v2.3.0

Old Guides - You are viewing the guides for Ember v2.3.0. **VIEW v2.11.0**

Acceptance Tests

To create an acceptance test, run `ember generate acceptance-test <name>`. For example:

```
ember g acceptance-test login
```

This generates this file:

tests/acceptance/login-test.js JS

```js
import { test } from 'qunit';
import moduleForAcceptance from 'people/tests/helpers/module-for-acceptanc

moduleForAcceptance('Acceptance | login');

test('visiting /login', function(assert) {
  visit('/login');

  andThen(function() {
    assert.equal(currentURL(), '/login');
  });
});
```

`moduleForAcceptance` deals with application setup and teardown. The last few lines, within the function `test`, contain an example test.

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.

For example:

tests/acceptance/new-post-appears-first-test.js JS

```js
test('should add new post', function(assert) {
  visit('/posts/new');
  fillIn('input.title', 'My new post');
  click('button.submit');
  andThen(() => assert.equal(find('ul.posts li:first').text(), 'My new post
});
```

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 (i.e. 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 acceptance 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. 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.

- `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.
- `fillIn(selector, value)`
  - Fills in the selected input with the given value and returns a promise that fulfills when all resulting async behavior is complete. Works with `<select>` elements as well as `<input>` elements. Keep in mind that with `<select>` elements, `value` must be set to the *value* of the `<option>` tag, rather than its *content* (for example, `true` rather than `"Yes"`).
- `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.
- `visit(url)`
  - Visits the given route and returns a promise that fulfills when all resulting async behavior is complete.

Synchronous Helpers

Synchronous helpers are performed immediately when triggered.

- `currentPath()`
  - Returns the current path.
- `currentRouteName()`

- Returns the currently active route name.

- `currentURL()`
  - Returns the current URL.
- `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.

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.

tests/acceptance/new-post-appears-first-test.js JS

```
test('should add new post', function(assert) {
  visit('/posts/new');
  fillIn('input.title', 'My new post');
  click('button.submit');
  andThen(() => assert.equal(find('ul.posts li:first').text(), 'My new post
});
```

First we 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 `assert.equal` which makes an assertion that the text found in the first li of the ul whose class is "posts" is equal to "My new post".

Custom Test Helpers

For creating your own test helper, run `ember generate test-helper <helper-name>`. Here is the result of running `ember g test-helper shouldHaveElementWithCount`:

tests/helpers/should-have-element-with-count.js JS

```js
export default Ember.Test.registerAsyncHelper(
    'shouldHaveElementWithCount', function(app) {
});
```

`Ember.Test.registerAsyncHelper` and `Ember.Test.registerHelper` are used to register test helpers that will be injected when `startApp` 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. Other parameters, such as assert, need to be provided when calling the helper. Helpers need to be registered prior to calling `startApp`, but ember-cli will take care of it for you.

Here is an example of a non-async helper:

tests/helpers/should-have-element-with-count.js JS

```js
export default Ember.Test.registerHelper('shouldHaveElementWithCount',
  function(app, assert, selector, n, context) {
    const el = findWithAssert(selector, context);
    const count = el.length;
    assert.equal(n, count, `found ${count} times`);
  }
);

// shouldHaveElementWithCount(assert, 'ul li', 3);
```

Here is an example of an async helper:

tests/helpers/dblclick.js JS

```js
export default Ember.Test.registerAsyncHelper('dblclick',
  function(app, assert, selector, context) {
    let $el = findWithAssert(selector, context);
    Ember.run(() => $el.dblclick());
  }
);

// dblclick(assert, '#person-1')
```

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

tests/helpers/add-contact.js JS

```js
export default Ember.Test.registerAsyncHelper('addContact',
  function(app, name) {
    fillIn('#name', name);
    click('button.create');
  }
);

// addContact('Bob');
// addContact('Dan');
```

Finally, don't forget to add your helpers in `tests/.jshintrc` and in `tests/helpers/start-app.js`. In `tests/.jshintrc` you need to add it in the `predef` section, otherwise you will get failing jshint tests:

tests/.jshintc

```
{
  "predef": [
    "document",
    "window",
    "location",
    ...
    "shouldHaveElementWithCount",
    "dblclick",
    "addContact"
  ],
  ...
}
```

In `tests/helpers/start-app.js` you need to import the helper file: it will be registered then.

tests/helpers/start-app.js JS

```js
import Ember from 'ember';
import Application from '../../app';
import Router from '../../router';
import config from '../../config/environment';
import './should-have-element-with-count';
import './dblclick';
import './add-contact';
```

‹ Introduction

Unit Testing Basics ›