Unit testing in Office Add-ins

Article • 02/22/2023

Unit tests check your add-in's functionality without requiring network or service connections, including connections to the Office application. Unit testing server-side code, and client-side code that does *not* call the Office JavaScript APIs, is the same in Office Add-ins as it is in any web application, so it requires no special documentation. But client-side code that calls the Office JavaScript APIs is challenging to test. To solve these problems, we have created a library to simplify the creation of mock Office objects in unit tests: Office-Addin-Mock. The library makes testing easier in the following ways:

- The Office JavaScript APIs must initialize in a webview control in the context of an Office application (Excel, Word, etc.), so they cannot be loaded in the process in which unit tests run on your development computer. The Office-Addin-Mock library can be imported into your test files, which enables the mocking of Office JavaScript APIs inside the Node.js process in which the tests run.
- The application-specific APIs have load and sync methods that must be called in a particular order relative to other functions and to each other. Moreover, the `load` method must be called with certain parameters depending on what what properties of Office objects are going to be read in by code *later* in the function being tested. But unit testing frameworks are inherently stateless, so they cannot keep a record of whether `load` or `sync` was called or what parameters were passed to `load`. The mock objects that you create with the Office-Addin-Mock library have internal state that keeps track of these things. This enables the mock objects to emulate the error behavior of actual Office objects. For example, if the function that is being tested tries to read a property that was not first passed to `load`, then the test will return an error similar to what Office would return.

The library doesn't depend on the Office JavaScript APIs and it can be used with any JavaScript unit testing framework, such as:

- Jest
- Mocha
- Jasmine

The examples in this article use the Jest framework. There are examples using the Mocha framework at the Office-Addin-Mock home page.

Prerequisites

This article assumes that you are familiar with the basic concepts of unit testing and mocking, including how to create and run test files, and that you have some experience with a unit testing framework.

> **Tip**
>
> If you are working with Visual Studio, we recommend that you read the article **Unit testing JavaScript and TypeScript in Visual Studio** for some basic information about JavaScript unit testing in Visual Studio and then return to this article.

Install the tool

To install the library, open a command prompt, navigate to the root of your add-in project, and then enter the following command.

```
command line
npm install office-addin-mock --save-dev
```

Basic usage

1. Your project will have one or more test files. (See the instructions for your test framework and the example test files in Examples below.) Import the library, with either the `require` or `import` keyword, to any test file that has a test of a function that calls the Office JavaScript APIs, as shown in the following example.

   ```
   JavaScript
   const OfficeAddinMock = require("office-addin-mock");
   ```

2. Import the module that contains the add-in function that you want to test with either the `require` or `import` keyword. The following is an example that assumes your test file is in a subfolder of the folder with your add-in's code files.

   ```
   JavaScript
   const myOfficeAddinFeature = require("../my-office-add-in");
   ```

3. Create a data object that has the properties and subproperties that you need to mock to test the function. The following is an example of an object that mocks the

Excel Workbook.range.address property and the Workbook.getSelectedRange method. This isn't the final mock object. Think of it as a seed object that is used by `OfficeMockObject` to create the final mock object.

```javascript
const mockData = {
  workbook: {
    range: {
      address: "C2:G3",
    },
    getSelectedRange: function () {
      return this.range;
    },
  },
};
```

4. Pass the data object to the `OfficeMockObject` constructor. Note the following about the returned `OfficeMockObject` object.

   - It is a simplified mock of an OfficeExtension.ClientRequestContext object.
   - The mock object has all the members of the data object and also has mock implementations of the `load` and `sync` methods.
   - The mock object will mimic crucial error behavior of the `ClientRequestContext` object. For example, if the Office API you are testing tries to read a property without first loading the property and calling `sync`, then the test will fail with an error similar to what would be thrown in production runtime: "Error, property not loaded".

```javascript
const contextMock = new OfficeAddinMock.OfficeMockObject(mockData);
```

> **Note**
>
> Full reference documentation for the `OfficeMockObject` type is at **Office-Addin-Mock**.

5. In the syntax of your test framework, add a test of the function. Use the `OfficeMockObject` object in place of the object that it mocks, in this case the `ClientRequestContext` object. The following continues the example in Jest. This example test assumes that the add-in function that is being tested is called `getSelectedRangeAddress`, that it takes a `ClientRequestContext` object as a

parameter, and that it is intended to return the address of the currently selected range. The full example is later in this article.

JavaScript

```
test("getSelectedRangeAddress should return the address of the range",
async function () {
  expect(await getSelectedRangeAddress(contextMock)).toBe("C2:G3");
});
```

6. Run the test in accordance with documentation of the test framework and your development tools. Typically, there is a **package.json** file with a script that executes the test framework. For example, if Jest is the framework, **package.json** would contain the following:

JSON

```
"scripts": {
  "test": "jest",
  -- other scripts omitted --
}
```

To run the test, enter the following in a command prompt in the root of the project.

command line

```
npm test
```

Examples

The examples in this section use Jest with its default settings. These settings support CommonJS modules. See the Jest documentation for how to configure Jest and Node.js to support ECMAScript modules and to support TypeScript. To run any of these examples, take the following steps.

1. Create an Office Add-in project for the appropriate Office host application (for example, Excel or Word). One way to do this quickly is to use the Yeoman generator for Office Add-ins.
2. In the root of the project, install Jest.
3. Install the office-addin-mock tool.
4. Create a file exactly like the first file in the example and add it to the folder that contains the project's other source files, often called `\src`.

5. Create a subfolder to the source file folder and give it an appropriate name, such as `\tests`.
6. Create a file exactly like the test file in the example and add it to the subfolder.
7. Add a `test` script to the **package.json** file, and then run the test, as described in Basic usage.

Mocking the Office Common APIs

This example assumes an Office Add-in for any host that supports the Office Common APIs (for example, Excel, PowerPoint, or Word). The add-in has one of its features in a file named `my-common-api-add-in-feature.js`. The following shows the contents of the file. The `addHelloWorldText` function sets the text "Hello World!" to whatever is currently selected in the document; for example; a range in Word, or a cell in Excel, or a text box in PowerPoint.

JavaScript

```
const myCommonAPIAddinFeature = {

    addHelloWorldText: async () => {
        const options = { coercionType: Office.CoercionType.Text };
        await Office.context.document.setSelectedDataAsync("Hello World!",
options);
    }
}

module.exports = myCommonAPIAddinFeature;
```

The test file, named `my-common-api-add-in-feature.test.js` is in a subfolder, relative to the location of the add-in code file. The following shows the contents of the file. Note that the top level property is `context`, an Office.Context object, so the object that is being mocked is the parent of this property: an Office object. Note the following about this code:

- The `OfficeMockObject` constructor does *not* add all of the Office enum classes to the mock `Office` object, so the `CoercionType.Text` value that is referenced in the add-in method must be added explicitly in the seed object.
- Because the Office JavaScript library isn't loaded in the node process, the `Office` object that is referenced in the add-in code must be declared and initialized.

JavaScript

```
const OfficeAddinMock = require("office-addin-mock");
const myCommonAPIAddinFeature = require("../my-common-api-add-in-feature");
```

```
// Create the seed mock object.
const mockData = {
    context: {
      document: {
        setSelectedDataAsync: function (data, options) {
          this.data = data;
          this.options = options;
        },
      },
    },
    // Mock the Office.CoercionType enum.
    CoercionType: {
      Text: {},
    },
};

// Create the final mock object from the seed object.
const officeMock = new OfficeAddinMock.OfficeMockObject(mockData);

// Create the Office object that is called in the addHelloWorldText
function.
global.Office = officeMock;

/* Code that calls the test framework goes below this line. */

// Jest test
test("Text of selection in document should be set to 'Hello World'", async
function () {
    await myCommonAPIAddinFeature.addHelloWorldText();
    expect(officeMock.context.document.data).toBe("Hello World!");
});
```

Mocking the Outlook APIs

Although strictly speaking, the Outlook APIs are part of the Common API model, they have a special architecture that is built around the Mailbox object, so we have provided a distinct example for Outlook. This example assumes an Outlook that has one of its features in a file named `my-outlook-add-in-feature.js`. The following shows the contents of the file. The `addHelloWorldText` function sets the text "Hello World!" to whatever is currently selected in the message compose window.

JavaScript

```
const myOutlookAddinFeature = {

    addHelloWorldText: async () => {
        Office.context.mailbox.item.setSelectedDataAsync("Hello World!");
      }
}
```

```
module.exports = myOutlookAddinFeature;
```

The test file, named `my-outlook-add-in-feature.test.js` is in a subfolder, relative to the location of the add-in code file. The following shows the contents of the file. Note that the top level property is `context`, an Office.Context object, so the object that is being mocked is the parent of this property: an Office object. Note the following about this code:

- The `host` property on the mock object is used internally by the mock library to identify the Office application. It's mandatory for Outlook. It currently serves no purpose for any other Office application.
- Because the Office JavaScript library isn't loaded in the node process, the `Office` object that is referenced in the add-in code must be declared and initialized.

JavaScript

```
const OfficeAddinMock = require("office-addin-mock");
const myOutlookAddinFeature = require("../my-outlook-add-in-feature");

// Create the seed mock object.
const mockData = {
  // Identify the host to the mock library (required for Outlook).
  host: "outlook",
  context: {
    mailbox: {
      item: {
          setSelectedDataAsync: function (data) {
          this.data = data;
        },
      },
    },
  },
};

// Create the final mock object from the seed object.
const officeMock = new OfficeAddinMock.OfficeMockObject(mockData);

// Create the Office object that is called in the addHelloWorldText
function.
global.Office = officeMock;

/* Code that calls the test framework goes below this line. */

// Jest test
test("Text of selection in message should be set to 'Hello World'", async
function () {
    await myOutlookAddinFeature.addHelloWorldText();
```

```
    expect(officeMock.context.mailbox.item.data).toBe("Hello World!");
});
```

Mocking the Office application-specific APIs

When you are testing functions that use the application-specific APIs, be sure that you are mocking the right type of object. There are two options:

- Mock a OfficeExtension.ClientRequestObject. Do this when the function that is being tested meets both of the following conditions:
  - It doesn't call a *Host*.`run` function, such as Excel.run.
  - It doesn't reference any other direct property or method of a *Host* object.
- Mock a *Host* object, such as Excel or Word. Do this when the preceding option isn't possible.

Examples of both types of tests are in the subsections below.

> **Note**
>
> The Office-Addin-Mock library doesn't currently support mocking collection type objects, which are all the objects in the application-specific APIs that are named on the pattern *Collection, such as WorksheetCollection. We are working hard to add this support to the library.

Mocking a ClientRequestContext object

This example assumes an Excel add-in that has one of its features in a file named `my-excel-add-in-feature.js`. The following shows the contents of the file. Note that the `getSelectedRangeAddress` is a helper method called inside the callback that is passed to `Excel.run`.

JavaScript

```javascript
const myExcelAddinFeature = {

    getSelectedRangeAddress: async (context) => {
        const range = context.workbook.getSelectedRange();
        range.load("address");

        await context.sync();

        return range.address;
```

```
    }
}

module.exports = myExcelAddinFeature;
```

The test file, named `my-excel-add-in-feature.test.js` is in a subfolder, relative to the location of the add-in code file. The following shows the contents of the file. Note that the top level property is `workbook`, so the object that is being mocked is the parent of an `Excel.Workbook`: a `ClientRequestContext` object.

JavaScript

```
const OfficeAddinMock = require("office-addin-mock");
const myExcelAddinFeature = require("../my-excel-add-in-feature");

// Create the seed mock object.
const mockData = {
    workbook: {
      range: {
        address: "C2:G3",
      },
      // Mock the Workbook.getSelectedRange method.
      getSelectedRange: function () {
        return this.range;
      },
    },
};

// Create the final mock object from the seed object.
const contextMock = new OfficeAddinMock.OfficeMockObject(mockData);

/* Code that calls the test framework goes below this line. */

// Jest test
test("getSelectedRangeAddress should return address of selected range",
async function () {
  expect(await
myOfficeAddinFeature.getSelectedRangeAddress(contextMock)).toBe("C2:G3");
});
```

Mocking a host object

This example assumes a Word add-in that has one of its features in a file named `my-word-add-in-feature.js`. The following shows the contents of the file.

JavaScript

```
const myWordAddinFeature = {

  insertBlueParagraph: async () => {
    return Word.run(async (context) => {
      // Insert a paragraph at the end of the document.
      const paragraph = context.document.body.insertParagraph("Hello World",
Word.InsertLocation.end);

      // Change the font color to blue.
      paragraph.font.color = "blue";

      await context.sync();
    });
  }
}

module.exports = myWordAddinFeature;
```

The test file, named `my-word-add-in-feature.test.js` is in a subfolder, relative to the location of the add-in code file. The following shows the contents of the file. Note that the top level property is `context`, a `ClientRequestContext` object, so the object that is being mocked is the parent of this property: a `Word` object. Note the following about this code:

- When the `OfficeMockObject` constructor creates the final mock object, it will ensure that the child `ClientRequestContext` object has `sync` and `load` methods.
- The `OfficeMockObject` constructor does *not* add a `run` function to the mock `Word` object, so it must be added explicitly in the seed object.
- The `OfficeMockObject` constructor does *not* add all of the Word enum classes to the mock `Word` object, so the `InsertLocation.end` value that is referenced in the add-in method must be added explicitly in the seed object.
- Because the Office JavaScript library isn't loaded in the node process, the `Word` object that is referenced in the add-in code must be declared and initialized.

JavaScript

```
const OfficeAddinMock = require("office-addin-mock");
const myWordAddinFeature = require("../my-word-add-in-feature");

// Create the seed mock object.
const mockData = {
  context: {
    document: {
      body: {
        paragraph: {
          font: {},
        },
```

```
        // Mock the Body.insertParagraph method.
        insertParagraph: function (paragraphText, insertLocation) {
          this.paragraph.text = paragraphText;
          this.paragraph.insertLocation = insertLocation;
          return this.paragraph;
        },
      },
    },
  },
  // Mock the Word.InsertLocation enum.
  InsertLocation: {
    end: "end",
  },
  // Mock the Word.run function.
  run: async function(callback) {
    await callback(this.context);
  },
};

// Create the final mock object from the seed object.
const wordMock = new OfficeAddinMock.OfficeMockObject(mockData);

// Define and initialize the Word object that is called in the
insertBlueParagraph function.
global.Word = wordMock;

/* Code that calls the test framework goes below this line. */

// Jest test set
describe("Insert blue paragraph at end tests", () => {

  test("color of paragraph", async function () {
    await myWordAddinFeature.insertBlueParagraph();

expect(wordMock.context.document.body.paragraph.font.color).toBe("blue");
  });

  test("text of paragraph", async function () {
    await myWordAddinFeature.insertBlueParagraph();
    expect(wordMock.context.document.body.paragraph.text).toBe("Hello
World");
  });
})
```

> **Note**
>
> Full reference documentation for the `OfficeMockObject` type is at **Office-Addin-Mock**.

See also

- Office-Addin-Mock npm page installation point.
- The open source repo is Office-Addin-Mock.
- Jest
- Mocha
- Jasmine

Usability testing for Office Add-ins

Article • 01/13/2025

A great add-in design takes user behaviors into account. Because your own preconceptions influence your design decisions, it's important to test designs with real users to make sure that your add-ins work well for your customers.

You can run usability tests in different ways. For many add-in developers, remote, unmoderated usability studies are the most time and cost effective. Popular testing services include:

- UserTesting.com
- Optimalworkshop.com
- Userzoom.com

These testing services help you to streamline test plan creation and remove the need to seek out participants or moderate the tests.

You need only five participants to uncover most usability issues in your design. Incorporate small tests regularly throughout your development cycle to ensure that your product is user-centered.

> **Note**
>
> We recommend that you test the usability of your add-in across multiple platforms. To publish your add-in to AppSource, it must work on all platforms that support the methods that you define.

1. Sign up for a testing service

For more information, see Selecting an Online Tool for Unmoderated Remote User Testing.

2. Develop your research questions

Research questions define the objectives of your research and guide your test plan. Your questions will help you identify participants to recruit and the tasks they'll perform. Understand when you need specific observations or broad input.

Specific question examples

- Do users notice the "free trial" link on the landing page?
- When users insert content from the add-in to their document, do they understand where in the document it's inserted?

Broad question examples

- What are the biggest pain points for the user in our add-in?
- Do users understand the meaning of the icons in our command bar, before they click on them?
- Can users easily find the settings menu?

User experience aspects

It's important to get data on the entire user journey – from discovering your add-in, to installing and using it. Consider research questions that address the following aspects of the add-in user experience.

- Finding your add-in in AppSource
- Choosing to install your add-in
- First-run experience
- Ribbon commands
- Add-in UI
- How the add-in interacts with the document space of the Office application
- How much control the user has over any content insertion flows

For more information, see Gathering factual responses vs. subjective data .

3. Identify participants to target

Remote testing services can give you control over many characteristics of your test participants. Think carefully about what kinds of users you want to target. In your early stages of data collection, it might be better to recruit a wide variety of participants to identify more obvious usability issues. Later, you might choose to target groups like advanced Office users, particular occupations, or specific age ranges.

4. Create the participant screener

The screener is the set of questions and requirements you present to prospective test participants to screen them for your test. Keep in mind that participants for services like

UserTesting.com have a financial interest in qualifying for your test. It's a good idea to include trick questions in your screener if you want to exclude certain users from the test.

For example, if you want to find participants who are familiar with GitHub, to filter out users who might misrepresent themselves, include fakes in the list of possible answers.

Which of the following source code repositories are you familiar with?
a. SourceShelf [*Reject*]
b. CodeContainer [*Reject*]
c. GitHub [*Must select*]
d. BitBucket [*May select*]
e. CloudForge [*May select*]

If you're planning to test a live build of your add-in, the following questions can screen for users who will be able to do this.

This test requires you to have the latest version of Microsoft PowerPoint. Do you have the latest version of PowerPoint?
a. Yes [*Must select*]
b. No [*Reject*]
c. I don't know [*Reject*]

This test requires you to install a free add-in for PowerPoint, and create a free account to use it. Are you willing to install an add-in and create a free account?
a. Yes [*Must select*]
b. No [*Reject*]

For more information, see Screener Questions Best Practices.

5. Create tasks and questions for participants

Try to prioritize what you want tested so that you can limit the number of tasks and questions for the participant. Some services pay participants only for a set amount of time, so you want to make sure not to go over.

Try to observe participant behaviors instead of asking about them, whenever possible. If you need to ask about behaviors, ask about what participants have done in the past, rather than what they would expect to do in a situation. This tends to give more reliable results.

The main challenge in unmoderated testing is making sure your participants understand your tasks and scenarios. Your directions should be *clear and concise*. Inevitably, if

there's potential for confusion, someone will be confused.

Don't assume that your user will be on the screen they're supposed to be on at any given point during the test. Consider telling them what screen they need to be on to start the next task.

For more information, see Writing Great Tasks.

6. Create a prototype to match the tasks and questions

You can either test your live add-in, or you can test a prototype. Keep in mind that if you want to test the live add-in, you need to screen for participants that have the latest version of Office, are willing to install the add-in, and are willing to sign up for an account (unless you have logon credentials to provide them.) You'll then need to make sure that they successfully install your add-in.

On average, it takes about 5 minutes to walk users through how to install an add-in. The following is an example of clear, concise installation steps. Adjust the steps based on the specifics of your test.

Please install the (insert your add-in name here) add-in for PowerPoint, using the following instructions.

1. Open Microsoft PowerPoint.
2. Select **Blank Presentation.**
3. Select **Home** > **Add-ins**, then select **Get Add-ins**.
4. In the popup window, choose **Store**.
5. Type (Add-in name) in the search box.
6. Choose (Add-in name).
7. Take a moment to look at the Store page to familiarize yourself with the add-in.
8. Choose **Add** to install the add-in.

You can test a prototype at any level of interaction and visual fidelity. For more complex linking and interactivity, consider a prototyping tool like Figma. If you just want to test static screens, you can host images online and send participants the corresponding URL, or give them a link to an online PowerPoint presentation.

7. Run a pilot test

It can be tricky to get the prototype and your task/question list right. Users might be confused by tasks, or might get lost in your prototype. You should run a pilot test with

1-3 users to work out the inevitable issues with the test format. This will help to ensure that your questions are clear, that the prototype is set up correctly, and that you're capturing the type of data you're looking for.

8. Run the test

After you order your test, you'll get email notifications when participants complete it. Unless you've targeted a specific group of participants, the tests are usually completed within a few hours.

9. Analyze results

This is the part where you try to make sense of the data you've collected. While watching the test videos, record notes about problems and successes the user has. Avoid trying to interpret the meaning of the data until you have viewed all the results.

A single participant having a usability issue isn't enough to warrant making a change to the design. Two or more participants encountering the same issue suggests that other users in the general population will also encounter that issue.

In general, be careful about how you use your data to draw conclusions. Don't fall into the trap of trying to make the data fit a certain narrative; be honest about what the data actually proves, disproves, or simply fails to provide any insight about. Keep an open mind; user behavior frequently defies designer's expectations.

See also

- How to Conduct Usability Testing
- Best Practices for UserTesting
- Minimizing Bias

Validate an Office Add-in's manifest

Article • 05/19/2025

You should validate your add-in's manifest file to ensure that it's correct and complete. Validation can also identify issues that are causing the error "Your add-in manifest is not valid" when you attempt to sideload your add-in. This article describes multiple ways to validate the manifest file. Except as specified otherwise, they work for both the unified manifest for Microsoft 365 and the add-in only manifest.

> **Note**
>
> For details about using runtime logging to troubleshoot issues with your add-in's manifest, see Debug your add-in with runtime logging.

Validate your manifest with the validate command

If you used Microsoft 365 Agents Toolkit or Yeoman generator for Office Add-ins to create your add-in, you can validate your project's manifest file with the following command in the root directory of your project.

command line

```
npm run validate
```

Microsoft 365 and Copilot store validation

The `validate` command also does Microsoft 365 and Copilot store validation but allows developer information like localhost URLs. If you'd like to run production-level Microsoft 365 and Copilot store validation, then run the following command.

command line

```
npm run validate -- -p
```

If you're having trouble with that command, try the following (replacing `MANIFEST_FILE` with the name of the manifest file).

command line

```
npx office-addin-manifest validate -p MANIFEST_FILE
```

Validate your manifest with office-addin-manifest

If you didn't use Microsoft 365 Agents Toolkit or Yeoman generator for Office Add-ins to create your add-in, you can validate the manifest by using office-addin-manifest.

1. Install Node.js.

2. Open a command prompt and install the validator with the following command.

   command line

   ```
   npm install -g office-addin-manifest
   ```

3. Run the following command *in the folder of your project that contains the manifest file* (replacing `MANIFEST_FILE` with the name of the manifest file).

   command line

   ```
   office-addin-manifest validate MANIFEST_FILE
   ```

   Note

   If this command isn't working, run the following command instead to force the use of the latest version of the office-addin-manifest tool (replacing `MANIFEST_FILE` with the name of the manifest file).

   command line

   ```
   npx office-addin-manifest validate MANIFEST_FILE
   ```

Validate the manifest in the UI of Agents Toolkit

If you're working in Agents Toolkit and using the unified manifest, you can use the toolkit's validation options. For instructions, see Validate application.

See also

- Office Add-ins manifest
- Clear the Office cache
- Debug your add-in with runtime logging