8
Testing AJAX

In this book, we have tested the filling of text fields on a form, clicking on buttons, and the resulting HTML document. This makes us ready to test a traditional form-based request-response application, but typical modern applications are usually more complex than that as they make use of asynchronous HTTP requests, that somehow update the document without having to refresh it. This is because they use AJAX.

Our application emits AJAX requests when presented with the to-do item list page; a user can drag an item and drop it in the new position. The code that we placed in the `public/js/todos.js` file catches the change and calls the server `/todos/sort` URL, changing the item order in the database.

Let's see how we can use Zombie to test this drag-and-drop feature. The topics covered in this chapter include:

- Using Zombie to trigger an AJAX call
- Using Zombie to test the result of an AJAX call

By the end of this section, you will know how to use Zombie to test an application that uses AJAX.

Inflicting drag-and-drop

Let's add some tests to the `test/todos.js` file.

1. We start off by adding a new describe scope before the end of the `Todo list` scope:

   ```
   describe('When there are some items on the list', function() {
   ```

   This new scope allows us to setup a to-do item fixture list in the database before any test inside this scope is run.

2. Now, let's add this new `beforeEach` hook inside the new scope:

```
beforeEach(function(done) {
  // insert todo items
  db.insert(fixtures.todos, fixtures.user.email, done);
});
```

3. Then we start the test by logging in:

```
it('should allow me to reorder items using drag and drop',
  login(function(browser, done) {
```

4. We start the test by making sure that we have three to-do items in our item list page:

```
var items = browser.queryAll('#todo-list tr');
assert.equal(items.length, 3, 'Should have 3 items and has ' +
  items.length);
```

5. Then we declare a helper function that will assist us in verifying the contents of that list:

```
function expectOrder(order) {
  var itemTexts = browser.queryAll('#todo-list tr .what').map(
    function(node) {
      return node.textContent.trim();
    }   assert.equal(index + 1, itemPos);
  });
}
```

   This function gets an array of strings and asserts that the `what` and `pos` fields of each to-do item in the page are placed in the expected order.

6. Then we use this new `expectOrder` function to actually test that the order is the expected one:

```
expectOrder(['Do the laundry', 'Call mom', 'Go to gym']);
```

   As you may remember, this is the order of the to-do items as declared in the `test/fixtures.json` file that were loaded on the `beforeEach` hook.

7. Next we create another helper function that will help us fabricate and inject mouse events:

```
function mouseEvent(name, target, x, y) {
  var event = browser.document.createEvent('MouseEvents');
  event.initEvent(name, true, true);
  event.clientX = event.screenX = x;
  event.clientY = event.screenY = y;
  event.which = 1;
  browser.dispatchEvent(item, event);
}
```

   This function simulates a user mouse event, sets the `x` and `y` coordinates on it, sets the mouse button (`event.which = 1`), and dispatches the event into the browser, specifying which item the event happened on.

8. Next we select which to-do item we will be dragging; in this case, we drag the first one:

```
var item = items[0];
```

9. Then we use the `mouseEvent` helper function to inject a sequence of fabricated events:

```
mouseEvent('mousedown', item, 50, 50);
mouseEvent('mousemove', browser.document, 51, 51);
mouseEvent('mousemove', browser.document, 51, 150);
mouseEvent('mouseup',  browser.document, 51, 150);
```

   There are several important aspects to these events, namely, the sequence of events, the target element, and the mouse coordinates. Let's analyze them.

   These are the events that compose a drag and a drop. First we press the mouse button, we move it a bit, then we move it some more and finally we release the mouse button. The `x` and `y` values for the mouse event location we're using here aren't really important, what is important is the relative difference between them so that the drag is detected and the drag mode begins.

On the first event, the `mousedown`, we're using an arbitrary coordinate of `50, 50`. On the second event, the `mousemove`, we're incrementing this coordinate by one pixel; this starts the drag.

The second `mousemove` event continues the drag on the y axis. It looks superfluous and redundant, but it's required so that the drag detection works, giving continuity to the drag movement we were performing.

Finally we have the `mouseup` where the user releases the mouse. This event uses the same coordinates as the previous `mousemove`, indicating that the user dropped the element after the drag.

Let's now analyze the target elements in the events:

The second argument of the `mouseEvent()` helper function takes the target element. In the first `mousedown` event injection, we're targeting the to-do item in the `item` variable, which refers the item we want to drag. This indicates which item we will be dragging, once the drag mode gets activated. The remaining three events target the browser document, since the user will be dragging the to-do item across the document.

Some further clarification of the mouse coordinates we're using:

Zombie does not render the items, so it doesn't know the location of each of them. This is the only way we can use to indicate which element we are dragging. The x and y coordinates in this case are irrelevant for that.

Since Zombie doesn't render the elements, it doesn't keep the location of each element. In fact, they are all placed at (0, 0), which means that our `mouseup` event placed the dragged item after the last item.

As mentioned earlier, the initial value and the drag distance is completely arbitrary, and you will find that changing these will still make the test work.

10. After injecting these mouse events into the browser event queue, we wait for these to be fully processed using `browser.wait()` function:

```
browser.wait(function(err) {
            if (err) throw err;
```

    At this stage, the browser has changed the element order and made an AJAX request posting the new order to the server.

11. Now we verify that the to-do items are in the new order:

```
expectOrder(['Call mom', 'Go to gym', 'Do the laundry']);
```

12. We also verify whether the browser performed the HTTP request we intended:

```
var lastRequest = browser.lastRequest;
assert.equal(lastRequest.url, 'http://localhost:3000/todos/sort');
assert.equal(lastRequest.method, 'POST');
```

Notice that we're using the `browser.lastRequest()` function to access the last AJAX request the browser made.

If you needed to access every HTTP request that the browser made, you can inspect the `browser.resources` object.

Now that we know that the browser made an `HTTP POST` request commanding that the server sorts the to-do items, we need to make sure the to-do items were correctly updated in the database. To verify this we do something similar to what a human tester would; we reload the page using `browser.reload()` and verify to see if the order is indeed the expected one:

```
browser.reload(function(err) {
  if (err) throw err;

  expectOrder(['Call mom', 'Go to gym', 'Do the laundry']);

  done();

});
```

Summary

Using Zombie you can inject custom events to imitate some complex user actions. You can also detect what URL and method the browser performed an HTTP request to by using `browser.lastRequest()`.