E-BOOK

BUILDING PCF CONTROLS FOR BEGINNERS

LEARN TO BUILD POWER APPS COMPONENT FRAMEWORK CONTROLS

CARL DE SOUZA

Version: 1.0

Introduction

Welcome to this PCF Controls Full Course for Beginners and congrats on the beginning of your PCF journey!

I'm Carl de Souza, a Microsoft developer and architect, and I created this course to help people learn how to build PCF controls quickly. PCF Controls, or Power Apps Component Framework Controls can be difficult to learn, and I hope the material here helps you to to get past the basics.

I want to give special thanks to the many people who inspired me to produce this course, many of whom are listed in the course below and in the accompanying video.

To use this book, you can follow each of the pages to build your own PCF controls. You can also see the video of this course on YouTube. It's 2.5 hours long and goes into a lot of detail. If you find it useful, please like the video and subscribe to my channel! It will help me to produce more content like this.

The video is located at https://youtu.be/897DPWMJQ20?feature=shared. My YouTube channel is located at https://www.youtube.com/@carldesouza. Finally, my blog can be found at https://carldesouza.com. You will find many more articles and videos on my YouTube channel and blog to help you learn the Microsoft Power Platform.

Hope you get some value out of this free course.

Thank you,
Carl de Souza

PCF Create, Build, Deploy Cheat Sheet

The following are steps to create, build, and deploy PCF controls. You will need to install the Power Platform CLI and have msbuild (from Visual Studio). Aliases are used where possible.

```
Usage: pac pcf init [--namespace] [--name] [--template] [--framework]
[--outputDirectory] [--run-npm-install]

  --namespace                   The namespace for the component. (alias: -ns)
  --name                        The name for the component. (alias: -n)
  --template                    Choose a template for the component. (alias: -t)
                                Values: field, dataset
  --framework                   The rendering framework for control. The default value
                                is 'none', which means HTML. (alias: -fw)
                                Values: none, react
  --outputDirectory             Output directory (alias: -o)
  --run-npm-install             Auto run 'npm install' after the control is created. The
                                default value is 'false'. (alias: -npm)
```

Initialize Project

How to initialize a project.

Field Templates

The following are **field** type templates.

Standard Field

Create Field (Standard template, non-React) without npm install (you would as an additional manual step):

```
pac pcf init -n <control name> -ns <namespace> -t field
```

<controlName> is used in the class in index.ts and in the constructor, display-name-key and description-key in the ControlManifest.input.xml.
<namespace> is used in the ControlManifest.input.xml.

Create Field (Standard template, non-React) with npm install:

```
pac pcf init -n <control name> -ns <namespace> -t field -npm
```

React (Virtual) Field

Create Field (React template) with npm install:

```
pac pcf init -n <control name> -ns <namespace> -t field -fw react -npm
```

Dataset Templates

The following are dataset templates.

Standard Dataset

Create Dataset (Standard template, non-React) with npm install:

```
pac pcf init -n <control name> -ns <namespace> -t dataset -npm
```

React (Virtual) Dataset

Create Dataset (React) with npm install:

```
pac pcf init -n <control name> -ns <namespace> -t dataset -fw react -npm
```

Build and Package

Build with npm:

```
npm run build
```

Test harness and debug with npm:

```
npm start watch
```

Increment the version in the ControlManifest.input.xml file if you are deploying via a solution.

Create a subdirectory called **Solution**, then go to that directory to run this command:

```
pac solution init --publisher-name developer --publisher-prefix dev
```

This creates a src subfolder in the solution directory and a Solution.cdsproj file. The src\Other directory has the files Customizations.xml, Relationships.xml, Solution.xml.

The publisher name is the name of the publisher that will be created (or used) by this command. The prefix provided will be used on the publisher and the control, e.g. if prefix is carlprefix_ then the control will be carlprefix_carlcontrolnamespace.carl_controlname.

Add reference to base (non-Solution) folder:

```
pac solution add-reference --path <path to base control folder>
```

This adds a “ProjectReference” back to ..\PCFBuildTest.pcfproj in the Solution.cdsproj file.

Run this also from Solution folder using the Visual Studio Developer command prompt:

```
msbuild /t:build /restore
```

This creates an **unmanaged** solution.zip file in the Solution\bin\Debug directory.

```
msbuild /p:configuration=release
```

This creates a **managed** solution (and smaller build) in the Solution\bin\release directory.

Deploy

If you have not created an authorization profile:

```
pac auth create -u <URL to Dynamics or Power Apps organization>
```

If you already have an authorization profile you can use below to view the list:

```
pac auth list
```

And below to select which org to deploy to:

```
pac auth select -i <index>
```

If pushing directly to an environment (without using a solution), run this from the base control directory (not Solution subfolder):

```
pac pcf push --publisher-prefix <publisher prefix>
```

If deploying a solution, be sure to increment the version in the ControlManifest.Input.xml file.

Folder Structure

Note the folder structure created above:

– Folder you created manually, e.g. C:\PCF\MyControl. This will contain config files such as package.json, package-lock.json, pcfconfig.json, tsconfig.json, .eslintrc.json. The .pcfproj file takes the name from this folder, e.g. MyControl.pcfproj
— Folder for **Control Name** specified above, e.g. if name is MyPCFControl, folder is called MyPCFControl. This can be a different name from the root folder above.

— node_modules, created as a result of **npm install**

— Solution, the directory you create to hold the solution. This gets populated when you run the solution commands, but that does not generate the solution.zip file.

— Solution\bin\Debug holds the unmanaged solution.zip file.

— Solution\bin\release holds the managed solution.zip file.

Errors

Error: Cannot find module ‘ajv/dist/compile/codegen’

```
npm install --save-dev ajv
```

Building and Deploying Your First PCF Control (without React)

PCF controls are custom built Power Apps Component Framework controls built by developers to extend the functionality of Microsoft Power Apps. In this post we will look at building these controls from scratch.

First, let's give a high-level overview of PCF controls.

Overview

One of the first questions that comes up is which frameworks can you use to build these controls. Currently there are 2 templates of controls, Standard and React.

There are several PCFs that you can download and install from the PCF Gallery website.

Tools You Will Need

1. Visual Studio Code with Node.js installed
2. Power Apps CLI (command line interface). Be sure to grab the latest with **pac install latest**. You can use the Windows version or the VS Code Extension.
3. msbuild (installed through Visual Studio)

Go ahead and follow the steps to install the above software on your machine, and we should be ready to go.

Setup

Now, let's start the setup of a control. First we will create a folder to hold our control. I created a folder located at C:\PCF\SampleControl:

Now we can initialize the control. In a command prompt, browse to the created directory. We will first explore the pac pcf commands available in the cli, which we can do with:

```
pac pcf help
```

This gives us the options for init, push and version:

```
C:\PCF\SampleControl>pac pcf help
Microsoft PowerPlatform CLI
Version: 1.29.10+g0c8cbfa
Online documentation: https://aka.ms/PowerPlatformCLI
Feedback, Suggestions, Issues: https://github.com/microsoft/powerplatform-build-tools/discussions

Help:
Commands for working with Power Apps component framework projects

Commands:
Usage: pac pcf [init] [push] [version]

  init                      Initializes a directory with a new Power Apps component framework project
  push                      Import the Power Apps component framework project into the current Dataverse organization
  version                   Patch version for controls
```

Let's explore **init** more as this is the command we will use to initialize our new control project. We can provide the following:

- namespace
- name
- template (field or dataset, more on this later)
- framework (none or react)
- output directory
- run-npm-install (auto run npm install)

```
C:\PCF\SampleControl>pac pcf init help
Microsoft PowerPlatform CLI
Version: 1.29.10+g0c8cbfa
Online documentation: https://aka.ms/PowerPlatformCLI
Feedback, Suggestions, Issues: https://github.com/microsoft/powerplatform-build-tools/discussions

Help:
Initializes a directory with a new Power Apps component framework project

Commands:
Usage: pac pcf init [--namespace] [--name] [--template] [--framework] [--outputDirectory] [--run-npm-install]

  --namespace               The namespace for the component. (alias: -ns)
  --name                    The name for the component. (alias: -n)
  --template                Choose a template for the component. (alias: -t)
                            Values: field, dataset
  --framework               The rendering framework for control. The default value is 'none', which means HTML. (alias: -fw)
                            Values: none, react
  --outputDirectory         Output directory (alias: -o)
  --run-npm-install         Auto run 'npm install' after the control is created. The default value is 'false'. (alias: -npm)
```

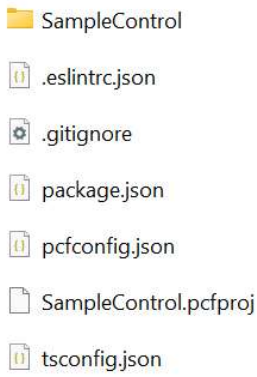
Let's create a control with the command:

```
pac pcf init --name SampleControl --namespace carl --template field
```

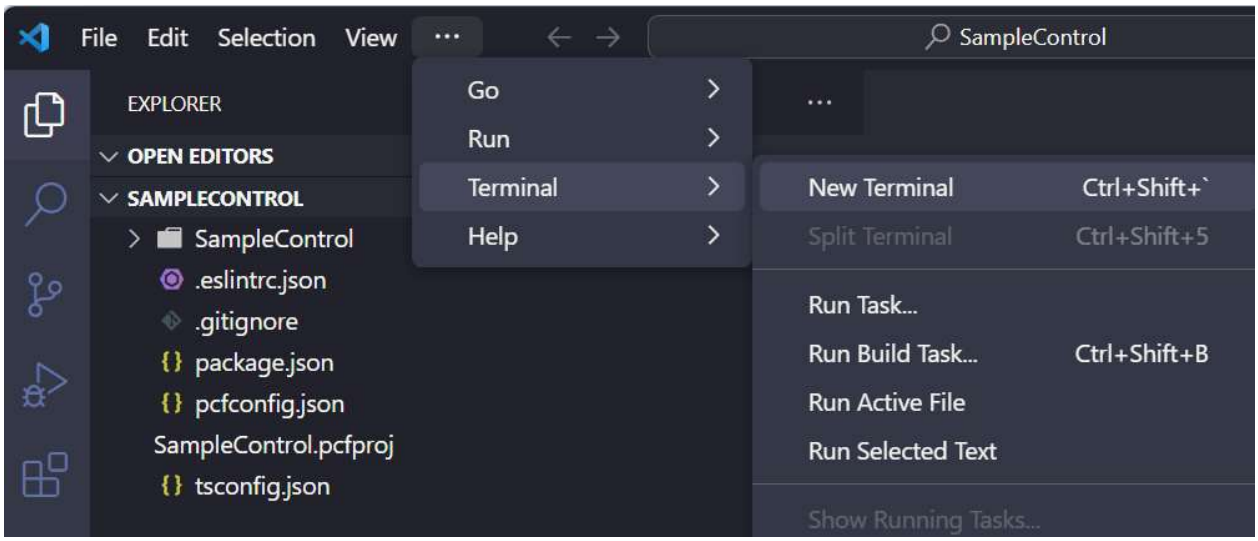
Note I am not using all the switches above, some will default and others are not required. Also note you can use the shortcuts to shorten the command (with a single dash instead of double), so the above command is the same as:

```
pac pcf init -n SampleControl -ns carl -t field
```

Once complete, we see the message to "Be sure to run 'npm install' or equivalent in this directory to install project dependencies." That will be our next step. Before we do that, let's take a look at the folder. We see the following files have been created.



Let's open the folder in VS Code, and let's open a new terminal:



And run:

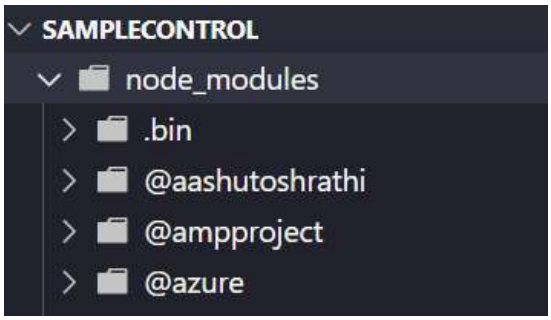
```
npm install
```

Note we could have automatically run this with the initiation of our project with the switch –run-npm-install.

This may take a few minutes to complete, and you may see different messages regarding modules:

```
PS C:\PCF\SampleControl> npm install
npm WARN deprecated @babel/plugin-proposal-nullish-coalescing-operator@
npm WARN deprecated @babel/plugin-proposal-object-rest-spread@7.20.7: T
er maintained. Please use @babel/plugin-transform-object-rest-spread ir

added 722 packages, and audited 723 packages in 1m

124 packages are looking for funding
  run `npm fund` for details

4 moderate severity vulnerabilities

To address all issues, run:
  npm audit fix
```

Once complete, we see this node_modules folder installed with the modules needed for this PCF control:



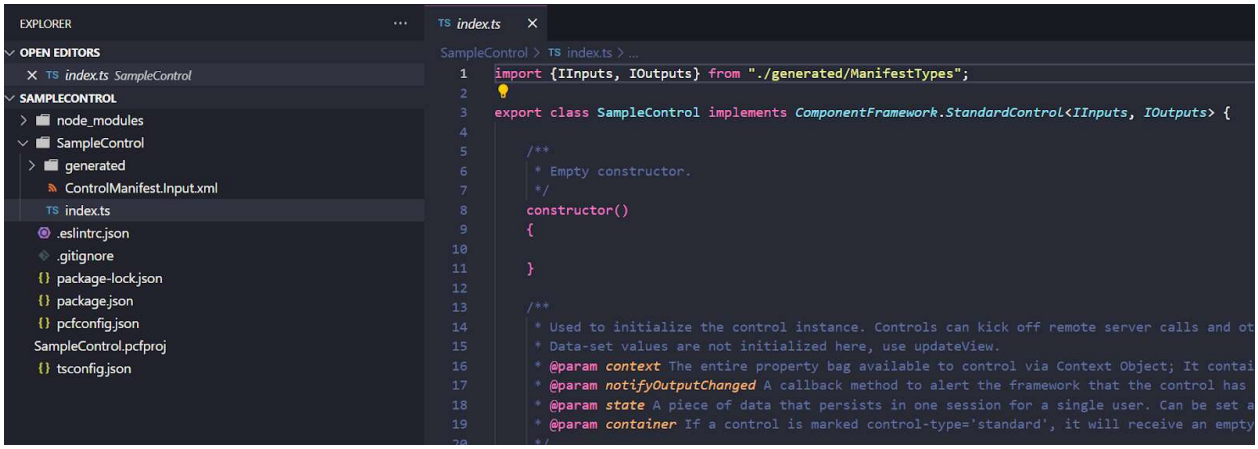
Now let's take a closer look at some of the files that have been generated.

Index.ts

Firstly, note we did not include the React flag above, so the files generated are without import React. There are 4 main functions in the file:

- init – used to initialize the component. You can do things here such as calling remote servers.
- updateView – called when any value in the property bag changes.
- getOutputs – called prior to a component receiving new data
- destroy – used for cleanup to release memory

The Index.ts file is the entry point into our app, let's look at some key areas:



First, notice what is imported. In our case we are not using React, so we just have one import line.

Another file of interest is the ControlManifestInput.xml. This file holds the properties for our control, and you can see there is one sample property already added called sampleProperty of type SingleLine.Text. Note you can see the types supported here. It is a “bound” type, meaning the component can change the bound field (other types are input, which is read-only, and output). You can also add resources to the manifest such as paths to css files that support your control.

Let’s try running this control in a test harness by typing into the console:

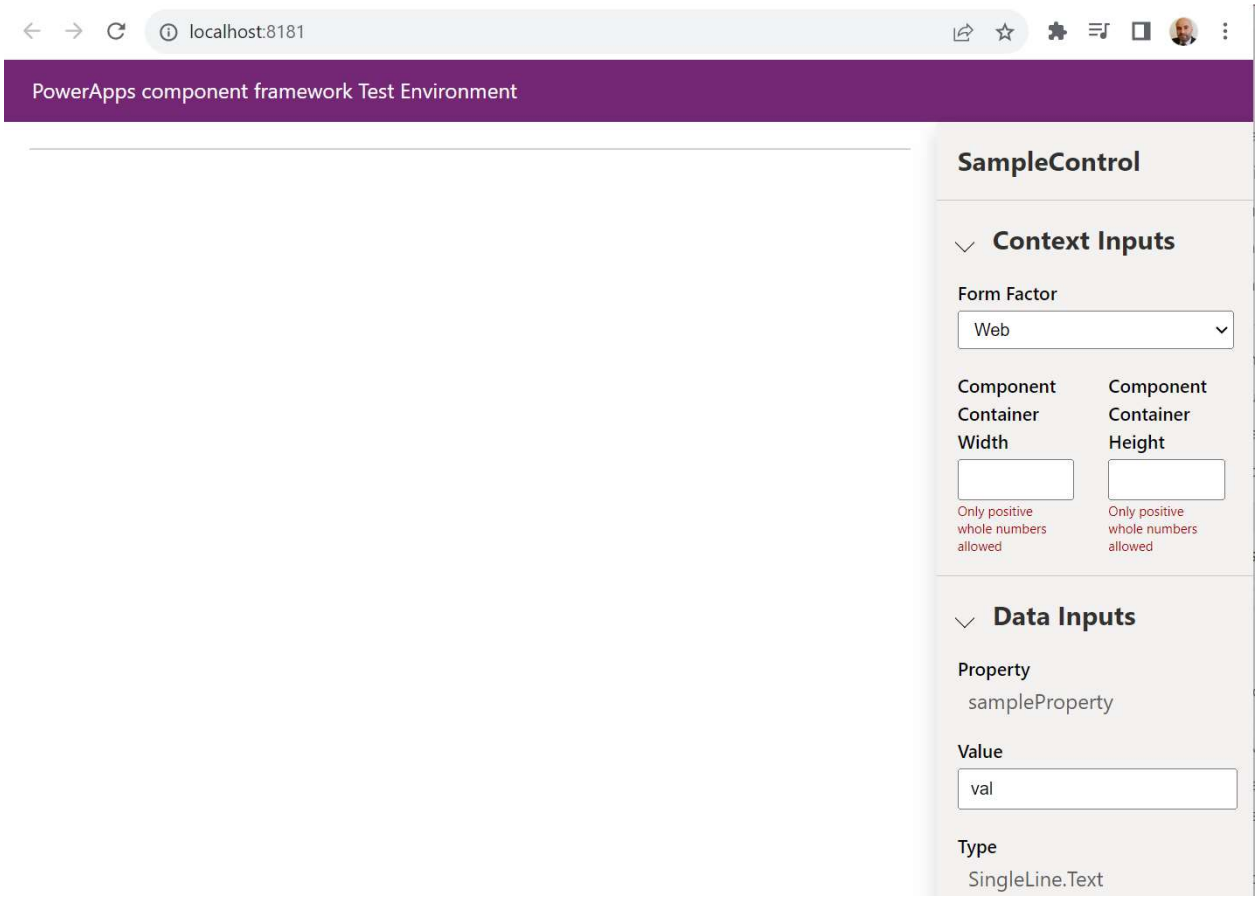
```
npm start
```

You can also use npm start watch which will look for on the fly changes:

```
npm start watch
```

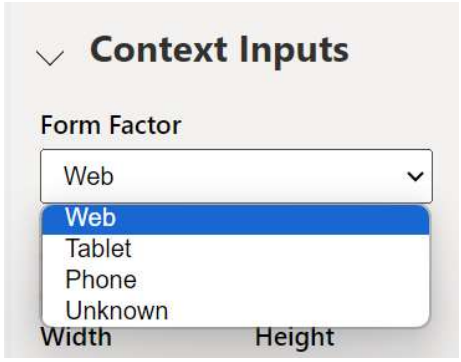
You may run into issues here with module errors, the best thing is to see which modules are causing errors and potentially install or update them.

You should now see a browser open at http://localhost:8181 with our sample control:

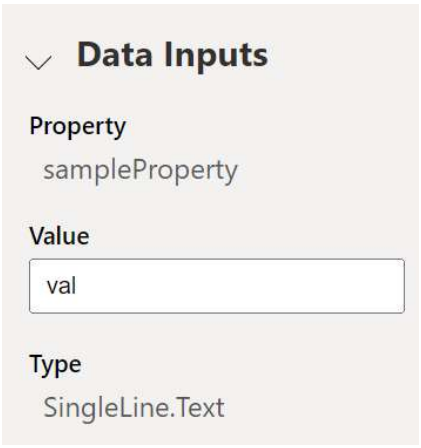


The form factor options are:

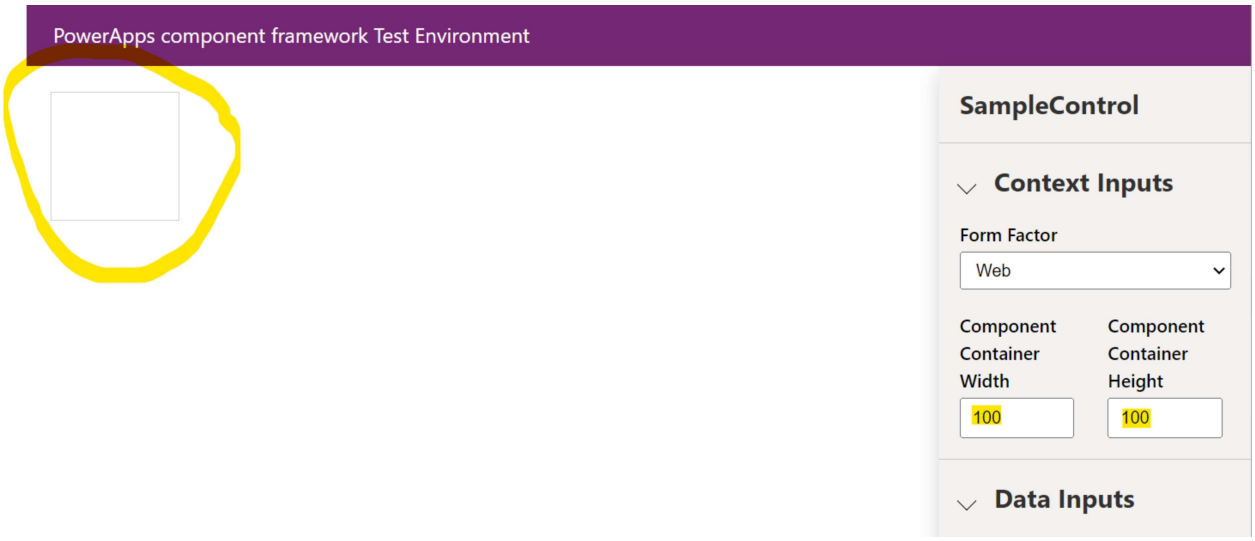
- Web
- Tablet
- Phone
- Unknown



And our sample property is listed below:



The next important thing here is the container. We can set the container size here, and you can see the container is redrawn to the size specified:



In our init function we have access to 4 parameters:

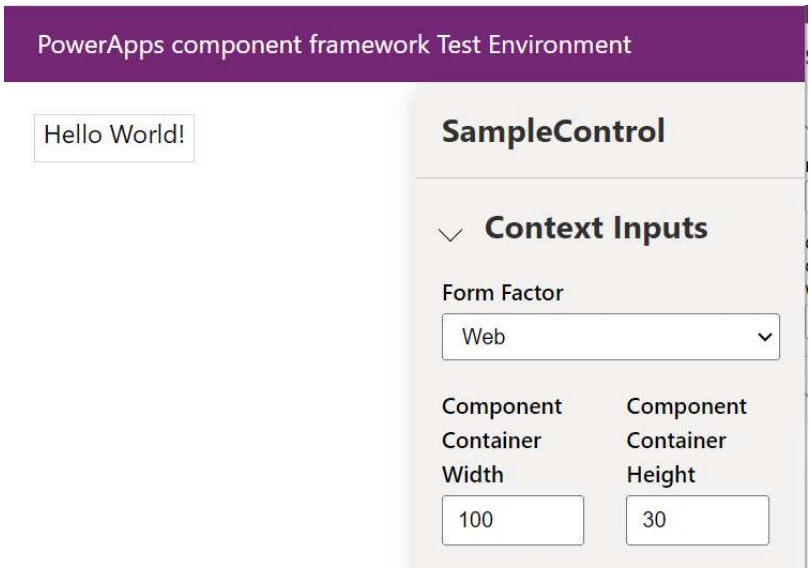
- context
- notifyOutputChanged
- state
- container

Let's add a simple line to our init function that appends to our container the text "Hello world":

```
container.appendChild(document.createTextNode("Hello World!"));
```

```
/**
 * Used to initialize the control instance. Controls can kick off remote server calls and other initialization actions here.
 * Data-set values are not initialized here, use updateView.
 * @param context The entire property bag available to control via Context Object; It contains values as set up by the customizer mapped to property names
 * @param notifyOutputChanged A callback method to alert the framework that the control has new outputs ready to be retrieved asynchronously.
 * @param state A piece of data that persists in one session for a single user. Can be set at any point in a controls life cycle by calling 'setControlSta
 * @param container If a control is marked control-type='standard', it will receive an empty div element within which it can render its content.
 */
public init(context: ComponentFramework.Context<IInputs>, notifyOutputChanged: () => void, state: ComponentFramework.Dictionary, container:HTMLDivElement)
{
    // Add control initialization code
    container.appendChild(document.createTextNode("Hello World!"));
}
```

On saving this, we see the change reflected in the control:



At this point, let's deploy the control to the Power Platform and see what it looks like. We will deploy this to a model-driven app.

The first thing to do is **create a subdirectory** called **Solution** or you will run into the error "Error: CDS project creation failed. The current directory already contains a project. Please create a new directory and retry the operation.". Create the subdirectory, then in the console go into that directory.

To package our component, we use the command below:

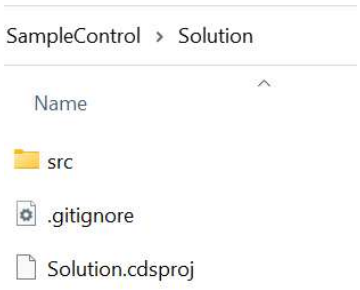
```
pac solution init --publisher-name developer --publisher-prefix dev
```

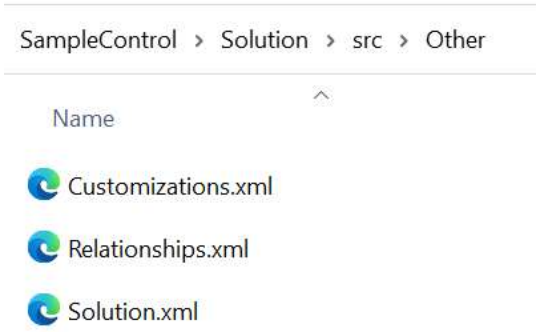
In my case, this will be:

```
pac solution init --publisher-name Carl --publisher-prefix carl
```

```
C:\PCF\SampleControl>cd Solution

C:\PCF\SampleControl\Solution>pac solution init --publisher-name Carl --publisher-prefix carl
Dataverse solution project with name 'Solution' created successfully in: 'C:\PCF\SampleControl\Solution'
Dataverse solution files were successfully created for this project in the sub-directory Other, using solution name Solution, publisher name Carl, and customization prefix ca
rl.
Please verify the publisher information and solution name found in the Solution.xml file.
```

The following files are created:





Next, we need to add references to where our component is located, in our case:

```
pac solution add-reference --path C:\PCF\SampleControl
```

We get “Project reference successfully added to Dataverse solution project”:

```
C:\PCF\SampleControl\Solution>pac solution add-reference --path C:\PCF\SampleControl
Project reference successfully added to Dataverse solution project.
```

Now we need to run msbuild to generate the actual solution zip file. If you run this in a command prompt and it comes back with “‘msbuild’ is not recognized as an internal or external command” you may need to run this from the **Developer Command Prompt for VS**, which can be located in the Start Menu:

Run the command below in the Solution directory:

```
msbuild /t:build /restore
```

```
C:\PCF\SampleControl\Solution>msbuild /t:build /restore
MSBuild version 17.6.3+07e294721 for .NET Framework
Build started 11/30/2023 9:27:26 PM.

Project "C:\PCF\SampleControl\Solution\Solution.cdsproj" on node 1 (Restore target(s)).
_GetAllRestoreProjectPathItems:
  Determining projects to restore...
Restore:
  X.509 certificate chain validation will use the default trust store selected by .NET for code signing.
  X.509 certificate chain validation will use the default trust store selected by .NET for timestamping.
  Restoring packages for C:\PCF\SampleControl\SampleControl.pcfproj...
  Restoring packages for C:\PCF\SampleControl\Solution\Solution.cdsproj...
```

Once complete, you should see “Build Succeeded” with no errors or warnings, and the end result is a Solution.zip file:

We can now connect to our org and deploy the file. The pac cli can do this for us. The first thing to do is create a connection using pac auth create and then the URL to the org:

```
pac auth create --url https://org19307bf5.crm.dynamics.com/
```

We get a login prompt, then the connection is created:

```
C:\PCF\SampleControl\Solution>pac auth create --url https://org19307bf5.crm.dynamics.c
'admin@CRM274130.onmicrosoft.com' authenticated successfully.
Validating connection...
Default organization: Customer Service Trial
Connected to... Customer Service Trial
Connected as admin@CRM274130.onmicrosoft.com
Authentication profile created
    * DATAVERSE                                https://org19307bf5.crm.dynamics.com/
m          Public
```

If you already have a connection set up, you can use pac auth list and pac auth select to connect to the org.

Go to the Component folder again, in my case the SampleComponent directory, and run the command:

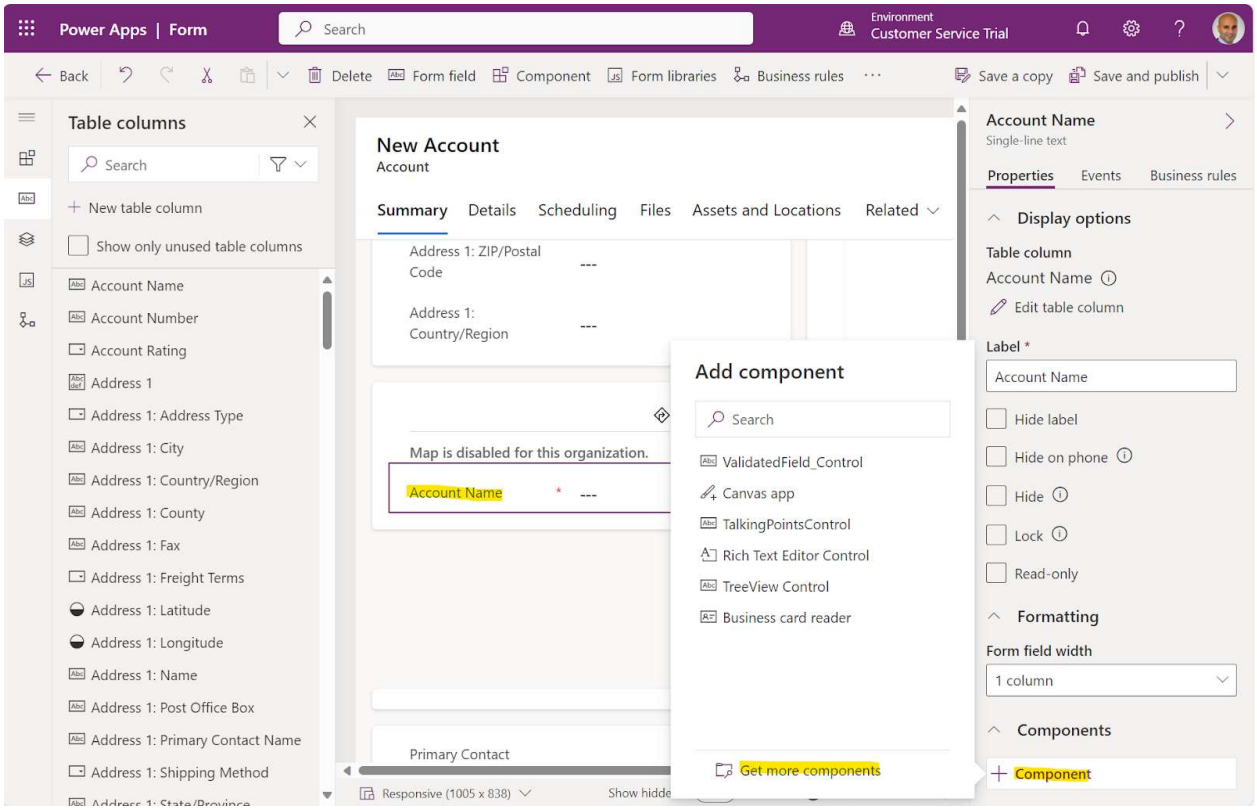
```
pac pcf push --publisher-prefix <publisher prefix>
```

```
C:\PCF\SampleControl>pac pcf push --publisher-prefix carl
Connected to... Customer Service Trial
Connected as admin@CRM274130.onmicrosoft.com
Using publisher prefix 'carl'.
Checking if the control 'carl_carl.SampleControl' already exists in the current org.
Using full update.
Creating a temporary solution wrapper to push the component.
Building the temporary solution wrapper.

  Determining projects to restore...
  Restored C:\PCF\SampleControl\obj\PowerAppsToolsTemp_carl\PowerAppsToolsTemp_carl.c
  1 of 2 projects are up-to-date for restore.

  > pcf-project@1.0.0 clean
  > pcf-scripts clean --noColor --buildMode development --outDir C:\PCF\SampleControl

  [9:39:14 PM] [clean]  Initializing...
  [9:39:14 PM] [clean]  Cleaning build outputs...
  [9:39:14 PM] [clean]  Succeeded
  Cleaning output directory: bin\Debug\, Intermediate directory: obj\Debug\ and Solut
\Debug\
  Removing log file: obj\Debug\SolutionPackager.log and generated solution package: b
p

  > pcf-project@1.0.0 build
  > pcf-scripts build --noColor --buildMode development --outDir C:\PCF\SampleControl
```

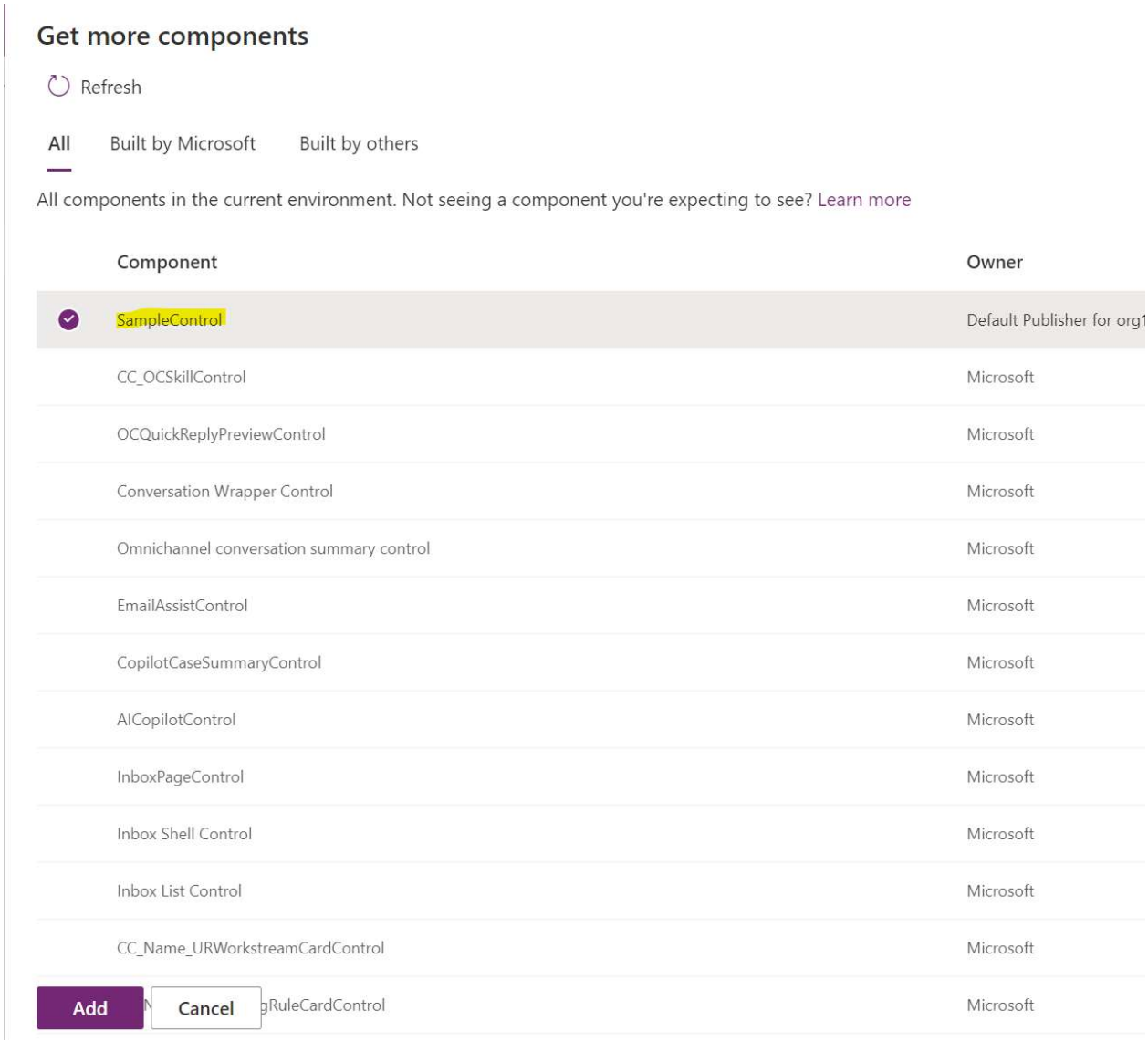
This will take a few minutes to run, and finally you will see:

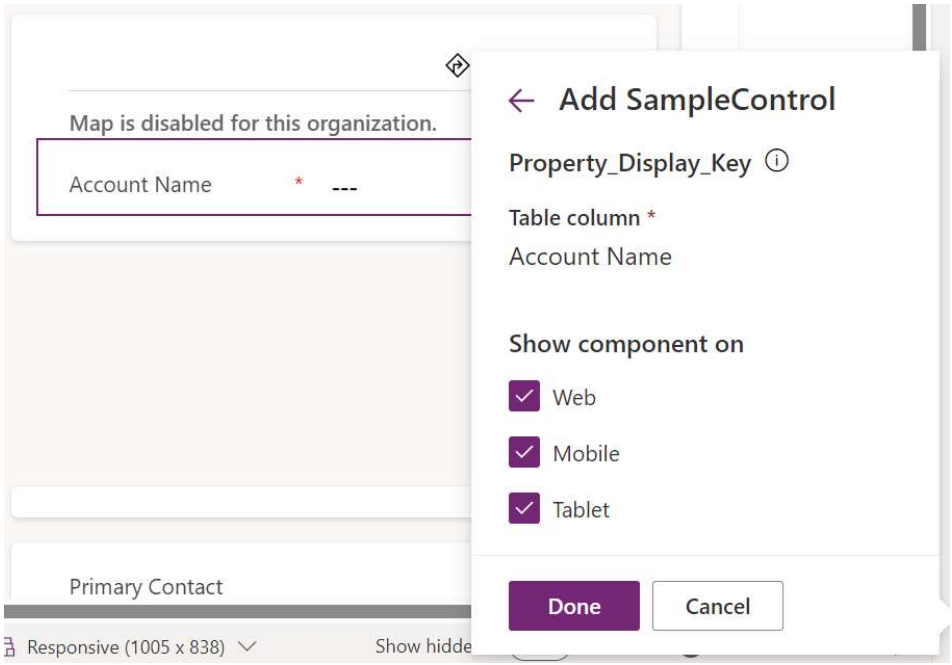
```
Published All Customizations.
Updating the control in the current org: done.
```

In our Power Apps maker portal located at https://make.powerapps.com let's add our component to a form. This part might be slightly unintuitive depending on what your PCF control does, but basically we are adding a field to our form and changing the component of the field to be our new component. Click on Component->Get New Components:



We see our SampleControl and can Add it:

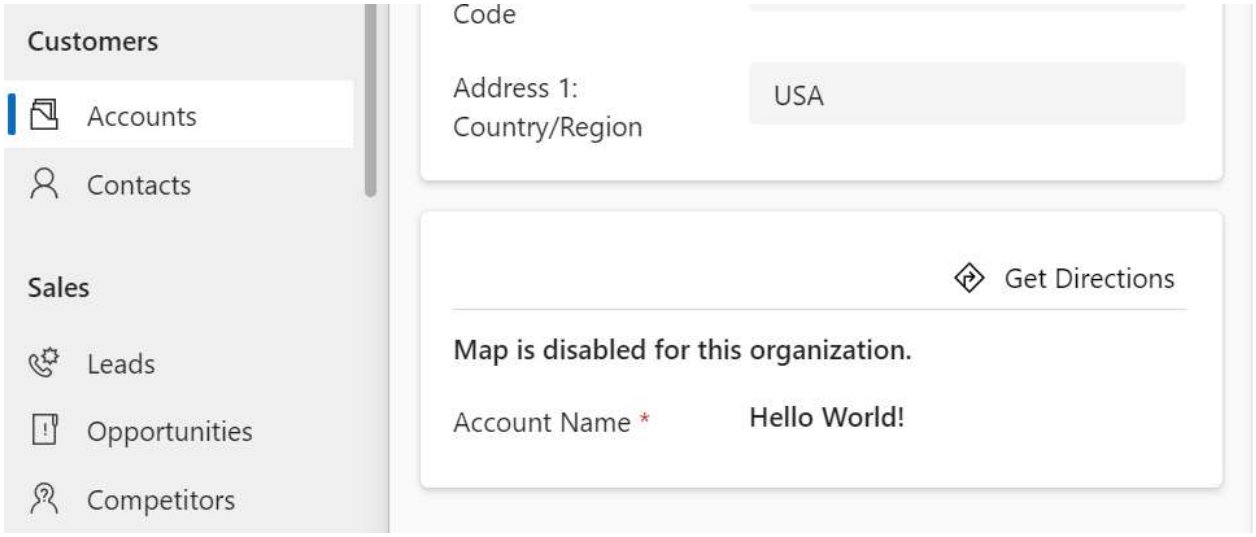

Get more components
Refresh
All
Built by Microsoft
Built by others
All components in the current environment. Not seeing a component you're expecting to see? Learn more
Component
Owner
SampleControl
Default Publisher for org1
CC_OCSkillControl
Microsoft
OCQuickReplyPreviewControl
Microsoft
Conversation Wrapper Control
Microsoft
Omnichannel conversation summary control
Microsoft
EmailAssistControl
Microsoft
CopilotCaseSummaryControl
Microsoft
AICopilotControl
Microsoft
InboxPageControl
Microsoft
Inbox Shell Control
Microsoft
Inbox List Control
Microsoft
CC_Name_URWorkstreamCardControl
Microsoft
Add
Cancel
gRuleCardControl
Microsoft



Our control is now rendering with the words “Hello World!”:

As I mentioned, depending on what your control does, you may want to hide the label of the field. If your control is related to the field, e.g. it’s a date calendar lookup control, you would want to keep the field label such as “Date”. If the PCF control is more stand-alone, such as an iFrame, you may want to hide the label.

After saving and publishing, we see on our Account form our new field. Note the field is a label and not an input type, i.e. we can’t change the value of the field. Let’s look at how to do that next.


Customers
Accounts
Contacts
Sales
Leads
Opportunities
Competitors
Code
Address 1:
Country/Region
USA
Get Directions
Map is disabled for this organization.
Account Name *
Hello World!

Creating a TextBox PCF Control

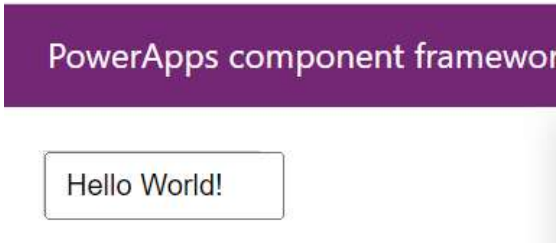
Previously, we created a simple PCF control that displayed an HTML label. Let's look at how to make the control a text box.

A very basic example of adding an HTML DOM input textbox can be found here. The code we will add to the init function in our index.ts file:

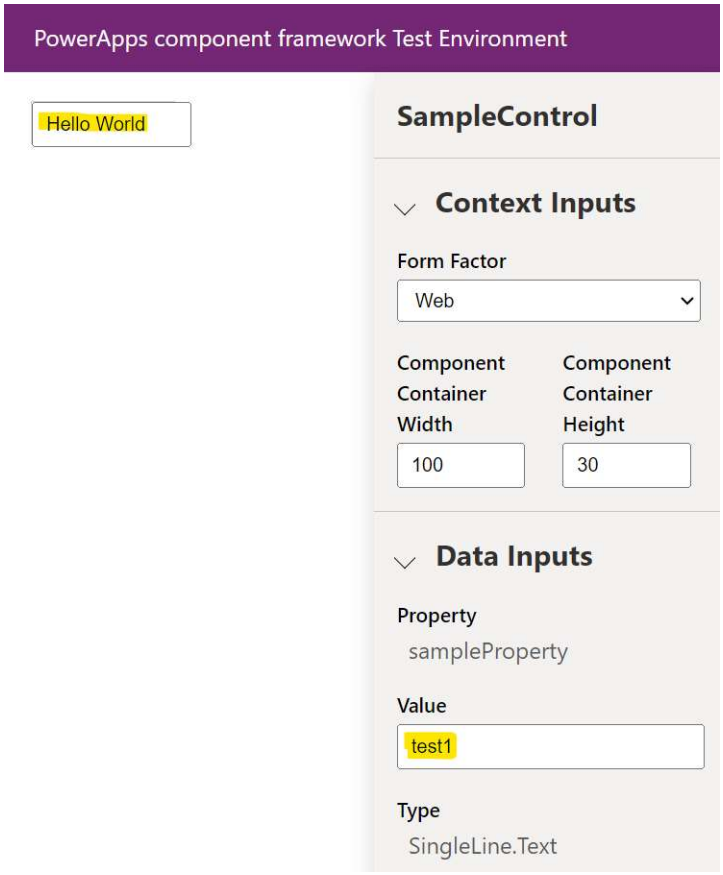
```
var x = document.createElement("INPUT");
x.setAttribute("type", "text");
x.setAttribute("value", "Hello World!");
container.appendChild(x);
```

```
public init(context: ComponentFramework.Context<I
{
    var x = document.createElement("INPUT");
    x.setAttribute("type", "text");
    x.setAttribute("value", "Hello World!");
    container.appendChild(x);
}
```

On saving, our test harness is now updated with a textbox control:



However there is a disconnect between what is in the textbox and what is the value. Changing either doesn't change the other:



Firstly, let's add some code to set the value on init. From our SampleProperty bound property that is currently in our manifest, we can access the value through:

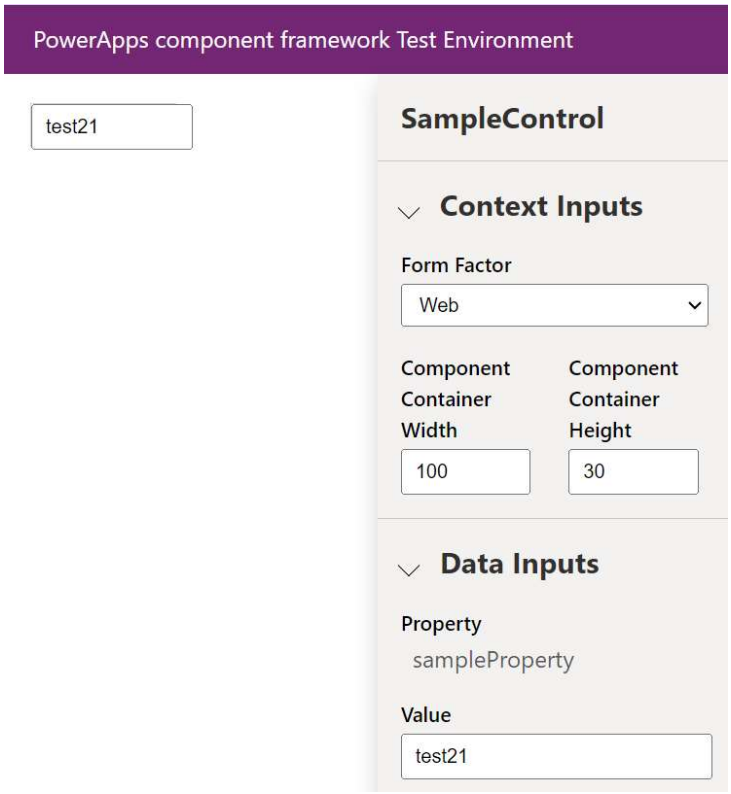
`context.parameters.sampleProperty.raw`

```
public init(context: ComponentFramework.Context<IInputs>, notifyOutputChange
{
    var x = document.createElement("INPUT");
    x.setAttribute("type", "text");
    x.setAttribute("value", context.parameters.sampleProperty.raw || "");
    container.appendChild(x);
}
```

Let's set the value of our textbox to this property:

`x.setAttribute("value", context.parameters.sampleProperty.raw || "");`

On refreshing the test harness, we see the value that I enter in the sampleProperty on the right is reflected in our textbox on the left:

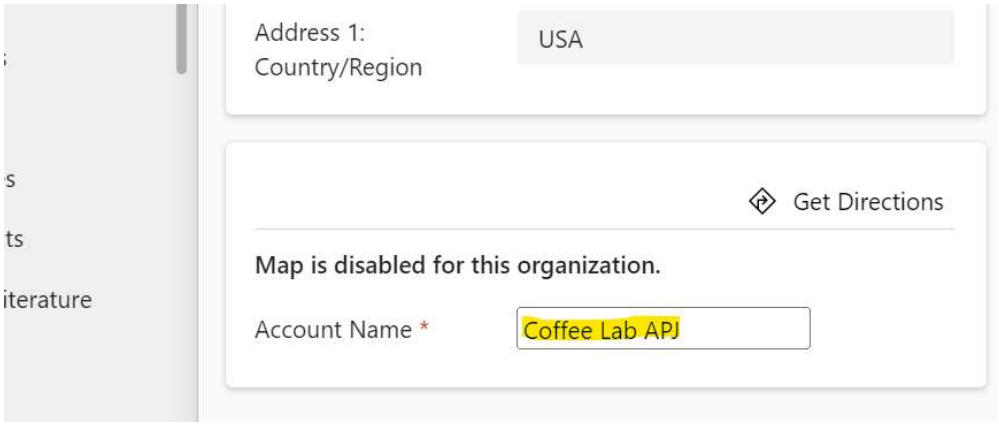


However, if I change either textbox on the right or left, the other textbox does not change. More on that shortly.

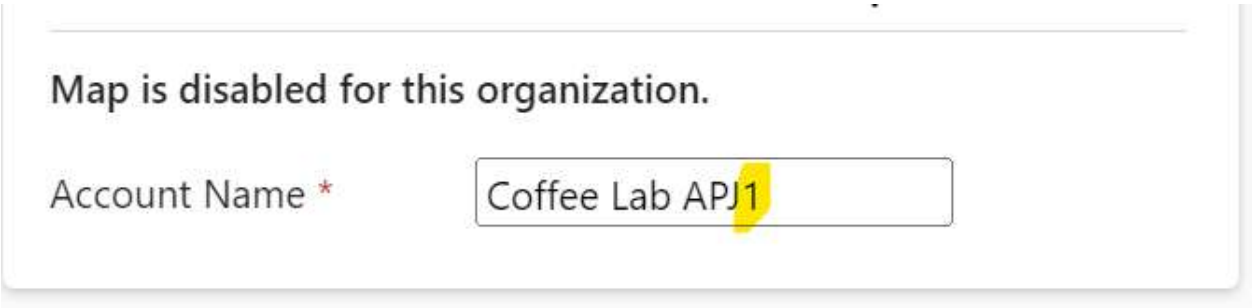
Let's push this control up to the Power Apps environment and see what it looks like. We're going to use the same command as before from the SampleControl folder.

```
pac pcf push --publisher-prefix carl
```

Now if we look in our org, we can see the field is a textbox, and the value is actually defaulting to the account name, i.e. the field our SampleProperty is bound to:



Now if we try to change the value of the field and save the record, nothing happens and the record doesn't save. In fact, the page doesn't even register that the form is dirty.



To resolve this, let's go back to VS Code and take a look at some of the other methods. The one we're interested in is updateView, which is "called when any value in the property bag has changed. This includes field values, data-sets, global values". So we want to update our HTML element here, but we will need to change our init code a little to make our properties accessible from these other methods we're calling.

So instead of our code looking like this:

```
public init(context: ComponentFramework.Context<IInputs>, notifyOutputChanged:
{
    var x = document.createElement("INPUT");
    x.setAttribute("type", "text");
    x.setAttribute("value", context.parameters.sampleProperty.raw || "");
    container.appendChild(x);
}
```

We will create a new private member:

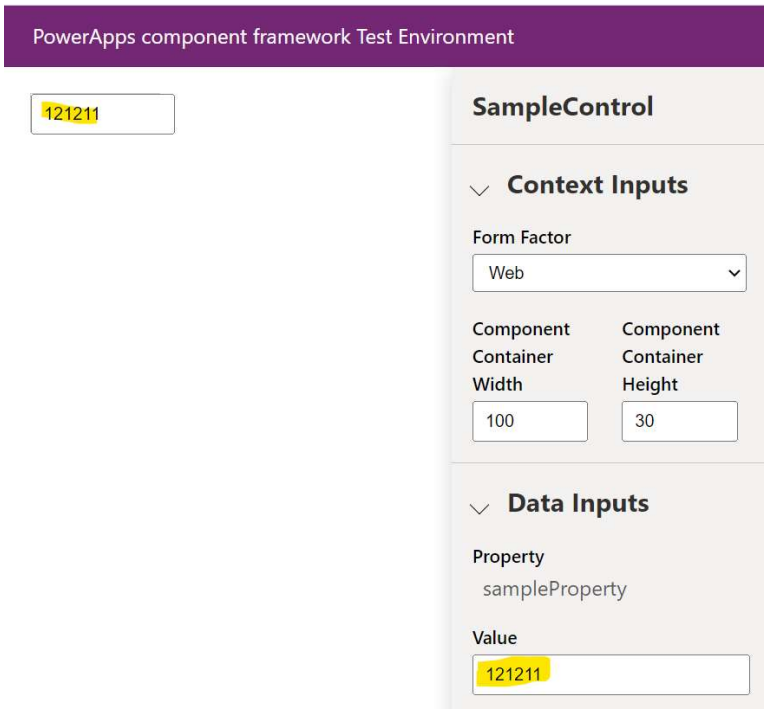
```
private _element: HTMLInputElement;
```

And new code for init to use this member:

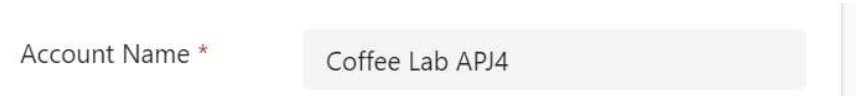
```
this._element = document.createElement("INPUT") as HTMLInputElement;
this._element.setAttribute("type", "text");
this._element.setAttribute("value", context.parameters.sampleProperty.raw ||
"");
container.appendChild(this._element);
```

```
private _element: HTMLInputElement;

public init(context: ComponentFramework.Context<IInputs>, notifyOutputChanged: () => void, s
{
    this._element = document.createElement("INPUT") as HTMLInputElement;
    this._element.setAttribute("type", "text");
    this._element.setAttribute("value", context.parameters.sampleProperty.raw || "");
    container.appendChild(this._element);
}
```

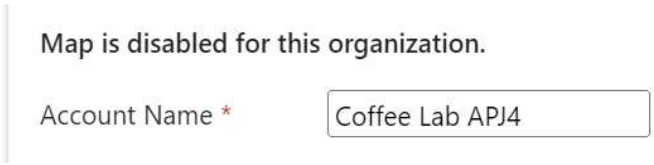
Let's save and test this. Now as I type on the right, the control on the left is automatically updating, and without having to press enter:



Let's push this up to our org and observe what's happening. We have 2 fields on the Account form that are the Account Name, one is the original field with its out of the box control, and the other is the same field bound to our custom PCF control. If we change the out of the box field and tab off it:



We see our custom PCF is updated:



This is due to the updateView functionality we added.

However, if we update our PCF control's value we still don't get a form dirty updated value of the field. Let's now add code to do that using notifyOutputChanged(). This is a method to tell the system that a change has been made to our control, and it's up to us to invoke it. In this example, let's add a change event to the textbox control, and when it fires we will call our notifyOutputChanged().

We will add this to our init:

```
this._element.addEventListener("change", (event) => {
    notifyOutputChanged();
});
```

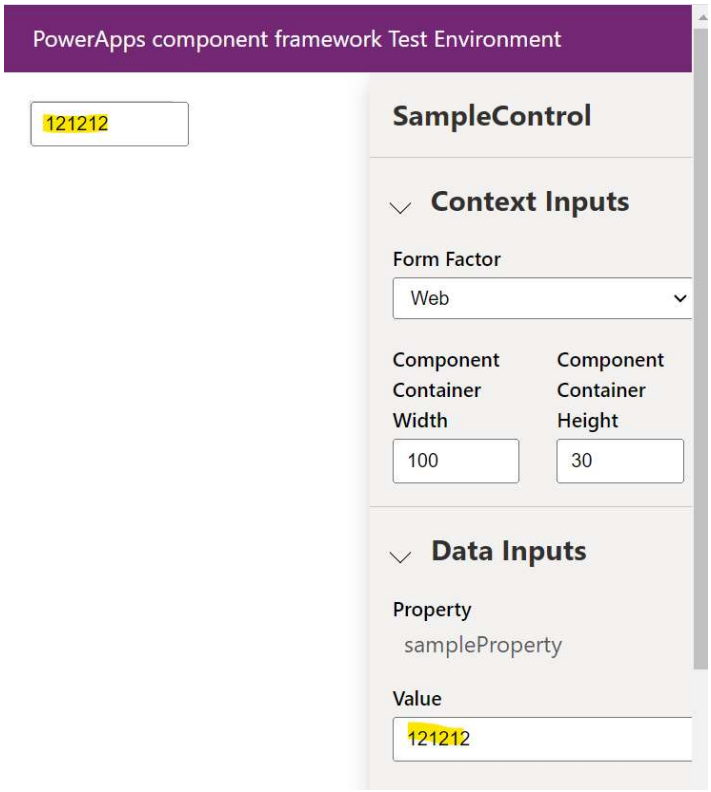
```
public init(context: ComponentFramework.Context<IInputs>, notifyOutputChanged: () => voi
{
    this._element = document.createElement("INPUT") as HTMLInputElement;
    this._element.setAttribute("type", "text");
    this._element.setAttribute("value", context.parameters.sampleProperty.raw || "");
    container.appendChild(this._element);

    this._element.addEventListener("change", (event) => {
        notifyOutputChanged();
      });
}
```

Now when the text in our control changes and the user tabs off the control, our notifyOutputChanged() runs, and this in turn natively invokes the getOutputs() function. In this function we want to set our SampleProperty to be the value of our element:

```
sampleProperty: this._element.value
```

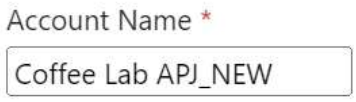
```
public getOutputs(): IOutputs
{
    return {
        sampleProperty: this._element.value
    };
}
```

Now when we make a change to the value of our control on the left and tab off the field, our value gets updated on the right:

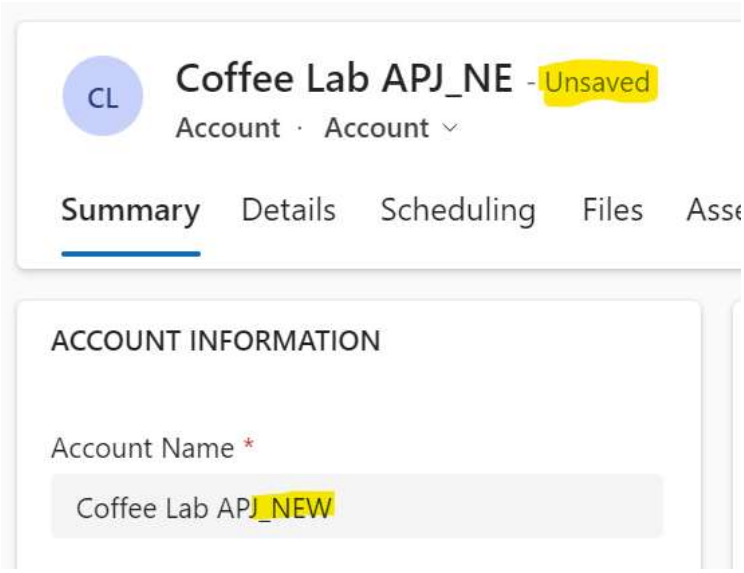


Let's publish this latest control to our org and see how it looks.

When we change the value in our PCF control, we see now the out of the box Account Name field also changes its value, and the form becomes dirty. Our control:



Form is now dirty with our change, out of the box field updated:



This is a simple example of how these pieces fit together.

Finally, instead of updating the value when the user tabs off the control, we can do it when any text changes on the control. For this we would use “input” instead of change:

```
this._element.addEventListener("input", (event) => {
    notifyOutputChanged();
});
```

That’s it! You can now build a text PCF control.

Using Context Navigation and User Settings in PCF Controls

We will now look at how to use context in PCF Controls. The context provides us with many features such as being able to call Power Apps and Dynamics 365 native functionality that we may normally access through JavaScript and the API. Being able to do this from within a PCF control is very useful for building our controls.

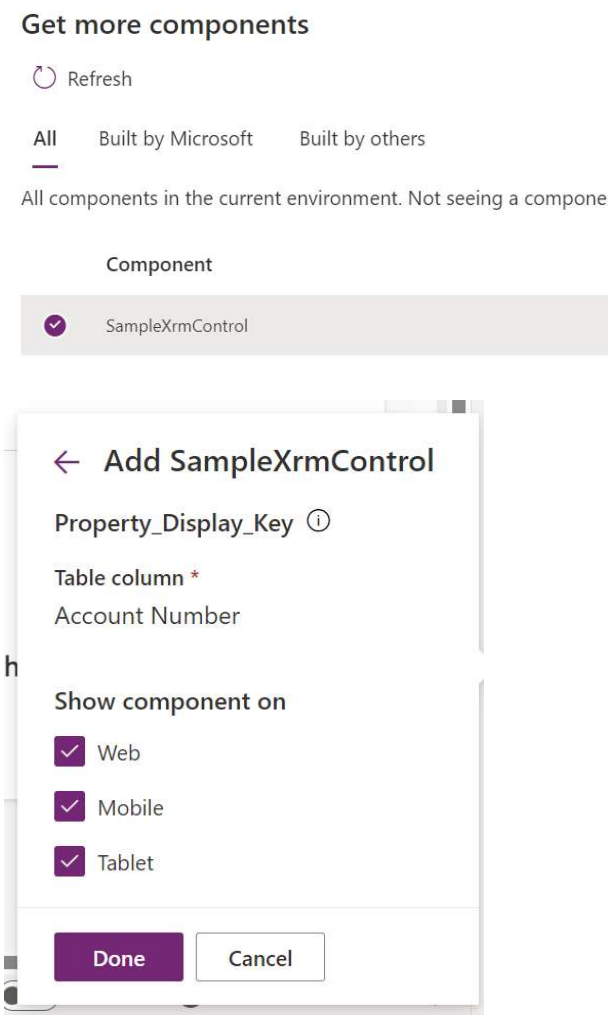
First, let's create a new PCF control called SampleXrmControl, we will use the standard template (not the React template) for this example:

```
pac pcf init -n SampleXrmControl -ns carl -t field
```

Let's add some simple code to display a button in the init function. This HTMLButtonElement button will call an onclick function when clicked. The function calls a standard JavaScript alert, and then it calls the context that we get with PCF controls. The context has navigation capabilities, and these have openAlertDialog, which we use when building Power apps and Dynamics apps. Let's open a dialog this way:

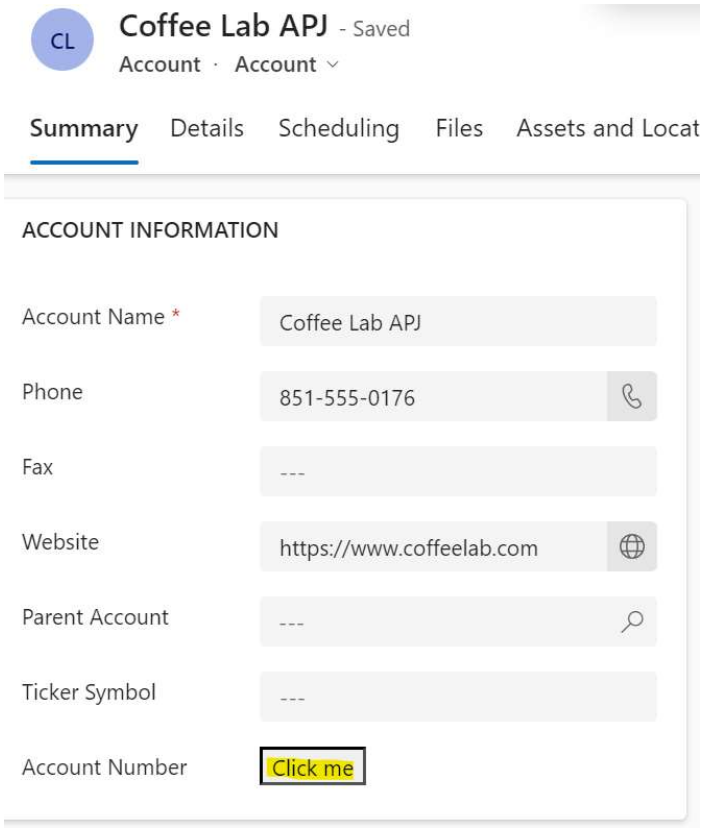
```
const button: HTMLButtonElement = document.createElement("button");
button.innerHTML = "Click me";
button.onclick = () => {
      alert("Hello World");
      var alertStrings = { confirmButtonLabel: "Yes", text: "This is an
alert.", title: "Sample title" };
      var alertOptions = { height: 120, width: 260 };
      context.navigation.openAlertDialog(alertStrings, alertOptions).then(
          function (success) {
              console.log("Alert dialog closed");
          },
          function (error) {
              console.log(error.message);
          });
  }
container.appendChild(button);
```

That's all we need to do. We can publish this to our org using the same steps as previously described in this series. Once published, add a field to a form and go to Components to change it to our new PCF component. Here we will bind it to the Account Number field on the Account form, and we can select our SampleXrmControl:



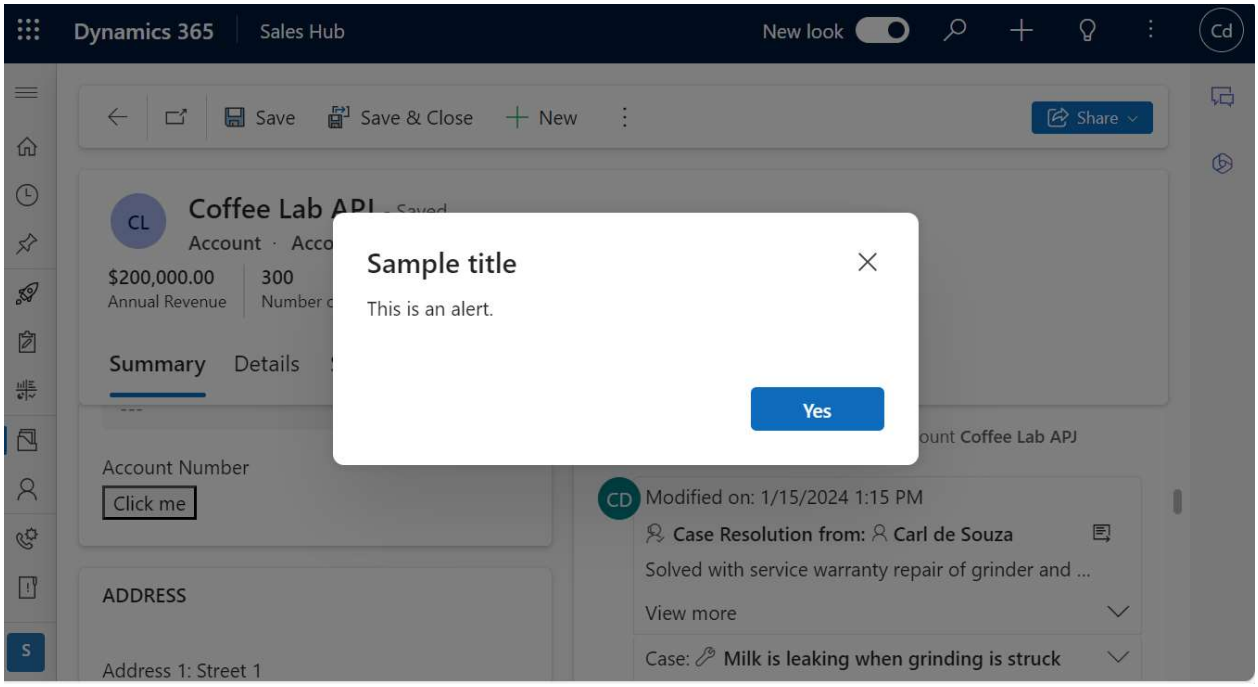
Once added we see the control rendered with the Click Me button:

Let's Save and Publish so we can see it on a record. We now see:



Clicking the button, the standard JS alert:

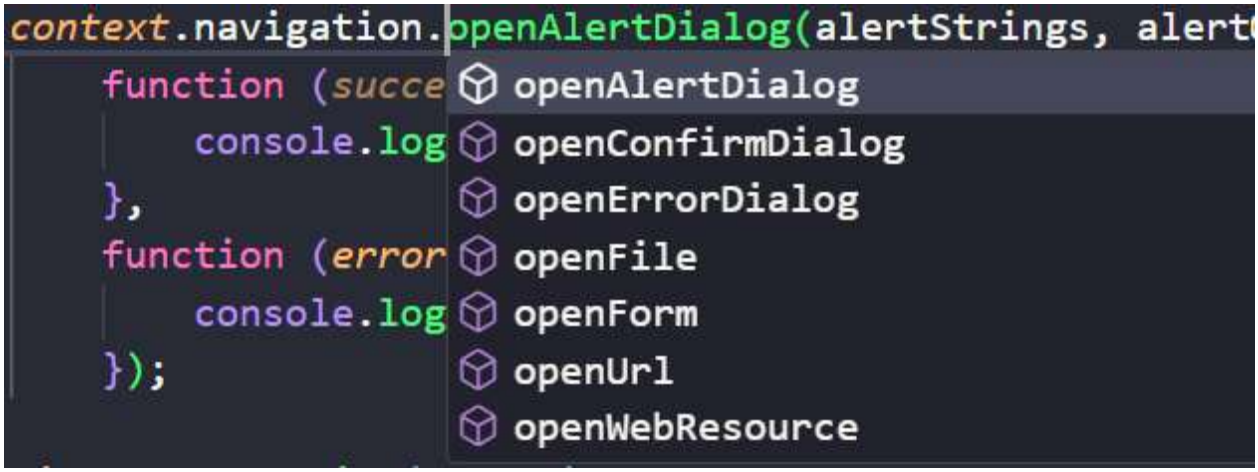
And the Dynamics 365 alert dialog is displayed:



And clicking Yes closes the alert dialog and logs to the console:

```
Alert dialog closed
```

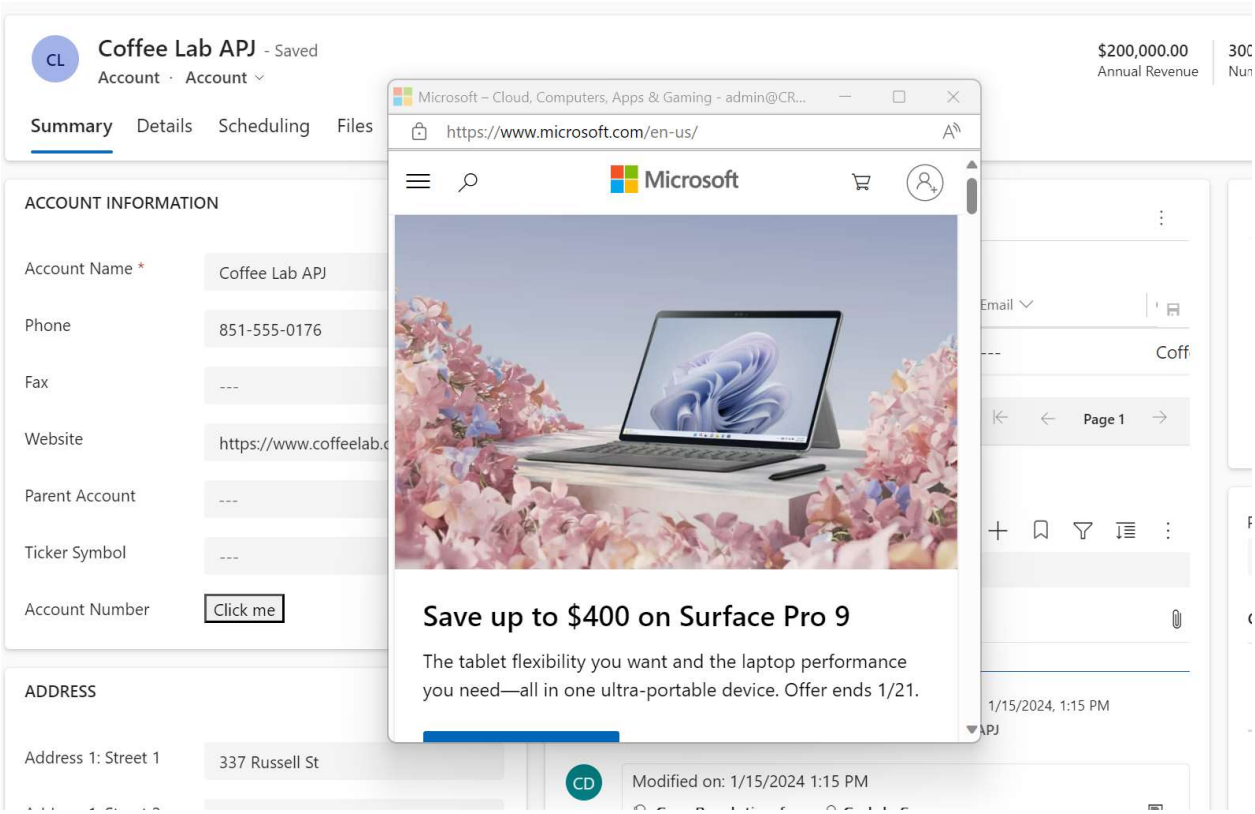
Let's look at some of the other functionality we get with context.navigation. We have openConfirmDialog, openErrorDialog, openFile, openForm, openUrl, and openWebResource:



Let's add some code to do the openUrl:

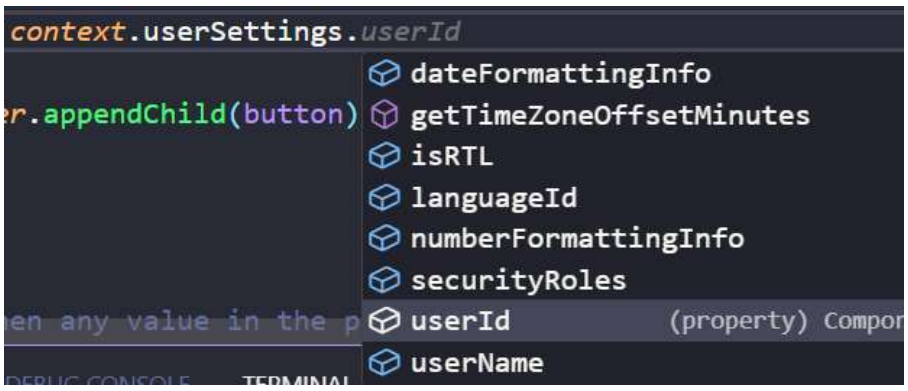
```
var openUrlOptions = { height: 500, width: 500 };
context.navigation.openUrl("https://microsoft.com",openUrlOptions);
```

We now see a browser window open to the dimensions and URL specified:



Let's now take a look at the **userSettings** context. We have the following:

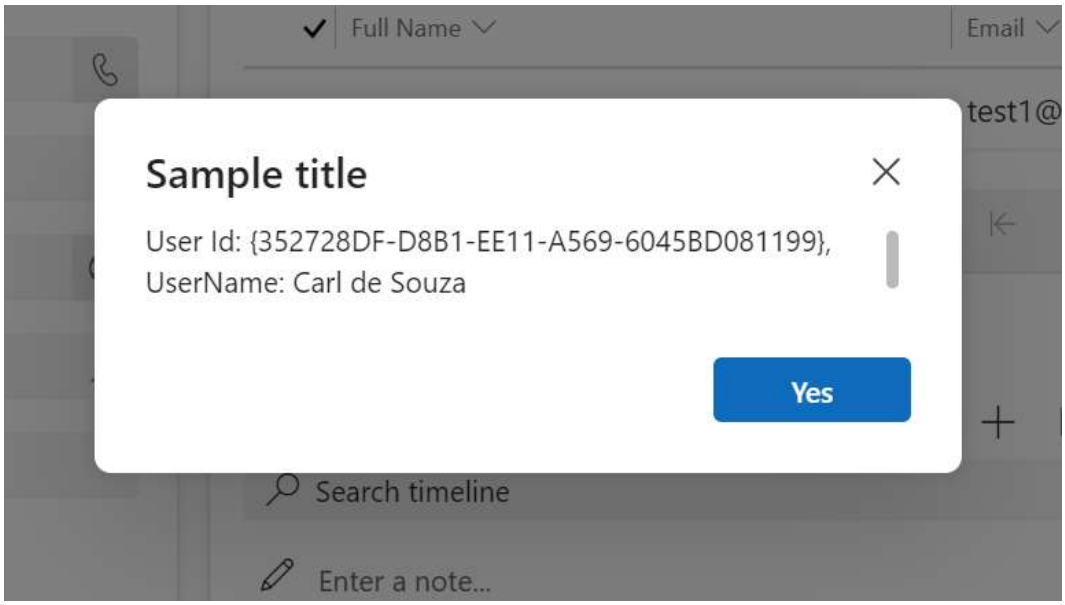
- dateFormattingInfo
- getTimeZoneOffsetMinutes
- isRTL
- languageId
- numberFormattingInfo
- securityRoles
- userId
- userName



Let's try the userId and userName, printing it on the screen using the previously used openAlertDialog:

```
var DisplayText = `User Id: ${context.userSettings.userId}, UserName:
${context.userSettings.userName}`;
var alertStrings = { confirmButtonLabel: "Yes", text: DisplayText, title:
"Sample title" };
var alertOptions = { height: 120, width: 260 };
context.navigation.openAlertDialog(alertStrings, alertOptions).then(
    function (success) {
        console.log("Alert dialog closed");
    },
    function (error) {
        console.log(error.message);
    });
```

On running this, we see the User Id and User Name displayed:



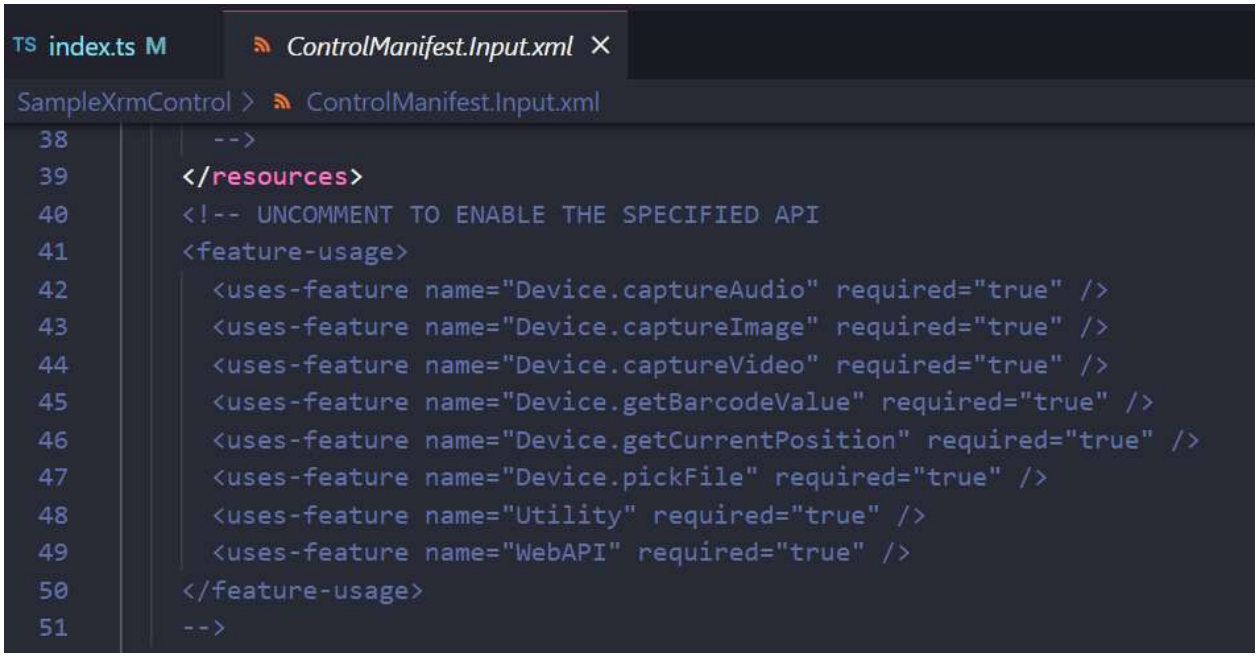
Building a PCF Control that Calls Context WebApi

Power Apps Component Framework, or PCF Controls, have capabilities to call WebApi operations. Let's look at how to build a PCF control that calls the Context WebApi. Previously, we built a control that called the navigation features in a PCF control, as well as user settings. Let's use that as a base to run some WebApi code.

In looking at the context.webAPI, we see we have the following methods:

- createRecord
- deleteRecord
- retrieveMultipleRecords
- retrieveRecord
- updateRecord

Before we can use the WebAPI, we need to uncomment the feature-usage area in the ControlManifest.Input.xml file. It will look like below before you uncomment it:



We will only include the <uses-feature name=”WebAPI” required=”true” /> line:

Note without enabling this feature you may run into a “Feature is required to be specified in the <uses-feature> section in ControlManifest” error.

Let’s add some code. We will the following code to our index.ts, which will create a new account record:

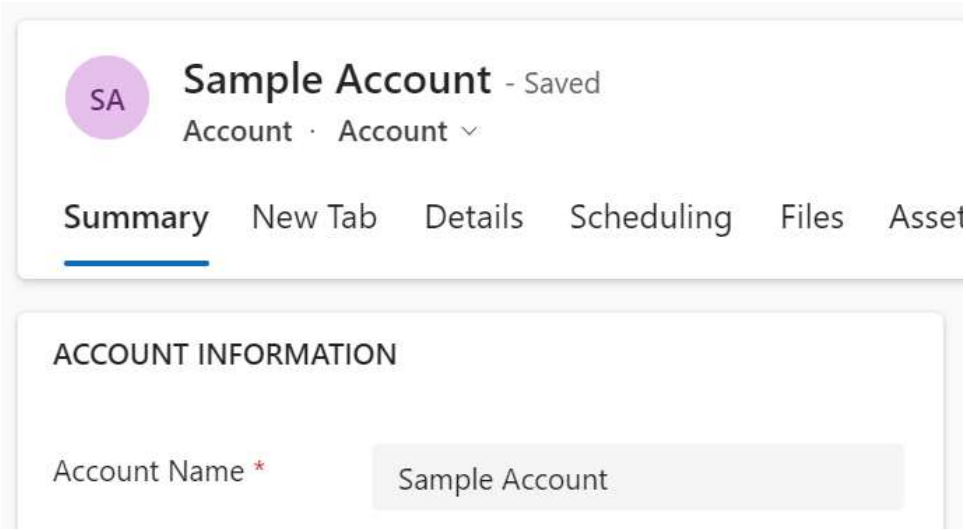
```
context.webAPI.createRecord("account", { name: "Sample Account" }).then(
    function (success) {
        console.log("Account created with ID: " + success.id);
    },
    function (error) {
        console.log(error.message);
    }
);
```

```
public init(context: ComponentFramework.Context<IInputs>, notifyOutputChanged: () => void,
{
    // Add control initialization code
    const button: HTMLButtonElement = document.createElement("button");
    button.innerHTML = "Click me";
    button.onclick = () => {
        // Create an account record with the name "Sample Account"
        context.webAPI.createRecord("account", { name: "Sample Account" }).then(
            function (success) {
                console.log("Account created with ID: " + success.id);
            },
            function (error) {
                console.log(error.message);
            }
        );
    }
    container.appendChild(button);
```

Now when we deploy this to our org and run this by pressing our button, we get “Account created with Id” in our console:


```
Account created with ID: 860476ad-58ba-ee11-a569-000d3a3b9c87
```

And our sample account is created:



Our webApi is working nicely! Let's try the other methods.

To delete an account:

```
var Id = "860476ad-58ba-ee11-a569-000d3a3b9c87";
context.webAPI.deleteRecord("account", Id).then(() => {
    alert("Account deleted");
}).catch((error) => {
    alert(error.message);
});
```

This deletes the account:

To retrieveMultiple accounts:

```
context.webAPI.retrieveMultipleRecords("account", "?$select=name").then(
    function success(result) {
        // perform operations on on retrieved records
        console.log("Retrieved " + result.entities.length + " records");
        for (var i = 0; i < result.entities.length; i++) {
            console.log(result.entities[i]["name"]);
        }},
    function (error) {
        console.log(error.message);
        // handle error conditions
    });
```

This returns:

```
Retrieved 4 records
Trey Research
Coffee Lab APJ
Fourth Coffee
Graphic Design Institute
```

To retrieve a single account:

```
var Id = "da59721f-02b2-ea11-a812-000d3a1b14a2";
context.webAPI.retrieveRecord("account", Id, "?$select=name").then(
    function success(result) {
        alert("Account Name: " + result["name"]);
    },
    function (error) {
        alert(error.message);
    }
);
```

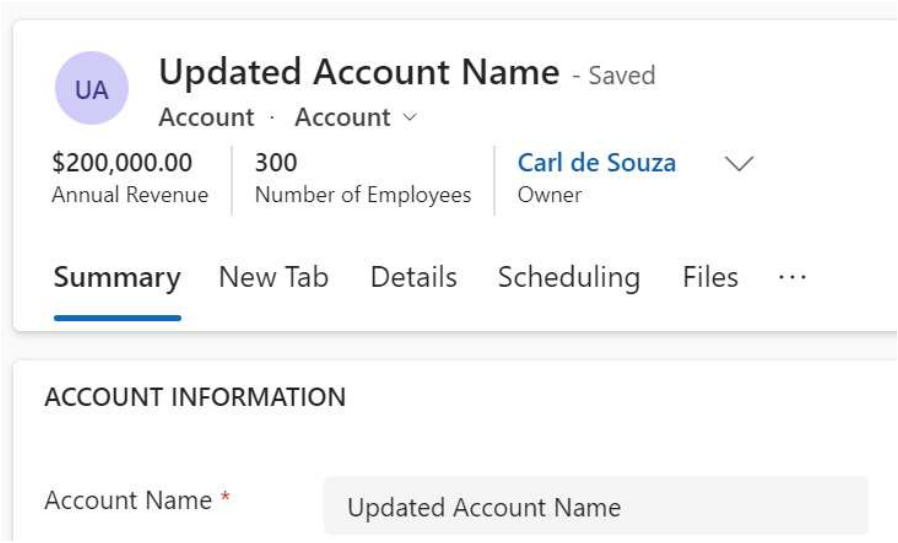
This returns:

And to update an account:

```
var Id = "da59721f-02b2-ea11-a812-000d3a1b14a2";
context.webAPI.updateRecord("account", Id, { name: "Updated Account Name"
}).then(
    function success(result) {
        console.log("Account name updated.");
    },
    function (error) {
        console.log(error.message);
    }
);
```

This produces an updated account record:

```
Account name updated.
```



Those are the WebApi methods, you are now able to perform these in your PCF controls.

Creating a Virtual PCF React Control with Fluent UI v8

Previously, we looked at how to create a sample PCF control. In this post, we will look at creating a React PCF control.

The current templates for PCF Controls use **Fluent UI v8**.

For v8 documentation, see https://developer.microsoft.com/en-us/fluentui#/controls/web . v8 controls use @fluentui/react.

The latest version is **Fluent UI v9**. We can use v9 controls in our PCF controls, but it won't use the out of the box template.

For v9 documentation, see https://react.fluentui.dev/. v9 controls use @fluentui/react-components.

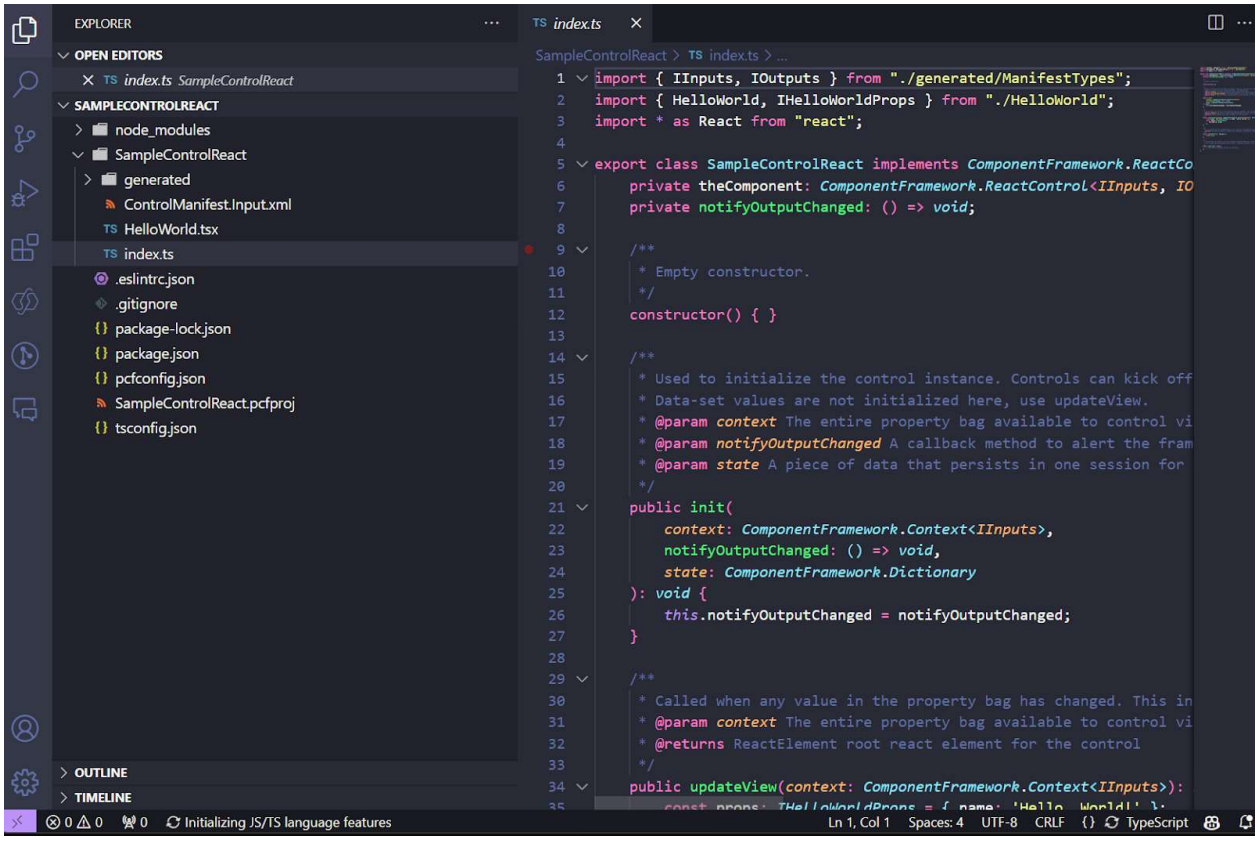
Let's use a v8 component.

First, ensure all your software is installed as we went through earlier. Let's create a folder called SampleControlReact and run the command below with the react switch:

```
pac pcf init -n SampleControlReact -ns carl -t field -fw react
```

```
C:\PCF\SampleControlReact>pac pcf init -n SampleControlReact -ns carl -t field -fw react
The Power Apps component framework project was successfully created in 'C:\PCF\SampleControlReact'

Be sure to run 'npm install' or equivalent in this directory to install project dependencies.
```

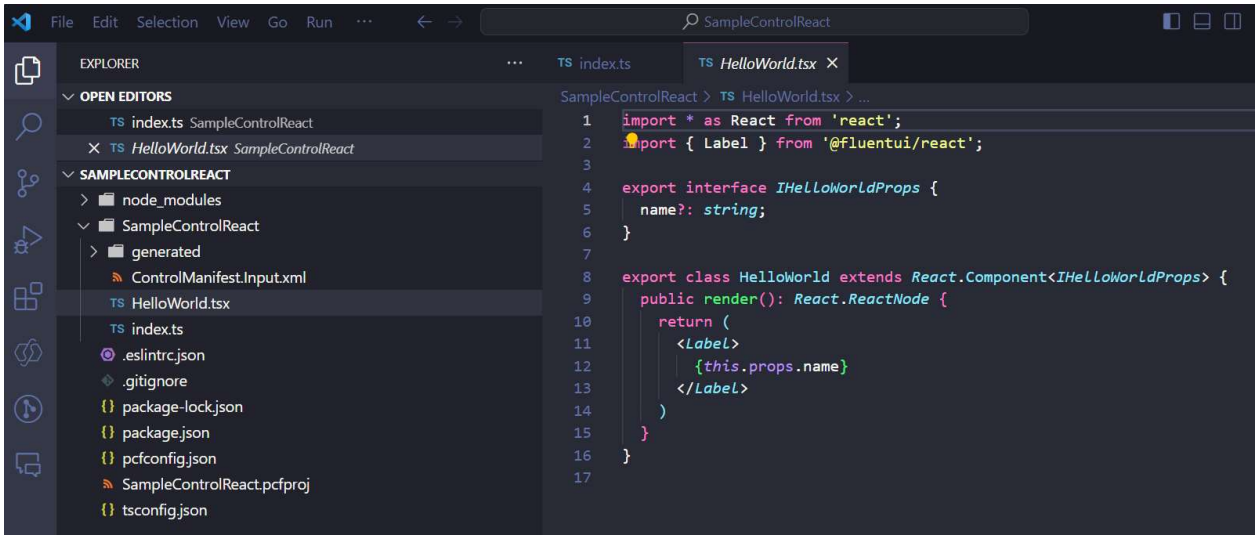
Then run **npm install**, open the folder in VS Code and navigate to the index.ts file:



We can see at the top of this file we're automatically importing react, which is different from the non-react sample component we previously made. Along with the constructor, we have 4 main functions:

- init – used to initialize the component. You can do things here such as calling remote servers.
- updateView – called when any value in the property bag changes.
- getOutputs – called prior to a component receiving new data
- destroy – used for cleanup to release memory

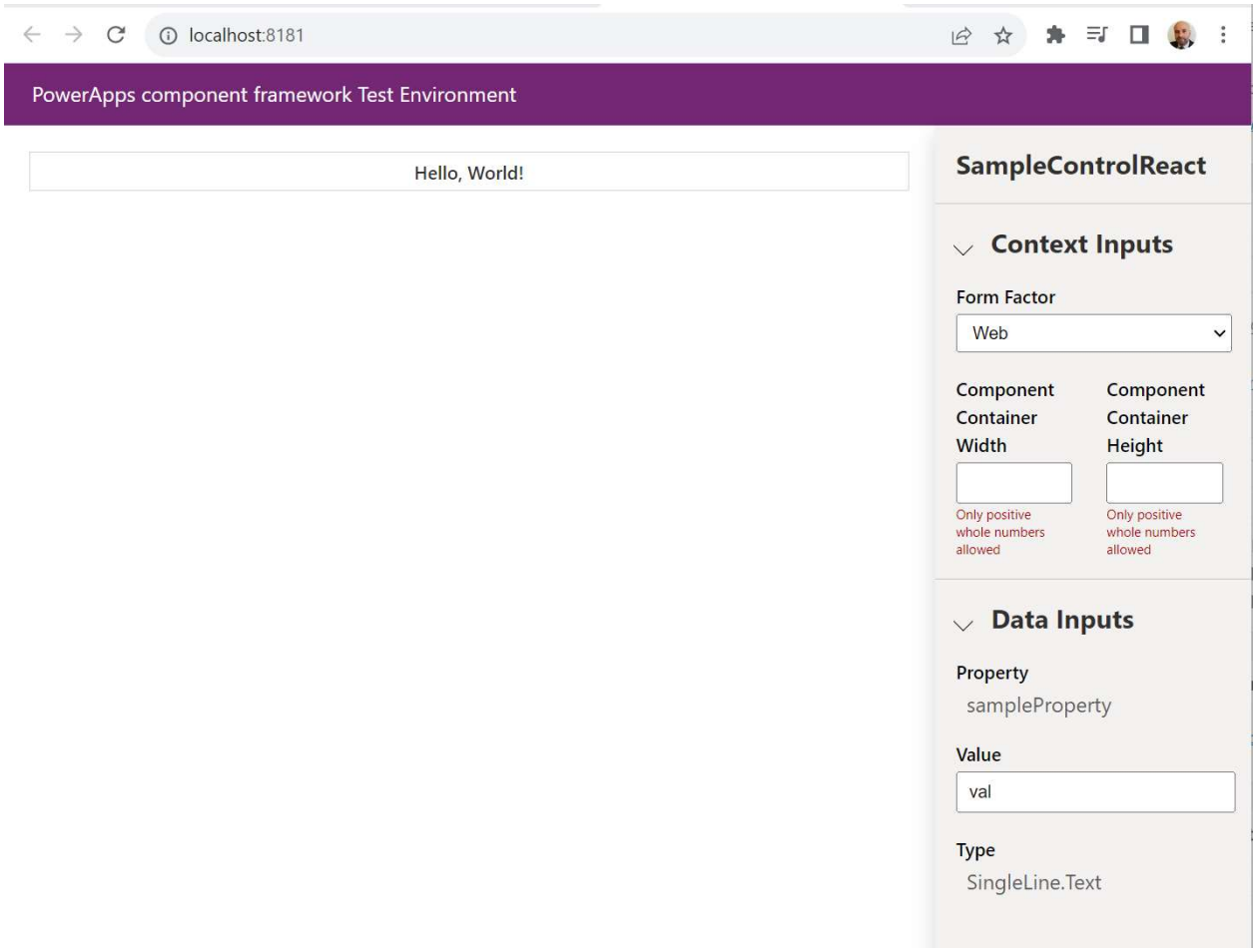
Also, we have a HelloWorld.tsx file generated. We can see that the template here returns a **label** field, setting the value of the label to the name that we pass in through props (which are properties in React):



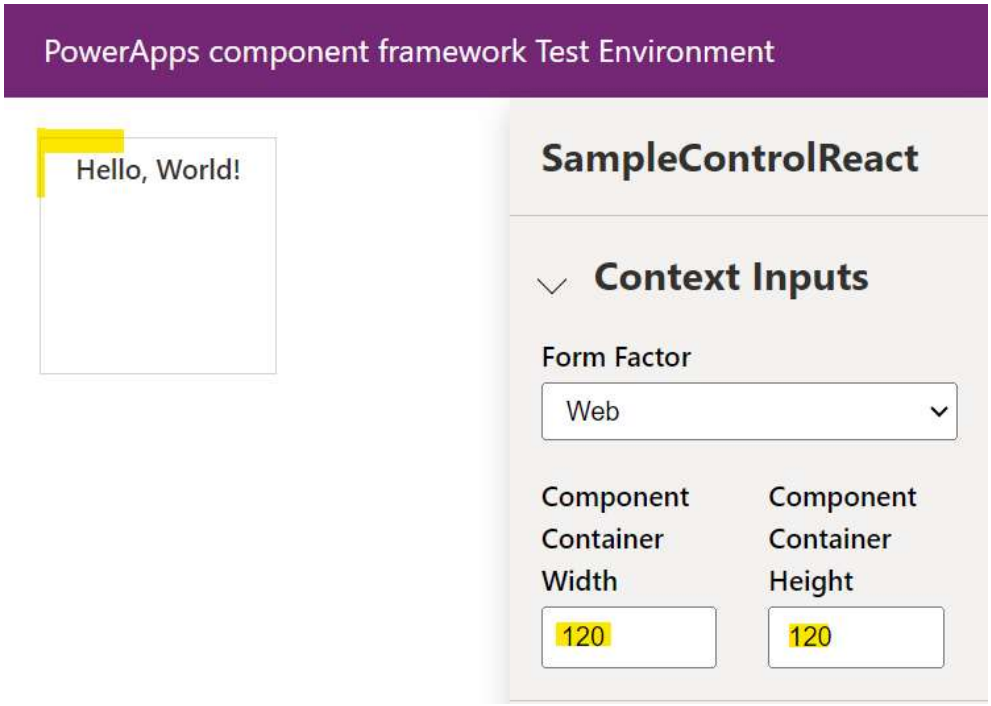
We're passing in the actual prop name value in the updateView, here it is "Hello, World!":

```
public updateView(context: ComponentFramework.Context<IInputs>): React.ReactElement {
    const props: IHelloWorldProps = { name: 'Hello, World!' };
    return React.createElement(
        HelloWorld, props
    );
}
```

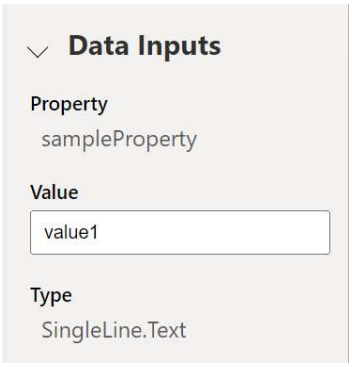
Running our test harness with npm start, we see our Hello World control:



As with the non-react example, we have the ability to set the width and height:



And we have our sampleProperty which isn't doing anything right now:



The sampleProperty again is defined in our ControlManifest.Input.xml file :

```
18        -->
19      </external-service-usage>
20      <!-- property node identifies a specific, configurable piece of data tha
21      <property name="sampleProperty" display-name-key="Property_Display_Key"
22      <!--
```

Let's deploy this to our Dataverse org and see what it looks like. We go through the same packaging steps as we did with the non-React control.

First, create a subdirectory called Solution and in the console go into that directory, then run the solution init. In my case, the developer is Carl and the solution prefix is carl:

```
pac solution init --publisher-name Carl --publisher-prefix carl
```

Next add references to the project using the project path:

```
pac solution add-reference --path C:\PCF\SampleControlReact
```

Then run msbuild (see my blog on deploying a standard control if you run into errors):

```
msbuild /t:build /restore
```

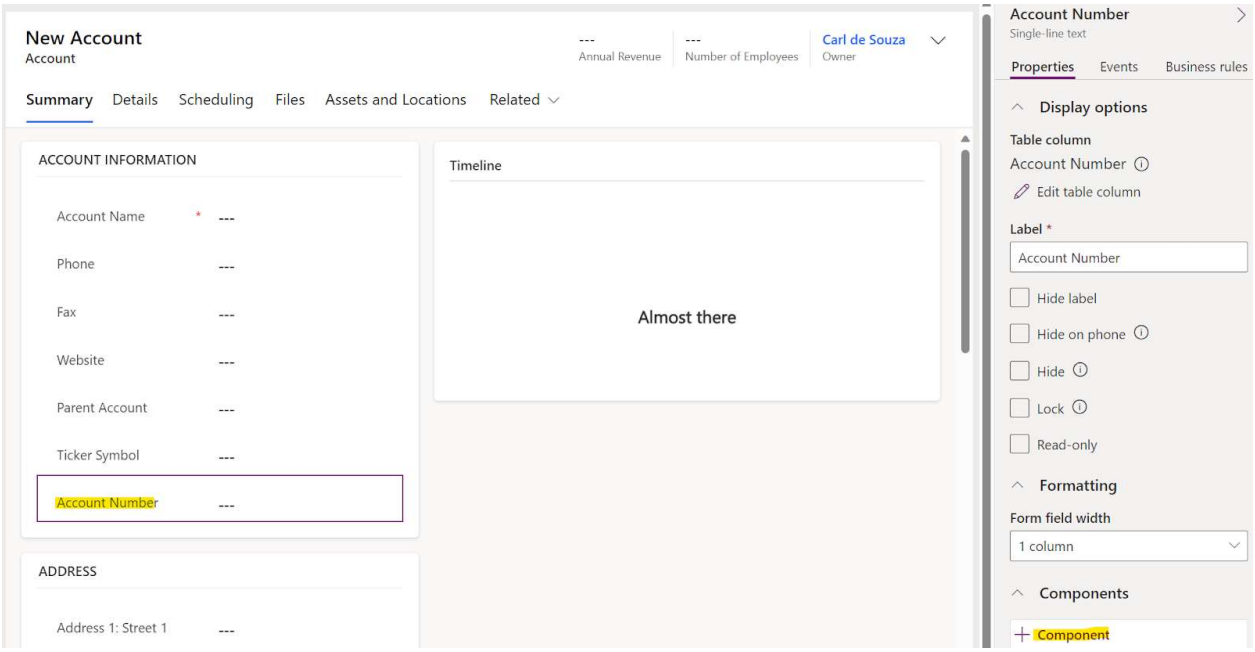
Now connect to the Dataverse (replace the org below with yours):

```
pac auth create --url https://org876ac311.crm.dynamics.com/
```

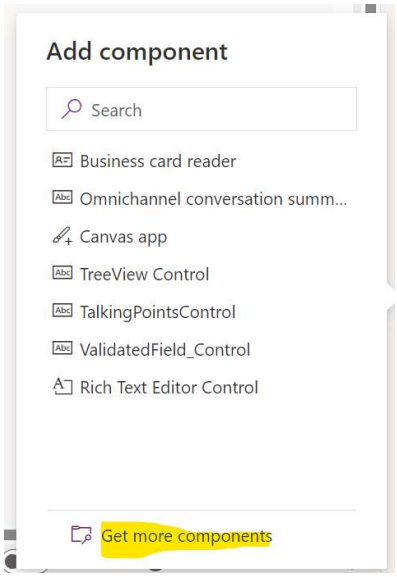
Now go back to the component folder and do the final step to push (replace the prefix with yours):

```
pac pcf push --publisher-prefix carl
```

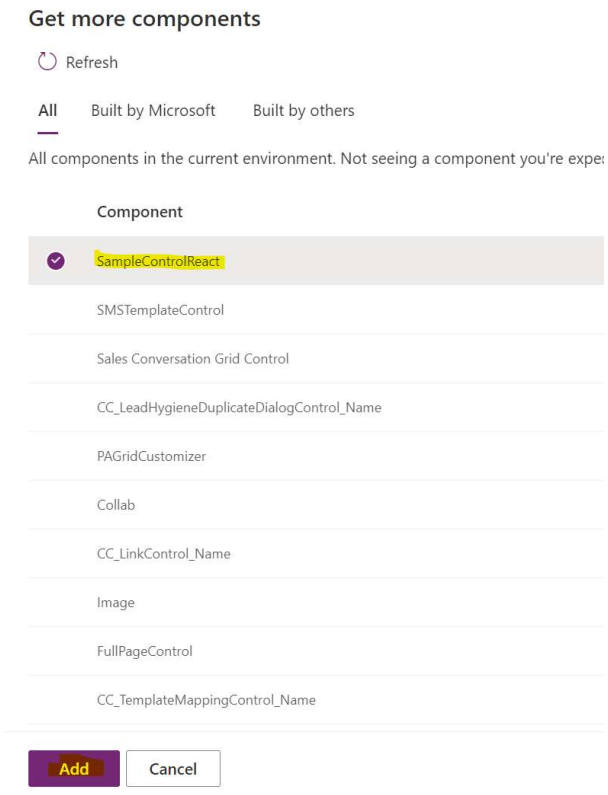
Now go to the Dataverse org and we can set our component to a field on a form. We will bind our control to the Account Number field on the Account form. Drag the field onto the form and go to + Component:



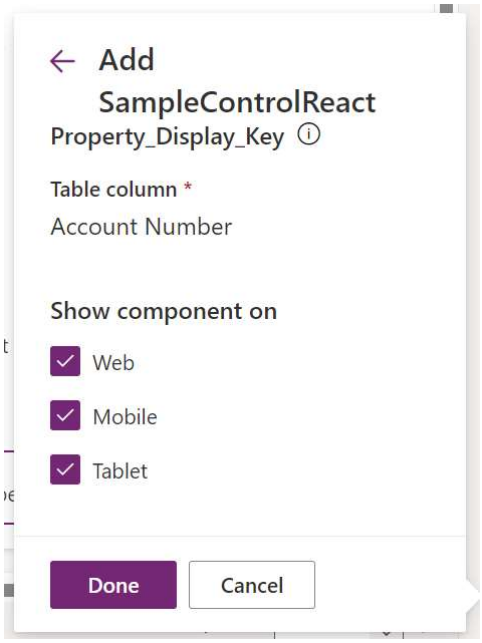
Then **Get more components**:



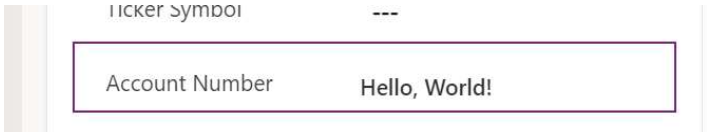
Select the control then click Add:

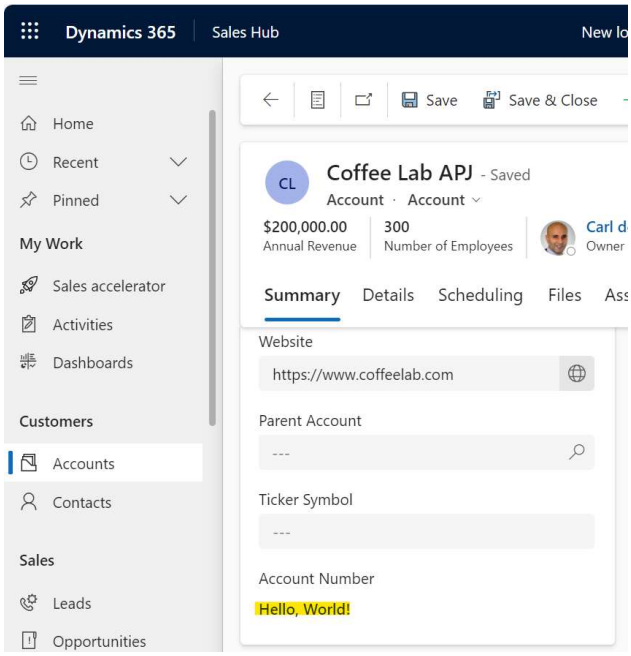


Select it again on the next screen and click Add:



We see our control rendered:





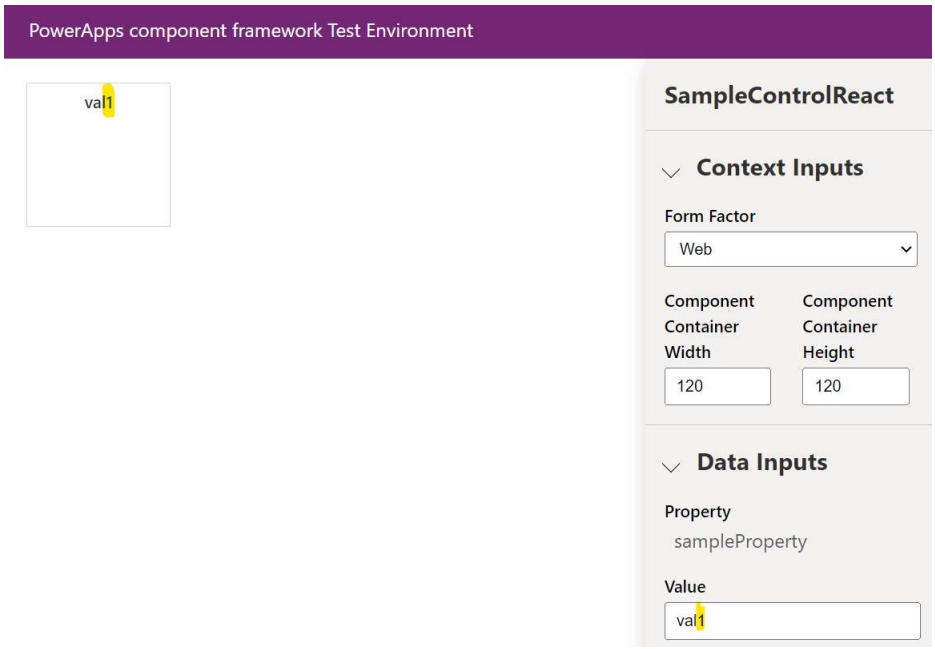
Now let's extend the control.

Let's change it so it uses the value of the property, not the preset Hello World. We will change the updateView to use:

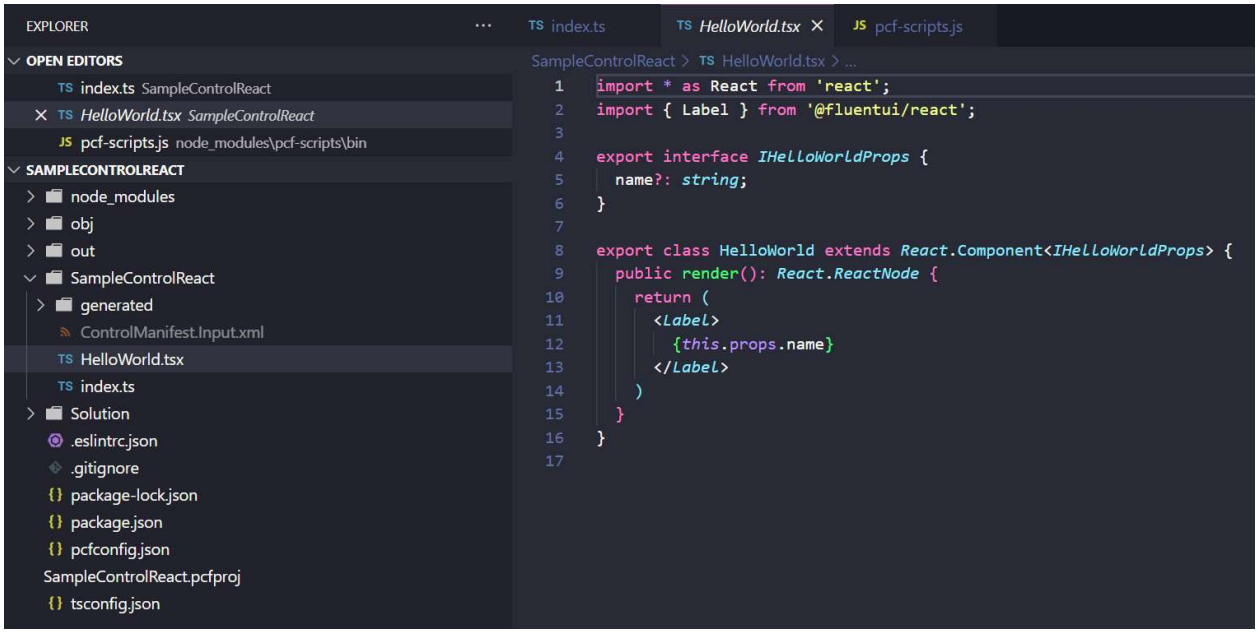
```
context.parameters.sampleProperty.raw || ""
```

```
public updateView(context: ComponentFramework.Context<IInputs>): React.ReactElement {
    const props: IHelloWorldProps = { name: context.parameters.sampleProperty.raw || "" };
    return React.createElement(
        HelloWorld, props
    );
}
```

Now when we run in the test harness with npm start watch, any change to the value changes the label in the control immediately without tabbing off the field:



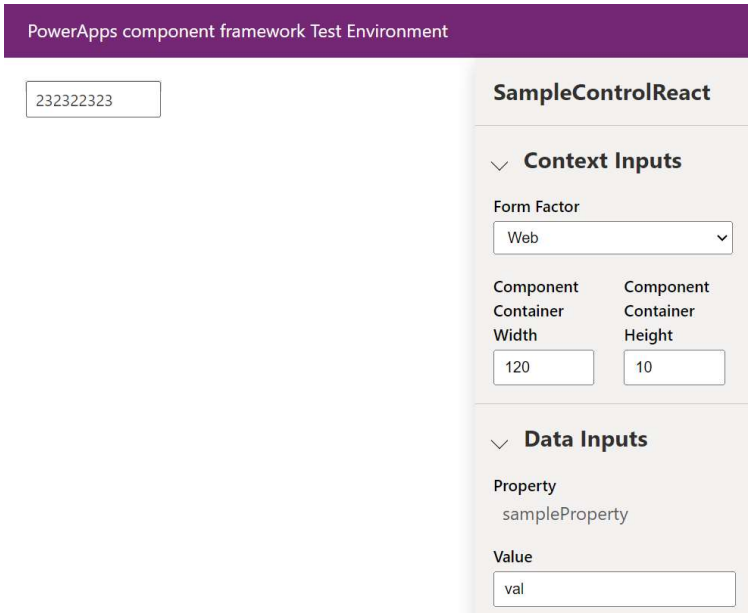
Let's go the other way now and change the label to a textbox. We can do this by changing the HelloWorld.tsx code. Currently the code returns a react Label:



Note the label comes from the fluentui/react library imported at the top. To change this to an input textbox we will use FluentUI as well by importing **TextField**:

```
1  import * as React from 'react';
2  import { TextField } from '@fluentui/react';
3
4  export interface IHelloWorldProps {
5    name?: string;
6  }
7
8  export class HelloWorld extends React.Component<IHelloWorldProps> {
9    public render(): React.ReactNode {
10     return <TextField />;
11   }
12 }
```

Running this, we see we now have a textbox instead of a label, however the value between the control and the sampleProperty value are not synced:



This is because we don't have any mechanism to do an onchange on the field currently. Let's add some code.

In our HelloWorld.tsx we can change our <TextField> to set the value from the props.name just like the label above, and we will add an onChange event which will call a method called handleOnChange:

```
import * as React from 'react';
import { TextField } from '@fluentui/react';

export interface IHelloWorldProps {
  name?: string;
  updateValue: (value: any) => void;
}

export class HelloWorld extends React.Component<IHelloWorldProps> {
  public render(): React.ReactNode {
    return <TextField
      value={this.props.name}
      onChange={this.handleOnChange}
      />
  }

  private handleOnChange = (event: React.FormEvent<HTMLInputElement | HTMLTextAreaElement>, newValue?: string) => {
    console.log(newValue);
    this.props.updateValue(newValue);
  }

}
```

In our index.tsx, we have a new property called _value which will hold our new text value:

```
export class SampleControlReact
  implements ComponentFramework.ReactControl<IInputs, IOutputs>
{
//  private theComponent: ComponentFramework.ReactControl<IInputs,
  private notifyOutputChanged: () => void;
  public _value: string | undefined;
```

We also have a new method called updateValue, which sets our new value and runs notifyOutputChanged, which then calls our updateView:

```
  public updateView(
    context: ComponentFramework.Context<IInputs>
  ): React.ReactElement {
    const props: IHelloWorldProps = {
      name: context.parameters.sampleProperty.raw || "",
      updateValue: this.updateValue.bind(this),
    };
    return React.createElement(HelloWorld, props);
  }

  private updateValue(value: any) {
      this._value = value;
      //this._props.value = "";

    this.notifyOutputChanged();
  }
```

The index.tsx code:

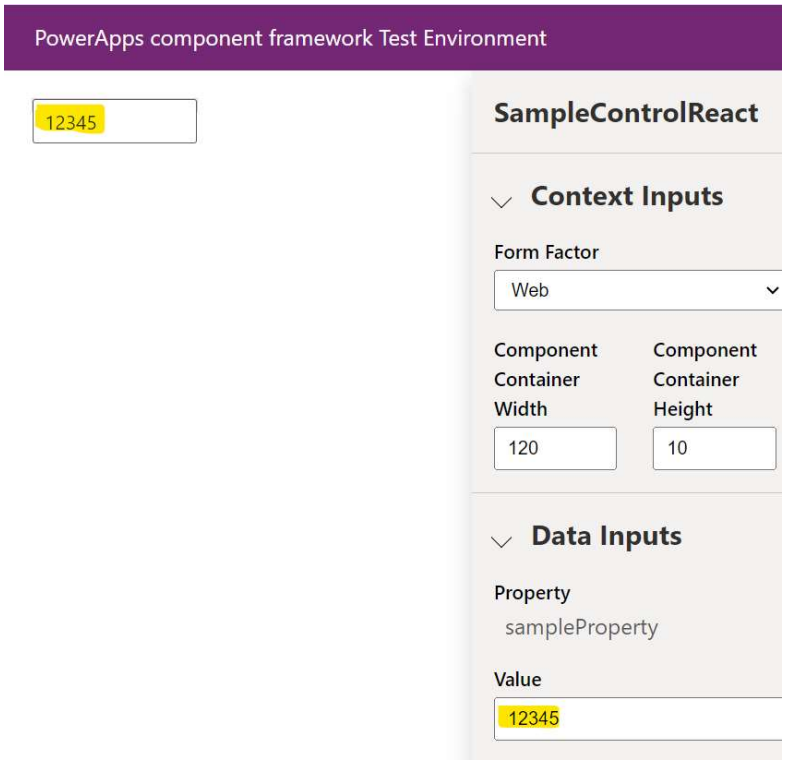
```
 public init(
    context: ComponentFramework.Context,
    notifyOutputChanged: () => void,
    state: ComponentFramework.Dictionary
  ): void {
    this.notifyOutputChanged = notifyOutputChanged;
  }

  /**
```

```
   * Called when any value in the property bag has changed. This includes
field values, data-sets, global values such as container height and width,
offline status, control metadata values such as label, visible, etc.
   * @param context The entire property bag available to control via Context
Object; It contains values as set up by the customizer mapped to names defined
in the manifest, as well as utility functions
   * @returns ReactElement root react element for the control
   */
  public updateView(
    context: ComponentFramework.Context
  ): React.ReactElement {
    const props: IHelloWorldProps = {
      name: context.parameters.sampleProperty.raw || "",
      updateValue: this.updateValue.bind(this),
    };
    return React.createElement(HelloWorld, props);
  }

  private updateValue(value: any) {
      this._value = value;
      //this._props.value = "";

    this.notifyOutputChanged();
  }
```

And the HelloWorld.tsx code:

```
export interface IHelloWorldProps {
  name?: string;
  updateValue: (value: any) => void;
}

export class HelloWorld extends React.Component {
  public render(): React.ReactNode {
    return
  }

  private handleOnChange = (event: React.FormEvent, newValue?: string) => {
    console.log(newValue);
    this.props.updateValue(newValue);
  }

}
```

Running this in our test harness, we can type into the text field or the SampleProperty value and both will get updated without tabbing off the field:



And running this in our Dynamics 365 org, we see the text box is rendered with our text field. Entering any text makes the form dirty, and saving preserves the value on the record as expected:

That's a simple react PCF control.

Building a Dataset PCF Control with Styling (without React)

Previously, we created PCF field controls, which are controls that attach to Power Apps fields. Let's look at how to create dataset PCF controls, which replace datasets in Power Apps and Dynamics 365.

To do this, let's create a new folder called SampleDatasetControl and initialize the project using the command. Note the **-t dataset** for the dataset template:

```
pac pcf init -n SampleDatasetControl -ns carl -t dataset
```

Let's open this in VS Code. In the index.ts file, the import area looks a little different to the field template, we have a couple of new lines regarding the DataSetApi:

```
import {IInputs, IOutputs} from "./generated/ManifestTypes";
import DataSetInterfaces = ComponentFramework.PropertyHelper.DataSetApi;
type DataSet = ComponentFramework.PropertyTypes.DataSet;
```

Generally the rest of the index.ts file looks the same as the field template, with the methods init, updateView, getOutputs, and destroy.

```
/**
 * Used to initialize the control instance. Controls can kick off remote server calls and other initialization a
 * Data-set values are not initialized here, use updateView.
 * @param context The entire property bag available to control via Context Object; It contains values as set up
 * @param notifyOutputChanged A callback method to alert the framework that the control has new outputs ready to
 * @param state A piece of data that persists in one session for a single user. Can be set at any point in a con
 * @param container If a control is marked control-type='standard', it will receive an empty div element within
 */
public init(context: ComponentFramework.Context<IInputs>, notifyOutputChanged: () => void, state: ComponentFrame
{
    // Add control initialization code
}


/**
 * Called when any value in the property bag has changed. This includes field values, data-sets, global values s
 * @param context The entire property bag available to control via Context Object; It contains values as set up
 */
public updateView(context: ComponentFramework.Context<IInputs>): void
{
    // Add code to update control view
}

/**
 * It is called by the framework prior to a control receiving new data.
 * @returns an object based on nomenclature defined in manifest, expecting object[s] for property marked as "bou
 */
public getOutputs(): IOutputs
{
    return {};
}

/**
 * Called when the control is to be removed from the DOM tree. Controls should use this call for cleanup.
 * i.e. cancelling any pending remote calls, removing listeners, etc.
 */
public destroy(): void
{
    // Add code to cleanup control if necessary
}
```

In the ControlManifest.Input.xml we see the control:

```
<control namespace="carl" constructor="SampleDatasetControl" version="0.0.1" display-name-key="SampleDatasetControl" description-key="SampleDatasetControl description" control-type="stand
  <!--external-service-usage node declares whether this 3rd party PCF control is using external service or not, if yes, this control will be considered as premium and please also add the
  If it is not using any external service, please set the enabled="false" and DO NOT add any domain below. The "enabled" will be false by default.
```

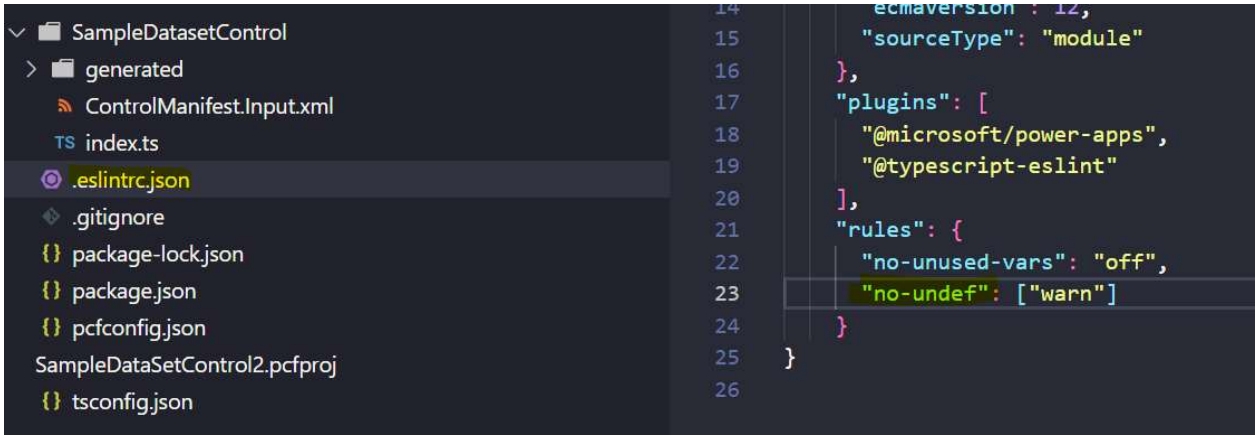
The big difference in the manifest is the data-set area:

```
<data-set name="sampleDataSet" display-name-key="Dataset_Display_Key">
  <!-- 'property-set' node represents a unique, configurable property that each record in the dataset must provide. -->
  <!-- UNCOMMENT TO ADD PROPERTY-SET NODE
  <property-set name="samplePropertySet" display-name-key="Property_Display_Key" description-key="Property_Desc_Key" of-type="SingleLine.Text" usage="bound" required="true" />
  -->
</data-set>
```

Let's build this with:

```
npm run build
```

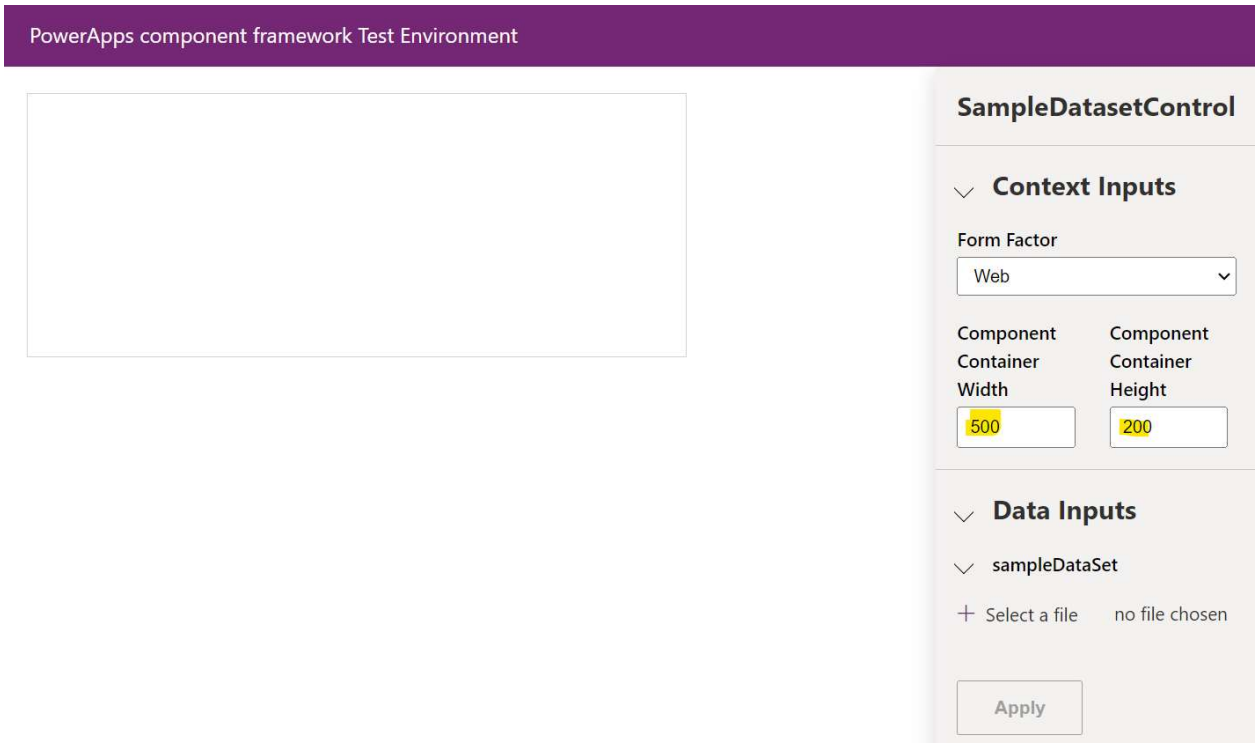
Note if you run into the error **'PropertyHelper' is not defined no-undef** you can resolve this by changing the .eslint.rc.json file to include a new line for **"no-undef": ["warn"]** under the **rules** section:



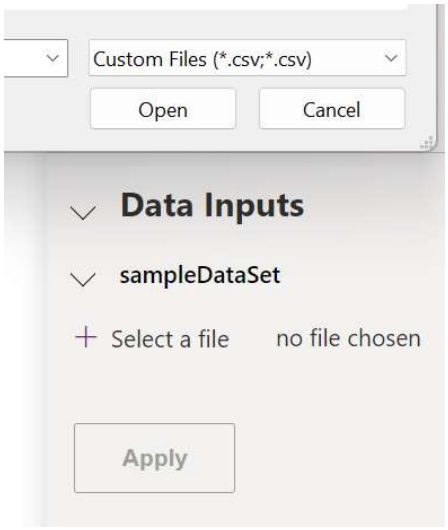
And run this in our test harness with:

```
npm run start
```

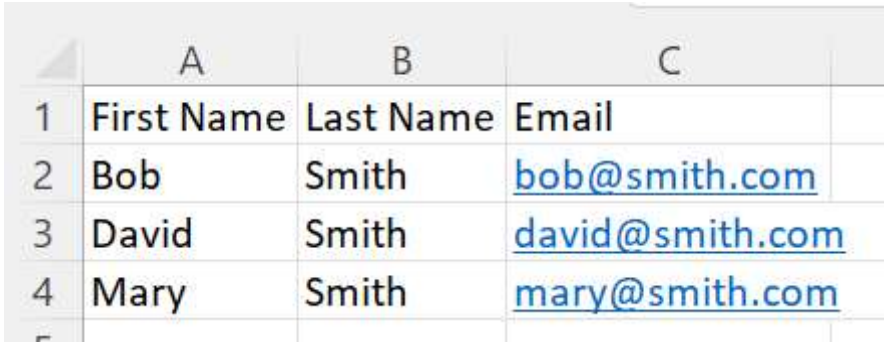
We see our test harness launch. Let's change the size of the control, and notice the option below that to select a sampleDataSet:



Clicking on this shows it accepts csv files:

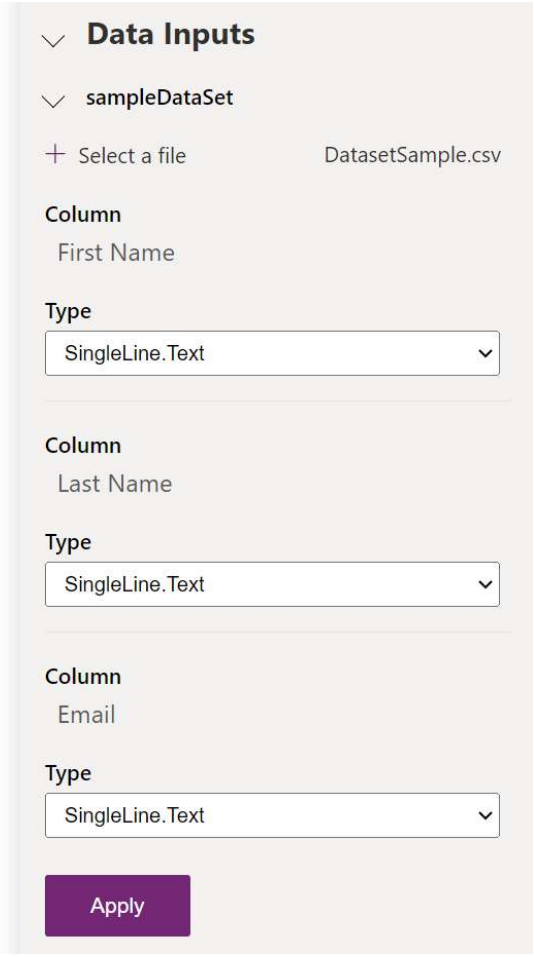


Let's upload the file a simple CSV file below:

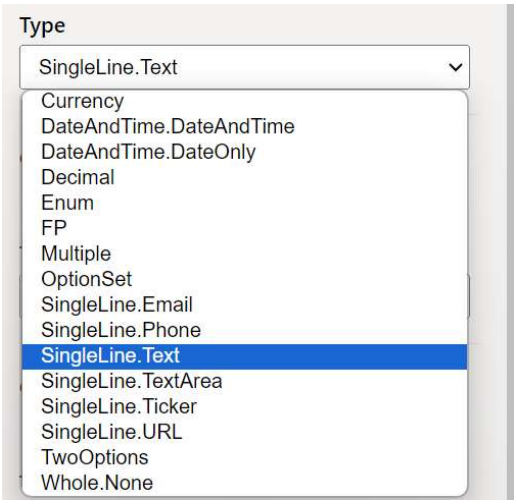


	A	B	C
1	First Name	Last Name	Email
2	Bob	Smith	bob@smith.com
3	David	Smith	david@smith.com
4	Mary	Smith	mary@smith.com

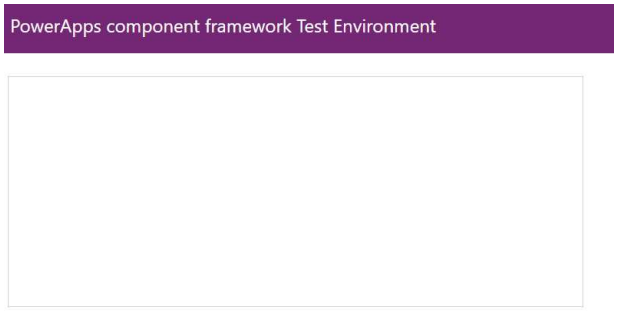
We now get to assign the data types to each field:



Note the different data type options:



At this point, nothing is displayed in our grid:



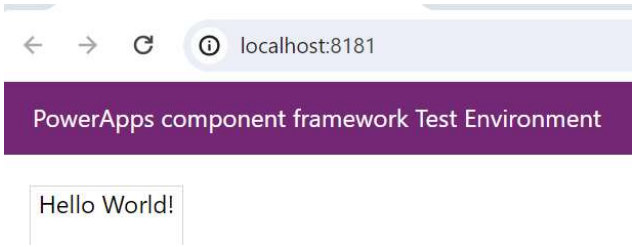
We'll need to add some code to render the dataset. Let's add some code. As with the **field** component we have an HTMLDivElement container:

```
public init(context: ComponentFramework.Context<IInputs>, notifyOutputChanged: () => void, state: ComponentFramework.Dictionary, container:HTMLDivElement): void
{
    // Add control initialization code
}
```

Simply speaking, we have a component where we could put anything to display, it doesn't have to be a grid, but of course we're using this template because we want to display a grid. Let's show as an example that we could return here a text field just like the field control:

```
public init(context: ComponentFramework.Context<IInputs>, notifyOutput
{
    // Add control initialization code
    container.appendChild(document.createTextNode("Hello World!"));

}
```

This displays our text box in the test harness:



Let's deploy this up to an org and see what it looks like as a label. We will go through the same deployment process as when deploying a field control.

First, let's package our component. I will create a subdirectory called Solution, go into it and then run the command:

```
pac solution init --publisher-name Carl --publisher-prefix carl
```

Next, add references to where our component is located:

```
pac solution add-reference --path C:\PCF\SampleDataSetControl2
```

Now run the below in the Solution directory:

```
msbuild /t:build /restore
```

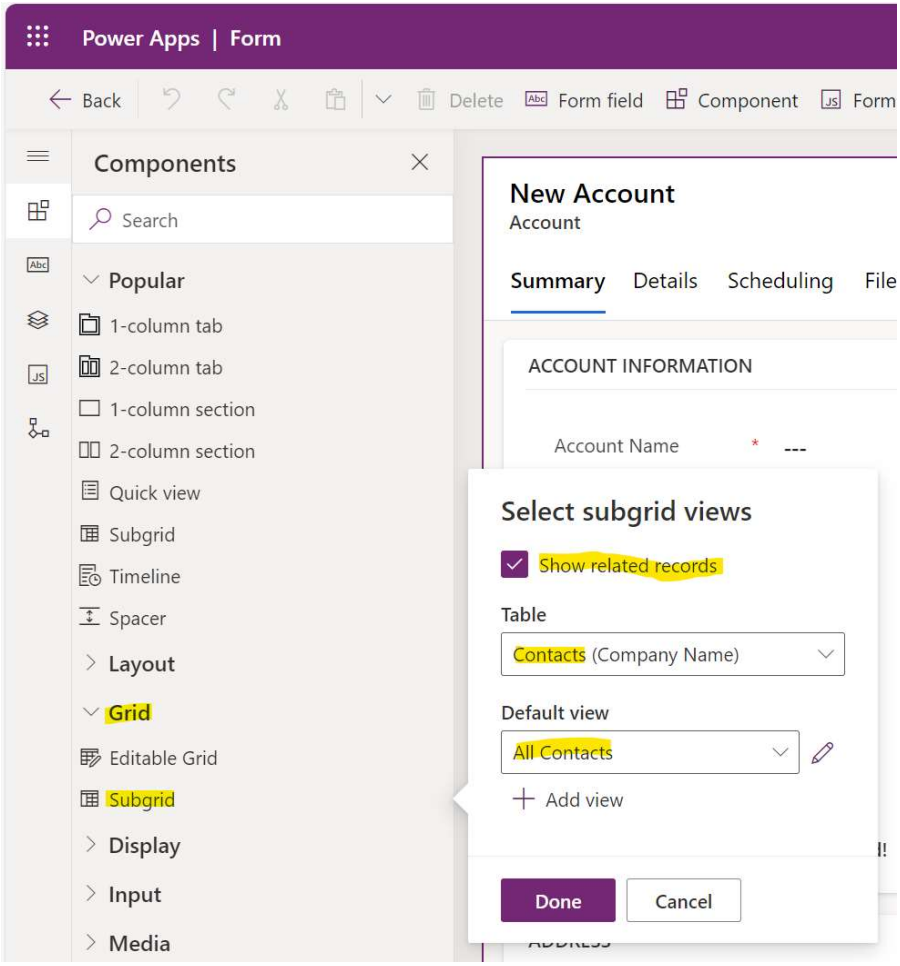
Now connect to your org:

```
pac auth create --url https://org876ac311.crm.dynamics.com/
```

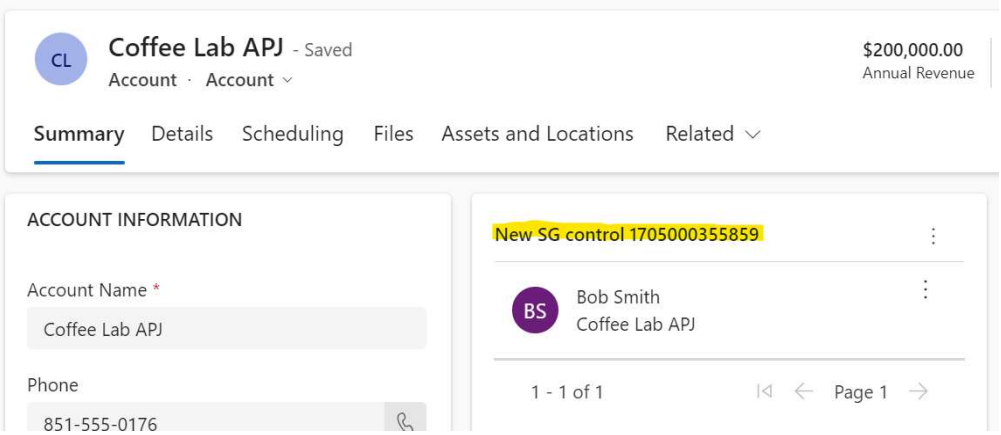
And from the component folder (not the Solution folder) do push:

```
pac pcf push --publisher-prefix carl
```

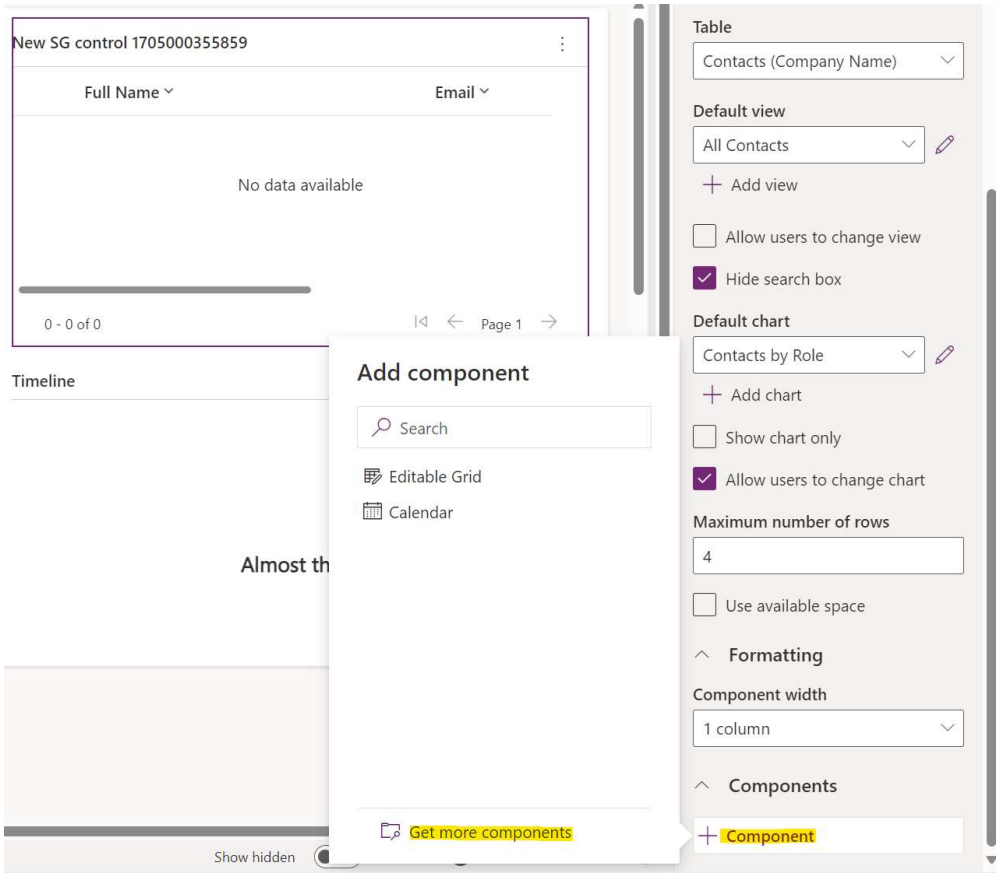
Now let's go to the Power Apps maker portal and to a form. We will add a subgrid:



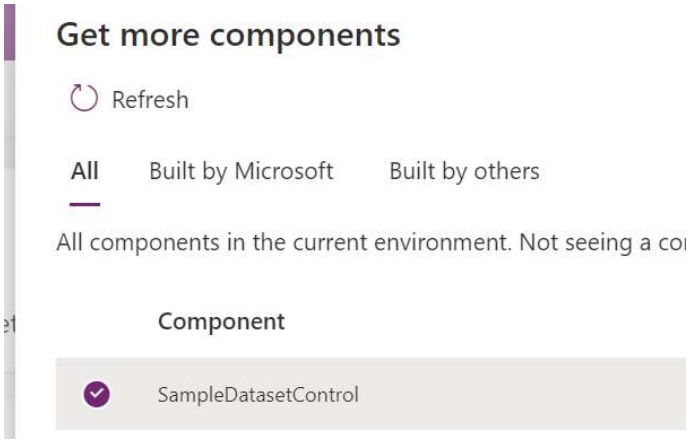
Our new subgrid appears on the form:



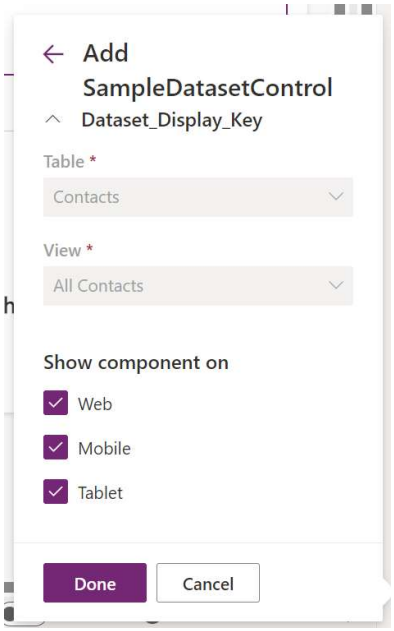
Now let's wire our subgrid to our custom control:



Select our new SampleDatasetControl:

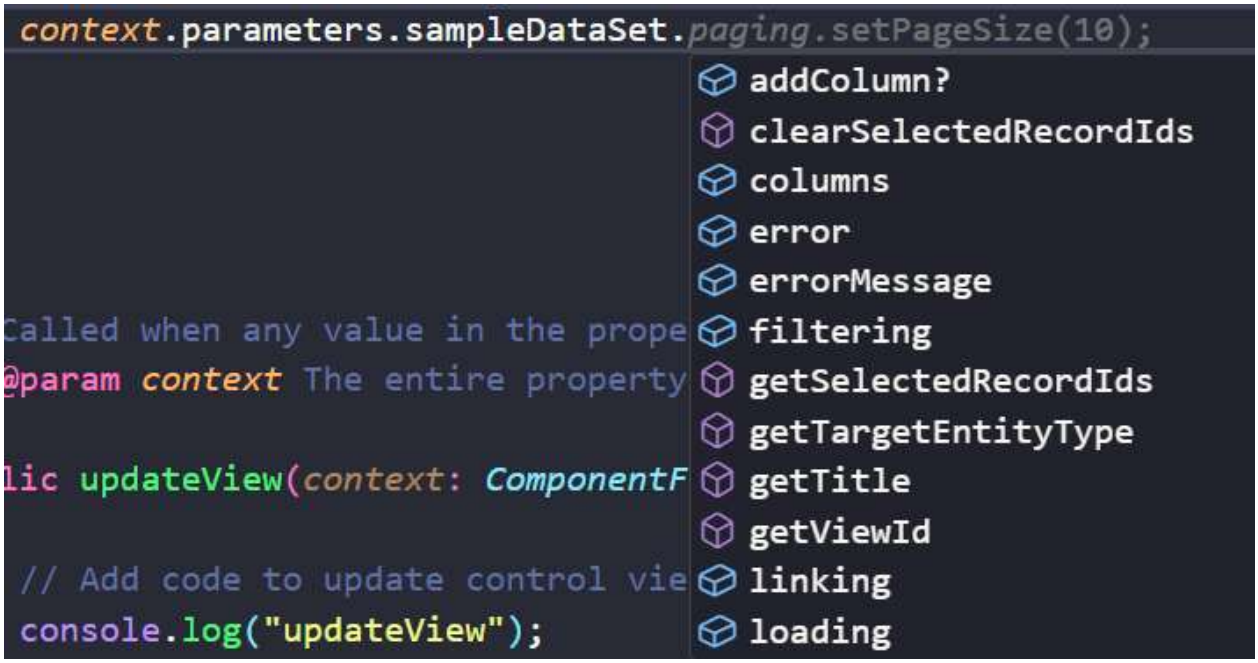


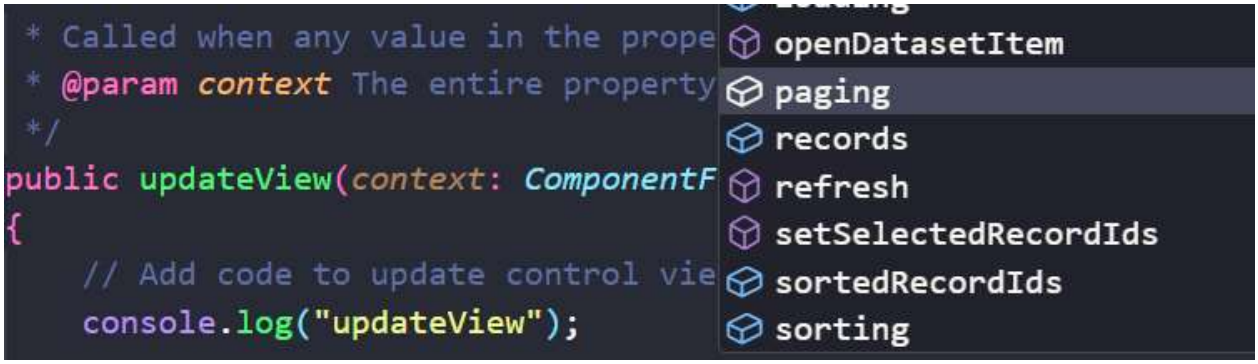
Select it again and click Done:



Right away we see our Hello World coming through:

Now that we see the dataset control is simply another control, let's look at how to wire it into a subgrid. Let's see what our context parameters gives us for datasets. We see in context.parameters.sampleDataSet we get a few useful properties:





Let's write out the columns to the console using the following code, and let's add it to the UpdateView:

```
context.parameters.sampleDataSet.columns.forEach(function(columnItem){
   console.log("HEADER: " + columnItem.displayName + " - " + columnItem.name);
});
```

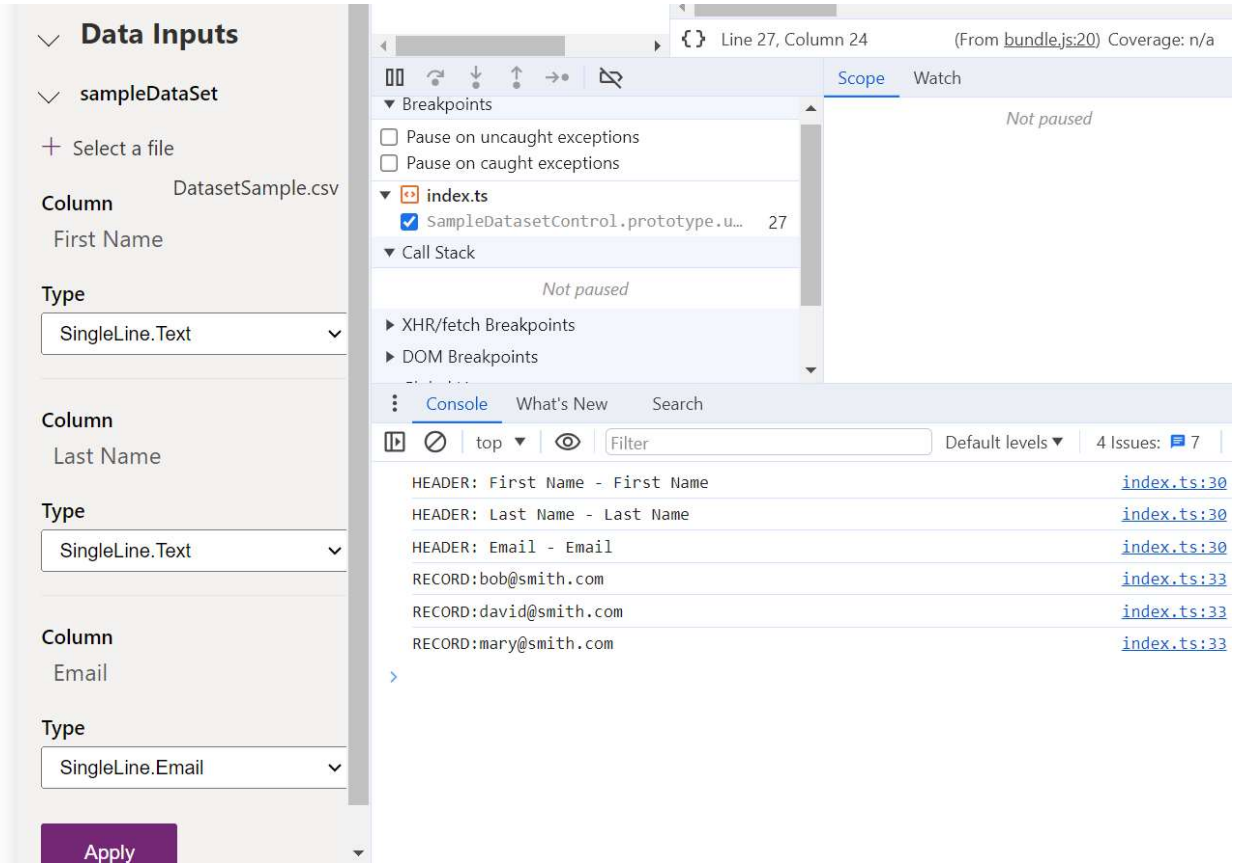
And we will write out the email addresses of each record using the code below:

```
context.parameters.sampleDataSet.sortedRecordIds.forEach(function(recordId){
    console.log("RECORD:" +
context.parameters.sampleDataSet.records[recordId].getFormattedValue("Email"))
;
});
```

It should look like below:

```
public updateView(context: ComponentFramework.Context<IInputs>): void
{
    // Add code to update control view
    context.parameters.sampleDataSet.columns.forEach(function(columnItem){
        console.log("HEADER: " + columnItem.displayName + " - " + columnItem.name);
    });

    context.parameters.sampleDataSet.sortedRecordIds.forEach(function(recordId){
        console.log("RECORD:" + context.parameters.sampleDataSet.records[recordId].getFormattedValue("Email"));
    });
}
```

Now let's test this. In our test harness, upload the CSV file again. We see in the console our header and records are now printing out from our CSV file!



OK, let's actually display this data in place of the grid. We will create a variable below to hold our container:

```
private _container : HTMLDivElement;
```

```
export class SampleDatasetControl implements ComponentFramework.StandardControl<IInputs, IOutputs>

    private _container : HTMLDivElement;
```

And let's initialize this in our init:

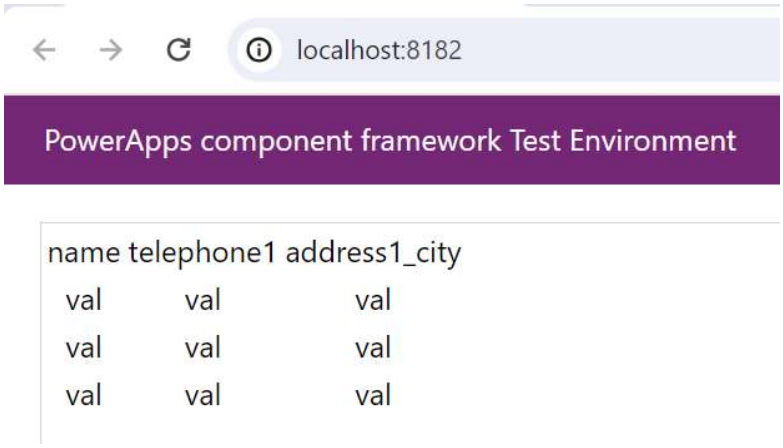
```
this._container = container;
```

```
public init(context: ComponentFramework.Context<IInputs>,
{
    // Add control initialization code
    this._container = container;
}
```

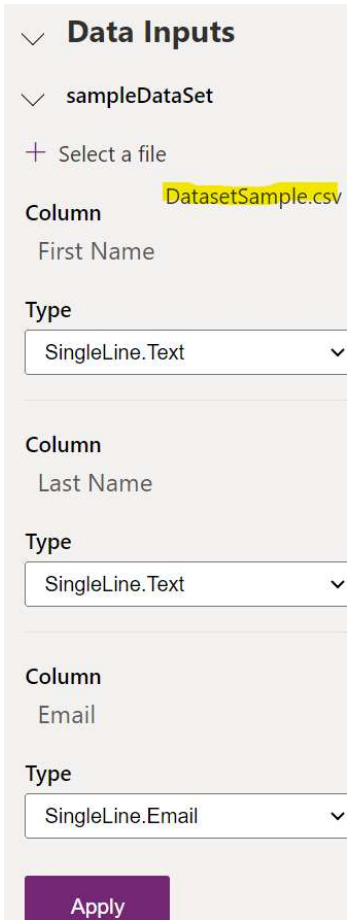
And to the **updateView** let's add the code below. The code basically takes our container, and firstly sets it to blank to wipe out any older rendering. We then go through each of the **columns** like above and attach them to a row in a new table. We then go through each of the records in **sortedRecordIds** to display each record (getting the column names as well). We then append our table to the container just like in other PCF examples:

```
      this._container.innerHTML = "";
       // Add code to update control view
       var table = document.createElement("table");
       var tr = document.createElement("tr");
       context.parameters.sampleDataSet.columns.forEach(function(columnItem){
         var td = document.createElement("td");
         td.appendChild(document.createTextNode(columnItem.displayName));
         tr.appendChild(td);
         table.appendChild(tr);
       });


context.parameters.sampleDataSet.sortedRecordIds.forEach(function(recordId){
       var tr = document.createElement("tr");


context.parameters.sampleDataSet.columns.forEach(function(columnItem){
         var td = document.createElement("td");

td.appendChild(document.createTextNode(context.parameters.sampleDataSet.record
s[recordId].getFormattedValue(columnItem.name)));
         tr.appendChild(td);
       });
         table.appendChild(tr);
     });

     this._container.appendChild(table);
```

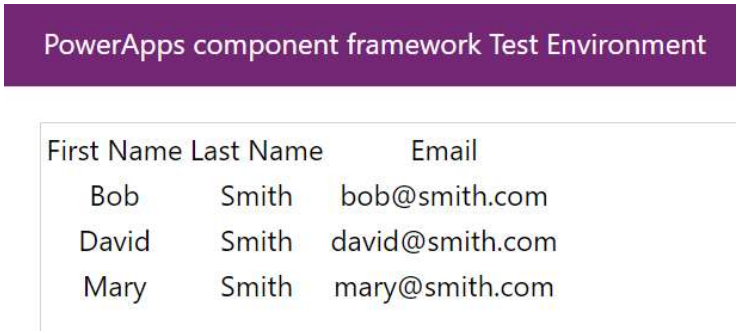
When we run this the first time in our harness we get the following displayed, with name, telephone1, and address1_city. This appears to be a background sample in the developer tools, not our sample:



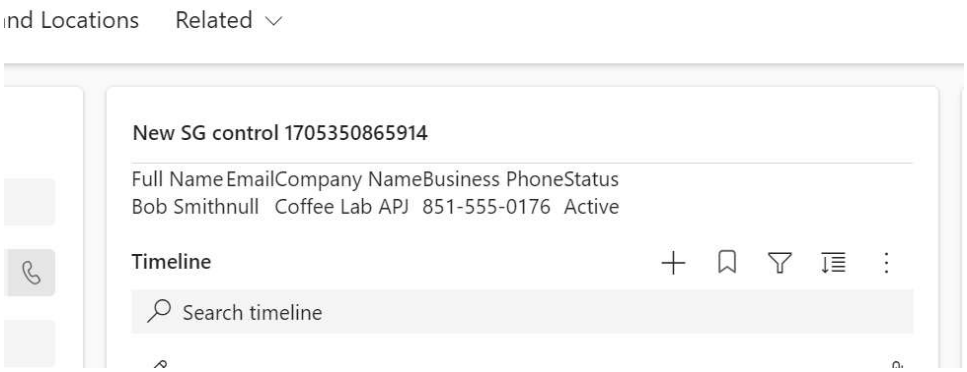
We can select our dataset on the right, like above and make sure the field types are set, then click Apply:



Once Apply is clicked our dataset is rendered:



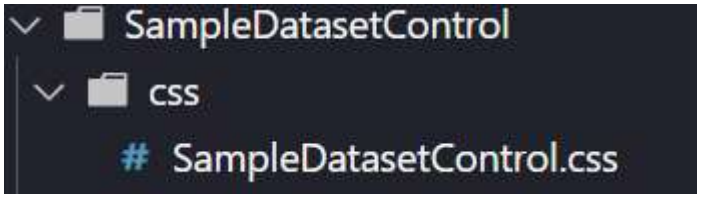
Let's deploy this back up to Power Apps and take a look. We can see our new subgrid control displays the fields from the view and the rows from the view (1 record in this case):



This doesn't look so good visually, so let's add a CSS to our code.

In our ControlManifest.Input.xml we will uncomment the line for the CSS under Resources. Note the path to the folder by default is css/SampleDataSetControl.css:

We will create a file and folder in this location:



And let's tweak our code a little, we will change our header creation in the updateView to be **th** instead of **td**:

```
context.parameters.sampleDataSet.columns.forEach(function(columnItem){
    var th = document.createElement("th");
    th.appendChild(document.createTextNode(columnItem.displayName));
    tr.appendChild(th);
    table.appendChild(tr);
});
```

Now our css, a simple styling for our table:

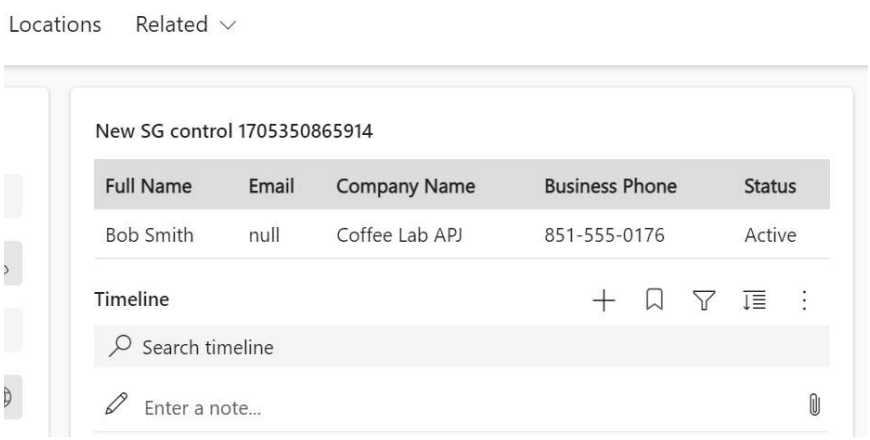
```
table {
    width: 100%;
    border-collapse: collapse;
}

table tr,
table td {
    padding: 8px;
    text-align: left;
    border-bottom: 1px solid #ddd;
}

table th {
    padding: 8px;
    text-align: left;
    border-bottom: 1px solid #ddd;
    background-color: #ddd;
    font-weight: bold;
}
```

Our test harness now looks like:

PowerApps component framework Test Environment

First Name	Last Name	Email
Bob	Smith	bob@smith.com
David	Smith	david@smith.com
Mary	Smith	mary@smith.com

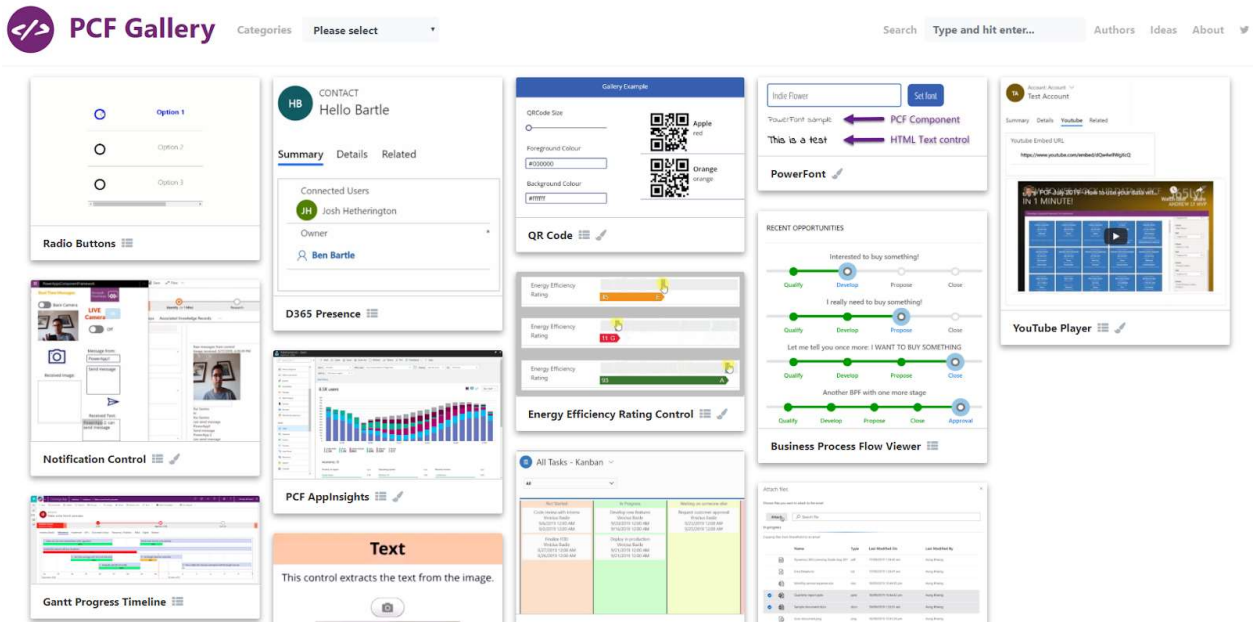
Let's push this to our org. It looks much better:


Locations
Related
New SG control 1705350865914
Full Name
Email
Company Name
Business Phone
Status
Bob Smith
null
Coffee Lab APJ
851-555-0176
Active
Timeline
Search timeline
Enter a note...

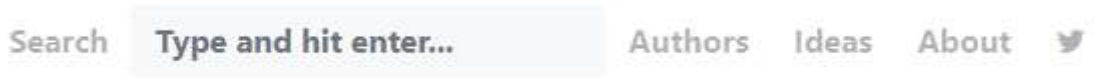
How to Install Power Apps Components from PCF.Gallery in your Org

The PCF.gallery site contains many PowerApps custom components written by people in the PowerApps community. Some of the controls are really cool, and now we will look at how to use the site and install controls in your PowerApps / Dynamics 365 environment to make your orgs even more usable.

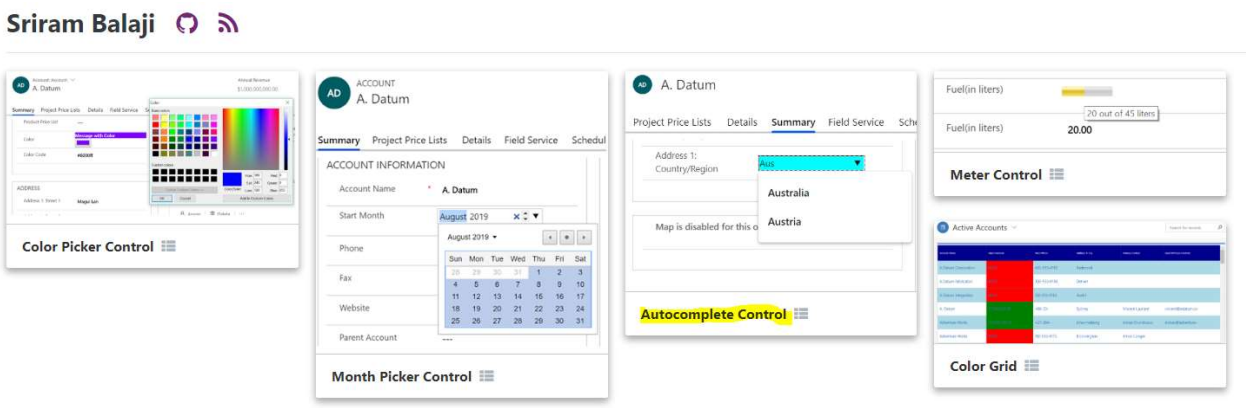
First, head over to PCF.gallery. You will see links to various controls:



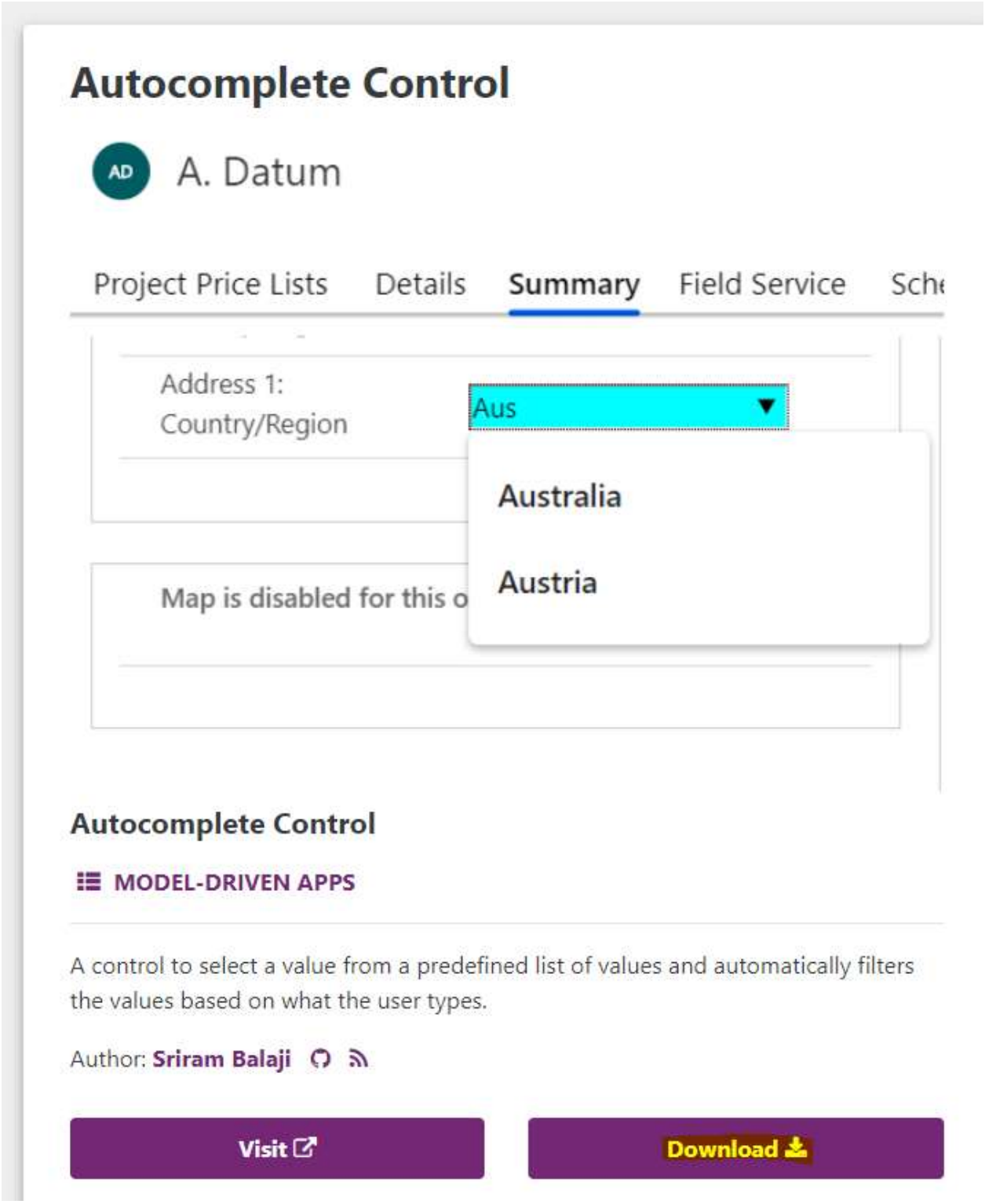
You can click on a link to select a control, or from the top right, search for a control or select Authors to find controls that way:



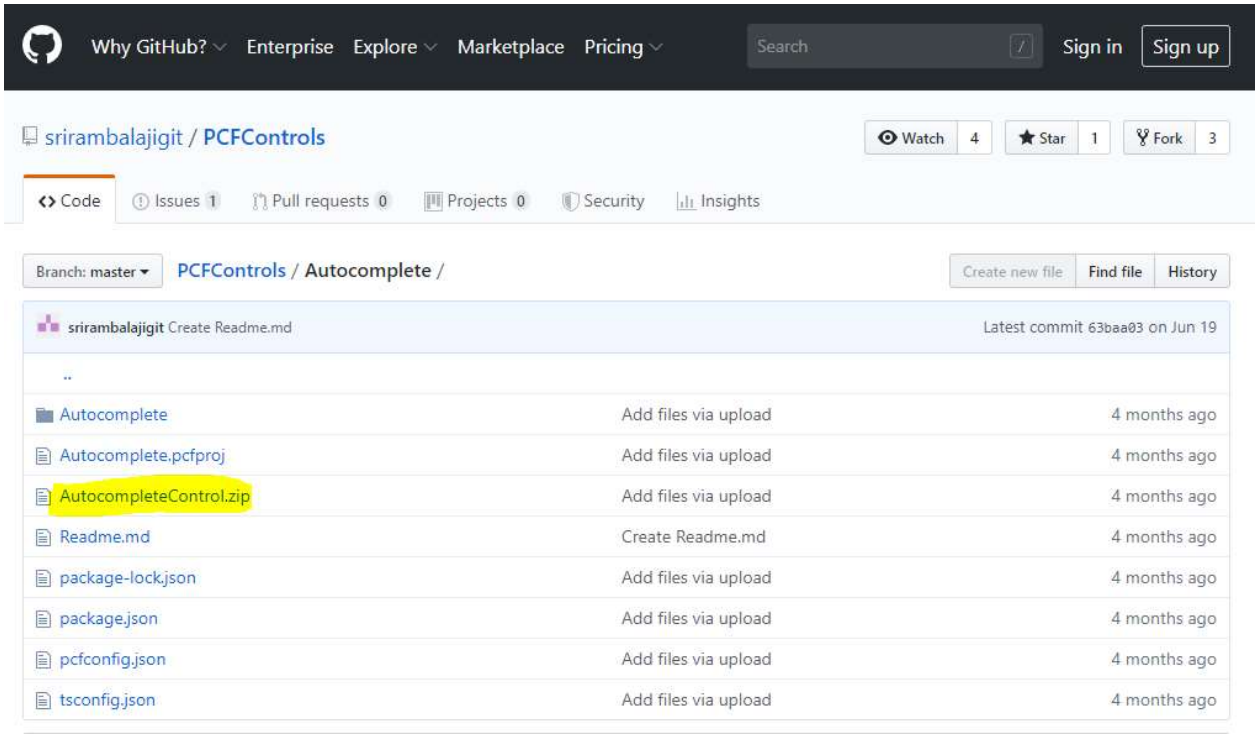
Let's install the AutoComplete control by Sriram Balaji:



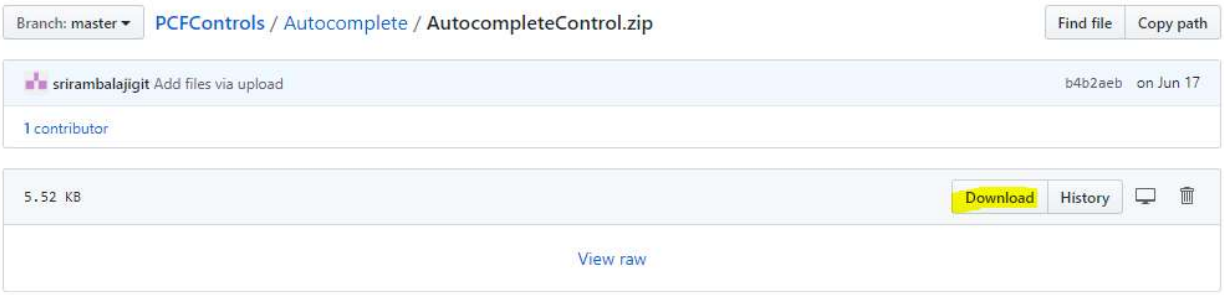
The link to the PCF.gallery page is https://pcf.gallery/autocomplete-control/. Select Download:



This takes us to the GitHub repository with the ZIP file we need (direct link here):

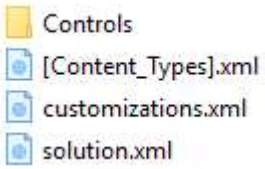


Click Download:

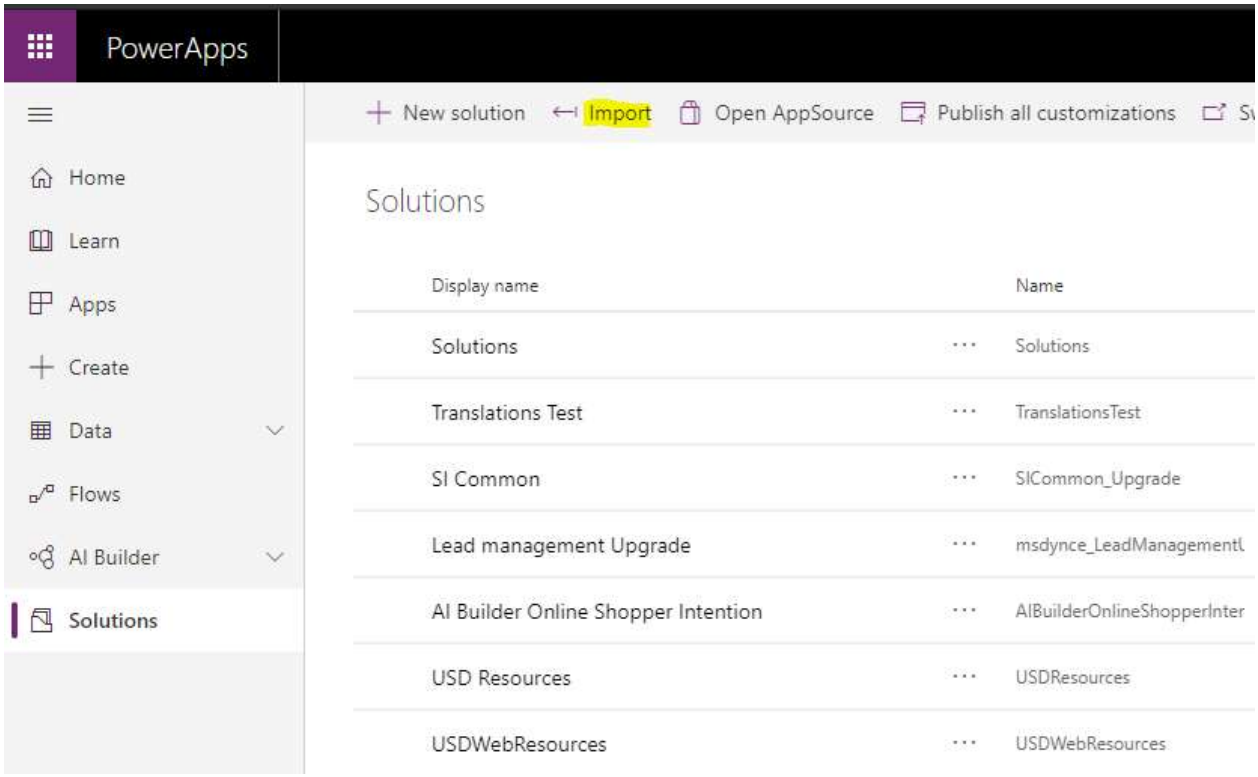


This downloads the file:

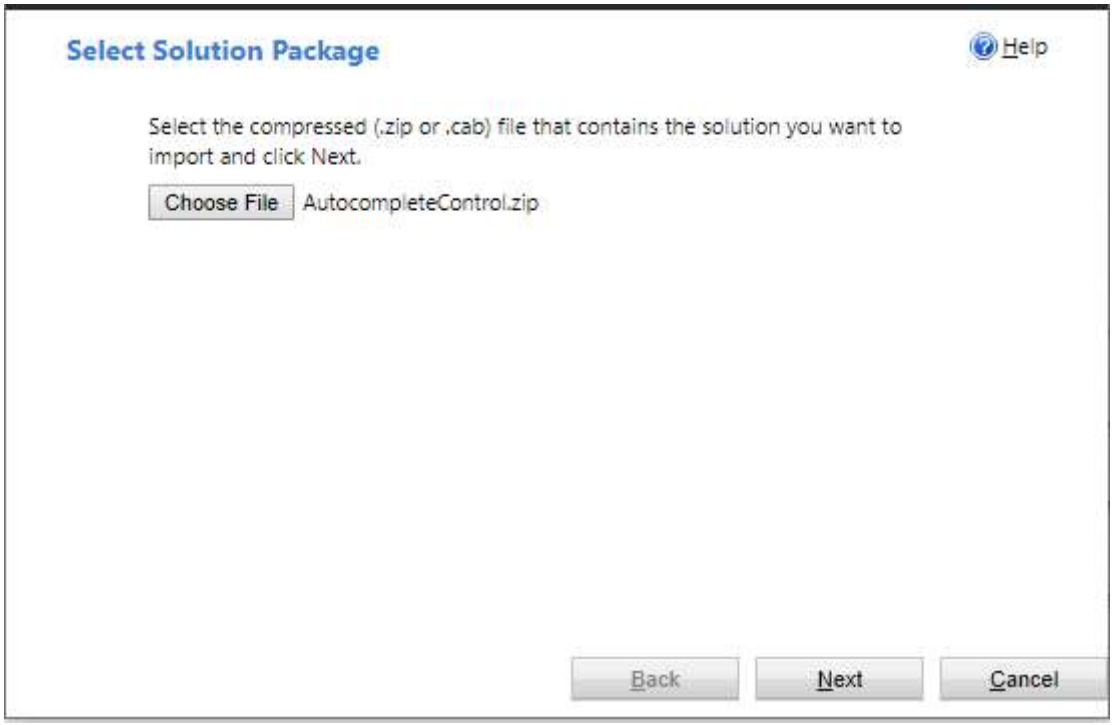
Which if we open is a solution file:



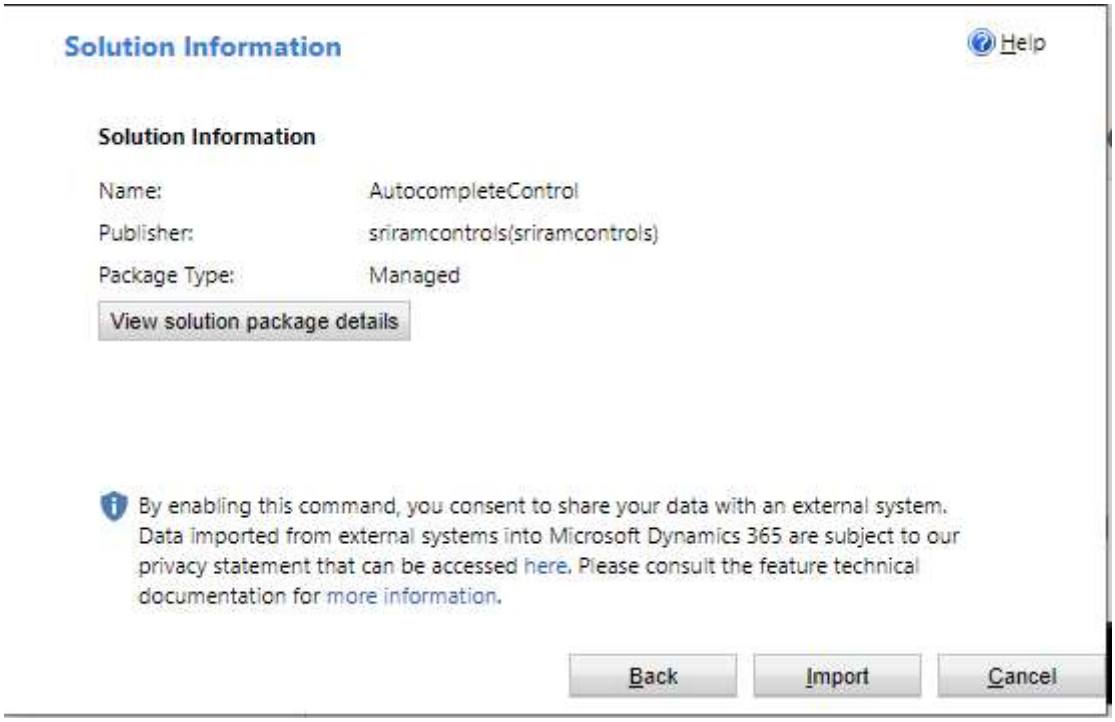
Let's install it in our org. Go to PowerApps at https://make.powerapps.com/ and select your environment, then Solutions and Import:



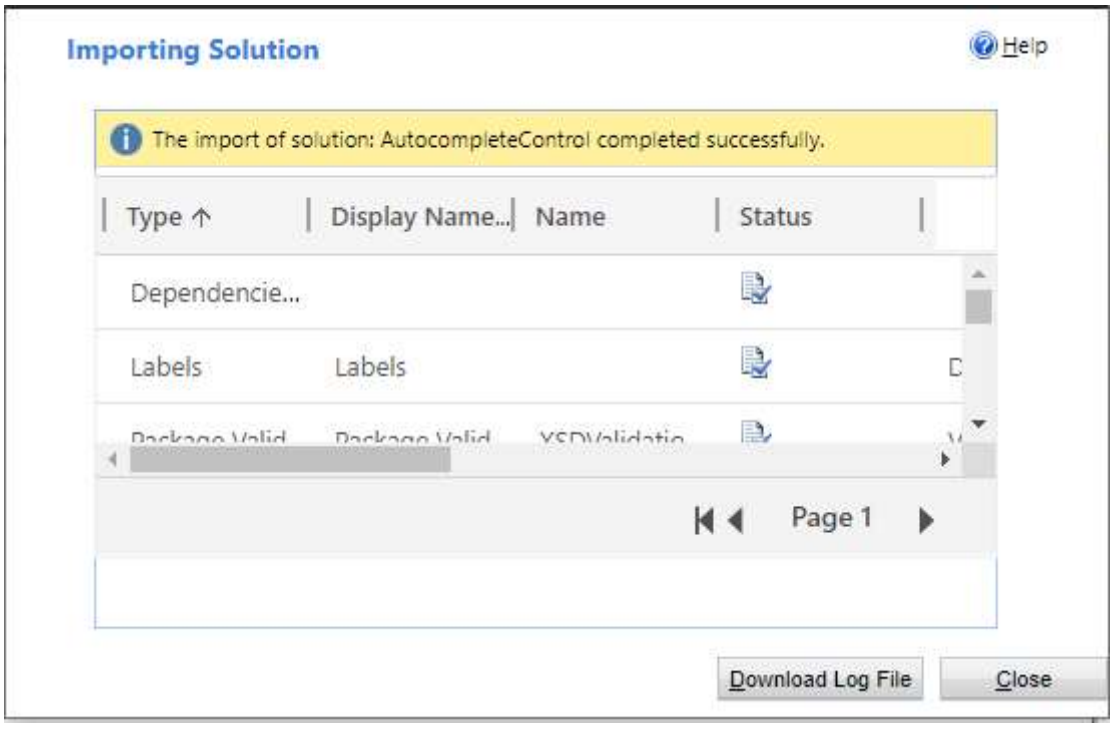
Select the ZIP file, then click Next:



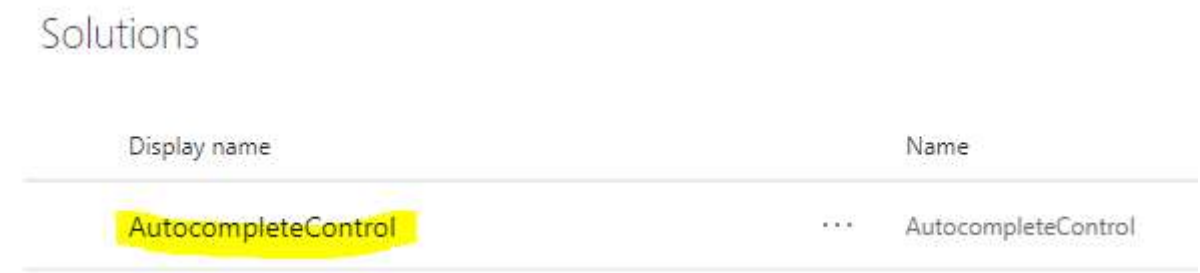
Then Import:



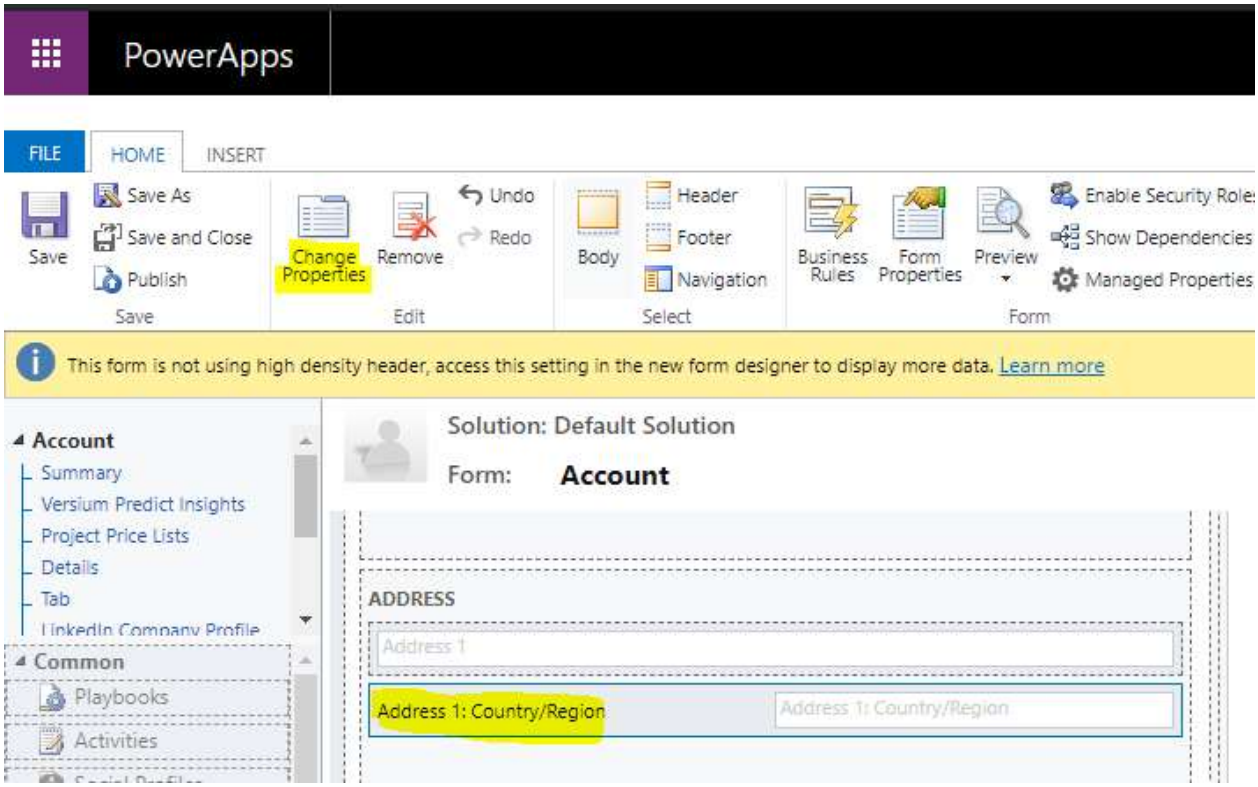
Then click Close:



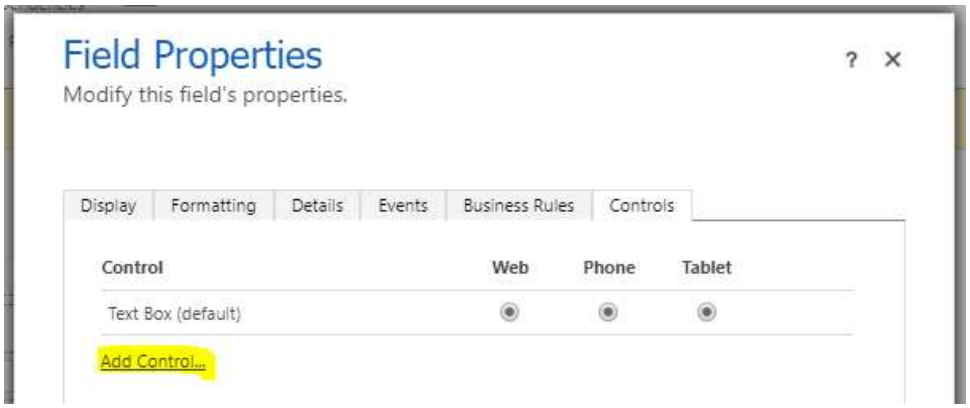
We see the solution has imported:



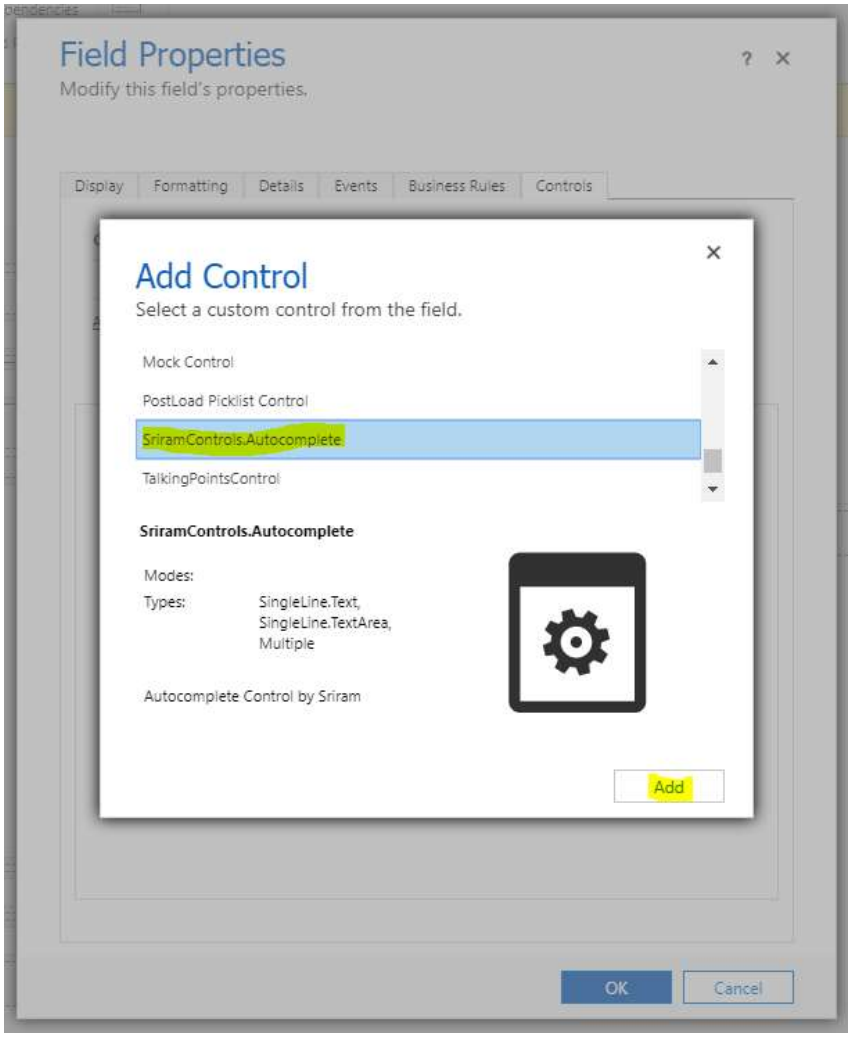
Now let's use the new control. We will add it to the Account form on the Country field, select it and click Change Properties.



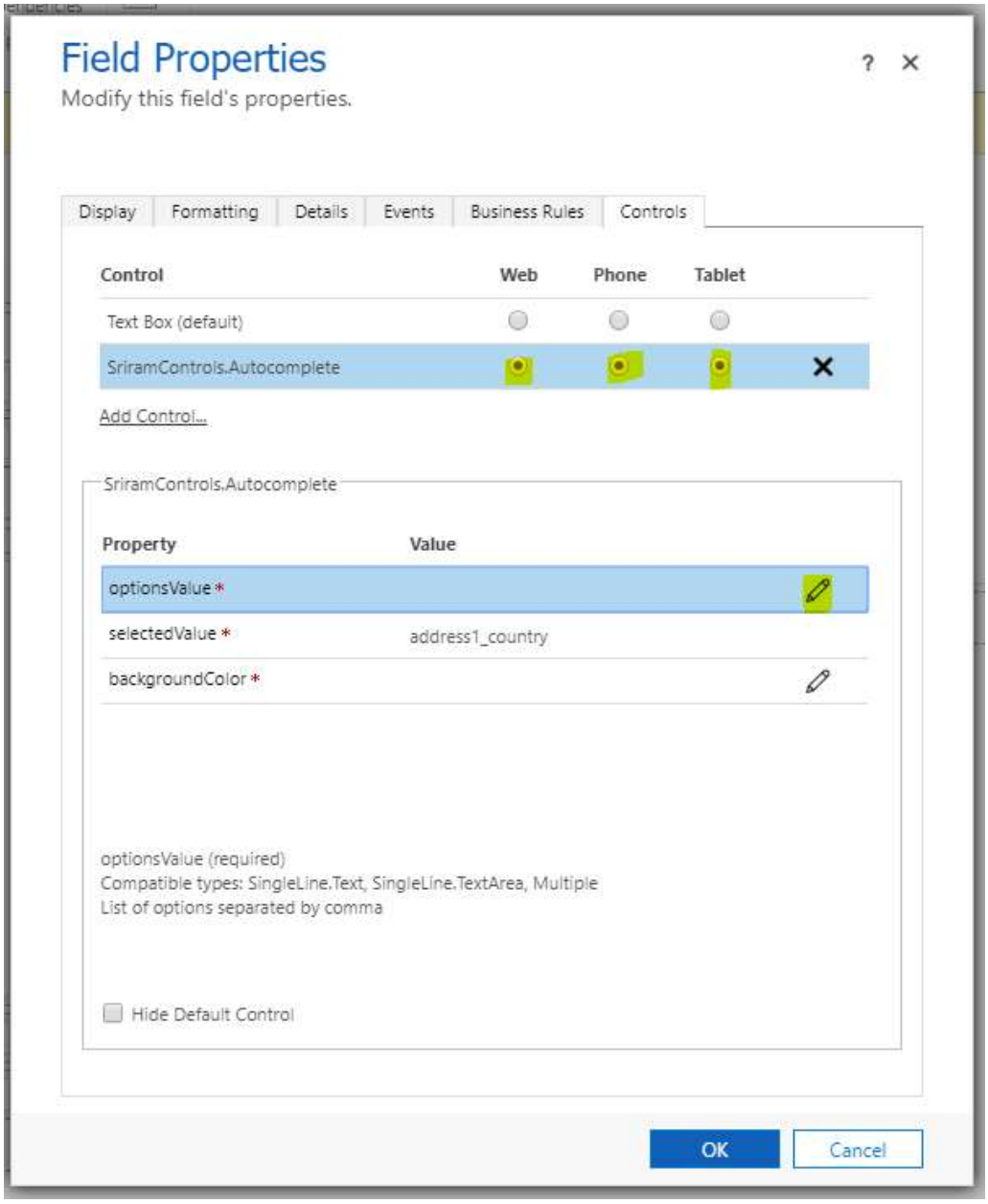
Go to Controls and select Add Control:



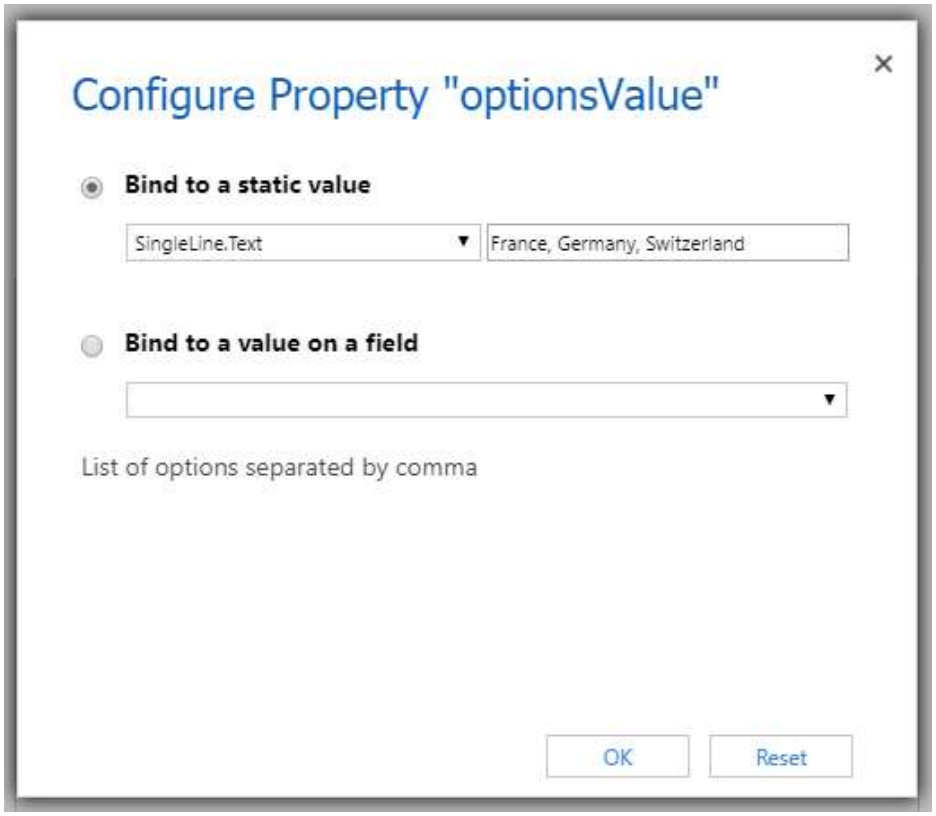
Find SriramControls.Autocomplete and click Add:



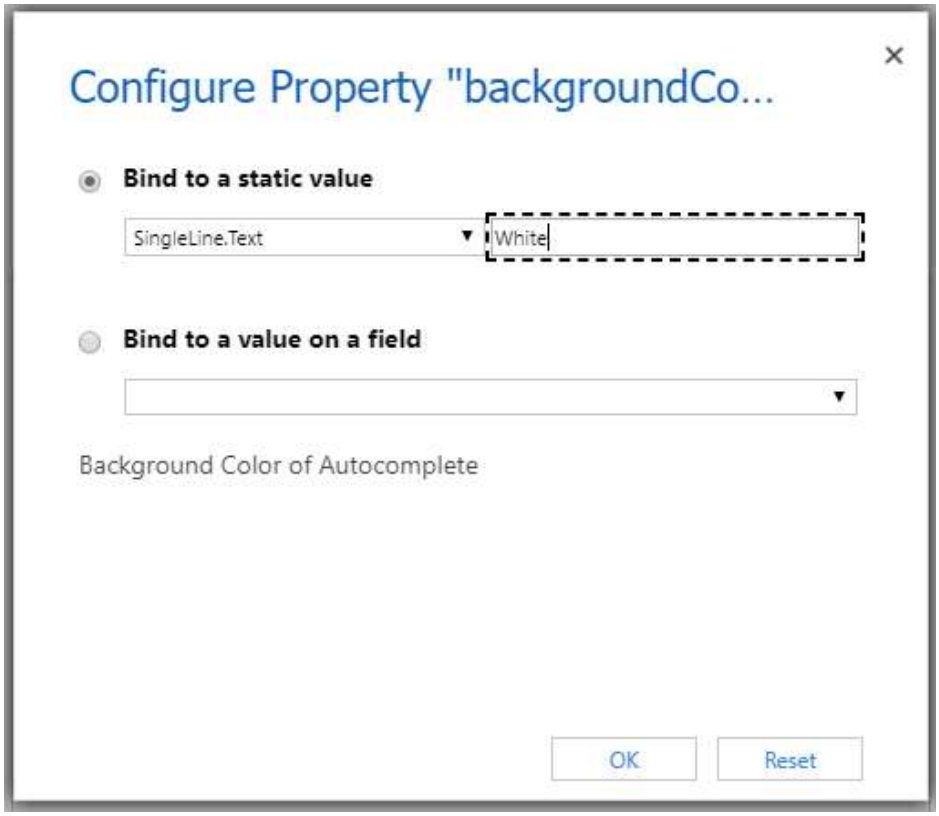
Select to use the control, and select the optionsValue:



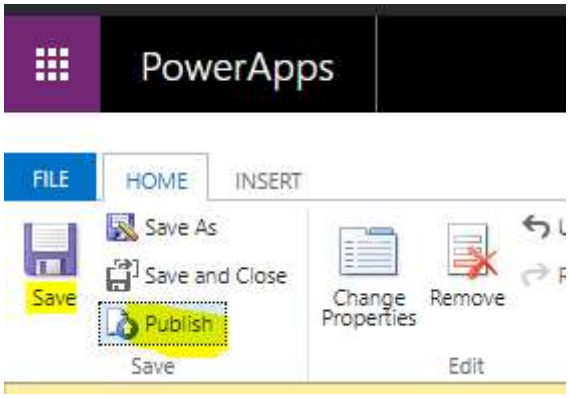
We will bind to static values, and enter in the countries that will be available for the user to select:



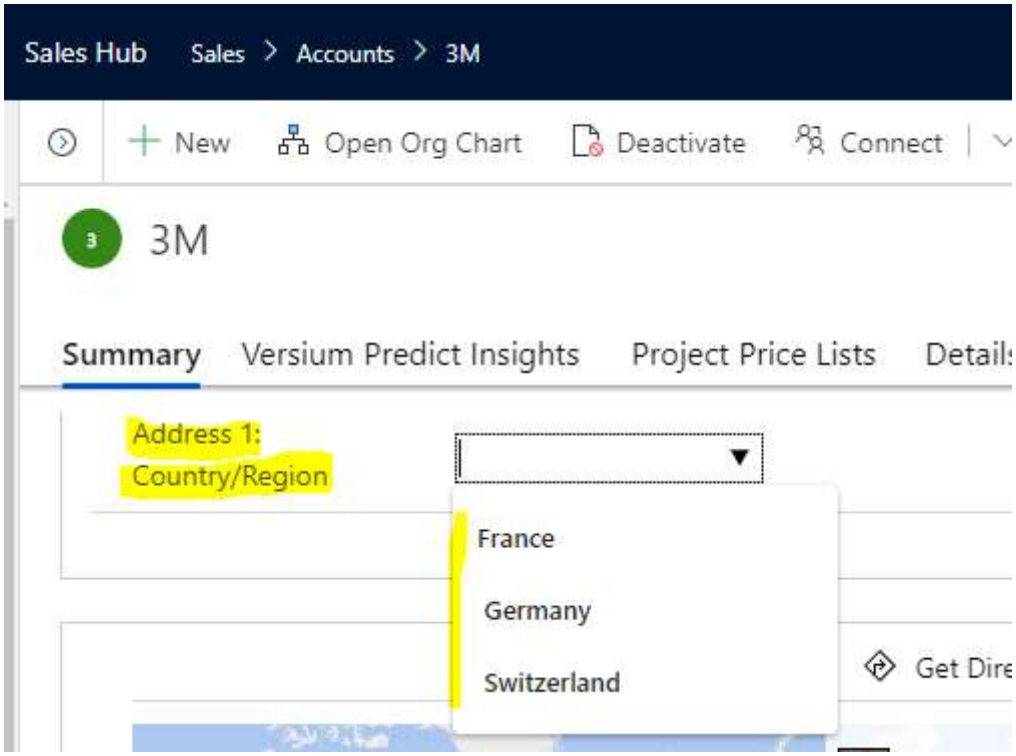
And also the background color:



Click OK, then Save and Publish the form:



Now open an Account. You will see the new control:



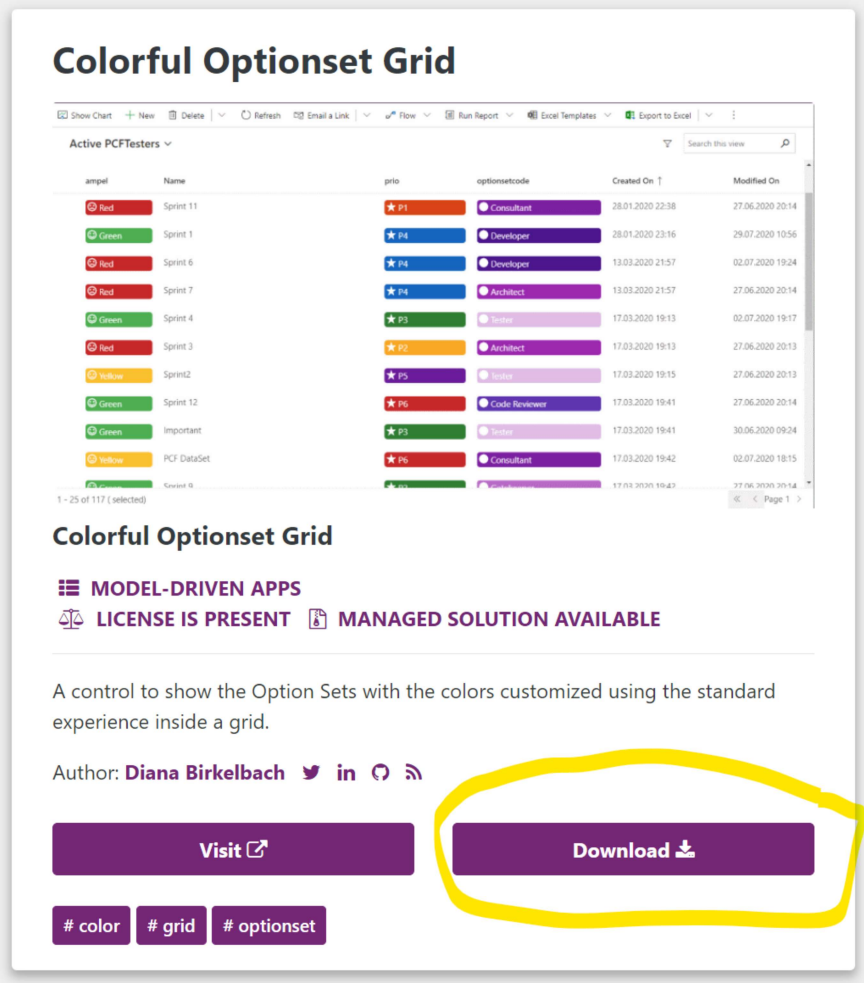
And when typing it will auto-complete:

Now, in some cases, the PCF.Gallery component will not contain a ZIP solution. We will look at how to install these in a future post.

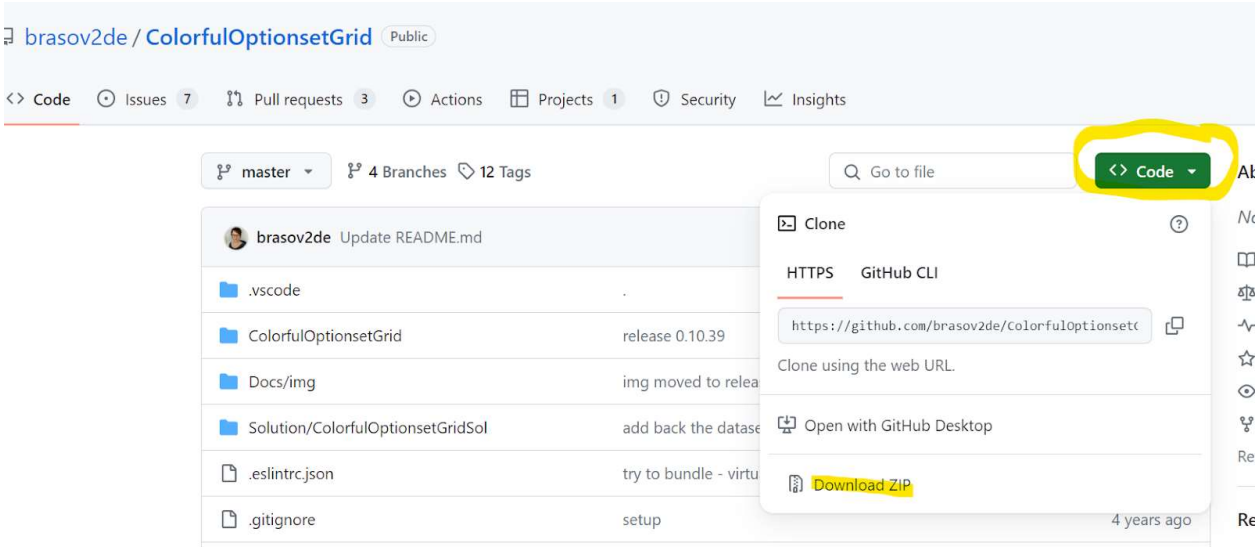
Installing PCF.Gallery Components When No Solution.zip Exists

Previously, we looked at how to use the PCF.Gallery site to install custom PCF components when a solution has been provided by the developer in GitHub. In some cases, the solution.zip file may not exist. Let's look at how to deploy these components to an org so we can use them. Note you will need to go through these instructions here to set up your PowerApps Component Framework environment:

Let's look at the Colorful Optionset Grid by Diana Birkelbach. Click on Download:

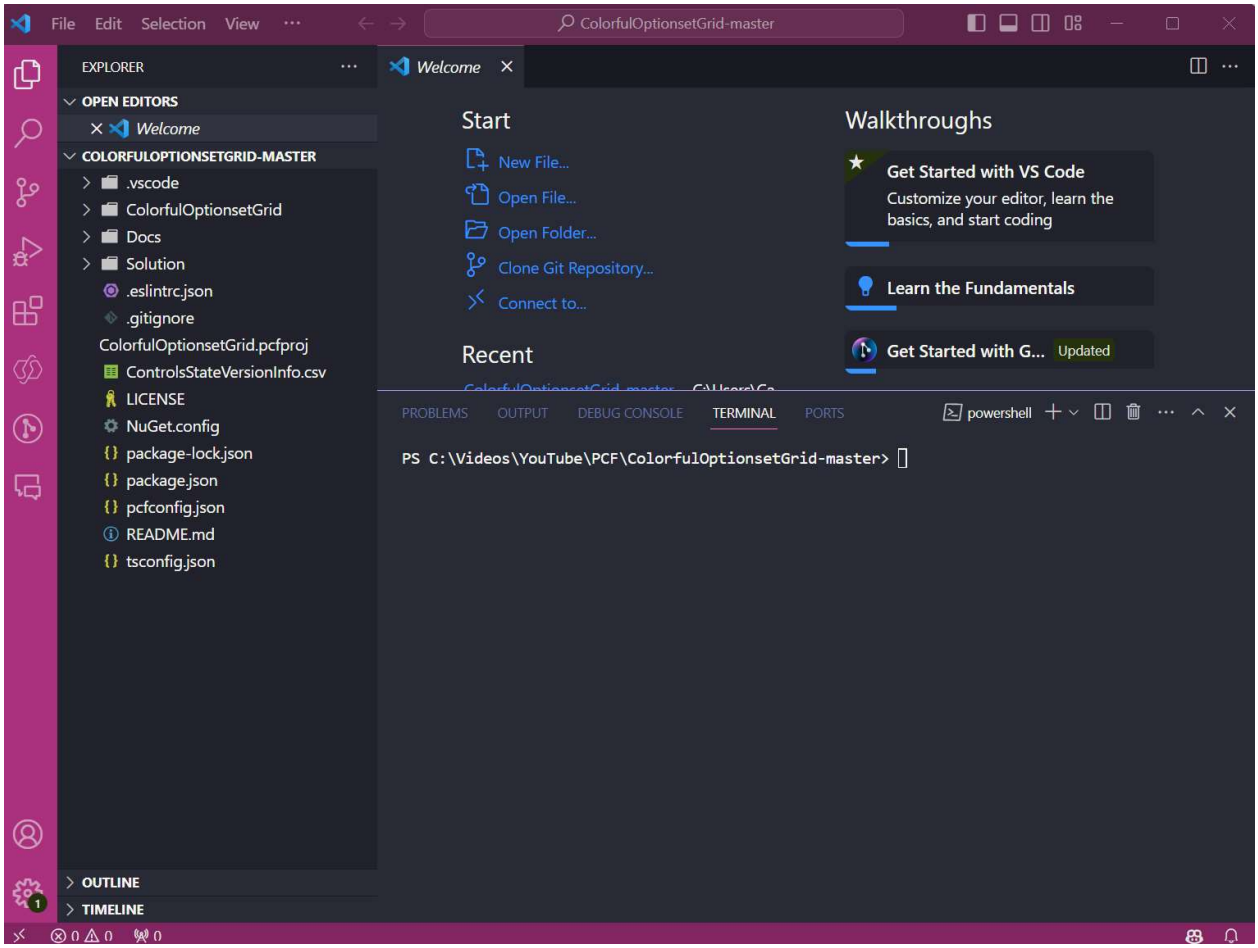


This takes us to the GitHub repo for the control. select Code->Download ZIP (or you can clone this project):



This downloads the PCF Controls:

Unblock the ZIP if you need to, then extract the files and copy to somewhere you can build your components, now open VS code to the YouTube directory, and open the terminal (Diana has such attention to detail she even has a .vscode setting to set the colors of the project!):



Now run:

```
npm install
```

```
PS C:\Videos\YouTube\PCF\ColorfulOptionsetGrid-master> npm install

added 593 packages, and audited 594 packages in 49s

100 packages are looking for funding
  run `npm fund` for details

18 vulnerabilities (12 moderate, 5 high, 1 critical)

To address all issues, run:
  npm audit fix

Run `npm audit` for details.
PS C:\Videos\YouTube\PCF\ColorfulOptionsetGrid-master>
```

Now you can deploy to your org using one of a few ways.

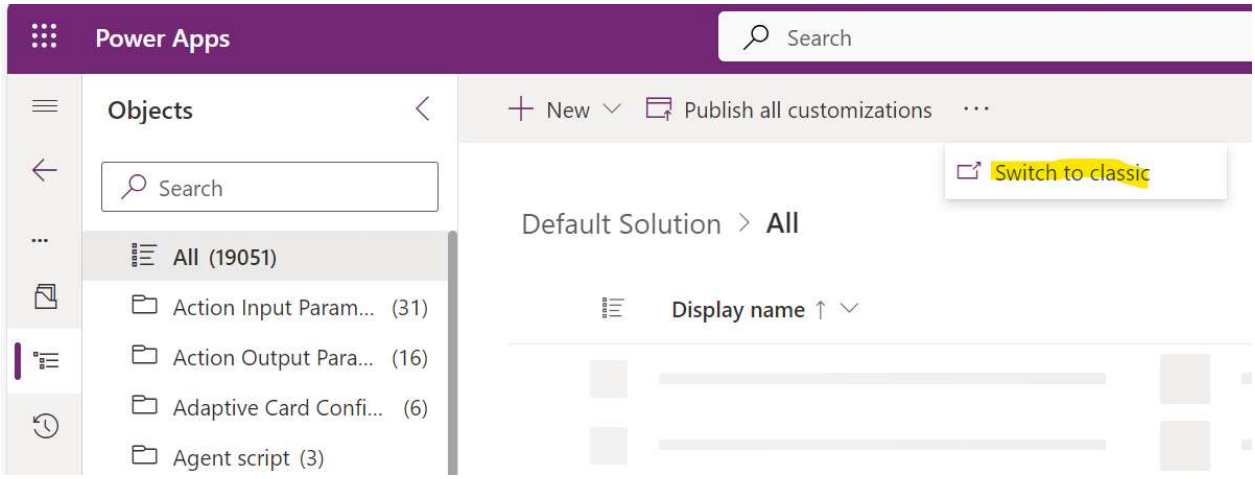
You can deploy via solutions (managed or unmanaged), or you can deploy using:

```
pac pcf push --publisher-prefix <publisher prefix>
```

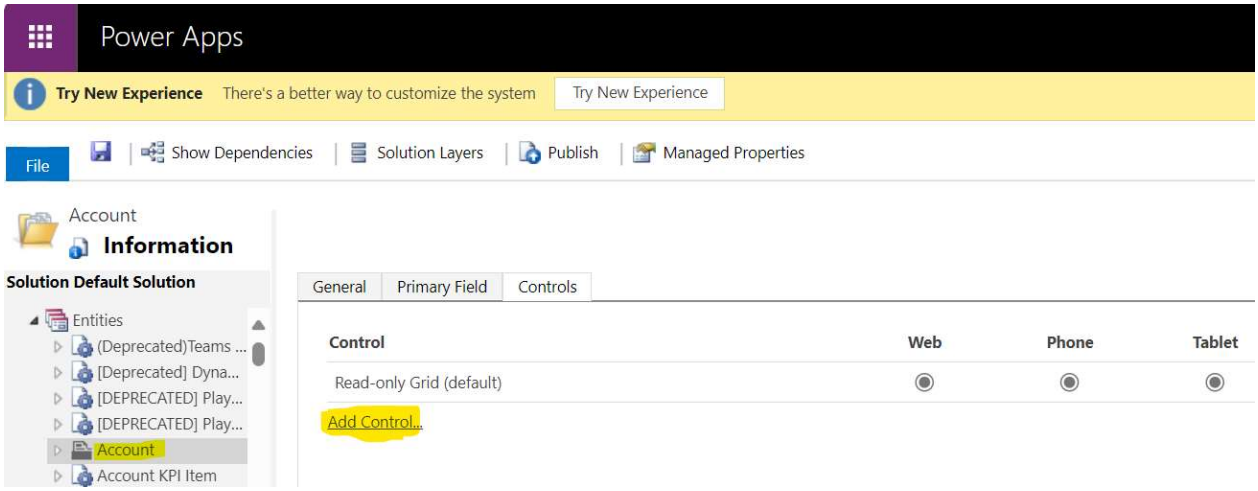
We will deploy using pac pcf push:

```
Solution Importing...

Solution Imported successfully.
Importing the temporary solution wrapper into the current org: done.

Publishing All Customizations...

Published All Customizations.
Updating the control in the current org: done.
```

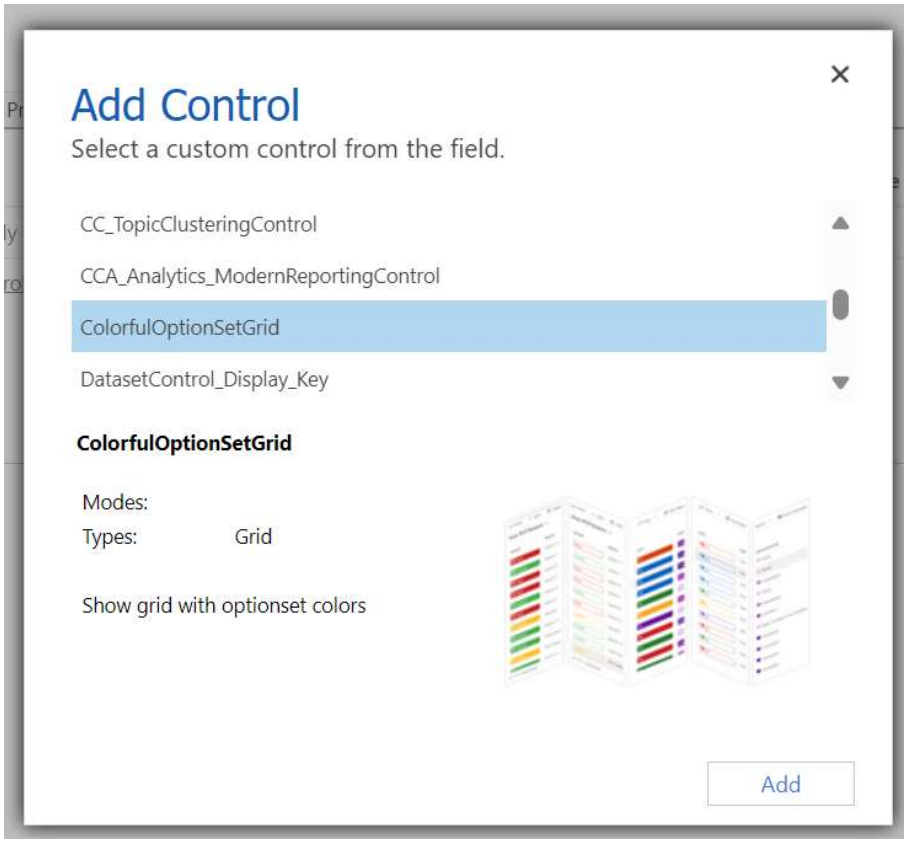
Now, when we go into our org, we can bind this to a dataset view, in our case we will bind it to an Account view. Let's open the Account table in the classic interface:



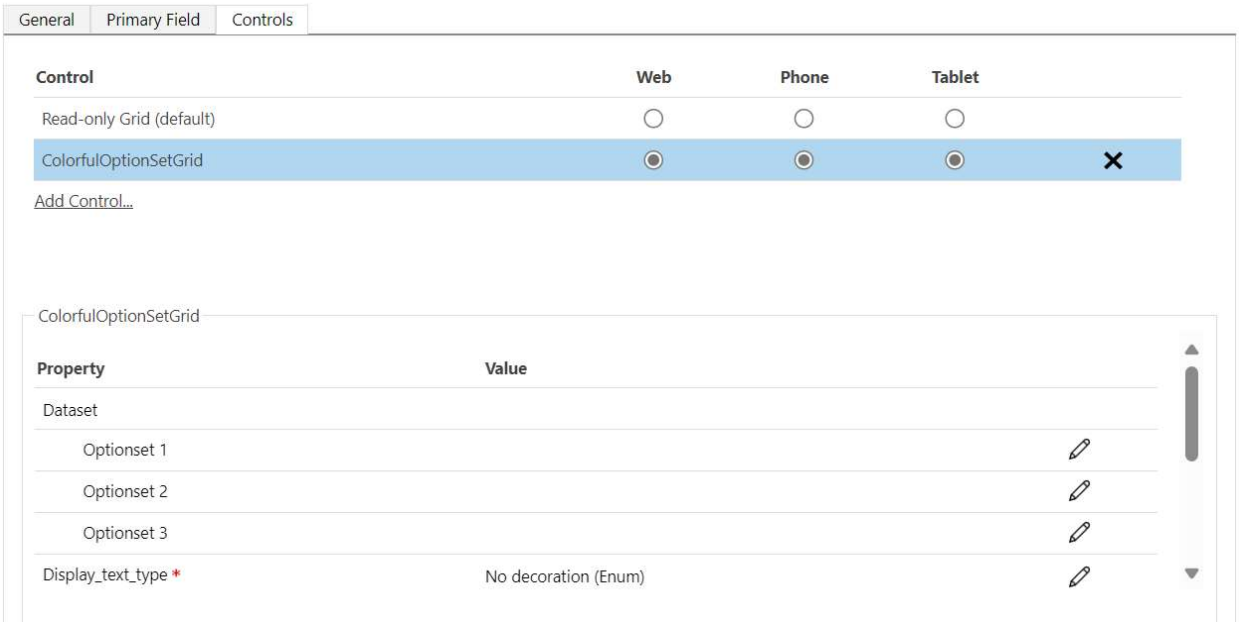
And we will add control:



And let's select the ColorfulOptionSetGrid:



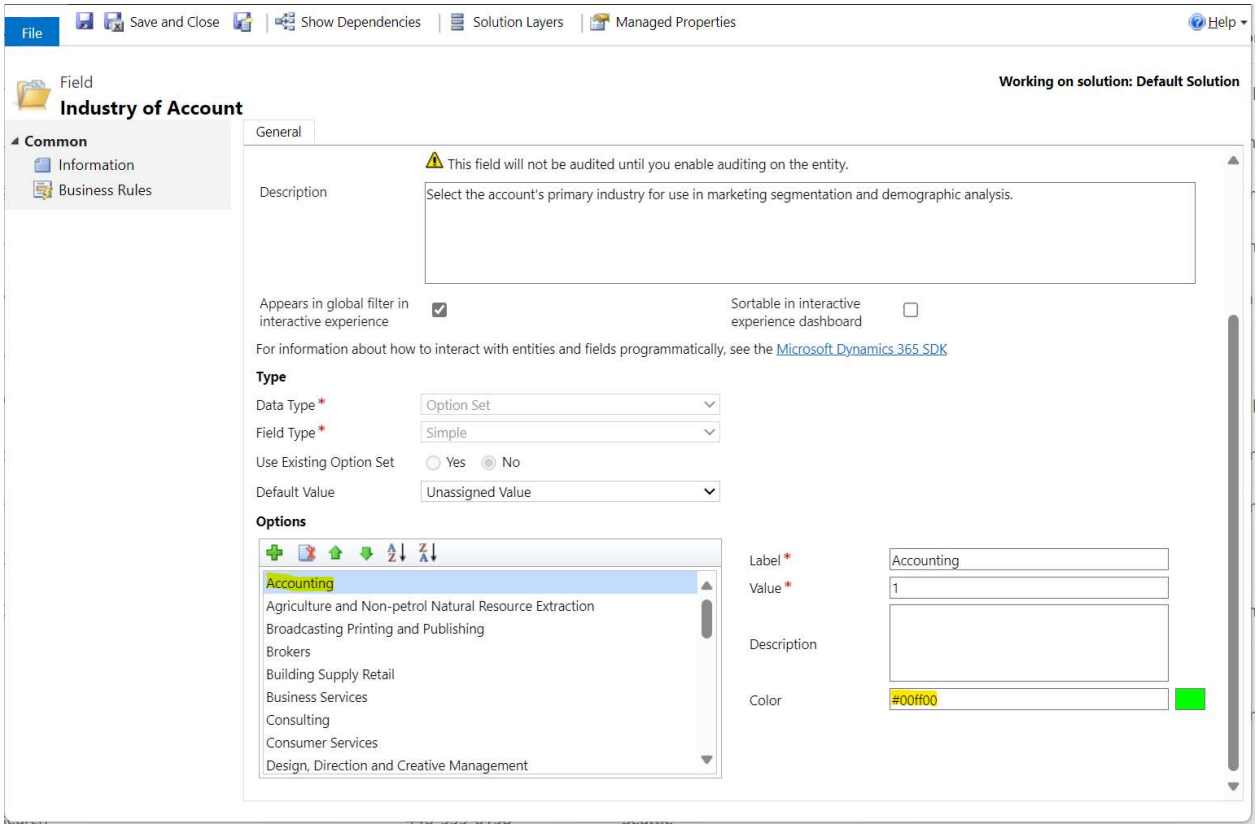
Now we need to set where this control will be displayed, and then set any properties required:



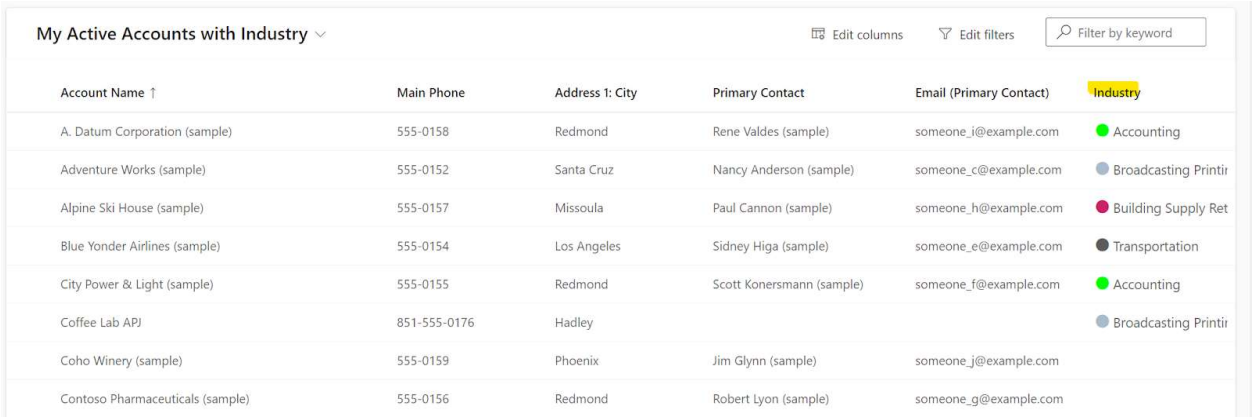
Scrolling down we see many more items we can customize:

If I edit Optionset 1, I can select a field that I want to apply this colorful option set to. I'm going to select the Industry field:

Make sure the option set you use has colors defined for the options, and if not, add them:



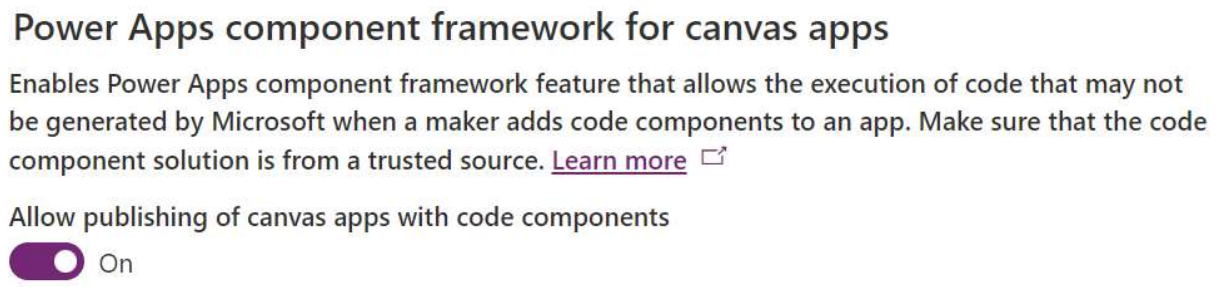
Let's save and publish, and see what this looks like. In My Accounts view, I have the Industry field, and we can see the colorful option set is displayed!



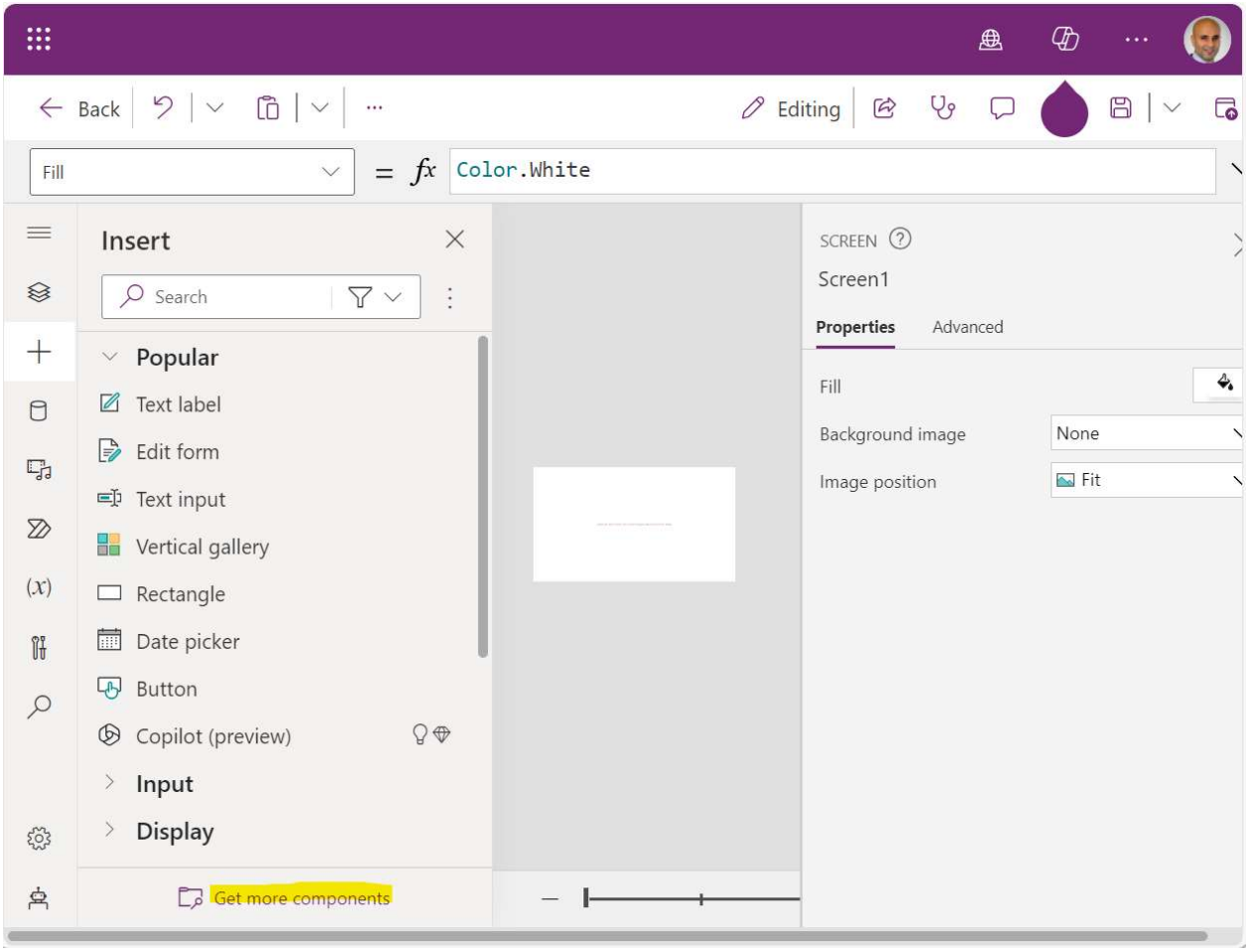
Using PCF Controls in Canvas Apps and Custom Pages

Previously, we created some Power Apps Component Framework (PCF) controls for use in Model-Driven Apps. Let's look at how to use PCF controls in Canvas Apps and Custom Pages.

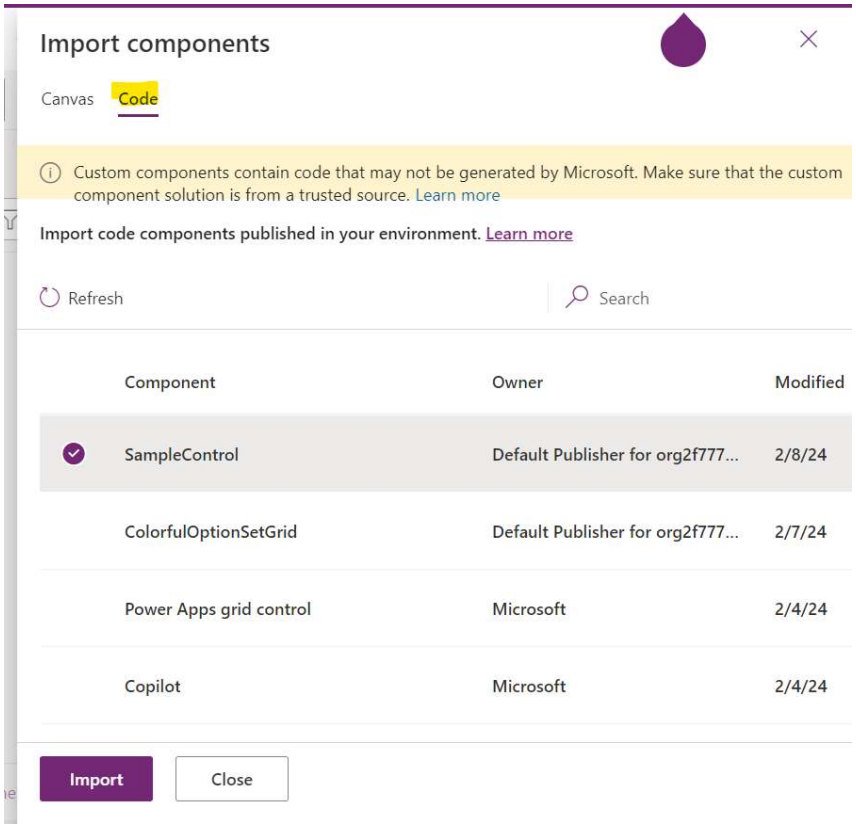
First, let's turn on a feature that allows us to use custom components in Canvas Apps. In the Power Platform Admin Center (https://admin.powerplatform.microsoft.com), select your environment, then **Settings->Features** and find the setting **Power Apps component framework for canvas apps** and turn it on (don't forget to Save in the bottom right). You will need this for both canvas apps and custom pages.



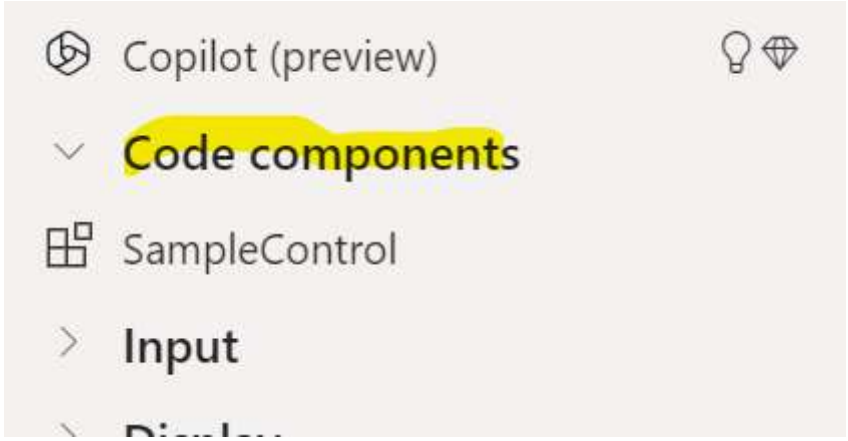
Now let's open a canvas app and click on the insert (+) option, then **Get More Components**:



Go to the Code tab and select a PCF control:



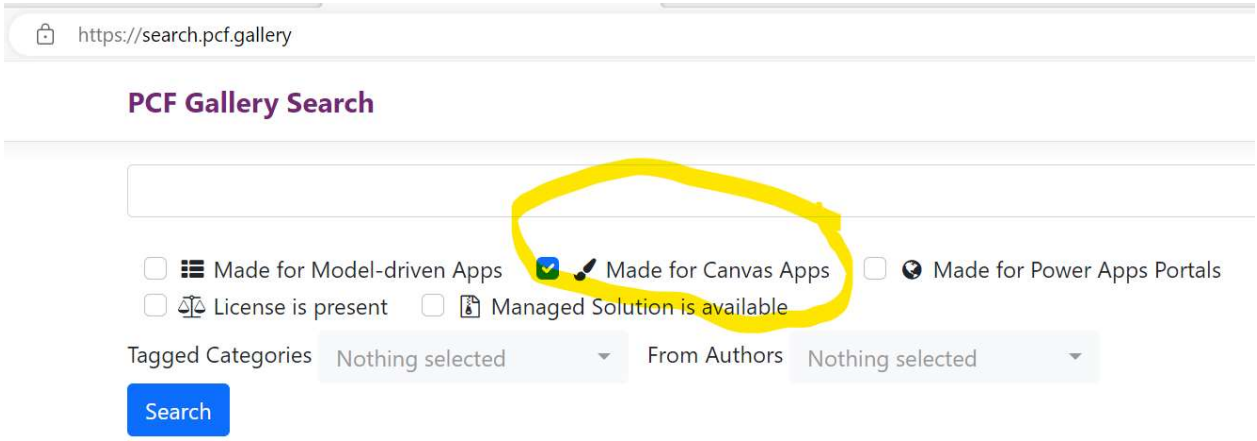
We will now see Code Components on the left, and expanding this we see our SampleControl:



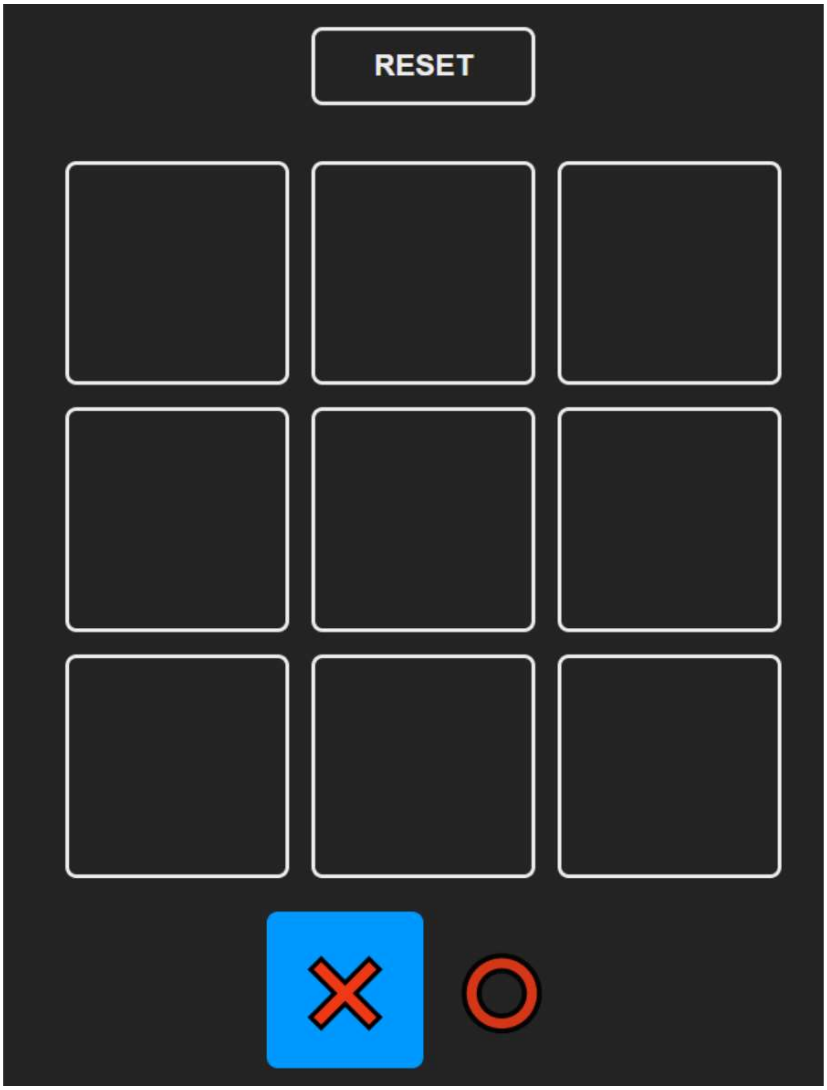
We can drag the field onto the canvas, and set properties, in this case the Property_Display_Key:

We can now use our PCF component in a canvas app.

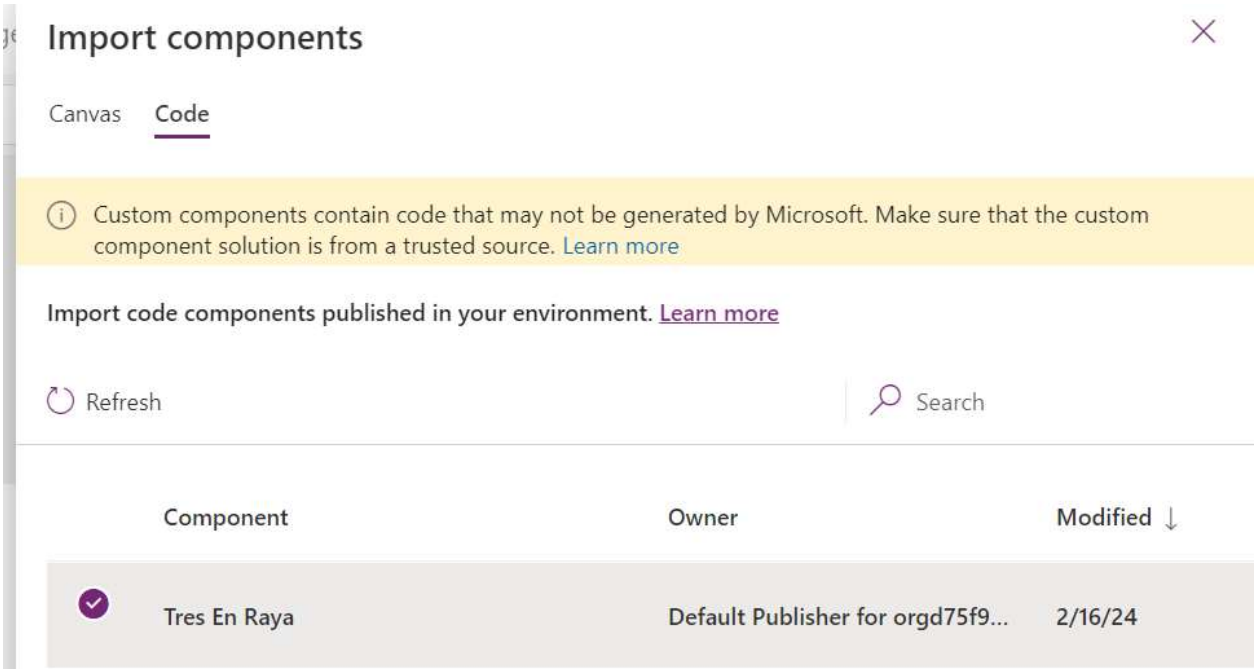
To find other controls for Canvas Apps, let's search in the https://pcf.gallery. Go to the search page and select Made for Canvas Apps:



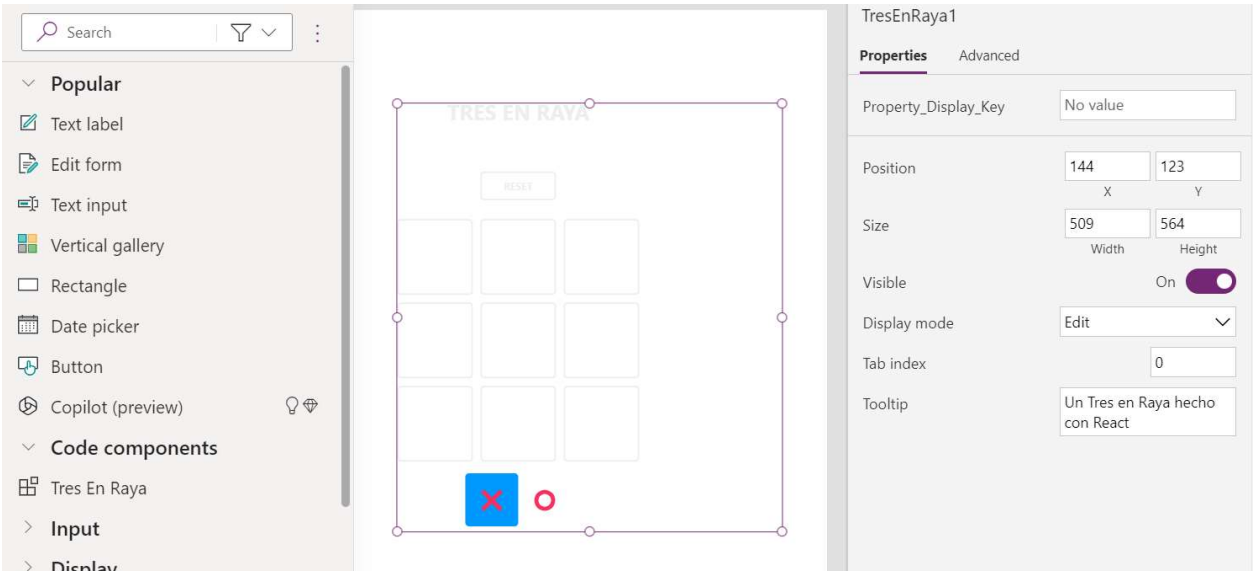
We will use the Tic Tac Toe control built by David Martínez Alcántara:



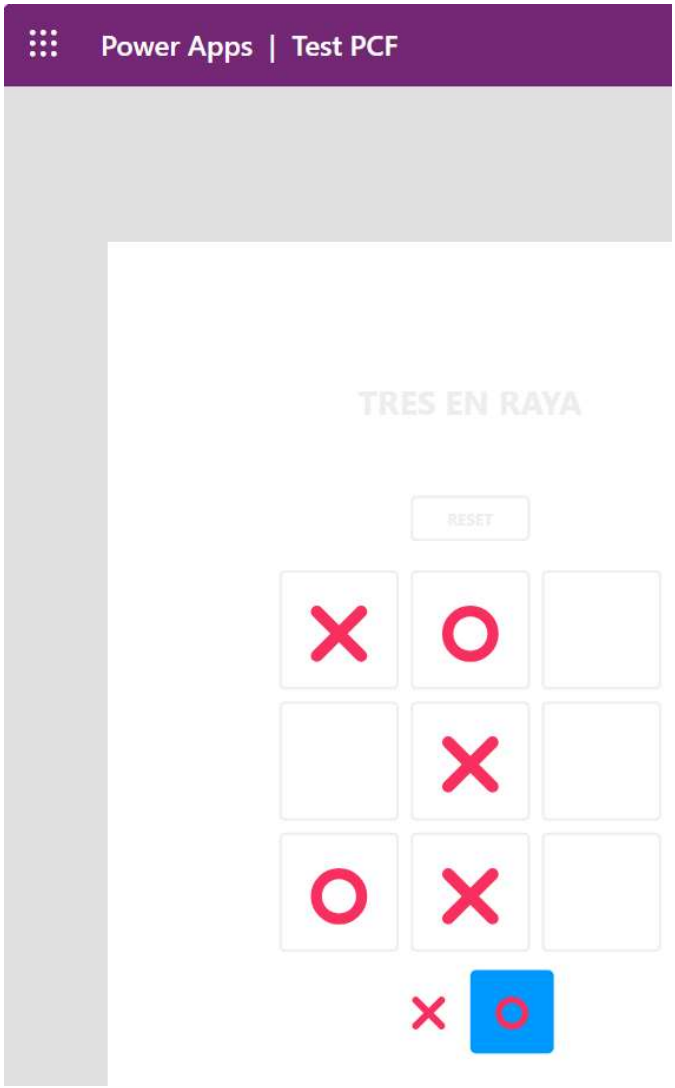
Upload and import the component:



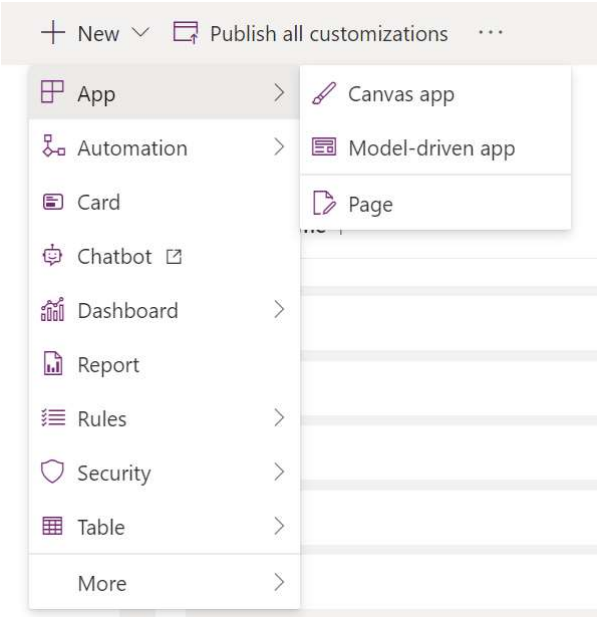
We can drag the control onto the canvas:



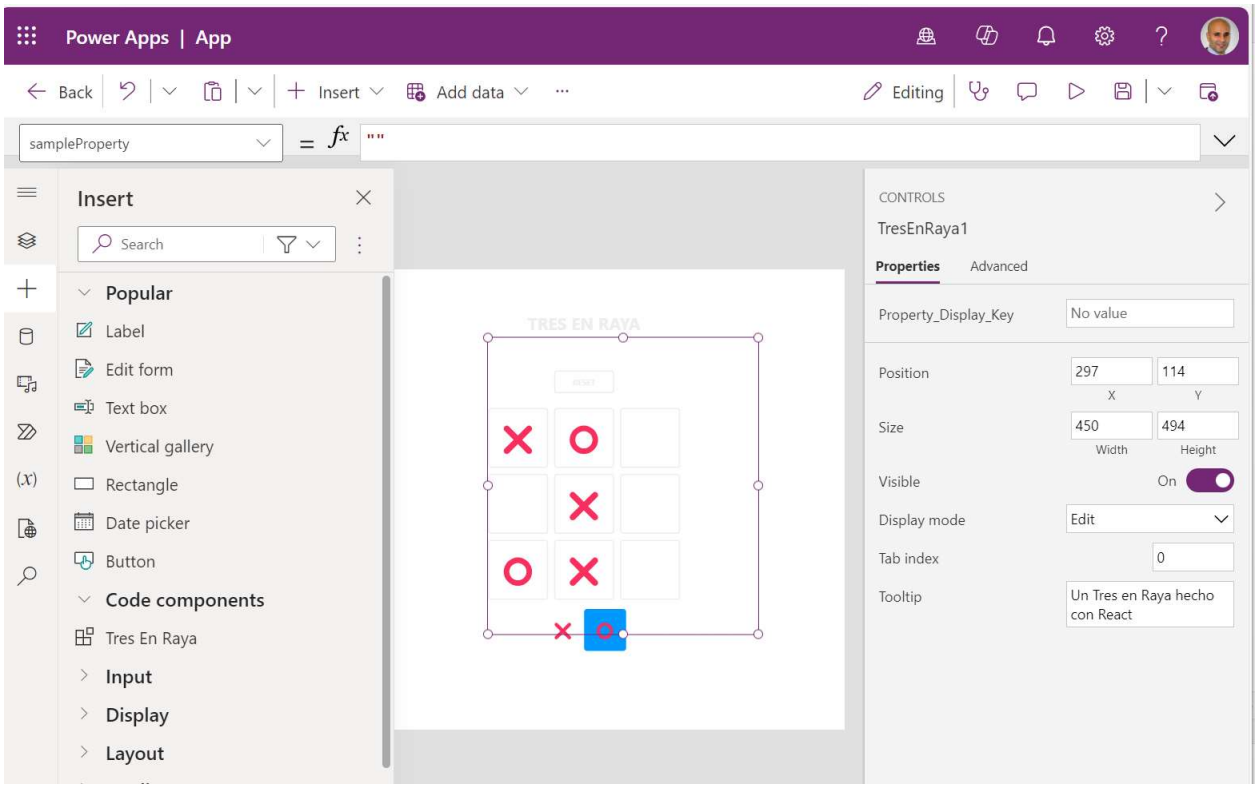
We can now use this PCF control inside our canvas app:



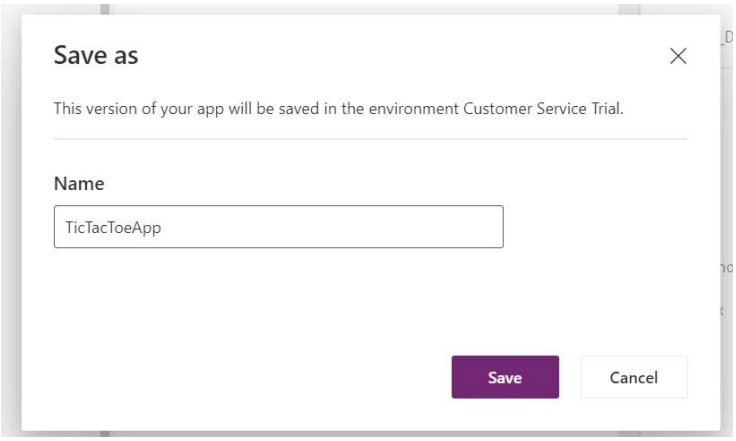
The same applies to custom pages. We can add a new App->Page:



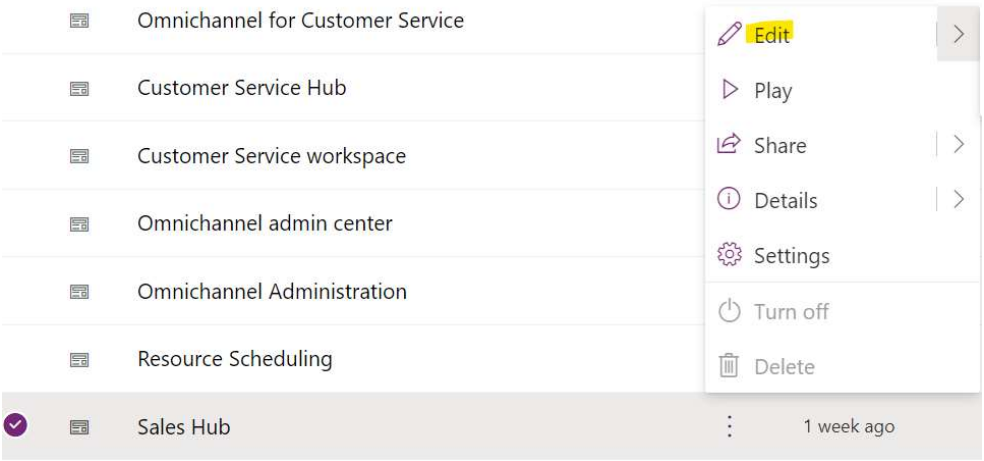
And import and add the new PCF control to the page. Let's add Tic Tac Toe to a custom page:



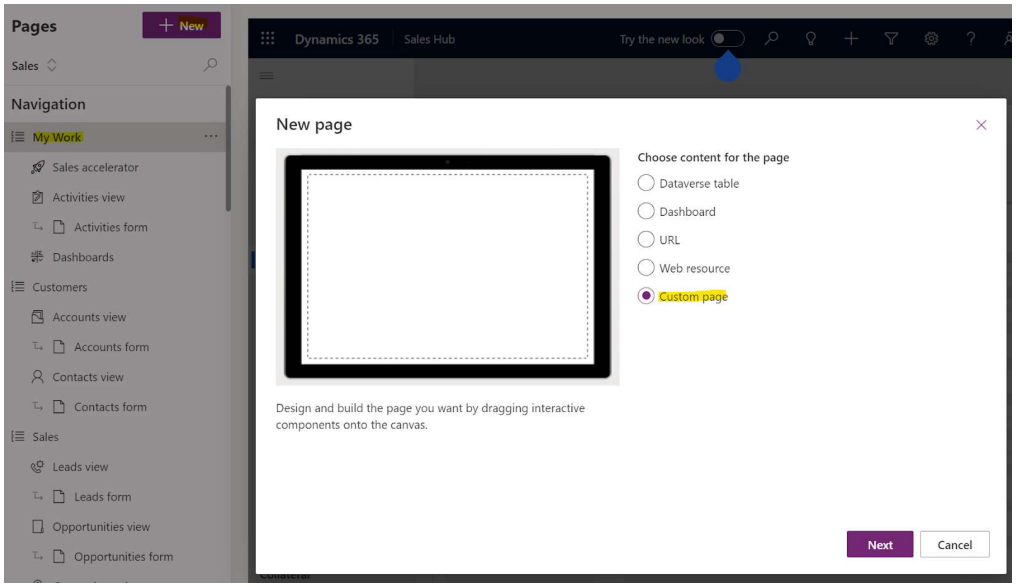
Save the app and publish it:



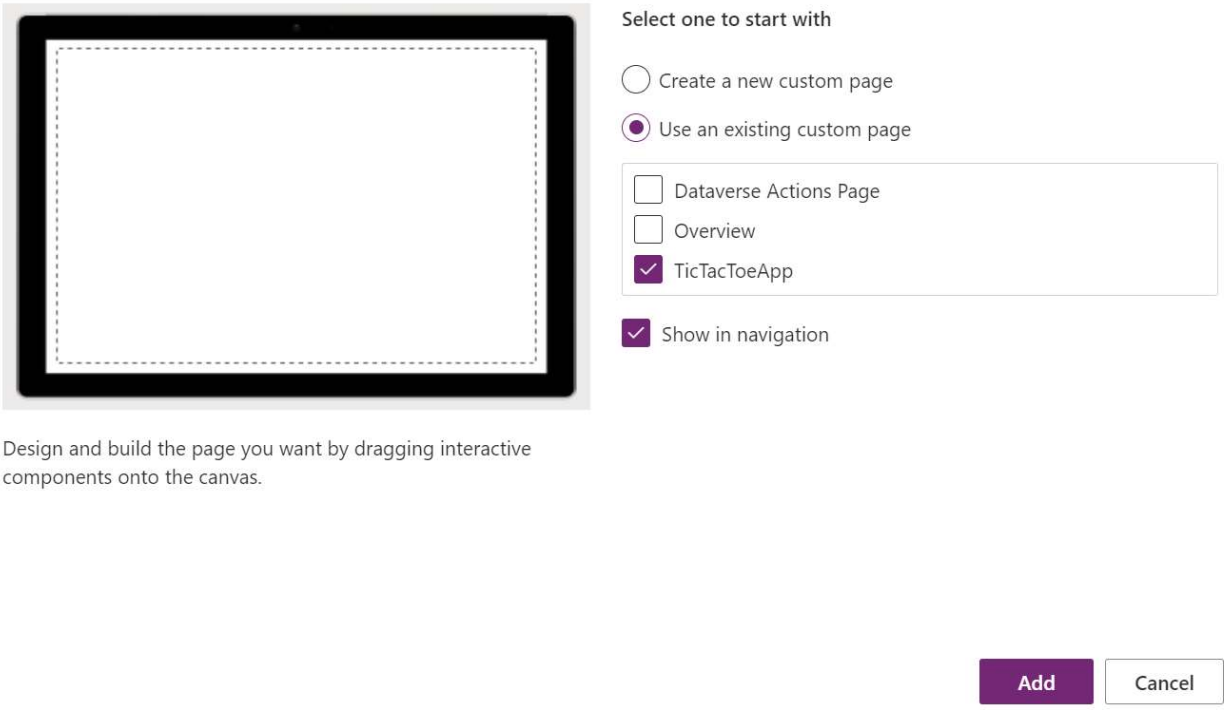
Now let's add the custom page to our model-driven app. We will add it to the site map. Let's edit the Sales Hub app:



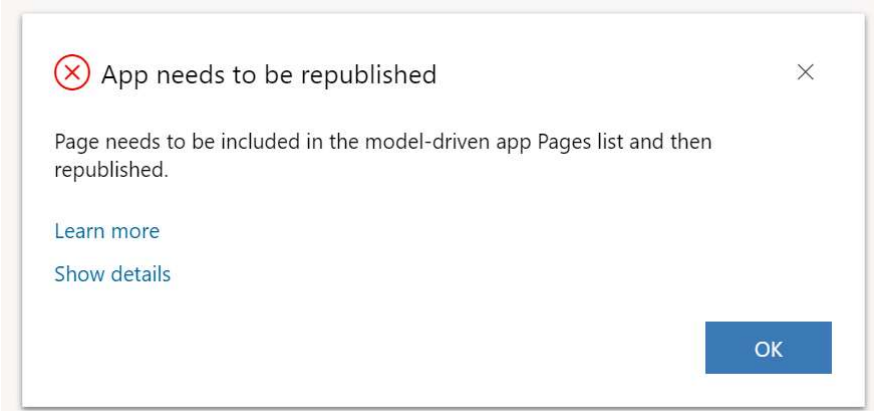
Under My Work, let's select New->Custom Page:



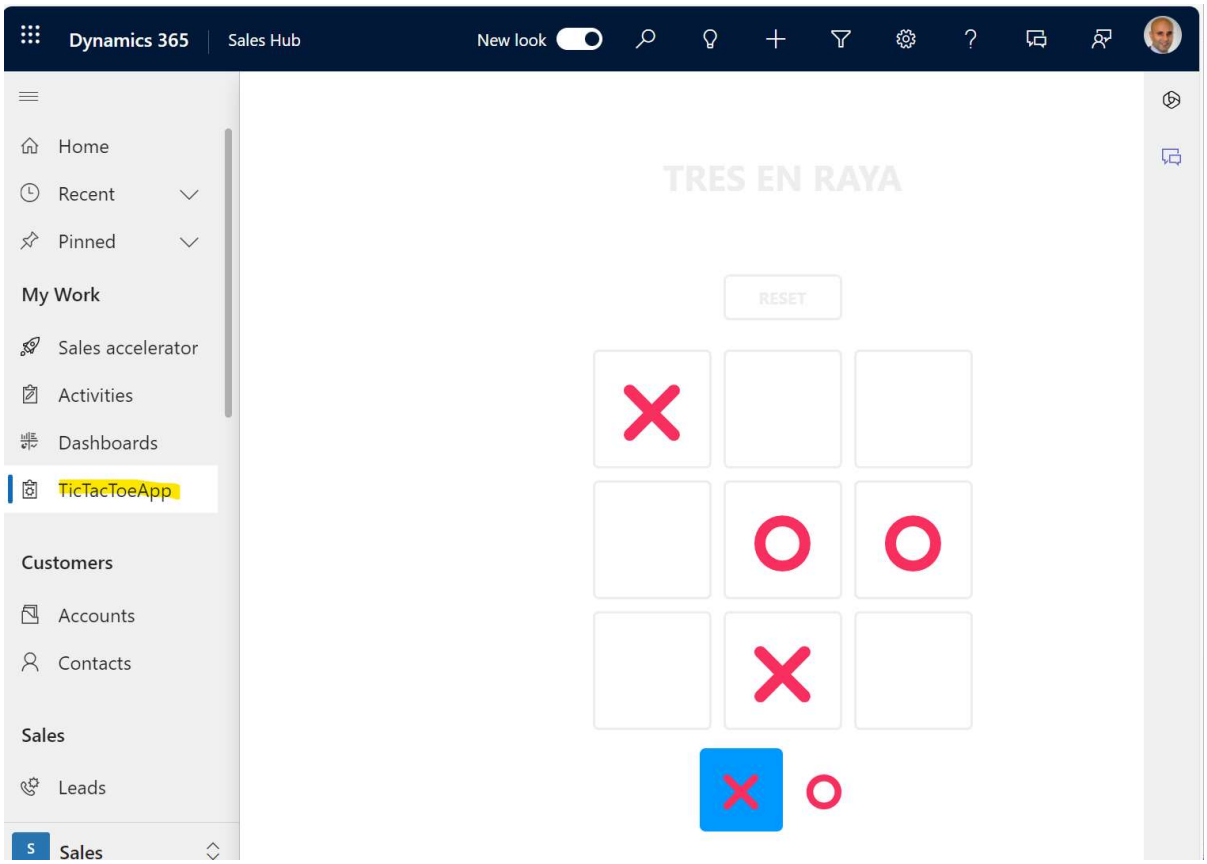
Select the Tic Tac Toe app and click Add:



You may see:



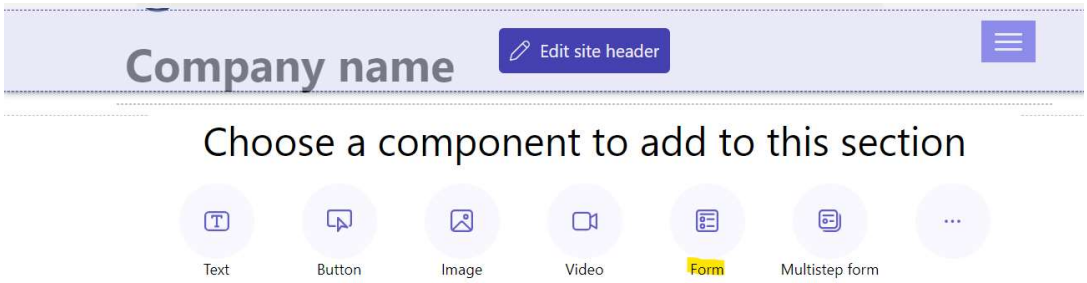
After saving and publishing, we see our custom page running our PCF control is available in the model-driven app:



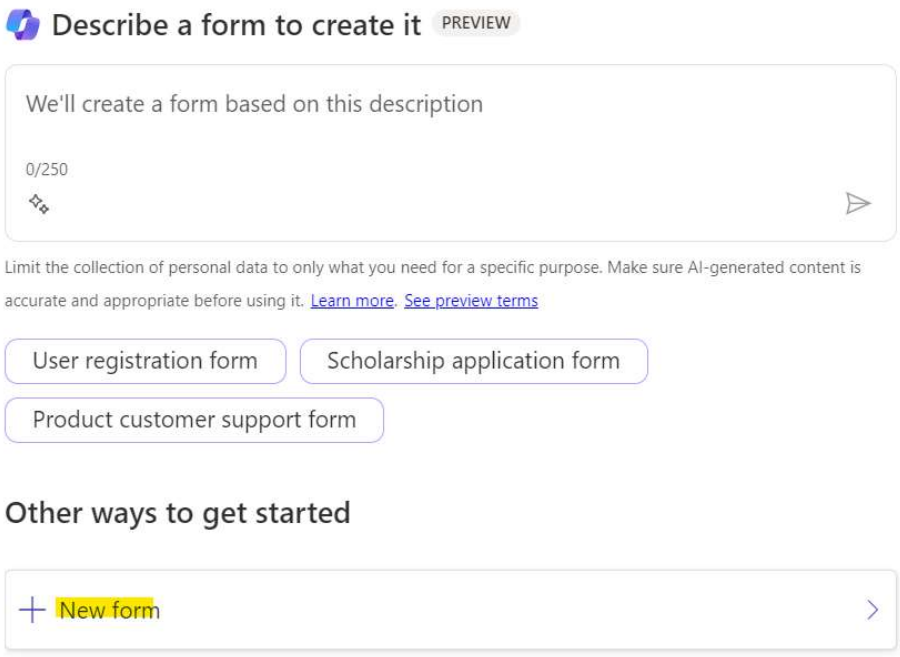
How to Use PCF Controls in Power Pages

We will now look at how PCF Controls can be used in Power Pages. This is continuing on in our series on PCF Controls.

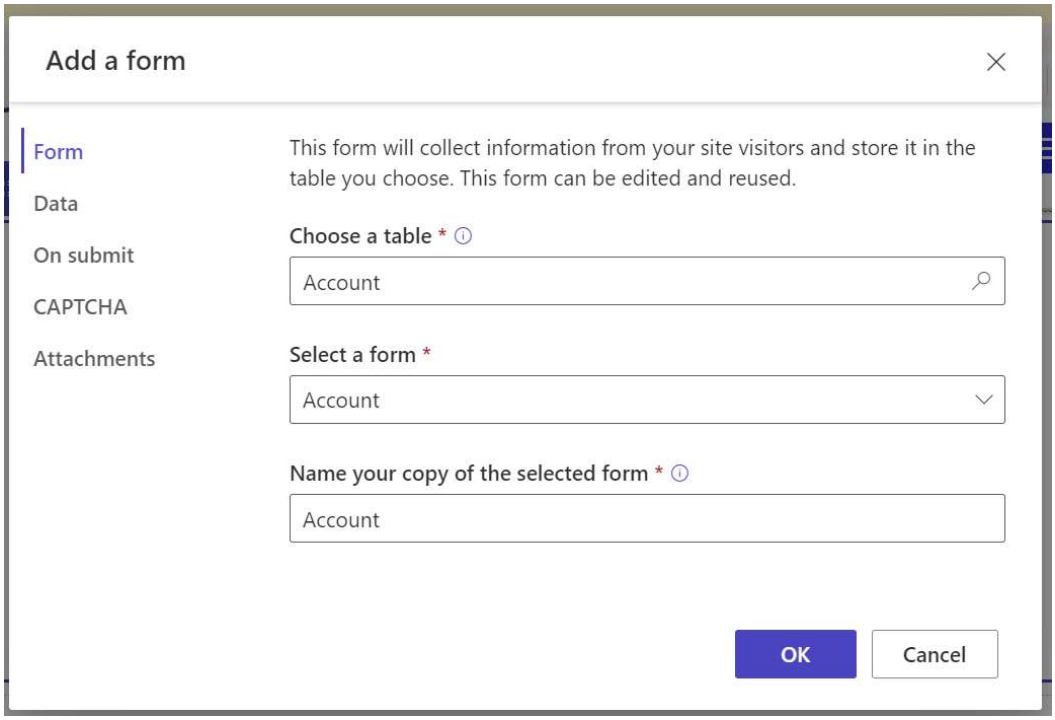
First, let's create a new Power Pages site, and we will add a form to the home page:



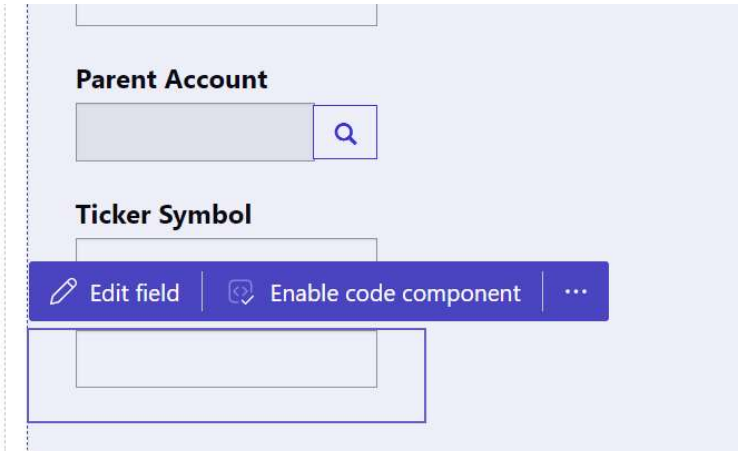
Select New Form:



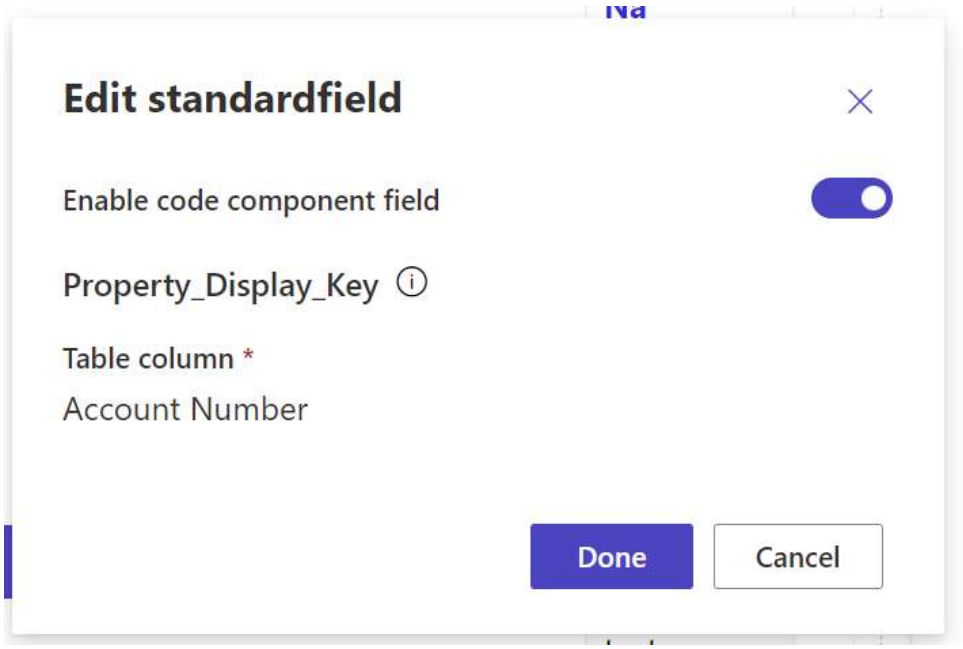
And we will select our Account Dataverse form:



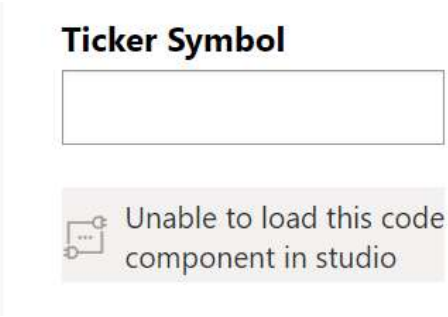
We see the field which contains our PCF code components, which was added previously to the form:



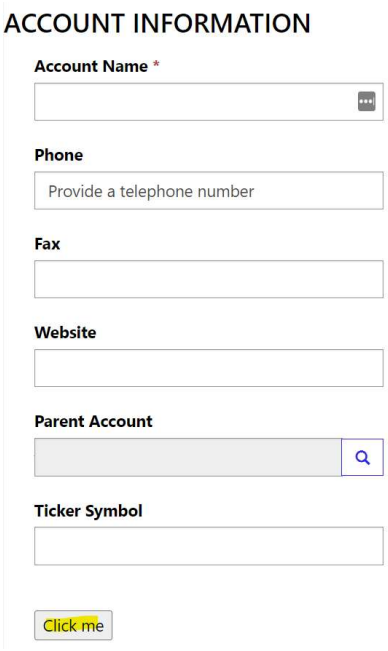
Click **Enable Code Component**, and then click Done:



We see the message “Unable to load this component in studio”:



For the purpose of the demo, we will make the site public. We can now see the component rendered on the form (the PCF control is a button):



There are restrictions in running PCF controls in Power Pages, which you can read about here.

Next Steps

Congratulations on completing this course!

If you haven't already, be sure to check out the video of this course located at https://youtu.be/897DPWMJQ20?feature=shared. My YouTube channel is located at https://www.youtube.com/@carldesouza and my blog can be found at https://carldesouza.com where you will find many more articles and videos to help you learn the Microsoft Power Platform.