via the layout[273] property. For example:

```
App.MyFooComponent = Ember.Component.extend({

  // layout supercedes template when rendered
  layout: Ember.Handlebars.compile(
    "<h2>I'm a little {{noun}}</h2><br/>" +
    "<button {{action 'changeName'}}>Click Me</button>"
  ),

  noun: 'teapot',

  actions:{
    changeName: function(){
      this.set('noun', 'embereño');
    }
  }
});
```

In this example, we would still perform our test by interacting with the DOM.

```
moduleForComponent('my-foo', 'MyFooComponent');

test('clicking link updates the title', function() {
  var component = this.subject();

  // append the component to the DOM
  this.append();

  // assert default state
  equal(find('h2').text(), "I'm a little teapot");

```

[272] http://jsbin.com/liqog

[273] http://emberjs.com/api/classes/Ember.Component.html#property_layout

```
  // perform click action
  click('button');

  andThen(function() { // wait for async helpers to complete
    equal(find('h2').text(), "I'm a little embereño");
  });
});
```

Live Example

Testing Components with Built-in Layout[274]

Programmatically interacting with components

Another way we can test our components is to perform function calls directly on the component instead of through DOM interaction. Let's use the same code example we have above as our component, but perform the tests programatically:

```
moduleForComponent('my-foo', 'MyFooComponent');

test('clicking link updates the title', function() {
  var component = this.subject();

  // append the component to the DOM, returns DOM instance
  var $component = this.append();

  // assert default state
  equal($component.find('h2').text(), "I'm a little teapot");

  // send action programmatically
  Ember.run(function(){
    component.send('changeName');
  });

  equal($component.find('h2').text(), "I'm a little embereño");
});
```

Live Example

Programatically Testing Components[275]

[274] http://jsbin.com/mazef

[275] http://jsbin.com/davuf

`sendAction` validation in components

Components often utilize `sendAction`, which is a way to interact with the Ember application. Here's a simple component which sends the action `internalAction` when a button is clicked:

```
App.MyFooComponent = Ember.Component.extend({
  layout:Ember.Handlebars.compile("<button {{action 'doSomething'}}></button>"),

  actions:{
    doSomething: function(){
      this.sendAction('internalAction');
    }
  }
});
```

In our test, we will create a dummy object that receives the action being sent by the component.

```
moduleForComponent('my-foo', 'MyFooComponent');

test('trigger external action when button is clicked', function() {
  // tell our test to expect 1 assertion
  expect(1);

  // component instance
  var component = this.subject();

  // component dom instance
  var $component = this.append();

  var targetObject = {
    externalAction: function(){
      // we have the assertion here which will be
      // called when the action is triggered
      ok(true, 'external Action was called!');
    }
  };

  // setup a fake external action to be called when
  // button is clicked
  component.set('internalAction', 'externalAction');

  // set the targetObject to our dummy object (this
  // is where sendAction will send it's action to)
  component.set('targetObject', targetObject);

  // click the button
  click('button');
});
```

Live Example

sendAction Validation in Components[276]

Components Using Other Components

Sometimes components are easier to maintain when broken up into parent and child components. Here is a simple example:

```
App.MyAlbumComponent = Ember.Component.extend({
  tagName: 'section',
  layout: Ember.Handlebars.compile(
      "<section>" +
      "  <h3>{{title}}</h3>" +
      "  {{yield}}" +
      "</section>"
  ),
  titleBinding: ['title']
});

App.MyKittenComponent = Ember.Component.extend({
  tagName: 'img',
  attributeBindings: ['width', 'height', 'src'],
  src: function() {
    return 'http://placekitten.com/' + this.get('width') + '/' + this.get('heigh\
t');
  }.property('width', 'height')
});
```

Usage of this component might look something like this:

[276] http://jsbin.com/siwil

```
{{#my-album title="Cats"}}
  {{my-kitten width="200" height="300"}}
  {{my-kitten width="100" height="100"}}
  {{my-kitten width="50" height="50"}}
{{/my-album}}
```

Testing components like these which include child components is very simple using the `needs` callback.

```
moduleForComponent('my-album', 'MyAlbumComponent', {
  needs: ['component:my-kitten']
});

test('renders kittens', function() {
  expect(2);

  // component instance
  var component = this.subject({
    template: Ember.Handlebars.compile(
      '{{#my-album title="Cats"}}' +
      '  {{my-kitten width="200" height="300"}}' +
      '  {{my-kitten width="100" height="100"}}' +
      '  {{my-kitten width="50" height="50"}}' +
      '{{/my-album}}'
    )
  });

  // append component to the dom
  var $component = this.append();

  // perform assertions
  equal($component.find('h3:contains("Cats")').length, 1);
  equal($component.find('img').length, 3);
});
```

Live Example

Components with Embedded Components[277]

[277] http://jsbin.com/xebih

Testing Controllers

Unit testing methods and computed properties follows previous patterns shown in Unit Testing Basics[278] because Ember.Controller extends Ember.Object.

Unit testing controllers is very simple using the unit test helper moduleFor[279] which is part of the ember-qunit framework.

Testing Controller Actions

Here we have a controller `PostsController` with some computed properties and an action `setProps`.

```
App.PostsController = Ember.ArrayController.extend({

  propA: 'You need to write tests',
  propB: 'And write one for me too',

  setPropB: function(str) {
    this.set('propB', str);
  },

  actions: {
    setProps: function(str) {
      this.set('propA', 'Testing is cool');
      this.setPropB(str);
    }
  }
});
```

`setProps` sets a property on the controller and also calls a method. To write a test for this action, we would use the `moduleFor` helper to setup a test container:

```
moduleFor('controller:posts', 'Posts Controller');
```

Next we use `this.subject()` to get an instance of the `PostsController` and write a test to check the action. `this.subject()` is a helper method from the `ember-qunit` library that returns a singleton instance of the module set up using `moduleFor`.

[278] http://emberjs.com/guides/testing/unit-testing-basics
[279] http://emberjs.com/guides/testing/unit

```
test('calling the action setProps updates props A and B', function() {
  expect(4);

  // get the controller instance
  var ctrl = this.subject();

  // check the properties before the action is triggered
  equal(ctrl.get('propA'), 'You need to write tests');
  equal(ctrl.get('propB'), 'And write one for me too');

  // trigger the action on the controller by using the `send` method,
  // passing in any params that our action may be expecting
  ctrl.send('setProps', 'Testing Rocks!');

  // finally we assert that our values have been updated
  // by triggering our action.
  equal(ctrl.get('propA'), 'Testing is cool');
  equal(ctrl.get('propB'), 'Testing Rocks!');
});
```

Live Example

Unit Testing Controllers “Actions”[280]

Testing Controller Needs

Sometimes controllers have dependencies on other controllers. This is accomplished by using needs[281]. For example, here are two simple controllers. The `PostController` is a dependency of the `CommentsController`:

```
App.PostController = Ember.ObjectController.extend({
  // ...
});

App.CommentsController = Ember.ArrayController.extend({
  needs: 'post',
  title: Ember.computed.alias('controllers.post.title'),
});
```

This time when we setup our `moduleFor` we need to pass an options object as our third argument that has the controller’s `needs`.

[280] http://jsbin.com/sanaf

[281] http://emberjs.com/guides/controllers/dependencies-between-controllers

```
moduleFor('controller:comments', 'Comments Controller', {
  needs: ['controller:post']
});
```

Now let's write a test that sets a property on our `post` model in the `PostController` that would be available on the `CommentsController`.

```
test('modify the post', function() {
  expect(2);

  // grab an instance of `CommentsController` and `PostController`
  var ctrl = this.subject(),
      postCtrl = ctrl.get('controllers.post');

  // wrap the test in the run loop because we are dealing with async functions
  Ember.run(function() {

    // set a generic model on the post controller
    postCtrl.set('model', Ember.Object.create({ title: 'foo' }));

    // check the values before we modify the post
    equal(ctrl.get('title'), 'foo');

    // modify the title of the post
    postCtrl.get('model').set('title', 'bar');

    // assert that the controllers title has changed
    equal(ctrl.get('title'), 'bar');

  });
});
```

Live Example

Unit Testing Controllers "Needs"[282]

Testing Routes

Unit testing methods and computed properties follows previous patterns shown in Unit Testing Basics[283] because Ember.Route extends Ember.Object.

[282] http://jsbin.com/busoz

[283] http://emberjs.com/guides/testing/unit-testing-basics

Testing routes can be done both via integration or unit tests. Integration tests will likely provide better coverage for routes because routes are typically used to perform transitions and load data, both of which are tested more easily in full context rather than isolation.

That being said, sometimes it is important to unit test your routes. For example, let's say we'd like to have an alert that can be triggered from anywhere within our application. The alert function `displayAlert` should be put into the `ApplicationRoute` because all actions and events bubble up to it from sub-routes, controllers and views.

```
App.ApplicationRoute = Em.Route.extend({
  actions: {
    displayAlert: function(text) {
      this._displayAlert(text);
    }
  },

  _displayAlert: function(text) {
    alert(text);
  }
});
```

This is made possible by using `moduleFor`.

In this route we've separated our concerns[284]: The action `displayAlert` contains the code that is called when the action is received, and the private function `_displayAlert` performs the work. While not necessarily obvious here because of the small size of the functions, separating code into smaller chunks (or "concerns"), allows it to be more readily isolated for testing, which in turn allows you to catch bugs more easily.

Here is an example of how to unit test this route:

```
moduleFor('route:application', 'Unit: route/application', {
  setup: function() {
    originalAlert = window.alert; // store a reference to the window.alert
  },
  teardown: function() {
    window.alert = originalAlert; // restore original functions
  }
});

test('Alert is called on displayAlert', function() {
  expect(1);
```

[284] http://en.wikipedia.org/wiki/Separation_of_concerns

```

  // with moduleFor, the subject returns an instance of the route
  var route = this.subject(),
      expectedText = 'foo';

  // stub window.alert to perform a qunit test
  window.alert = function(text) {
    equal(text, expectedText, 'expected ' + text + ' to be ' + expectedText);
  }

  // call the _displayAlert function which triggers the qunit test above
  route._displayAlert(expectedText);
});
```

Live Example

Custom Test Helpers[285]

Testing Models

Unit testing methods and computed properties follows previous patterns shown in Unit Testing Basics[286] because DS.Model extends Ember.Object.

Ember Data[287] Models can be tested using the `moduleForModel` helper.

Let's assume we have a `Player` model that has `level` and `levelName` attributes. We want to call `levelUp()` to increment the `level` and assign a new `levelName` when the player reaches level 5.

```
App.Player = DS.Model.extend({
  level:     DS.attr('number', { defaultValue: 0 }),
  levelName: DS.attr('string', { defaultValue: 'Noob' }),

  levelUp: function() {
    var newLevel = this.incrementProperty('level');
    if (newLevel === 5) {
      this.set('levelName', 'Professional');
    }
  }
});
```

Now let's create a test which will call `levelUp` on the player when they are level 4 to assert that the `levelName` changes. We will use `moduleForModel`:

[285] http://jsbin.com/xivoy

[286] http://emberjs.com/guides/testing/unit-testing-basics

[287] https://github.com/emberjs/data

```
moduleForModel('player', 'Player Model');

test('levelUp', function() {
  // this.subject aliases the createRecord method on the model
  var player = this.subject({ level: 4 });

  // wrap asynchronous call in run loop
  Ember.run(function() {
    player.levelUp();
  });

  equal(player.get('level'), 5);
  equal(player.get('levelName'), 'Professional');
});
```

Live Example

Unit Testing Ember Data Models[288]

Testing Relationships

For relationships you probably only want to test that the relationship declarations are setup properly. Assume that a `User` can own a `Profile`.

```
App.Profile = DS.Model.extend({});

App.User = DS.Model.extend({
  profile: DS.belongsTo('profile')
});
```

Then you could test that the relationship is wired up correctly with this test.

[288] http://jsbin.com/naqif

```
moduleForModel('user', 'User Model', {
  needs: ['model:profile']
});

test('profile relationship', function() {
  var User = this.store().modelFor('user');
  var relationship = Ember.get(User, 'relationshipsByName').get('profile');

  equal(relationship.key, 'profile');
  equal(relationship.kind, 'belongsTo');
});
```

Live Example

Unit Testing Models (Relationships : One-to-One)[289]

Ember Data contains extensive tests around the functionality of relationships, so you probably don't need to duplicate those tests. You could look at the Ember Data tests[290] for examples of deeper relationship testing if you feel the need to do it.

Automating Tests with Runners

When it comes to running your tests there are multiple approaches that you can take depending on what best suits your workflow. Finding a low friction method of running your tests is important because it is something that you will be doing quite often.

<a name="browser"></a>The Browser

The simplest way of running your tests is just opening a page in the browser. The following is how to put a test "harness" around your app with qunit so you can run tests against it:

First, get a copy of `qunit` (both the JavaScript and the css) from here[291].

Next, create an HTML file that includes qunit and it's css that looks like the following example.

[289] href=”http://jsbin.com/zuvak

[290] https://github.com/emberjs/data/tree/master/packages/ember-data/tests

[291] http://qunitjs.com/

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>QUnit Example</title>
  <link rel="stylesheet" href="qunit.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script src="qunit.js"></script>
  <script src="your_ember_code_here.js"></script>
  <script src="your_test_code_here.js"></script>
</body>
</html>
```

Finally, launch your browser of choice and open the above html file.

That's it. You're done and your tests are running. No need to install and configure any other tools or have any other processes running. After adding or updating tests and/or code just reload the page and you're off to the races running your tests.

If that meets your needs, read no further. However, if you would like a more automated way of running your tests, read on.

Manually opening and refreshing a browser may prove to be a bit of a tedious workflow for you. While you get the benefit of knowing that your code (and your tests) work in every browser that you are able to launch, it's still up to you to do the launching (and then refreshing) each time you make a change. Getting rid of repetition is why we use computers, so this can be a problem.

Luckily there are tools to help with this. These tools allow you to run your tests in actual browsers (yes browsers - as in more than one at the same time) and then report the results back to you in a consolidated view. These tools are run from the command line and they are also capable of automatically re-running tests when changes are made to files. They require a bit more setup than creating a simple html file but they will likely save time in the long run.

The Testem Runner

Testem[292] is a simple tool to setup and use. In a nutshell it will collect all of your application code, your test code, your testing framework of choice and build a test "harness" automatically. It will then launch each browser (that you specify), run the tests and report the results back to you. It has a nice terminal-based user interface that will display test results for each browser. There are many features built into testem, but it does not seem to have any 3rd party plugins or extensions available.

[292] https://github.com/airportyh/testem

To get started using `testem`, you'll need to install the `testem` node.js module. Assuming you have node[293] installed, run the following command:

```
npm install -g --save-dev testem
```

`Testem` is now available to run your tests. There is just a little bit of configuration that needs to be done first.

```
// testem.json
{
    "framework": "qunit",
    "src_files": [
      "your_ember_code_here.js",
      "your_test_code_here.js"
    ],
    "launch_in_dev": ["PhantomJS"],
    "launch_in_ci": ["PhantomJS"]
}
```

That's it. Everything you need is installed and configured. Let's go over the configuration in more detail.

- `framework`
- This represents the testing framework that you are going to be using. Qunit is what we are using in this example. `Testem` takes care of getting the qunit library loaded up so you don't have to worry about it.
- `src_files`
- This represents which of your source files (including both production and test code) that you want `testem` to load when running tests.
- `launch_in_dev`
- This allows you to configure which browsers to launch and run the tests. This can be one or more browsers. When multiple are specified your tests will run in all browsers concurrently.
- `launch_in_ci`
- This allows you to configure which browsers to launch and run the tests in 'ci' mode. This is specifically geared towards continuous integration[294] environments that may be headless.

There are plenty of other options that you can configure as well if you would like. To see a list of available options you can check out the testem documentation[295].

To start `testem` run the following command.

[293] http://nodejs.org/download/

[294] http://en.wikipedia.org/wiki/Continuous_integration

[295] https://github.com/airportyh/testem

```
testem
```

This will start testem and launch all of your browsers listed in the `launch_in_dev` setting. A tabbed view, one tab for each browser listed, will appear that you can cycle through using the arrow keys to see the test results in each browser. There are other commands that you can use as well, run `testem -h` to see the list of all available commands in the tabbed view. `Testem` will continually run and re-run your tests when changes are made to your files listed in the `src_files` setting.

The `launch_in_ci` setting comes into play when you run `testem` with the following command.

```
testem ci
```

Much like running `testem` with no arguments, the `ci` option will use your same configuration except it will use the `launch_in_ci` rather than the `launch_in_dev` list of browsers. This `ci` option will also cause `testem` to run all of the tests once and exit printing the results to the terminal.

The Karma Test Runner

Karma[296] is another simple tool to setup and use. It is similar to testem in that it will collect all of your application code, your test code, your testing framework of choice and build a test "harness" automatically. It will then launch each browser (that you specify), run the tests and report the results back to you. The terminal user interface is not as fancy as testem, but there is a colored display of test results for each browser. Karma has many features as well as many plugins. For information about writing karma plugins checkout the docs[297]. To find some available karma plugins start with karma_runner[298] on github.

To get started using `karma` you will need to install a few node modules. Here is an example of a package.json[299] file which includes everything that you will need to get started.

```
// package.json
{
  "name": "your_project_name",
  "version": "0.1.0",
  "devDependencies": {
    "karma-qunit": "0.1.1",
    "karma-phantomjs-launcher": "0.1.2",
    "karma": "0.12.1"
  }
}
```

[296] http://karma-runner.github.io/

[297] http://karma-runner.github.io/0.10/config/plugins.html

[298] https://github.com/karma-runner?query=launcher

[299] https://www.npmjs.org/doc/json.html

The three dependencies are `karma` itself, `karma-qunit` which includes everything that you will need to run qunit tests and `karma-phantomjs-launcher` which is what `karma` will use to fire up an instance of the headless PhantomJS browser to run your tests in. There are a number of different launchers that you can plug into the `karma` test runner including but not limited to Google Chrome, FireFox, Safari, IE, and even Sauce Labs[300]. To see a complete list of all of the available launchers check out Karma's Github[301].

Now that you've got a `package.json` containing everything that you will need to get started with `karma` run the following command (in the same directory as your `package.json` file) to download and install everything.

```
npm install
```

`Karma` along with everything else that you need to start running your tests is now available. There is a little bit of configuration that needs to be done first. If you want to generate the default `karma` configuration you can run `karma init` and that will create a `karma.conf.js` file in your current directory. There are many configuration options available, so here's a pared down version: ie, the minimum configuration that Karma requires to run your tests.

```
// karma.conf.js
module.exports = function(config) {
  config.set({
    frameworks: ['qunit'],
    files: [
      'your_ember_code_here.js',
      'your_test_code_here.js'
    ],
    autoWatch: true,
    singleRun: true,
    browsers: ['PhantomJS']
  });
};
```

There is one last thing that you need to install: Karma's command line interface.

```
npm install -g karma-cli
```

That's it. Everything you need is installed and configured. Let's go over the configuration in more detail.

[300] https://saucelabs.com/
[301] https://github.com/karma-runner?query=launcher

- `frameworks`
- This represents the testing frameworks that you're going to use. We're using QUnit in this example. Karma takes care of loading up the QUnit library for you.
- `files`
- This represents which of your source files (including both production and test code) that you want `karma` to load when running tests.
- `autoWatch`
- A value of `true` will mean that `karma` will watch all of the `files` for changes and rerun the tests only when `singleRun` is `false`.
- `singleRun`
- A value of `true` will run all of the tests one time and shut down, whereas a value of `false` will run all of your tests once, then wait for any files to change which will trigger re-running all your tests.
- `browsers`
- This allows you to configure which browsers to launch and run the tests. This can be one or more browsers. When multiple are specified your tests will run in all browsers concurrently.

There are plenty of other options that you can configure as well if you would like. To see a list of available options you can check out the Karma documentation[302] or instead of manually creating `karma.conf.js` you can run the following command.

```
karma init
```

To start `karma` run

```
karma start
```

Depending on your configuration it will either run the tests and exit or run the tests and wait for file changes to run the tests again.

Build Integration

Both `testem` and `karma` are capable of being integrated into larger build processes. For example, you may be using CoffeeScript[303], ES6[304] or something else and need to transpile[305] your source into `JavaScript`. If you happen to be using `grunt` you can use `grunt-contrib-testem` for `testem` or `grunt-karma` for `karma` integration into your existing build process. Both `testem` and `karma` have preprocessing configuration options available as well. For more information on other available configuration options see the docs for karma[306] or testem[307].

[302] http://karma-runner.github.io/

[303] http://coffeescript.org/

[304] http://square.github.io/es6-module-transpiler/

[305] http://en.wikipedia.org/wiki/Source-to-source_compiler

[306] http://karma-runner.github.io/

[307] https://github.com/airportyh/testem

Generating Reports

Oftentimes it's useful to get the results of your tests in different formats. For example, if you happen to use Jenkins[308] as a ci[309] server, you may want to get your test results in XML format so Jenkins can build some graphs of your test results over time. Also, you may want to measure your code coverage[310] and have Jenkins track that over time as well. With these test runners, it's possible to generate reports from the results in various formats, as well as record other information such as code-test coverage, etc.

XML Test Results from Testem

To get junit xml[311] from the `testem` test runner you can simply add a flag to the command when you run `testem` and pipe the output to a file like the following command.

```
testem ci -R xunit > test-results.xml
```

That's it! Now you can use `test-results.xml` to feed into another tool.

XML Test Results from Karma

To get junit xml[312] from the `karma` test runner you will need to install a new node.js module. You can do so with the following command.

```
npm install --save-dev karma-junit-reporter
```

Once that is done you will need to update your karma configuration to include the following.

```
module.exports = function(config) {
  config.set({
    /* snip */
    reporters: ['progress', 'junit'],
    /* snip */
  });
};
```

The reporters option determines how your test results are communicated back to you. The `progress` reporter will display a line that says something like this.

[308] http://jenkins-ci.org/

[309] http://en.wikipedia.org/wiki/Continuous_integration

[310] http://en.wikipedia.org/wiki/Code_coverage

[311] http://ant.apache.org/manual/Tasks/junitreport.html

[312] http://ant.apache.org/manual/Tasks/junitreport.html

```
PhantomJS 1.9.7 (Mac OS X): Executed 2 of 2 SUCCESS (0.008 secs / 0.002 secs)
```

The `junit` reporter will create an xml file called `test-results.xml` in the current directory that contains junit xml which can be used as input to other tools. This file can be renamed to whatever you would like. For more information see the docs for karma junit reporter[313].

Code Coverage from Testem

Getting coverage from `testem` is a bit more involved at the moment, though there **is** a way to do it. Check the testem docs[314] for more information.

Code Coverage from Karma

To measure your code coverage[315] from the `karma` test runner you will need to install a new node.js module. You can do so with the following command.

```
npm install --save-dev karma-coverage
```

Once that's done you will need to update your karma configuration to include the following.

```
module.exports = function(config) {
  config.set({
    /* snip */
    reporters: ['progress', 'coverage'],
    preprocessors: {
      "your_ember_code_here.js": "coverage",
      "your_test_code_here.js": "coverage"
    },
    coverageReporter: {
        type: "text",
    }
    /* snip */
  });
};
```

That's it. Now, running `karma` normally will display code coverage information in the terminal. The `coverageReporter.type` option can be set to a number of different values. The value in the example, `text`, will only display to the console. Some other options are `lcov`, `html` and `cobertura` which can be used as input to other tools. For additional configuration options on coverage reporting from `karma` check out their docs[316].

[313] https://github.com/karma-runner/karma-junit-reporter
[314] https://github.com/airportyh/testem/tree/master/examples/coverage_istanbul
[315] http://en.wikipedia.org/wiki/Code_coverage
[316] http://karma-runner.github.io/0.8/config/coverage.html

Configuring Ember.js

Disabling Prototype Extensions

By default, Ember.js will extend the prototypes of native JavaScript objects in the following ways:

- `Array` is extended to implement the `Ember.Enumerable`, `Ember.MutableEnumerable`, `Ember.MutableArray` and `Ember.Array` interfaces. This polyfills ECMAScript 5 array methods in browsers that do not implement them, adds convenience methods and properties to built-in arrays, and makes array mutations observable.
- `String` is extended to add convenience methods, such as `camelize()` and `fmt()`.
- `Function` is extended with methods to annotate functions as computed properties, via the `property()` method, and as observers, via the `observes()` or `observesBefore()` methods.

This is the extent to which Ember.js enhances native prototypes. We have carefully weighed the tradeoffs involved with changing these prototypes, and recommend that most Ember.js developers use them. These extensions significantly reduce the amount of boilerplate code that must be typed.

However, we understand that there are cases where your Ember.js application may be embedded in an environment beyond your control. The most common scenarios are when authoring third-party JavaScript that is embedded directly in other pages, or when transitioning an application piecemeal to a more modern Ember.js architecture.

In those cases, where you can't or don't want to modify native prototypes, Ember.js allows you to completely disable the extensions described above.

To do so, simply set the `EXTEND_PROTOTYPES` flag to `false`:

```
window.ENV = {};
ENV.EXTEND_PROTOTYPES = false;
```

Note that the above code must be evaluated **before** Ember.js loads. If you set the flag after the Ember.js JavaScript file has been evaluated, the native prototypes will already have been modified.

Life Without Prototype Extension

In order for your application to behave correctly, you will need to manually extend or create the objects that the native objects were creating before.

Arrays

Native arrays will no longer implement the functionality needed to observe them. If you disable prototype extension and attempt to use native arrays with things like a template's `{{#each}}` helper, Ember.js will have no way to detect changes to the array and the template will not update as the underlying array changes.

Additionally, if you try to set the model of an `Ember.ArrayController` to a plain native array, it will raise an exception since it no longer implements the `Ember.Array` interface.

You can manually coerce a native array into an array that implements the required interfaces using the convenience method `Ember.A`:

```
var islands = ['Oahu', 'Kauai'];
islands.contains('Oahu');
//=> TypeError: Object Oahu,Kauai has no method 'contains'

// Convert `islands` to an array that implements the
// Ember enumerable and array interfaces
Ember.A(islands);

islands.contains('Oahu');
//=> true
```

Strings

Strings will no longer have the convenience methods described in the Ember.String API reference.[317]. Instead, you can use the similarly-named methods of the `Ember.String` object and pass the string to use as the first parameter:

```
"my_cool_class".camelize();
//=> TypeError: Object my_cool_class has no method 'camelize'

Ember.String.camelize("my_cool_class");
//=> "myCoolClass"
```

Functions

To annotate computed properties, use the `Ember.computed()` method to wrap the function:

[317] http:emberjs.com/api/classes/Ember.String.html

```
// This won't work:
fullName: function() {
  return this.get('firstName') + ' ' + this.get('lastName');
}.property('firstName', 'lastName')


// Instead, do this:
fullName: Ember.computed('firstName', 'lastName', function() {
  return this.get('firstName') + ' ' + this.get('lastName');
})
```

Observers are annotated using `Ember.observer()`:

```
// This won't work:
fullNameDidChange: function() {
  console.log("Full name changed");
}.observes('fullName')


// Instead, do this:
fullNameDidChange: Ember.observer('fullName', function() {
  console.log("Full name changed");
})
```

Embedding Applications

In most cases, your application's entire UI will be created by templates that are managed by the router.

But what if you have an Ember.js app that you need to embed into an existing page, or run alongside other JavaScript frameworks?

Changing the Root Element

By default, your application will render the application template[318] and attach it to the document's `body` element.

You can tell the application to append the application template to a different element by specifying its `rootElement` property:

[318] http://emberjs.com/guides/templates/the-application-template

```
App = Ember.Application.create({
  rootElement: '#app'
});
```

This property can be specified as either an element or a jQuery-compatible selector string[319].

Disabling URL Management

You can prevent Ember from making changes to the URL by changing the router's `location`[320] to `none`:

```
App.Router = Ember.Router.extend({
  location: 'none'
});
```

Feature Flags

About Features

When a new feature is added to Ember they will be written in such a way that the feature can be conditionally included in the generated build output and enabled (or completely removed) based on whether a particular flag is present. This allows newly developed features to be selectively released when they are considered ready for production use.

Feature Life-Cycle

When a new feature is flagged it is only available in canary builds (if enabled at runtime). When it is time for the next beta cycle to be started (generally 6-12 week cycles) each feature will be evaluated and those features that are ready will be enabled in the next `beta` (and subsequently automatically enabled in all future canary builds).

If a given feature is deemed unstable it will be disabled in the next beta point release, and not be included in the next stable release. It may still be included in the next beta cycle if the issues/concerns have been resolved.

Once the beta cycle has completed the final release will include any features that were enabled during that cycle. At this point the feature flags will be removed from the canary and future beta branches, and the feature flag will no longer be used.

[319] http://api.jquery.com/category/selectors/

[320] http://emberjs.com/guides/routing/specifying-the-location-api

Flagging Details

The flag status in the generated build output is controlled by the `features.json` file in the root of the project. This file lists all features and their current status.

A feature can have one of a few different statuses:

- `true` - The feature is **enabled**: the code behind the flag is always enabled in the generated build.
- `false` - The feature is **disabled**: the code behind the flag is not present in the generated build at all.
- `null` - The feature is **present** in the build output, but must be enabled at runtime (it is still behind feature flags).

The process of removing the feature flags from the resulting build output is handled by `defeatureify`.

Feature Listing (`FEATURES.md`)

When a new feature is added to the `canary` channel (aka `master` branch), an entry is added to `FEATURES.md`[321] explaining what the feature does (and linking the originating pull request). This listing is kept current, and reflects what is available in each branch (`stable`,`beta`, and `master`).

Enabling At Runtime

The only time a feature can be enabled at runtime is if the `features.json` for that build contains `null` (technically, anything other than `true` or `false` will do, but `null` is the chosen value).

A global `EmberENV` object will be used to initialize the `Ember.ENV` object, and any feature flags that are enabled/disabled under `EmberENV.FEATURES` will be migrated to `Ember.FEATURES`; those features will be enabled based on the flag value. **Ember only reads** the `EmberENV` value upon initial load so setting this value after Ember has been loaded will have no affect.

Example:

```
EmberENV = {FEATURES: {'link-to': true}};
```

Additionally you can define `EmberENV.ENABLE_ALL_FEATURES` to force all features to be enabled.

[321] https://github.com/emberjs/ember.js/blob/master/FEATURES.md

Cookbook

Introduction

Welcome to the Ember.js Cookbook! The Cookbook provides answers and solutions to common Ember questions and problems. Anyone is welcome to contribute[322].

Here are all of the available recipes:

Contributing

1. Understanding the Cookbook Format[323]
2. Participating If You Know Ember[324]
3. Participating If You Don't Know Ember[325]
4. Deciding If A Recipe is a Good Fit[326]
5. Suggesting A Recipe[327]

User Interface & Interaction

1. Adding CSS Classes to Your Components[328]
2. Adding CSS Classes to Your Components Based on Properties[329]
3. Focusing a Textfield after It's Been Inserted[330]
4. Displaying Formatted Dates With Moment.js[331]
5. Specifying Data-Driven Areas of Templates That Do Not Need To Update[332]
6. Using Modal Dialogs[333]
7. Resetting scroll on route changes[334]

[322] http://emberjs.com/guides/cookbook/contributing

[323] http://emberjs.com/guides/cookbook/contributing/understanding_the_cookbook_format

[324] http://emberjs.com/guides/cookbook/contributing/participating_if_you_know_ember

[325] http://emberjs.com/guides/cookbook/contributing/participating_if_you_dont_know_ember

[326] http://emberjs.com/guides/cookbook/contributing/deciding_if_a_recipe_is_a_good_fit

[327] http://emberjs.com/guides/cookbook/contributing/suggesting_a_recipe

[328] http://emberjs.com/guides/cookbook/user_interface_and_interaction/adding_css_classes_to_your_components

[329] http://emberjs.com/guides/cookbook/user_interface_and_interaction/adding_css_classes_to_your_components_based_on_properties

[330] http://emberjs.com/guides/cookbook/user_interface_and_interaction/focusing_a_textfield_after_its_been_inserted

[331] http://emberjs.com/guides/cookbook/user_interface_and_interaction/displaying_formatted_dates_with_moment_js

[332] http://emberjs.com/guides/cookbook/user_interface_and_interaction/specifying_data_driven_areas_of_templates_that_do_not_need_to_update

[333] http://emberjs.com/guides/cookbook/user_interface_and_interaction/using_modal_dialogs

[334] http://emberjs.com/guides/cookbook/user_interface_and_interaction/resetting_scroll_on_route_changes

Event Handling & Data Binding

1. Binding Properties of an Object to Its Own Properties[335]

Helpers & Components

1. Creating Reusable Social Share Buttons[336]
2. A Spinning Button for Asynchronous Actions[337]
3. Adding Google Analytics Tracking[338]

Working with Objects

1. Incrementing Or Decrementing A Property[339]
2. Setting Multiple Properties At Once[340]
3. Continuous Redrawing of Views[341]

If you would like to see more recipes, take a look at the Suggesting A Recipe[342] section.

Contributing

The Ember Cookbook provides answers and solutions to common Ember questions and problems. Anyone is welcome to contribute[343].

If you are new to Ember, we recommend that you spend some time reading the guides and tutorials before coming to the Cookbook. Cookbook recipes assume that you have a basic understanding of Ember's concepts.

If you have experience with Ember and would like to contribute to the Cookbook, the discussion section of each recipe is a great place to start.

Recipes

1. Understanding the Cookbook Format[344]
2. Participating If You Know Ember[345]
3. Participating If You Don't Know Ember[346]

[335] http://emberjs.com/guides/cookbook/event_handling_and_data_binding/binding_properties_of_an_object_to_its_own_properties

[336] http://emberjs.com/guides/cookbook/helpers_and_components/creating_reusable_social_share_buttons

[337] http://emberjs.com/guides/cookbook/helpers_and_components/spin_button_for_asynchronous_actions

[338] http://emberjs.com/guides/cookbook/helpers_and_components/adding_google_analytics_tracking

[339] http://emberjs.com/guides/cookbook/working_with_objects/incrementing_or_decrementing_a_property

[340] http://emberjs.com/guides/cookbook/working_with_objects/setting_multiple_properties_at_once

[341] http://emberjs.com/guides/cookbook/working_with_objects/continuous_redrawing_of_views

[342] http://emberjs.com/guides/cookbook/contributing/suggesting_a_recipe

[343] http://emberjs.com/guides/cookbook/contributing/understanding_the_cookbook_format

[344] http://emberjs.com/guides/cookbook/contributing/understanding_the_cookbook_format

[345] http://emberjs.com/guides/cookbook/contributing/participating_if_you_know_ember

[346] http://emberjs.com/guides/cookbook/contributing/participating_if_you_dont_know_ember

4. Deciding If A Recipe is a Good Fit[347]
5. Suggesting A Recipe[348]

User Interface & Interaction

Here are some recipes that will help you provide a better user experience.

1. Adding CSS Classes to Your Components[349]
2. Adding CSS Classes to Your Components Based on Properties[350]
3. Focusing a Textfield after It's Been Inserted[351]
4. Displaying Formatted Dates With Moment.js[352]
5. Specifying Data-Driven Areas of Templates That Do Not Need To Update[353]
6. Using Modal Dialogs[354]
7. Resetting scroll on route changes[355]

Event Handling & Data Binding

Problem

You want to base the value of one property on the value of another property.

Solution

Use one of the computed property macros like `Ember.computed.alias` or `Ember.computed.gte`

[347] http://emberjs.com/guides/cookbook/contributing/deciding_if_a_recipe_is_a_good_fit

[348] http://emberjs.com/guides/cookbook/contributing/suggesting_a_recipe

[349] http://emberjs.com/guides/cookbook/user_interface_and_interaction/adding_css_classes_to_your_components

[350] http://emberjs.com/guides/cookbook/user_interface_and_interaction/adding_css_classes_to_your_components_based_on_properties

[351] http://emberjs.com/guides/cookbook/user_interface_and_interaction/focusing_a_textfield_after_its_been_inserted

[352] http://emberjs.com/guides/cookbook/user_interface_and_interaction/displaying_formatted_dates_with_moment_js

[353] http://emberjs.com/guides/cookbook/user_interface_and_interaction/specifying_data_driven_areas_of_templates_that_do_not_need_to_update

[354] http://emberjs.com/guides/cookbook/user_interface_and_interaction/using_modal_dialogs

[355] http://emberjs.com/guides/cookbook/user_interface_and_interaction/resetting_scroll_on_route_changes

```
App.Person = Ember.Object.extend({
        firstName : null,
        lastName : null,
        surname : Ember.computed.alias("lastName"),
        eligibleForRetirement: Ember.computed.gte("age", 65)
});
```

Discussion

Ember.js includes a number of macros that will help create properties whose values are based on the values of other properties, correctly connecting them with bindings so they remain updated when values change. These all are stored on the `Ember.computed` object and documented in the API documentation[356]

Example

JS Bin[357]

Helpers & Components

Problem

You want to create a reusable Tweet button[358] for your application.

Solution

Write a custom component that renders the Tweet button with specific attributes passed in.

```
{{share-twitter data-url="http://emberjs.com"
                data-text="EmberJS Components are Amazing!"
                data-size="large"
                data-hashtags="emberjs"}}
```

[356] http://emberjs.com/api/#method_computed

[357] http://jsbin.com/AfufoSO

[358] https://dev.twitter.com/docs/tweet-button

```
App.ShareTwitterComponent = Ember.Component.extend({
  tagName: 'a',
  classNames: 'twitter-share-button',
  attributeBindings: ['data-size', 'data-url', 'data-text', 'data-hashtags']
});
```

Include Twitter's widget code in your HTML:

```
<script type="text/javascript" src="http://platform.twitter.com/widgets.js" id="\
twitter-wjs"></script>
```

Discussion

Twitter's widget library expects to find an `<a>` tag on the page with specific `data-` attributes applied. It takes the values of these attributes and, when the `<a>` tag is clicked, opens an iFrame for twitter sharing.

The `share-twitter` component takes four options that match the four attributes for the resulting `<a>` tag: `data-url`, `data-text`, `data-size`, `data-hashtags`. These options and their values become properties on the component object.

The component defines certain attributes of its HTML representation as bound to properties of the object through its `attributeBindings` property. When the values of these properties change, the component's HTML element's attributes will be updated to match the new values.

An appropriate tag and css class are applied through the `tagName` and `classNames` properties.

Example

JS Bin[359]

Working with Objects

Here are some recipes to help you understand working with Ember Objects.

1. Incrementing Or Decrementing A Property[360]
2. Setting Multiple Properties At Once[361]
3. Continuous Redrawing of Views[362]

[359] http://jsbin.com/OpocEPu

[360] http://emberjs.com/guides/cookbook/working_with_objects/incrementing_or_decrementing_a_property

[361] http://emberjs.com/guides/cookbook/working_with_objects/setting_multiple_properties_at_once

[362] http://emberjs.com/guides/cookbook/working_with_objects/continuous_redrawing_of_views

Understanding Ember.js

The View Layer

This guide goes into extreme detail about the Ember.js view layer. It is intended for an experienced Ember developer, and includes details that are unnecessary for getting started with Ember.

Ember.js has a sophisticated system for creating, managing and rendering a hierarchy of views that connect to the browser's DOM. Views are responsible for responding to user events, like clicks, drags, and scrolls, as well as updating the contents of the DOM when the data underlying the view changes.

View hierarchies are usually created by evaluating a Handlebars template. As the template is evaluated, child views are added. As the templates for *those* child views are evaluated, they may have child views added, and so on, until an entire hierarchy is created.

Even if you do not explicitly create child views from your Handlebars templates, Ember.js internally uses the view system to update bound values. For example, every Handlebars expression `{{value}}` creates a view behind-the-scenes that knows how to update the bound value if it changes.

You can also dynamically make changes to the view hierarchy at application runtime using the `Ember.ContainerView` class. Rather than being template-driven, a container view exposes an array of child view instances that can be manually managed.

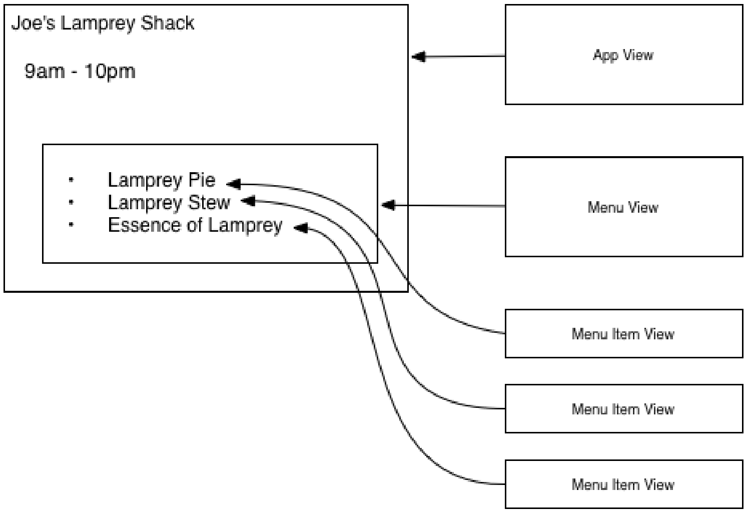
Views and templates work in tandem to provide a robust system for creating whatever user interface you dream up. End users should be isolated from the complexities of things like timing issues while rendering and event propagation. Application developers should be able to describe their UI once, as a string of Handlebars markup, and then carry on with their application without having to worry about making sure that it remains up-to-date.

What problems does it solve?

Child Views

In a typical client-side application, views may represent elements nested inside of each other in the DOM. In the naïve solution to this problem, separate view objects represent each DOM element, and ad-hoc references help the various view object keep track of the views conceptually nested inside of them.

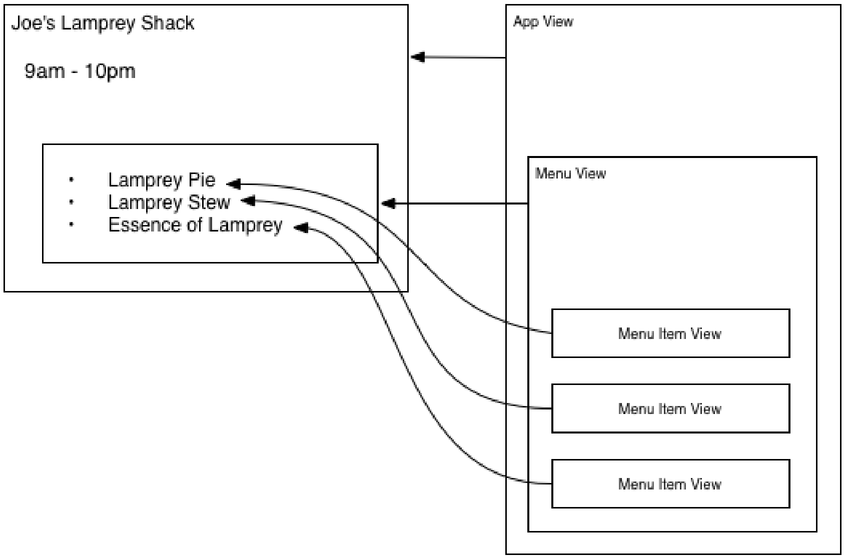
Here is a simple example, representing one main app view, a collection nested inside of it, and individual items nested inside of the collection.



This system works well at first glance, but imagine that we want to open Joe's Lamprey Shack at 8am instead of 9am. In this situation, we will want to re-render the App View. Because the developer needed to build up the references to the children on an ad-hoc basis, this re-rendering process has several problems.

In order to re-render the App View, the App View must also manually re-render the child views and re-insert them into App View's element. If implemented perfectly, this process works well, but it relies upon a perfect, ad hoc implementation of a view hierarchy. If any single view fails to implement this precisely, the entire re-render will fail.

In order to avoid these problems, Ember's view hierarchy has the concept of child views baked in.



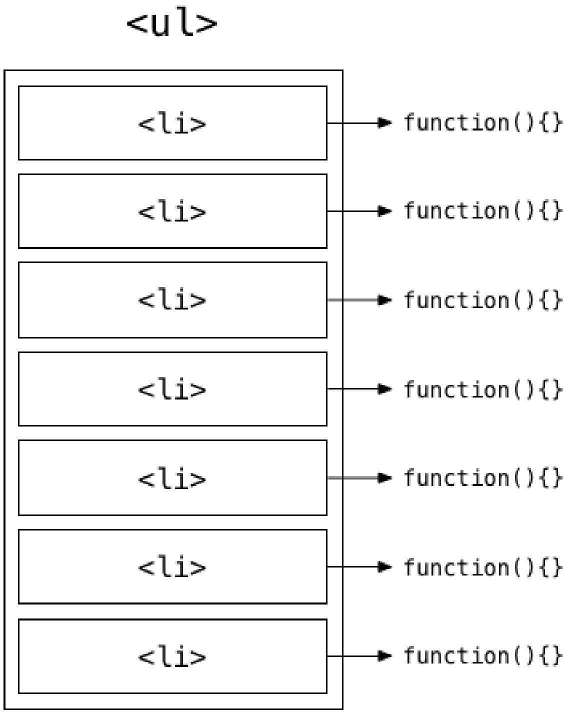
When the App View re-renders, Ember is responsible for re-rendering and inserting the child views, not application code. This also means that Ember can perform any memory management for you, such as cleaning up observers and bindings.

Not only does this eliminate quite a bit of boilerplate code, but it eliminates the possibility that an imperfectly implemented view hierarchy will cause unexpected failures.

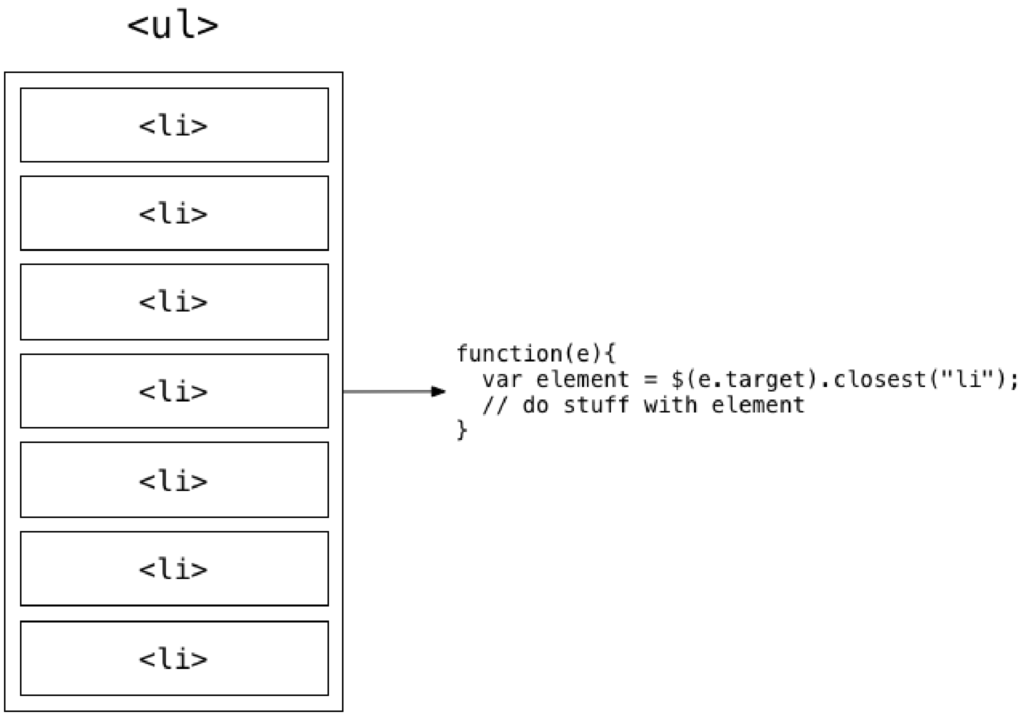
Event Delegation

In the past, web developers have added event listeners to individual elements in order to know when the user interacts with them. For example, you might have a `<div>` element on which you register a function that gets called when the user clicks it.

However, this approach often does not scale when dealing with large numbers of interactive elements. For example, imagine a `<ul>` with 100 `<li>`s in it, with a delete button next to each item. Since the behavior is the same for all of these items, it would be inefficient to create 100 event listeners, one for each delete button.



To solve this problem, developers discovered a technique called "event delegation". Instead of registering a listener on each element in question, you can register a single listener for the containing element and use `event.target` to identify which element the user clicked on.



```
function(e){
  var element = $(e.target).closest("li");
  // do stuff with element
}
```

Implementing this is a bit tricky, because some events (like `focus`, `blur` and `change`) don't bubble. Fortunately, jQuery has solved this problem thoroughly; using jQuery's `on` method reliably works for all native browser events.

Other JavaScript frameworks tackle this problem in one of two ways. In the first approach, they ask you to implement the naïve solution yourself, creating a separate view for each element. When you create the view, it sets up an event listener on the view's element. If you had a list of 500 items, you would create 500 views and each would set up a listener on its own element.

In the second approach, the framework builds in event delegation at the view level. When creating a view, you can supply a list of events to delegate and a method to call when the event occurs. This leaves identifying the context of the click (for example, which item in the list) to the method receiving the event.

You are now faced with an uncomfortable choice: create a new view for each item and lose the benefits of event delegation, or create a single view for all of the items and have to store information about the underlying JavaScript object in the DOM.

In order to solve this problem, Ember delegates all events to the application's root element (usually the document `body`) using jQuery. When an event occurs, Ember identifies the nearest view that handles the event and invokes its event handler. This means that you can create views to hold a JavaScript context, but still get the benefit of event delegation.

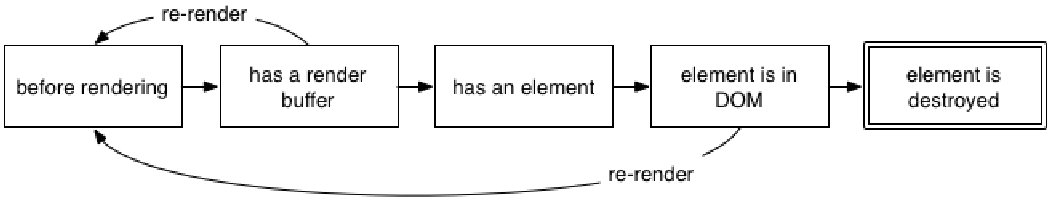
Further, because Ember registers only one event for the entire Ember application, creating new views never requires setting up event listeners, making re-renders efficient and less error-prone. When a view has child views, this also means that there is no need to manually undelegate views that the re-render process replaces.

The Rendering Pipeline

Most web applications specify their user interface using the markup of a particular templating language. For Ember.js, we've done the work to make templates written using the Handlebars templating language automatically update when the values used inside of them are changed.

While the process of displaying a template is automatic for developers, under the hood there are a series of steps that must be taken to go from the original template to the final, live DOM representation that the user sees.

This is the approximate lifecycle of an Ember view:



1. Template Compilation

The application's templates are loaded over the network or as part of the application payload in string form. When the application loads, it sends the template string to Handlebars to be compiled into a function. Once compiled, the template function is saved, and can be used by multiple views repeatedly, each time they need to re-render.

This step may be omitted in applications where the templates are pre-compiled on the server. In those cases, the template is transferred not as the original, human-readable template string but as the compiled code.

Because Ember is responsible for template compilation, you don't have to do any additional work to ensure that compiled templates are reused.

2. String Concatenation

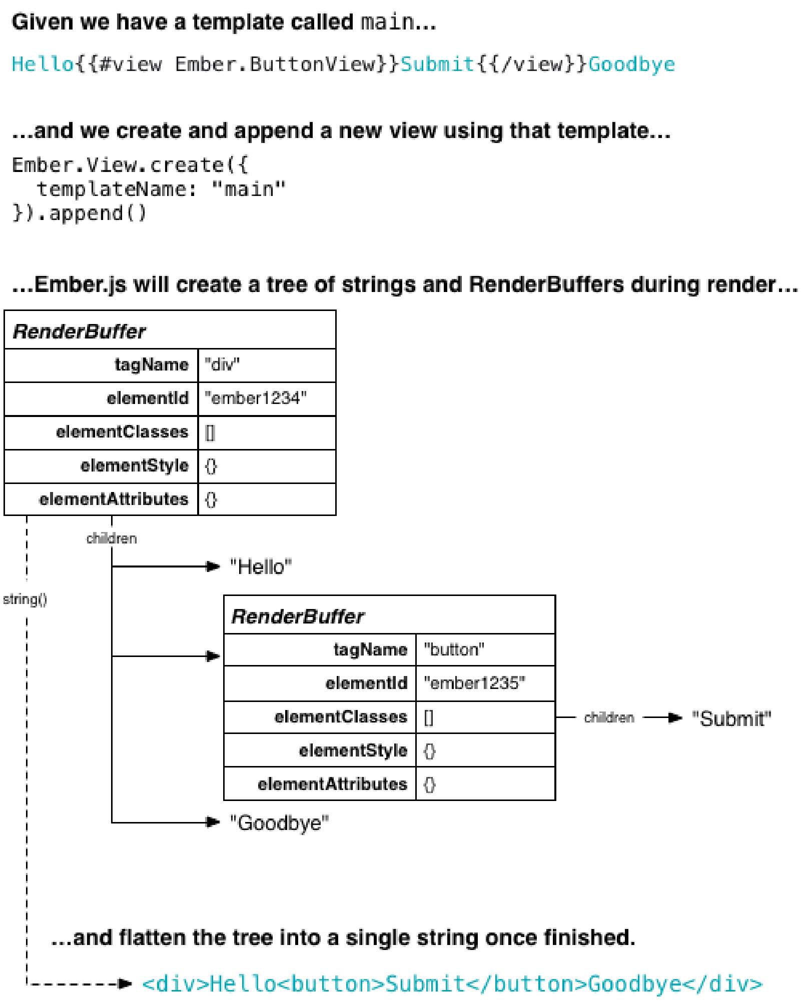
A view's rendering process is kickstarted when the application calls `append` or `appendTo` on the view. Calling `append` or `appendTo` **schedules** the view to be rendered and inserted later. This allows any deferred logic in your application (such as binding synchronization) to happen before rendering the element.

To begin the rendering process, Ember creates a `RenderBuffer` and gives it to the view to append its contents to. During this process, a view can create and render child views. When it does so, the parent view creates and assigns a `RenderBuffer` for the child, and links it to the parent's `RenderBuffer`.

Ember flushes the binding synchronization queue before rendering each view. By syncing bindings before rendering each view, Ember guarantees that it will not render stale data it needs to replace right away.

Once the main view has finished rendering, the render process has created a tree of views (the "view hierarchy"), linked to a tree of buffers. By walking down the tree of buffers and converting them into Strings, we have a String that we can insert into the DOM.

Here is a simple example:



Given we have a template called `main`**...**

```
Hello{{#view Ember.ButtonView}}Submit{{/view}}Goodbye
```

...and we create and append a new view using that template...

```
Ember.View.create({
  templateName: "main"
}).append()
```

...Ember.js will create a tree of strings and RenderBuffers during render...

RenderBuffer	
tagName	"div"
elementId	"ember1234"
elementClasses	[]
elementStyle	{}
elementAttributes	{}

RenderBuffer	
tagName	"button"
elementId	"ember1235"
elementClasses	[]
elementStyle	{}
elementAttributes	{}

...and flatten the tree into a single string once finished.

```
<div>Hello<button>Submit</button>Goodbye</div>
```

In addition to children (Strings and other `RenderBuffers`), a `RenderBuffer` also encapsulates the element's tag name, id, classes, style, and other attributes. This makes it possible for the render process to modify one of these properties (style, for example), even after its child Strings have rendered. Because many of these properties are controlled via bindings (e.g. using `bind-attr`), this makes the process robust and transparent.

3. Element Creation and Insertion

At the end of the rendering process, the root view asks the `RenderBuffer` for its element. The `RenderBuffer` takes its completed string and uses jQuery to convert it into an element. The view assigns that element to its `element` property and places it into the correct place in the DOM (the location specified in `appendTo` or the application's root element if the application used `append`).

While the parent view assigns its element directly, each child views looks up its element lazily. It does this by looking for an element whose `id` matches its `elementId` property. Unless explicitly provided, the rendering process generates an `elementId` property and assigns its value to the view's `RenderBuffer`, which allows the view to find its element as needed.

4. Re-Rendering

After the view inserts itself into the DOM, either Ember or the application may want to re-render the view. They can trigger a re-render by calling the `rerender` method on a view.

Rerendering will repeat steps 2 and 3 above, with two exceptions:

- Instead of inserting the element into an explicitly specified location, `rerender` replaces the existing element with the new element.
- In addition to rendering a new element, it also removes the old element and destroys its children. This allows Ember to automatically handle unregistering appropriate bindings and observers when re-rendering a view. This makes observers on a path more viable, because the process of registering and unregistering all of the nested observers is automatic.

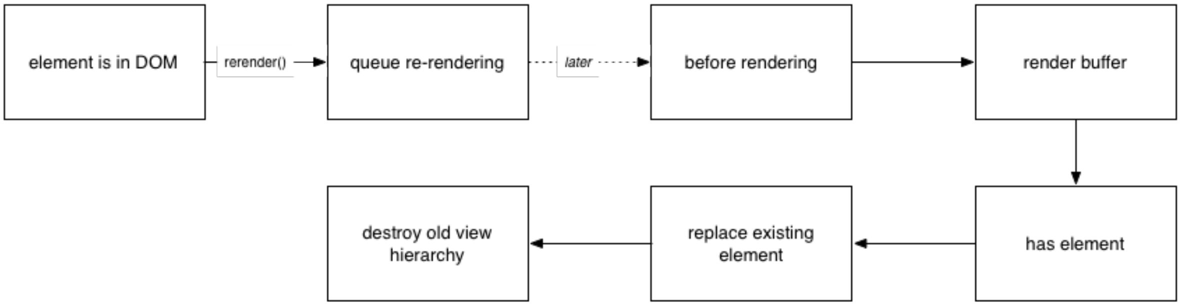
The most common cause of a view re-render is when the value bound to a Handlebars expression (`{{foo}}`) changes. Internally, Ember creates a simple view for each expression, and registers an observer on the path. When the path changes, Ember updates the area of the DOM with the new value.

Another common case is an `{{#if}}` or `{{#with}}` block. When rendering a template, Ember creates a virtual view for these block helpers. These virtual views do not appear in the publicly available view hierarchy (when getting `parentView` and `childViews` from a view), but they exist to enable consistent re-rendering.

When the path passed to an `{{#if}}` or `{{#with}}` changes, Ember automatically re-renders the virtual view, which will replace its contents, and importantly, destroy all child views to free up their memory.

In addition to these cases, the application may sometimes want to explicitly re-render a view (usually a `ContainerView`, see below). In this case, the application can call `rerender` directly, and Ember will queue up a re-rendering job, with the same semantics.

The process looks something like:



The View Hierarchy

Parent and Child Views

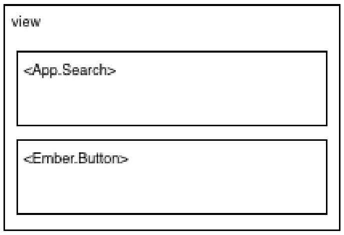
As Ember renders a templated view, it will generate a view hierarchy. Let's assume we have a template `form`.

```
{{view App.Search placeholder="Search"}}
{{#view Ember.Button}}Go!{{/view}}
```

And we insert it into the DOM like this:

```
var view = Ember.View.create({
  templateName: 'form'
}).append();
```

This will create a small view hierarchy that looks like this:



You can move around in the view hierarchy using the `parentView` and `childViews` properties.

```
var children = view.get('childViews') // [ <App.Search>, <Ember.Button> ]
children.objectAt(0).get('parentView') // view
```

One common use of the `parentView` method is inside of an instance of a child view.

```
App.Search = Ember.View.extend({
  didInsertElement: function() {
    // this.get('parentView') in here references `view`
  }
})
```

Lifecycle Hooks

In order to make it easy to take action at different points during your view's lifecycle, there are several hooks you can implement.

- `willInsertElement`: This hook is called after the view has been rendered but before it has been inserted into the DOM. It does not provide access to the view's `element`.
- `didInsertElement`: This hook is called immediately after the view has been inserted into the DOM. It provides access to the view's `element` and is most useful for integration with an external library. Any explicit DOM setup code should be limited to this hook.
- `willDestroyElement`: This hook is called immediately before the element is removed from the DOM. This is your opportunity to tear down any external state associated with the DOM node. Like `didInsertElement`, it is most useful for integration with external libraries.
- `willClearRender`: This hook is called immediately before a view is re-rendered. This is useful if you want to perform some teardown immediately before a view is re-rendered.
- `becameVisible`: This hook is called after a view's `isVisible` property, or one of its ancestor's `isVisible` property, changes to true and the associated element becomes visible. Note that this hook is only reliable if all visibility is routed through the `isVisible` property.
- `becameHidden`: This hook is called after a view's `isVisible` property, or one of its ancestor's `isVisible` property, changes to false and the associated element becomes hidden. Note that this hook is only reliable if all visibility is routed through the `isVisible` property.

Apps can implement these hooks by defining a method by the hook's name on the view. Alternatively, it is possible to register a listener for the hook on a view:

```
view.on('willClearRender', function() {
  // do something with view
});
```

Virtual Views

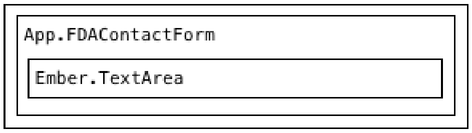
As described above, Handlebars creates views in the view hierarchy to represent bound values. Every time you use a Handlebars expression, whether it's a simple value or a block helper like `{{#with}}` or `{{#if}}`, Handlebars creates a new view.

Because Ember uses these views for internal bookkeeping only, they are hidden from the view's public `parentView` and `childViews` API. The public view hierarchy reflects only views created using the `{{view}}` helper or through `ContainerView` (see below).

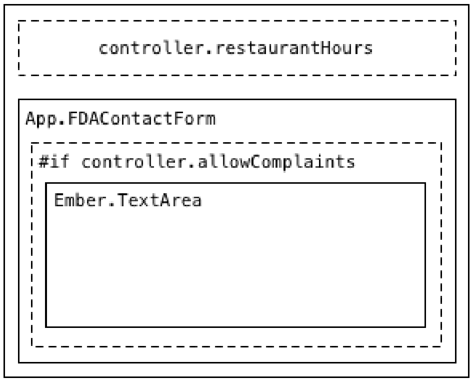
For example, consider the following Handlebars template:

```
<h1>Joe's Lamprey Shack</h1>
{{controller.restaurantHours}}

{{#view App.FDAContactForm}}
  If you are experiencing discomfort from eating at Joe's Lamprey Shack,
please use the form below to submit a complaint to the FDA.

  {{#if controller.allowComplaints}}
    {{view Ember.TextArea valueBinding="controller.complaint"}}
    <button {{action 'submitComplaint'}}>Submit</button>
  {{/if}}
{{/view}}
```

Rendering this template would create a hierarchy like this:



Behind the scenes, Ember tracks additional virtual views for the Handlebars expressions:



From inside of the `TextArea`, the `parentView` would point to the `FDAContactForm` and the `FDAContactForm`'s `childViews` would be an array of the single `TextArea` view.

You can see the internal view hierarchy by asking for the `_parentView` or `_childViews`, which will include virtual views:

```
var _childViews = view.get('_childViews');
console.log(_childViews.objectAt(0).toString());
//> <Ember._HandlebarsBoundView:ember1234>
```

Warning! You may not rely on these internal APIs in application code. They may change at any time and have no public contract. The return value may not be observable or bindable. It may not be an Ember object. If you feel the need to use them, please contact us so we can expose a better public API for your use-case.

Bottom line: This API is like XML. If you think you have a use for it, you may not yet understand the problem enough. Reconsider!

Event Bubbling

One responsibility of views is to respond to primitive user events and translate them into events that have semantic meaning for your application.

For example, a delete button translates the primitive `click` event into the application-specific "remove this item from an array."

In order to respond to user events, create a new view subclass that implements that event as a method:

```
App.DeleteButton = Ember.View.create({
  click: function(event) {
    var item = this.get('model');
    this.get('controller').send('deleteItem', item);
  }
});
```

When you create a new `Ember.Application` instance, it registers an event handler for each native browser event using jQuery's event delegation API. When the user triggers an event, the application's event dispatcher will find the view nearest to the event target that implements the event.

A view implements an event by defining a method corresponding to the event name. When the event name is made up of multiple words (like `mouseup`) the method name should be the camelized form of the event name (`mouseUp`).

Events will bubble up the view hierarchy until the event reaches the root view. An event handler can stop propagation using the same techniques as normal jQuery event handlers:

- `return false` from the method
- `event.stopPropagation`

For example, imagine you defined the following view classes:

```
App.GrandparentView = Ember.View.extend({
  click: function() {
    console.log('Grandparent!');
  }
});

App.ParentView = Ember.View.extend({
  click: function() {
    console.log('Parent!');
    return false;
  }
});

App.ChildView = Ember.View.extend({
  click: function() {
    console.log('Child!');
  }
});
```

And here's the Handlebars template that uses them:

```
{{#view App.GrandparentView}}
  {{#view App.ParentView}}
    {{#view App.ChildView}}
      <h1>Click me!</h1>
    {{/view}}
  {{/view}}
{{/view}}
```

If you clicked on the `<h1>`, you'd see the following output in your browser's console:

```
Child!
Parent!
```

You can see that Ember invokes the handler on the child-most view that received the event. The event continues to bubble to the `ParentView`, but does not reach the `GrandparentView` because `ParentView` returns false from its event handler.

You can use normal event bubbling techniques to implement familiar patterns. For example, you could implement a `FormView` that defines a `submit` method. Because the browser triggers the `submit` event when the user hits enter in a text field, defining a `submit` method on the form view will "just work".

```
App.FormView = Ember.View.extend({
  tagName: "form",

  submit: function(event) {
    // will be invoked whenever the user triggers
    // the browser's `submit` method
  }
});
```

```
{{#view App.FormView}}
  {{view Ember.TextField valueBinding="controller.firstName"}}
  {{view Ember.TextField valueBinding="controller.lastName"}}
  <button type="submit">Done</button>
{{/view}}
```

Adding New Events

Ember comes with built-in support for the following native browser events:

<table class=”figure”> <thead> <tr><th>Event Name</th><th>Method Name</th></tr> </thead>
<tbody> <tr><td>touchstart</td><td>touchStart</td></tr> <tr><td>touchmove</td><td>touchMove</td></tr>
<tr><td>touchend</td><td>touchEnd</td></tr> <tr><td>touchcancel</td><td>touchCancel</td></tr>
<tr><td>keydown</td><td>keyDown</td></tr> <tr><td>keyup</td><td>keyUp</td></tr> <tr><td>keypress</td><t
<tr><td>mousedown</td><td>mouseDown</td></tr> <tr><td>mouseup</td><td>mouseUp</td></tr>
<tr><td>contextmenu</td><td>contextMenu</td></tr> <tr><td>click</td><td>click</td></tr> <tr><td>dblclick</td
<tr><td>mousemove</td><td>mouseMove</td></tr> <tr><td>focusin</td><td>focusIn</td></tr> <tr><td>focusout
<tr><td>mouseenter</td><td>mouseEnter</td></tr> <tr><td>mouseleave</td><td>mouseLeave</td></tr>
<tr><td>submit</td><td>submit</td></tr> <tr><td>change</td><td>change</td></tr> <tr><td>dragstart</td><td>
<tr><td>drag</td><td>drag</td></tr> <tr><td>dragenter</td><td>dragEnter</td></tr> <tr><td>dragleave</td><td
<tr><td>dragover</td><td>dragOver</td></tr> <tr><td>drop</td><td>drop</td></tr> <tr><td>dragend</td><td>dr
</tbody> </table>

You can add additional events to the event dispatcher when you create a new application:

```
App = Ember.Application.create({
  customEvents: {
    // add support for the loadedmetadata media
    // player event
    'loadedmetadata': "loadedMetadata"
  }
});
```

In order for this to work for a custom event, the HTML5 spec must define the event as “bubbling”, or jQuery must have provided an event delegation shim for the event.

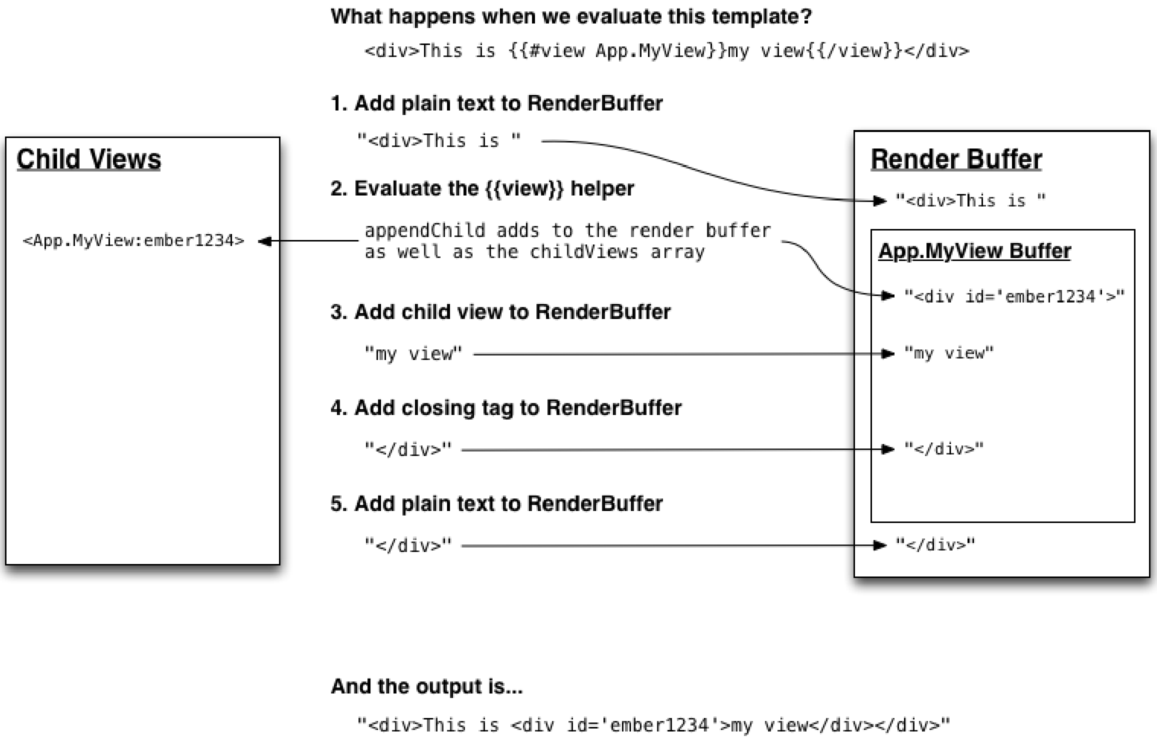
Templated Views

As you've seen so far in this guide, the majority of views that you will use in your application are backed by a template. When using templates, you do not need to programmatically create your view hierarchy because the template creates it for you.

While rendering, the view's template can append views to its child views array. Internally, the template's `{{view}}` helper calls the view's `appendChild` method.

Calling `appendChild` does two things:

1. Adds the child view to the `childViews` array.
2. Immediately renders the child view and adds it to the parent's render buffer.


What happens when we evaluate this template?
<div>This is {{#view App.MyView}}my view{{/view}}</div>
Child Views
<App.MyView:ember1234>
1. Add plain text to RenderBuffer
"<div>This is "
2. Evaluate the {{view}} helper
appendChild adds to the render buffer as well as the childViews array
3. Add child view to RenderBuffer
"my view"
4. Add closing tag to RenderBuffer
"</div>"
5. Add plain text to RenderBuffer
"</div>"
Render Buffer
"<div>This is "
App.MyView Buffer
"<div id='ember1234'>"
"my view"
"</div>"
"</div>"
And the output is...
"<div>This is <div id='ember1234'>my view</div></div>"

You may not call `appendChild` on a view after it has left the rendering state. A template renders "mixed content" (both views and plain text) so the parent view does not know exactly where to insert the new child view once the rendering process has completed.

In the example above, imagine trying to insert a new view inside of the parent view's `childViews` array. Should it go immediately after the closing `</div>` of `App.MyView`? Or should it go after the closing `</div>` of the entire view? There is no good answer that will always be correct.

Because of this ambiguity, the only way to create a view hierarchy using templates is via the `{{view}}` helper, which always inserts views in the right place relative to any plain text.

While this works for most situations, occasionally you may want to have direct, programmatic control of a view's children. In that case, you can use `Ember.ContainerView`, which explicitly exposes a public API for doing so.

Container Views

Container views contain no plain text. They are composed entirely of their child views (which may themselves be template-backed).

`ContainerView` exposes two public APIs for changing its contents:

1. A writable `childViews` array into which you can insert `Ember.View` instances.
2. A `currentView` property that, when set, inserts the new value into the child views array. If there was a previous value of `currentView`, it is removed from the `childViews` array.

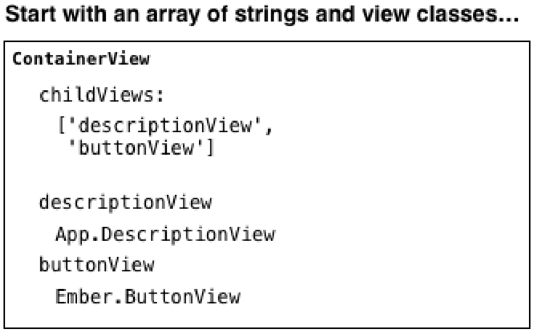
Here is an example of using the `childViews` API to create a view that starts with a hypothetical `DescriptionView` and can add a new button at any time by calling the `addButton` method:

```
App.ToolbarView = Ember.ContainerView.create({
  init: function() {
    var childViews = this.get('childViews');
    var descriptionView = App.DescriptionView.create();

    childViews.pushObject(descriptionView);
    this.addButton();

    return this._super();
  },

  addButton: function() {
    var childViews = this.get('childViews');
    var button = Ember.ButtonView.create();

    childViews.pushObject(button);
  }
});
```

As you can see in the example above, we initialize the `ContainerView` with two views, and can add additional views during runtime. There is a convenient shorthand for doing this view setup without having to override the `init` method:

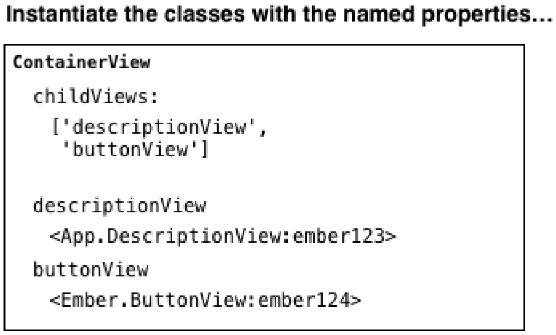
```
App.ToolbarView = Ember.ContainerView.create({
  childViews: ['descriptionView', 'buttonView'],

  descriptionView: App.DescriptionView,
  buttonView: Ember.ButtonView,

  addButton: function() {
    var childViews = this.get('childViews');
    var button = Ember.ButtonView.create();

    childViews.pushObject(button);
  }
});
```

As you can see above, when using this shorthand, you specify the `childViews` as an array of strings. At initialization time, each of the strings is used as a key to look up a view instance or class. That view is automatically instantiated, if necessary, and added to the `childViews` array.



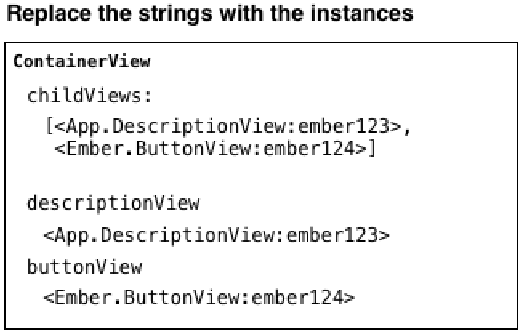
Start with an array of strings and view classes...

```
ContainerView
  childViews:
    ['descriptionView',
     'buttonView']

  descriptionView
    App.DescriptionView
  buttonView
    Ember.ButtonView
```



Instantiate the classes with the named properties...

```
ContainerView
  childViews:
    ['descriptionView',
     'buttonView']

  descriptionView
    <App.DescriptionView:ember123>
  buttonView
    <Ember.ButtonView:ember124>
```



Replace the strings with the instances

```
ContainerView
  childViews:
    [<App.DescriptionView:ember123>,
     <Ember.ButtonView:ember124>]

  descriptionView
    <App.DescriptionView:ember123>
  buttonView
    <Ember.ButtonView:ember124>
```

Template Scopes

Standard Handlebars templates have the concept of a *context*–the object from which expressions will be looked up.

Some helpers, like `{{#with}}`, change the context inside their block. Others, like `{{#if}}`, preserve the context. These are called "context-preserving helpers."

When a Handlebars template in an Ember app uses an expression (`{{#if foo.bar}}`), Ember will automatically set up an observer for that path on the current context.

If the object referenced by the path changes, Ember will automatically re-render the block with the appropriate context. In the case of a context-preserving helper, Ember will re-use the original context when re-rendering the block. Otherwise, Ember will use the new value of the path as the context.

```
{{#if controller.isAuthenticated}}
  <h1>Welcome {{controller.name}}</h1>
{{/if}}

{{#with controller.user}}
  <p>You have {{notificationCount}} notifications.</p>
{{/with}}
```

In the above template, when the `isAuthenticated` property changes from false to true, Ember will render the block, using the original outer scope as its context.

The `{{#with}}` helper changes the context of its block to the `user` property on the current controller. When the `user` property changes, Ember re-renders the block, using the new value of `controller.user` as its context.

View Scope

In addition to the Handlebars context, templates in Ember also have the notion of the current view. No matter what the current context is, the `view` property always references the closest view.

Note that the `view` property never references the internal views created for block expressions like `{{#if}}`. This allows you to differentiate between Handlebars contexts, which always work the way they do in vanilla Handlebars, and the view hierarchy.

Because `view` points to an `Ember.View` instance, you can access any properties on the view by using an expression like `view.propertyName`. You can get access to a view's parent using `view.parentView`.

For example, imagine you had a view with the following properties:

```
App.MenuItemView = Ember.View.create({
  templateName: 'menu_item_view',
  bulletText: '*'
});
```

…and the following template:

```
{{#with controller}}
  {{view.bulletText}} {{name}}
{{/with}}
```

Even though the Handlebars context has changed to the current controller, you can still access the view's `bulletText` by referencing `view.bulletText`.

Template Variables

So far in this guide, we've been handwaving around the use of the `controller` property in our Handlebars templates. Where does it come from?

Handlebars contexts in Ember can inherit variables from their parent contexts. Before Ember looks up a variable in the current context, it first checks in its template variables. As a template creates new Handlebars scope, they automatically inherit the variables from their parent scope.

Ember defines these `view` and `controller` variables, so they are always found first when an expression uses the `view` or `controller` names.

As described above, Ember sets the `view` variable on the Handlebars context whenever a template uses the `{{#view}}` helper. Initially, Ember sets the `view` variable to the view rendering the template.

Ember sets the `controller` variable on the Handlebars context whenever a rendered view has a `controller` property. If a view has no `controller` property, it inherits the `controller` variable from the most recent view with one.

Other Variables

Handlebars helpers in Ember may also specify variables. For example, the `{{#with controller.person as tom}}` form specifies a `tom` variable that descendent scopes can access. Even if a child context has a `tom` property, the `tom` variable will supersede it.

This form has one major benefit: it allows you to shorten long paths without losing access to the parent scope.

It is especially important in the `{{#each}}` helper, which provides the `{{#each person in people}}` form. In this form, descendent context have access to the `person` variable, but remain in the same scope as where the template invoked the `each`.

```
{{#with controller.preferences}}
  <h1>Title</h1>
  <ul>
  {{#each person in controller.people}}
    {{! prefix here is controller.preferences.prefix }}
    <li>{{prefix}}: {{person.fullName}}</li>
  {{/each}}
  <ul>
{{/with}}
```

Note that these variables inherit through `ContainerViews`, even though they are not part of the Handlebars context hierarchy.

Accessing Template Variables from Views

In most cases, you will need to access these template variables from inside your templates. In some unusual cases, you may want to access the variables in-scope from your view's JavaScript code.

You can do this by accessing the view's `templateVariables` property, which will return a JavaScript object containing the variables that were in scope when the view was rendered. `ContainerViews` also have access to this property, which references the template variables in the most recent template-backed view.

At present, you may not observe or bind a path containing `templateVariables`.

Managing Asynchrony

Many Ember concepts, like bindings and computed properties, are designed to help manage asynchronous behavior.

Without Ember

We'll start by taking a look at ways to manage asynchronous behavior using jQuery or event-based MVC frameworks.

Let's use the most common asynchronous behavior in a web application, making an Ajax request, as an example. The browser APIs for making Ajax requests provide an asynchronous API. jQuery's wrapper does as well:

```
jQuery.getJSON('/posts/1', function(post) {
  $("#post").html("<h1>" + post.title + "</h1>" +
    "<div>" + post.body + "</div>");
});
```

In a raw jQuery application, you would use this callback to make whatever changes you needed to make to the DOM.

When using an event-based MVC framework, you move the logic out of the callback and into model and view objects. This improves things, but doesn't get rid of the need to explicitly deal with asynchronous callbacks:

```
Post = Model.extend({
  author: function() {
    return [this.salutation, this.name].join(' ')
  },

  toJSON: function() {
    var json = Model.prototype.toJSON.call(this);
    json.author = this.author();
    return json;
  }
});

PostView = View.extend({
  init: function(model) {
    model.bind('change', this.render, this);
  },

  template: _.template("<h1><%= title %></h1><h2><%= author %></h2><div><%= body\
 %></div>"),

  render: function() {
    jQuery(this.element).html(this.template(this.model.toJSON());
    return this;
  }
});

var post = Post.create();
var postView = PostView.create({ model: post });
jQuery('#posts').append(postView.render().el);

jQuery.getJSON('/posts/1', function(json) {
  // set all of the JSON properties on the model
  post.set(json);
});
```

This example doesn't use any particular JavaScript library beyond jQuery, but its approach is typical of event-driven MVC frameworks. It helps organize the asynchronous events, but asynchronous behavior is still the core programming model.

Ember's Approach

In general, Ember's goal is to eliminate explicit forms of asynchronous behavior. As we'll see later, this gives Ember the ability to coalesce multiple events that have the same result.

It also provides a higher level of abstraction, eliminating the need to manually register and unregister event listeners to perform most common tasks.

You would normally use ember-data for this example, but let's see how you would model the above example using jQuery for Ajax in Ember.

```
App.Post = Ember.Object.extend({

});

App.PostController = Ember.ObjectController.extend({
  author: function() {
    return [this.get('salutation'), this.get('name')].join(' ');
  }.property('salutation', 'name')
});

App.PostView = Ember.View.extend({
  // the controller is the initial context for the template
  controller: null,
  template: Ember.Handlebars.compile("<h1>{{title}}</h1><h2>{{author}}</h2><div>\
{{body}}</div>")
});

var post = App.Post.create();
var postController = App.PostController.create({ model: post });

App.PostView.create({ controller: postController }).appendTo('body');

jQuery.getJSON("/posts/1", function(json) {
  post.setProperties(json);
});
```

In contrast to the above examples, the Ember approach eliminates the need to explicitly register an observer when the `post`'s properties change.

The `{{title}}`, `{{author}}` and `{{body}}` template elements are bound to those properties on the `PostController`. When the `PostController`'s model changes, it automatically propagates those changes to the DOM.

Using a computed property for `author` eliminated the need to explicitly invoke the computation in a callback when the underlying property changed.

Instead, Ember's binding system automatically follows the trail from the `salutation` and `name` set in the `getJSON` callback to the computed property in the `PostController` and all the way into the DOM.

Benefits

Because Ember is usually responsible for propagating changes, it can guarantee that a single change is only propagated one time in response to each user event.

Let's take another look at the author computed property.

```
App.PostController = Ember.ObjectController.extend({
  author: function() {
    return [this.get('salutation'), this.get('name')].join(' ');
  }.property('salutation', 'name')
});
```

Because we have specified that it depends on both salutation and name, changes to either of those two dependencies will invalidate the property, which will trigger an update to the {{author}} property in the DOM.

Imagine that in response to a user event, I do something like this:

```
post.set('salutation', "Mrs.");
post.set('name', "Katz");
```

You might imagine that these changes will cause the computed property to be invalidated twice, causing two updates to the DOM. And in fact, that is exactly what would happen when using an event-driven framework.

In Ember, the computed property will only recompute once, and the DOM will only update once.

How?

When you make a change to a property in Ember, it does not immediately propagate that change. Instead, it invalidates any dependent properties immediately, but queues the actual change to happen later.

Changing both the salutation and name properties invalidates the author property twice, but the queue is smart enough to coalesce those changes.

Once all of the event handlers for the current user event have finished, Ember flushes the queue, propagating the changes downward. In this case, that means that the invalidated author property will invalidate the {{author}} in the DOM, which will make a single request to recompute the information and update itself once.

This mechanism is fundamental to Ember. In Ember, you should always assume that the side-effects of a change you make will happen later. By making that assumption, you allow Ember to coalesce repetitions of the same side-effect into a single call.

In general, the goal of evented systems is to decouple the data manipulation from the side effects produced by listeners, so you shouldn't assume synchronous side effects even in a more event-focused system. The fact that side effects don't propagate immediately in Ember eliminates the temptation to cheat and accidentally couple code together that should be separate.

Side-Effect Callbacks

Since you can't rely on synchronous side-effects, you may be wondering how to make sure that certain actions happen at the right time.

For example, imagine that you have a view that contains a button, and you want to use jQuery UI to style the button. Since a view's `append` method, like everything else in Ember, defers its side-effects, how can you execute the jQuery UI code at the right time?

The answer is lifecycle callbacks.

```
App.Button = Ember.View.extend({
  tagName: 'button',
  template: Ember.Handlebars.compile("{{view.title}}"),

  didInsertElement: function() {
    this.$().button();
  }
});

var button = App.Button.create({
  title: "Hi jQuery UI!"
}).appendTo('#something');
```

In this case, as soon as the button actually appears in the DOM, Ember will trigger the `didInsertElement` callback, and you can do whatever work you want.

The lifecycle callbacks approach has several benefits, even if we didn't have to worry about deferred insertion.

First, relying on synchronous insertion means leaving it up to the caller of `appendTo` to trigger any behavior that needs to run immediately after appending. As your application grows, you may find that you create the same view in many places, and now need to worry about that concern everywhere.

The lifecycle callback eliminates the coupling between the code that instantiates the view and its post-append behavior. In general, we find that making it impossible to rely on synchronous side-effects leads to better design in general.

Second, because everything about the lifecycle of a view is inside the view itself, it is very easy for Ember to re-render parts of the DOM on-demand.

For example, if this button was inside of an `{{#if}}` block, and Ember needed to switch from the main branch to the `else` section, Ember can easily instantiate the view and call the lifecycle callbacks.

Because Ember forces you to define a fully-defined view, it can take control of creating and inserting views in appropriate situations.

This also means that all of the code for working with the DOM is in a few sanctioned parts of your application, so Ember has more freedom in the parts of the render process outside of these callbacks.

Observers

In some rare cases, you will want to perform certain behavior after a property's changes have propagated. As in the previous section, Ember provides a mechanism to hook into the property change notifications.

Let's go back to our salutation example.

```
App.PostController = Ember.ObjectController.extend({
  author: function() {
    return [this.get('salutation'), this.get('name')].join(' ');
  }.property('salutation', 'name')
});
```

If we want to be notified when the author changes, we can register an observer. Let's say that the view object wants to be notified:

```
App.PostView = Ember.View.extend({
  controller: null,
  template: Ember.Handlebars.compile("<h1>{{title}}</h1><h2>{{author}}</h2><div>\
{{body}}</div>"),

  authorDidChange: function() {
    alert("New author name: " + this.get('controller.author'));
  }.observes('controller.author')
});
```

Ember triggers observers after it successfully propagates the change. In this case, that means that Ember will only call the `authorDidChange` callback once in response to each user event, even if both of `salutation` and `name` changed.

This gives you the benefits of executing code after the property has changed, without forcing all property changes to be synchronous. This basically means that if you need to do some manual work in response to a change in a computed property, you get the same coalescing benefits as Ember's binding system.

Finally, you can also register observers manually, outside of an object definition:

```
App.PostView = Ember.View.extend({
  controller: null,
  template: Ember.Handlebars.compile("<h1>{{title}}</h1><h2>{{author}}</h2><div>\
{{body}}</div>"),

  didInsertElement: function() {
    this.addObserver('controller.author', function() {
      alert("New author name: " + this.get('controller.author'));
    });
  }
});
```

However, when you use the object definition syntax, Ember will automatically tear down the observers when the object is destroyed. For example, if an `{{#if}}` statement changes from truthy to falsy, Ember destroys all of the views defined inside the block. As part of that process, Ember also disconnects all bindings and inline observers.

If you define an observer manually, you need to make sure you remove it. In general, you will want to remove observers in the opposite callback to when you created it. In this case, you will want to remove the callback in `willDestroyElement`.

```
App.PostView = Ember.View.extend({
  controller: null,
  template: Ember.Handlebars.compile("<h1>{{title}}</h1><h2>{{author}}</h2><div>\
{{body}}</div>"),

  didInsertElement: function() {
    this.addObserver('controller.author', function() {
      alert("New author name: " + this.get('controller.author'));
    });
  },

  willDestroyElement: function() {
    this.removeObserver('controller.author');
  }
});
```

If you added the observer in the `init` method, you would want to tear it down in the `willDestroy` callback.

In general, you will very rarely want to register a manual observer in this way. Because of the memory management guarantees, we strongly recommend that you define your observers as part of the object definition if possible.

Routing

There's an entire page dedicated to managing async within the Ember Router: Asynchronous Routing[363]

Keeping Templates Up-to-Date

In order to know which part of your HTML to update when an underlying property changes, Handlebars will insert marker elements with a unique ID. If you look at your application while it's running, you might notice these extra elements:

```
My new car is
<script id="metamorph-0-start" type="text/x-placeholder"></script>
blue
<script id="metamorph-0-end" type="text/x-placeholder"></script>.
```

Because all Handlebars expressions are wrapped in these markers, make sure each HTML tag stays inside the same block. For example, you shouldn't do this:

```
{{! Don't do it! }}
<div {{#if isUrgent}}class="urgent"{{/if}}>
```

If you want to avoid your property output getting wrapped in these markers, use the `unbound` helper:

```
My new car is {{unbound color}}.
```

Your output will be free of markers, but be careful, because the output won't be automatically updated!

```
My new car is blue.
```

Debugging

Debugging Ember and Ember Data

Here are some tips you can use to help debug your Ember application.

Also, check out the ember-extension[364] project, which adds an Ember tab to Chrome DevTools that allows you to inspect Ember objects in your application.

[363] http://emberjs.com/guides/routing/asynchronous-routing

[364] https://github.com/tildeio/ember-extension

Routing

Log router transitions

```
window.App = Ember.Application.create({
  // Basic logging, e.g. "Transitioned into 'post'"
  LOG_TRANSITIONS: true,

  // Extremely detailed logging, highlighting every internal
  // step made while transitioning into a route, including
  // `beforeModel`, `model`, and `afterModel` hooks, and
  // information about redirects and aborted transitions
  LOG_TRANSITIONS_INTERNAL: true
});
```

View all registered routes

```
Ember.keys(App.Router.router.recognizer.names)
```

Get current route name / path

Ember installs the current route name and path on your app's `ApplicationController` as the properties `currentRouteName` and `currentPath`. `currentRouteName`'s value (e.g. `"comments.edit"`) can be used as the destination parameter of `transitionTo` and the `{{linkTo}}` Handlebars helper, while `currentPath` serves as a full descriptor of each parent route that has been entered (e.g. `"admin.posts.show.comments.edit"`).

```
// From within a Route
this.controllerFor("application").get("currentRouteName");
this.controllerFor("application").get("currentPath");

// From within a controller, after specifying `needs: ['application']`
this.get('controllers.application.currentRouteName');
this.get('controllers.application.currentPath');

// From the console:
App.__container__.lookup("controller:application").get("currentRouteName")
App.__container__.lookup("controller:application").get("currentPath")
```

Views / Templates

Log view lookups

```
window.App = Ember.Application.create({
  LOG_VIEW_LOOKUPS: true
});
```

Get the View object from its DOM Element's ID

```
Ember.View.views['ember605']
```

View all registered templates

```
Ember.keys(Ember.TEMPLATES)
```

Handlebars Debugging Helpers

```
{{debugger}}
{{log record}}
```

Controllers

Log generated controller

```
window.App = Ember.Application.create({
  LOG_ACTIVE_GENERATION: true
});
```

Ember Data

View ember-data's identity map

```
// all records in memory
App.__container__.lookup('store:main').recordCache

// attributes
App.__container__.lookup('store:main').recordCache[2].get('data.attributes')

// loaded associations
App.__container__.lookup('store:main').recordCache[2].get('comments')
```

Observers / Binding

See all observers for a object, key

```
Ember.observersFor(comments, keyName);
```

Log object bindings

```
Ember.LOG_BINDINGS = true
```

Miscellaneous

View an instance of something from the container

```
App.__container__.lookup("controller:posts")
App.__container__.lookup("route:application")
```

Dealing with deprecations

```
Ember.ENV.RAISE_ON_DEPRECATION = true
Ember.LOG_STACKTRACE_ON_DEPRECATION = true
```

Implement an Ember.onerror hook to log all errors in production

```
Ember.onerror = function(error) {
    Em.$.ajax('/error-notification', {
      type: 'POST',
      data: {
        stack: error.stack,
        otherInformation: 'exception message'
      }
    });
}
```

Import the console

If you are using imports with Ember, be sure to import the console:

```
Ember = {
  imports: {
    Handlebars: Handlebars,
    jQuery: $,
    console: window.console
  }
};
```

Errors within an `RSVP.Promise`

There are times when dealing with promises that it seems like any errors are being 'swallowed', and not properly raised. This makes it extremely difficult to track down where a given issue is coming from. Thankfully, `RSVP` has a solution for this problem built in.

You can provide an `onerror` function that will be called with the error details if any errors occur within your promise. This function can be anything but a common practice is to call `console.assert` to dump the error to the console.

```
Ember.RSVP.configure('onerror', function(error) {
  Ember.Logger.assert(false, error);
});
```

Errors within `Ember.run.later` (Backburner.js)

Backburner has support for stitching the stacktraces together so that you can track down where an erroring `Ember.run.later` is being initiated from. Unfortunately, this is quite slow and is not appropriate for production or even normal development.

To enable this mode you can set:

```
Ember.run.backburner.DEBUG = true;
```

The Run Loop

Ember's internals and most of the code you will write in your applications takes place in a run loop. The run loop is used to batch, and order (or reorder) work in a way that is most effective and efficient.

It does so by scheduling work on specific queues. These queues have a priority, and are processed to completion in priority order.

Why is this useful?

Very often, batching similar work has benefits. Web browsers do something quite similar by batching changes to the DOM.

Consider the following HTML snippet:

```
<div id="foo"></div>
<div id="bar"></div>
<div id="baz"></div>
```

and executing the following code:

```
foo.style.height = "500px" // write
foo.offsetHeight // read (recalculate style, layout, expensive!)

bar.style.height = "400px" // write
bar.offsetHeight // read (recalculate style, layout, expensive!)

baz.style.height = "200px" // write
baz.offsetHeight // read (recalculate style, layout, expensive!)
```

In this example, the sequence of code forced the browser to recalculate style, and relayout after each step. However, if we were able to batch similar jobs together, the browser would have only needed to recalulate the style and layout once.

```
foo.style.height = "500px" // write
bar.style.height = "400px" // write
baz.style.height = "200px" // write

foo.offsetHeight // read (recalculate style, layout, expensive!)
bar.offsetHeight // read (fast since style and layout is already known)
baz.offsetHeight // read (fast since style and layout is already known)
```

Interestingly, this pattern holds true for many other types of work. Essentially, batching similar work allows for better pipelining, and further optimization.

Let’s look at a similar example that is optimized in Ember, starting with a `User` object:

```
var User = Ember.Object.extend({
  firstName: null,
  lastName: null,
  fullName: function() {
    return this.get('firstName') + ' ' + this.get('lastName');
  }.property('firstName', 'lastName')
});
```

and a template to display its attributes:

```
{{firstName}}
{{fullName}}
```

If we execute the following code without the run loop:

```
var user = User.create({firstName:'Tom', lastName:'Huda'});
user.set('firstName', 'Yehuda');
// {{firstName}} and {{fullName}} are updated

user.set('lastName', 'Katz');
// {{lastName}} and {{fullName}} are updated
```

We see that the browser will rerender the template twice.

```
var user = User.create({firstName:'Tom', lastName:'Huda'});
user.set('firstName', 'Yehuda');
user.set('lastName', 'Katz');

// {{firstName}}  {{lastName}} and {{fullName}} are updated
```

However, if we have the run loop in the above code, the browser will only rerender the template once the attributes have all been set.

```
var user = User.create({firstName:'Tom', lastName:'Huda'});
user.set('firstName', 'Yehuda');
user.set('lastName', 'Katz');
user.set('firstName', 'Tom');
user.set('lastName', 'Huda');
```

In the above example with the run loop, since the user's attributes end up at the same values as before execution, the template will not even rerender!

It is of course possible to optimize these scenarios on a case-by-case basis, but getting them for free is much nicer. Using the run loop, we can apply these classes of optimizations not only for each scenario, but holistically app-wide.

How does the Run Loop work in Ember?

As mentioned earlier, we schedule work (in the form of function invocations) on queues, and these queues are processed to completion in priority order.

What are the queues, and what is their priority order?

```
Ember.run.queues
// => ["sync", "actions", "routerTransitions", "render", "afterRender", "destroy\
"]
```

Because the priority is first to last, the “sync” queue has higher priority than the “render” or “destroy” queue.

What happens in these queues?

- The `sync` queue contains binding synchronization jobs
- The `actions` queue is the general work queue and will typically contain scheduled tasks e.g. promises
- The `routerTransitions` queue contains transition jobs in the router
- The `render` queue contains jobs meant for rendering, these will typically update the DOM
- The `afterRender` contains jobs meant to be run after all previously scheduled render tasks are complete. This is often good for 3rd-party DOM manipulation libraries, that should only be run after an entire tree of DOM has been updated
- The `destroy` queue contains jobs to finish the teardown of objects other jobs have scheduled to destroy

In what order are jobs executed on the queues?

The algorithm works this way:

1. Let the highest priority queue with pending jobs be: `CURRENT_QUEUE`, if there are no queues with pending jobs the run loop is complete
2. Let a new temporary queue be defined as `WORK_QUEUE`
3. Move jobs from `CURRENT_QUEUE` into `WORK_QUEUE`
4. Process all the jobs sequentially in `WORK_QUEUE`
5. Return to Step 1

An example of the internals

Rather than writing the higher level app code that internally invokes the various run loop scheduling functions, we have stripped away the covers, and shown the raw run-loop interactions.

Working with this API directly is not common in most Ember apps, but understanding this example will help you to understand the run-loops algorithm, which will make you a better Ember developer.

http://emberjs.com.s3.amazonaws.com/run-loop-guide/index.html

FAQs

What do I need to know to get started with Ember?

For basic Ember app development scenarios, nothing. All common paths are paved nicely for you and don’t require working with the run loop directly.

What do I need to know to actually build an app?

It is possible to build good apps without working with the run loop directly, so if you don’t feel the need to do so, don’t.

What scenarios will require me to understand the run loop?

The most common case you will run into is integrating with a non-Ember API that includes some sort of asynchronous callback. For example:

- AJAX callbacks
- DOM update and event callbacks
- Websocket callbacks
- `setTimeout` and `setInterval` callbacks
- `postMessage` and `messageChannel` event handlers

You should begin a run loop when the callback fires.

How do I tell Ember to start a run loop?

```
$('a').click(function(){
  Ember.run(function(){  // begin loop
    // Code that results in jobs being scheduled goes here
  }); // end loop, jobs are flushed and executed
});
```

What happens if I forget to start a run loop in an async handler?

As mentioned above, you should wrap any non-Ember async callbacks in `Ember.run`. If you don’t, Ember will try to approximate a beginning and end for you. Here is some pseudocode to describe what happens:

```
$('a').click(function(){
  // Ember or runloop related code.
  Ember.run.start();

  // 1. we detect you need a run-loop
  // 2. we start one for you, but we don't really know when it ends, so we guess

  nextTick(function() {
    Ember.run.end()
  }, 0);
});
```

This is suboptimal because the current JS frame is allowed to end before the run loop is flushed, which sometimes means the browser will take the opportunity to do other things, like garbage collection. GC running in between data changing and DOM rerendering can cause visual lag and should be minimized.

When I am in testing mode, why are run-loop autoruns disabled?

Some of Ember's test helpers are promises that wait for the run loop to empty before resolving. This leads to resolving too early if there is code that is outside the run loop and gives erroneous test failures. Disabling autoruns help you identify these scenarios and helps both your testing and your application!

Contributing To Ember.js

Adding New Features

In general, new feature development should be done on master.

Bugfixes should not introduce new APIs or break existing APIs, and do not need feature flags.

Features can introduce new APIs, and need feature flags. They should not be applied to the release or beta branches, since SemVer requires bumping the minor version to introduce new features.

Security fixes should not introduce new APIs, but may, if strictly necessary, break existing APIs. Such breakages should be as limited as possible.

Bug Fixes

Urgent Bug Fixes

Urgent bugfixes are bugfixes that need to be applied to the existing release branch. If possible, they should be made on master and prefixed with [BUGFIX release].

Beta Bug Fixes

Beta bugfixes are bugfixes that need to be applied to the beta branch. If possible, they should be made on master and tagged with [BUGFIX beta].

Security Fixes

Security fixes need to be applied to the beta branch, the current release branch, and the previous tag. If possible, they should be made on master and tagged with [SECURITY].

Features

Features must always be wrapped in a feature flag. Tests for the feature must also be wrapped in a feature flag.

Because the build-tools will process feature-flags, flags must use precisely this format. We are choosing conditionals rather than a block form because functions change the surrounding scope and may introduce problems with early return.

```
if (Ember.FEATURES.isEnabled("feature")) {
  // implementation
}
```

Tests will always run with all features on, so make sure that any tests for the feature are passing against the current state of the feature.

Commits

Commits related to a specific feature should include a prefix like [FEATURE htmlbars]. This will allow us to quickly identify all commits for a specific feature in the future. Features will never be applied to beta or release branches. Once a beta or release branch has been cut, it contains all of the new features it will ever have.

If a feature has made it into beta or release, and you make a commit to master that fixes a bug in the feature, treat it like a bugfix as described above.

Feature Naming Conventions

```
Ember.FEATURES["<packageName>-<feature>"] // if package specific
Ember.FEATURES["container-factory-injections"]
Ember.FEATURES["htmlbars"]
```

Builds

The Canary build, which is based off master, will include all features, guarded by the conditionals in the original source. This means that users of the canary build can enable whatever features they want by enabling them before creating their Ember.Application.

```
Ember.FEATURES["htmlbars"] = true;
```

features.json

The root of the repository will contain a features.json file, which will contain a list of features that should be enabled for beta or release builds.

This file is populated when branching, and may not gain additional features after the original branch. It may remove features.

```
{
  "htmlbars": true
}
```

The build process will remove any features not included in the list, and remove the conditionals for features in the list.

Travis Testing

For a new PR:

1. Travis will test against master with all feature flags on.
2. If a commit is tagged with [BUGFIX beta], Travis will also cherry-pick the commit into beta, and run the tests on that branch. If the commit doesn't apply cleanly or the tests fail, the tests will fail.
3. If a commit is tagged with [BUGFIX release], Travis will also cherry-pick the commit into release, and run the test on that branch. If the commit doesn't apply cleanly or the tests fail, the tests will fail.

For a new commit to master:

1. Travis will run the tests as described above.
2. If the build passes, Travis will cherry-pick the commits into the appropriate branches.

The idea is that new commits should be submitted as PRs to ensure they apply cleanly, and once the merge button is pressed, Travis will apply them to the right branches.

Go/No-Go Process

Every six weeks, the core team goes through the following process.

Beta Branch

All remaining features on the beta branch are vetted for readiness. If any feature isn't ready, it is removed from features.json.

Once this is done, the beta branch is tagged and merged into release.

Master Branch

All features on the master branch are vetted for readiness. In order for a feature to be considered "ready" at this stage, it must be ready as-is with no blockers. Features are a no-go even if they are close and additional work on the beta branch would make it ready.

Because this process happens every six weeks, there will be another opportunity for a feature to make it soon enough.

Once this is done, the master branch is merged into beta. A `features.json` file is added with the features that are ready.

Beta Releases

Every week, we repeat the Go/No-Go process for the features that remain on the beta branch. Any feature that has become unready is removed from the features.json.

Once this is done, a Beta release is tagged and pushed.

Repositories

Ember is made up of several libraries. If you wish to add a feature or fix a bug please file a pull request against the appropriate repository. Be sure to check the libraries listed below before making changes in the Ember.js repository.

Main Repositories

Ember.js - The main repository for Ember.

- https://github.com/emberjs/ember.js[365]

Ember Data - A data persistence library for Ember.js.

- https://github.com/emberjs/data[366]

Ember Website - Source for http://www.emberjs.com[367] including these guides.

- https://github.com/emberjs/website[368]

[365] https://github.com/emberjs/ember.js
[366] https://github.com/emberjs/data
[367] http://www.emberjs.com
[368] https://github.com/emberjs/website

Libraries Used By Ember

These libraries are part of the Ember.js source, but development of them takes place in a seperate repository.

`packages/ember-metal/lib/vendor/backburner.js`

- **backburner.js** - Implements the Ember run loop.
- https://github.com/ebryn/backburner.js[369]

`packages/ember-routing/lib/vendor/route-recognizer.js`

- **route-recognizer.js** - A lightweight JavaScript library that matches paths against registered routes.
- https://github.com/tildeio/route-recognizer[370]

`packages/ember-routing/lib/vendor/router.js`

- **router.js** - A lightweight JavaScript library that builds on route-recognizer and rsvp to provide an API for handling routes.
- https://github.com/tildeio/router.js[371]

`packages/metamorph`

- **Metamorph.js** - Used by Ember for databinding handlebars templates
- https://github.com/tomhuda/metamorph.js[372]

`packages/rsvp`

- **RSVP.js** - Implementation of the of Promises/A+ spec used by Ember.
- https://github.com/tildeio/rsvp.js[373]

[369] https://github.com/ebryn/backburner.js

[370] https://github.com/tildeio/route-recognizer

[371] https://github.com/tildeio/router.js

[372] https://github.com/tomhuda/metamorph.js

[373] https://github.com/tildeio/rsvp.js