BUILDING OFFICE ADD-INS USING OFFICE.JS

MICHAEL ZLATKOVSKY

Building Office Add-ins using Office.js

Michael Zlatkovsky

This book is for sale at http://leanpub.com/buildingofficeaddins

This version was published on 2017-03-06

Tweet This Book!

Please help Michael Zlatkovsky by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Building Office Add-ins using Office.js #buildingofficeaddins #addins #officejs #reading

The suggested hashtag for this book is #buildingofficeaddins.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#buildingofficeaddins

Contents

1. The book and its structure

1.1 The “evergreen”, in-progress book

This sample contains *a selection of topics* from the “Building Office Add-ins using Office.js” book.

The book itself is published using a “lean” methodology – publishing early, and publishing often. It is an evergreen, in-progress book.

Since writing the book is strictly my “moonlighting” activity – my day-job is to be a *developer* on Office.js APIs, not a technical writer – it will take a good long while before I am actually “done” with all the content I want to write about. I would rather ship an “early” version of the book, addressing the most common questions and issues that I see folks struggling with, and then keep iterating on it from there.

In buying this e-book through LeanPub (where I’m also discounting it for the “early readers”), **you are entitled to free updates to the book**. So, expect to see more content appear month-to-month (and, if you sign up for email notification, to receive periodic emails updates about what new content got added).

I welcome any and all feedback about the writing, explanations, code-samples, and whatever else. I also welcome topic suggestions, if you see something I didn’t cover; or, if you see something that I *plan to cover*, but haven’t yet, just let me know and I can see if I can prioritize it. Section “*Bug reports / topic suggestions*” below will describe the best way for filing these sorts of issues/suggestions. Or, if you want to reach me directly, you can do so at michael@buildingofficeaddins.com.

Yours truly,

~ Michael Zlatkovsky The author

1.2 Release notes

You are currently reading version **1.2** of the book. Release notes for major updates are posted on **http://buildingofficeaddins.com/release-notes**

Version 1.2 (*Feb 20, 2017*)

- Added a topic on the different flavors of Office 2016 / Office 365 – and the practical implications for developers.
- Added a topic on API Versioning and Requirement Sets.
- Greatly expanded the "TypeScript-based Add-ins" topic, adding instructions for the updated Yeoman generator.
- Added a topic for attaching the debugger to Add-ins (breakpoints, DOM explorer, etc.)
- Added a link to the book's companion Twitter account.
- Addressed a number of other reader-reported issues (view the already-addressed ones, or file your issues or topic requests, at http://buildingofficeaddins.com/issues).

Version 1.1 (*Jan 22, 2017*)

- Re-pivoted the book around TypeScript and the `async/await` syntax. Moved the JS-specific content to a separate Appendix
- Added an information-packed JavaScript & TypeScript crash-course, tailored specifically at Office.js concepts, for those who are new to the world of JS/TS.
- With TypeScript now a first-class citizen of the book, added "*Getting started with building TypeScript-based Add-ins*" (section 3.2). I expect to continue to expand this section in future releases.
- Added an in-depth explanation of the internal workings of the Office.js pipeline and proxy-object model. See "*Implementation details, for those who want to know how it really works*" (section 5.5).

- Re-arranged, edited, and added to the content of the "*Office.js APIs: Core concepts*" chapter (chapter 5).
- Added links to downloadable code samples, for a few of the larger samples.
- Addressed a variety of smaller feedback items.

Updates to the book are free for all existing readers (which is what lean-publishing, and the concept of an evergreen book, is all about!). Simply go to **https://leanpub.com/user_dashboard/library**, select the book, and download your newly-updated copy!

1.3 Bug reports / topic suggestions

Having now had experience in both, I think that writing an [evergreen] book is akin to writing an [evergreen] add-in / website. Try as you may, there will be bugs; and there will also be not-yet-implemented features, or new ideas that simply hadn't occurred before.

To this end, I'd like to provide readers with a way to easily log and track content issues and topic suggestions. Issues can be:

- Simple content issues: A misspelling, an incomplete phrase, a sentence that no longer makes sense.
- Requests for additional detail in *existing* topics.
- Requests for *brand new topics*. I might already be planning to write something about it eventually, but this will let you subscribe to finding out when the content is made available; and will also help me gauge topic interest.
- Issues with sample code or related assets: code is unclear; the design is sub optimal; something needs a comment; or perhaps an existing comment needs to be clarified or removed.
- Anything else?

The issue repo can be accessed through **http://buildingofficeaddins.com/issues**

1.4 Twitter

As reader of the evergreen book, you will receive periodic updates from LeanPub when I publish a major version (unless you opt out, that is). I expect to send out such communications once every month or two.

If you'd like to receive more frequent status updates – both about the book, articles that I put up on my site, interesting StackOverflow questions, or re-tweets of interesting updates about Office Add-ins that I find on the web – I encourage you to follow my twitter feed at

https://twitter.com/BuildingAddins

or view the feed embedded on my site, at

http://buildingofficeaddins.com/tweets/

By the same token, if you blog or tweet about the book, I would be much obliged if you can use the tag **#buildingofficeaddins** and/or **officejs**, and also @mention me: **@BuildingAddins**.

1.5 Who should read this book

This book is aimed at the *professional developer* who is tasked with creating an Office Add-in (a.k.a an *Office Web Add-in*, and formerly known as an *App for Office*). In particular, it is aimed at the "*new*" Office 2016+ wave of Office.js APIs – which, at the time of writing, is supported in Word, Excel, and OneNote[1].

Office has a rich and wonderful legacy of programmability, with VBA spanning the full gamut of developer skill levels – from novice programmers tweaking simple macros, to professional developers writing complex Office customization. But for purposes of this book (and due to the more complex nature of the underlying Office Add-ins platform – for now, anyway), it is really the *professional developer* that is the target audience. With that definition, I mean someone who is well-versed in code, who is eager to learn something new, and who is unfazed by occasional difficulty (which you will naturally get, being on the cutting edge of a developing Office Add-ins platform). A working knowledge of JavaScript is a plus, as is power-user knowledge of the Office application that you are targeting.

Importantly, this book is *not* meant as a "standard" API reference manual, which might walk you through a series of tasks or APIs, focusing on the particulars. From my perspective, we have dozens of object types, and hundreds of methods and properties, in each of the Office hosts, all of these are dutifully documented in our online API reference documentation. To put these in print, in the form of a static book, would serve very little use.

Instead, this book is about the underlying principles that these APIs, however diverse, have in common. It is about the conceptual threads that bind the API surface area with the common runtime that they share. My goal is to paint the overarching picture of the API model, and to zoom in on details that transcend beyond any given API object. In this regard, this book is about going

[1] As of December 2016, Word, Excel, and OneNote have all adopted the *new 2016 wave* of Office.js APIs. Outlook – though steadily continuing to expand its APIs – is continuing to use the "Office 2013" style of APIs even in its Office 2016 applications (partially because many of its scenarios are more about data consumption, and less about object-by-object automation, which is where the new model would shine). PowerPoint and Access have, so far, remained unchanged in their API surface relative to 2013; and Project, though it has added a number of APIs, is also still using the "Office 2013" style.

from copy-pasting online sample code, to understanding the key concepts that make these samples work. In short, this book is about writing effective, efficient, testable code, and the tools that will help you do so.

As you'll read in the introduction to Office Add-ins, Office 2016 – and, of course, its Office Online & Mac & iOS equivalents – represents a major re-birth of the APIs, with each of the supported hosts getting a custom-tailored object model that dives far deeper than the original set of Office 2013's "common APIs". This *new* (2016+) wave of Office.js APIs is the subject of this book. You may find that a part of the content in this book â€“ especially the more general topics like an introduction to Office Add-ins, prerequisites, debugging techniques, pro tips, and various "beyond-API" topics â€“ will apply to both the 2013 and 2016 Office API models, but the book's focus and *raison d'etre* is the *new Office 2016 wave* of Office.js APIs.

1.6 Book structure

Part 1: Introduction, pre-requisites, and JavaScript & Promises primer

The first part of the book set the stage for learning about Office Add-ins.

This first chapter is a brief introduction to the book, its author, and the information that will be presented over the course of the book.

The second chapter serves as an introduction to what Office Add-ins are, how they're different from VBA/VSTO/COM, and what "async" is all about. It is useful information for getting yourself oriented in the Office space, especially if you're coming from VBA or VSTO. At the very minimum, I would recommend reading the section on why Office.js has an asynchronous programming model, as "async" permeates the entirety of our APIs.

The third chapter is about pre-requisites: getting your developer environment, ensuring you have the right version of Office, referencing the CDN, configuring IntelliSense, and so forth. You will probably want to return to this chapter when you have particular issues (i.e., lack of IntelliSense, or wanting to switch to a Beta CDN), but you can probably skim/skip through it on your first read.

The fourth chapter is a JavaScript and Promises primer, for those who are new(ish) to JavaScript and/or async-programming and/or Promises. If you're a JS pro, you can safely skip this chapter – there is nothing Office.js-specific here. On the other hand, if you're *not*, I highly recommend reading this chapter and probably returning to it a time or two later for reference. JavaScript concepts, and especially Promises, are 100% prevalent and essential for understanding Office.js. Jumping straight ahead to Office.js concepts without understanding the JavaScript & async-programming basics, is a bit like going off a ski jump before getting your ski footing on solid ground first[a].

[a] Says the person who, in high school, effectively did just that on his first day at a ski resort... And so can speak with great authority on the subject.

Part 2: Office.js API Concepts:

This middle part is the meat of the book's content. [*Note that some of chapters # 6 - 8 have been written, but haven't been ported to LeanPub yet, son only chapter 5 is currently available. Stay tuned...*]

Chapter 5 starts off with core Office.js *new API* concepts. It will guide you through a canonical code sample, and then cover the most essential basics of the new Office.js model: understanding proxy objects, loading properties, syncing changes to the underlying document, and catching errors.

Chapter 6 goes beyond the basics, and dives much deeper into the topics first covered in Chapter 5. When and how to load the proxy objects most effectively? How to break asynchronous work across functions, and how to detect if you accidentally "broke the Promise chain"? Why are there different ways to access items on collections, and when should you use one over the other? The chapter also explores the details of how Office.js code executes, covering topics like how write more efficient code, best ways to await user input, and how to use objects outside of the "linear" flow of batch-execution.

Chapter 7 covers some common mistakes for concepts that were covered in the preceding two chapters, but that are none-the-less very common to make. The chapter is effectively a mix between quizzing the reader with real-world code samples I've seen people write, and then reviewing the underlying concepts that were incorrect in the code.

Chapter 8 covers other Office.js API topics that are not *directly* tied to the "new APIs", but are essential when developing the Add-ins. Topics include the organization of APIs and their corresponding namespaces, a note on internal vs. external APIs, instructions for calling web services and performing authentication, and a description of how to use TypeScript within your Office Add-in project. The chapter also covers the "meta"-topics of how to contribute to the API Design process, and where & how to report bugs.

Part 3: Debugging, Styling, and Publishing Add-ins

The last part of the book shifts focus from the particulars of Office.js APIs, and instead focuses on the overall development experience for creating Office Add-ins. *[Note: these topics are *planned**, but have not been written yet.]*

Chapter 9 dives into the world of Office.js debugging, exploring the error information that is available on failures, providing instructions on attaching and using a debugger, offering debugging tips that are specific to Add-in Commands (ribbon customization) and Dialogs, and suggesting a handy method for exposing a "debug mode" for your add-in.

Chapter 10 offers some "User-Interface" (UI) advice: Covering common UX patterns, drilling into the topic of surfacing errors to users, and explaining how to use frameworks such as Angular.js and Office's Fabric UI framework for providing compelling user interfaces.

Finally, Chapter 11 cover topic related to publishing Office Add-ins. The chapter explains what's involved in the "publishing" process, and discusses the differences between publishing internally (within an organization), versus publishing to the Store for public consumption. The chapter also touches upon how to license (and validate proper licensing status) of a store-bound Add-in, how to protect the Intellectual Property within your code, and how to optimize the load time of your Add-in via minification & packing.

1.7 A few brief notes

A few brief notes before we get started:

- This book assumes some general programming experience, and ideally a working knowledge of JavaScript. "Chapter 4. JavaScript & Promises primer (as pertaining to our APIs)" can get you somewhat up to speed, but if you're intending to develop add-ins professionally, you'll likely want a more formal JS training / book.
- While you may need various other web skills (HTML & CSS expertise, experience writing back-end services and databases, and so forth) in order to create the full-fledged Add-in, these topics are not Office Add-in specific, and are covered in countless books and web articles. Beyond the occasional reference or link – or just enough explanation to show an example – this book will focus mostly on the "*Office.js*" part of writing Office Add-ins.
- You should have a reasonably working knowledge (and ideally, "Power User" knowledge) of the host application that you will be targeting. Here and elsewhere throughout the book, "host application" refers to the Office application that your add-in will be running in (Word, Excel, etc.). After all, the host application is what your end-users will use, and so you'll need to have the expertise to test that it all works under all circumstances! An example: in Excel, worksheet names can only contain up to 31 characters, and may not include a number of reserved characters. This means that, to avoid having the application stop midway through an operation, you will want to pre-validate the user input, or handle the error through other means.
- The concepts and tools throughout this book are applicable to all of the Office applications that have adopted the "new" Office.js model (which, at the time of writing, is Excel, Word, and OneNote). In this book, I will generally use Excel-based samples, both because I love Excel, and because – at least for VSTO – more developers were programming against Excel than against any of the other hosts. But again, if you substitute *Word* or *OneNote* in place of *Excel* in the sample code – and if you substitute in the analogous objects – the exact same concepts will apply. I'll try to sprinkle in some Word examples too, for fairness' sake.

- Finally, while the book is written by someone who works on the Office Extensibility Platform team at Microsoft, the information and viewpoints presented in this book represent the *viewpoints of the author*, not of Microsoft as a company.

2. Introduction to Office Add-ins

Chapter structure & planned content

Note:
- Sections that are part of this sample are **bolded and hyperlinked**
- Sections that aren't part of the sample, but *are* finished and are part of the book are in **bold**
The rest are planned topics, but have not been written (or polished) yet.

- Office Add-ins (a.k.a. Office Web Add-ins)
  - What are they, and how are they different from other Office Extensibility technologies?
  - Web Add-ins? What is meant by "web", and what technologies are involved?
  - Types of Office Web Add-ins (content add-ins, task-pane add-ins, add-in commands)
- **What's new in the Office 2016 APIs (relative to 2013)?**
- **What about VBA, VSTO, & COM add-ins?**
- **"But can Office.js do XYZ?"**
- **A word on JavaScript and TypeScript**
- **Office.js: The asynchronous / deferred-execution programming model**
  - **Why is Office.js async?**
  - **What is meant by "the server"?**

2.1 What's "new" in the Office 2016 APIs (relative to 2013)?

As you'll read in greater detail in *"Office.js: The asynchronous / deferred-execution programming model"*, the Office.js API model was first introduced in Office 2013, under the name of "Apps for Office". The model was an ambitious – if somewhat limited –- attempt to create a new kind of web technology based APIs, and have those APIs work cross platform and across multiple host applications.

Office 2016 has seen a complete overhaul of the original 2013 API model, creating new and host specific APIs. These APIs offer a rich client-side object model that accurately reflects the unique characteristics of each object type for each of the hosts. For example, the Excel APIs have classes for Worksheets, Ranges, Tables, Charts, and so forth, with each object type offering dozens of properties and methods. Similarly, the Word API now have the notion of Paragraphs, and Content Controls, and Images, and more. The team has also changed the overarching design process, building out the APIs with great transparency in the form of "open specs", which are posted online for the community to give feedback on before implementation ever takes place. In a similar vein, several colleagues on my team – and an increasing number of folks from the external community – are active participants on StackOverflow, where we actively monitor any questions that are tagged with "**office-js**".

The Office 2016 release also signifies a much deeper commitment to cross-platform consistency in the APIs. For example, all of the new APIs in Excel that are available in Office 2016 for Windows are also available in Excel Online, Excel for Mac and Excel on iOS. To keep up with developers' demands, the new APIs are shipped continuously, usually once per quarter, with those APIs becoming immediately available to users of Office 365[1] on the Desktop, on Office Online, and Mac/iOS.

Even beyond the APIs, Office 2016 offers enhancements to other parts of the overall programming framework – most notably, the ability for Office Add-ins to appear in the ribbon and to launch dialog windows. If you were previously

[1] For more information on the "MSI" vs. subscription-based Office installations, see *"Office versions: Office 2016 vs. Office 365 (Click-to-Run vs. MSI), Deferred vs. Current channel"*.

on the fence for whether to look into Office Add-ins, I hope that the APIs described in this book will help change your mind.

2.2 Office.js: The asynchronous / deferred-execution programming model

2.2.1 Why is Office.js async?

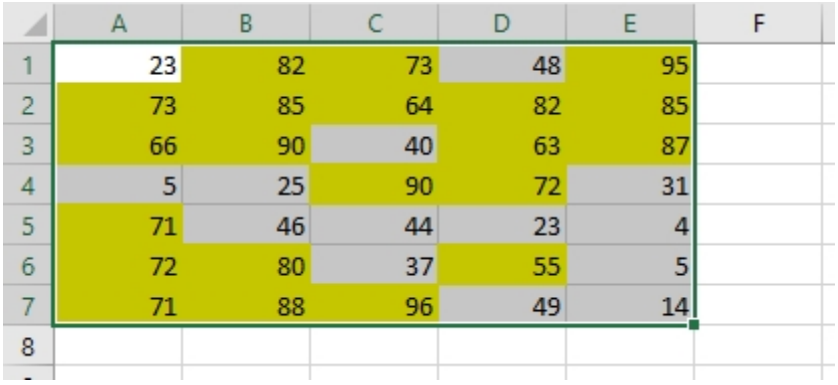
For those who have used VBA before, you will know that VBA code was always executed in a linear (synchronous) fashion. This is very natural for an automation task, where you're essentially manipulating a document through a series of steps (and where, more often than not, the steps are similar to the sequential series of steps that a human would do). For example, if you needed to analyze the current selection in Excel and highlight any values that were greater than 50, you might write something like this:

VBA macro for highlighting values over 50

```
Dim selectionRange As Range
Set selectionRange = Selection
Call selectionRange.ClearFormats

Dim row As Integer
Dim column As Integer
Dim cell As Range

For row = 1 To selectionRange.Rows.Count
    For column = 1 To selectionRange.Columns.Count
        Set cell = selectionRange.Cells(row, column)
        If cell.Value > 50 Then
            cell.Interior.Color = RGB(255, 255, 0)
        End If
    Next column
Next row
```



	A	B	C	D	E	F
1	23	82	73	48	95	
2	73	85	64	82	85	
3	66	90	40	63	87	
4	5	25	90	72	31	
5	71	46	44	23	4	
6	72	80	37	55	5	
7	71	88	96	49	14	
8						

Screenshot of the code in action

When run, such macro would execute line by line, reading cell values and manipulating them as it went. The macro have complete access to the in-memory representation of the workbook, and would run almost at the native Excel level, blocking out any other native Excel operations (or, for that matter, any user operations, since the VBA code executes on the UI thread).

With the rise of .NET, VSTO – Visual Studio Tools for Office – was introduced. VSTO still used the same underlying APIs that VBA accessed, and it still ran those APIs in a synchronous line-by-line fashion. However, in order to isolate the VSTO add-in from Excel – so that a faulty add-in would not crash Excel, and so that add-ins could be resilient against each other in a multi-add-in environment – VSTO had each code solution run within its own fully-isolated AppDomain. So while .NET code itself ran very fast – faster than VBA by pure numbers – the Object-Model calls into Excel suddenly incurred a significant cost of cross-domain marshaling (resulting in a ~3x slowdown compared to VBA, in my experience).

Thus, the VBA code above – translated into its VB.NET or C# equivalent – would continue to work, but it would run far less efficiently, since each subsequent *read* and *write* call would have to traverse the process boundary. To ameliorate that, the VSTO incarnation of the code could be made significantly faster if all of the *read* operations were lumped into a single *read* call at the very beginning. (The *write* operations, of setting the background of each cell

one-by-one, would still have to remain as individually-dispatched calls[2]).

Here is C# Version of the code that addresses the above scenario, but this time reads all of the cell values in bulk (see line **#4** below).

VSTO incarnation of the code, with bulk-reading of the selection values (line #4)

```
1  Range selectionRange = Globals.ThisAddIn.Application.Selection;
2  selectionRange.ClearFormats();
3
4  object[,] selectionValues = selectionRange.Value;
5
6  int rowCount = selectionValues.GetLength(0);
7  int columnCount = selectionValues.GetLength(1);
8
9  for (int row = 1; row <= rowCount; row++)
10 {
11     for (int column = 1; column <= columnCount; column++)
12     {
13         if (Convert.ToInt32(selectionValues[row, column]) > 50)
14         {
15             Range cell = selectionRange.Cells[row, column];
16             cell.Interior.Color = System.Drawing.Color.Yellow;
17         }
18     }
19 }
```

Note that the VSTO code, when interacting with the documents, still runs on (a.k.a., "blocks") the Excel UI thread, since – due to the way that Excel (and Word, and others) are architected – all operations that manipulate the document run on the UI thread. But fortunately for VSTO, the process-boundary cost – while an order of magnitude higher than that of VBA – is still relatively small in the grand scheme of things, and the batch-reading

[2]Technically not 100% true, since you could create a multi-area range and perform a single "write" operation to it – but there is a limitation on how many cells you can group together, and the performance boost is still not nearly as good as what the VBA performance would have been. There are some APIs, like reading and writing to values or formulas, that can accept bulk input/output, but most of the rest – like formatting – must be done on a range-by-range basis.

technique described above can alleviate a chunk of the performance issues… and so, VSTO could continue to use the same APIs in the same synchronous fashion as VBA, while letting developers use the new goodness of the .NET Framework.

With the introduction of Office Add-ins (known at the time as "Apps for Office") in Office 2013, the interaction between the add-in and the host application had to be re-thought. Office Add-ins, based in HTML and JavaScript and CSS, needed to run inside of a browser container. On Desktop, the browser container is an embedded Internet Explorer process[3], and perhaps the synchronous model could still have worked there, even if it took another performance hit. But the real death knell to synchronous Office.js programming came from the need to support the Office Online applications, such as Word Online, Excel Online, etc. In those applications, the Office Add-ins runs inside of an HTML iframe, which in turn is hosted by the *parent* HTML page that represents the document editor. While the iframe and its parent are at least part of the same browser window, the problem is that the bulk of the documents might not – and generally is *not* – loaded in the browser's memory [4]. This means that for most operations [5], the request would have to travel from the iframe to the parent HTML page, all the way to an Office 365 web server running in a remote data center. The request would then get executed on the server, which would dutifully send the response back to the waiting HTML page, which would pass it on to the iframe, which would finally invoke the Add-in code. Not surprisingly, such round-trip cost is not cheap.

[3] If you're curious for how the interaction between Office and the embedded Internet Explorer control is done: it is through a `window.external` API that IE provides, which acts as the pipe between IE and the process that created it. The same technique is possible in .NET and WinForms/WPF applications as well. See https://msdn.microsoft.com/en-us/library/system.windows.forms.webbrowser.objectforscripting(v=vs.110).aspx for more info on the latter.

[4] Imagine a 50MB Word document, complete a bunch of images. Does it make sense for the browser to receive all 50MB at once, or could it progressively load only the pages that it needs in the neighboring vicinity, and only serve up the compressed and optimized, rather than raw, copies of the images?

[5] The actual amount of code that can be run locally (i.e., does not require a roundtrip to the server) varies greatly depending on the host application. On one extreme end of the spectrum, Excel Online requires that pretty much *all* operations are executed remotely. On the opposite side, OneNote Online has a pretty good local cache of the document, and needs to call out to the server much less frequently.

To put it into perspective: imagine that the entire roundtrip described above takes 50 milliseconds, and we are running a hypothetical synchronous and JavaScript-icized version of the VSTO macro. Imagine that we have one hundred cells, of which 50 meet the criteria for needing to be highlighted. This would mean that we need to make one request to clear the formatting from the selection, another to fetch all of the data, and then 50 requests for each time that we set on individual cell's color. This means that the operation would take (2 + 50) * 50 milliseconds, or just over 2.5 seconds. Perhaps that doesn't sound all that terrible… but then again, we were operating on a mere 100 cells! For 1000 cells, we'd be looking at 25 seconds, and for 10,000 cells we would be at over four minutes. What would the user be doing – other than sitting back in this chair and sipping coffee – while waiting for the Add-in operation to complete?!

If synchronous programming was out, the only remaining choice was asynchrony. In the **Office 2013** model, this was embodied by a whole bunch of methods that ended with the word "Async", such as:

```
Office.context.document.setSelectedDataAsync(data, callback);
```

In this **Office 2013** design, ***every operation was a standalone call*** that was dispatched to the Office host application. The browser would then wait to be notified that the operation completed (sometimes merely waiting for notification, other times waiting for data to be returned back), before calling the callback function.

Thus, while the Office 2013 APIs solved the Async problem, the solution was very much a request-based Async solution, akin to server web requests, but not the sort of automation scenarios that VBA users were accustomed to. Moreover, the API design itself was limiting, as there were almost no backing objects to represent the richness of the Office document. The omission was no accident: a rich object model implies objects that have countless properties and methods, but making each of them an async call would have been not only cumbersome to use, but also highly inefficient. The user would still be waiting their 2.5 seconds or 25 seconds, or 4+ minutes for the operation to complete, albeit without having their browser window frozen.

The new **Office 2016 API model** offers a radical departure from the Office 2013 design. The object model – now under the `Excel` namespace for Excel,

`Word` for Word, `OneNote` for OneNote, etc., – is backed by strongly-typed object-oriented classes, with similar methods and properties to what you'd see in VBA. Interaction with the properties or methods is also simple and sequential, similar in spirit to what you'd do in VBA or VSTO code.

Whoa! How is this possible? The catch is that, underneath the covers, setting properties or methods **adds them to a queue of pending changes, but *doesn't dispatch them* until an explicit `.sync()` request. That is, the `.sync()` call is the *only* asynchrony in the whole system**. When this `sync()` method is called, any queued-up changes are dispatched to the document, and any data that was requested to be loaded is received and injected into the objects that requested it. Take a look at this incarnation of the cell-highlighting scenario, this time written using the new Office.js paradigm, in JavaScript:

Office.js incarnation of the VBA macro, with blocks of still-seemingly-synchronous code

```
Excel.run(function (context) {
    var selectionRange = context.workbook.getSelectedRange();
    selectionRange.format.fill.clear();

    selectionRange.load("values");

    return context.sync()
        .then(function () {
            var rowCount = selectionRange.values.length;
            var columnCount = selectionRange.values[0].length;
            for (var row = 0; row < rowCount; row++) {
                for (var column = 0; column < columnCount; column ++) {
                    if (selectionRange.values[row][column] > 50) {
                        selectionRange.getCell(row, column)
                            .format.fill.color = "yellow";
                    }
                }
            }
        })
        .then(context.sync);

}).catch(OfficeHelpers.Utilities.log);
```

As you can see, the code is pretty straightforward to read. Sure, there are the unfamiliar concepts of `load` and `sync`, but if you squint over the `load` and `sync` statements and the `Excel.run` wrapper (i.e., only look at lines **#2-3**, and then **#9-18**), **you still have seemingly-synchronous code with a familiar-looking object model**.

If instead of plain JavaScript you use TypeScript (see *A word on JavaScript and TypeScript*), the Office.js code becomes even cleaner.

*The same Office.js rendition, but this time making use of **TypeScript 2.1's 'async/await'** feature*

```
1  Excel.run(async function (context) {
2      let selectionRange = context.workbook.getSelectedRange();
3      selectionRange.format.fill.clear();
4
5      selectionRange.load("values");
6      await context.sync();
7
8      let rowCount = selectionRange.values.length;
9      let columnCount = selectionRange.values[0].length;
10     for (let row = 0; row < rowCount; row++) {
11         for (let column = 0; column < columnCount; column ++) {
12             if (selectionRange.values[row][column] > 50) {
13                 selectionRange.getCell(row, column)
14                     .format.fill.color = "yellow";
15             }
16         }
17     }
18
19     await context.sync();
20
21 }).catch(OfficeHelpers.Utilities.log);
```

In fact, if you ignore the `Excel.run` wrapper code (the first and last lines), and if you squint over the `load` and `sync` statements (lines **#5-6** and **#18**), the code looks reasonably similar to what you'd expect to write in VBA or VSTO!

Still, programming using the new Office.js model – and, for VBA or VSTO users, adapting to some of the differences – has a definite learning curve. This

book will guide you through getting started, understanding the API basics, learning the unconventional tricks of debugging Office.js code, and grasping the key concepts for writing performant and reliable add-ins. It will also give tips on making your development experience easier, on utilizing some of the lesser-known functions of Office.js framework, and on writing testable code. Finally, it will address some frequently asked questions, and give guidance on broader topics such as how to call external services, how to authenticate within Office add-ins, and how to publish and license your Add-ins.

3. Prerequisites

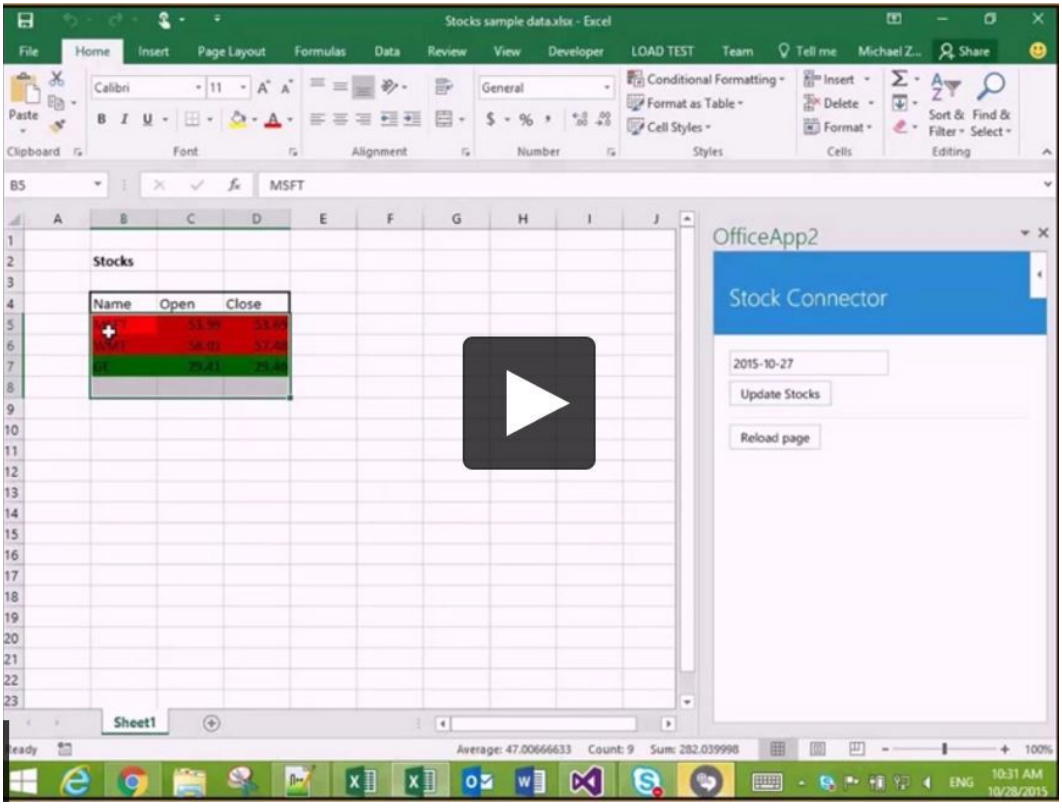
Chapter structure & planned content

Note:
- Sections that are finished and are part of this sample are **bolded and hyperlinked**
The rest are planned topics, but have not been written (or polished) yet.

- Configuring your Dev environment
- **A video walkthrough of building an Add-in using Visual Studio**
- **Getting started with building TypeScript-based add-ins**
  - **Using Visual Studio**
  - **Using Yeoman generator & Node/NPM**
- **Office versions: Office 2016 vs. Office 365 (MSI vs. Click-to-Run); Deferred vs. Current channels; Insider tracks**
- **Office.js API versioning**
  - **The JavaScript files**
  - **The host capabilities**
  - **The Beta Endpoint**
  - **How can you know than an API is “production-ready”?**
- **Debugging: the bare basics**
- Where to find Office.js? CDN, Beta CDN, local copy
- IntelliSense
  - JavaScript IntelliSense
  - TypeScript IntelliSense
- Resources
- Following along with code samples

3.1 Walkthrough: Building an Add-in using Visual Studio

If you're just getting started and want to use Visual Studio, I highly recommend watching a walkthrough tutorial that I recorded in late 2015: https://channel9.msdn.com/series/officejs/End-to-End-Walkthrough-of-Excel-JavaScript-Add-in-Development.



In the video, I walk through the end-to-end process of building an Office Add-in for Excel: from launching Visual Studio, to writing a bit of JavaScript code that uses the new Excel 2016 APIs, to adding some basic UI tweaks, to talking through the publishing options, debugging, and more.

The video touches on some API topics that are covered in much greater detail in this book – but it also shows the process of creating a project and debugging

using Visual Studio, which is crucial for getting started. If you've not built an Office Add-in before, I highly recommend the video.

For those looking for written instruction, just on the Visual Studio piece: there is also official documentation for creating a project on https://dev.office.com/docs/add-ins/get-started/create-and-debug-office-add-ins-in-visual-studio.

3.2 Getting started with building TypeScript-based Add-ins

As noted earlier, I firmly believe that TypeScript (as opposed to *plain* JavaScript) offers the premier Add-in coding experience. Depending on how comfortable you are with the emerging web technologies (i.e., Node, NPM, etc), you can either use the Office Yeoman generator to create a typescript-based project, or you can tweak a Visual-Studio-created JS project to convert it to TypeScript.

3.2.1 Using Visual Studio

Currently, the Visual Studio templates for Office Add-ins come only in a JavaScript-based flavor. Fortunately, it does not take much setup to convert the project to TypeScript. To do so, using Visual Studio, create the project *as if it's a regular JavaScript-based Add-ins project*, and then follow the few steps described in this excellent step-by-step blog-post: http://simonjaeger.com/use-typescript-in-a-visual-studio-office-add-in-project/. Once you've done it once, it's you'll see that it only takes a minute or two to do the next time, and is well-worth the trouble.

To get IntelliSense, be sure to add the Office.js d.ts (TypeScript definitions) file, available from https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/office-js/index.d.ts[1]. You can either copy-paste the file manually into your project (and check back periodically to ensure that you have the latest version), or install it via a NuGet package, which will make the package easier to keep up-to-date.

Note that to use the *async/await* features, you'll need to ensure you have TypeScript 2.1 or higher. For integration with Visual Studio, simply download a package from https://www.typescriptlang.org/index.html#download-links for Visual Studio 2015 (VS 2017, currently a Release-Candidate, already has TypeScript 2.1 built-in).

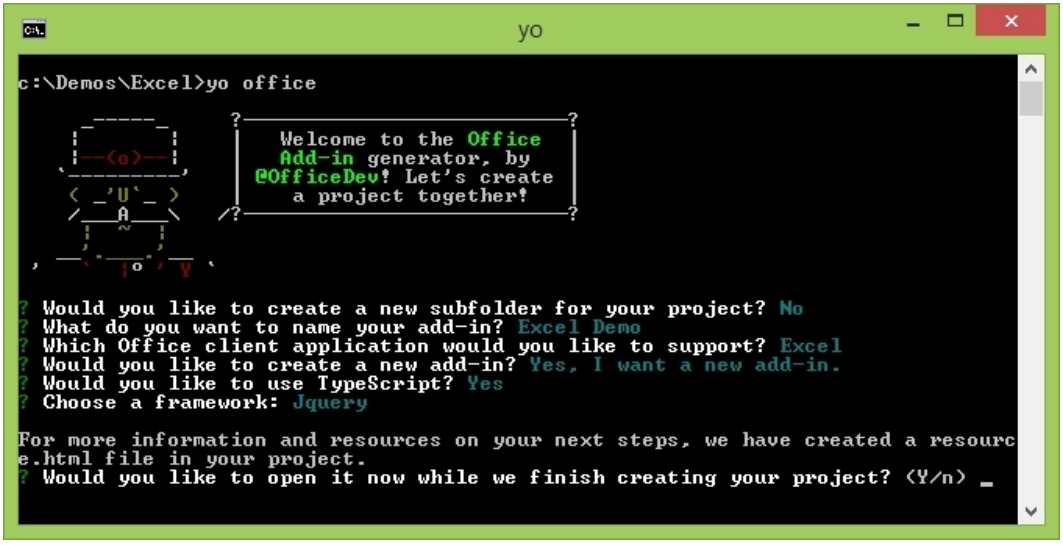
[1]Note that the file (and the rest of the files in the DefinitelyTyped repo) have recently been changed to have the library's main file be called "index.d.ts". This means that some of the older links you might encounter on the web will reference "…/office-js/*office-js.d.ts*", but in reality it should now be "…/office-js/***index.d.ts***").

3.2.2 Using Yeoman generator & Node/NPM

If you are comfortable with Node and NPM (Node Package Manager), you may find it easier to use the Yeoman template-generator for Office instead. Yeoman has been updated in early Feburary 2017 to include TypeScript templates, and to offer a bunch of other goodness (e.g., browser-sync, and the ability to auto-recompile TypeScript sources, etc). Yeoman also offers a way to use Angular 2, instead of the plain html/css/js that comes with the VS template. It requires a tad more setup, esp. the first time around (learning where to trust the SSL certificate and how to side-load the manifest), but the auto-recompilation, browser-sync, and the lightning speed of Visual Studio Code (a web-tailored cross-platform editor, not to be confused with Visual Studio proper) are worth it, if you don't mind going outside the familiar Visual Studio route.

The Add-in documentation team had put together an excellent step-by-step README for how to use the Office Yeoman generator. You can find this documentation here: https://github.com/OfficeDev/generator-office/blob/master/readme.md

Once you've installed the pre-requisites (Node, NPM), and also installed Yeoman and the Office Yeoman generator (`npm install -g yo generator-office`), you can type `yo office`, and Yeoman will guide you through a series of questions:

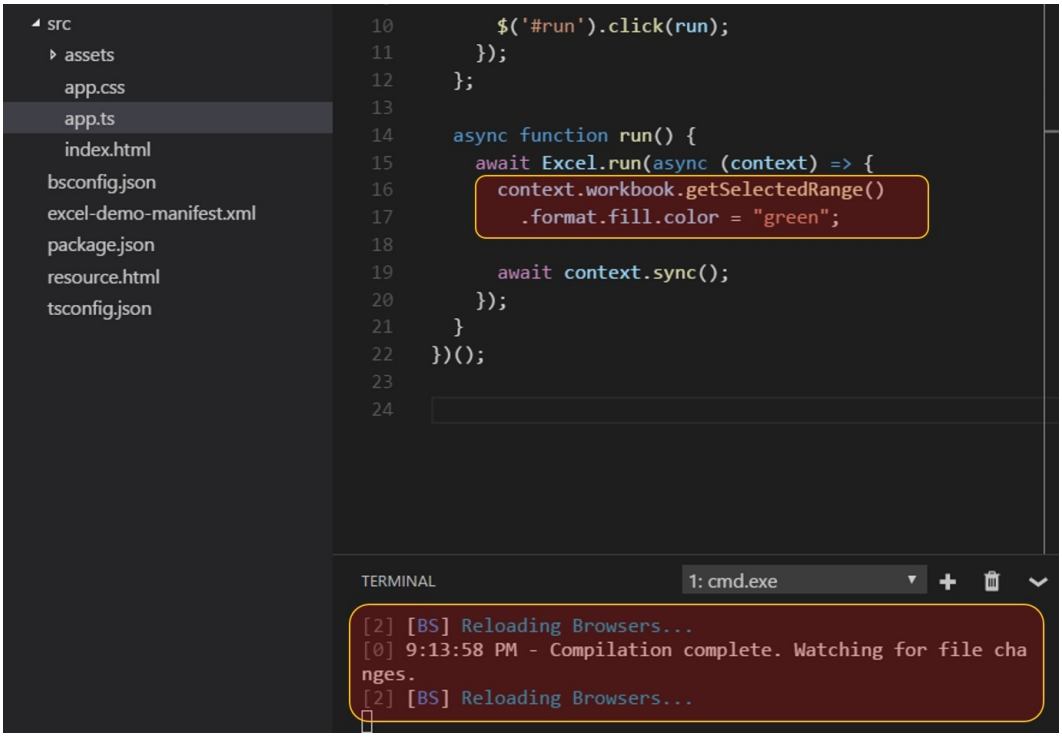


When it's done, you will have a project set up, complete with all the goodness of the NPM world and a couple of config files that wire the magic together. Run `npm start`, and the web portion of the Add-in will launch. Once you've done this, you need to do just a couple things:

1. Trust the SSL certificate. See https://github.com/OfficeDev/generator-office/blob/master/src/docs/ssl.md
2. Side-load the Add-in (via the Add-in manifest) into the Office application. The manifest will be created alongside all the rest of your project's files, in a file called `<your-project-name>-manifest.xml`
   Sideloading is very easy for Office Online, and somewhat more cumbersome on the Desktop, but just follow the instructions or step-by-step video here: https://dev.office.com/docs/add-ins/testing/create-a-network-shared-folder-catalog-for-task-pane-and-content-add-ins

When you're ready to write some code, open the folder in your favorite editor. **Visual Studio Code** is an absolutely excellent lightweight editor, which I have been using as a companion (and often, my go-to tool) for web things. You can even open a terminal straight within VS Code (use **ctrl** + ‘ [backtick])!

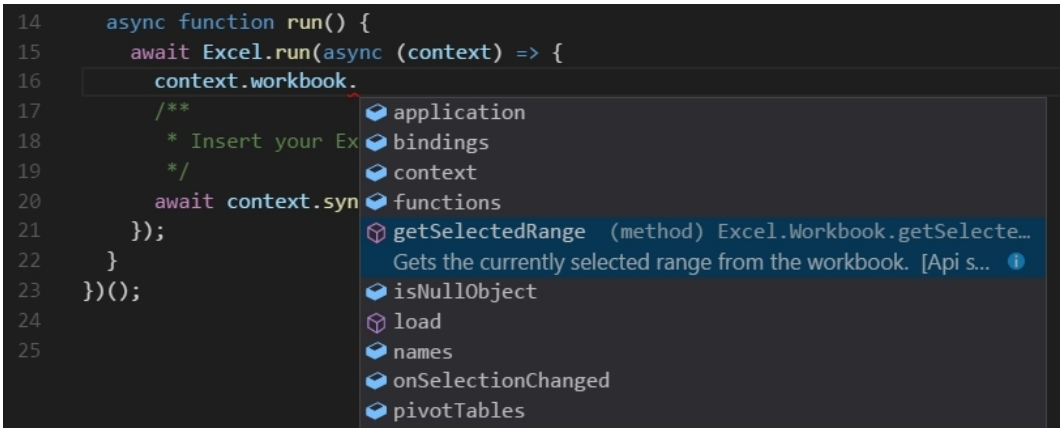
The really cool thing about using the Yeoman template and the browser-sync functionality is that as soon as you make a change to the code and save, your code gets automatically re-compiled and reloaded!



Note the auto-re-compilation of the files. The Add-in, too, will automatically re-load.

By the way, if you do not see IntelliSense when you type in something like "`Excel.`" or "`Word.`" or when you try to modify something inside of the `Excel.run` or `Word.run` block, please add a reference to the Office.js d.ts (TypeScript definitions) file. To add it, simply run

```
npm install @types/office-js
```



IntelliSense is back!

4. JavaScript & Promises primer (as pertaining to our APIs)

To those first entering JavaScript from the traditional C#/Java/C++ world, JavaScript may seem like the Wild West (and it is). While it's beyond the scope of this book to teach you JavaScript all-up, this chapter captures some important concepts in JavaScript – and particularly JavaScript's "Promises" pattern – that will greatly improve the quality and speed at which you write Office Add-ins.

If you are coming from VBA or VSTO, with little true JS experience, this chapter should be very helpful in establishing your footing. If you're a JavaScript pro, you may still find some nuggets of information useful, but you can probably skim the pages, or skip ahead entirely.

Chapter structure & planned content

Note:
- Sections that are part of this sample are **bolded and hyperlinked**
- Sections that aren't part of the sample, but *are* finished and are part of the book are in **bold**

The rest are planned topics, but have not been written (or polished) yet.

- **"JavaScript Garden", an excellent JS resource**
- **Crash-course on JavaScript & TypeScript (Office.js-tailored)**
  - **Variables**
  - **Variables & TypeScript**
  - **Strings**
  - **Assignments, comparisons, and logical operators**
  - **`if`, `for`, `while`**
  - **Arrays**
  - **Complex objects & JSON**
  - **Functions**
  - **Functions & TypeScript**
  - **Scope, closure, and avoiding polluting the global namespace**
  - **Misc.**
  - **jQuery**
- **Promises primer**
  - **Chaining Promises, the right way**
  - **Creating a new Promise**
  - **Promises, `try/catch`, and `async/await`**
  - Waiting on user action before proceeding
  - Waiting on initialization state
  - Await all vs. sequence all
- "Modern" Web programming

4.1 JavaScript & TypeScript crash-course (Office.js-tailored)

4.1.1 Variables

- Variables in plain ES5 JavaScript are declared using the **`var`** keyword: for example,

```
var taxRate = 8.49;
```

- With few exceptions, you generally call methods or access properties *on the object itself*. This should be familiar to a C#/Java audience, but might not as much to the VBA crowd, who is used to certain methods like `mid` or `len` to be global methods that accept an object, rather than being methods *on the object*.

- Variables in *plain* JavaScript do not have an *explicit* type. This means that you always declare something as `var` (or `let`). You do **not** specify the type during declaration (i.e., there is no "`dim taxRate as Integer`" or "`int taxRate`", as you'd see in VBA or C#, respectively).

- The primitive types are just:

  - Booleans: "`var keepGoing = true`" (or `false`).
  - Numbers. There is no distinction between integers and decimals ("`var count = 5`" versus "`var price = 7.99`"), they're all numbers.
  - Strings, discussed separately in a different section.

4.1.2 Variables & TypeScript

- TypeScript offers an additional keyword, "**`let`**", which can be used in place of `var` and has cleaner semantics in terms of scoping rules (i.e., variables declared inside a loop using `let` will only be visible/usable in that loop, just as you'd expect with VBA or C#, but *not* what you get with just the JavaScript `var`). Because of this, when using TypeScript, it is **always a best-practice to use "`let`"**. In fact, if you're only assigning a variable once and never re-assinging it, it is better to use an event stronger form of `let`: "**`const`**".

- TypeScript, true to its name, let you ***optionally*** (and gradually) add type information to variables, which you do by appending a colon and a type name after the declaration (you'll see examples momentarily). Common types include:

  - Primitive types, such as `number`, `string`, `boolean`.
  - Array of types. This can be written as "`number[]`" or "`Array<number>`", it's a stylistic preference.
  - Complex types (esp. ones that come with the Office.js object model). For example, "`Excel.Range`" or "`Word.ParagraphCollection`".

- Specifying types is generally used for:

  - Variable declarations, especially if it's necessary to declare a variable in advance, before assignment:

    ```
    let taxRate: number;
    ```

  - Parameter type declarations. This is supremely useful, as it allows you to split long procedures into multiple subroutines, and yet not lose IntelliSense and compile-time error-checking. For example, you can declare a function as

    ```
    function updateValues(range: Excel.Range) { ... }
    ```

    and be able to see all of the `range` properties/methods/etc. within the body of the function, as opposed to seeing a dishearteningly-empty IntelliSense list.

  - Function return-type declarations, such as

```
function retrieveTaxRate(): number { ... }
```

That way, when assigning the value to a variable, the variable will get implicitly typed. Thus,

```
var taxRate = retrieveTaxRate();
```

will now give you the full IntelliSense for a number, rather than an empty list for an unknown ("`any`") type.

- Very occasionally, you might have disagreement with the TypeScript compiler over a particular type. This happens primarily in Excel for 2D-array properties, such as `values` or `formulas` on an `Excel.Range` vobject. For those properties, the *return value* is always a 2D array, but they are more accepting in terms of their *inputs* (namely, they can accept a single non-arrayed value). TypeScript, however, does not recognize such subtleties.

  Fortunately, TypeScript offers a simple escape hatch from its type system: just stick an "**`<any>`**" in front of the value you're setting, and TypeScript will let the issue go. So, if you want to set a single value to a property that's technically a 2D-array, just do something like

  ```
  range.values = <any> 5;
  ```

  and both you and the TypeScript compiler will walk away happy.

5. Office.js APIs: Core concepts

Chapter structure & planned content

Note:
- Sections that are part of this sample are **bolded and hyperlinked**.
- The rest are finished, and are part of the book, but not the sample.

- **Canonical code sample: reading data and performing actions on the document**
- **Excel.run (Word.run, etc.)**
- **Proxy objects: the building-blocks of the Office 2016 API model**
  - **Setting document properties using proxy objects**
  - **The processing on the *JavaScript side*](#proxy-js-processing)**
  - **The processing on the *host application's side*](#proxy-host-processing)**
  - **Loading properties: the bare basics**
  - **De-mystifying `context.sync()`**
- **Handling errors(#handling-errors)**
- **Implementation details, for those who want to know how it *really* works**
- **More core `load` concepts**
  - **Scalar vs. navigation properties – and their impact on `load`**
  - **Loading and re-loading**
  - **Beware misspellings**
  - **Loading collections**
  - **Loading on methods versus properties**
  - **Values that load automatically (`ClientResult`-s)**
- **More core `sync` concepts**
  - **Real-world example of multiple `sync`-functions**
  - **When to sync**
  - **The final `context.sync` (in a multi-`sync` scenario)**

- **A more complex `context.sync` example**
- **Avoiding leaving the document in a “dirty” state**

5.1 Canonical code sample: reading data and performing actions on the document

To set the context for the sort of code you'll be writing, let's take a very simple but canonical example of an automation task. This particular example will use Excel, but the exact same concepts apply to any of the other applications (Word, OneNote) that have adopted the new host-specific/Office 2016+ API model.

Scenario: Imagine I have some data on the population of the top cities in the United States, taken from "Top 50 Cities in the U.S. by Population and Rank" at http://www.infoplease.com/ipa/a0763098.html. The data – headers and all, just like I found it on the website – describes the population over the last 20+ years.

Let's say the data is imported into Excel, into a table called "PopulationData". The table could just as easily have been a named range, or even just a selection – but having it be a table makes it possible to address columns by name rather than index. Tables are also very handy for end-users, as they can filter and sort them very easily. Here is a screenshot of a portion of the table:

	A	B	C	D	E	F
1	Size rank 2014	City	7/1/2014 population estimate	7/1/2013 population estimate	4/1/2010 census population	7/1/2005 population estimate
2	1	New York, N.Y.	8,491,079	8,405,837	8,175,133	8,143,197
3	2	Los Angeles, Calif.	3,928,864	3,884,307	3,792,621	3,844,829
4	3	Chicago, Ill.	2,722,389	2,718,782	2,695,598	2,842,518
5	4	Houston, Tex.	2,239,558	2,195,914	2,100,263	2,016,582
6	5	Philadelphia, Pa.	1,560,297	1,553,165	1,526,006	1,463,281
7	6	Phoenix, Ariz.	1,537,058	1,513,367	1,445,632	1,461,575
8	7	San Antonio, Tex.	1,436,697	1,409,019	1,327,407	1,256,509
9	8	San Diego, Calif.	1,381,069	1,355,896	1,307,402	1,255,540
10	9	Dallas, Tex.	1,281,047	1,257,676	1,197,816	1,213,825
11	10	San Jose, Calif.	1,015,785	998,537	945,942	912,332
12	11	Austin, Tex.	912,791	885,400	790,390	690,252
13	12	Jacksonville, Fla.	853,382	842,583	821,784	782,623
14	13	San Francisco , Calif.	852,469	837,442	805,235	739,426
15	14	Indianapolis, Ind.	848,788	843,393	820,445	784,118
16	15	Columbus, Ohio	835,957	822,553	787,033	730,657
17	16	Fort Worth , Tex.	812,238	792,727	741,206	624,067

The population data, imported into an Excel table

Now, suppose my task is to find the top 10 cities that have experienced the most growth (in absolute numbers) since 1990. How would I do that?

The code in the next few pages shows how to perform this classic automation scenario. As you look through the code, if not everything will be immediately obvious – and it probably won't be – don't worry: the details of this code is what the rest of this chapter (and, to some extent, the book) is all about! But I think it's still worth reading through the sample as is for the first time, to gain a general sense of how such task would be done via Office.js.

Note: In a more real-world scenario, this sample would be broken down into ~4 functions: one to read the data, another to calculate the top 10 changed

cities, a third to write out a table, and a forth to bind a chart to the data. For purposes of this sample, though – and in order to make it easily readable from top to bottom, rather than having the reader jump back and forth – I will do it in one long function. In a later section, "*A more complex* `context.sync` *example*", I will show an example of code where I do split out the tasks into smaller subroutines.

TIP: For those coming from VBA, one thing you'll immediately see – and what you'll need to adjust to – is that ***everything is zero-indexed***. For example, `worksheet.getCell(0, 0)`, which would be absolutely incorrect and throw an error in VBA, is the correct way to retrieve the first cell using the Office.js APIs.

Of course, for user-displayable things like the address of the cell, it would still report "`Sheet1!A1`", since that's what the user would have seen. But in terms of programmatic access, *everything* is ***zero-indexed***[1].

[1]There is hardly a rule without an exception. In Excel in particular, the API exposes the native Excel function to JavaScript under the `workbook.functions` namespace (i.e., `workbook.functions.vlookup(...)`. Within such functions, it made sense to keep the indexing consistent with the sort of native Excel formulas that a user would type (otherwise, what's the point?!) – and so there, any index is **1-indexed**. For example, if the third parameter to a "vlookup" call, "colIndexNum", equals to 3, this refers to the third column, *not* the 4th one. But in terms of the object model, everywhere else (and in JavaScript itself, of course!), everything is zero-indexed.

A TypeScript-based, canonical data-retrieval-and-reporting automation task

```
Excel.run(async (context) => {
    // Create proxy objects to represent entities that are
    // in the actual workbook. More information on proxy objects
    // will be presented in the very next section of this chapter.

    let originalTable = context.workbook.tables
        .getItem("PopulationTable");

    let nameColumn = originalTable.columns.getItem("City");
    let latestDataColumn = originalTable.columns.getItem(
        "7/1/2014 population estimate");
    let earliestDataColumn = originalTable.columns.getItem(
        "4/1/1990 census population");

    // Now, queue up a command to load the values for each of
    // the columns we want to read from.  Note that the actual
    // fetching and returning of the values is deferred
    // until a "context.sync()".

    nameColumn.load("values");
    latestDataColumn.load("values");
    earliestDataColumn.load("values");

    await context.sync();


    // Create an in-memory data representation, using an
    // array with JSON objects representing each city.
    let citiesData: Array<{name: string, growth: number}> = [];

    // Start at i = 1 (that is, 2nd row of the table --
    // remember the 0-indexing) in order to skip the header.
    for (let i = 1; i < nameColumn.values.length; i++) {
        let name = nameColumn.values[i][0];

        // Note that because "values" is a 2D array
```

```
37          // (even if, in this case, it's just a single
38          //  column), extract the 0th element of each row.
39          let pop1990 = earliestDataColumn.values[i][0];
40          let popLatest = latestDataColumn.values[i][0];
41
42          // A couple of the cities don't have data for 1990,
43          // so skip over those.
44          if (isNaN(pop1990) || isNaN(popLatest)) {
45              console.log('Skipping "' + name + '"');
46          }
47
48          let growth = popLatest - pop1990;
49          citiesData.push({name: name, growth: growth});
50      }
51
52      let sorted = citiesData.sort((city1, city2) => {
53          return city2.growth - city1.growth;
54          // Note the opposite order from the usual
55          // "first minus second" -- because want to sort in
56          // descending order rather than ascending.
57      });
58      let top10 = sorted.slice(0, 10);
59
60      // Now that we've computed the data, create a new worksheet
61      // for the output, and write in the data:
62      let sheetTitle = "Top 10 Growing Cities";
63      let sheetHeaderTitle = "Population Growth 1990 - 2014";
64      let tableCategories = ["Rank", "City", "Population Growth"];
65      let outputSheet = context.workbook.worksheets.add(sheetTitle);
66
67      let reportStartCell = outputSheet.getRange("B2");
68      reportStartCell.values = [[sheetHeaderTitle]];
69      reportStartCell.format.font.bold = true;
70      reportStartCell.format.font.size = 14;
71      reportStartCell.getResizedRange
72          (0, tableCategories.length - 1).merge();
73
74      let tableHeader = reportStartCell.getOffsetRange(2, 0)
```

```
75          .getResizedRange(0, tableCategories.length - 1);
76      tableHeader.values = [ tableCategories ];
77      let table = outputSheet.tables.add(
78          tableHeader, true /*hasHeaders*/);
79
80      for (let i = 0; i < top10.length; i++) {
81          let cityData = top10[i];
82          table.rows.add(
83              null /* null means "add to end" */,
84              [[i + 1, cityData.name, cityData.growth]]);
85
86          // Note: even though adding just a single row,
87          // the API still expects a 2D array (for
88          // consistency and with Range.values)
89      }
90
91      // Auto-fit the column widths, and set uniform
92      // thousands-separator number-formatting on the
93      // "Population" column of the table.
94      table.getRange().getEntireColumn().format.autofitColumns();
95      table.getDataBodyRange().getLastColumn()
96          .numberFormat = [["#,##"]];
97
98
99      // Finally, with the table in place, add a chart:
100     let fullTableRange = table.getRange();
101
102     // For the chart, no need to show the "Rank", so only use the
103     //     column with the city's name, and expand it one column
104     //     to the right to include the population data as well.
105     let dataRangeForChart = fullTableRange
106         .getColumn(1).getResizedRange(0, 1);
107
108     let chart = outputSheet.charts.add(
109         Excel.ChartType.columnClustered,
110         dataRangeForChart,
111         Excel.ChartSeriesBy.columns);
112
```

```
113     chart.title.text = "Population Growth between 1990 and 2014";
114
115     // Position the chart to start below the table, occupy
116     // the full table width, and be 15 rows tall
117     let chartPositionStart = fullTableRange
118         .getLastRow().getOffsetRange(2, 0);
119     chart.setPosition(chartPositionStart,
120         chartPositionStart.getOffsetRange(14, 0));
121
122     outputSheet.activate();
123
124     await context.sync();
125
126 }).catch((error) => {
127     console.log(error);
128     // Log additional information, if applicable:
129     if (error instanceof OfficeExtension.Error) {
130         console.log(error.debugInfo);
131     }
132 });
```

Download sample code

To view or download this code, follow this link:
http://www.buildingofficeaddins.com/samples/population-analysis

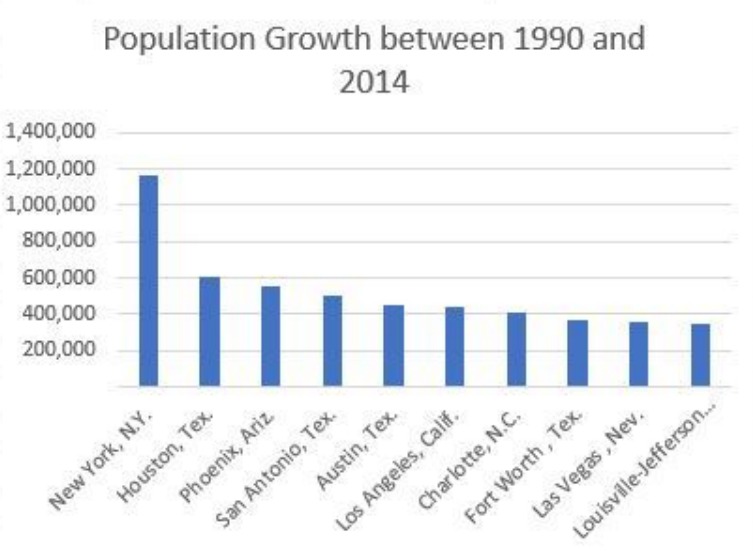
For those coming from VBA or VSTO, by far and away the biggest difference you'll notice is the need to explicitly call out which properties you want loaded (`nameColumn.load("values")`), and the two `await context.sync()`-s towards the beginning and very end of the operations. But for the rest of the logic, you have simple sequential code, not unlike VBA. This is the beauty

of the new 2016+ APIs – that they provide you with local proxy objects that "stand in" for document objects, and allow you to write mostly-synchronous code (interspersed with the occasional "load" and "sync"). You will read more about loading and syncing in the forthcoming sections.

If you run the above code, here is what the resulting sheet looks like:

Population Growth 1990 - 2014

Rank	City	Population Growth
1	New York, N.Y.	1,168,515
2	Houston, Tex.	609,005
3	Phoenix, Ariz.	553,655
4	San Antonio, Tex.	500,764
5	Austin, Tex.	447,169
6	Los Angeles, Calif.	443,466
7	Charlotte, N.C.	414,024
8	Fort Worth , Tex.	364,619
9	Las Vegas , Nev.	355,304
10	Louisville-Jefferson County, Ky.2	343,717



JavaScript-only

For folks using JavaScript instead of TypeScript, you will find it useful to reference the ***JavaScript-only version of the canonical code sample***.

Now, let's dive in and see how this sample works.

5.2 Proxy objects: the building-blocks of the Office 2016 API model

5.2.1 Setting document properties using proxy objects

At the fundamental level, the Office 2016+ API model consists of local JavaScript *proxy* objects, which are the local placeholders for real objects in the document. You get a proxy object by calling some property or method (or a chain of properties or methods) that originate somewhere off of the *request context*. For example, if you have an Excel object called `workbook`, you can get cells `A1:B2` on `Sheet1` by calling

```
var myRange = workbook.worksheets.getItem("Sheet1")
    .getRange("A1:B2");
```

To apply values to the object, you would simply set the corresponding properties. For example:

```
myRange.values = [["A", "B"], [1, 2]];
myRange.format.fill.color = "yellow";
```

The same applies to calling a method – you simply invoke it. For example,

```
myRange.clear();
```

Importantly, what happens beneath the covers is that the object – or, more accurately, the *request context* from which the object derives – accumulates the changes locally, much like a changelist in a version-control system. **Nothing is "committed" back into the document until you recite the magical incantation:**

```
context.sync();
```

5.2.2 Loading properties: the bare basics

The preceding section only talked about setting properties or calling methods on document objects. What about when you need to ***read data back*** from the document?

For reading back document data, there is a special command on each object, `object.load(properties)`. An identically-functioning method can also be found on the context object: `context.load(object, properties)`. The two are 100% equivalent (in fact, `object.load` calls `context.load` internally), so which one you use is purely stylistic preference. I generally prefer `object.load`, just because it feels like I'm dealing with the object directly, rather than going through some context middleman/overlord. But again, it's purely stylistic.

The load command is queued up just like any of the others, but at the completion of a `context.sync()` – which will be covered in greater detail in a later section – an internal post-processing step will automatically populate the object's properties with the requested data. The property names that you pass in into the `object.load(properties)` call are the very properties that – after a `sync` – you will be able to access directly off of the object.

For example, if you look at the IntelliSense for the Excel `Range` object, you will see that it contains properties like `range.values` and `range.numberFormat`. If you need these values and number formats in order to execute your function (maybe to display them to the user, or to calculate the minimum/maximum, or to copy them to a different section of the workbook, or to use them to make a web service call, etc.), you would simply list them as a comma-separated list in the `.load` function, as show in the example below:

Loading and using object properties

```
1  Excel.run(async (context) => {
2      let myRange = context.workbook.worksheets
3          .getItem("Sheet1").getRange("A1:B2");
4
5      myRange.load("values, numberFormat");
6
7      await context.sync();
8
9      console.log(JSON.stringify(myRange.values));
10     console.log(JSON.stringify(myRange.numberFormat));
```

```
11
12 }).catch(...);
```

TIP: In the above code, you see the use of `JSON.stringify`. For those relatively new to JavaScript: `JSON.stringify`, used above, is a very convenient way of viewing an object that might otherwise show up as `[object Object]` (typically a complex type or a class). The `JSON.stringify` also helps for types that might otherwise show up incorrectly, like a nested array (for example, the array `[[1,2],[3,4]]` shows up as the string `1,2,3,4` if you call `.toString()` on it, instead of its true representation of `[[1,2],[3,4]]`).

Importantly, just as in the code above, the reason to call `object.load` is that you ***intend to read back the values*** in the code following the `context.sync()`, and that your operation can't continue without having that information. This brings us to arguably the most important rule in all of Office.js:

THE GOLDEN RULE OF `OBJECT.LOAD(PROPERTIES)`: You ***ONLY*** need to load the object ***IF you are intending to read back its PRIMITIVE PROPERTIES (numbers, strings, booleans, etc.)***.

Just calling methods on the object, or setting its properties, or using it to navigate to another object, or doing ***anything else that doesn't involve reading and using the object's property values*** does ***NOT*** require a `load`.

By the way, the **return value of `object.load(...)`** is the object itself. This is done purely for convenience, saving a line or two of vertical space. That is,

```
let range = worksheet.getRange("A1").load("values");
```

Is 100% equivalent to the more verbose version:

```
let range = worksheet.getRange("A1");
range.load("values");
```

6. Appendix A: Using plain ES5 JavaScript (no `async/await`)

Chapter structure & planned content

Note:
- Sections in this chapter have all been written and are part of the book, but are not part of this sample.

- **Passing in functions to Promise `.then`-functions**
- **JavaScript-only version of the canonical code sample**
- **JavaScript-specific `sync` concepts**
  - **A JavaScript-based multi-sync template**
  - **Returning the `context.sync()` promise**
  - **Passing values across `.then`-functions**
  - **JavaScript example of multi-`sync` calls**

7. … More chapters to-be-ported into this book soon! …

There is a great many chapters – with some finished sections, and many more unfinished ones – that I need to move over to LeanPub. I had originally started writing in a different format, and so the migration to LeanPub takes a bit of time and manual copying. Stay tuned, as more content should be appearing here very soon…

Thanks for being part of an *in-progress / evergeen / lean-published* book!

∼ Michael Zlatkovsky The author