MAY 2, 2015

Acceptance testing in Ember-CLI, an introduction.

The only form of tests you can't really live without are acceptance tests, they always ensure that requirements are met, and almost always expose flaws.

Today we will (try to!?) cover the basics of acceptance testing in Ember-CLI Apps, Involving all parts of the stack but the back-end.

For operations that require interaction with the server, we'll use *Pretender* + Mirage to mock our back-end on a per-test basis, although it also supports having the same responses for both development and test environments.

We will be testing a form to create homework:

- Inputs for a title, a description and a year associated to it.
- On success, transitions to show the homework that was just created.

Let's start by generating an acceptance test inside a new dummy app:

`$ ember g acceptance-test create-homework`

And add a new test, checking for the presence of a homework that has inputs for both a title and a description:

```
// tests/acceptance/test-name.js
test('Creating a new homework', function(assert) {
  visit('/homeworks/new');

  const title = 'Underwater basket weaving';
  const description = 'an amazing the description about basket weaving';

  fillIn('.homework-title', title);
  fillIn('.homework-description', description);

  click('.submit-homework');
  andThen(function() {
    assert.equal(currentURL(), '/homeworks/1', 'creates a homework and
redirects to show page');
  });
});
```

NOTE: In case you're wondering why I'm not prepending `input` tag is because they might not always be one, in the future they could be any kind of tag, or maybe even a component. It's *considered a better practice to other selectors*

This will yield into a failing output:

We'll follow a very railsy approach to this, and generate a new resource and the appropriate route

```
$ ember g resource homework
```

```
$ ember g route homeworks/new
```

We're still in red now. However the error message's changed:

Now's the time to add more markup and *Emberify* template. Let's generate a model with the attributes `title` and `description`, set a new instance on the route's model hook + add the markup to tie all things together.

```
// app/models/homework.js
import DS from 'ember-data';
```

```
export default DS.Model.extend({
  title: DS.attr('string'),
  description: DS.attr('string')
});
```

```
// app/routes/homeworks/new.js
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    return this.store.createRecord('homework');
  }
});
```

```
{{!-- app/templates/homeworks/new.hbs --}}
<form role="form" {{action 'saveRecord' model on="submit"}}>
  <label for="title">Title</label>
  {{input class="homework-title" value=model.title}}
  <br>
  <label for="description">Description</label>
  {{input class="homework-description" value=model.description}}
  <br>
  <input type="submit" class="submit-homework">
</form>
```

Clicking the submit button will trigger the action `saveRecord` which will go through the route, passing the `model` as an argument. We're not handling this action at any level, so running our suite again will complain with:

It's time to implement this action in our route, for the purposes of this post let's only cover the *happy path* and implement a very naive action that will commit to the server without checking anything.

```
  actions: {
    saveRecord(homework) {
      homework.save().then(hw => this.transitionTo('homeworks.show',
hw));
    }
  }
```

NOTE: The syntax on line 3 might not seem familiar to everyone, I'm leveraging the fact that Ember-CLI uses Babel as a transpiler, giving developers the power to leverage the new features ES6 has to offer, like fat arrow functions

`instance.save()` will attempt to `POST` to the server in order to create a new resource, however in our local test environment it's not efficient nor fast to make real requests to the server, after setting it up, defining both the factory and the fixtures we can declare the routes under `app/mirage/config.js`

```
export default function() {
  this.post('/homeworks');
}
```

NOTE: Don't forget to visit Mirage's docs for better reference. By default declaring a route without passing in a callback will execute some default behaviour based on the verb. For `POST` it will push the given payload to the appropriate collection and return a 201.

Then we'll get the last error you'll see in this tutorial, complaining about the absence of the route `homeworks.show`

Now we generate that missing route, specifying a path:

`ember g route homeworks/show --path=:id`

And we'll see our tests going green! As you can see Ember-CLI provides a very robust project structure, alongside some really nice generators that ensure a very well-defined workflow.

I’m personally a fan of a test-first approach, always starting with the most superficial parts of the stack, although unit tests are always a good thing, specially when you need to validate the behaviour of a custom algorithm or similar.

Anything you’d change on this post? Contact me

75 KUDOS

@MarioGintili

SVBTLE

Terms • Privacy • Promise