Testing

Introduction

Testing allows you to ensure your application works the way you think it does, especially as your codebase changes over time. If you have good tests, you can refactor and rewrite code with confidence. Tests are also the most concrete form of documentation of expected behavior, since other developers can figure out how to use your code by reading the tests.

Automated testing is critical because it allows you to run a far greater set of tests much more often than you could manually, allowing you to catch regression errors immediately.

Types of tests

Entire books have been written on the subject of testing, so we will touch on some basics of testing here. The important thing to consider when writing a test is what part of the application you are trying to test, and how you are verifying the behavior works.

- **Unit test**: If you are testing one small module of your application, you are writing a unit test. You'll need to *stub* and *mock* other modules that your module usually leverages in order to *isolate* each test. You'll typically also need to *spy* on actions that the module takes to verify that they occur.
- **Integration test**: If you are testing that multiple modules behave properly in concert, you are writing an integration test. Such tests are much more complex and may require running code both on the client and on the server to verify that communication across that divide is working as expected. Typically an integration test will still isolate a part of the entire application and directly verify results in code.
- **Acceptance test**: If you want to write a test that can be run against any running version of your app and verifies at the browser level that the right things happen when you push the right buttons, then you are writing an acceptance test (sometimes called "end to end test"). Such tests typically try to hook into the application as little as possible, beyond perhaps setting up the right data to run a test against.
- **Load test**: Finally you may wish to test that your application works under typical load or see how much load it can handle before it falls over.

This is called a load test or stress test. Such tests can be challenging to set up and typically aren't run often but are very important for confidence before a big production launch.

Challenges of testing in Meteor

In most ways, testing a Meteor app is no different from testing any other full stack JavaScript application. However, compared to more traditional backend or front-end focused frameworks, two factors can make testing a little more challenging:

- **Client/server data**: Meteor's data system makes it possible to bridge the client-server gap and often allows you to build your application without thinking about how data moves around. It becomes critical to test that your code does actually work correctly across that gap. In traditional frameworks where you spend a lot of time thinking about interfaces between client and server you can often get away with testing both sides of the interface in isolation, but Meteor's full app test mode makes it possible to write integration tests that cover the full stack. Another challenge here is creating test data in the client context; we'll discuss ways to do this in the section on generating test data below.
- **Reactivity**: Meteor's reactivity system is "eventually consistent" in the sense that when you change a reactive input to the system, you'll see the user interface change to reflect this some time later. This can be a challenge when testing, but there are some ways to wait until those changes happen to verify the results, for example `Tracker.afterFlush()`.

The 'meteor test' command

The primary way to test your application in Meteor is the `meteor test` command.

This loads your application in a special "test mode". What this does is:

- *Doesn't* eagerly load *any* of our application code as Meteor normally would.
  - *This is a highly important note as Meteor wouldn't know of any methods/collections/publications unless you import them in your test files.*
- *Does* eagerly load any file in our application (including in `imports/` folders) that look like `*.test[s].*`, or `*.spec[s].*`
- Sets the `Meteor.isTest` flag to be true.
- Starts up the test driver package (see below).

  The Meteor build tool and the `meteor test` command ignore any files located in any `tests/` directory. This allows you to put tests in this directory that you can run using a test runner outside of Meteor's built-in test tools and still not have those files loaded in your application. See Meteor's default file load order rules.

What this means is that you can write tests in files with a certain filename pattern and know they'll not be included in normal builds of your app. When your app runs in test mode, those files will be loaded (and nothing else will), and they can import the modules you want to test. As we'll see this is ideal for unit tests and simple integration tests.

Additionally, Meteor offers a "full application" test mode. You can run this with `meteor test --full-app`.

This is similar to test mode, with key differences:

1. It loads test files matching `*.app-test[s].*` and `*.app-spec[s].*`.
2. It **does** eagerly load our application code as Meteor normally would.
3. Sets the `Meteor.isAppTest` flag to be true (instead of the `Meteor.isTest` flag).

This means that the entirety of your application (including for instance the web server and client side router) is loaded and will run as normal. This enables you to write much more complex integration tests and also load additional files for acceptance tests.

Note that there is another test command in the Meteor tool; `meteor test-packages` is a way of testing Atmosphere packages, which is discussed in the Writing Packages article.

Driver packages

When you run a `meteor test` command, you must provide a `--driver-package` argument. A test driver is a mini-application that runs in place of your app and runs each of your defined tests, whilst reporting the results in some kind of user interface.

There are two main kinds of test driver packages:

- **Web reporters**: Meteor applications that display a special test reporting web UI that you can view the test results in.
- **Console reporters**: These run completely on the command-line and are primary used for automated testing like continuous integration.

Recommended: Mocha

In this article, we'll use the popular Mocha test runner. And you can pair it with any assertion library you want likeChai or expect. In order to write and run tests in Mocha, we need to add an appropriate test driver package.

There are several options. Choose the ones that makes sense for your app. You may depend on more than one and set up different test commands for different situations.

- meteortesting:mocha Runs client and/or server package or app tests and reports all results in the server console. Supports various browsers for

running client tests, including PhantomJS, Selenium ChromeDriver, and Electron. Can be used for running tests on a CI server. Has a watch mode.

These packages don't do anything in development or production mode. They declare themselves `testOnly` so they are not even loaded outside of testing. But when our app is run in test mode, the test driver package takes over, executing test code on both the client and server, and rendering results to the browser.

Here's how we can add the `meteortesting:mocha` package to our app:

```
meteor add meteortesting:mocha
```

Test Files

Test files themselves (for example a file named `todos-item.test.js` or `routing.app-specs.coffee`) can register themselves to be run by the test driver in the usual way for that testing library. For Mocha, that's by using `describe` and `it`:

```
describe('my module', function () {
  it('does something that should be tested', function () {
    // This code will be executed by the test driver when the app is started
    // in the correct mode
  })
})
```

Note that arrow function use with Mocha is discouraged.

Test data

When your app is run in test mode, it is initialized with a clean test database.

If you are running a test that relies on using the database, and specifically the content of the database, you'll need to perform some *setup* steps in your test to ensure the database is in the state you expect.

```
import { Meteor } from 'meteor/meteor';
import expect from 'expect';

import { Notes } from './notes';

 describe('notes', function () {
   const noteOne = {
     _id: 'testNote1',
     title: 'Groceries',
     body: 'Milk, Eggs and Oatmeal'
   };

   beforeEach(function () {
     Notes.remove({});
     Notes.insert(noteOne);
```

```
  });
  ...
```

You can also use `xolvio:cleaner` which is useful for resetting the entire database if you wish to do so. You can use it to reset the database in a `beforeEach` block:

```
import { resetDatabase } from 'meteor/xolvio:cleaner';

describe('my module', function () {
  beforeEach(function () {
    resetDatabase();
  });
});
```

This technique will only work on the server. If you need to reset the database from a client test, `xolvio:cleaner` provides you with a built-in method called `xolvio:cleaner/resetDatabase`:

```
describe('my module', function (done) {
  beforeEach(function (done) {
    // We need to wait until the method call is done before moving on, so we
    // use Mocha's async mechanism (calling a done callback)
    Meteor.call('xolvio:cleaner/resetDatabase', done);
  });
});
```

You can also invoke `resetDatabase` in your methods in case you wanted to apply custom code before or after:

```
import { resetDatabase } from 'meteor/xolvio:cleaner';

// NOTE: Before writing a method like this you'll want to double check
// that this file is only going to be loaded in test mode!!
Meteor.methods({
  'test.resetDatabase': () => {
    // custom code goes here...
    resetDatabase()
    // or here
  }
});
```

As we've placed the code above in a test file, it *will not* load in normal development or production mode (which would be an incredibly bad thing!). If you create a Atmosphere package with a similar feature, you should mark it as `testOnly` and it will similarly only load in test mode.

Generating test data

Often it's sensible to create a set of data to run your test against. You can use standard `insert()` calls against your collections to do this, but often it's easier

to create *factories* which help encode random test data. A great package to use to do this is `dburles:factory`.

In the Todos example app, we define a factory to describe how to create a test todo item, using the `faker` npm package:

```
import faker from '@faker-js/faker';

Factory.define('todo', Todos, {
  listId: () => Factory.get('list'),
  text: () => faker.lorem.sentence(),
  createdAt: () => new Date(),
});
```

To use the factory in a test, we call `Factory.create`:

```
// This creates a todo and a list in the database and returns the todo.
const todo = Factory.create('todo');

// If we have a list already, we can pass in the id and avoid creating another:
const list = Factory.create('list');
const todoInList = Factory.create('todo', { listId: list._id });
```

Mocking the database

As `Factory.create` directly inserts documents into the collection that's passed into the `Factory.define` function, it can be a problem to use it on the client. However there's a neat isolation trick that you can do to replace the server-backed `Todos` client collection with a mocked out local collection, that's encoded in the `hwillson:stub-collections` package.

```
import StubCollections from 'meteor/hwillson:stub-collections';
import { Todos } from 'path/to/todos.js';

StubCollections.stub(Todos);

// Now Todos is stubbed to a simple local collection mock,
//   so for instance on the client we can do:
Todos.insert({ a: 'document' });

// Restore the `Todos` collection
StubCollections.restore();
```

In a Mocha test, it makes sense to use `stub-collections` in a `beforeEach`/`afterEach` block.

Unit testing

Unit testing is the process of isolating a section of code and then testing that the internals of that section work as you expect. As we've split our code base up into ES2015 modules it's natural to test those modules one at a time.

By isolating a module and testing its internal functionality, we can write tests that are *fast* and *accurate*—they can quickly tell you where a problem in your application lies. Note however that incomplete unit tests can often hide bugs because of the way they stub out dependencies. For that reason it's useful to combine unit tests with slower (and perhaps less commonly run) integration and acceptance tests.

A simple Blaze unit test

In the Todos example app, thanks to the fact that we've split our User Interface into smart and reusable components, it's natural to want to unit test some of our reusable components (we'll see below how to integration test our smart components).

To do so, we'll use a very simple test helper that renders a Blaze component off-screen with a given data context. As we place it in `imports`, it won't load in our app by in normal mode (as it's not required anywhere).

`imports/ui/test-helpers.js`:

```
import isString from 'lodash.isstring';
import { Template } from 'meteor/templating';
import { Blaze } from 'meteor/blaze';
import { Tracker } from 'meteor/tracker';

const withDiv = function withDiv(callback) {
  const el = document.createElement('div');
  document.body.appendChild(el);
  try {
    callback(el);
  } finally {
    document.body.removeChild(el);
  }
};

export const withRenderedTemplate = function withRenderedTemplate(template, data, callback)
  withDiv((el) => {
    const ourTemplate = isString(template) ? Template[template] : template;
    Blaze.renderWithData(ourTemplate, data, el);
    Tracker.flush();
    callback(el);
  });
};
```

An example of a reusable component to test is the `Todos_item` template. Here's what a unit test looks like (you can see some others in the app repository).

`imports/ui/components/client/todos-item.tests.js`:

```
/* eslint-env mocha */
```

```
/* eslint-disable func-names, prefer-arrow-callback */

import { Factory } from 'meteor/dburles:factory';
import chai from 'chai';
import { Template } from 'meteor/templating';
import $ from 'jquery';
import { Todos } from '../../../api/todos/todos';


import { withRenderedTemplate } from '../../test-helpers.js';
import '../todos-item.js';

describe('Todos_item', function () {
  beforeEach(function () {
    Template.registerHelper('_', key => key);
  });

  afterEach(function () {
    Template.deregisterHelper('_');
  });

  it('renders correctly with simple data', function () {
    const todo = Factory.build('todo', { checked: false });
    const data = {
      todo: Todos._transform(todo),
      onEditingChange: () => 0,
    };

    withRenderedTemplate('Todos_item', data, el => {
      chai.assert.equal($(el).find('input[type=text]').val(), todo.text);
      chai.assert.equal($(el).find('.list-item.checked').length, 0);
      chai.assert.equal($(el).find('.list-item.editing').length, 0);
    });
  });
});
```

Of particular interest in this test is the following:

Importing

When we run our app in test mode, only our test files will be eagerly loaded. In particular, this means that in order to use our templates, we need to import them! In this test, we import `todos-item.js`, which itself imports `todos.html` (yes, you do need to import the HTML files of your Blaze templates!)

Stubbing

To be a unit test, we must stub out the dependencies of the module. In this case,

thanks to the way we've isolated our code into a reusable component, there's not much to do; principally we need to stub out the `{% raw %}{{_}}{% endraw %}` helper that's created by the `tap:i18n` system. Note that we stub it out in a `beforeEach` and restore it the `afterEach`.

If you're testing code that makes use of globals, you'll need to import those globals. For instance if you have a global `Todos` collection and are testing this file:

```
// logging.js
export function logTodos() {
  console.log(Todos.findOne());
}
```

then you'll need to import `Todos` both in that file and in the test:

```
// logging.js
import { Todos } from './todos.js'
export function logTodos() {
  console.log(Todos.findOne());
}
```

```
// logging.test.js
import { Todos } from './todos.js'
Todos.findOne = () => {
  return {text: "write a guide"}
}

import { logTodos } from './logging.js'
// then test logTodos
...
```

Creating data

We can use the Factory package's `.build()` API to create a test document without inserting it into any collection. As we've been careful not to call out to any collections directly in the reusable component, we can pass the built `todo` document directly into the template.

A simple React unit test

We can also apply the same structure to testing React components and recommend the Enzyme package, which simulates a React component's environment and allows you to query it using CSS selectors. A larger suite of tests is available in the react branch of the Todos app, but let's look at a simple example for now:

```
import { Factory } from 'meteor/dburles:factory';
import React from 'react';
import { shallow } from 'enzyme';
import chai from 'chai';
import TodoItem from './TodoItem.jsx';
```

```
describe('TodoItem', () => {
  it('should render', () => {
    const todo = Factory.build('todo', { text: 'testing', checked: false });
    const item = shallow(<TodoItem todo={todo} />);
    chai.assert(item.hasClass('list-item'));
    chai.assert(!item.hasClass('checked'));
    chai.assert.equal(item.find('.editing').length, 0);
    chai.assert.equal(item.find('input[type="text"]').prop('defaultValue'), 'testing');
  });
});
```

The test is slightly simpler than the Blaze version above because the React sample app is not internationalized. Otherwise, it's conceptually identical. We use Enzyme's `shallow` function to render the `TodoItem` component, and the resulting object to query the document, and also to simulate user interactions. And here's an example of simulating a user checking the todo item:

```
import { Factory } from 'meteor/dburles:factory';
import React from 'react';
import { shallow } from 'enzyme';
import sinon from 'sinon';
import TodoItem from './TodoItem.jsx';
import { setCheckedStatus } from '../../api/todos/methods.js';

describe('TodoItem', () => {
  it('should update status when checked', () => {
    sinon.stub(setCheckedStatus, 'call');
    const todo = Factory.create('todo', { checked: false });
    const item = shallow(<TodoItem todo={todo} />);

    item.find('input[type="checkbox"]').simulate('change', {
      target: { checked: true },
    });

    sinon.assert.calledWith(setCheckedStatus.call, {
      todoId: todo._id,
      newCheckedStatus: true,
    });

    setCheckedStatus.call.restore();
  });
});
```

In this case, the `TodoItem` component calls a Meteor Method `setCheckedStatus` when the user clicks, but this is a unit test so there's no server running. So we stub it out using Sinon. After we simulate the click, we verify that the stub was

called with the correct arguments. Finally, we clean up the stub and restore the original method behavior.

Running unit tests

To run the tests that our app defines, we run our app in test mode:

```
TEST_WATCH=1 meteor test --driver-package meteortesting:mocha
```

As we've defined a test file (`imports/todos/todos.tests.js`), what this means is that the file above will be eagerly loaded, adding the `'builds correctly from factory'` test to the Mocha registry.

To run the tests, visit http://localhost:3000 in your browser. This kicks off `meteortesting:mocha`, which runs your tests both in the browser and on the server. It will display the test results in a div with ID mocha.

Usually, while developing an application, it makes sense to run `meteor test` on a second port (say `3100`), while also running your main application in a separate process:

```
# in one terminal window
meteor

# in another
meteor test --driver-package meteortesting:mocha --port 3100
```

Then you can open two browser windows to see the app in action while also ensuring that you don't break any tests as you make changes.

Isolation techniques

In the unit tests above we saw a very limited example of how to isolate a module from the larger app. This is critical for proper unit testing. Some other utilities and techniques include:

- The `velocity:meteor-stubs` package, which creates simple stubs for most Meteor core objects.
- Alternatively, you can also use tools like Sinon to stub things directly, as we'll see for example in our simple integration test.
- The `hwillson:stub-collections` package we mentioned above.

There's a lot of scope for better isolation and testing utilities.

Testing publications

Let's take this simple publication for example:

```
// server/publications/notes
 Meteor.publish('user.notes', function () {
    return Notes.find({ userId: this.userId });
  });
```

We access Meteor publications using `Meteor.server.publish_handlers`, then use `.apply` to provide the needed parameters for the publication and test what it returns.

```
import { Meteor } from 'meteor/meteor';
import expect from 'expect';

import { Notes } from './notes';

 describe('notes', function () {
   const noteOne = {
     _id: 'testNote1',
     title: 'Groceries',
     body: 'Milk, Eggs and Oatmeal'
     userId: 'userId1'
   };

   beforeEach(function () {
     Notes.remove({});
     Notes.insert(noteOne);
   });

 it('should return a users notes', function () {
      const res = Meteor.server.publish_handlers['user.notes'].apply({ userId: noteOne.user
      const notes = res.fetch();

      expect(notes.length).toBe(1);
      expect(notes[0]).toEqual(noteOne);
    });

  it('should return no notes for user that has none', function () {
      const res = Meteor.server.publish_handlers.notes.apply({ userId: 'testid' });
      const notes = res.fetch();

      expect(notes.length).toBe(0);
    });
 });
```

A useful package for testing publications is `johanbrook:publication-collector`, it allows you to test individual publication's output without needing to create a traditional subscription:

```
describe('notes', function () {
  it('should return a users notes', function (done) {
    // Set a user id that will be provided to the publish function as `this.userId`,
    // in case you want to test authentication.
    const collector = new PublicationCollector({userId: noteOne.userId});
```

```
    // Collect the data published from the `lists.public` publication.
    collector.collect('user.notes', (collections) => {
      // `collections` is a dictionary with collection names as keys,
      // and their published documents as values in an array.
      // Here, documents from the collection 'Lists' are published.
      chai.assert.typeOf(collections.Lists, 'array');
      chai.assert.equal(collections.Lists.length, 1);
      done();
    });
  });
});
```

Note that user documents – ones that you would normally query with `Meteor.users.find()` – will be available as the key `users` on the dictionary passed from a `PublicationCollector.collect()` call. See the tests in the package for more details.

Testing methods

We can also access methods using `Meteor.server.method_handlers` and apply the same principles. Take note of how we can use `sinon.fake()` to mock `this.unblock()`.

```
Meteor.methods({
  'notes.insert'(title, body) {
    if (!this.userId || Meteor.users.findOne({ _id: this.userId })) {
      throw new Meteor.Error('not-authorized', 'You have to be authorized');
    }

    check(title, String);
    check(body, String);

    this.unblock();

    return Notes.insert({
      title,
      body,
      userId: this.userId
    });
  },
  'notes.remove'(_id) {
    if (!this.userId || Meteor.users.findOne({ _id: this.userId })) {
      throw new Meteor.Error('not-authorized', 'You have to be authorized');
    }

    check(_id, String);
```

```
    Notes.remove({ _id, userId: this.userId });
  },
  'notes.update'(_id, {title, body}) {
    if (!this.userId || Meteor.users.findOne({ _id: this.userId })) {
      throw new Meteor.Error('not-authorized', 'You have to be authorized');
    }

    check(_id, String);
    check(title, String);
    check(body, String);

    Notes.update({
      _id,
      userId: this.userId
    }, {
      $set: {
        title,
        body
      }
    });
  }
});


describe('notes', function () {
    const noteOne = {
      _id: 'testNote1',
      title: 'Groceries',
      body: 'Milk, Eggs and Oatmeal'
      userId: 'testUserId1'
    };
    beforeEach(function () {
      Notes.remove({});
    });

    it('should insert new note', function () {
      const _id = Meteor.server.method_handlers['notes.insert'].apply({ userId: noteOne.user

      expect(Notes.findOne({ _id })).toMatchObject(
            expect.objectContaining(noteOne)
        );
    });

    it('should not insert note if not authenticated', function () {
      expect(() => {
        Meteor.server.method_handlers['notes.insert']();
```

```
  }).toThrow();
});

it('should remove note', function () {
  Meteor.server.method_handlers['notes.remove'].apply({ userId: noteOne.userId }, [noteO

  expect(Notes.findOne({ _id: noteOne._id})).toNotExist();
});

it('should not remove note if invalid _id', function () {
  expect(() => {
    Meteor.server.method_handlers['notes.remove'].apply({ userId: noteOne.userId});
  }).toThrow();
});

it('should update note', function () {
  const title = 'To Buy';
  const beef = 'Beef, Salmon'

  Meteor.server.method_handlers['notes.update'].apply({
    userId: noteOne.userId
  }, [
    noteOne._id,
    {title, body}
  ]);

  const note = Notes.findOne(noteOne._id);

  expect(note).toInclude({
    title,
    body
  });
});

it('should not update note if user was not creator', function () {
  const title = 'This is an updated title';

  Meteor.server.method_handlers['notes.update'].apply({
    userId: 'testid'
  }, [
    noteOne._id,
    { title }
  ]);

  const note = Notes.findOne(noteOne._id);
```

```
      expect(note).toInclude(noteOne);
    });
});
```

These examples are heavily inspired by Andrew Mead example app.

Integration testing

An integration test is a test that crosses module boundaries. In the simplest case, this means something very similar to a unit test, where you perform your isolation around multiple modules, creating a non-singular "system under test".

Although conceptually different to unit tests, such tests typically do not need to be run any differently to unit tests and can use the same `meteor test` mode and isolation techniques as we use for unit tests.

However, an integration test that crosses the client-server boundary of a Meteor application (where the modules under test cross that boundary) requires a different testing infrastructure, namely Meteor's "full app" testing mode.

Let's take a look at example of both kinds of tests.

Simple integration test

Our reusable components were a natural fit for a unit test; similarly our smart components tend to require an integration test to really be exercised properly, as the job of a smart component is to bring data together and supply it to a reusable component.

In the Todos example app, we have an integration test for the `Lists_show_page` smart component. This test ensures that when the correct data is present in the database, the template renders correctly – that it is gathering the correct data as we expect. It isolates the rendering tree from the more complex data subscription part of the Meteor stack. If we wanted to test that the subscription side of things was working in concert with the smart component, we'd need to write a full app integration test.

`imports/ui/components/client/todos-item.tests.js`:

```
/* eslint-env mocha */
/* eslint-disable func-names, prefer-arrow-callback */

import { Meteor } from 'meteor/meteor';
import { Factory } from 'meteor/dburles:factory';
import { Random } from 'meteor/random';
import chai from 'chai';
import StubCollections from 'meteor/hwillson:stub-collections';
import { Template } from 'meteor/templating';
import $ from 'jquery';
import { FlowRouter } from 'meteor/ostrio:flow-router-extra';
import sinon from 'sinon';
```

```
import { withRenderedTemplate } from '../../test-helpers.js';
import '../lists-show-page.js';

import { Todos } from '../../../api/todos/todos.js';
import { Lists } from '../../../api/lists/lists.js';

describe('Lists_show_page', function () {
  const listId = Random.id();

  beforeEach(function () {
    StubCollections.stub([Todos, Lists]);
    Template.registerHelper('_', key => key);
    sinon.stub(FlowRouter, 'getParam').returns(listId);
    sinon.stub(Meteor, 'subscribe').returns.({
      subscriptionId: 0,
      ready: () => true,
    });
  });

  afterEach(function () {
    StubCollections.restore();
    Template.deregisterHelper('_');
    FlowRouter.getParam.restore();
    Meteor.subscribe.restore();
  });

  it('renders correctly with simple data', function () {
    Factory.create('list', { _id: listId });
    const timestamp = new Date();
    const todos = [...Array(3).keys()].forEach(i => Factory.create('todo', {
      listId,
      createdAt: new Date(timestamp - (3 - i)),
    }));

    withRenderedTemplate('Lists_show_page', {}, el => {
      const todosText = todos.map(t => t.text).reverse();
      const renderedText = $(el).find('.list-items input[type=text]')
        .map((i, e) => $(e).val())
        .toArray();
      chai.assert.deepEqual(renderedText, todosText);
    });
  });
});
```

Of particular interest in this test is the following:

Importing

As we'll run this test in the same way that we did our unit test, we need to `import` the relevant modules under test in the same way that we did in the unit test.

Stubbing

As the system under test in our integration test has a larger surface area, we need to stub out a few more points of integration with the rest of the stack. Of particular interest here is our use of the `hwillson:stub-collections` package and of Sinon to stub out Flow Router and our Subscription.

Creating data

In this test, we used Factory package's `.create()` API, which inserts data into the real collection. However, as we've proxied all of the `Todos` and `Lists` collection methods onto a local collection (this is what `hwillson:stub-collections` is doing), we won't run into any problems with trying to perform inserts from the client.

This integration test can be run the exact same way as we ran unit tests above.

Full-app integration test

In the Todos example application, we have a integration test which ensures that we see the full contents of a list when we route to it, which demonstrates a few techniques of integration tests.

`imports/startup/client/routes.app-test.js`:

```
/* eslint-env mocha */
/* eslint-disable func-names, prefer-arrow-callback */

import { Meteor } from 'meteor/meteor';
import { Tracker } from 'meteor/tracker';
import { DDP } from 'meteor/ddp-client';
import { FlowRouter } from 'meteor/ostrio:flow-router-extra';
import { assert } from 'chai';

import { Promise } from 'meteor/promise';
import $ from 'jquery';

import { denodeify } from '../../utils/denodeify';
import { generateData } from './../../api/generate-data.app-tests.js';
import { Lists } from '../../api/lists/lists.js';
import { Todos } from '../../api/todos/todos.js';


// Utility -- returns a promise which resolves when all subscriptions are done
const waitForSubscriptions = () => new Promise(resolve => {
```

```
  const poll = Meteor.setInterval(() => {
    if (DDP._allSubscriptionsReady()) {
      Meteor.clearInterval(poll);
      resolve();
    }
  }, 200);
});

// Tracker.afterFlush runs code when all consequent of a tracker based change
//   (such as a route change) have occured. This makes it a promise.
const afterFlushPromise = denodeify(Tracker.afterFlush);

if (Meteor.isClient) {
  describe('data available when routed', () => {
    // First, ensure the data that we expect is loaded on the server
    //   Then, route the app to the homepage
    beforeEach(() => generateData()
      .then(() => FlowRouter.go('/'))
      .then(waitForSubscriptions)
    );

    describe('when logged out', () => {
      it('has all public lists at homepage', () => {
        assert.equal(Lists.find().count(), 3);
      });

      it('renders the correct list when routed to', () => {
        const list = Lists.findOne();
        FlowRouter.go('Lists.show', { _id: list._id });

        return afterFlushPromise()
          .then(waitForSubscriptions)
          .then(() => {
            assert.equal($('.title-wrapper').html(), list.name);
            assert.equal(Todos.find({ listId: list._id }).count(), 3);
          });
      });
    });
  });
}
```

Of note here:

- Before running, each test sets up the data it needs using the `generateData` helper (see the section on creating integration test data for more detail) then goes to the homepage.

- Although Flow Router doesn't take a done callback, we can use `Tracker.afterFlush` to wait for all its reactive consequences to occur.
- Here we wrote a little utility (which could be abstracted into a general package) to wait for all the subscriptions which are created by the route change (the `todos.inList` subscription in this case) to become ready before checking their data.

Running full-app tests

To run the full-app tests in our application, we run:

```
meteor test --full-app --driver-package meteortesting:mocha
```

When we connect to the test instance in a browser, we want to render a testing UI rather than our app UI, so the `mocha-web-reporter` package will hide any UI of our application and overlay it with its own. However the app continues to behave as normal, so we are able to route around and check the correct data is loaded.

Creating data

To create test data in full-app test mode, it usually makes sense to create some special test methods which we can call from the client side. Usually when testing a full app, we want to make sure the publications are sending through the correct data (as we do in this test), and so it's not sufficient to stub out the collections and place synthetic data in them. Instead we'll want to actually create data on the server and let it be published.

Similar to the way we cleared the database using a method in the `beforeEach` in the test data section above, we can call a method to do that before running our tests. In the case of our routing tests, we've used a file called `imports/api/generate-data.app-tests.js` which defines this method (and will only be loaded in full app test mode, so is not available in general!):

```
// This file will be auto-imported in the app-test context,
// ensuring the method is always available

import { Meteor } from 'meteor/meteor';
import { Factory } from 'meteor/dburles:factory';
import { resetDatabase } from 'meteor/xolvio:cleaner';
import { Random } from 'meteor/random';

import { denodeify } from '../utils/denodeify';

const createList = (userId) => {
  const list = Factory.create('list', { userId });
  [...Array(3).keys()].forEach(() => Factory.create('todo', { listId: list._id }));
  return list;
};
```

```
// Remember to double check this is a test-only file before
// adding a method like this!
Meteor.methods({
  generateFixtures() {
    resetDatabase();

    // create 3 public lists
    [...Array(3).keys()].forEach(() => createList());

    // create 3 private lists
    [...Array(3).keys()].forEach(() => createList(Random.id()));
  },
});

let generateData;
if (Meteor.isClient) {
  // Create a second connection to the server to use to call
  // test data methods. We do this so there's no contention
  // with the currently tested user's connection.
  const testConnection = Meteor.connect(Meteor.absoluteUrl());

  generateData = denodeify((cb) => {
    testConnection.call('generateFixtures', cb);
  });
}

export { generateData };
```

Note that we've exported a client-side symbol `generateData` which is a promisified version of the method call, which makes it simpler to use this sequentially in tests.

Also of note is the way we use a second DDP connection to the server in order to send these test "control" method calls.

Acceptance testing

Acceptance testing is the process of taking an unmodified version of our application and testing it from the "outside" to make sure it behaves in a way we expect. Typically if an app passes acceptance tests, we have done our job properly from a product perspective.

As acceptance tests test the behavior of the application in a full browser context in a generic way, there are a range of tools that you can use to specify and run such tests. In this guide we'll demonstrate using Cypress, an acceptance testing tool with a few neat Meteor-specific features that makes it easy to use.

Install Cypress as a dev dependency:

```
cd /your/project/path
meteor npm install cypress --save-dev
```

Designate a special directory for cypress test to avoid Meteor eagerly loading it:

```
mkdir tests
mv cypress/ tests/cypress
```

Create `cypress.json` file at the root of your project to config Cypress:

```
{
  "fixturesFolder": "tests/cypress/fixtures",
  "integrationFolder": "tests/cypress/integration",
  "pluginsFile": "tests/cypress/plugins/index.js",
  "screenshotsFolder": "tests/cypress/screenshots",
  "supportFile": "tests/cypress/support/index.js",
  "videosFolder": "tests/cypress/videos"
}
```

Add commands to your `package.json`

```
 "scripts": {
    "cypress:gui": "cypress open",
    "cypress:headless": "cypress run"
  },
```

Now, let's create a simple test by adding a new file called `signup_tests.js` in the `tests/cypress/integration/` directory.

```
describe("sign-up", () => {
  beforeEach(() => {
    cy.visit("http://localhost:3000/");
  });

  it("should create and log the new user", () => {
    cy.contains("Register").click();
    cy.get("input#at-field-email").type("jean-peter.mac.calloway@gmail.com");
    cy.get("input#at-field-password").type("awesome-password");
    cy.get("input#at-field-password_again").type("awesome-password");
    // I added a name field on meteor user accounts system
    cy.get("input#at-field-name").type("Jean-Peter");
    cy.get("button#at-btn").click();

    cy.url().should("eq", "http://localhost:3000/board");

    cy.window().then(win => {
      // this allows accessing the window object within the browser
      const user = win.Meteor.user();
      expect(user).to.exist;
      expect(user.profile.name).to.equal("Jean-Peter");
```

```
      expect(user.emails[0].address).to.equal(
        "jean-peter.mac.calloway@gmail.com"
      );
    });
  });
});
```

This is example is sampled from Jean-Denis Gallego post. You may also check out this entry on Meteor blog and marmelab article for more information.

Continuous Integration

Continuous integration testing is the process of running tests on every commit of your project.

There are two principal ways to do it: on the developer's machine before allowing them to push code to the central repository, and on a dedicated CI server after each push. Both techniques are useful, and both require running tests in a commandline-only fashion.

Command line

We've seen one example of running tests on the command line, using our `meteor npm run cypress:headless` mode.

We can also use a command-line driver for Mocha `meteortesting:mocha` to run our standard tests on the command line.

Adding and using the package is straightforward:

```
meteor add meteortesting:mocha
meteor test --once --driver-package meteortesting:mocha
```

(The `--once` argument ensures the Meteor process stops once the test is done).

We can also add that command to our `package.json` as a `test` script:

```
{
  "scripts": {
    "test": "meteor test --once --driver-package meteortesting:mocha"
  }
}
```

Now we can run the tests with `meteor npm test`.

CircleCI

CircleCI is a great continuous integration service that allows us to run (possibly time consuming) tests on every push to a repository like GitHub. To use it with the commandline test we've defined above, we can follow their standard getting started tutorial and use a `circle.yml` file similar to this:

```
machine:
  node:
```

```
    version: 0.10.43
dependencies:
  override:
    - curl https://install.meteor.com | /bin/sh
    - npm install
checkout:
  post:
    - git submodule update --init
```