Part 7 - Integration tests with `dataverse-ify`

This is part of the course 'Scott's guide to building Power Apps JavaScript Web Resources using TypeScript'.

In this seventh part we will cover calling the `WebApi` with easy using `dataverse-ify`. You can find full details about how the `WepApi` works at https://docs.microsoft.com/en-us/powerapps/developer/model-driven-apps/clientapi/reference/xrm-webapi.

`Dataverse-ify` is an open source library that aims to provide an interface to the Dataverse `WebApi` from TypeScript that works in a similar way to the C# `IOrganisationService` so that you do not need to code around the complexities of the native `WebApi` syntax. See https://github.com/scottdurow/dataverse-ify/wiki

Adding Integration Tests

Unit tests will enable you to quickly ensure that your code is functioning as expected - however the one unknown is how Dataverse will behave when calling the `WebApi`. Mocking assumes that it returns what you are expected, however an integration test will ensure it functions according to those expectations and can be a very effective way of ensure your code covers all scenarios and edge-cases. The principle of an integration test is that it should create the data necessary to run the test, and then delete it afterwards so that it leaves no footprint.

Calling Dataverse from inside VSCode

`dataverse-ify` contains an implementation of the `WebApi` that allows you to use the `Xrm.WebApi` API from inside your VSCode integration tests running locally without debugging inside the browser. This can reduce the amount of time it takes to develop and debug your code.

Inside your VSCode project, add an file `config\test.yaml`

```
nodewebapi:
 logging: verbose
 server:
  host: https://org.crm.dynamics.com
  version: 9.1
```

Replace `org.crm.dynamics.com` with the URL of your environment.

Add Integration Test

Now we have configured connectivity to our Dataverse environment, we can create a integration test similar to our unit test, however this one will actually communicate with the server.

Add a new file `src\Ribbon\__tests__\integration.AccountRibbon.test.ts`

```
/* eslint-disable camelcase */
import { XrmMockGenerator } from "xrm-mock";
import { SetupGlobalContext } from "dataverse-ify/lib/webapi";
import { Opportunity, OpportunityAttributes, opportunityMetadata } from
"../../dataverse-gen/entities/Opportunity";
import { Account, accountMetadata } from "../../dataverse-gen/entities/Account";
import { Entity, setMetadataCache, XrmContextCdsServiceClient } from "dataverse-
ify";
import { AccountRibbon } from "../AccountRibbon";
import { opportunity_opportunity_statecode } from "../../dataverse-
gen/enums/opportunity_opportunity_statecode";
import { metadataCache } from "../../dataverse-gen/metadata";
describe("AccountRibbon", () => {
  beforeEach(async () => {
    XrmMockGenerator.initialise();
    await SetupGlobalContext();
    setMetadataCache(metadataCache);
    Xrm.Utility.showProgressIndicator = jest.fn();
    Xrm.Utility.closeProgressIndicator = jest.fn();
    Xrm.Navigation.openAlertDialog = jest.fn();
    Xrm.Navigation.openErrorDialog = jest.fn().mockImplementation((ex) =>
console.debug(JSON.stringify(ex)));
    Xrm.Navigation.openConfirmDialog = jest.fn().mockReturnValue({ confirmed:
true } as Xrm.Navigation.ConfirmResult);
  });

  it("closes open opportunities", async () => {
    // Arrange
    const serviceClient = new XrmContextCdsServiceClient(Xrm.WebApi);

    const opportunity1 = {
      logicalName: opportunityMetadata.logicalName,
      name: "Opportunity Integration Test",
    } as Opportunity;

    const account1 = {
      logicalName: accountMetadata.logicalName,
      name: "Account Integration Test",
    } as Account;

    // Act
    try {
      account1.id = await serviceClient.create(account1);
      opportunity1.customerid = Entity.toEntityReference(account1);
      opportunity1.id = await serviceClient.create(opportunity1);

      await AccountRibbon.closeOpportunitiesInternal(serviceClient,
account1.id);

      // Assert
      // Check that the opportunity is closed
      const closedOp = await serviceClient.retrieve<Opportunity>
(opportunityMetadata.logicalName, opportunity1.id, [
        OpportunityAttributes.StateCode,
        OpportunityAttributes.StatusCode,
      ]);
      expect(closedOp.statecode).toBe(opportunity_opportunity_statecode.Won);
    } catch (ex) {
      fail(ex);
    } finally {
      // Tidy up
      if (opportunity1.id) await serviceClient.delete(opportunity1);
      if (account1.id) await serviceClient.delete(account1);
    }
  }, 100000);
});
```

> **Note** The 100000 at the end of the test code is the timeout in milliseconds that we allow for the test to run.

`SetupGlobalContext`

The key part to our integration test is the call to `await SetupGlobalContext();` This is a function that is imported via `import { SetupGlobalContext } from "dataverse-ify/lib/webapi"` and is responsible for replacing the `Xrm.WebApi` implementation locally to point to an implementation that uses the `config/test.yaml` file and call the corresponding Dataverse environment. You don't need to be using `Dataverse-ify's` `CdsService` to use this - it works for normal `Xrm.WebApi` calls.

> **Note:** `SetupGlobalContext` only ever needs to be called inside your unit tests, however `setMetadataCache` will need to be called before you make calls to `dataverse-ify` even in your web resource code.

Asserting integration test results

In our integration tests, we can easily query dataverse to check that the necessary operations have been carried out. This is similar to the mock exceptions we added to our unit tests earlier. These expectations use the `serviceClient` in a similar way to the code we are testing does!

```
const closedOp = await serviceClient.retrieve<Opportunity>
(opportunityMetadata.logicalName, opportunity1.id, [
        OpportunityAttributes.StateCode,
        OpportunityAttributes.StatusCode,
      ]);
expect(closedOp.statecode).toBe(opportunity_opportunity_statecode.Won);
```

Running Integration tests

Once you have written your integration tests, you can debug using F5 (with the test open) in the same way that you did for the unit tests. This has the advantage that you can check how your code works against your dataverse environment without continuously setting up records in the user interface. Once your integration tests work inside jest, you should have a high degree of confidence that the code will work once deployed to the Model Driven App!

Once you have debugged your tests, you can run them all using:

```
jest interation
```

Next Up

Now that we've created some fairly complex JavaScript logic and tested it (unit and integration) we are ready to deploy and test inside our Model Driven App.