Part 3 – Developing and Testing locally inside VSCode

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

In this third part we will cover developing your TypeScript web resource and testing first before deploying to your Dataverse environment.

Install `jest` and `xrm-mock`

Our unit tests will use a library called `jest` to define tests and `xrm-mock` to simulate the `Xrm` Client Side API.

At the command prompt of your VSCode project, type:

```
npm install jest ts-jest xrm-mock @types/jest --save-dev
```

- `jest` – a library that is used to write and run tests (https://jestjs.io/)
- `ts-jest` – a jest pre-set that allows using `jest` with TypeScript
- `xrm-mock` – a library for mocking the form context in Model Driven Apps (https://github.com/camelCaseDave/xrm-mock)

Add `jest.config.js`

Add the `jest.config.js` to the root of your project:

```
module.exports = {
  preset: "ts-jest",
  testEnvironment: "node",
};
```

This file is automatically used by jest when running tests and used to determine the settings to use.

- `ts-jest` - this tells jest that our tests are written in TypeScript and must be transpiled first
- `node` - this is the environment that our tests will run in. Another valid option is us `jsdom` that would emulate running inside a browser (by adding features such as `window`, `document` and `DomParser`) - however our tests will use node so that they can communicate with the Dataverse environment later when we write integration tests.

Add `launch.json`

When you press F5 inside a unit test, it is convenient to have VSCode allow you to debug your unit tests and hit breakpoints. To setup this up, add a `launch.json` to your `.vscode` folder:

https://github.com/scottdurow/dataverse-jswebresource-template/blob/master/.vscode/launch.json

> Note: If you don't add a `jest.config` then you will get the error 'SyntaxError: Cannot use import statement outside a module' later when you run the tests.

Create unit tests for Account Form onload

Create a new folder for our Account Form logic tests: `src/Forms/__tests__`

> **Note:** The `__tests__` folder is the default name used by jest to search for tests, but this can be changed inside the `jest.config` (see https://jestjs.io/docs/configuration).

Create a new file for our TypeScript tests: `src/Forms/__tests__/unit.AccountForm.test.ts`

> **Note:** by naming the file `.test.ts` – VSCode will show the icon of the file differently to normal TypeScript files. By prefixing the file with unit we can later select which type of tests to run easily (e.g. unit or integration tests).

Add a simple test

The jest library provides a very simple way of defining tests:

- `describe` is used to divide your tests into components – you can nest describe if needed.
- `it` is used to describe an individual sentence – add the description so that it reads like a sentence.
- `beforeEach` - a function that is run before each `it` test. This is useful for adding mocking code that is used by every test in the describe suite. Tests in each `describe` suite are run in sequence that they appear.

Our Account Form onload function will add an event that validates the format of URLs entered into the website field. Add the following to `src\Forms\__tests__\unit.AccountForm.test.ts`:

```
import { AccountForm } from "../AccountForm";
import { XrmMockGenerator } from "xrm-mock";

describe("AccountForm.onload", () => {
  beforeEach(() => {
    XrmMockGenerator.initialise();
  });

  it("notifies invalid website addresses", () => {
    const context = XrmMockGenerator.getEventContext();
    const websiteMock = XrmMockGenerator.Attribute.createString("websiteurl",
"foobar");
    websiteMock.controls.itemCollection[0].setNotification = jest.fn();
    AccountForm.onload(context);
    websiteMock.fireOnChange();
    expect(websiteMock.controls.itemCollection[0].setNotification).toBeCalled();
  });

  it("clears notification on valid website address", () => {
    const context = XrmMockGenerator.getEventContext();
    const websiteMock = XrmMockGenerator.Attribute.createString("websiteurl",
"foo");
    websiteMock.controls.itemCollection[0].setNotification = jest.fn();
    websiteMock.controls.itemCollection[0].clearNotification = jest.fn();
    AccountForm.onload(context);
    websiteMock.fireOnChange();

 expect(websiteMock.controls.itemCollection[0].setNotification).toBeCalledWith(e
xpect.any(String), "websiteurl");
```

```
    websiteMock.value = "https://learn.develop1.net";
    websiteMock.fireOnChange();

  expect(websiteMock.controls.itemCollection[0].clearNotification).toBeCalledWith
("websiteurl");
  });
});
```

If you now run jest, you will see that these two test fail since we have not implemented the logic yet!

At the command prompt of your VSCode project, type:

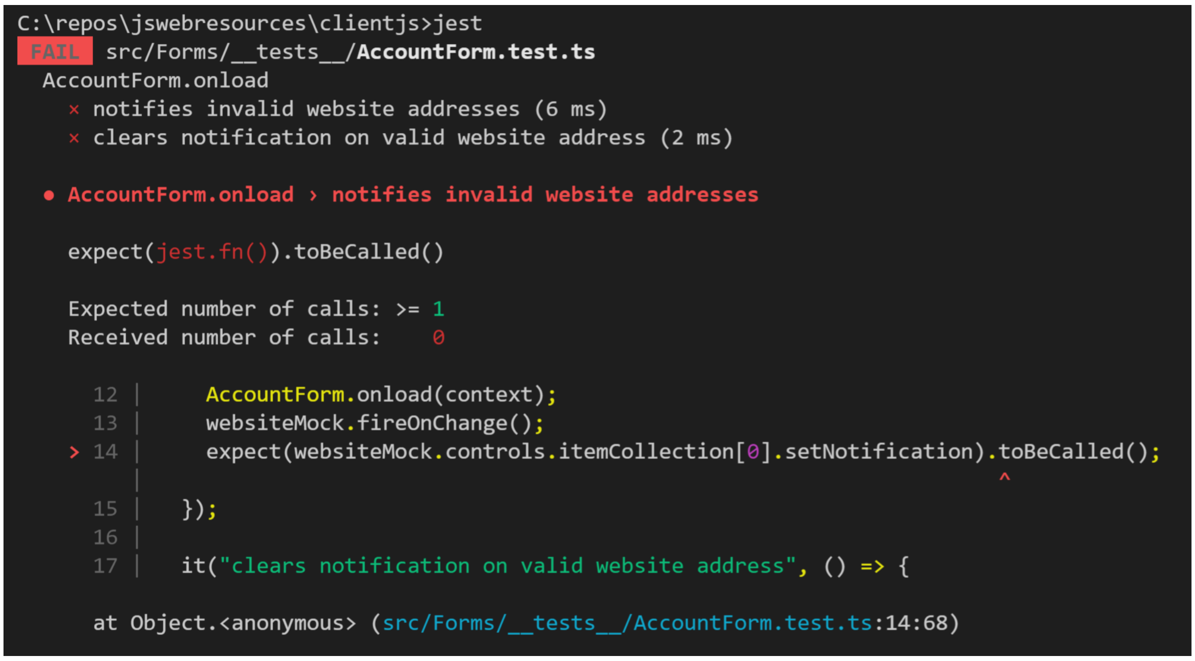
```
npx jest
```

Or you can install `jest` globally using:

```
npm install jest -g
```

This would allow you to then simply use the following at the command line:

```
jest
```



```
C:\repos\jswebresources\clientjs>jest
 FAIL  src/Forms/__tests__/AccountForm.test.ts
  AccountForm.onload
    × notifies invalid website addresses (6 ms)
    × clears notification on valid website address (2 ms)

  ● AccountForm.onload › notifies invalid website addresses

    expect(jest.fn()).toBeCalled()

    Expected number of calls: >= 1
    Received number of calls:    0

      12 |     AccountForm.onload(context);
      13 |     websiteMock.fireOnChange();
    > 14 |     expect(websiteMock.controls.itemCollection[0].setNotification).toBeCalled();
         |                                                                    ^
      15 |   });
      16 |
      17 |   it("clears notification on valid website address", () => {

      at Object.<anonymous> (src/Forms/__tests__/AccountForm.test.ts:14:68)
```

These tests obviously will fail at the moment because we have not added any code!

To define which unit tests we should run for the project, we can add the following script to the `package.json`:

```
"scripts": {

    "test": "jest",

```

This allows us to then simply run the following to run the tests:

```
npm test
```

Later, we will define a different set of tests for integration and unit tests using different scripts to run the different suites of tests.

Write the code

Currently the code does not perform any validation, so we add the following to the `src\Forms\AccountForm.ts`

```
export class AccountForm {
  static async onload(context: Xrm.Events.EventContext): Promise<void> {

 context.getFormContext().getAttribute("websiteurl").addOnChange(AccountForm.onW
ebsiteChanged);
  }
  static onWebsiteChanged(context: Xrm.Events.EventContext): void {
    const formContext = context.getFormContext();
    const websiteAttribute = formContext.getAttribute("websiteurl");
    const websiteRegex = /^(https?:\/\/)?([\w\d]+\.)?[\w\d]+\.\w+\/?.+$/g;

    let isValid = true;
    if (websiteAttribute && websiteAttribute.getValue()) {
      const match = websiteAttribute.getValue().match(websiteRegex);
      isValid = match != null;
    }

    websiteAttribute.controls.forEach((c) => {
      if (isValid) {
        (c as Xrm.Controls.StringControl).clearNotification("websiteurl");
      } else {
        (c as Xrm.Controls.StringControl).setNotification("Invalid Website
Address", "websiteurl");
      }
    });
  }
}
```

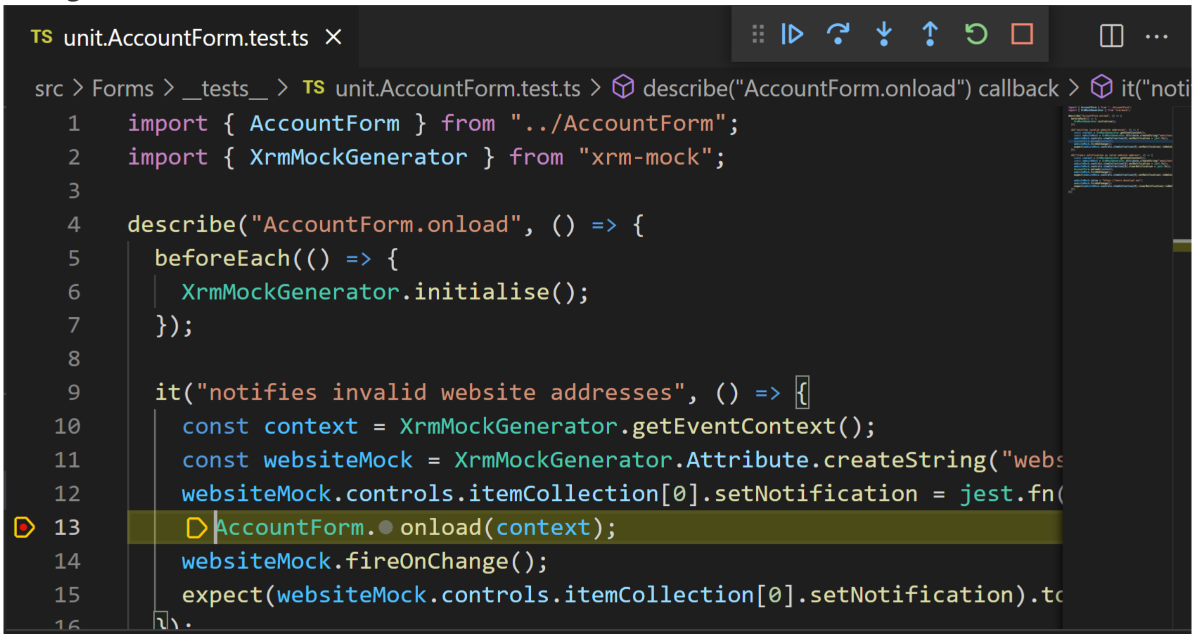
This code uses the `@types/xrm` definitions we imported earlier to easily access the Client Side SDK in a strongly typed way. This is one of the significant advantages of using TypeScript since the types will be checked when transpiling. You will also note `ESLint` doing it's job and ensuring your code is consistent in layout and style.

See more info on `setNotification` here -https://docs.microsoft.com/en-us/powerapps/developer/model-driven-apps/clientapi/reference/controls/setnotification

Debug your tests

If all of the above has been performed correctly, you now should be able to open your `unit.AccountForm.test.ts` file in VSCode and add a breakpoint, then press F5 to start debugging.

VSCode should then allow you to debug your tests and step through each line using the standard debug controls.



unit.AccountForm.test.ts

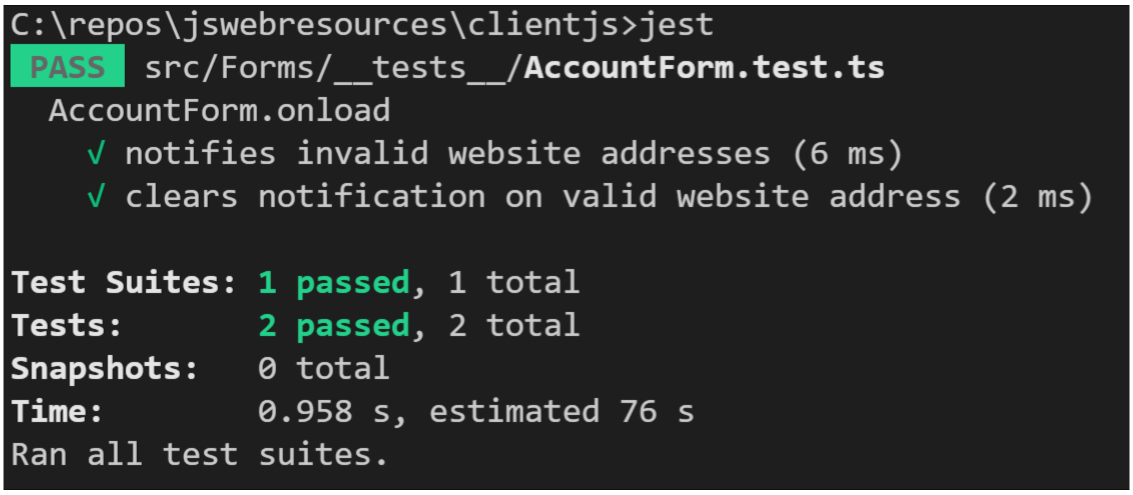
src > Forms > __tests__ > unit.AccountForm.test.ts > describe("AccountForm.onload") callback > it("noti

```
import { AccountForm } from "../AccountForm";
import { XrmMockGenerator } from "xrm-mock";

describe("AccountForm.onload", () => {
  beforeEach(() => {
    XrmMockGenerator.initialise();
  });

  it("notifies invalid website addresses", () => {
    const context = XrmMockGenerator.getEventContext();
    const websiteMock = XrmMockGenerator.Attribute.createString("webs
    websiteMock.controls.itemCollection[0].setNotification = jest.fn(
    AccountForm.onload(context);
    websiteMock.fireOnChange();
    expect(websiteMock.controls.itemCollection[0].setNotification).to
  });
```

If you type `jest` at the command line you will also see that now your tests pass:



```
C:\repos\jswebresources\clientjs>jest
 PASS  src/Forms/__tests__/AccountForm.test.ts
  AccountForm.onload
    √ notifies invalid website addresses (6 ms)
    √ clears notification on valid website address (2 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.958 s, estimated 76 s
Ran all test suites.
```

If you wanted to specifically test an individual test you can also use the following (after jest is installed globally as described above):

```
jest unit.AccountForm
```

This will just run the single `AccountForm` test suite by using the parameter provided to match against the TypeScript file names that contain tests.

A note about VSCode Test Runners

There are many excellent VSCode extensions that provide a convenient user interface and code lens for your tests. I would recommend starting with the command line approach first, and then move to a user interface provided by an extension. The extensions are simply automating what you would be doing at the command line.

Up Next...

Now that you have built, tested and bundled your JavaScript Webresource, the next step is to deploy it and test it inside the browser.