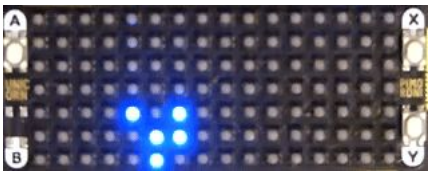
Intro To MAUI For Makers

Dennis Dunn <ansofive@gmail.com>



Hello everybody! It's really good to be back at Stir Trek. I've missed you guys!

In this session we're going to take a look at a gadget and a couple of different ways to control it.

So, what types of control might a gadget need?

- Initiate state changes: Inactive -> Active
- Configuration changes: Provisioning WIFI
- Real-time control: Drive a robot around the house.
- Real-time parameter changes: Increase the setpoint on a microbrewery.

First we'll look at the gadget side of things and then we'll dive into the Android app that controls it. The mobile app is built using Microsoft's **.NET MAUI** framework. The **Multi-Platform Application UI** framework helps you build apps for multiple platforms from a single codebase. That means one solution targeting Android, IOS, Windows, and Tizen. Of course there are specific tasks that must be done for each of the platforms but they are all managed from a single source.

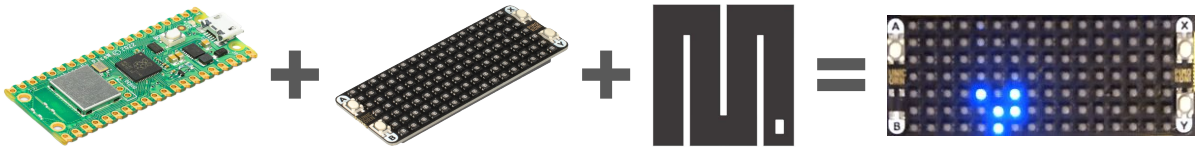
I'll be using *Microsoft Visual Studio 2022 Community Edition* and *VS Code*. If you want to see MAUI development with the Apple ecosystem check out **Sam Basu's 'Open Source XAML Takes You Places!'** at **1:00 PM** in **Sith**.

By the end of the session I hope that you have ideas for new ways to control your own gadgets and the confidence to try out mobile development with .NET MAUI.

gadget /găjʹĭt/

noun

1. A small specialized mechanical or electronic device; a contrivance.
2. Any device or machine, especially one whose name cannot be recalled. Often either clever or complicated.



The gadget we'll be controlling is made from a Raspberry Pi Pico W, a Pimoroni Pico Unicorn Pack, and the MicroPython runtime.

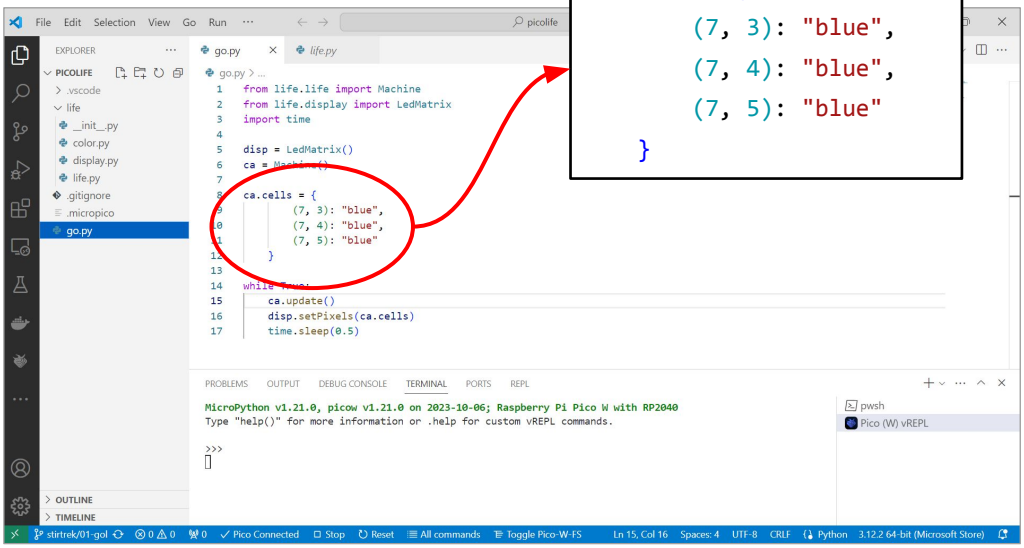
The Pi Pico is a dual-core microcontroller with 2 MB of flash memory, 264 KB of SRAM, WIFI and Bluetooth.

The Unicorn display is a 7x16 matrix of RGB LEDs.

The firmware is the MicroPython runtime provided by Pimoroni which contains support for their product line.

Altogether they make up the *Conway's Game Of Life Gadget*.

Configuring the Seed

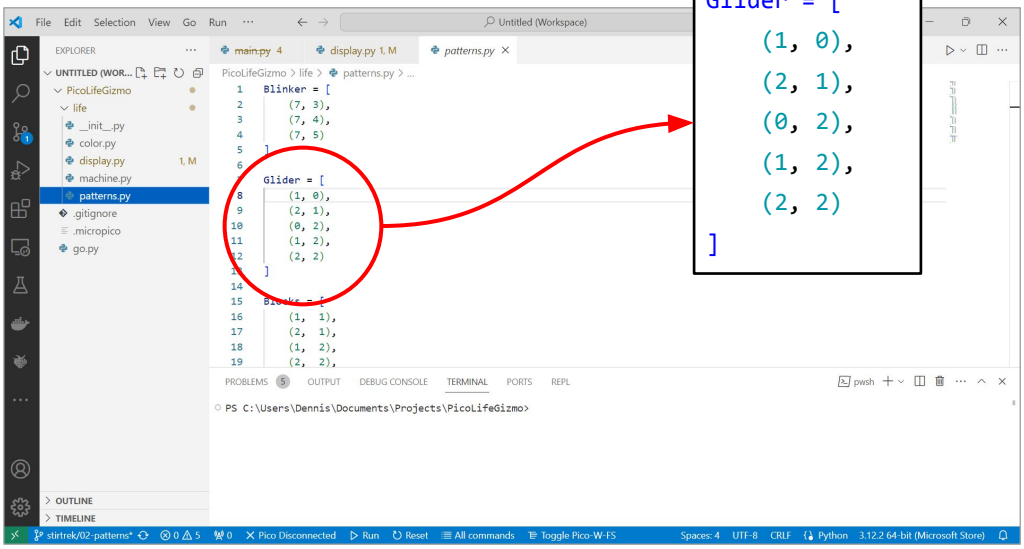


In the first version of the Game Of Life gadget, the state machines initial state, the “seed”, is hardcoded as a Python dictionary literal. Changing the seed, involves:

1. Start the computer.
2. Launch an editor.
3. Change the source code.
4. Upload the source to the microcontroller.
5. Reboot the gadget.

This is, shall we say, “sub-optimal.”

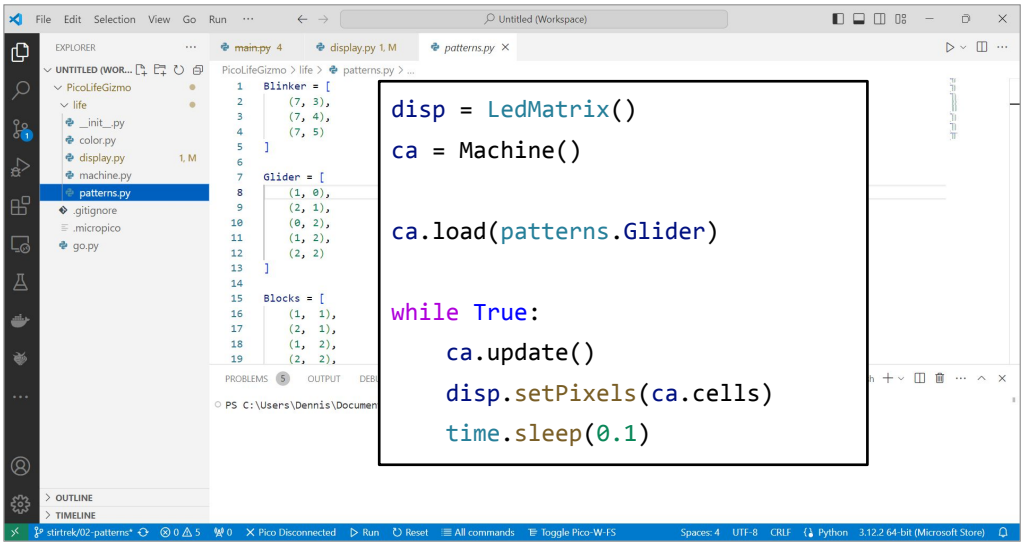
Configuring the Seed as a List



Let's change how we configure a seed. We'll make it a named list of tuples instead of a dictionary.

The state machine will have a **load()** method to transform the list into the dictionary that the FSM needs.

Configuring the Seed as a List

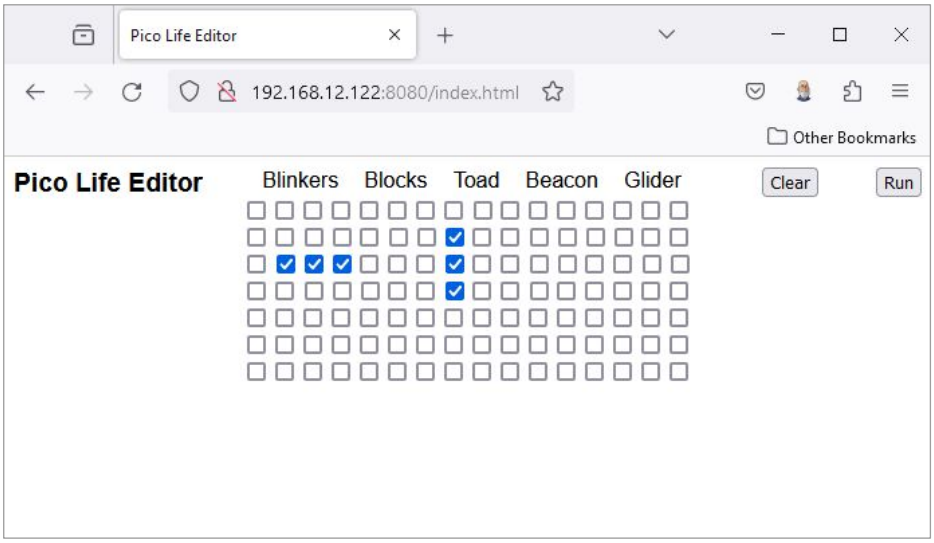


This isn't much better than the previous method of configuring the gadget but it does give us a couple of advantages:

1. We can store many seeds in a Python module and select among them by name.
2. The **load()** method allows us to take any properly formatted list as a seed.

Do those lists need to be hard coded into a module? Of course not!

Web-based Configuration

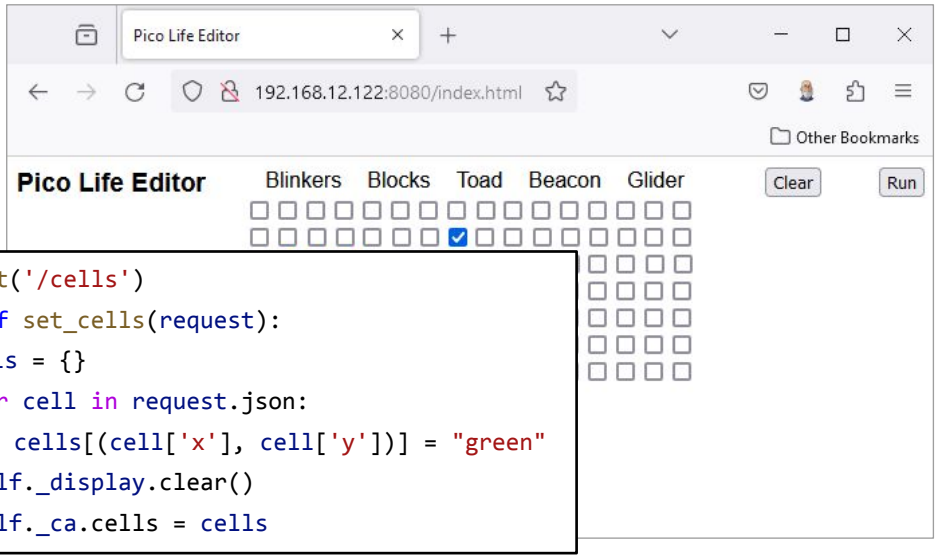


Since the Pico W has a WIFI radio, how about we use that?
We'll install some 3rd party code to make our lives easier.

- Mm_wlan for attaching to a WIFI network.
- Microdot for a web server.

We'll POST the results of a form to a web server running on Pico.

Web-based Configuration



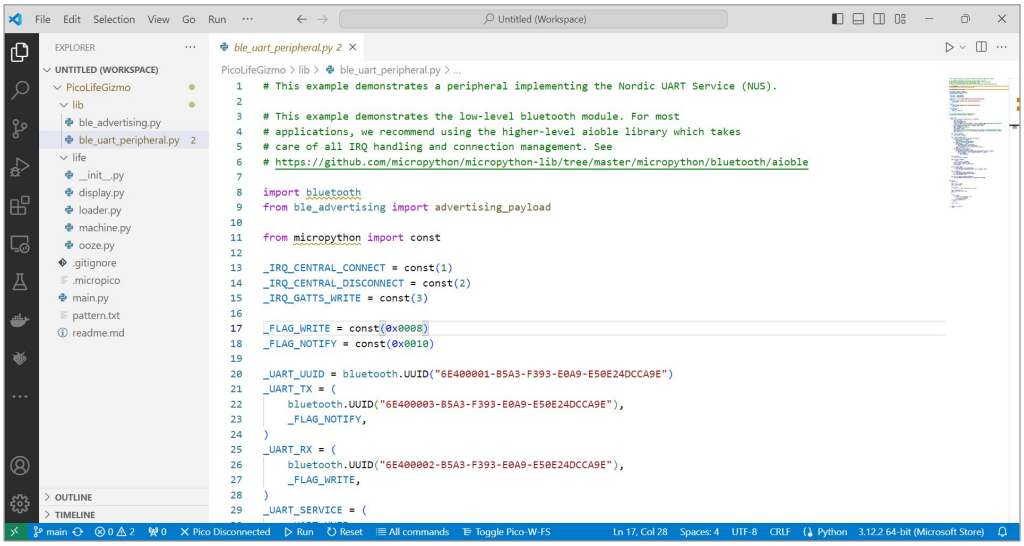

```
@app.post('/cells')
async def set_cells(request):
    cells = {}
     for cell in request.json:
        cells[(cell['x'], cell['y'])] = "green"
     self._display.clear()
     self._ca.cells = cells
```

Even with my awesome web skills and keen sense of design this solution to the configuration problem isn't any easier than the previous solution.

1. We need to provision the WIFI by hardcoding the SSID and the password.
2. We need to know the URL of the configuration page.
3. The UI is, how should i put this? Ugly.

Well, how about the other radio on the Pico W - Bluetooth?

Github is Your Friend



First, we need to get the Bluetooth radio on the Pico W to wake up. We have two options here, do it ourselves or find a library and use that. Since we want to succeed, let's go with Option 2.

The MicroPython repository on Github has plenty of example code that will give you a start with the potentially mind-boggling amount of Bluetooth information.

We'll use the **Nordic UART** service for this project. It provides a two-way channel for text data.

I copied two files from the examples repo into my project because I may not now all of the in-and-outs of MicroPython and Bluetooth but I do know how to read code.

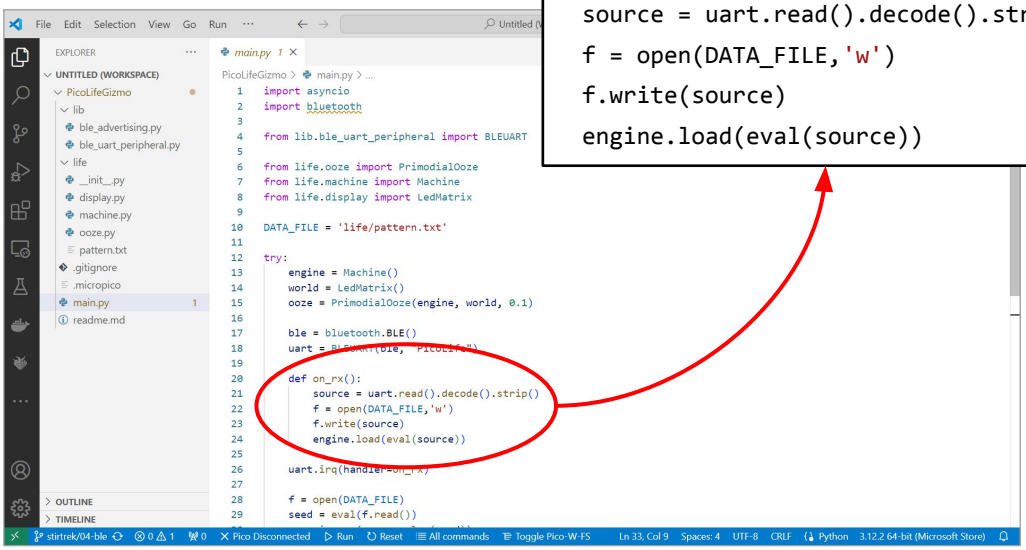
When looking at example code to determine what to incorporate into a project I have two criteria:

1. Simplicity
2. Pragmatism

It's hard to argue with "It Works."

Using the Bluetooth Module

```
def on_rx():
  source = uart.read().decode().strip()
  f = open(DATA_FILE,'w')
  f.write(source)
  engine.load(eval(source))
```

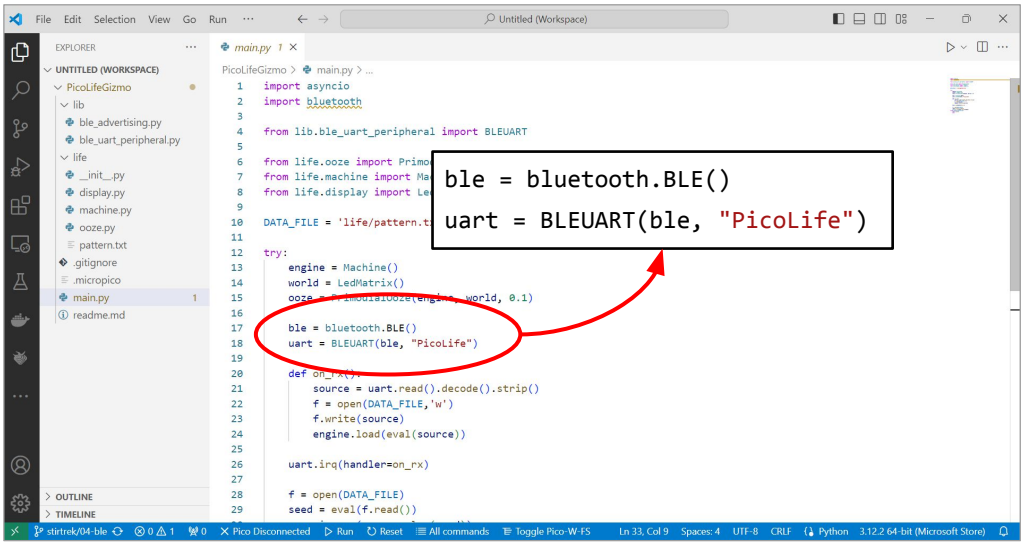


Actually using the module is simple.

First, we'll define a receive callback. The receive callback accomplishes two tasks;

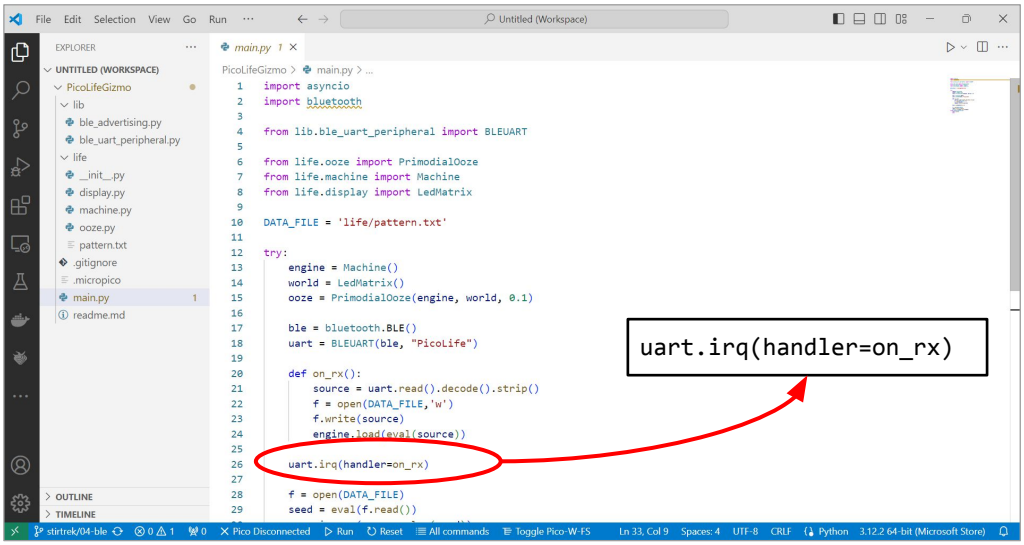
1. It saves the object to the gadget so that it is loaded upon boot.
2. It loads the received list of tuples into the engine.

Using the Bluetooth Module



Next, instantiate a Bluetooth object. This will also start advertising so that the gadget will be visible to Bluetooth scanning.

Using the Bluetooth Module



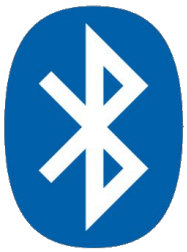
Finally, set the services callback.

If you need even more functionality out of your UART connection take a look at the **ble_uart_repl.py** example.

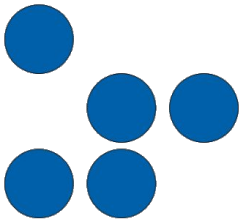
Working Together

MicroPython Scheduler

Bluetooth Process



Gadget Process

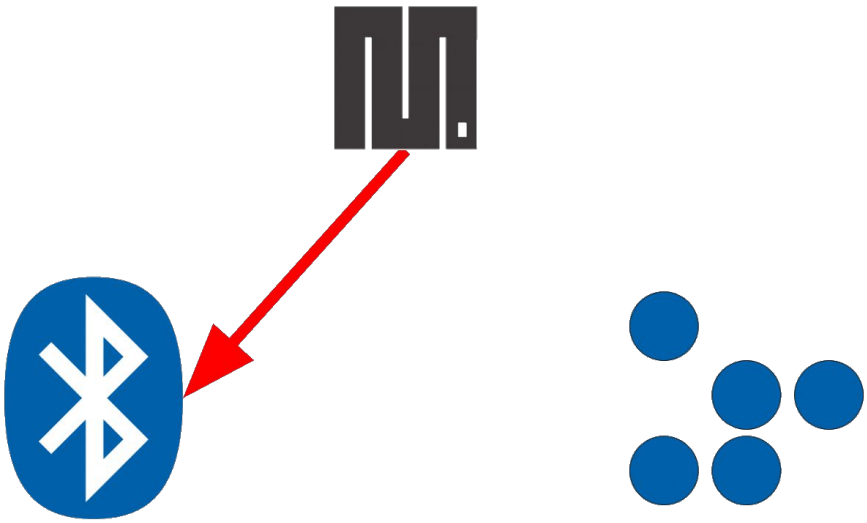


We want the gadget to continue blinking as well as respond to Bluetooth events. There are a couple of ways to do this;

- Preemptive multiprocessing
- Cooperative multiprocessing

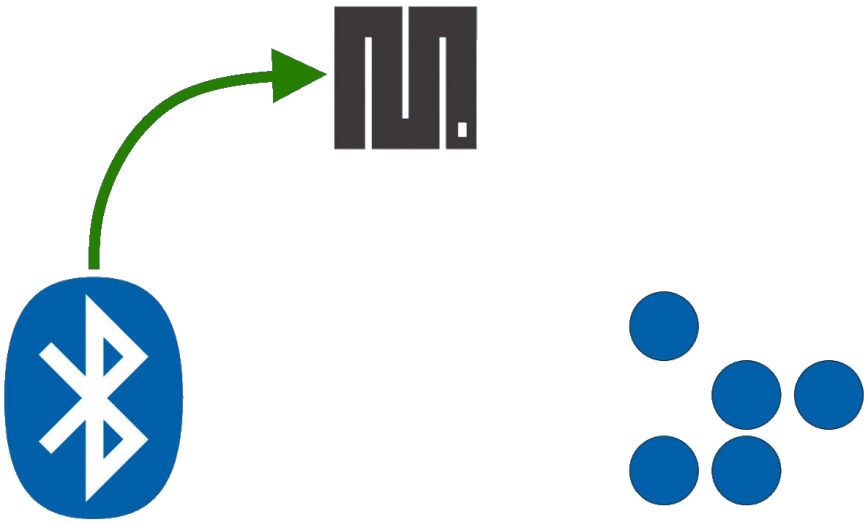
Both have a set of “processes” or “tasks” that are selected to run by a “scheduler.” The difference is how the scheduler regains control from a process.

Preemptive Scheduling



In a preemptive scheduler, a process is interrupted by the scheduler and control passed to the next process. Since the scheduler can interrupt the process at any point in its execution, the scheduler needs to manage the processes state.

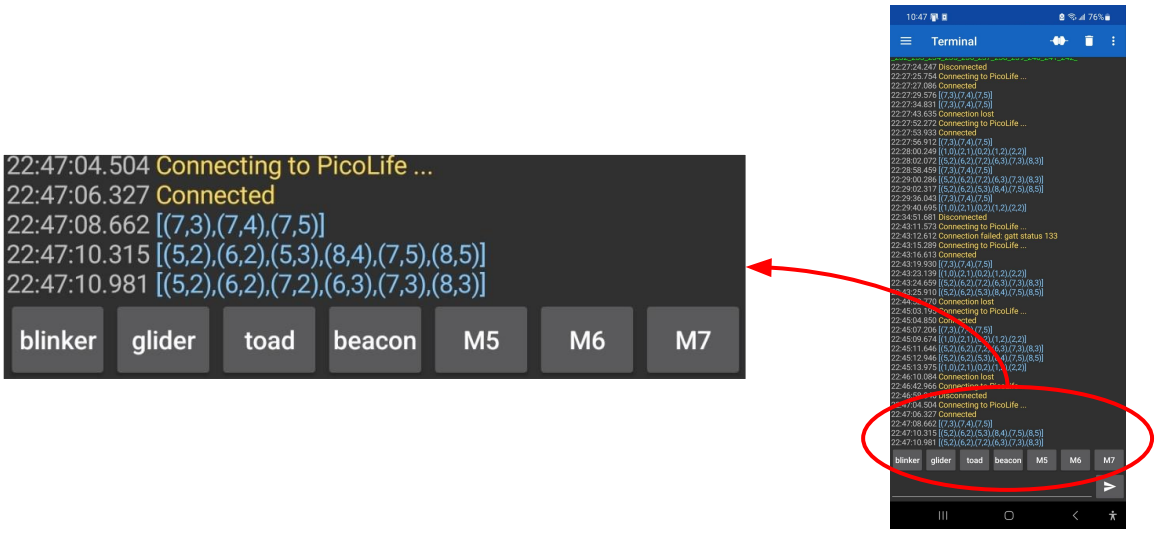
Cooperative Scheduling



With a cooperative scheduler, it is up to the process to ensure that it relinquishes control only when it is in a valid state. The scheduler doesn't need to manage as much state and can therefore be simpler. Simple is good when you have a microcontroller with limited RAM.

MicroPython provides a cooperative multitasking environment in the form of the **asyncio** library.

Bluetooth-based Configuration

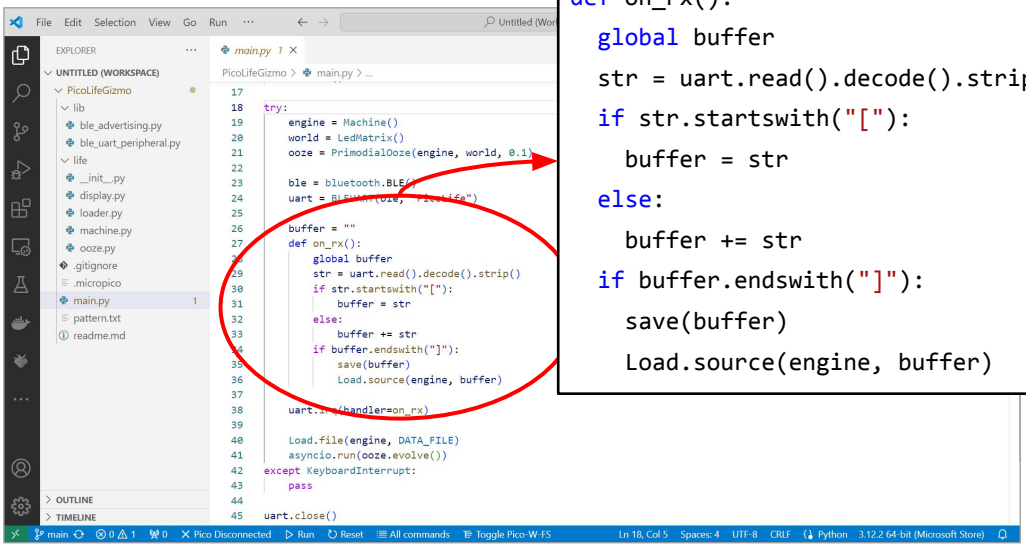


We have Bluetooth running on the Pico W, right? Are we sure? How do we test it out without writing the entire app?

This is an app named *Serial Bluetooth Terminal* that I download off of the Google App Store. It opens a connection to a Bluetooth device and then sends and receives text with the device.

It has a handy macro function and I assigned some Python source code to named macros. This is the proof-of-concept environment to test the Bluetooth code running on the Pico.

Buffering the UART Input



While testing with the terminal program, I found that a number of the macros didn't seem to be working properly; the longer macros were causing the gadget to throw a syntax-error exception.

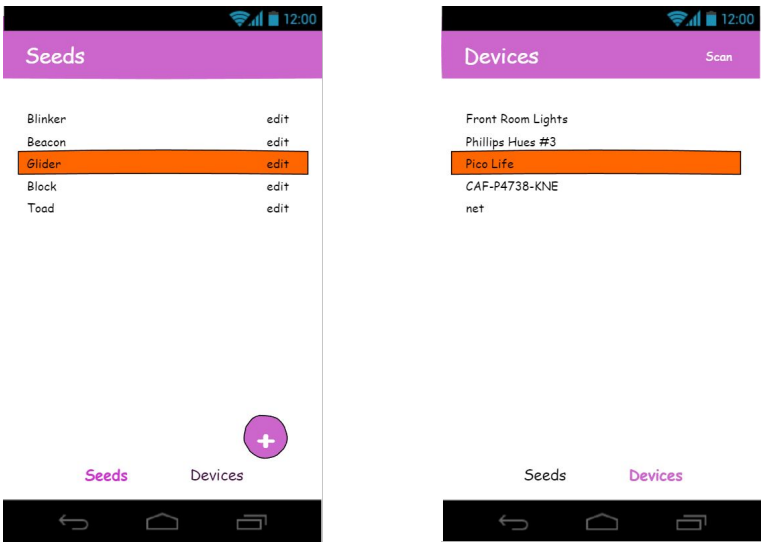
A few calls to **print()** revealed that my strings were being truncated. No problem. I'll just buffer the data until I get a complete string delimited by square brackets.

And that concludes the evolution of the gadgets Bluetooth configuration interface. So far we have:

1. A gadget made with a Raspberry Pi Pico W.
2. A way to set configurations over Bluetooth.
3. A mechanism to persist configurations across reboots.
4. A tool to make sure it's all working properly.

Next we'll look at the Android app we'll build to configure our gadget.

Mobile App Mockup



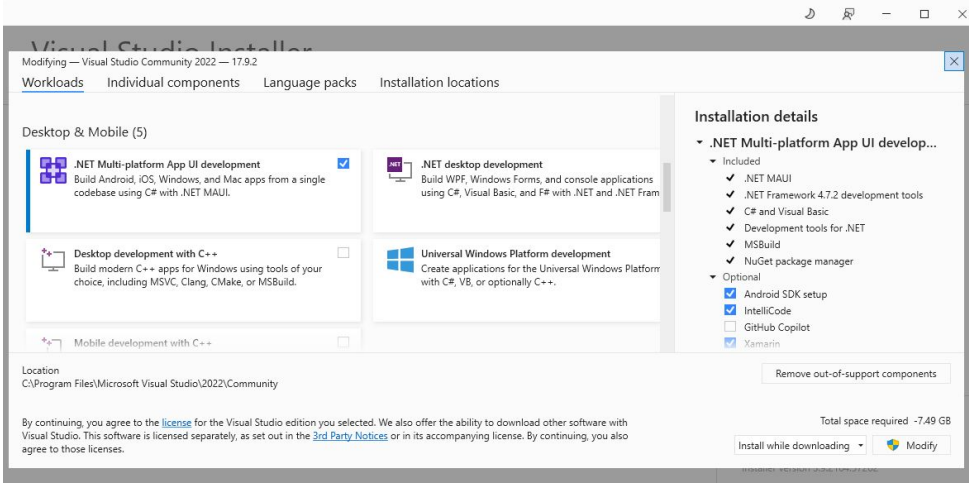
Let's get an idea of what we want to build. A ballpoint-and-napkin is all you really need or you can use a wireframe tool like Balsamiq or Pencil. You can also just crack open Visual Studio and start coding but after you've familiarized yourself with what's available I still think that it's a good idea to take a step back and clarify your ideas by sketching them out.

The mobile app is going to have two main screens.

The **Seeds** screen will show a list of patterns with buttons to add a new pattern, edit an existing pattern, and send the pattern to the gadget.

The **Devices** screen will manage the Bluetooth connection and provide buttons for scanning for available devices and connecting to a specific gadget.

Visual Studio Setup

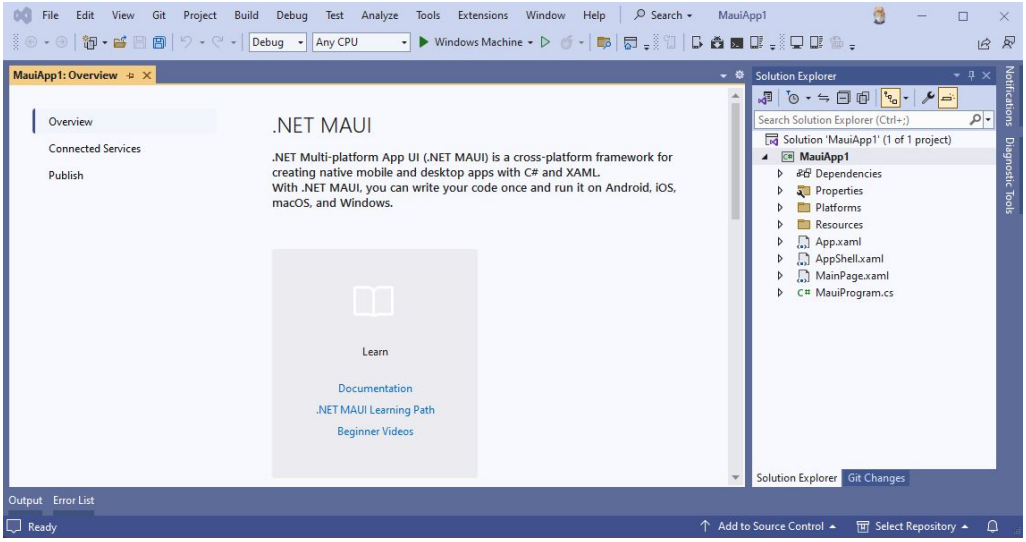


We'll be using *Microsoft Visual Studio 2022 Community Edition* to build the mobile app. Because I have an Android phone and not an iPhone I'll be concentrating on building an Android app. If you want to build for iPhone there are tutorials on the web that will help out.

Fire up the *Visual Studio Installer* and install the **.NET Multi-Platform App UI development** workload.

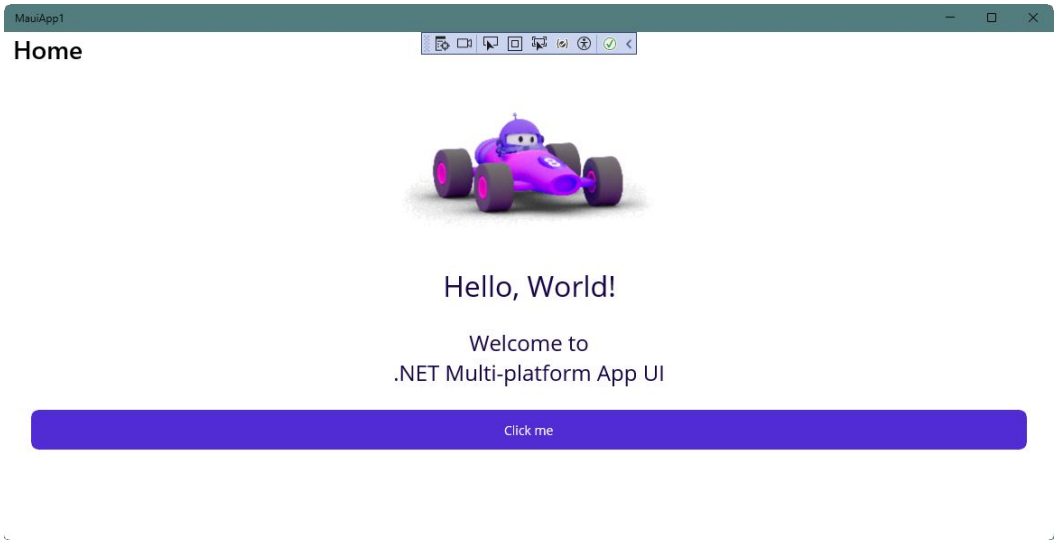
After installing the workload, launch *Visual Studio* and create a new MAUI app solution.

Visual Studio Setup



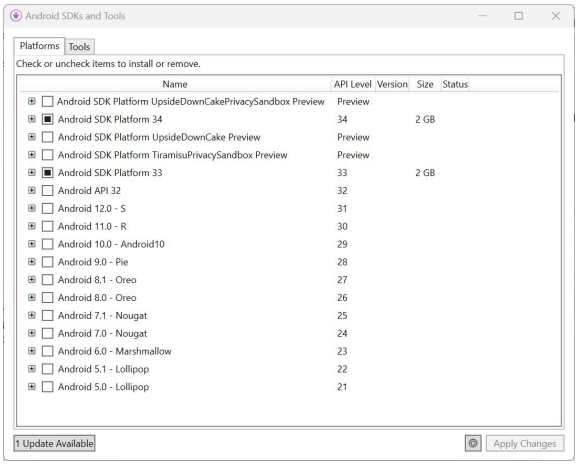
The new solution has some example code and links for documentation and tutorials. I don't know about you but I like to run any example code to make sure that my environment is set up correctly. Remember that the **M** in **MAUI** stands for *multi-platform* so let's hit the *Run* button…

Visual Studio Setup



… to see the example running under the *Windows Machine* target.

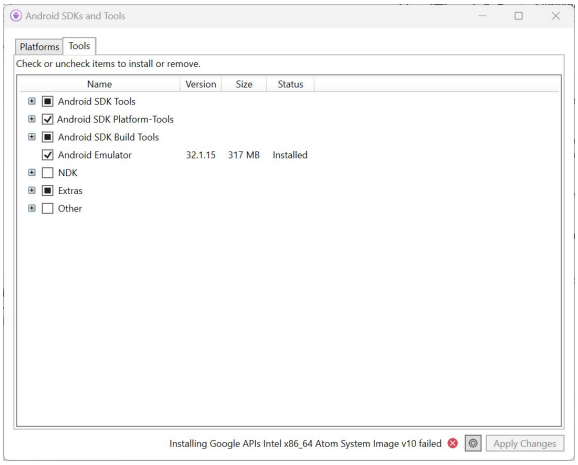
Visual Studio Setup for Android Development



It's nice that the example compiles and runs but that doesn't get us to having *multi-* covered in the *multi-platform* promise. There are some tasks that we need to accomplish before we can debug our mobile app.

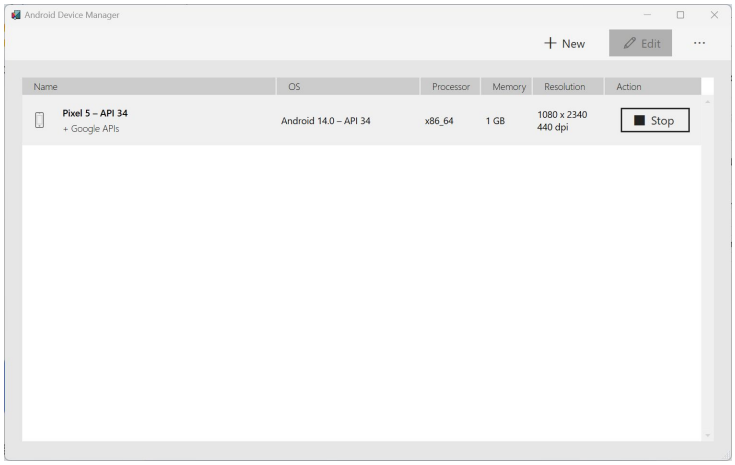
First we need to make sure that the Android SDK is installed, from the *Tools* menu select *Android SDK Manager…*

Visual Studio Setup for Android Development



… with the correct tools.

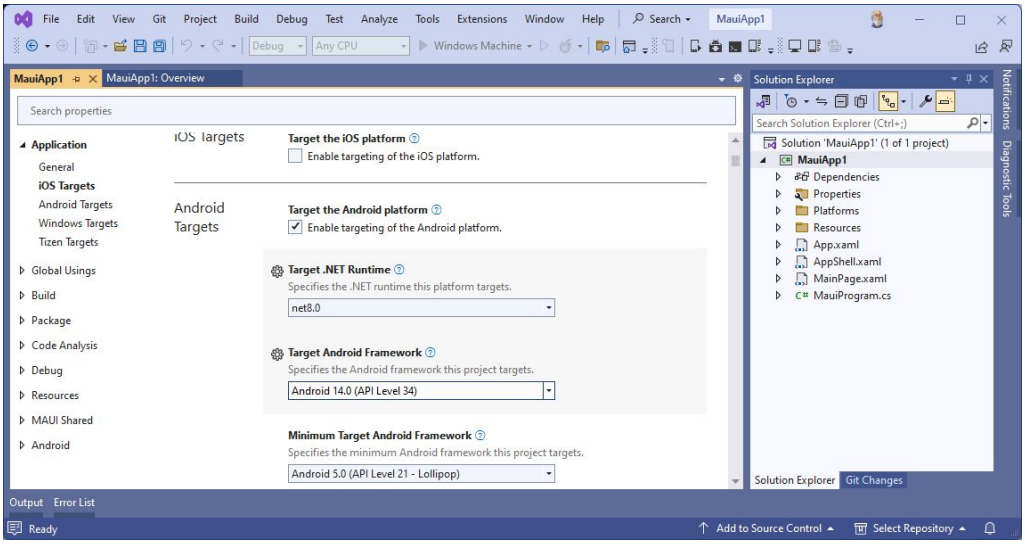
Visual Studio Setup for Android Development



Make sure we have an emulator configured, from the *Tools* menu select *Android Device Manager…*

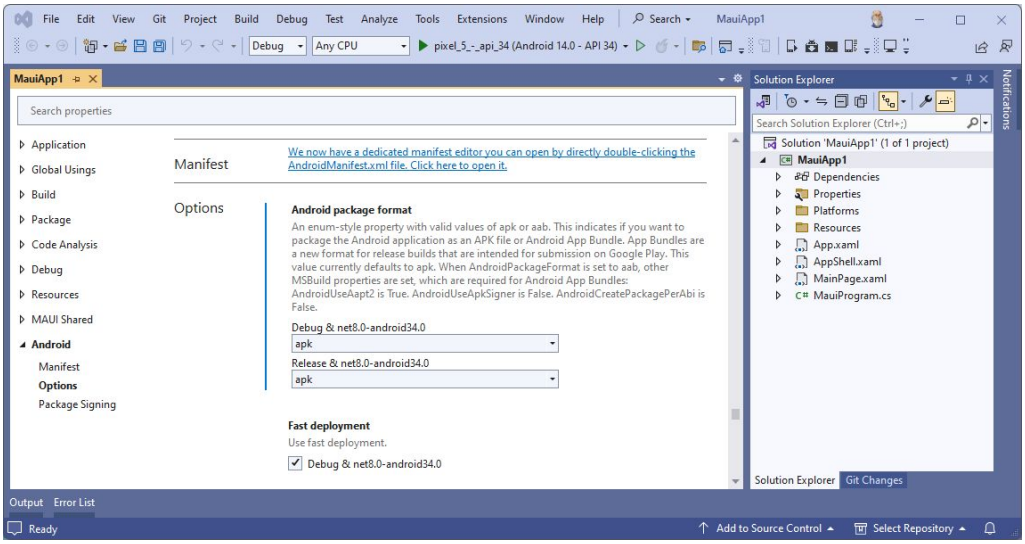
a. Create an emulator
b. Start the emulator

Visual Studio Project Setup for Android Development



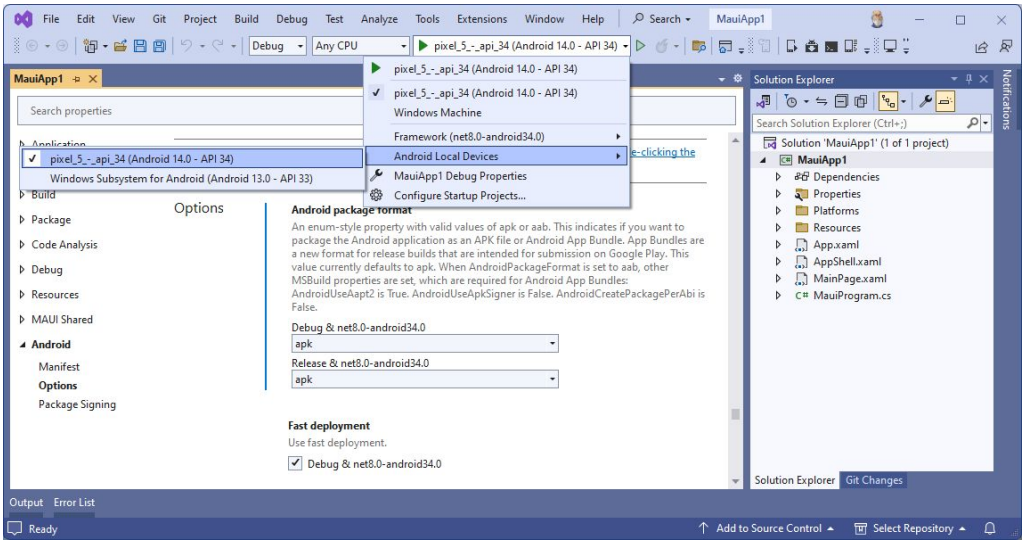
Next, we need to enable the Android target in the projects properties sheet.

Visual Studio Project Setup for Android Development



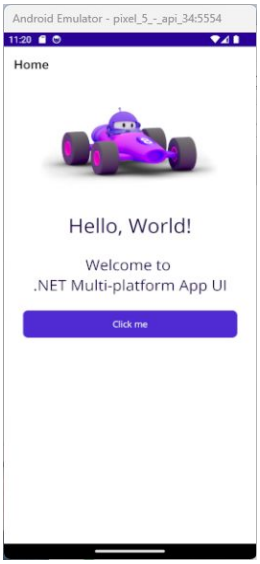
Because we'll be side loading the app, set the Release package format to **apk**, also in project properties.

Visual Studio Setup



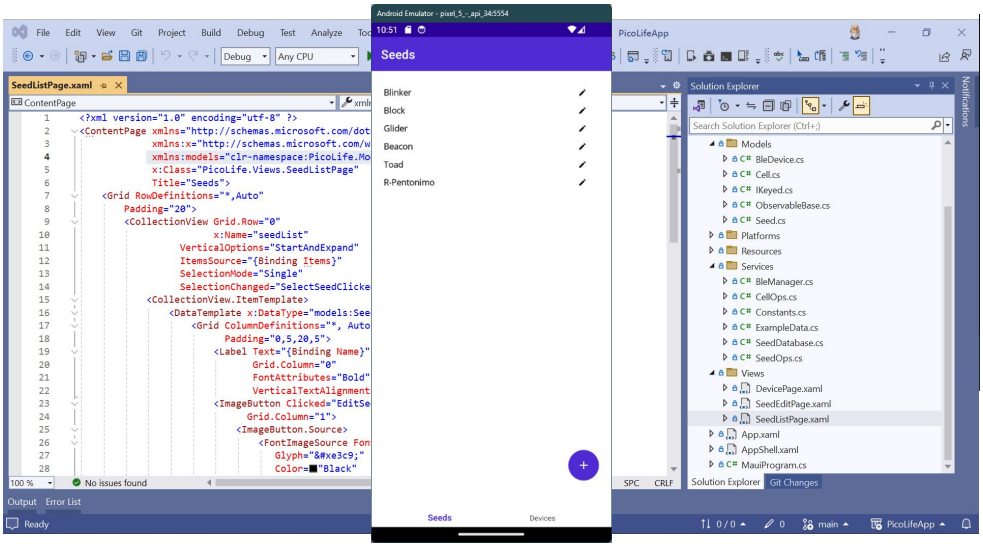
Select the emulator that we created earlier, hit the **Run** button…

Visual Studio Setup



…and marvel at our multi-platform app.

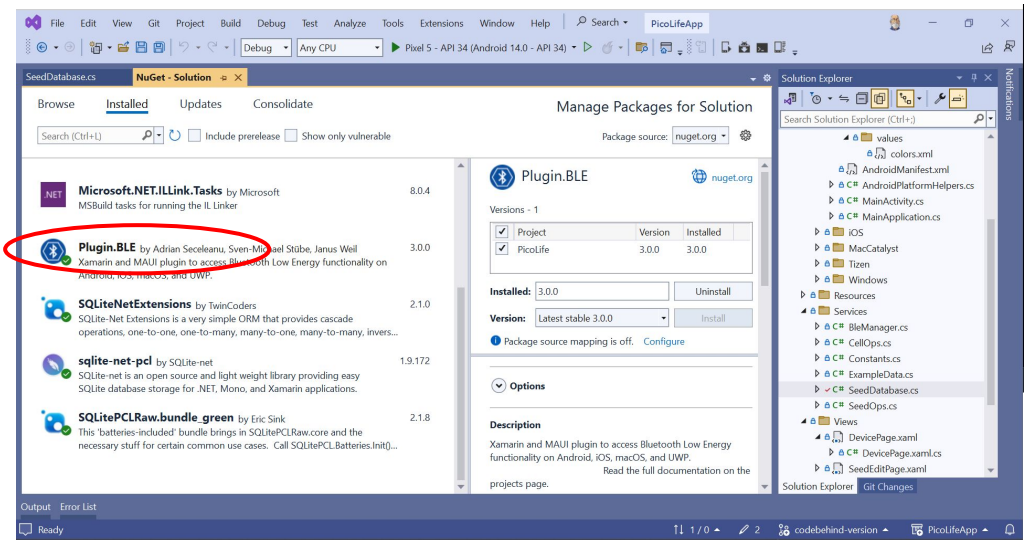
Pico Life App



Finally we get to the whole point of this session - Multi-platform development with MAUI and .NET.

I'll be concentrating on describing the features that I used to build the configuration tool for my gadget and the places where I stumbled. Remember the MAUI default project "Welcome" screen? There is a huge amount of information, both from Microsoft and from 3rd parties, on how to build apps with MAUI.

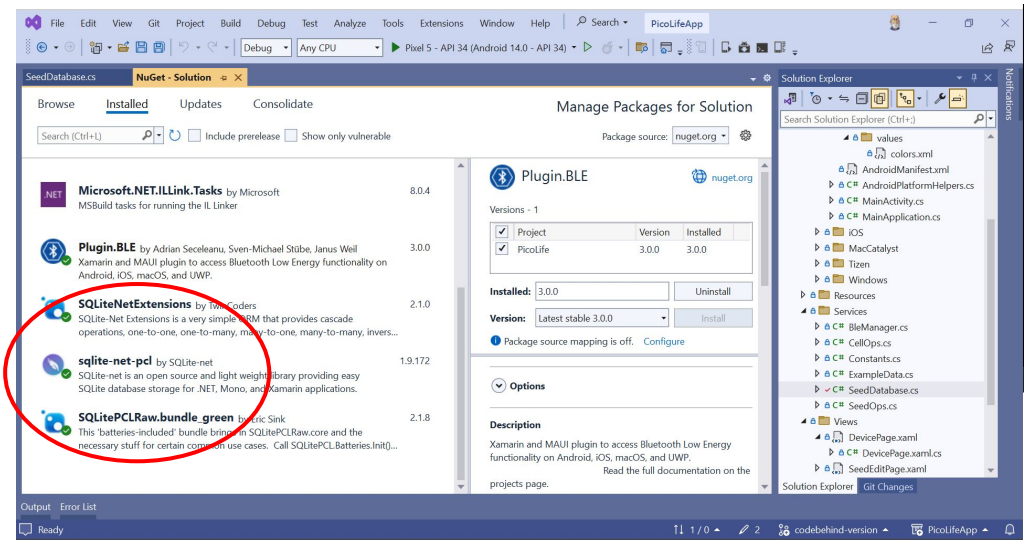
Pico Life App



Just as we depended on third-party Python libraries for the microcontroller project, we'll use third-party packages on the mobile app side as well.

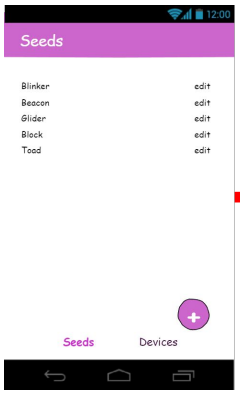
The **Plugin.BLE** gives easy access to the Bluetooth Low Energy functionality on the mobile device.

Pico Life App

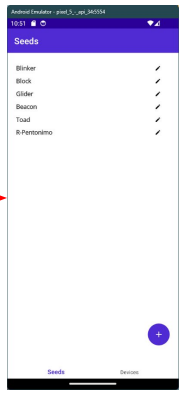


While these **SQLite** packages give us a simple, on-device, relational store for our list of seeds.

From Mockup To Mobile



1. MVVM vs. Code-behind
2. Dependency Injection
3. Defining the UI with XAML
4. Navigating between Screens
5. Interacting with the Hardware
6. Debugging on the Metal
7. Deploying to Android



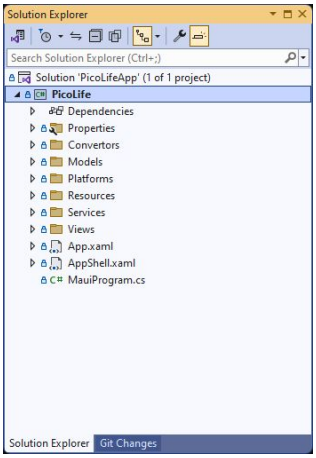
Here's what we'll cover in the rest of the session:

1. MVVM vs Code-behind
2. Dependency Injection
3. Defining the UI with XAML
4. Navigating between Screens
5. Functionality as Service
6. Interacting with the Hardware
7. Debugging on the Metal
8. Deploying to Android

That seems like a long list so I guess we'd better be at it.

MVVM vs. Code-behind

Model - View - ViewModel	Code-behind
Loosely coupled	Tightly coupled
Rich example code	Has examples
Easily testable	Difficult to test
Separation of concerns	Big ball of mud
More maintainable	Spaghetti



The **Model-View-ViewModel** pattern splits functionality into loosely coupled classes to provide better testability and maintainability of an application. In a large, commercial application you would want to use the **MVVM** pattern for just those reasons.

If you look at the *Solution Explorer* pane for the project you won't see a folder called **ViewModels**. Like all patterns in software development, MVVM works best when there is a reason to use it.

Our application is very simple so we are going to dispense with the complexity of the **MVVM** pattern and use the **code-behind** model. Each view will consist of an XAML file describing the widgets and a C# file containing the code for manipulating the UI. The XAML will bind to properties of the class and event handlers will process user interactions.

If you decide to use the **MVVM** pattern in your app, you will definitely want to use the *MVVM Community Toolkit*. Many of the examples that you find on StackOverflow will be using the toolkit which is a plus and you'll see how the toolkit helps to manage the complexity inherent in the **MVVM** model.

Dependency Injection

```
public DevicePage()
{
    InitializeComponent();

    BleManager = new BleManager();
}
```

```
public DevicePage(BleManager ble)
{
    InitializeComponent();

    BleManager = ble;
}
```

Before getting into XAML, let's look at interfacing the app with the mobile device. Your phone has a few radios, a camera, storage, and various internal sensors. Your app needs software to read from those devices and to control some of them. In our case we'll have a class to manage the Bluetooth radio.

Here are two methods for getting an instance of the *BleManager* into our **DevicePage** instance. In the first example, the BleManager is instantiated by the **DevicePage** constructor. This leads to tightly-coupled code. In the second example, an instance of the class is passed in via the constructor.

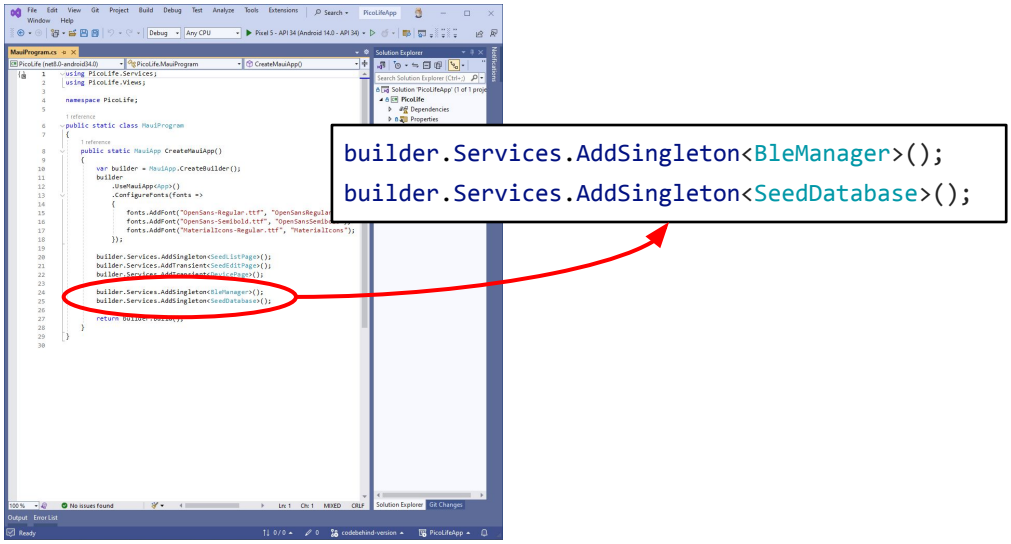
Dependency Injection

```
public DevicePage()
{
    InitializeComponent();

    BleManager = new BleManager();
}
```

```
public DevicePage(BleManager ble)
{
    InitializeComponent();

    BleManager = ble;
}
```

We don’t want instance creation to be mucking about in our UI code-behind files so we’ll inject it into the app using **dependency injection**. The magic of automatically passing objects via the constructor is provided by an **Inversion of Control** container. This breaks the coupling between the **DevicePage** and the *BleManager* which in turn allows for better maintainability and a cleaner code base.

Dependency Injection

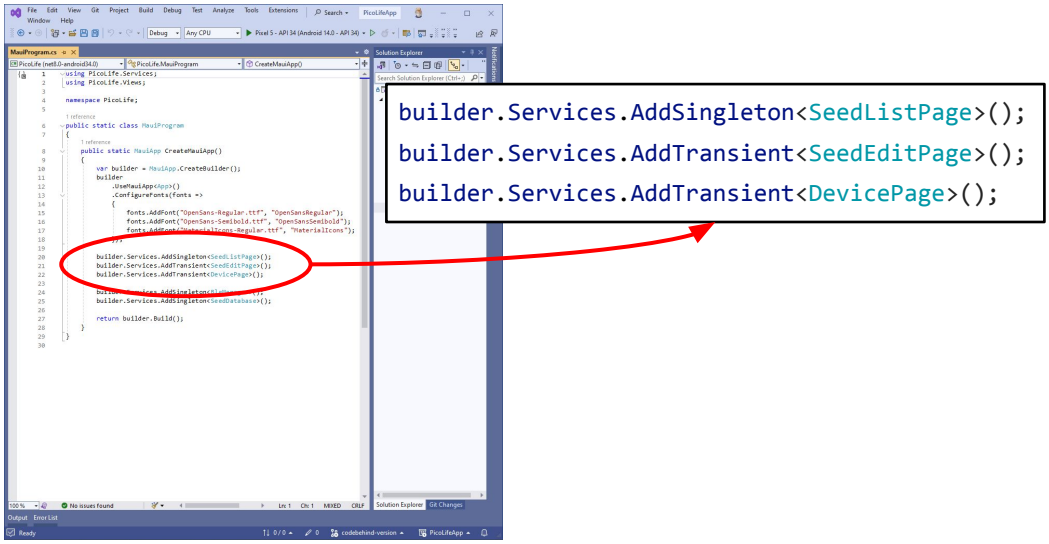


So how do we invoke this sweet, sweet magic?

The *MauiProgram* class is the entrypoint of the app and is created as part of the solution template.

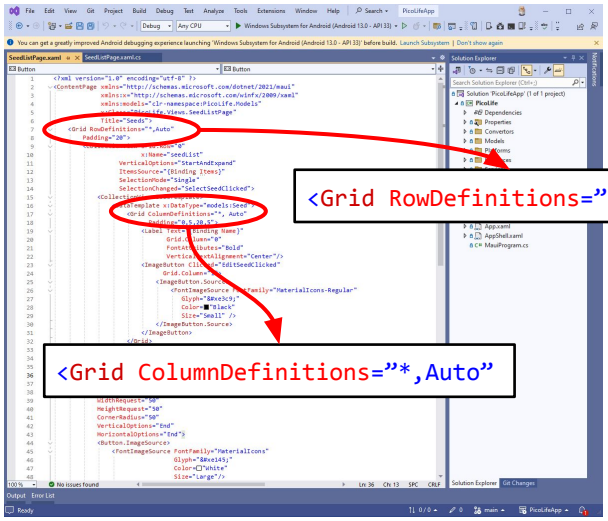
We **register** the types of our service classes with the container. Here we are telling the container that whenever a *BleManager* is required, use the singleton instance that has already been created.

Dependency Injection



We can also register transient objects. In this case the container will create a new instance of the *SeedEditPage* everytime we navigate to that page. The page will already have instances of dependencies that were injected during construction.

Defining the UI with XAML



Layouts

- StackLayout
- AbsoluteLayout
- Grid
- Canvas

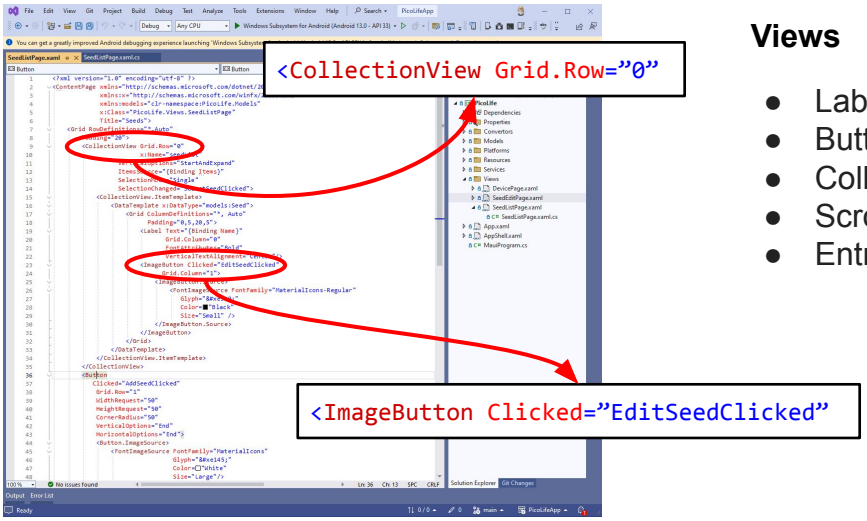
Extensible Application Markup Language or **XAML** is how we describe the user interface of our application. A XAML element describes either a **Layout** or a **View**. Properties on those elements refine visual details of the element, define child elements, and bind values to the underlying page instance.

Layouts

A layout determines how items are positioned on the screen. A **StackLayout** will position elements in a single row or column. A **Grid** layout will position things based on row and column identifiers. A **Canvas** gives you something to draw on.

- StackLayout
- AbsoluteLayout
- Grid
- Canvas

Defining the UI with XAML



Views

- Label
- Button
- CollectionView
- ScrollView
- Entry

Views

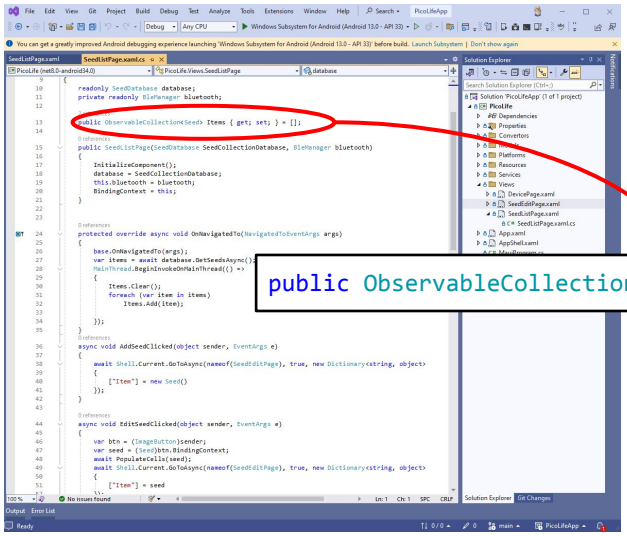
Views are the actual components that the user will interact with. If you had a really long list of items to display you might put a **CollectionView** inside of a **ScrollView**. Your **CollectionView** might layout the properties of each item in a **Grid** layout which in turn uses **Label** views to display the items values.

- Label
- Button
- CollectionView
- ScrollView
- Entry

The hierarchical nature of XAML might make you wish for a visual designer; that tool is named **Blend** and is included in a couple of the Visual Studio workloads. It is geared towards desktop and UWP development and is available with the *Visual Studio Community Edition*.

A simple app such as this doesn’t need a visual designer, using **Hot XAML Reload** you can easily tweak the UI.

Defining the UI with XAML



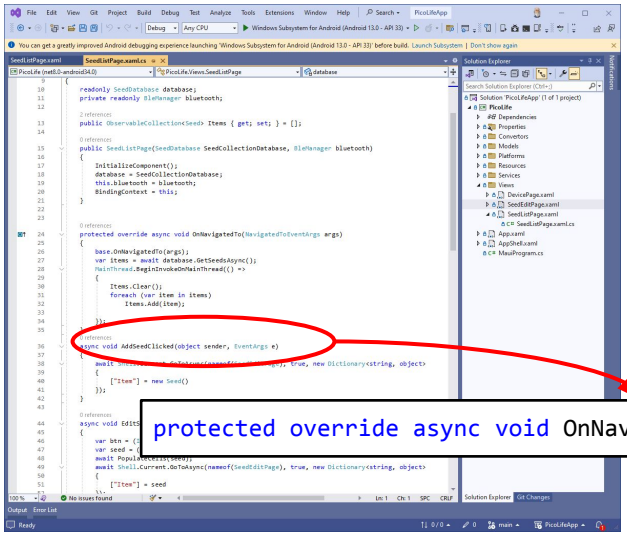
Code behind contents

- Properties
- Event handlers
- Service references

Every XAML file is going to have a corresponding C# file, aka the **code-behind** file. It is going to contain the actual code that is run when the user interacts with the app.

Properties hold values that the XAML will bind to for display. Properties can be scalars or collections.

Defining the UI with XAML



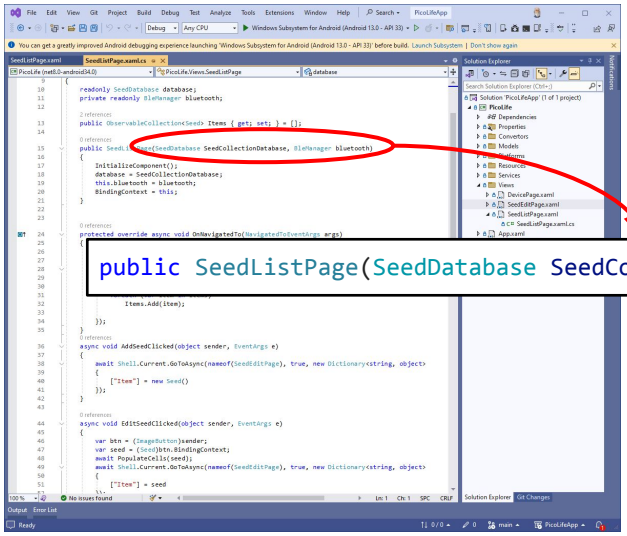
```
protected override async void OnNavigatedTo(NavigatedToEventArgs args)
```

Code behind contents

- Properties
- Event handlers
- Service references

Event Handlers are invoked when the user interacts with the app or lifecycle events occur.

Defining the UI with XAML



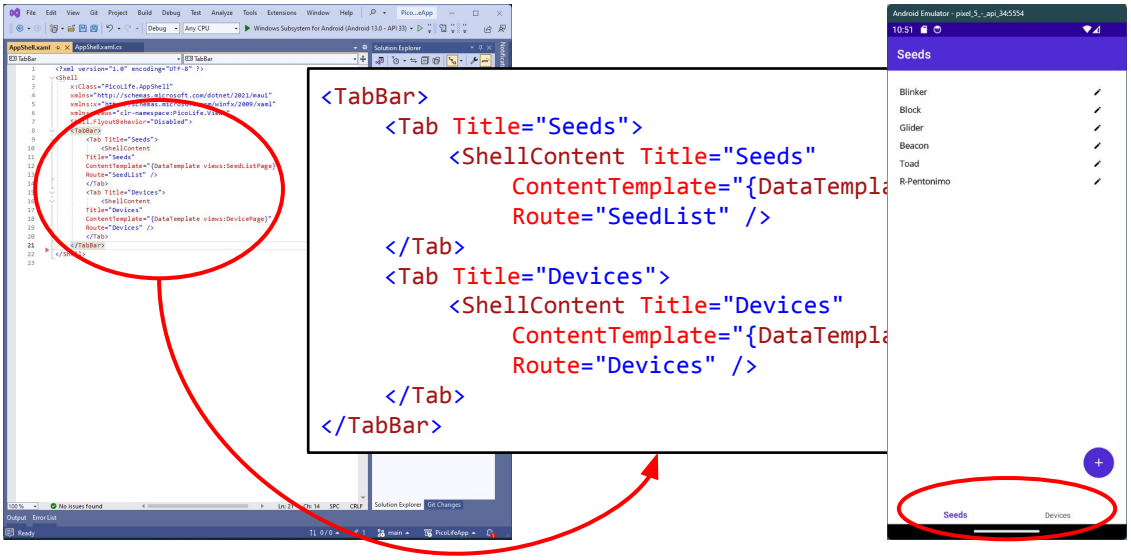
```
public SeedListPage(SeedDatabase SeedCollectionDatabase, BleManager bluetooth)
```

Code behind contents

- Properties
- Event handlers
- Service references

Service references are instances of classes that provide persistence, validation, and other business logic. The code-behind file is where services are injected by the IoC container.

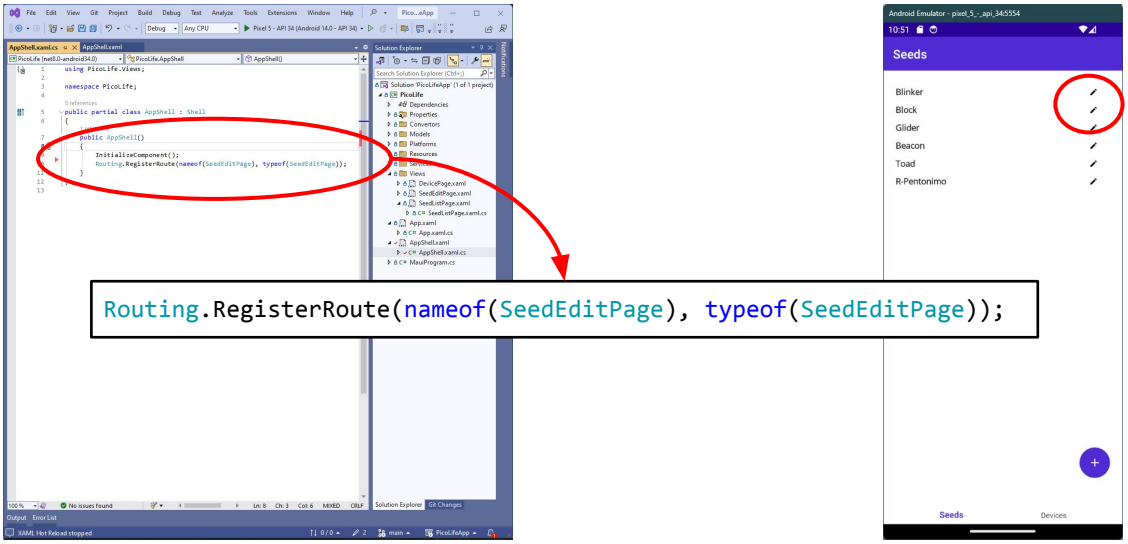
Navigating between Pages



The MAUI *Shell* contains a URI-based navigation mechanism that uses *Routes* to navigate to different pages of the app.

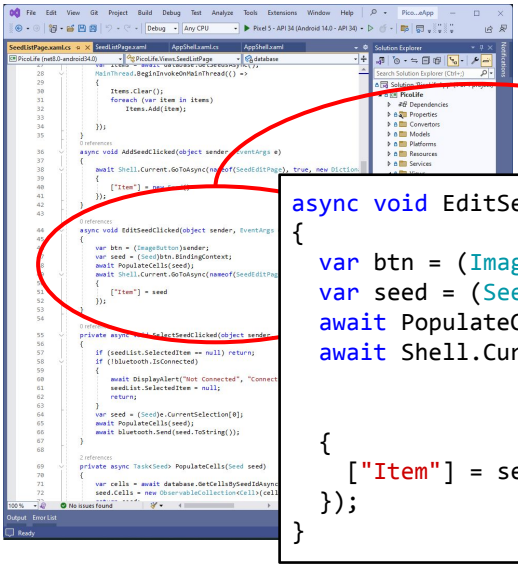
Using XAML, we can define a tabbar for navigating between the two main pages of the app.

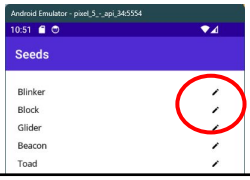
Navigating between Pages



What about other routes in the application? The buttons on the right of the *Seeds* list navigates to a page for editing a seed. We register that route in the code behind of the **AppShell.xaml.cs** …

Navigating between Pages






```
async void EditSeedClicked(object sender, EventArgs e)
{
  var btn = (ImageButton)sender;
  var seed = (Seed)btn.BindingContext;
  await PopulateCells(seed);
  await Shell.Current.GoToAsync(nameof(SeedEditPage),
                                true,
                                new Dictionary<string, object>
  {
    ["Item"] = seed
  });
}
```

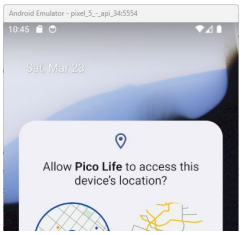
… and in the event handler for the button we call *Shell.Current.GoToAsync()* with the seed as a parameter.

Interacting with the Hardware

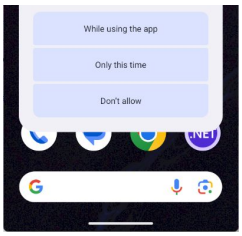
```
public interface IBleManager {
        bool IsConnected { get; }
        bool IsScanning { get; }
        ObservableCollection<BleDevice> Devices { get; }
        BleDevice ConnectedDevice { get; }
        Task ScanAsync(int timeout = 10000);
        void OnScanTimeout(object sender, EventArgs e);
        Task CancelAsync();
        Task ConnectAsync(BleDevice device);
        Task DisconnectAsync(BleDevice device);
        Task DisconnectCurrentAsync();
        Task Send(string data);
        void Dispose();
}
```

We've seen how the Bluetooth service is injected into the **DevicePage**; what does the service actually do? The *BleManager* class controls scanning, connecting, and disconnecting from the gadget over the Bluetooth radio as well as sending data to the gadget. This is not the hard part, in fact with the amount of example code on the web it is pretty easy to take those examples and bend them into a shape that fits your app.

Interacting with the Hardware



https://developer.android.com/guide/topics/connectivity/bluetooth

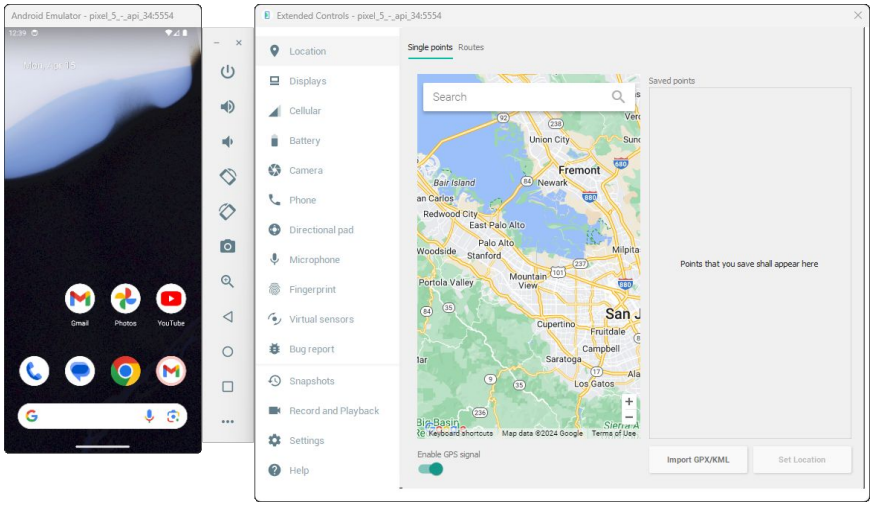


The hard part is getting the **PERMISSIONS** right. It seemed like every blog entry or Stackoverflow post listed a different set of permissions for getting Bluetooth running on Android.

CLICK

Here is the secret. Bookmark the developer documentation for any platform that you are targeting. Once I added the correct permissions to the Android manifest file and the *DroidPlatformHelpers* class I was able to scan for the gadget.

Debugging on the ~~Emulator~~ Metal

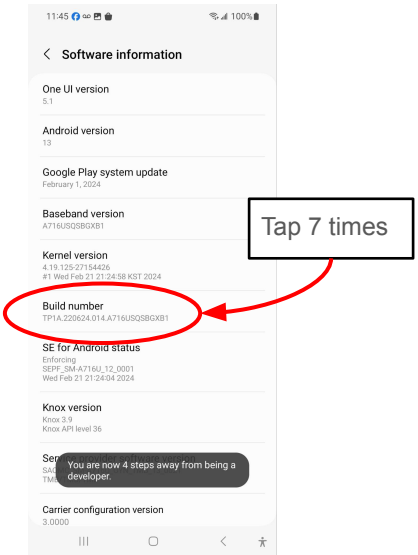


The emulator gives you access to many parts of the emulated device such as the camera, mapping, and audio. Notice that the emulator has no support for emulating Bluetooth functionality.

How are we going to debug the Bluetooth code?

We could build a distribution, copy the **apk** to the device, install the app, and run the app. This could tell us that the app isn't working properly but it doesn't tell us *where* the problem is. If we want to debug the scanning and connection features of the app we need to do that on an actual mobile device.

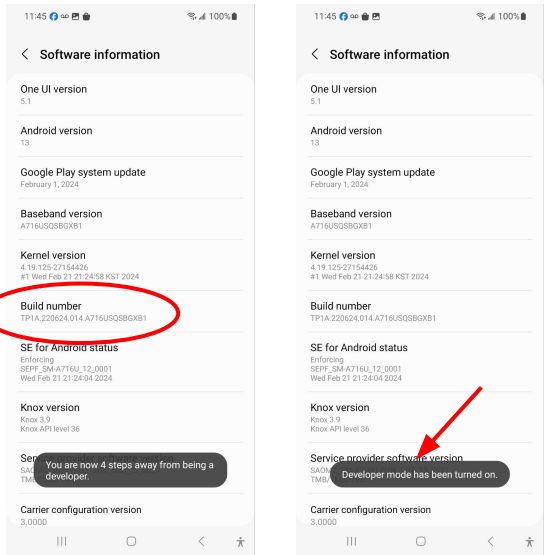
Debugging on the Metal



We need to activate “developer mode” and “USB Debugging” on the phone.

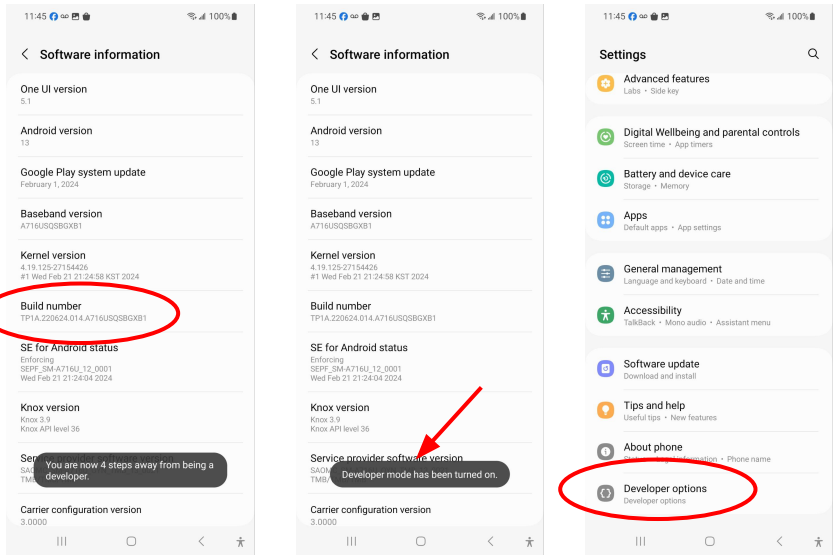
Open up **Settings** on your Android device and navigate to *About Phone*, *Software Information* and tap 7 times on *Build number* …

Debugging on the Metal



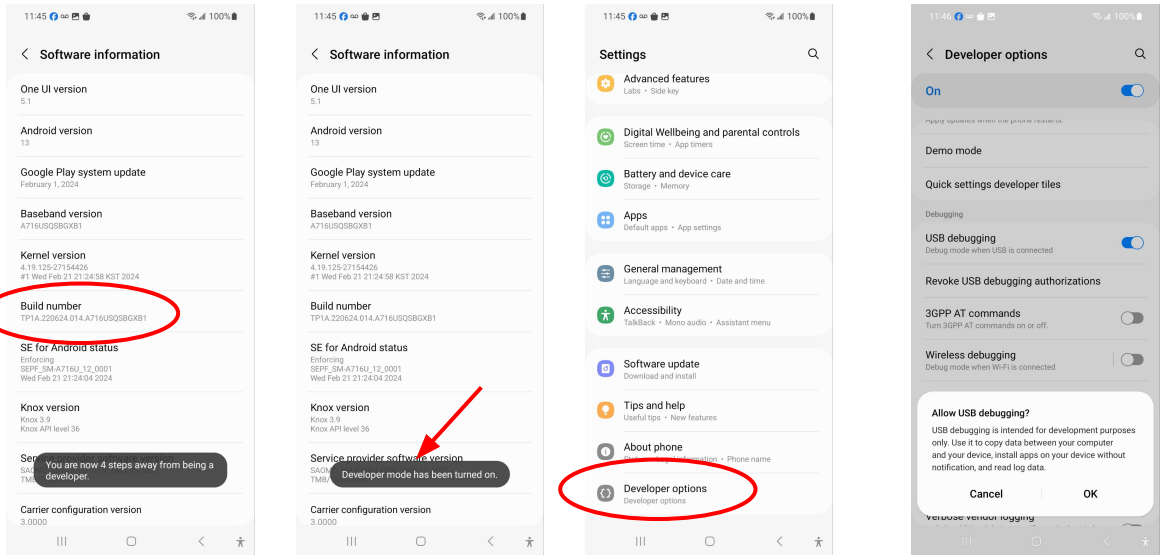
… which will activate developer mode.

Debugging on the Metal



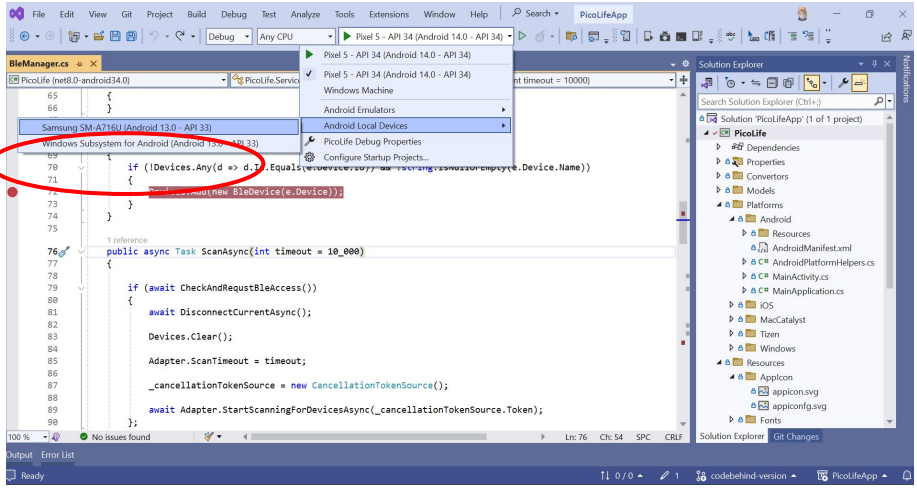
Navigate back to *Settings* and select *Developer Options*…

Debugging on the Metal



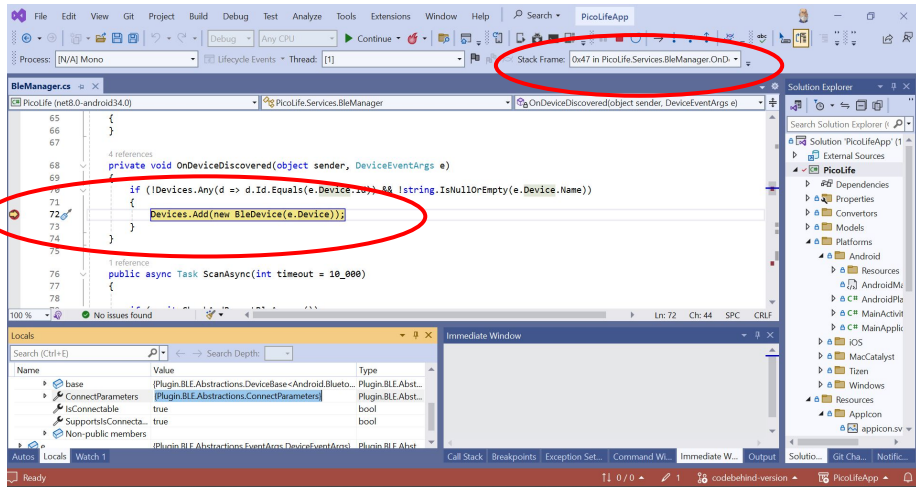
… and enable *USB Debugging*

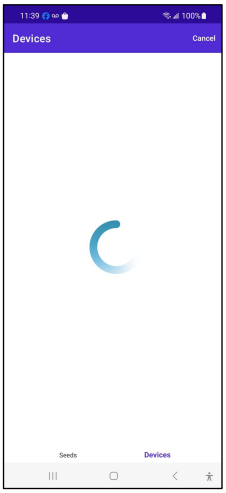
Debugging on the Metal



After plugging in your device to a USB port on your development machine, you should be able to select the device, hit F5 …

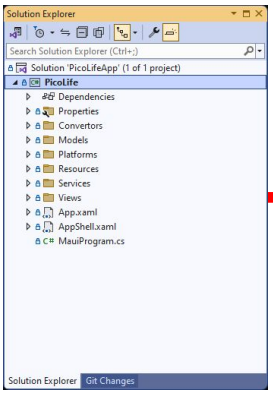
Debugging on the Metal



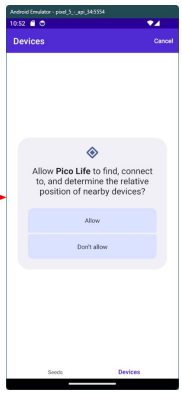


… and debug the code as it's running on the device. Here I've set a breakpoint in the device detected handler and the device is waiting patiently to continue.

From Dev to Device



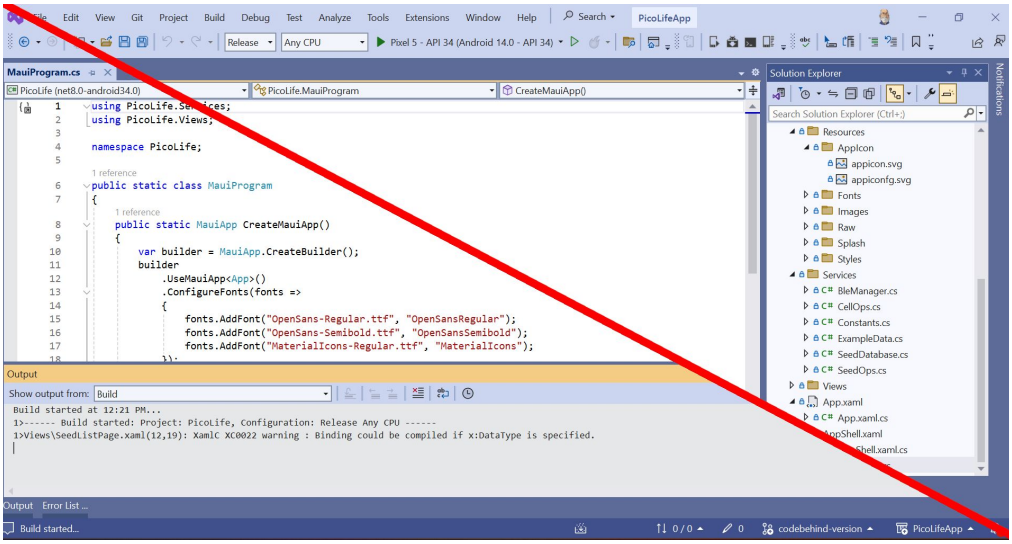
1. Build a Release
2. Publish
3. Distribute
4. Sign
5. Copy to device
6. Install



Up until now we've been working in our development environment. It's time to get our app onto a device so that we can actually use it when we are out and about in the real world. To do that, we need to:

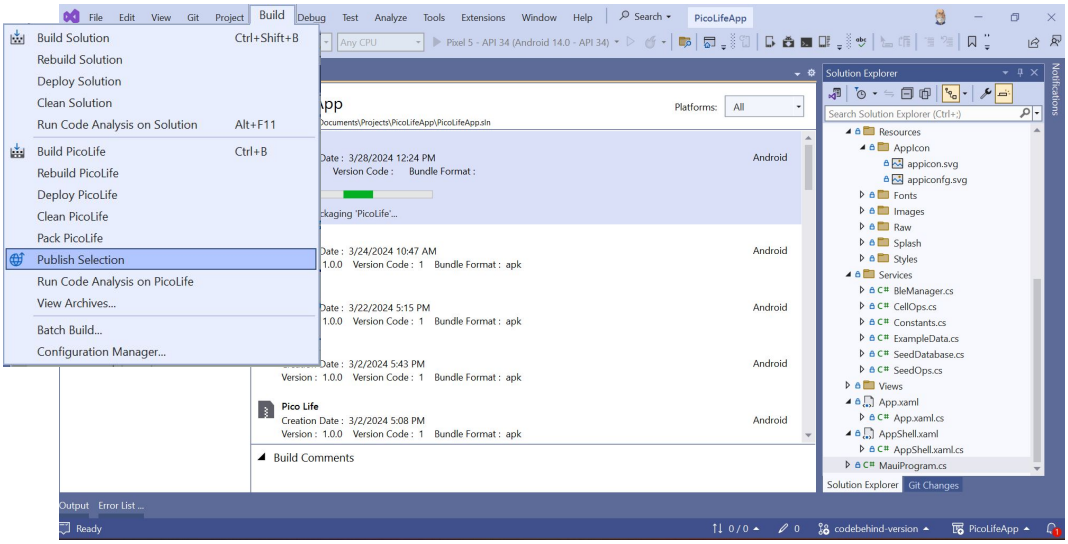
1. Build a Release
2. Publish
3. Distribute
4. Sign
5. Copy to device
6. Install

Build a Release



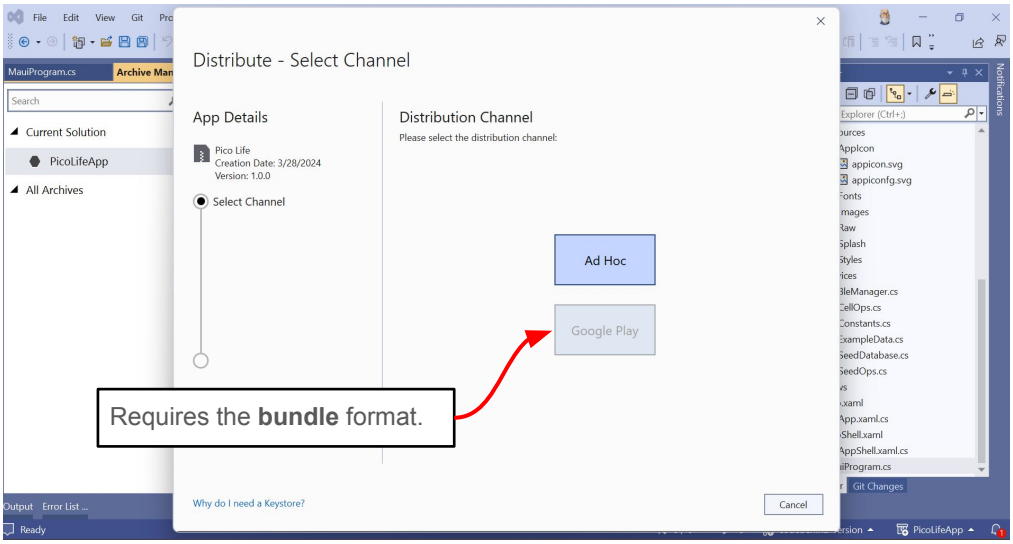
First we'll build a release. Select **Release** from the Solution Configuration drop down and then **Build** from the menu.

Package the Binaries and Resources



Next we need to package the release into a form that we can transfer to our mobile device. Selecting **Publish** from the menu will package up the code and resources into an **apk** file. Remember when we were setting up the Android properties and we select **apk** as the package format? This is why.

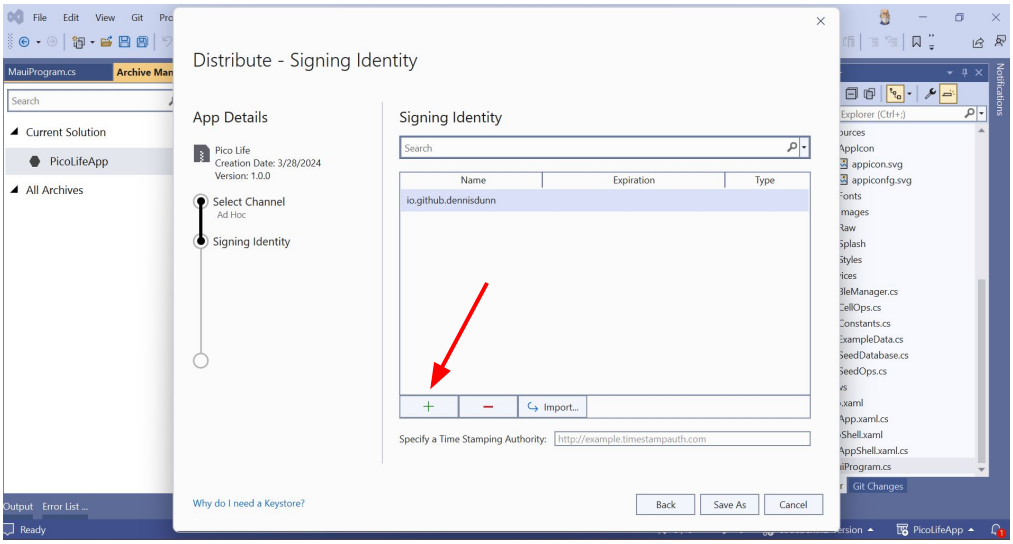
Select the Distribution Channel



Click the **Distribute** button to bring up the channel selection dialog.

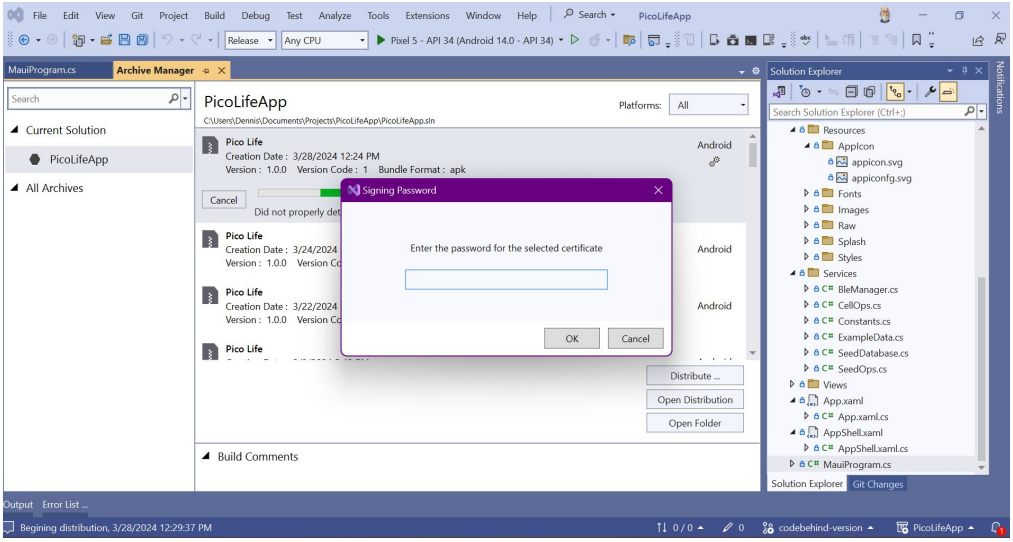
Because the package format is **apk** the **Ad Hoc** button is highlighted. If you want to distribute your app over **Google Play** you need to change the package format to **bundle** in the project properties.

Create a Signing Identity



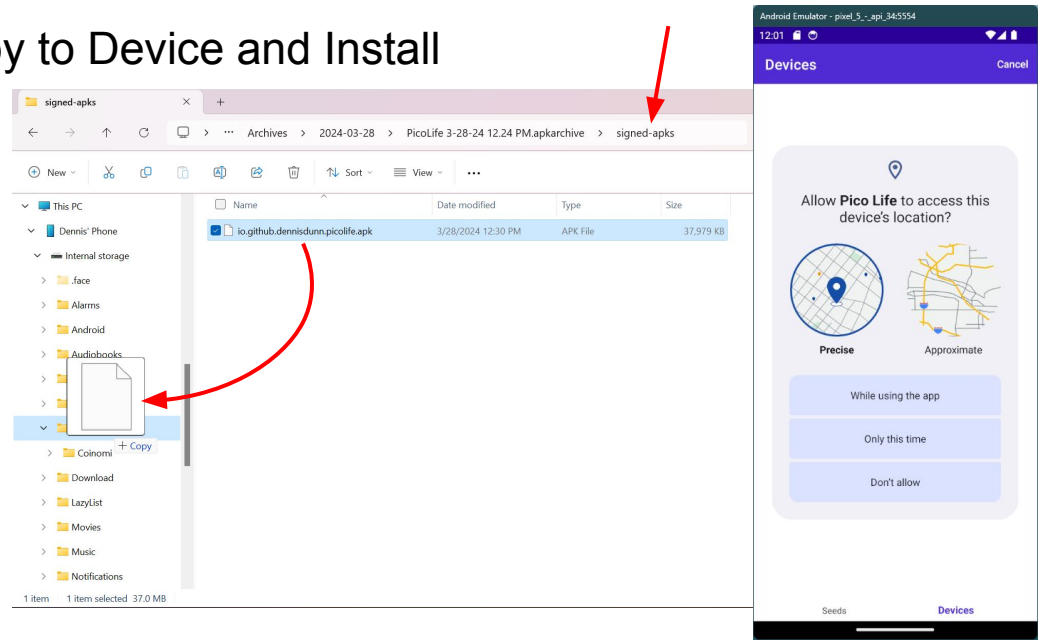
Finally, we need to sign the package. Create a signing identity by clicking the big **PLUS** button…

Sign the Package for Distribution



… and after saving the file we'll sign the package.

Copy to Device and Install



We're almost there!

- Copy the signed **apk** file to your mobile device.
- Navigate to the **Downloads** folder on your device and open the **apk** file.
    - If you get a "Bad Format…" error it just means that you are trying to install an unsigned package. Make sure that you copy the file from the **signed apks** folder.
- Once the app finishes installing, click the **Open** button to run the app.

We've covered a lot of ground this morning! We went from configuring a gadget by changing the source code to configuring the gadget with a mobile app.

By now you should have an idea of how .NET and the MAUI framework can be incorporated into your own projects. However, screenshots and gifs can only tell part of the story.

“The digital conveyor is more art than science.”

So is live coding!

As it turns out, the buttons on the configuration app that navigate to the edit screen are very small. More often than not, when tapping a button to edit a seed, we send the seed to the gadget instead.

Another problem is that we check for Bluetooth permissions when we start scanning. That’s not a problem itself but it is a little jarring the first time it happens because the assumption is that after you install the app it should be ready for use. Instead, let’s check for those permissions on first launch of the app. It’s a subtle difference but it is the kind that can make your app nicer for your users.

An offering to the Demo gods.

1. Change the button size to Medium.
2. Change ImageButton to Button.
3. Add BackgroundColor={StaticResource Primary} to the Button.
4. Debug to show the bigger buttons..
5. Click Edit and crash
6. Set breakpoint in SeedListPage EditSeedbuttonClicked
7. Click Edit
    a. Hit the breakpoint.
    b. Note that the cast is no invalid since we changed the button class.
    c. Change the cast in SeelListPage.cs EditSeedButtonClicked()
    d. Stop/Start the debugger
8. Click Edit

    a. Pause at breakpoint
    b. F5 to continue
    c. ???
    d. Profit!
2. Add permissions check to the SeedsList page.
3. Reset the emulator.
4. Debug to show desired behaviour.

Intro To MAUI For Makers

Dennis Dunn

ansofive@gmail.com

https://dennisdunn.github.io

TL;DR; Multi-platform development is easy with MAUI.

I hope that I've left you with enough knowledge and confidence that you'll try writing apps for your own gadgets. There is still much more in the MAUI eco-system. Some things we didn't cover are:

- Sharing resources across views with resource dictionaries
- Advanced binding paths
- Changing icons and colors
- Local data persistence

That being said; **Don't worry about it!** Just learn what you need to build apps for your gadgets.

Thanks for coming to my session. I'll see you around.

Intro To MAUI For Makers

Pico Life Gadget

- https://github.com/dennisdunn/PicoLifegadget
- https://en.wikipedia.org/wiki/Conway's_Game_of_Life
- https://github.com/monkmakes/mm_wlan
- https://github.com/miguelgrinberg/microdot
- https://github.com/micropython/micropython/blob/master/examples/bluetooth/

Pico Life App

- https://github.com/dennisdunn/PicoLifeApp
- https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui?view=net-maui-8.0
- https://learn.microsoft.com/en-us/dotnet/architecture/maui/
- https://learn.microsoft.com/en-us/dotnet/communitytoolkit/mvvm/

Android Development

- https://developer.android.com/guide

ACHTUNG!

Alles touristen und non-technischen looken peepers!

Das computermachine ist nicht fuer gefingerpoken und mittengrabben.

Ist easy schnappen der springenwerk, blowenfusen und poppencorken mit spitzensparken. Ist nicht fuer gewerken bei das dumpkopfen.

Das rubbernecken sichtseeren keepen das cotten-pickenen hans in das pockets muss; relaxen und watchen das blinkenlichten.