overview of building an addin

The addin chapter can be a real difference maker. Making an addin is a great way to bring together a large number of related features that would be a pain to keep together anyways.

Some topics to hit:

- The advantages of always having an addin available to save new code into
- How to actually create an addin (the actual steps of saving it)
- Working with the Ribbon
- Adding keyboard shortcuts to the addin (via events)
- Managing the source code for an addin?

In addition to the VBA addin, it would be good to discuss the other options for extend Excel.

Include a section about Excel DNA and creating more powerful addins over there?

introduction to creating an addin

This chapter will focus on creating an addin for Excel using VBA. There are other ways to create an addin but using VBA is simple because it can be done entirely from Excel and the Visual Basic Editor. The main distinction between an addin and other VBA code is that an addin is meant to be available to all open

Workbooks without having to put the code inside a Workbook. This can be a very nice thing to have if you are regularly do the same or similar operations across different Workbooks. The alternative to an addin is often to maintain a library of code that you regularly export/import into macro enabled files as needed. This can create a mess as you change code in one file but not in another. The alternative also typically requires you to put the code inside a the Workbook and make it macro enabled. For certain applications, this is a non-starter. The one other alternative to a true addin is to create a Workbook that contains the code you want, and then you can open that file and execute the code in the context of whatever other files are open. This works, and creating an addin can be viewed as the logical conclusion of this approach. More than the logical conclusion, this is actually the first step for creating an addin.

When considering whether or not to create a proper addin with your code, consider the following:

- An addin provides a nice package for helper code and UDFs that might be used in multiple places
- An addin has easy access to the Ribbon and can create its own Ribbon tab
- An addin can be put in a central location and used as a repository of code for an organization (works best if the file is read-only)

Item 1 in the list above is typically enough of a reason to consider creating an addin. A common example of an addin is as a personal repository of VBA code. This typically replaces the use of the Personal Workbook, which I have never found to work well.

When considering a personal addin, one of the biggest upsides is that you can always open the VBE and have immediate access to your library of code. This makes it easy to make edits and save the new addin. Immediately, your updated code is available for future use in all your Workbooks.

There are a couple of downsides related to addins:

- UDFs from an addin require that anyone opening the spreadsheet has the addin loaded
- For code in a single Workbook, it is often easier to simply use a macro enabled Workbook and save the code directly there
- Some folks are highly resistant to “installing an addin” but will happily open a XLSM file. These are equivalent in the case of opening an addin, but the hesitation still exists.

Point 2 above is worth expanding on. Sometimes it’s tempting to add code to an existing addin that make sense only in the context of a single file. This works well if you and everyone else have the addin. This starts to become a nuisance when you are constantly going through your addin to find code that should have been place in a Workbook to start. The cleaner way to store code that may be useful later is to place a copy of it in a personal addin. This ensures that the original code is always available in the Workbook and that future updates to the code don’t break the original application.

creating an addin

Creating an addin is a relatively simple process. You start with a normal XLSM macro enabled file. From there, you save it as the add-in type (XLAM). That's it.

If you want to get more complicated, there is a property in the VBE that can be toggled to change the addin status. (TODO: add picture of that). You would only need to change that flag if for some reason you wanted to save something back to a normal XLSM workbook without changing the extension.

There is one additional process that can be done to change how the addin is created is that is if you are modifying the Ribbon for your addin. To do that, you will need to manually edit the XLAM file and change a file within it to add Ribbon support. You can do this manually or you can use a tool to help you out. Check the later section for details on that process.

specific aspects to addin development

Depending on the addin that you are creating, you may expect for it to have a handful of features available. In general, those types of features include keyboard shortcuts, special forms or user prompts, and possibly automatic features that fire depending on the user's action or the state of the workbook or Application.

Keyboard Shortcuts

The simplest thing to do is to add keyboard shortcuts to your addin. There are two ways to do that:

- Open up the Macros form on the Developer tab. You can then hit “options” for a given Sub and assign a keyboard shortcut (TODO: add picture of this)
- That approach can sometimes be a pain to edit later, so you can also add code to your addin to add the shortcut.

The latter approach is nice because you can easily change the shortcut or the calling method. For addins, I will nearly always take the latter approach since it is much easier to deal with alter. For XLSM workbooks, I will do the former since it is easier to change from a workbook.

If you want to add the keyboard shortcut using code, use the code below. Ideally, you would put this in a Workbook_Open event that is called when the workbook opens. You can also use this approach to add/remove shortcuts depending on user input.

```
Public Sub SetUpKeyboardHooksForSelection()


    'SHIFT =    +
    'CTRL =     ^
    'ALT =      %

    'set up the keys for the selection mover
    Application.OnKey "^%{RIGHT}", "SelectionOffsetRight"
    Application.OnKey "^%{LEFT}", "SelectionOffsetLeft"
    Application.OnKey "^%{UP}", "SelectionOffsetUp"
    Application.OnKey "^%{DOWN}", "SelectionOffsetDown"

    'set up the keys for the indent level
    Application.OnKey "+^%{RIGHT}", "Formatting_IncreaseIndentLevel"
    Application.OnKey "+^%{LEFT}", "Formatting_DecreaseIndentLevel"

End Sub
```

USer Forms

One of the nice features of an addin are adding custom forms to provide the user with a better experience.

Creating a UserForm in VBA is dead simple, and this is the best bang for your buck in terms of creating a professional looking product. The simplest of forms with the simplest of features can save the end user hours and hours of time (I've seen it happen).

The nice thing here is that creating a UserForm in an addin is not any different than creating them normally. You simply create the UserForm. The only extra step is that you need to manage how/when the form is created and what information it has access to. Typically this is done by adding a button or using a keyboard shortcut. The only other issue is that you need to be aware of which Workbook or Worksheet is active when opening a UserForm if you are using ActiveSheet or ActiveWorkbook for anything. In general, inside an addin, you need to be careful with this commands since it is not always obvious that the ActiveXXX is the one you want to access.

Helpful Commands

There are a couple of commands that exist outside of addins that become far more useful inside the addin. They are included below for reference:

- `ThisWorkbook` refers to the workbook that contains the code being executed. This is the surefire way to refer to the XLAM file that is running instead of the ActiveWorkbook. IN general, your addin will never be the ActiveWorkbook. This becomes relevant if your addin workbook contains sheets of data that may need to be accessed during runtime. You would use THisWorkbook to refer to those sheet.

- TODO: add any other commands that are addin specific

Other functionality

THe other functionality that you can add is related to Events. You have great power when it comes to listening to events and triggering various actions. THe real difficulty is deciding what is an appropriate use of that power. Namely, when will you create an experience that benefits the user versus creating a very confusing workbook that is prone to breaking?

Before diving into what events can do, it's worth nting that potential downfalls of using them:

- They can be quite finicky sometimes. That is, using events adds a layer of complexity that tends to just complicate Excel and VBA. I don't have a technical explanation, but there seem to be a number of bugs that creep out of the dark once you start really using events.
- Your user can disable events at will and it can be quite difficult to determine when that was done. This is done with `Application.EnableEvents = False`.
- Events are triggered all the time for all sorts of reasons. If you are doing a lot of checking in Events, you will dramatically slow down the workbook.

With all of those warnings, there is nothing wrong with using Events. They generally do what you want and can be quite powerful. I add the caveats only because I have seen them ruin an otherwise working workbook. That complexity gets amped up a level when your Event code is inside an addin instead of the main workbook.

To really make the most of Events, you are going to need to use Class Modules. The reason is that your Events need to "latch on" to the host workbooks or worksheets, and the only way to do that is by using Class Modules. Normally, outside of an addin, you can simply open up the relevant VBA object (Workbook or Worksheet) and add the event code there. For an addin, you cannot add that code outside of the addin so you are in a bind. How then can you hook onto the Event? Fortunately, VBA makes this possible with the `With Events` command inside of a Class Module.

TODO: provide a concrete example of using this code

UI features for addins, Ribbon, toolbars, UserForms

There are a number of UIs that can be provided for an addin. The most common involve using the Ribbon or providing UserForms (typically accessible from a keyboard shortcut). Those two approaches will discussed in detail. It is also worth mentioning that if you are going to support Excel 2007 and before, that the Ribbon did not exist back then. For those prior versions of Excel, the interfaces were built using toolbars and menu items. That’s before my day, so I’ll just say that if you use that code today, it will show up in the Ribbon. If you need to support those versions of Excel, you would do well to find a different book.

the Ribbon

When using the Ribbon, there are a couple of items to consider:

- Do you want your UI to show up in an existing tab or on your own?
- How interactive do you want your UI to be? This can range from simple buttons that trigger actions to text boxes and other more interactive features that are able to detect user input and respond accordingly.
- How do you prefer to edit the file? How fast do you want your developments cycle to be with respect to the UI?

For the first point, this is a simple preferences. For a given addin it may make sense to simply put the buttons and other access on an existing tab (Developer and Data are popular!) and provide that level of access. For an addin that has a dedicated purposed independent of other Excel features, it starts to make sense to add your own tab exclusively for your addin. This is good for helping your users find your features. It can also be more consistent in terms of keyboard shortcuts. If you are going to modify an existing tab, be abolustley certain that you verify that the keyboard shortcuts work as expected. There is nothing worse than having an addin break the ALT+A+R+Y shortcut which is supposed to reapply an autofilter. It is not fun when that shortcut becomes ALT+A+R+Y2. Seriously?

For the second point, you will need to consider the average user and their expectations. Keep in mind that the default Excel Ribbon includes a number of locations where user input is collected nad used beyond a simple button. This includes things like some number input (font size, page layout) and other drop downs. There is a willingness for Excel users to ues these features where it make sense. For what it's worth, from the Excel VBA point of view, it is much simpler to not try nad collect user input. This can be done (TODO: add examples), but the effort here is typically not worth the user experience. If you choose to go this route, I would highly recommend using drop downs and other inputs that provide some automatic filtering of user input. Trying to validate user input off the Ribbon is a pain and does not provide a good experience. Having said that, if you are designing for power users, you can build a very slick interface in the Ribbon that is unmatched.

The final point gets down to the nitty gritty of actually editing the Ribbon. The problem is that the Ribbon is defined in a file inside your XLAM file and is not editable from any part of the VBE or Exce interface. This means that it is a real pain to edit the Ribbon definition in the same way that you can edit the other VBA code. I have typically taken the approach of using a button on the Ribbon to launch a form that exists in VBA. That form can then be edited without having to touch the Ribbon definiont. This sounds tribial, but it can make a huge difference if you are designing an addin that has a lot of possible interactivity; it is very difficult to edit the Ribbon in real time. Having said that, there is one addin that makes this process much more manageable. It is from Andy Pope (TODO: add link) and works great for building out the interface. Even using that addin, it is a pian to add the callback necessary to tie the Ribbon to VBA. Don't be dissuaded from creating a nice Ribbon UI, but realize that it takes time and effort and attention to detail to properly detail out the Ribbon UI aspects.

editing the Ribbon Note that the Ribbon is defined in an XML file inside your XLAM file. Remember that an XLAM file is simply a ZIP file of a bunch of different folders. By default, the Ribbon definition is not included and you must add the folder and file. To do this, simple create the `customUi` folder and then create a XXX XML file inside there. This file will define the specific changes you are making to the Ribbon.

callbacks Once you hav the Ribbon XML set up, you will be defining callbacks that need to actually exist in the VBA code. I always like to create a `Ribbon` module in the XLAM file which is solely responsible for callbacks. This is nice for larger addins with a large number of callbacks because it provides a single place. It also avoids debugging errors later when you accidentally put some critical code into a callback and forgot to check that out.

The callbacks take an odd signature. I always use Andy Pope's addin or copy a previous one. I have very seldom used the parameters in the callback for accessing the Ribbon information. My approach has always been to avoid extra interactivity with the ribbon. I have done it before, and it works, but the problem is that it is just not intuitive to do it that way. It is much easier to add a keyboard shortcut which shows a USerForm than to attempt to get the user to focus on the Ribbon (using the keyboard or mouse) and then provide the best info.

UserForms

If you are going to use UI features within your addin, you are going to use UserForms. They provide the cleanest and easiest interface for the vast majority of automation and other tasks. THe one exception to using a UserForm is when you can get by with a simple `InputBox`. You should alwa sprefer the INputBox because the procedure for calling them and obtaining a value is dead simple. Also, the InputBox is the best way to ask the user for a Range input via selection. You can technically use the `RefEdit` control, but hta control is very sensitive when it works.

If you are building a USerFOrm, there is very little than tis different from a normal USerForm. The only thing to be aware of are the logistics of creating, showing, and hiding the USeForm. I have previously tried to keep an instance of a given form live in order to use the previous values. This has worked well inside a single Workbook btu seems to be very finicky when working across multiple Workbooks. The procedure here is very simple:

```
DIm frm as UserForm
Set frm = New UserForm

fmr.Show
```

The code above is all that is required to create a new instance of a form and show it to the user. From there, the code is the same as before: you simply create the form and call the various Subs you want. One thing which is helpful is to hide the form when you are done. This is done with the `Unload Me` command.

One other item to be aware of is that the default UserForm is set to `ShowModal = True` which applies the modal property. A "modal" dialog is one who steals focus from any other elements and must be dealt with before you can go back to your previously focusbale elements. This is often good for certain workflows where you do not want the user to change the underlying spreadsheet while you collect their input. There are other instances however where it makes sense to allow the user to change the active Workbook, Worksheet, or Selection and then interact with you rform. To allow for this behavior, set `ShowModal = False`. THs will allow your form to exit even when the user clicks off and interacts with the spreadsheet again. This is a real game changer when you are working with code that operates on the current selection. You are then able to leave your form up while the user changes the selection. From there, they are able to call the cod ethey want on the objects they want. I have used this technique to great effect when working with Charts: allow the user to select their charts and then hit a button.