COEN 168/268

Mobile Web Application Development

Testing Ember

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University

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

http://emberjs.com/guides/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] 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 that combines its first and last name
- A post has a title

Testing Frameworks

[QUnit] is the default testing framework for this guide, but others are supported through third-party adapters.

Integration Tests

What Are Integration Tests?

- Integration tests are generally used to test important workflows within your application.
- They emulate user interaction and confirm expected results.

Setting Up Integration Tests

- 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.

Setting Up Integration Tests, Cont'd

```
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).

Setting Up Integration Tests, Cont'd

```
App.setupForTesting();
```

This injects the test helpers into the window's scope.

```
App.injectTestHelpers();
```

Setting Up Integration Tests, Cont'd

- 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.

Setting Up Integration Tests, Cont'd

```
1 module("Integration Tests", {
2   teardown: function() {
3     App.reset();
4   }
5 });
```

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

Web Apps Are Event Driven

- 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 an be executed in any order.

An Example

- Let's say a user clicks two buttons, one after another and both load data from different servers.
- They take different times to respond.
- Therefore, you have account for that in your tests.
- Ember provides helpers to perform: **asynchronous** and **synchronous** tests

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.

Asynchronous Helpers

- `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.

Asynchronous Helpers, Cont'd

- `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.

Asynchronous Helpers, Cont'd

- `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.

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.

Synchronous Helpers, Cont'd

- `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.

```
 1 test("simple test", function(){
 2   expect(1); // Ensure that we will perform one assertion
 3
 4   visit("/posts/new");
 5   fillIn("input.title", "My new post");
 6   click("button.submit");
 7
 8   // Wait for asynchronous helpers above to complete
 9   andThen(function() {
10     equal(find("ul.posts li:last").text(), "My new post");
11   });
12 });
```

What Happened In This Example?

- 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

What Happened In This Example?

- 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.

What is the difference?

- 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:

```
1 Ember.Test.registerHelper('shouldHaveElementWithCount',
2   function(app, selector, n, context) {
3     var el = findWithAssert(selector, context);
4     var count = el.length;
5     equal(n, count, 'found ' + count + ' times');
6   }
7 );
8
9 // shouldHaveElementWithCount("ul li", 3);
```

Here is an example of an async helper:

```
1  Ember.Test.registerAsyncHelper('dblclick',
2    function(app, selector, context) {
3      var $el = findWithAssert(selector, context);
4      Ember.run(function() {
5        $el.dblclick();
6      });
7    }
8  );
9
10 // dblclick("#person-1")
```

Async helpers also come in handy when you want to group interaction into one helper. For example:

```
1 Ember.Test.registerAsyncHelper('addContact',
2   function(app, name, context) {
3     fillIn('#name', name);
4     click('button.create');
5   }
6 );
7
8 // addContact("Bob");
9 // addContact("Dan");
```

Testing User Interaction

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:

```
1 test('root lists first page of posts', function(){
2   visit('/posts');
3   andThen(function() {
4     equal(find('ul.posts li').length, 3, 'The first page should have 3 posts');
5   });
6 });
```

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');
  });
});
```

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.

```
1 App.ProfileRoute = Ember.Route.extend({
2   beforeModel: function() {
3     var user = this.modelFor('application');
4     if (Em.isEmpty(user)) {
5       this.transitionTo('login');
6     }
7   }
8 });
```

We could use the route helpers to ensure that the user would be redirected to the login page when the restricted URL is visited.

```
1  module('Integration: Transitions', {
2    teardown: function() {
3      App.reset();
4    }
5  });
6
7  test('redirect to login if not authenticated', function() {
8    visit('/');
9    click('.profile');
10
11   andThen(function() {
12     equal(currentRouteName(), 'login');
13     equal(currentPath(), 'login');
14     equal(currentURL(), '/login');
15   });
16 });
```

Unit Testing Basics

What Are Unit Tests?

- 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.

Testing `Ember.Object`

- As the basic object in Ember, testing `Ember.Object` sets 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.

Testing Computed Properties

Let's start by looking at an object that has a `computedFoo` computed property based on a `foo` property.

```
1 App.SomeThing = Ember.Object.extend({
2   foo: 'bar',
3   computedFoo: function(){
4     return 'computed ' + this.get('foo');
5   }.property('foo')
6 });
```

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.

```
1 module('Unit: SomeThing');
2
3 test('computedFoo correctly concats foo', function() {
4   var someThing = App.SomeThing.create();
5   someThing.set('foo', 'baz');
6   equal(someThing.get('computedFoo'), 'computed baz');
7 });
```

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).

```
1 App.SomeThing = Ember.Object.extend({
2   foo: 'bar',
3   testMethod: function() {
4     this.set('foo', 'baz');
5   }
6 });
```

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.

```
1 module('Unit: SomeThing');
2
3 test('calling testMethod updates foo', function() {
4   var someThing = App.SomeThing.create();
5   someThing.testMethod();
6   equal(someThing.get('foo'), 'baz');
7 });
```

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.

```
1 App.SomeThing = Ember.Object.extend({
2   count: 0,
3   calc: function() {
4     this.incrementProperty('count');
5     return 'count: ' + this.get('count');
6   }
7 });
```

The test would call the `calc` method and assert it gets back the correct value.

```
1 module('Unit: SomeThing');
2
3 test('testMethod returns incremented count', function() {
4   var someThing = App.SomeThing.create();
5   equal(someThing.calc(), 'count: 1');
6   equal(someThing.calc(), 'count: 2');
7 });
```

Testing Observers

Suppose we have an object that has an observable method based on the `foo` property.

```
1 App.SomeThing = Ember.Object.extend({
2   foo: 'bar',
3   other: 'no',
4   doSomething: function(){
5     this.set('other', 'yes');
6   }.observes('foo')
7 });
```

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.

```
1 module('Unit: SomeThing');
2
3 test('doSomething observer sets other prop', function() {
4   var someThing = App.SomeThing.create();
5   someThing.set('foo', 'baz');
6   equal(someThing.get('other'), 'yes');
7 });
```

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], [AMD], 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] is the default *unit* testing helper suite for Ember.
- Can and should be used as a template for other test helpers.
- It uses your application's resolver to find and automatically create test subjects for you using the `moduleFor` and `test` helpers.
- 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 test writer.

Available Helpers

By including `Ember-QUnit`, you will have access to a number of test helpers.

Let's go through them...

`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()`.

```
Ember.setupForTesting();
```

Unit Testing Setup

- 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 a Module-Based App

- 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 }))
```

If You Have A Custom Resolver

Otherwise, you would require the custom resolver and pass it to `setResolver` like this *(ES6 example)*:

```
1 import Resolver from './path/to/resolver';
2 import { setResolver } from 'ember-qunit';
3 setResolver(Resolver.create());
```

Testing Components

Setup

Before testing components, be sure to add testing application div to your testing html file:

```
1 <!-- as of time writing, ID attribute needs to be named exactly ember-testing -->
2 <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:

```
1 App.PrettyColorComponent = Ember.Component.extend({
2   classNames: ['pretty-color'],
3   attributeBindings: ['style'],
4   style: function() {
5     return 'color: ' + this.get('name') + ';';
6   }.property('name')
7 });
```

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:

```
1  test('changing colors', function(){
2
3    // this.subject() is available because we used moduleForComponent
4    var component = this.subject();
5
6    // we wrap this with Ember.run because it is an async function
7    Ember.run(function(){
8      component.set('name','red');
9    });
10
11   // first call to $() renders the component.
12   equal(this.$().attr('style'), 'color: red;');
13
14   // another async function, so we need to wrap it with Ember.run
15   Ember.run(function(){
16     component.set('name', 'green');
17   });
18
19   equal(this.$().attr('style'), 'color: green;');
20 });
```

Another test that we might perform on this component would be to ensure the template is being rendered properly.

```
1  test('template is rendered with the color name', function(){
2
3    // this.subject() is available because we used moduleForComponent
4    var component = this.subject();
5
6    // first call to $() renders the component.
7    equal($.trim(this.$().text()), 'Pretty Color:');
8
9    // we wrap this with Ember.run because it is an async function
10   Ember.run(function(){
11     component.set('name', 'green');
12   });
13
14   equal($.trim(this.$().text()), 'Pretty Color: green');
15 });
```

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] to interact with the rendered component:

```
1  moduleForComponent('my-foo', 'MyFooComponent');
2
3  test('clicking link updates the title', function() {
4    var component = this.subject();
5
6    // append the component to the DOM
7    this.append();
8
9    // assert default state
10   equal(find('h2').text(), 'Hello World');
11
12   // perform click action
13   click('button');
14
15   andThen(function() { // wait for async helpers to complete
16     equal(find('h2').text(), 'Hello Ember World');
17   });
18 });
```

Components with built in layout

- Some components do not use a separate template.
- The template can be embedded into the component via the [layout] 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.

```
1  moduleForComponent('my-foo', 'MyFooComponent');
2
3  test('clicking link updates the title', function() {
4    var component = this.subject();
5
6    // append the component to the DOM
7    this.append();
8
9    // assert default state
10   equal(find('h2').text(), "I'm a little teapot");
11
12   // perform click action
13   click('button');
14
15   andThen(function() { // wait for async helpers to complete
16     equal(find('h2').text(), "I'm a little embereño");
17   });
18 });
```

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:

Programmatically interacting with components

```
1  moduleForComponent('my-foo', 'MyFooComponent');
2
3  test('clicking link updates the title', function() {
4    var component = this.subject();
5
6    // append the component to the DOM, returns DOM instance
7    var $component = this.append();
8
9    // assert default state
10   equal($component.find('h2').text(), "I'm a little teapot");
11
12   // send action programmatically
13   Em.run(function(){
14     component.send('changeName');
15   });
16
17   equal($component.find('h2').text(), "I'm a little embereño");
18 });
```

`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');
});
```

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('height');
  }.property('width', 'height')
});
```

Usage of this component might look something like this:

```
1 {{#my-album title="Cats"}}
2   {{my-kitten width="200" height="300"}}
3   {{my-kitten width="100" height="100"}}
4   {{my-kitten width="50" height="50"}}
5 {{/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);
});
```

Testing Controller Actions

Here we have a controller `PostsController` with some computed properties and an action `setProps`.

```
1  App.PostsController = Ember.ArrayController.extend({
2
3    propA: 'You need to write tests',
4    propB: 'And write one for me too',
5
6    setPropB: function(str) {
7      this.set('propB', str);
8    },
9
10   actions: {
11     setProps: function(str) {
12       this.set('propA', 'Testing is cool');
13       this.setPropB(str);
14     }
15   }
16 });
```

`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:

```
1 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`.

```
1  test('calling the action setProps updates props A and B', function() {
2    expect(4);
3
4    // get the controller instance
5    var ctrl = this.subject();
6
7    // check the properties before the action is triggered
8    equal(ctrl.get('propA'), 'You need to write tests');
9    equal(ctrl.get('propB'), 'And write one for me too');
10
11   // trigger the action on the controller by using the `send` method,
12   // passing in any params that our action may be expecting
13   ctrl.send('setProps', 'Testing Rocks!');
14
15   // finally we assert that our values have been updated
16   // by triggering our action.
17   equal(ctrl.get('propA'), 'Testing is cool');
18   equal(ctrl.get('propB'), 'Testing Rocks!');
19 });
```

Testing Controller Needs

- Sometimes controllers have dependencies on other controllers.
- This is accomplished by using [needs].

For example, here are two simple controllers. The `PostController` is a dependency of the `CommentsController`:

```
1 App.PostController = Ember.ObjectController.extend({
2   // ...
3 });
4
5 App.CommentsController = Ember.ArrayController.extend({
6   needs: 'post',
7   title: Ember.computed.alias('controllers.post.title'),
8 });
```

This time when we setup our `moduleFor` we need to pass an options object as our third argument that has the controller's `needs`.

```
1 moduleFor('controller:comments', 'Comments Controller', {
2   needs: ['controller:post']
3 });
```

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');

  });
});
```

Testing Routes

Testing Routes

- 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:
  - 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);

  // 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);
});
```

Testing Models

Testing Models

- [Ember Data] 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`:

```
1  moduleForModel('player', 'Player Model');
2
3  test('levelUp', function() {
4    // this.subject aliases the createRecord method on the model
5    var player = this.subject({ level: 4 });
6
7    // wrap asynchronous call in run loop
8    Ember.run(function() {
9      player.levelUp();
10   });
11
12   equal(player.get('level'), 5);
13   equal(player.get('levelName'), 'Professional');
14 });
```

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`.

```
1 App.Profile = DS.Model.extend({});
2
3 App.User = DS.Model.extend({
4   profile: DS.belongsTo(App.Profile)
5 });
```

Then you could test that the relationship is wired up correctly with this test.

```
moduleForModel('user', 'User Model', {
  needs: ['model:profile']
});

test('profile relationship', function() {
  var relationships = Ember.get(App.User, 'relationships');
  deepEqual(relationships.get(App.Profile), [
    { name: 'profile', kind: 'belongsTo' }
  ]);
});
```

Automating Tests With Runners

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.

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)

Next, create an HTML file that includes qunit and it's css that looks like the following example.

```
<!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, you are done!

Manually Refreshing the Browser Is Tedious

- 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 work in every browser you 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.

Test Runners to The Rescue!

- 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.

We Won't Cover Them In Lecture

Go to `http://emberjs.com/guides/testing/test-runners/` for more info

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

http://emberjs.com/guides/testing/

COEN 168/268

Mobile Web Application Development

Testing Ember

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University