

E-BOOK

# **BUILDING PCF CONTROLS FOR BEGINNERS**

*LEARN TO BUILD POWER APPS  
COMPONENT FRAMEWORK CONTROLS*



**CARL DE SOUZA**

Version: 1.0  
© Copyright 2024

<b>Introduction</b>	<b>4</b>
<b>PCF Create, Build, Deploy Cheat Sheet</b>	<b>5</b>
<b>Building and Deploying Your First PCF Control (without React)</b>	<b>8</b>
<b>Creating a TextBox PCF Control</b>	<b>23</b>
<b>Using Context Navigation and User Settings in PCF Controls</b>	<b>31</b>
<b>Building a PCF Control that Calls Context WebApi</b>	<b>37</b>
<b>Creating a Virtual PCF React Control with Fluent UI v8</b>	<b>43</b>
<b>Building a Dataset PCF Control with Styling (without React)</b>	<b>57</b>
<b>How to Install Power Apps Components from PCF.Gallery in your Org</b>	<b>73</b>
<b>Using PCF Controls in Canvas Apps and Custom Pages</b>	<b>91</b>
<b>How to Use PCF Controls in Power Pages</b>	<b>102</b>
<b>Next Steps</b>	<b>105</b>

# 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 do this 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:

Local Disk (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)
```

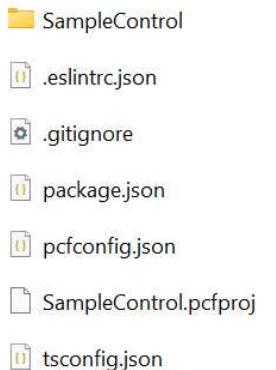
Let's create a control with the command:

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

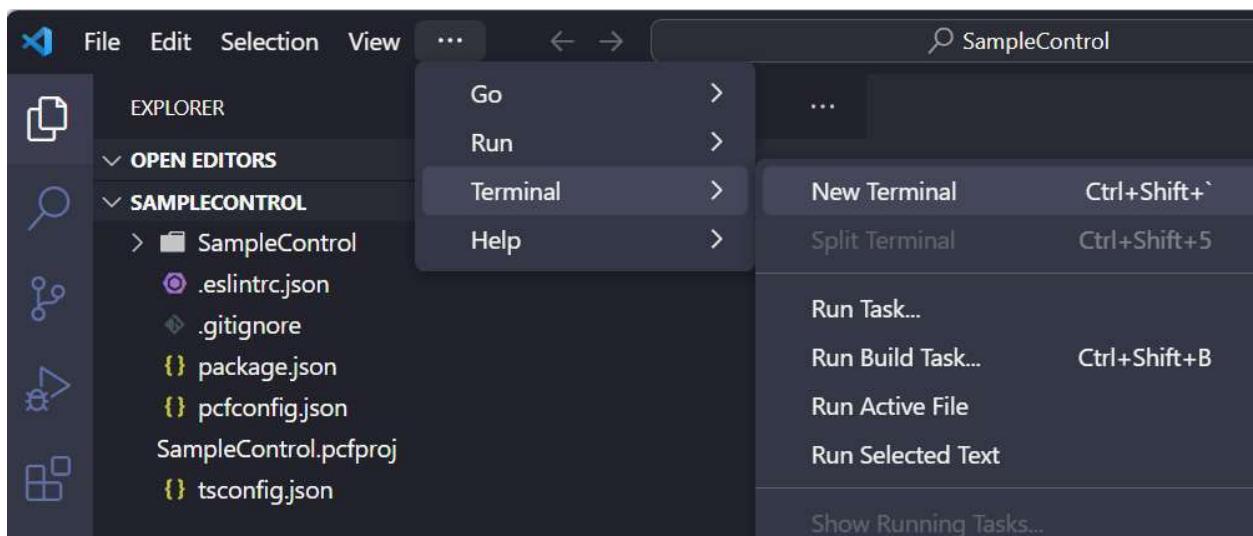
Note I am not using all the switches above, some will default and others are not required. Also note you can use the shortcuts to shorten the command (with a single dash instead of double), so the above command is the same as:

```
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:

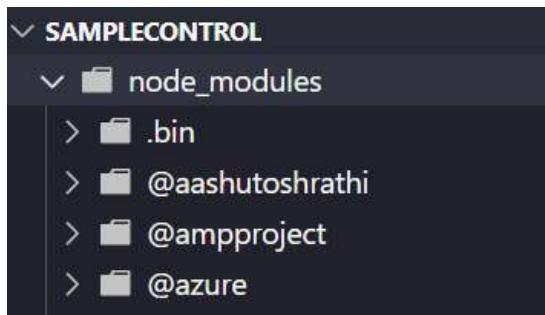
```
PS C:\PCF\SampleControl> npm install
npm WARN deprecated @babel/plugin-proposal-nullish-coalescing-operator@7.20.7: This is no longer maintained. Please use @babel/plugin-transform-object-rest-spread instead
npm WARN deprecated @babel/plugin-proposal-object-rest-spread@7.20.7: This is no longer maintained. Please use @babel/plugin-transform-object-rest-spread instead
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:

```

1 import {IInputs, IOOutputs} from "./generated/ManifestTypes";
2
3 export class SampleControl implements ComponentFramework.StandardControl<IInputs, IOOutputs> {
4
5   /**
6    * Empty constructor.
7    */
8   constructor() {
9   }
10
11 /**
12  * Used to initialize the control instance. Controls can kick off remote server calls and ot
13  * Data-set values are not initialized here, use updateView.
14  * @param context The entire property bag available to control via Context Object; It contain
15  * @param notifyOutputChanged A callback method to alert the framework that the control has
16  * @param state A piece of data that persists in one session for a single user. Can be set a
17  * @param container If a control is marked control-type='standard', it will receive an empty
18  */
19
20
21
22
23
24
25
26

```

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.

```

1  <?xml version="1.0" encoding="utf-8" ?>
2  <manifest>
3    <control namespace="car1" constructor="SampleControl" version="0.0.1" display-name-key="SampleControl" description-key="SampleControl_description" control-type="standard" >
4      <!--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
5      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.
6      Example1:
7        <external-service-usage enabled="true">
8          | <domain>www.Microsoft.com</domain>
9        </external-service-usage>
10     Example2:
11       <external-service-usage enabled="false">
12         | </external-service-usage>
13       -->
14       <external-service-usage enabled="false">
15         | <!--UNCOMMENT TO ADD EXTERNAL DOMAINS
16         | <domain></domain>
17         | <domain></domain>
18       -->
19       </external-service-usage>
20       <!-- property node identifies a specific, configurable piece of data that the control expects from CDS -->
21       <property name="sampleProperty" display-name-key="Property_Display_Key" description-key="Property_Desc_Key" of-type="SingleLine.Text" usage="bound" required="true" />
22       <!--
23       Property node's of-type attribute can be of-type-group attribute.
24       Example:
25         <type-group name="numbers">
26           <type>Whole_None</type>

```

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:

PowerApps component framework Test Environment

**SampleControl**

**Context Inputs**

Form Factor

Web

Component Container Width

Component Container Height

Only positive whole numbers allowed

Only positive whole numbers allowed

**Data Inputs**

Property

sampleProperty

Value

val

Type

SingleLine.Text

The form factor options are:

- Web
- Tablet
- Phone
- Unknown

**Context Inputs**

Form Factor

Web

Web  
Tablet  
Phone  
Unknown

Width      Height

And our sample property is listed below:

▽ **Data Inputs**

**Property**  
sampleProperty

**Value**

**Type**  
SingleLine.Text

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:

PowerApps component framework Test Environment

**SampleControl**

▽ **Context Inputs**

Form Factor

Component Container Width <input type="text" value="100"/>	Component Container Height <input type="text" value="100"/>
---	--

▽ **Data Inputs**

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":

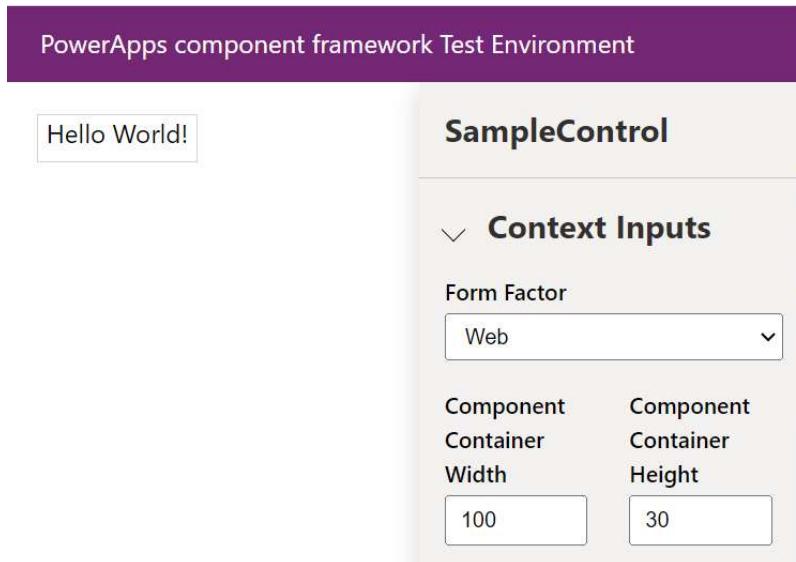
```
container.appendChild(document.createTextNode("Hello World!"));
```

```

    /**
     * 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 'setControlState'.
     * @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:

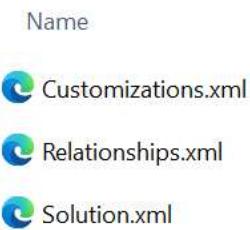
```
pac solution init --publisher-name Carl --publisher-prefix carl
```

```
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 carl.
Please verify the publisher information and solution name found in the Solution.xml file.
```

The following files are created:



SampleControl > Solution > src > Other



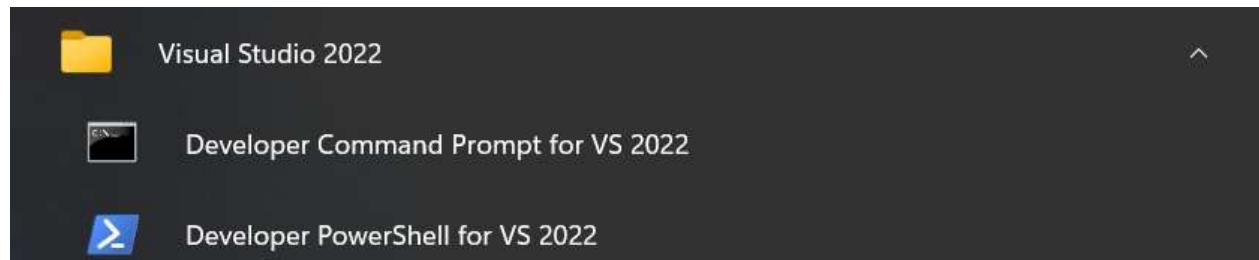
Next, we need to add references to where our component is located, in our case:

```
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
```

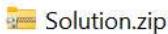
```
C:\PCF\SampleControl\Solution>msbuild /t:build /restore
MSBuild version 17.6.3+07e294721 for .NET Framework
Build started 11/30/2023 9:27:26 PM.

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:

SampleControl > Solution > bin > Debug

Name



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.com/
'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/
    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.csproj
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 Solution directory: obj\Debug\
Removing log file: obj\Debug\SolutionPackager.log and generated solution package: bin\Debug\SampleControl.sln
> 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:

The screenshot shows the Microsoft Power Apps 'Form' editor. On the left, there's a sidebar titled 'Table columns' containing a search bar and a list of columns: Account Name, Account Number, Account Rating, Address 1, Address 1: Address Type, Address 1: City, Address 1: Country/Region, Address 1: County, Address 1: Fax, Address 1: Freight Terms, Address 1: Latitude, Address 1: Longitude, Address 1: Name, Address 1: Post Office Box, Address 1: Primary Contact Name, Address 1: Shipping Method, and Address 1: State/Province. In the center, a 'New Account' form is displayed with tabs for Summary, Details, Scheduling, Files, Assets and Locations, and Related. The 'Summary' tab is selected, showing fields for Address 1: ZIP/Postal Code and Address 1: Country/Region. A modal window titled 'Add component' is open, listing various controls: ValidatedField\_Control, Canvas app, TalkingPointsControl, Rich Text Editor Control, TreeView Control, and Business card reader. The 'Account Name' field is highlighted with a yellow border. On the right, the 'Properties' panel is open, showing settings for the 'Account Name' column, including 'Label \*' (Account Name), 'Hide label', 'Hide on phone', 'Hide', 'Lock', and 'Read-only'. Below the properties are sections for 'Formatting' (Form field width: 1 column) and 'Components' (a list box with '+ Component').

We see our SampleControl and can Add it:

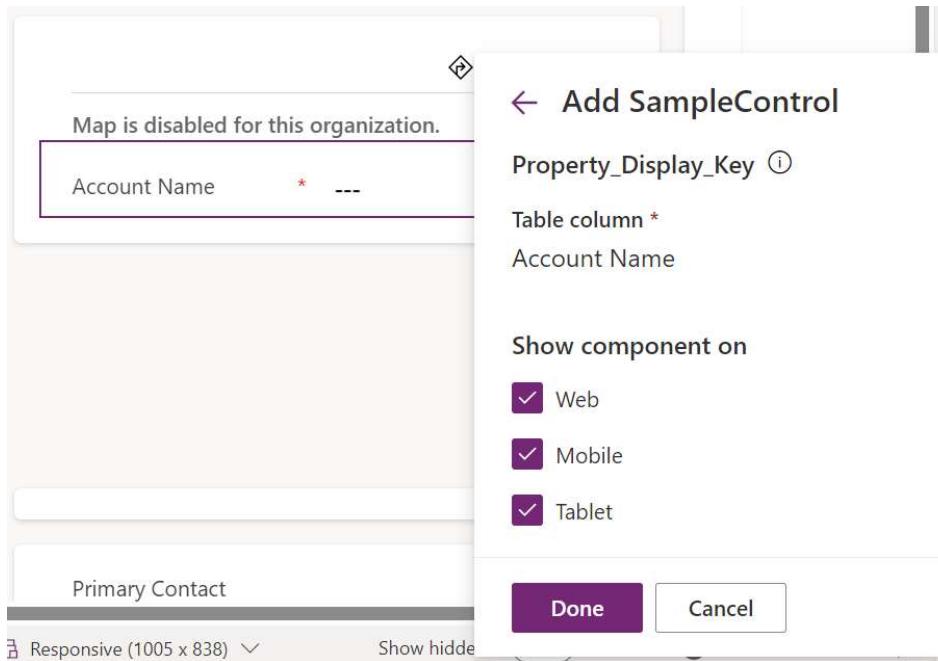
## Get more components

⟳ 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:

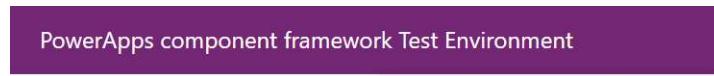
```
var x = document.createElement("INPUT");
x.setAttribute("type", "text");
x.setAttribute("value", "Hello World!");
container.appendChild(x);
```

```
public init(context: ComponentFramework.Context<IComponent>)
{
    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: Function): void
{
    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:

PowerApps component framework Test Environment

test21

## SampleControl

### Context Inputs

Form Factor  
Web

Component Container Width	Component Container Height
100	30

### Data Inputs

Property sampleProperty  
Value test21

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:

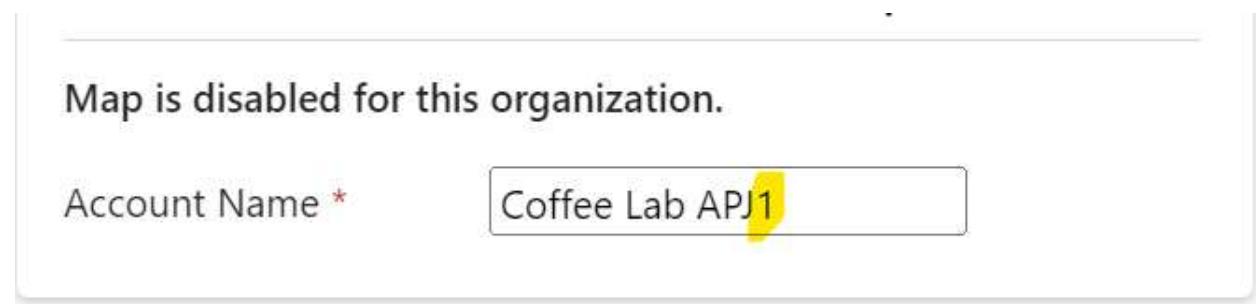
Address 1:  
Country/Region USA

Get Directions

Map is disabled for this organization.

Account Name \* Coffee Lab APJ

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: boolean) {
        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);
```

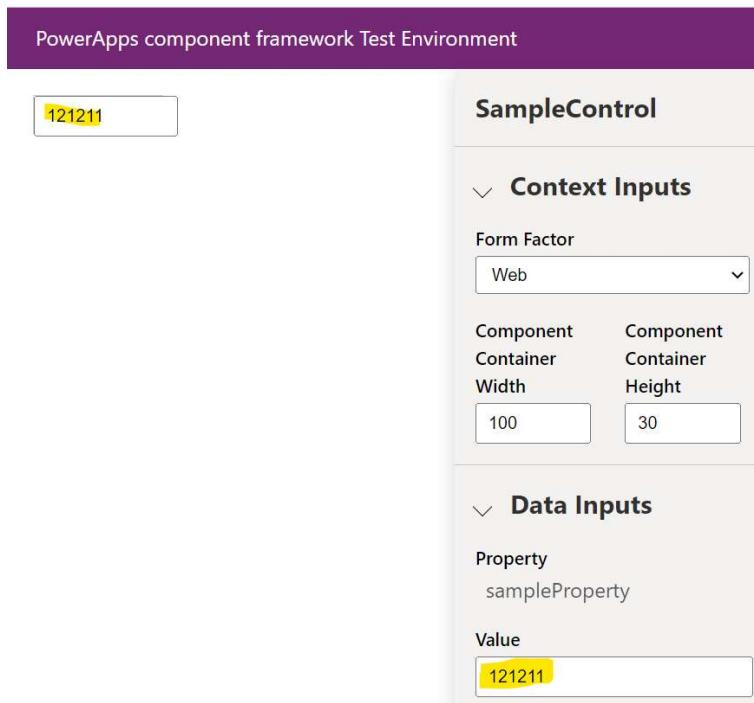
```

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:

Account Name \*

Coffee Lab APJ4

We see our custom PCF is updated:

Map is disabled for this organization.

Account Name \*

Coffee Lab APJ4

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<IIInputs>, notifyOutputChanged: () => void) {
    this._element = document.createElement("INPUT") as HTMLInputElement;
    this._element.setAttribute("type", "text");
    this._element.setAttribute("value", context.parameters.sampleProperty.raw || "");
    container.appendChild(this._element);

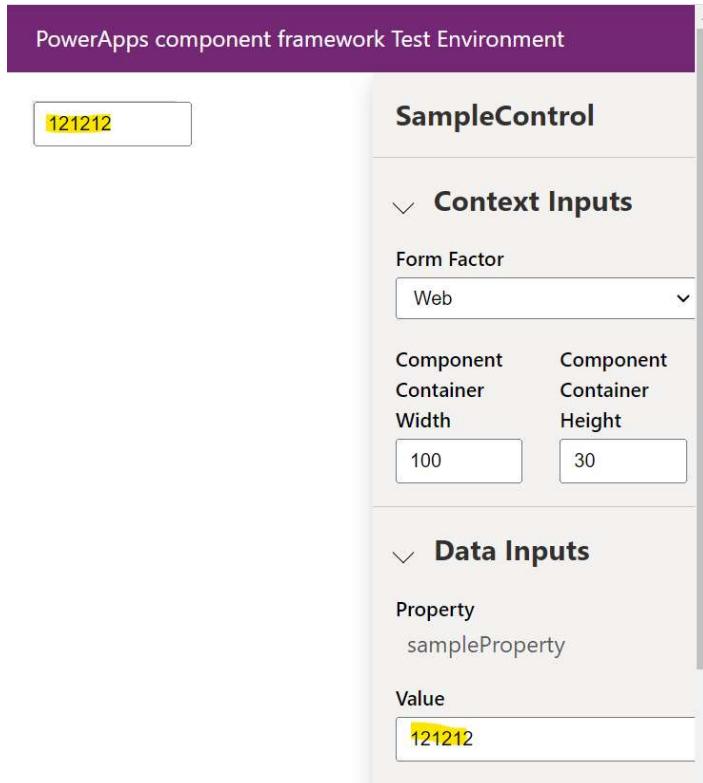
    this._element.addEventListener("change", (event) => {
        notifyOutputChanged();
    });
}
```

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:

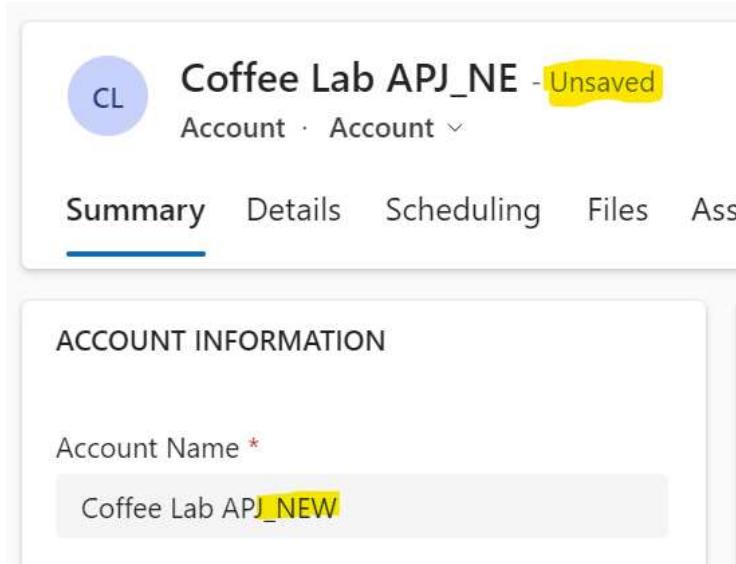


Let's publish this latest control to our org and see how it looks.

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:

Account Name \*

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:

## Get more components

⟳ Refresh

All    Built by Microsoft    Built by others

All components in the current environment. Not seeing a component

Component

- SampleXrmControl

← Add SampleXrmControl

Property\_Display\_Key ⓘ

Table column \*

Account Number

Show component on

- Web
- Mobile
- Tablet

**Done** **Cancel**

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

Ticker Symbol ---

Account Number	Click me
----------------	----------

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

 **Coffee Lab APJ** - Saved  
Account · Account ▾

Summary Details Scheduling Files Assets and Locat

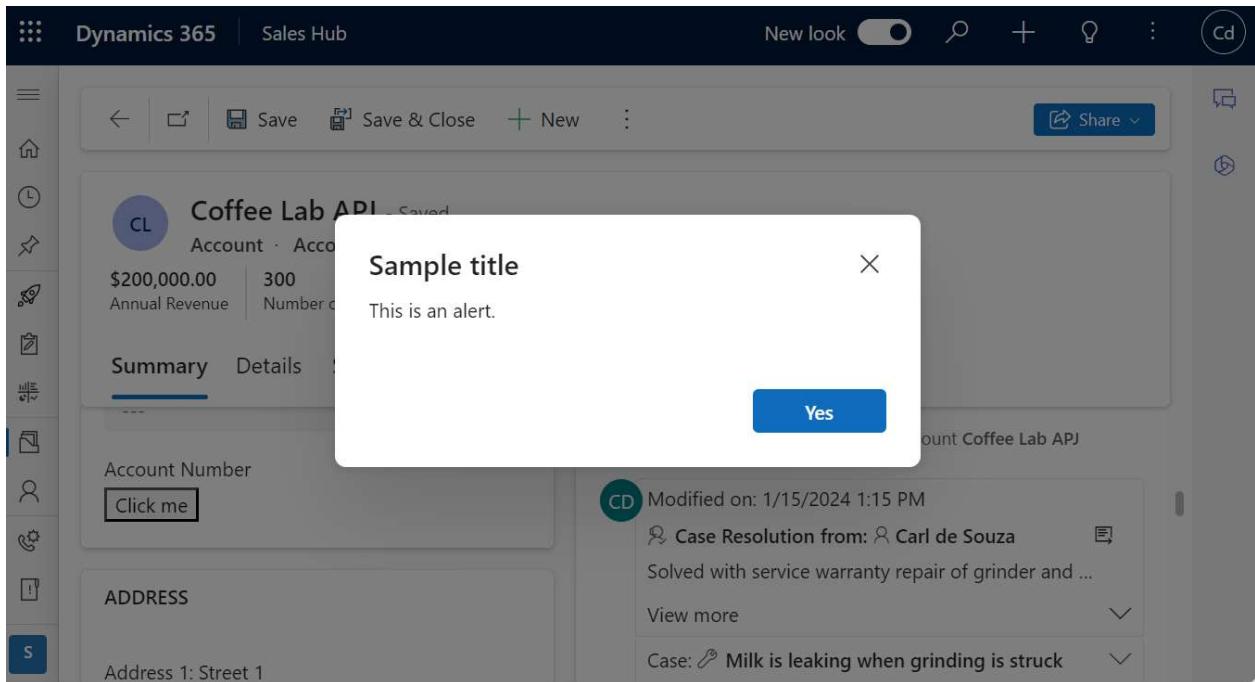
ACCOUNT INFORMATION

Account Name *	Coffee Lab APJ
Phone	851-555-0176
Fax	---
Website	<a href="https://www.coffeelab.com">https://www.coffeelab.com</a>
Parent Account	---
Ticker Symbol	---
Account Number	<b>Click me</b>

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
```

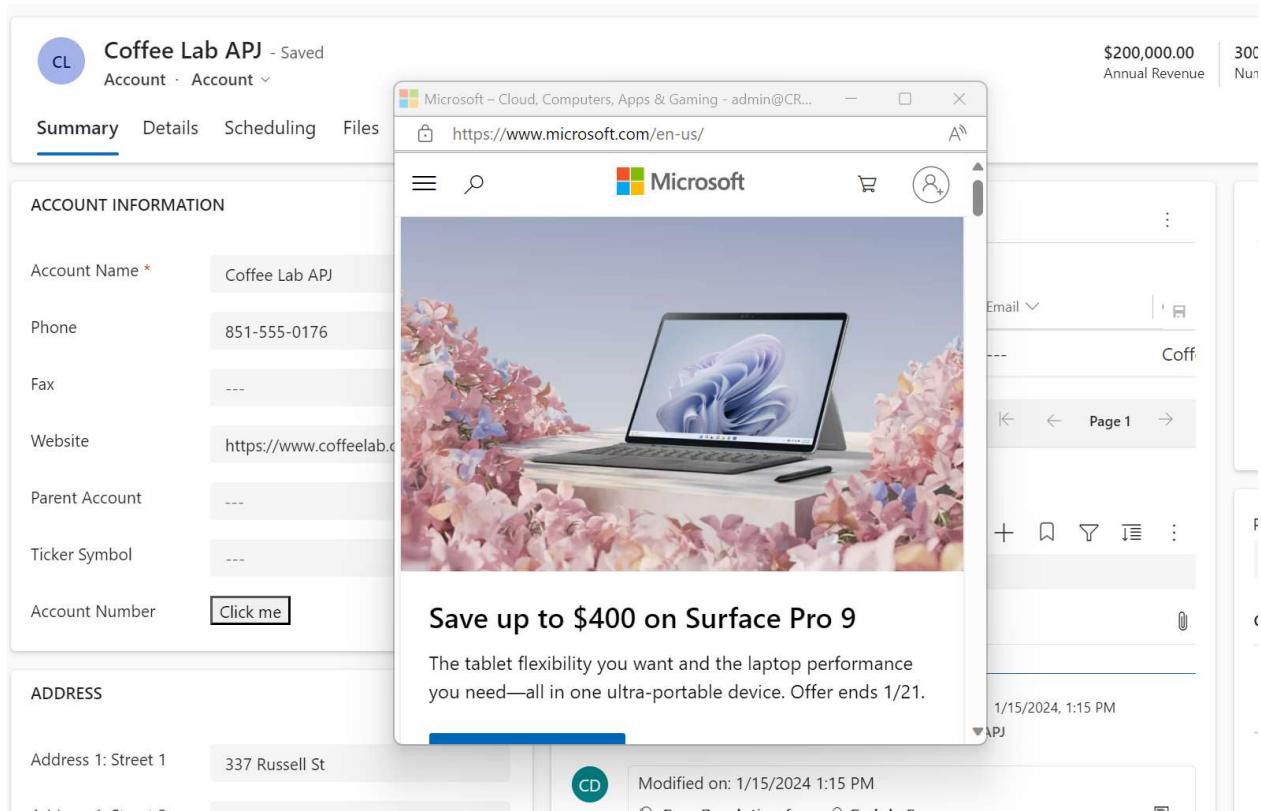
Let's look at some of the other functionality we get with context.navigation. We have `openConfirmDialog`, `openErrorDialog`, `openFile`, `openForm`, `openUrl`, and `openWebResource`:

```
context.navigation.openAlertDialog(alertStrings, alertOptions);
function (success) {
    console.log("openAlertDialog");
},
function (error) {
    console.log("openErrorDialog");
});
```

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:

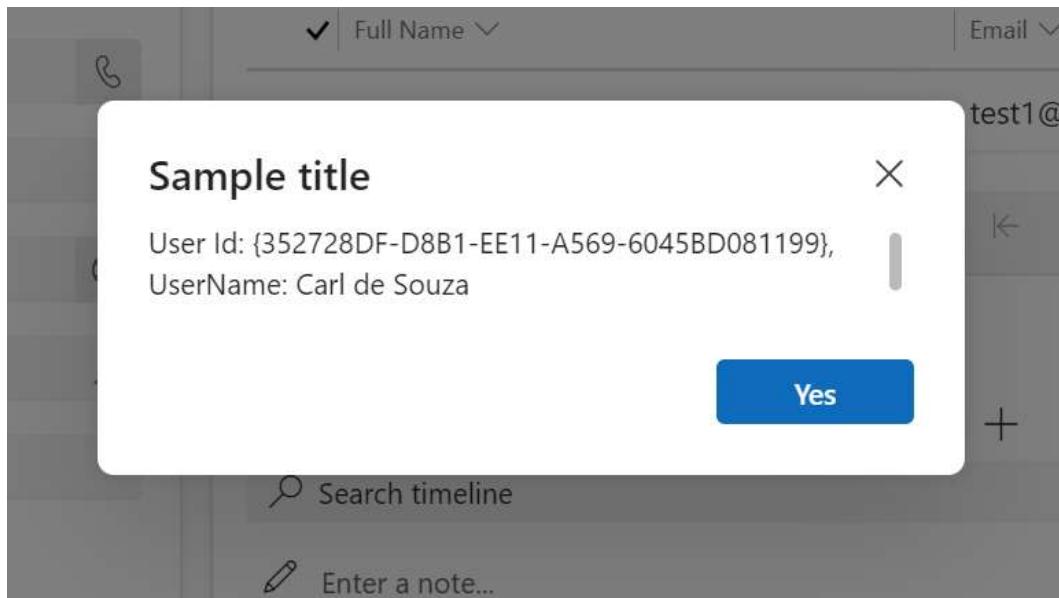
- `dateFormattingInfo`
- `getTimeZoneOffsetMinutes`
- `isRTL`
- `languageId`
- `numberFormattingInfo`
- `securityRoles`
- `userId`
- `userName`

```
context.userSettings.userId
    ⚡ dateFormattingInfo
    ⚡ getTimeZoneOffsetMinutes
    ⚡ isRTL
    ⚡ languageId
    ⚡ numberFormattingInfo
    ⚡ securityRoles
    ⚡ userId      (property) Component
    ⚡ 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:



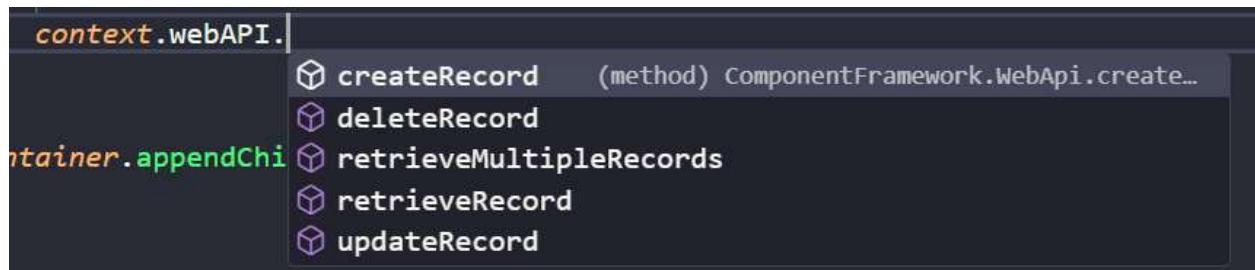
# Building a PCF Control that Calls Context WebApi

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



```
context.webAPI.  
    ⚡ createRecord    (method) ComponentFramework.WebApi.create...  
    ⚡ deleteRecord  
    ⚡ retrieveMultipleRecords  
    ⚡ retrieveRecord  
    ⚡ updateRecord  
  container.appendChi
```

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:

The screenshot shows a code editor with two tabs: 'index.ts' and 'ControlManifest.Input.xml'. The 'ControlManifest.Input.xml' tab is active, displaying XML code for a Microsoft Power Apps control manifest. The code includes sections for resources, feature usage, and various device features like audio capture and barcode scanning. Lines 38 through 51 are shown, with line 51 containing a closing tag for the feature usage section.

```
38     -->
39     </resources>
40     <!-- UNCOMMENT TO ENABLE THE SPECIFIED API
41     <feature-usage>
42         <uses-feature name="Device.captureAudio" required="true" />
43         <uses-feature name="Device.captureImage" required="true" />
44         <uses-feature name="Device.captureVideo" required="true" />
45         <uses-feature name="Device.getBarcodeValue" required="true" />
46         <uses-feature name="Device.getCurrentPosition" required="true" />
47         <uses-feature name="Device.pickFile" required="true" />
48         <uses-feature name="Utility" required="true" />
49         <uses-feature name="WebAPI" required="true" />
50     </feature-usage>
51     -->
```

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

```
<feature-usage>
    <!-- <uses-feature name="Device.captureAudio" required="true" />
    <uses-feature name="Device.captureImage" required="true" />
    <uses-feature name="Device.captureVideo" required="true" />
    <uses-feature name="Device.getBarcodeValue" required="true" />
    <uses-feature name="Device.getCurrentPosition" required="true" />
    <uses-feature name="Device.pickFile" required="true" />
    <uses-feature name="Utility" required="true" /> -->
    <uses-feature name="WebAPI" required="true" />
</feature-usage>
```

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 add 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:



And our sample account is created:

The screenshot shows a Dynamics 365 account record page. At the top left is a purple circular icon with 'SA'. The title is 'Sample Account - Saved'. Below the title is a breadcrumb 'Account · Account'. A navigation bar below the title includes 'Summary' (underlined), 'New Tab', 'Details', 'Scheduling', 'Files', and 'Asset'. The main content area is titled 'ACCOUNT INFORMATION' and contains a field for 'Account Name \*' with the value 'Sample Account'.

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:

**org2f7773cc.crm.dynamics.com says**

Account deleted

OK

To retrieveMultiple accounts:

```
context.webAPI.retrieveMultipleRecords("account", "?$select=name").then(
    function success(result) {
        // perform operations 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 API
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:

**org2f7773cc.crm.dynamics.com says**

Account Name: Coffee Lab API

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.

The screenshot shows a Microsoft Dynamics 365 account record. At the top left is a purple circular icon with 'UA'. The title is 'Updated Account Name - Saved'. Below it is a breadcrumb 'Account · Account'. The main data area has three sections: 'Annual Revenue' (\$200,000.00), 'Number of Employees' (300), and 'Owner' (Carl de Souza). Below this is a navigation bar with tabs: 'Summary' (underlined), 'New Tab', 'Details', 'Scheduling', 'Files', and '...'. A section titled 'ACCOUNT INFORMATION' follows, containing a 'Account Name \*' field with the value 'Updated Account Name'. A note at the bottom states: '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:

The screenshot shows the SAP PCF Control React component code in VS Code. The Explorer sidebar on the left lists files like `index.ts`, `HelloWorld.tsx`, and configuration files. The main editor area displays the `index.ts` file content:

```

1  import { IInputs, IOOutputs } from "./generated/ManifestTypes";
2  import { HelloWorld, IHelloWorldProps } from "./HelloWorld";
3  import * as React from "react";
4
5  export class SampleControlReact implements ComponentFramework.ReactControl<IInputs, IOOutputs> {
6      private theComponent: ComponentFramework.ReactControl<IInputs, IOOutputs>;
7      private notifyOutputChanged: () => void;
8
9      /**
10      * Empty constructor.
11      */
12      constructor() { }
13
14      /**
15      * Used to initialize the control instance. Controls can kick off
16      * Data-set values are not initialized here, use updateView.
17      * @param context The entire property bag available to control via props
18      * @param notifyOutputChanged A callback method to alert the framework when output changes
19      * @param state A piece of data that persists in one session for each component
20      */
21      public init(
22          context: ComponentFramework.Context<IInputs>,
23          notifyOutputChanged: () => void,
24          state: ComponentFramework.Dictionary
25      ): void {
26          this.notifyOutputChanged = notifyOutputChanged;
27      }
28
29      /**
30      * Called when any value in the property bag has changed. This includes
31      * @param context The entire property bag available to control via props
32      * @returns ReactElement root react element for the control
33      */
34      public updateView(context: ComponentFramework.Context<IInputs>): ReactElement {
35          const props: IHelloWorldProps = { name: "Hello World!" };

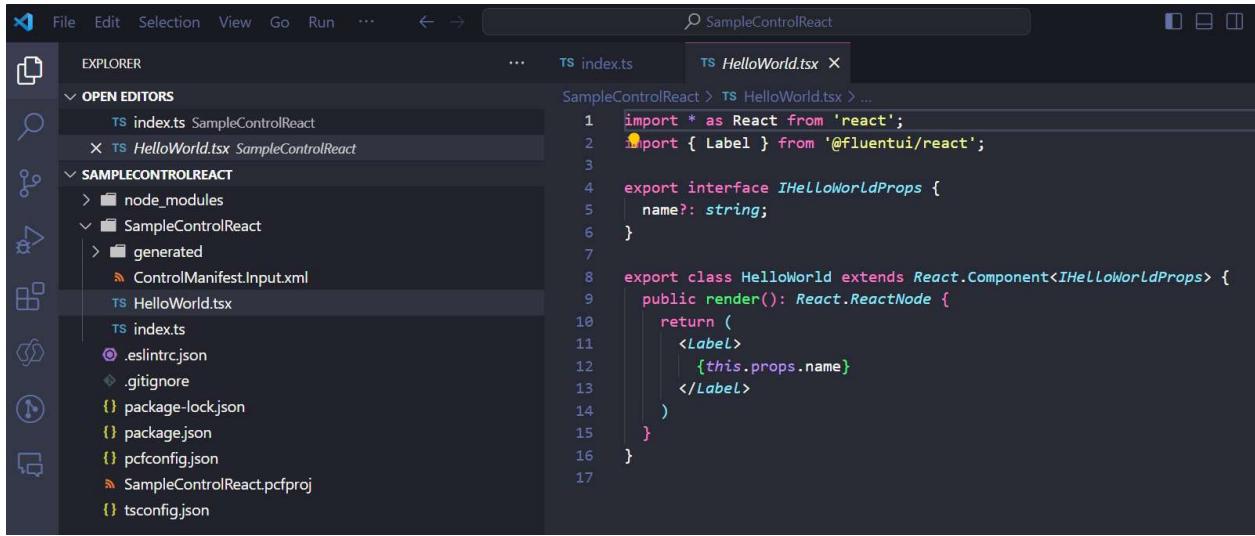
```

The status bar at the bottom indicates "Initializing JS/TS language features".

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):



The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, ...
- Search Bar:** SampleControlReact
- Explorer:** Shows the project structure under SAMPLECONTROLREACT:
  - node\_modules
  - generated
  - ControlManifest.Input.xml
  - index.ts
  - .eslintrc.json
  - .gitignore
  - package-lock.json
  - package.json
  - pcfconfig.json
  - SampleControlReact.pcfproj
  - tsconfig.json
- Editor:** TS HelloWorld.tsx (active tab)  
Content:

```
1 import * as React from 'react';
2 import { Label } 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 (
11       <Label>
12         {this.props.name}
13       </Label>
14     )
15   }
16 }
17
```

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:

The screenshot shows a web browser window with the URL `localhost:8181`. The title bar reads "PowerApps component framework Test Environment". The main content area displays a component named "SampleControlReact".

**Context Inputs:**

- Form Factor:** Web
- Component Container Width:**  Only positive whole numbers allowed
- Component Container Height:**  Only positive whole numbers allowed

**Data Inputs:**

- Property:** sampleProperty
- Value:**
- Type:** SingleLine.Text

A text box at the top contains the text "Hello, World!".

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

The screenshot shows the PowerApps component framework Test Environment. On the left, there is a preview window displaying the text "Hello, World!". To the right, the control is named "SampleControlReact". Under "Context Inputs", there is a "Form Factor" dropdown set to "Web". Below it, there are two pairs of "Component Container" inputs: "Width" is set to 120 and "Height" is also set to 120.

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

The screenshot shows the "Data Inputs" section. It contains one property named "sampleProperty" with a value of "value1" and a type of "SingleLine.Text".

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 that
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
```

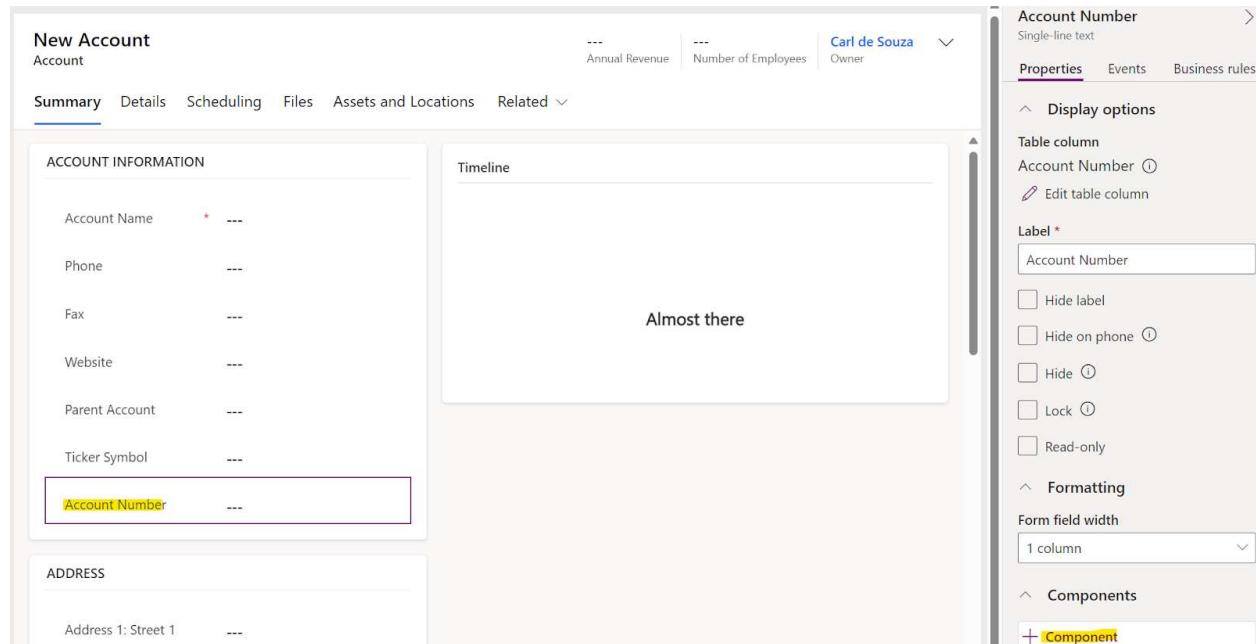
Now connect to the Dataverse (replace the org below with yours):

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

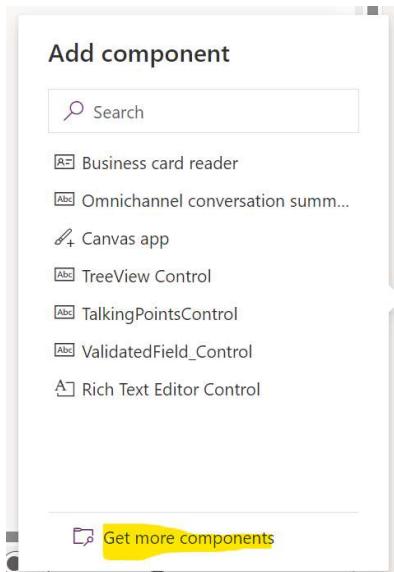
Now go back to the component folder and do the final step to push (replace the prefix with yours):

```
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:

Get more components

Refresh

All   Built by Microsoft   Built by others

All components in the current environment. Not seeing a component you're exper:

Component
SampleControlReact
SMSTemplateControl
Sales Conversation Grid Control
CC_LeadHygieneDuplicateDialogControl_Name
PAGridCustomizer
Collab
CC_LinkControl_Name
Image
FullPageControl
CC_TemplateMappingControl_Name

Add   Cancel

Select it again on the next screen and click Add:



We see our control rendered:

Dynamics 365 | Sales Hub

**Coffee Lab APJ - Saved**

Account · Account

\$200,000.00    300  
Annual Revenue    Number of Employees

Carl d Owner

**Summary** Details Scheduling Files Ass

Website: https://www.coffeelab.com

Parent Account: ---

Ticker Symbol: ---

Account Number: Hello, World!

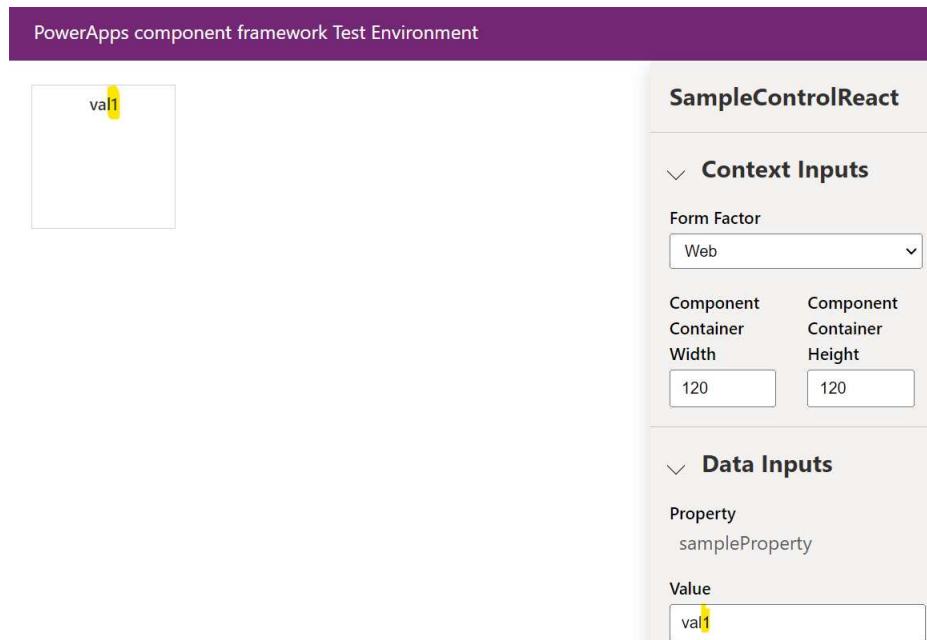
Now let's extend the control.

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

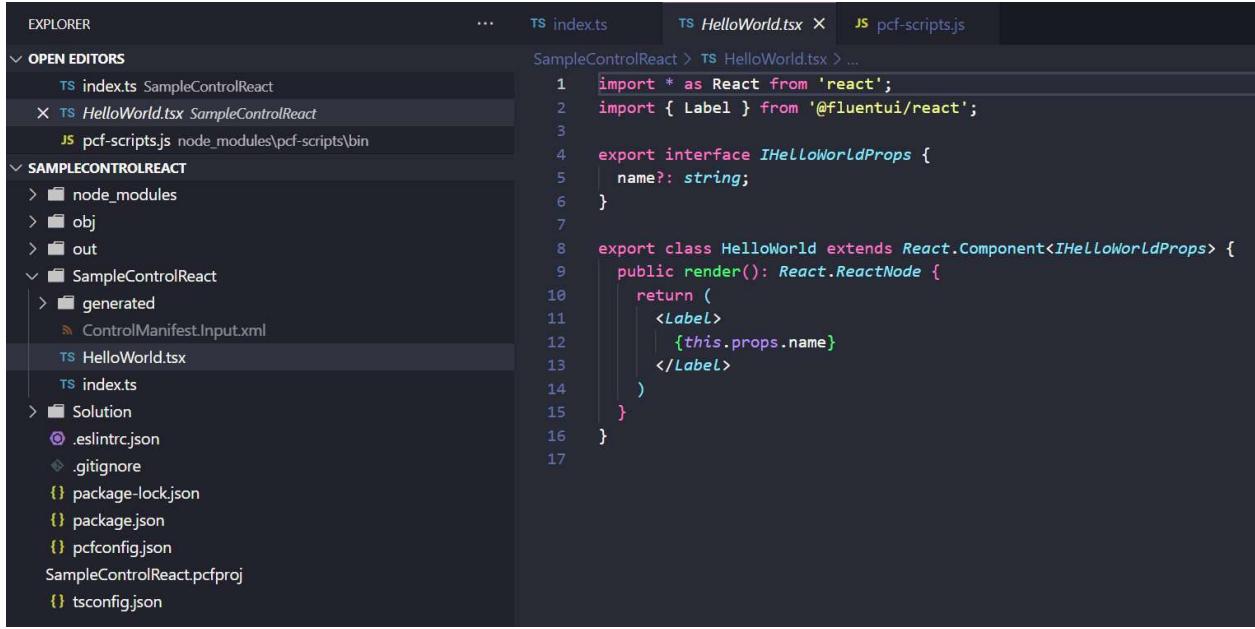
```
context.parameters.sampleProperty.raw || ""
```

```
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:



The screenshot shows the VS Code interface with the following details:

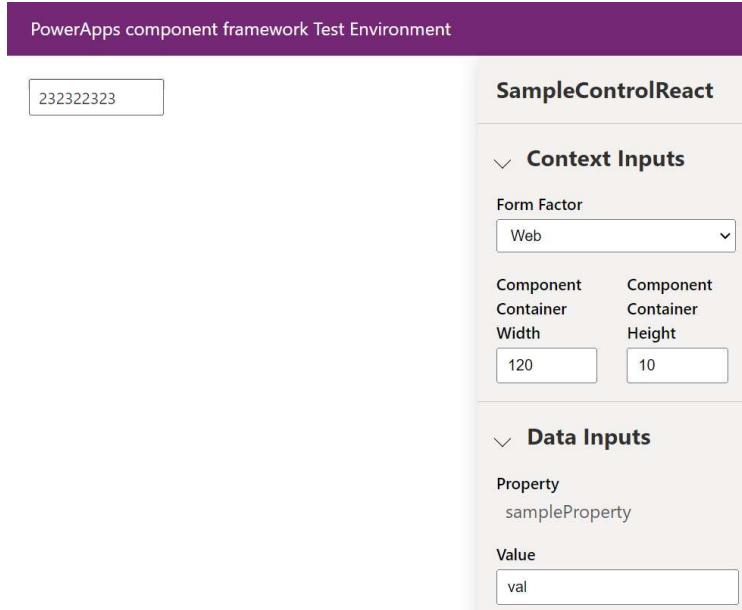
- EXPLORER**: Shows the project structure with files like `index.ts`, `HelloWorld.tsx`, and `pcf-scripts.js`.
- OPEN EDITORS**: Shows the current editor tabs: `index.ts`, `HelloWorld.tsx`, and `pcf-scripts.js`.
- SAMPLECONTROLREACT**: Shows the folder structure including `node_modules`, `obj`, `out`, and the `SampleControlReact` folder which contains `generated`, `ControlManifest.Input.xml`, `HelloWorld.tsx`, `index.ts`, `Solution`, `.eslintrc.json`, `.gitignore`, `package-lock.json`, `package.json`, `pcfconfig.json`, `SampleControlReact.pcfproj`, and `tsconfig.json`.
- CODE**: Displays the `HelloWorld.tsx` file content:

```
1 import * as React from 'react';
2 import { Label } 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 (
11       <Label>
12         {this.props.name}
13       </Label>
14     )
15   }
16 }
17
```

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.handleChange}
    />
  }

  private handleChange = (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.ReactNode {
  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.ReactNode {
        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:

```

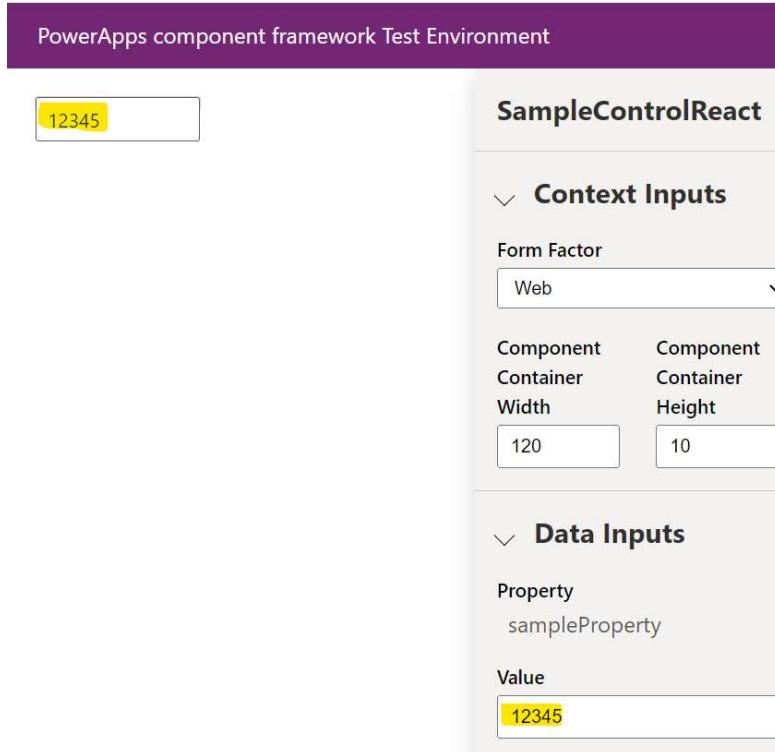
export interface IHelloWorldProps {
    name?: string;
    updateValue: (value: any) => void;
}

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.PropertyTypeDataSet;
```

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 as
     * 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 control
     * @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: ComponentFramework.State)
    {
        // Add control initialization code
    }

    /**
     * Called when any value in the property bag has changed. This includes field values, data-sets, global values etc.
     * @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 "bound"
     */
    public getOutputs(): IOoutputs
    {
        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="standard">
    <!-- external-service-usage mode 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:

```

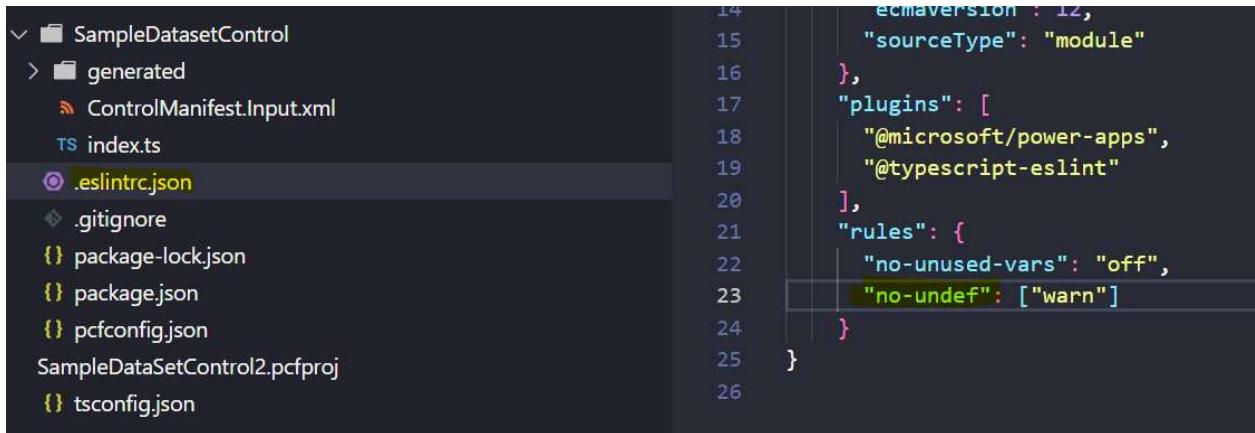
<data-set name="sampleDataSet" display-name-key="Dataset_Display_Key">
    <!-- 'property-set' node represents a unique, configurable property that each record in the dataset must provide. -->
    <!-- UNCOMMENT TO ADD PROPERTY-SET NODE
    <property-set name="samplePropertySet" display-name-key="Property_Display_Key" 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:



```

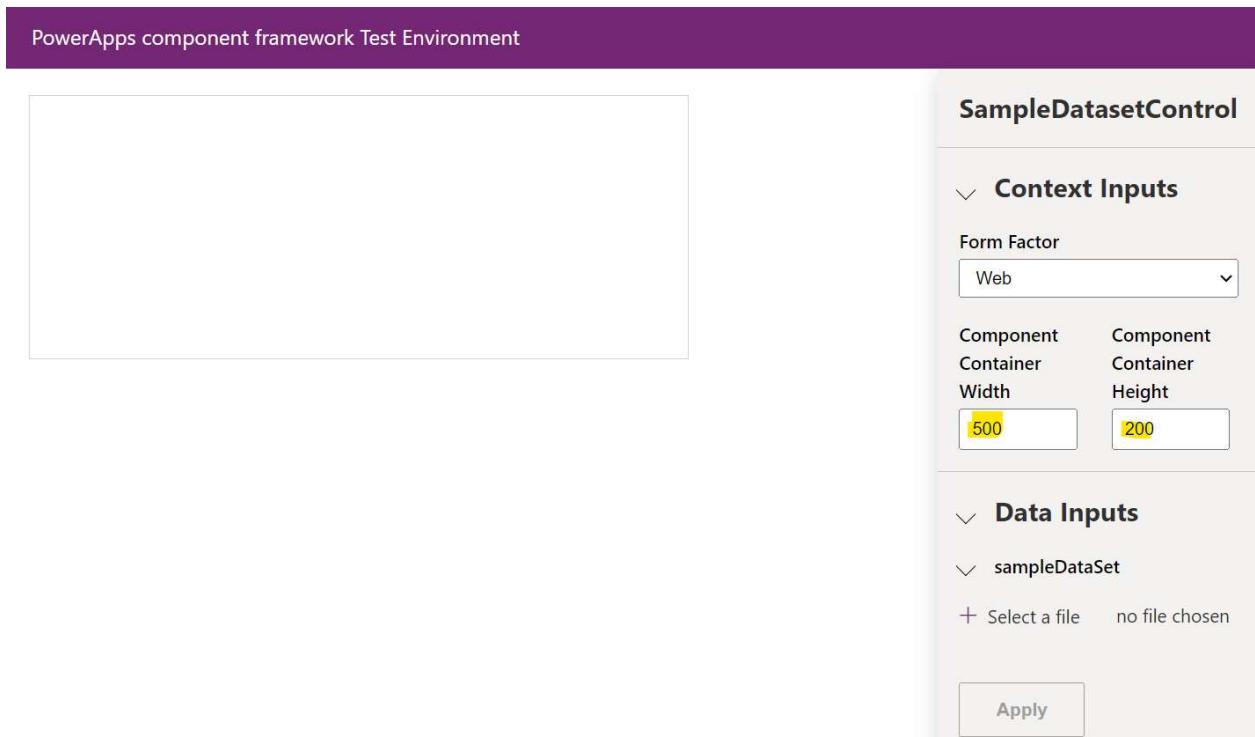
14     "ecmaVersion": 12,
15     "sourceType": "module"
16   },
17   "plugins": [
18     "@microsoft/power-apps",
19     "@typescript-eslint"
20   ],
21   "rules": {
22     "no-unused-vars": "off",
23     "no-undef": ["warn"]
24   }
25 }
26

```

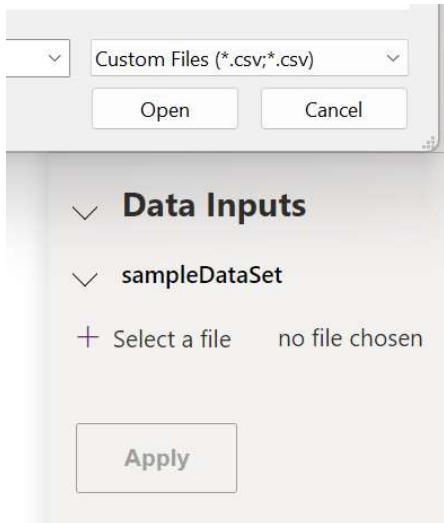
And run this in our test harness with:

```
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	<a href="mailto:bob@smith.com">bob@smith.com</a>
3	David	Smith	<a href="mailto:david@smith.com">david@smith.com</a>
4	Mary	Smith	<a href="mailto:mary@smith.com">mary@smith.com</a>

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

✓ **Data Inputs**

✓ sampleDataSet

+ Select a file      DatasetSample.csv

**Column**

First Name

Type

SingleLine.Text

---

**Column**

Last Name

Type

SingleLine.Text

---

**Column**

Email

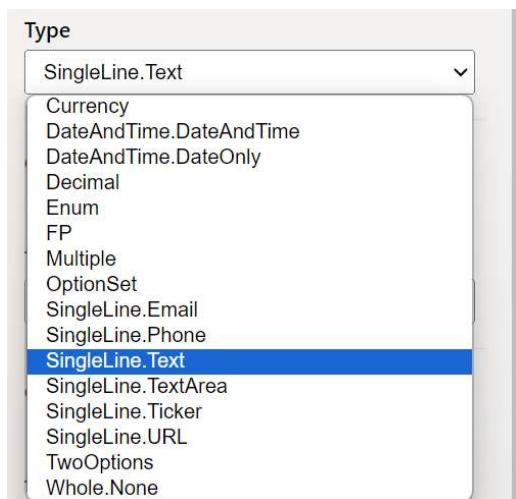
Type

SingleLine.Text

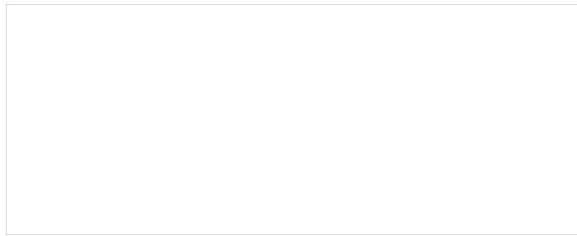
---

**Apply**

Note the different data type options:



At this point, nothing is displayed in our grid:



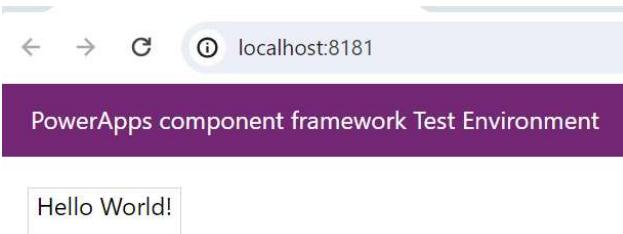
We'll need to add some code to render the dataset. Let's add some code. As with the **field** component we have an **HTMLDivElement** container:

```
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>, notifyOutputChanged: () => void, state: ComponentFramework.Dictionary, container: HTMLDivElement): void
    {
        // 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
```

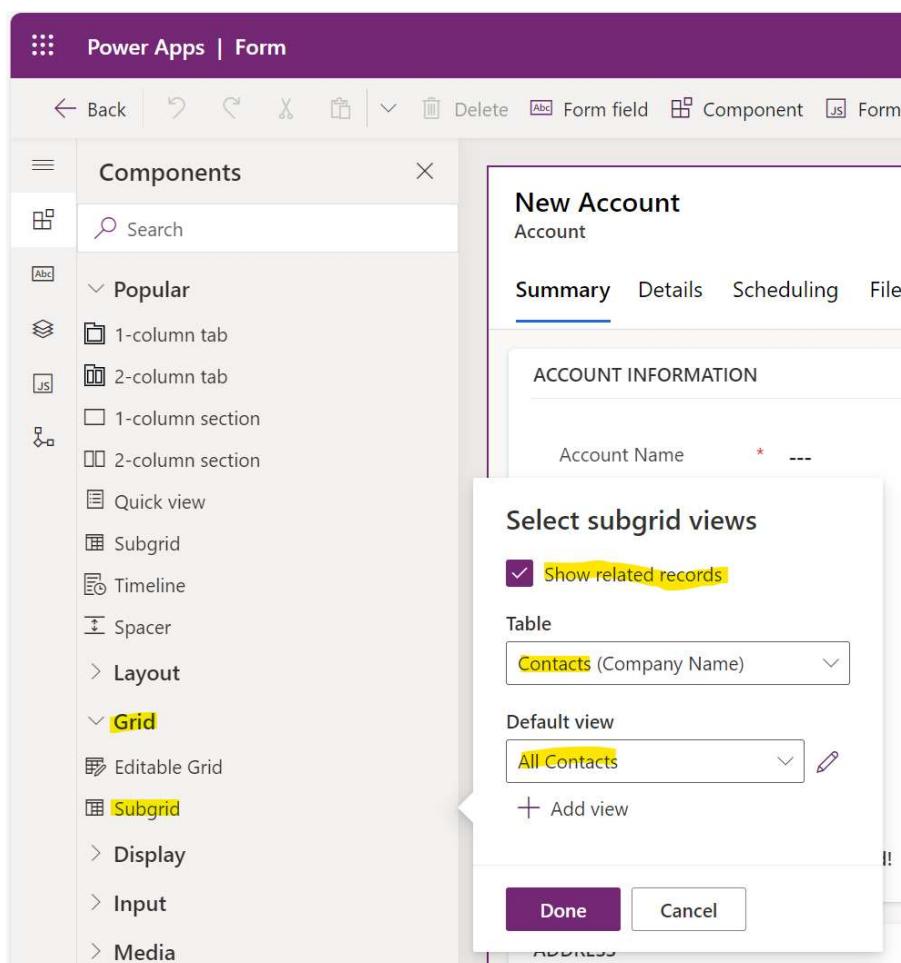
Now connect to your org:

```
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:

**Coffee Lab APJ** - Saved  
Account · Account

**Summary Details Scheduling Files Assets and Locations Related**

**ACCOUNT INFORMATION**

Account Name \*  
Coffee Lab APJ

Phone  
851-555-0176

New SG control 1705000355859

Bob Smith  
Coffee Lab APJ

1 - 1 of 1    Page 1

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

New SG control 1705000355859

Full Name Email

No data available

0 - 0 of 0

Timeline

Almost th

Add component

Search

Editable Grid

Calendar

Get more components

Show hidden

Table

Contacts (Company Name)

Default view

All Contacts

+ Add view

Allow users to change view

Hide search box

Default chart

Contacts by Role

+ Add chart

Show chart only

Allow users to change chart

Maximum number of rows

4

Use available space

Formatting

Component width

1 column

Components

+ Component

Select our new SampleDatasetControl:

## Get more components

The screenshot shows a search interface with a purple refresh button and three filter tabs: 'All', 'Built by Microsoft', and 'Built by others'. Below the search bar, a message says 'All components in the current environment. Not seeing a component you like?'. A list of components is shown, with 'SampleDatasetControl' highlighted in a purple box.

Select it again and click Done:

The screenshot shows the 'Add' dialog for the 'SampleDatasetControl'. It includes fields for 'Dataset\_Display\_Key' (set to 'Contacts'), 'Table \*' (set to 'Contacts'), 'View \*' (set to 'All Contacts'), and a section for 'Show component on' (with 'Web', 'Mobile', and 'Tablet' checked). At the bottom are 'Done' and 'Cancel' buttons.

Right away we see our Hello World coming through:

The screenshot shows a web page with a purple header containing the text 'New SG control 1705000355859'. Below the header is a purple box containing the text 'Hello World!'. The entire page has a light gray background.

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:

```
context.parameters.sampleDataSet.paging.setPageSize(10);  
    ↳ addColumn?  
    ↳ clearSelectedRecordIds  
    ↳ columns  
    ↳ error  
    ↳ errorMessage  
    ↳ filtering  
    ↳ getSelectedRecordIds  
    ↳ getTargetEntityType  
    ↳ getTitle  
    ↳ getViewId  
    ↳ linking  
    ↳ loading  
  
Called when any value in the property changes.  
@param context The entire property object.  
  
public updateView(context: ComponentFactory): void  
{  
    // Add code to update control view.  
    console.log("updateView");  
  
    * Called when any value in the property changes.  
    * @param context The entire property object.  
    */  
    public updateView(context: ComponentFactory): void  
    {  
        // Add code to update control view.  
        console.log("updateView");  
  
        ↳ addColumn?  
        ↳ clearSelectedRecordIds  
        ↳ columns  
        ↳ error  
        ↳ errorMessage  
        ↳ filtering  
        ↳ getSelectedRecordIds  
        ↳ getTargetEntityType  
        ↳ getTitle  
        ↳ getViewId  
        ↳ linking  
        ↳ loading  
        ↳ openDatasetItem  
        ↳ paging  
        ↳ records  
        ↳ refresh  
        ↳ setSelectedRecordIds  
        ↳ sortedRecordIds  
        ↳ sorting
```

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<IIInputs>): 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!

The screenshot shows the SAP UI5 Test Harness configuration for a dataset and the browser's developer tools.

**Data Inputs Configuration:**

- sampleDataSet:**
  - DatasetSample.csv** (selected)
  - First Name**: Column, Type: SingleLine.Text
  - Last Name**: Column, Type: SingleLine.Text
  - Email**: Column, Type: SingleLine.Email

**Developer Tools (Scope Tab):**

- Line 27, Column 24: (From bundle.js:20) Coverage: n/a
- Breakpoints: Pause on uncaught exceptions (unchecked), Pause on caught exceptions (unchecked)
- index.ts: SampleDatasetControl.prototype.u... (checked)
- Call Stack: Not paused (XHR/fetch Breakpoints, DOM Breakpoints)
- Console: Shows log entries:
  - HEADER: First Name - First Name index.ts:30
  - HEADER: Last Name - Last Name index.ts:30
  - HEADER: Email - Email index.ts:30
  - RECORD: bob@smith.com index.ts:33
  - RECORD: david@smith.com index.ts:33
  - RECORD: mary@smith.com index.ts:33

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<IIInputs, IOOutputs>
{
    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.records[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:

localhost:8182

PowerApps component framework Test Environment

name	telephone1	address1_city
val	val	val
val	val	val
val	val	val

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

**Data Inputs**

sampleDataSet

+ Select a file

Column      DatasetSample.csv

First Name

Type

SingleLine.Text

Column

Last Name

Type

SingleLine.Text

Column

Email

Type

SingleLine.Email

Apply

Once Apply is clicked our dataset is rendered:

## 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 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):

Find Locations Related ▾

The screenshot shows a subgrid control within a Power Apps form. The subgrid has a header row with columns: Full Name, Email, Company Name, Business Phone, and Status. Below the header, there is one data row containing the values: Bob Smith, null, Coffee Lab APJ, 851-555-0176, and Active. To the left of the subgrid, there is a sidebar with icons for Find Locations, Related, Timeline, and a search bar labeled "Search timeline".

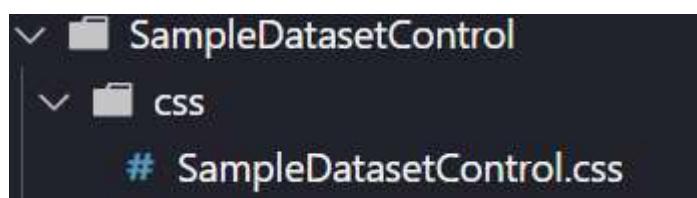
This doesn't look so good visually, so let's add a CSS to our code.

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:

A screenshot of a file explorer showing the contents of a folder named "SampleDatasetControl". Inside the folder, there is a subfolder "generated" containing "ControlManifest.Input.xml". The file "ControlManifest.Input.xml" is open in a code editor. The code shows a section for resources:

```
</data-set>
<resources>
    <code path="index.ts" order="1"/>
    <!-- UNCOMMENT TO ADD MORE RESOURCES -->
    <css path="css/SampleDatasetControl.css" order="1" />
    <!-- <resx path="strings/SampleDatasetControl.1033.res" />
</resources>
```

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 ▾

The screenshot shows a Microsoft Dynamics 365 form for a new contact record. The title bar reads "New SG control 1705350865914". The main content area has a table with columns: Full Name, Email, Company Name, Business Phone, and Status. A single row is present with values: Bob Smith, null, Coffee Lab APJ, 851-555-0176, and Active. Below the table is a "Timeline" section with a search bar and a note input field.

Full Name	Email	Company Name	Business Phone	Status
Bob Smith	null	Coffee Lab APJ	851-555-0176	Active

Timeline

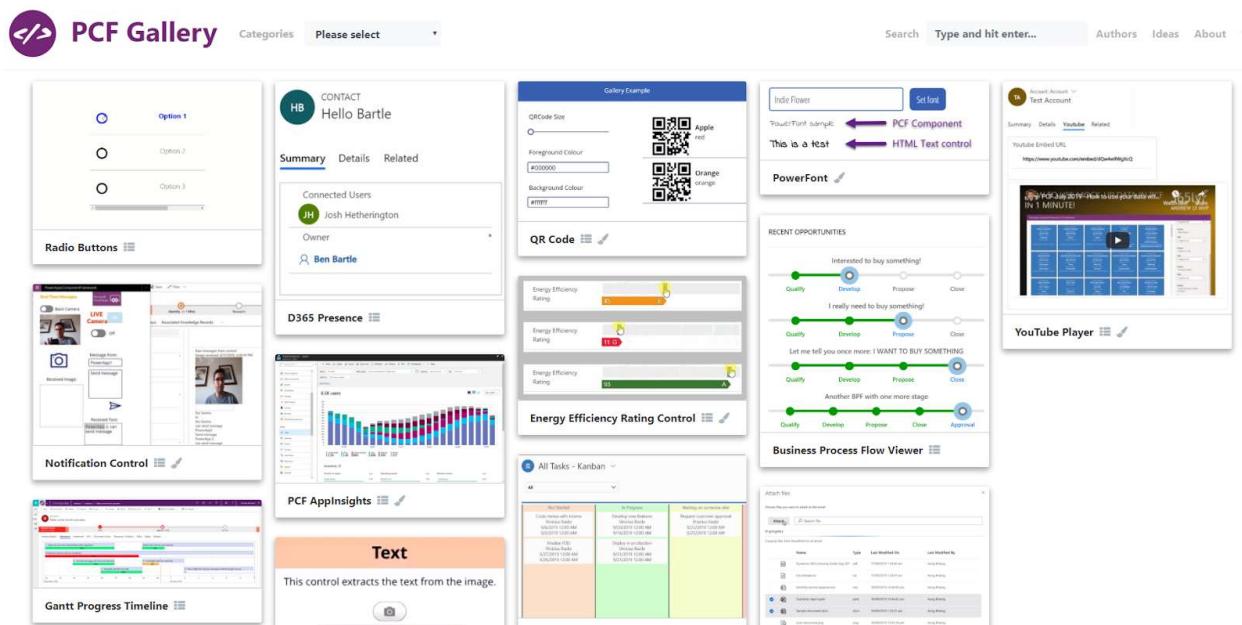
Search timeline

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.

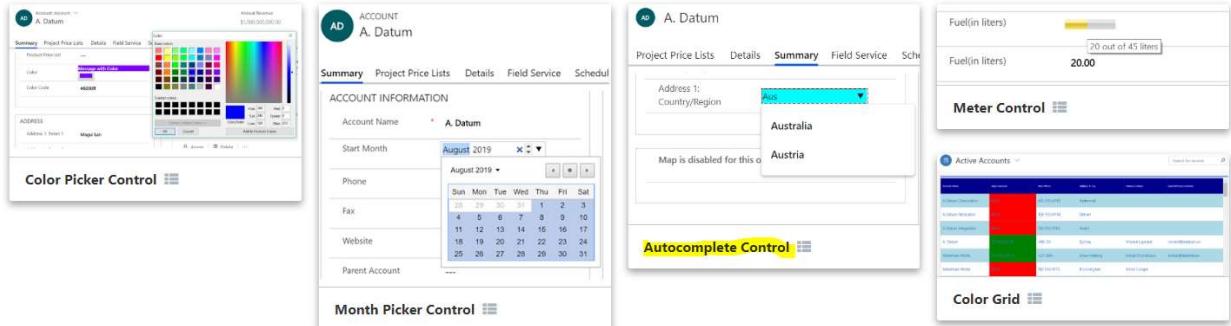
First, head over to [PCF.gallery](#). You will see links to various controls:



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:

Search [Type and hit enter...](#) Authors Ideas About [Twitter](#)

Let's install the AutoComplete control by [Sriram Balaji](#):



The image displays four separate PowerApp screens, each featuring a different control component:

- Color Picker Control:** Shows a color palette with a selected color and a corresponding hex code.
- Month Picker Control:** Shows a month calendar for August 2019, with specific dates highlighted.
- Autocomplete Control:** Shows a dropdown menu with options like "Australia" and "Austria".
- Color Grid:** Shows a grid of colored squares.

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

## Autocomplete Control

AD

A. Datum

Project Price Lists Details **Summary** Field Service Sch

Address 1:  
Country/Region

Aus ▼

Australia

Austria

Map is disabled for this o

## Autocomplete Control

### MODEL-DRIVEN APPS

A control to select a value from a predefined list of values and automatically filters the values based on what the user types.

Author: **Sriram Balaji**  

[Visit ↗](#)

[Download ↴](#)

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

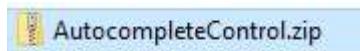
The screenshot shows a GitHub repository page. At the top, there are links for 'Why GitHub?', 'Enterprise', 'Explore', 'Marketplace', and 'Pricing'. A search bar and 'Sign in'/'Sign up' buttons are also at the top. Below the header, the repository name 'srirambalajigit / PCFControls' is shown, along with 'Watch' (4), 'Star' (1), 'Fork' (3) buttons. A navigation bar below the repository name includes 'Code', 'Issues 1', 'Pull requests 0', 'Projects 0', 'Security', and 'Insights'. The main content area shows a list of files under the branch 'master':

File	Action	Last Commit
Autocomplete	Add files via upload	4 months ago
Autocomplete.pcfproj	Add files via upload	4 months ago
AutocompleteControl.zip	Add files via upload	4 months ago
Readme.md	Create Readme.md	4 months ago
package-lock.json	Add files via upload	4 months ago
package.json	Add files via upload	4 months ago
pcfconfig.json	Add files via upload	4 months ago
tsconfig.json	Add files via upload	4 months ago

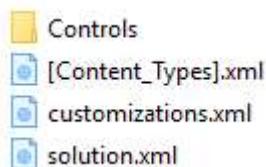
Click Download:

The screenshot shows the download page for the file 'AutocompleteControl.zip'. It includes the file's details: 'srirambalajigit' uploaded it on Jun 17, and it has 1 contributor. The file size is 5.52 KB. At the bottom right, there are buttons for 'Download' (highlighted with a yellow box), 'History', and trash/recycle bin.

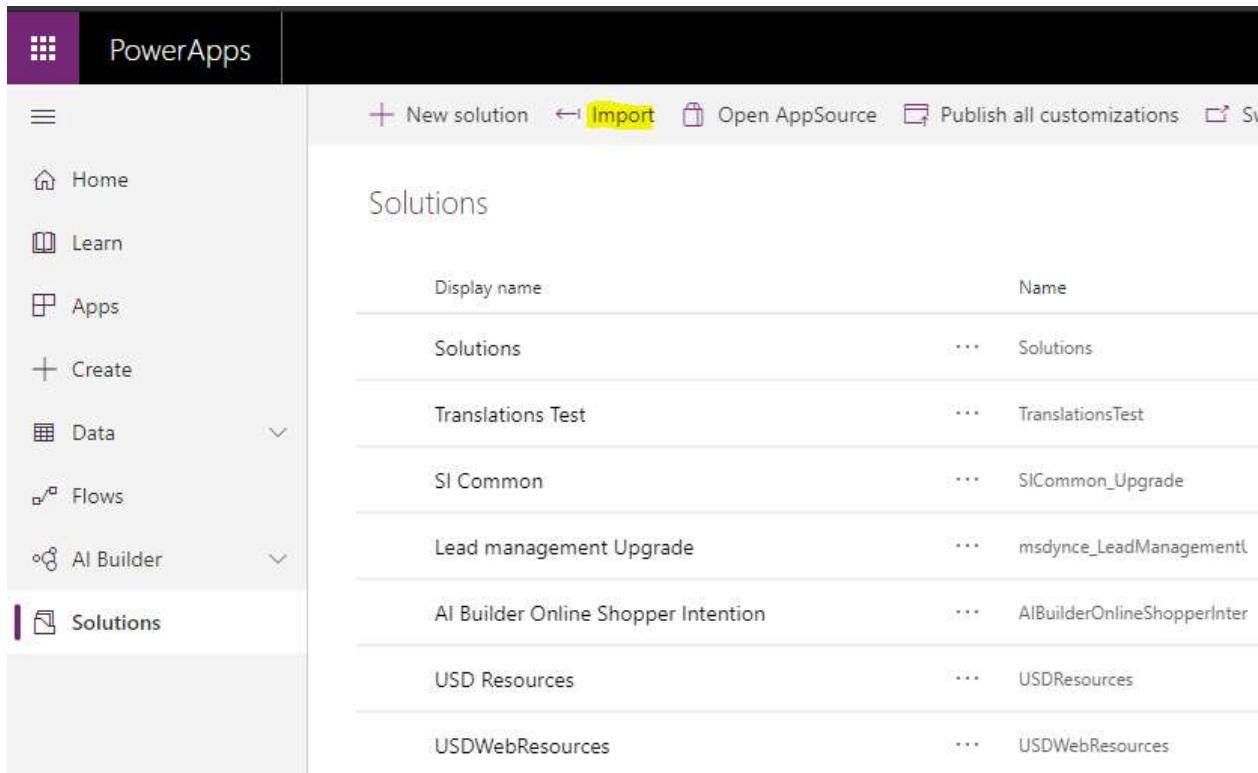
This downloads the file:



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:



The screenshot shows the PowerApps home page. On the left, there's a navigation bar with icons for Home, Learn, Apps, Create, Data, Flows, AI Builder, and Solutions. The 'Solutions' icon is highlighted with a purple bar. The main area is titled 'Solutions' and lists several solutions with their display names and names:

Display name	Name
Solutions	Solutions
Translations Test	TranslationsTest
SI Common	SICommon_Upgrade
Lead management Upgrade	msdynce_LeadManagementU
AI Builder Online Shopper Intention	AIBuilderOnlineShopperInten
USD Resources	USDResources
USDWebResources	USDWebResources

Select the ZIP file, then click Next:



This is a 'Select Solution Package' dialog box. It contains a label instructing the user to select a compressed (.zip or .cab) file for import, followed by a 'Choose File' button and a text input field showing 'AutocompleteControl.zip'. At the bottom are 'Back', 'Next', and 'Cancel' buttons.

Then Import:

## Solution Information

Help

### Solution Information

Name: AutocompleteControl  
Publisher: sriramcontrols(sriramcontrols)  
Package Type: Managed

[View solution package details](#)

 By enabling this command, you consent to share your data with an external system. Data imported from external systems into Microsoft Dynamics 365 are subject to our privacy statement that can be accessed [here](#). Please consult the feature technical documentation for more information.

[Back](#)

[Import](#)

[Cancel](#)

Then click Close:

## Importing Solution

Help

 The import of solution: AutocompleteControl completed successfully.

Type ↑	Display Name...	Name	Status
Dependencie...			
Labels	Labels		
Package Valid	Package Valid	Validation	

◀ ▶ Page 1 ▶

[Download Log File](#)

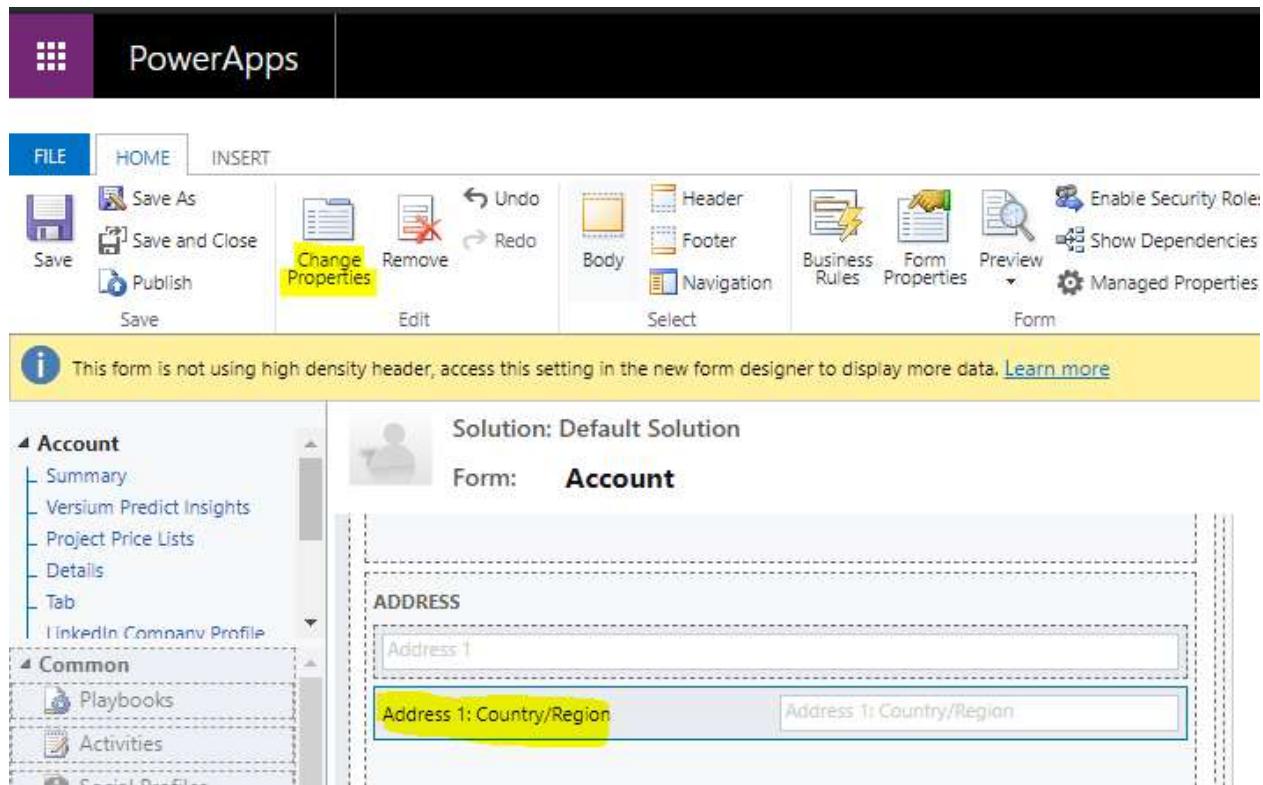
[Close](#)

We see the solution has imported:

## Solutions

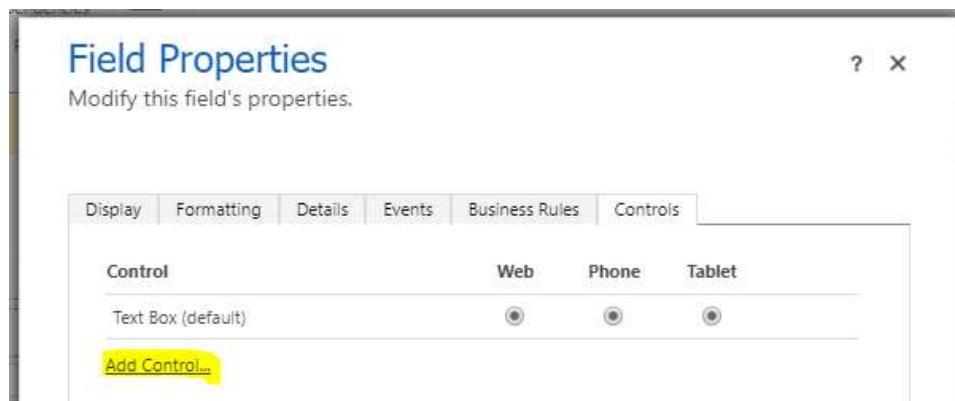
Display name	Name
AutocompleteControl	AutocompleteControl

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.



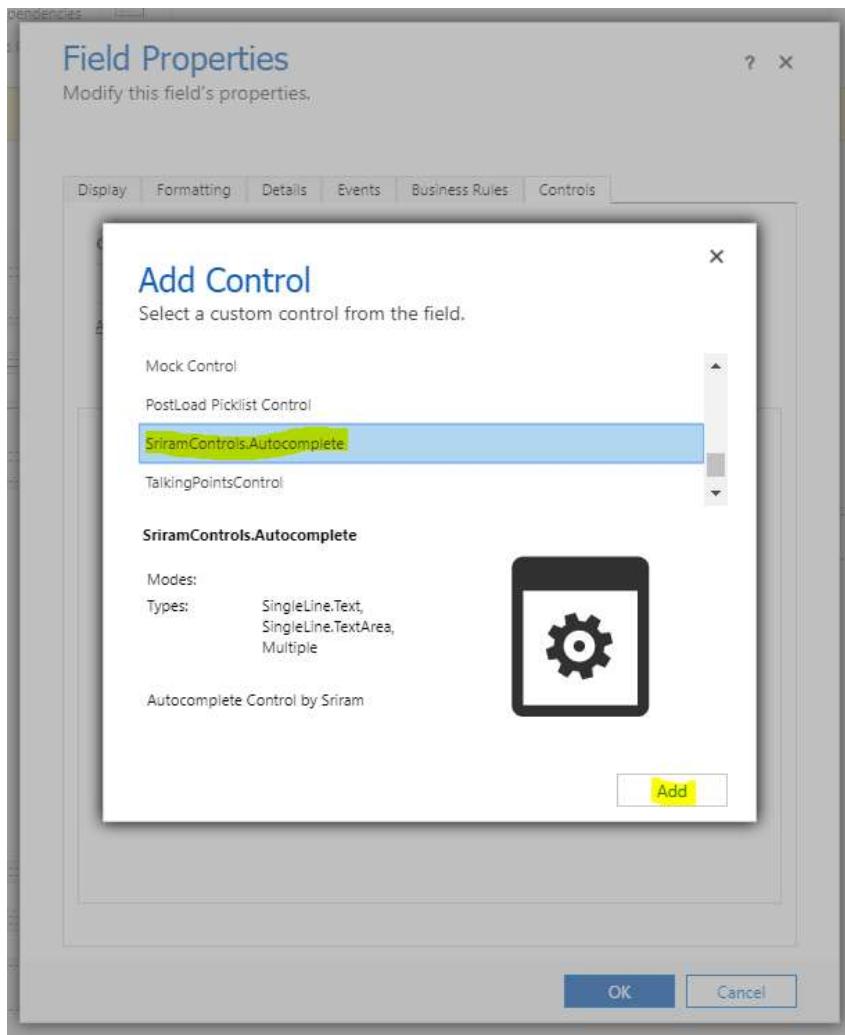
The screenshot shows the PowerApps form designer interface. At the top, there's a navigation bar with 'PowerApps' and tabs for 'FILE', 'HOME', and 'INSERT'. Under the 'HOME' tab, there are buttons for 'Save As', 'Save and Close', 'Publish', 'Change Properties' (which is highlighted with a yellow box), 'Remove', 'Undo', 'Redo', 'Header', 'Footer', 'Navigation', 'Business Rules', 'Form Properties', 'Preview', 'Show Dependencies', and 'Managed Properties'. A message bar at the top says, 'This form is not using high density header, access this setting in the new form designer to display more data. [Learn more](#)'. On the left, there's a sidebar with sections for 'Account' (Summary, Versum Predict Insights, Project Price Lists, Details, Tab, LinkedIn Company Profile) and 'Common' (Playbooks, Activities). The main area shows the 'Solution: Default Solution' and 'Form: Account'. Below that, there's a section labeled 'ADDRESS' with two input fields: 'Address 1' and 'Address 1: Country/Region' (which is also highlighted with a yellow box).

Go to Controls and select Add Control:

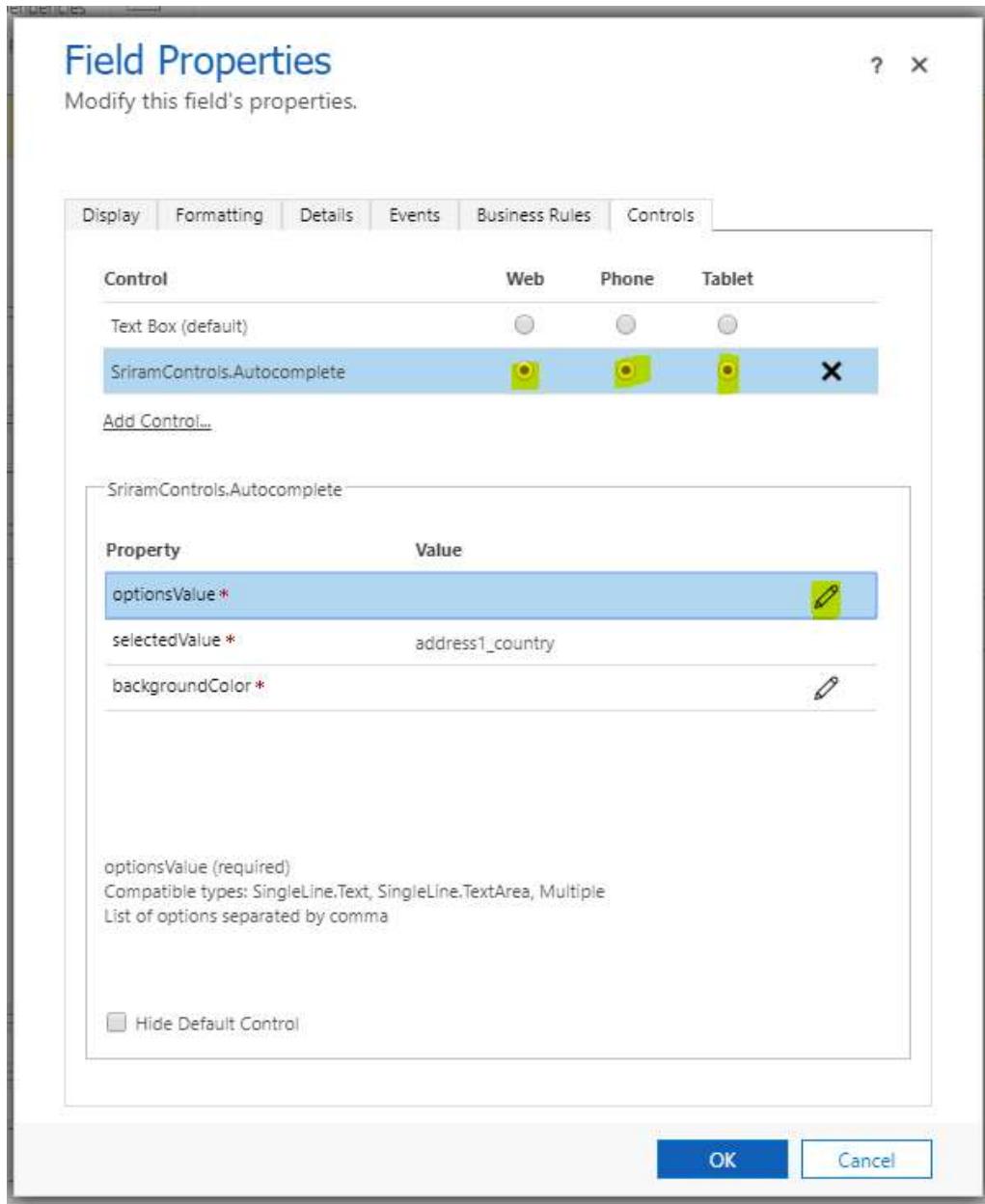


The screenshot shows the 'Field Properties' dialog box. At the top, it says 'Field Properties' and 'Modify this field's properties.' Below that is a toolbar with tabs for 'Display', 'Formatting', 'Details', 'Events', 'Business Rules', and 'Controls' (which is the active tab, indicated by a yellow box). Under the 'Controls' tab, there's a table with columns for 'Control', 'Web', 'Phone', and 'Tablet'. The 'Text Box (default)' control is selected. At the bottom of the dialog box, there's a button labeled 'Add Control...' (also highlighted with a yellow box).

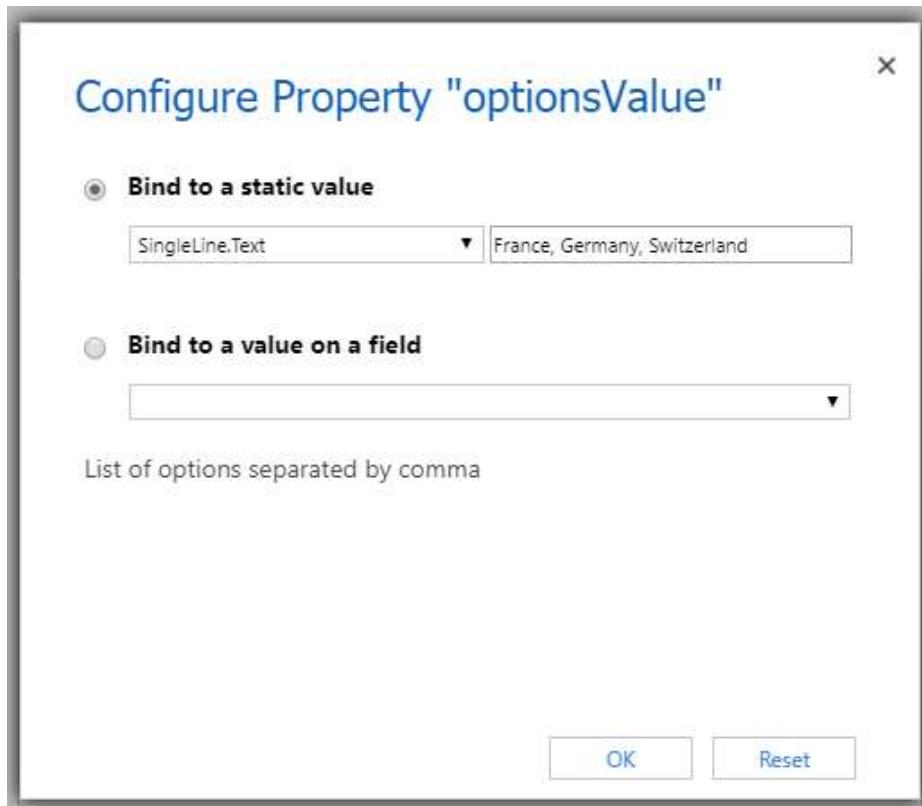
Find SriramControls.AutoComplete and click Add:



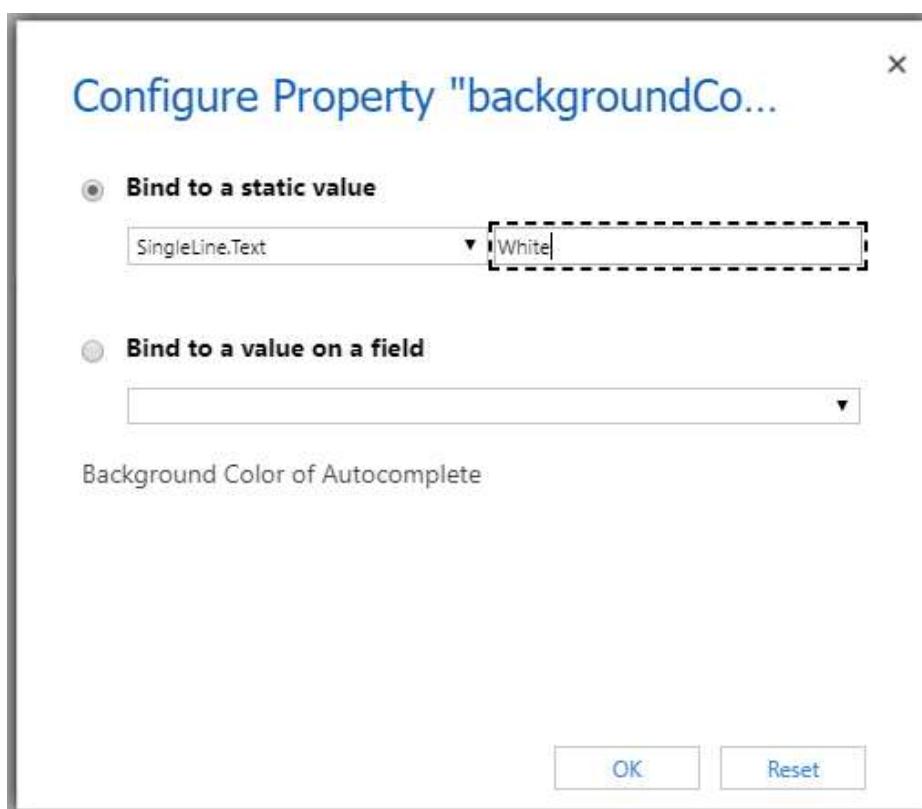
Select to use the control, and select the optionsValue:



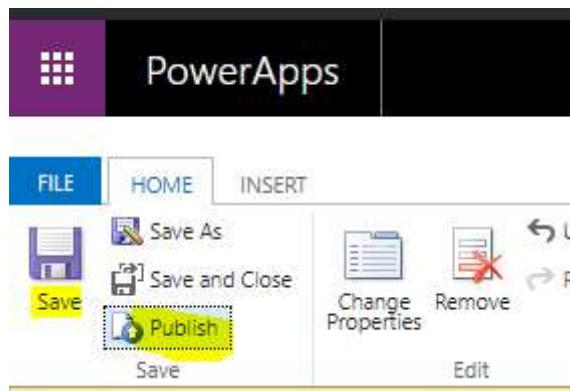
We will bind to static values, and enter in the countries that will be available for the user to select:



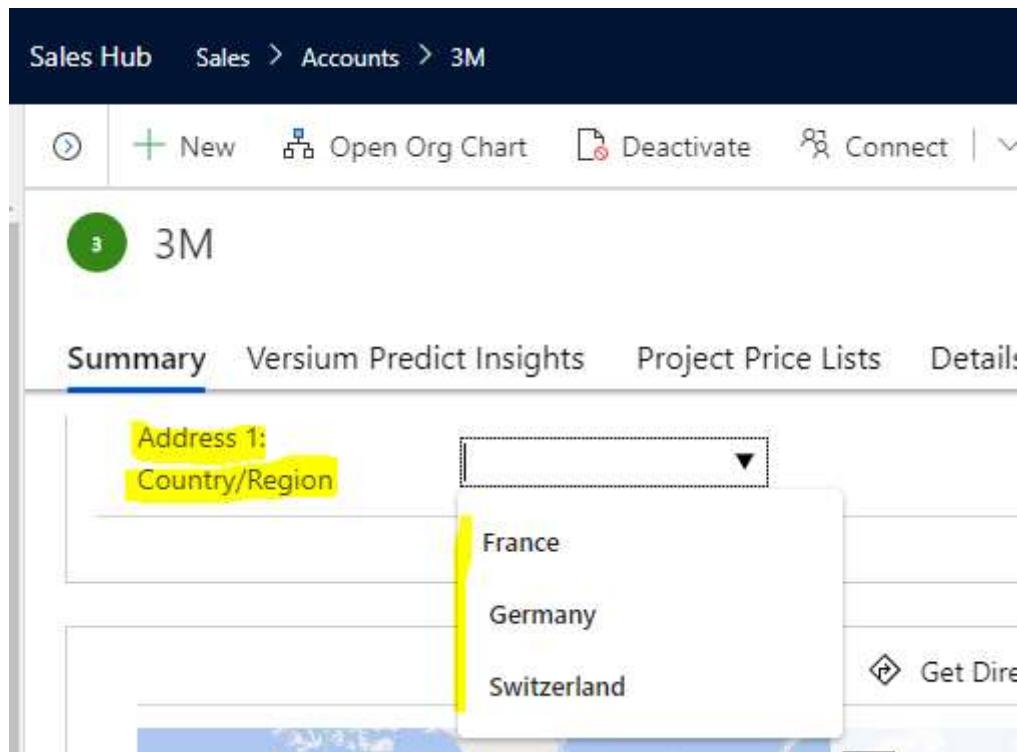
And also the background color:



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:

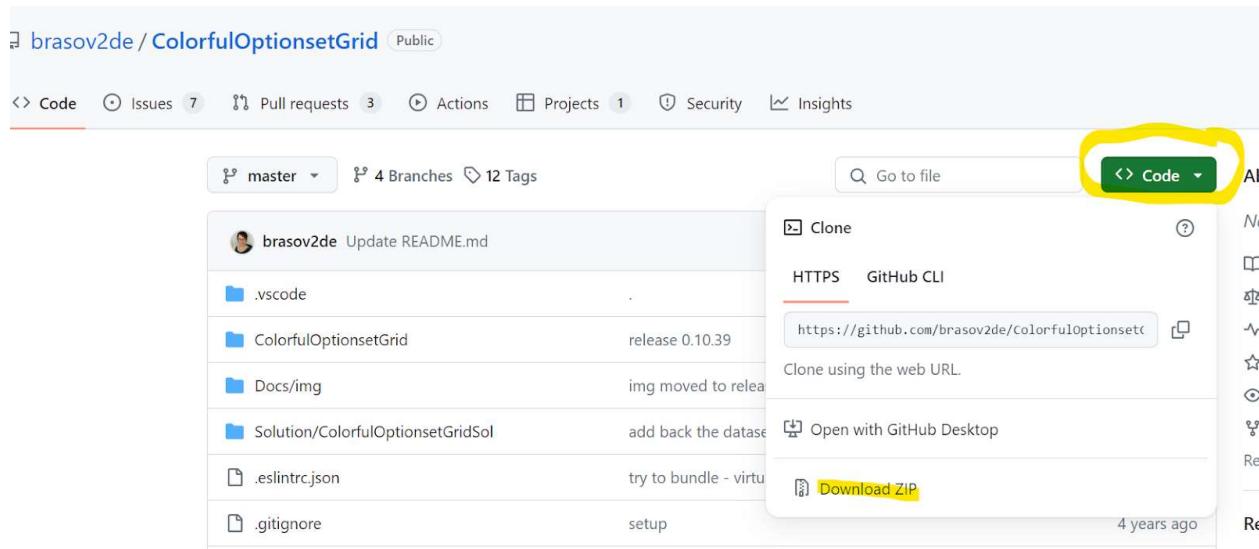
The screenshot shows the PCF.Gallery page for the "Colorful Optionset Grid" component. At the top, there is a navigation bar with various icons and a search bar. Below the navigation is a table titled "Active PCFTesters" showing a list of items with columns for ampel (color), Name, prio, optionsetcode, Created On, and Modified On. The table contains 117 rows. Below the table, it says "1 - 25 of 117 (selected)".

The main content area features the component's name, "Colorful Optionset Grid". Below the name are three status indicators: "MODEL-DRIVEN APPS" (with a bar chart icon), "LICENSE IS PRESENT" (with a scale icon), and "MANAGED SOLUTION AVAILABLE" (with a briefcase icon).

A descriptive text block states: "A control to show the Option Sets with the colors customized using the standard experience inside a grid." Below this text, the author is listed as "Diana Birkelbach" with links to Twitter, LinkedIn, and GitHub. There are two purple buttons: "Visit" and "Download". The "Download" button is circled in yellow.

At the bottom, there are three purple buttons with white text: "# color", "# grid", and "# optionset".

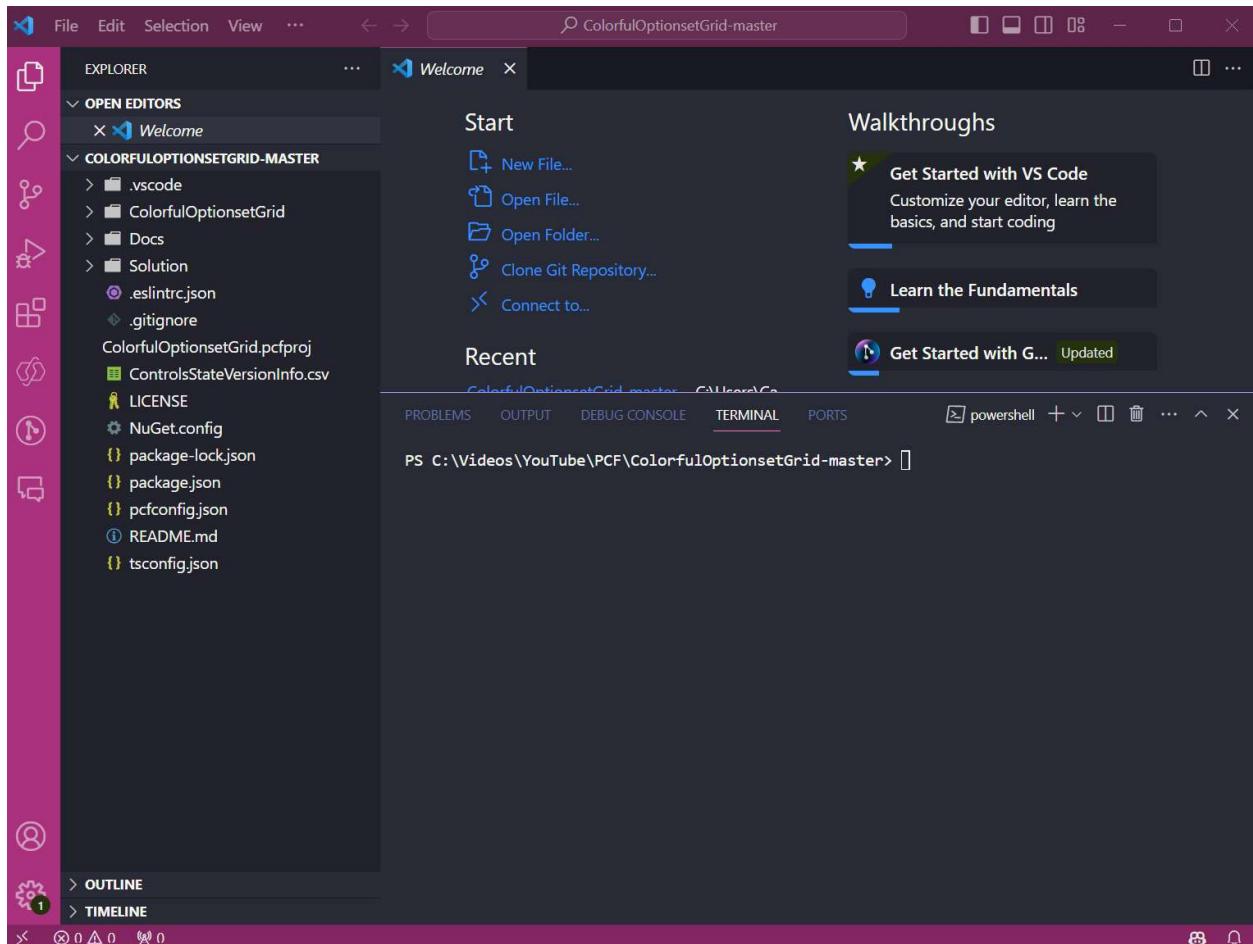
This takes us to the GitHub repo for the control. select Code->Download ZIP (or you can clone this project):



This downloads the PCF Controls:

 **ColorfulOptionsetGrid-master.zip**

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:

The screenshot shows the Power Apps classic interface. The left sidebar has a 'Objects' section with a search bar and a list of categories: 'All (19051)', 'Action Input Param...', 'Action Output Para...', 'Adaptive Card Config...', and 'Agent script (3)'. The right pane shows a list of items under 'Default Solution > All', with a 'Display name ↑' header. A yellow box highlights the 'Switch to classic' button in the top right corner of the interface.

And we will add control:

The screenshot shows the Power Apps Solution Editor interface. The top navigation bar includes 'Power Apps', 'Try New Experience' (with a note 'There's a better way to customize the system'), and 'Try New Experience' again. The ribbon has 'File' selected, followed by 'Show Dependencies', 'Solution Layers', 'Publish', and 'Managed Properties'. The left sidebar is titled 'Information' and shows 'Solution Default Solution' with 'Entities' expanded, listing 'Account', 'Teams', 'Dynamics 365', 'Playlists', and 'Account KPI Item'. The right panel is titled 'Control' and shows 'Read-only Grid (default)' selected under 'Control' for 'Account'. There are radio buttons for 'Web', 'Phone', and 'Tablet'.

And let's select the ColorfulOptionSetGrid:

The screenshot shows the 'Add Control' dialog box. It lists several controls: 'CC\_TopicClusteringControl', 'CCA\_Analytics\_ModernReportingControl', 'ColorfulOptionSetGrid' (highlighted with a blue box), and 'DatasetControl\_Display\_Key'. Below the list, there is a section titled 'ColorfulOptionSetGrid' with 'Modes:' and 'Types: Grid'. A preview image shows a grid with various colored rows. At the bottom is an 'Add' button.

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

General Primary Field Controls

Control	Web	Phone	Tablet	
Read-only Grid (default)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	
ColorfulOptionSetGrid	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	X
Add Control...				

ColorfulOptionSetGrid

Property	Value
Dataset	
Optionset 1	<input type="button" value="edit"/>
Optionset 2	<input type="button" value="edit"/>
Optionset 3	<input type="button" value="edit"/>
Display_text_type *	No decoration (Enum)

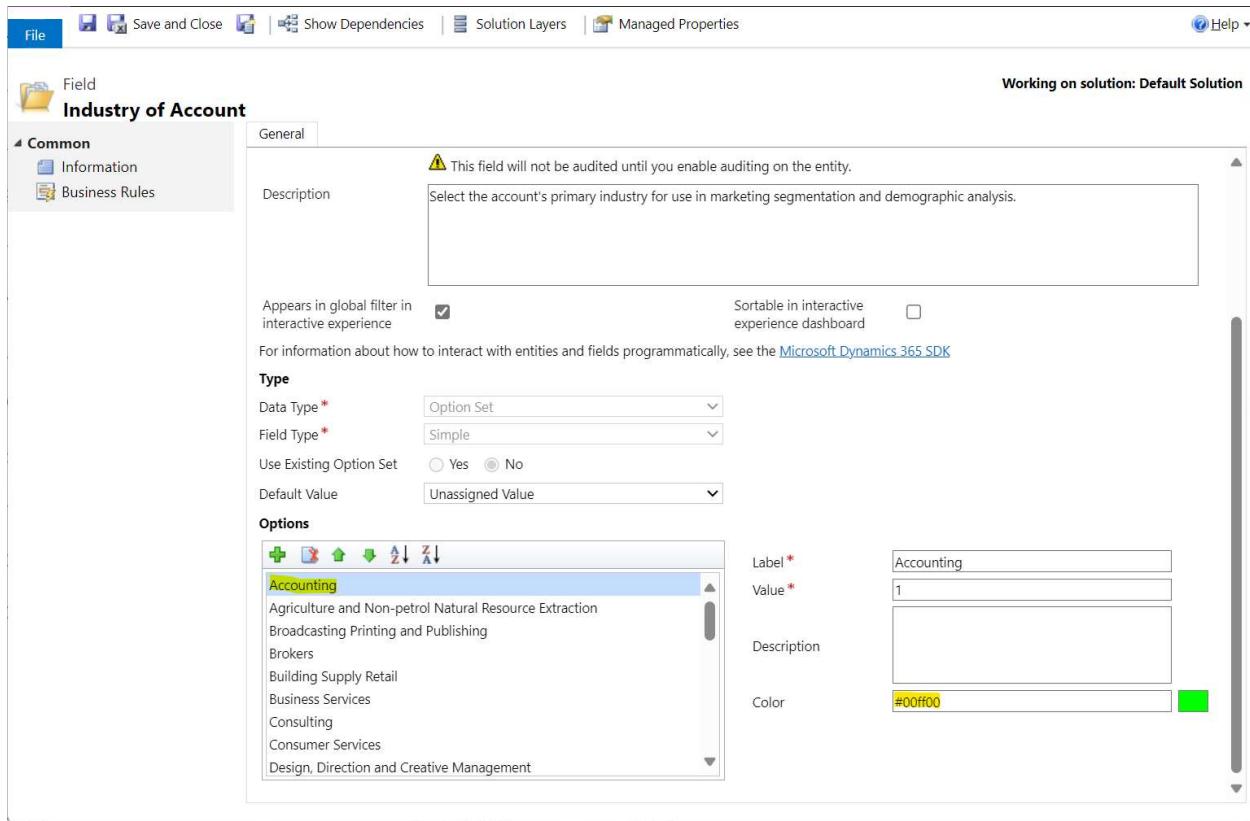
Scrolling down we see many more items we can customize:

ColorfulOptionSetGrid	
Icon definition *	<input type="button" value="edit"/>
isEditable *	<input type="button" value="edit"/>
Default Icon name	<input type="button" value="edit"/>
iconConfig1	<input type="button" value="edit"/>
iconConfig2	<input type="button" value="edit"/>
iconConfig3	<input type="button" value="edit"/>

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!

My Active Accounts with Industry					
Account Name ↑	Main Phone	Address 1: City	Primary Contact	Email (Primary Contact)	Industry
A. Datum Corporation (sample)	555-0158	Redmond	Rene Valdes (sample)	someone_i@example.com	● Accounting
Adventure Works (sample)	555-0152	Santa Cruz	Nancy Anderson (sample)	someone_c@example.com	● Broadcasting Printir
Alpine Ski House (sample)	555-0157	Missoula	Paul Cannon (sample)	someone_h@example.com	● Building Supply Ret
Blue Yonder Airlines (sample)	555-0154	Los Angeles	Sidney Higa (sample)	someone_e@example.com	● Transportation
City Power & Light (sample)	555-0155	Redmond	Scott Konersmann (sample)	someone_f@example.com	● Accounting
Coffee Lab API	851-555-0176	Hadley			● Broadcasting Printir
Coho Winery (sample)	555-0159	Phoenix	Jim Glynn (sample)	someone_j@example.com	
Contoso Pharmaceuticals (sample)	555-0156	Redmond	Robert Lyon (sample)	someone_g@example.com	

# 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.

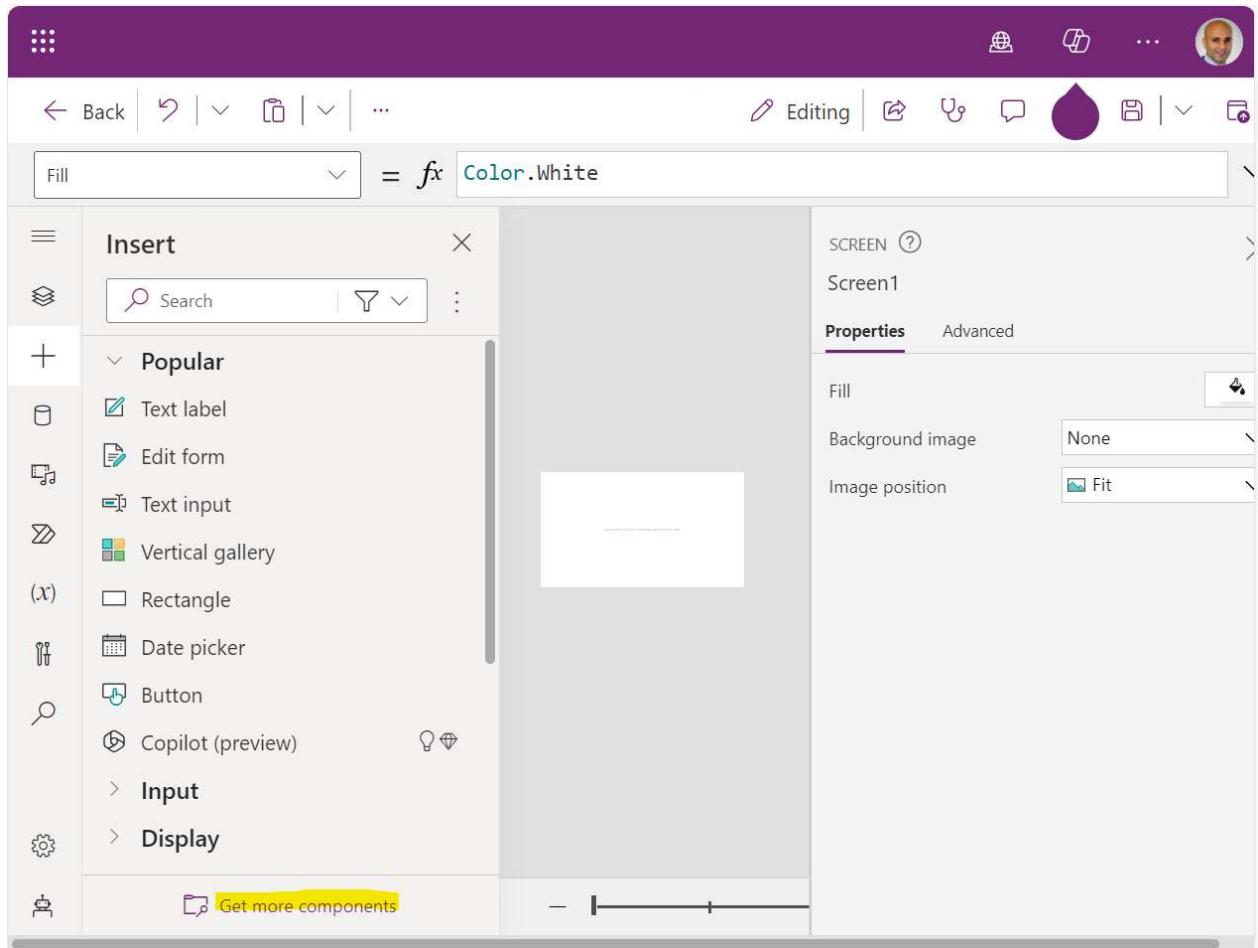
## Power Apps component framework for canvas apps

Enables Power Apps component framework feature that allows the execution of code that may not be generated by Microsoft when a maker adds code components to an app. Make sure that the code component solution is from a trusted source. [Learn more](#) 

Allow publishing of canvas apps with code components



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:

The screenshot shows the 'Import components' dialog. At the top, there are tabs for 'Canvas' and 'Code', with 'Code' being the active tab. A yellow banner at the top provides a warning about custom components: 'Custom components contain code that may not be generated by Microsoft. Make sure that the custom component solution is from a trusted source. [Learn more](#)'. Below the banner, a section titled 'Import code components published in your environment.' has a link to 'Learn more'. There are 'Refresh' and 'Search' buttons. The main area lists components with columns for 'Component', 'Owner', and 'Modified'. One component, 'SampleControl', is selected and highlighted with a purple checkmark. The table data is as follows:

Component	Owner	Modified
SampleControl	Default Publisher for org2f777...	2/8/24
ColorfulOptionSetGrid	Default Publisher for org2f777...	2/7/24
Power Apps grid control	Microsoft	2/4/24
Copilot	Microsoft	2/4/24

At the bottom are 'Import' and 'Close' buttons.

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

The screenshot shows the Microsoft Power Apps sidebar. The 'Code components' section is expanded, revealing the 'SampleControl' component. Other items like 'Copilot (preview)', 'Input', and 'Data' are also listed.

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.

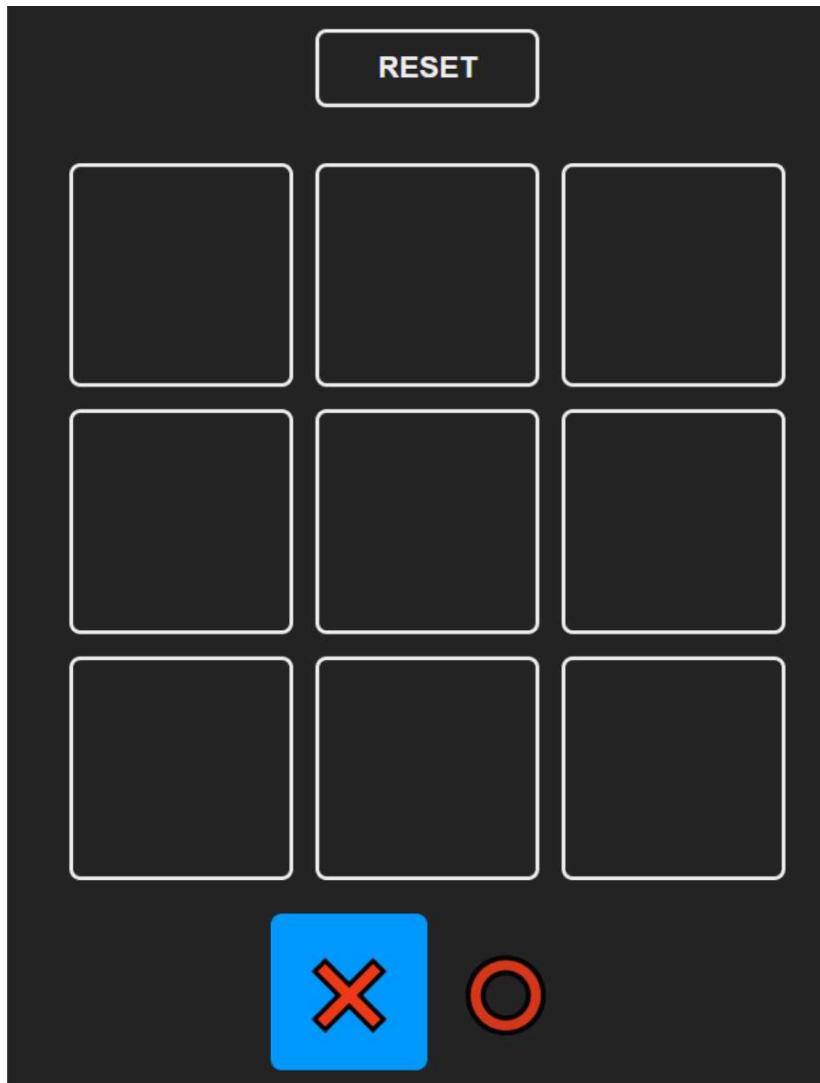
To find other controls for Canvas Apps, let's search in the <https://pcf.gallery>. Go to the search page and select Made for Canvas Apps:

A screenshot of the PCF Gallery Search interface. The URL in the address bar is <https://search.pcf.gallery>. The search results page shows several filters at the top:

- Made for Model-driven Apps
- Made for Canvas Apps
- Made for Power Apps Portals
- License is present
- Managed Solution is available

Below the filters, there are dropdowns for "Tagged Categories" (Nothing selected) and "From Authors" (Nothing selected), and a blue "Search" button.

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



Upload and import the component:

## Import components

Canvas **Code**

ⓘ Custom components contain code that may not be generated by Microsoft. Make sure that the custom component solution is from a trusted source. [Learn more](#)

Import code components published in your environment. [Learn more](#)

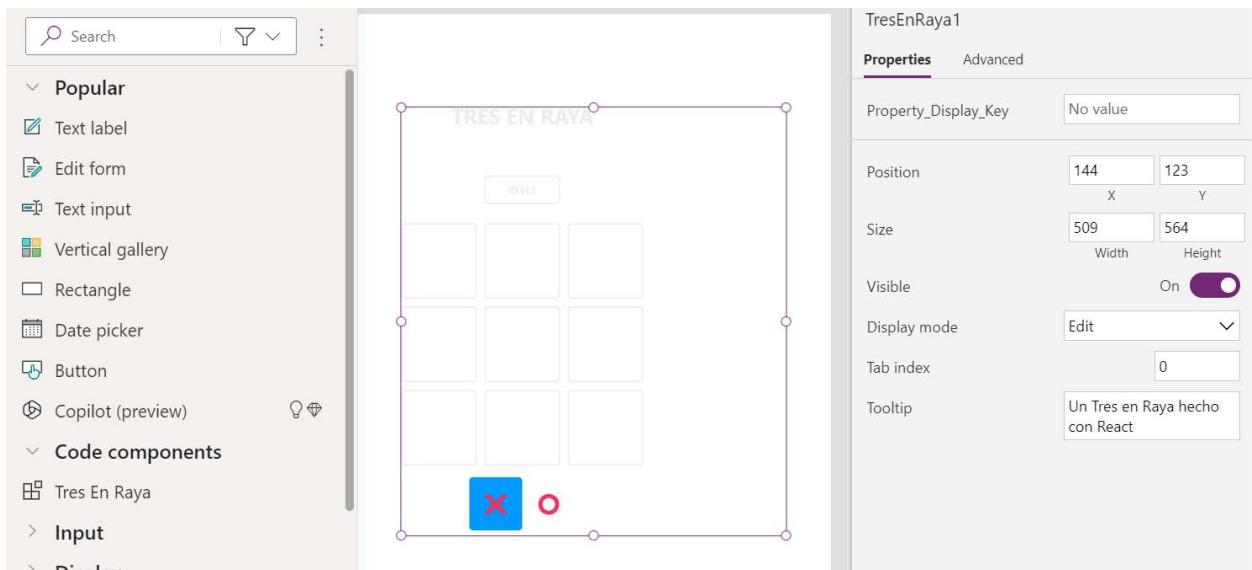
⟳ Refresh

🔍 Search

Component	Owner	Modified ↓
-----------	-------	------------

Tres En Raya	Default Publisher for orgd75f9...	2/16/24
--------------	-----------------------------------	---------

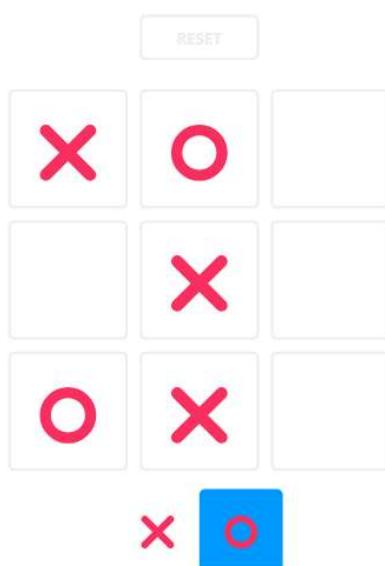
We can drag the control onto the canvas:



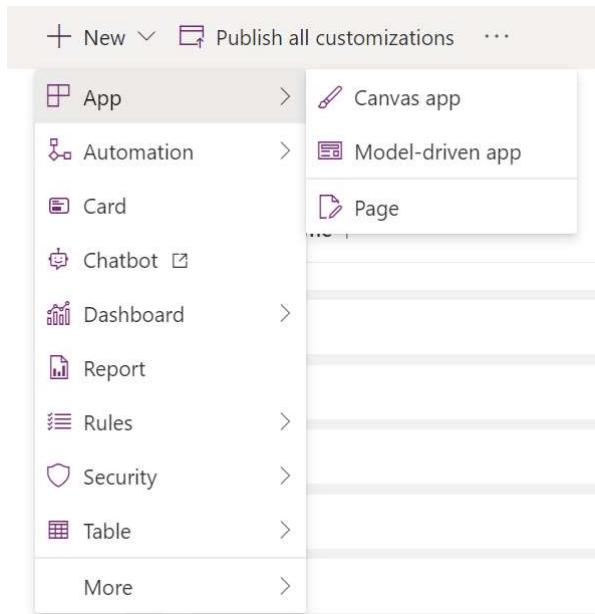
We can now use this PCF control inside our canvas app:



### TRES EN RAYA



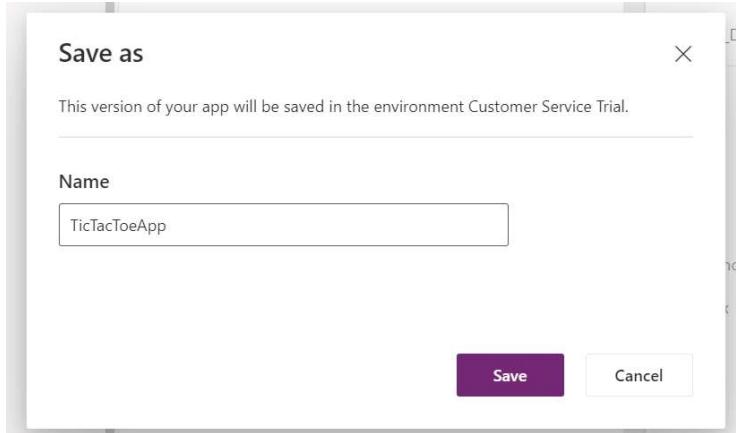
The same applies to custom pages. We can add a new App->Page:



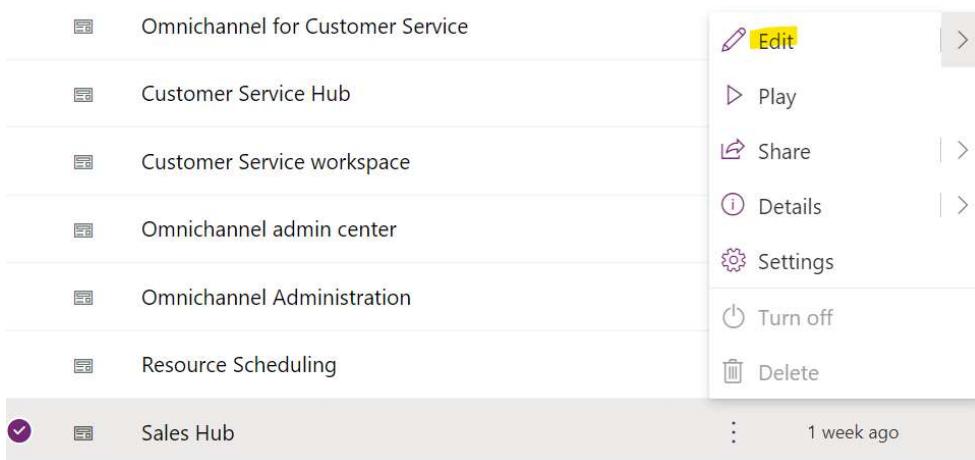
And import and add the new PCF control to the page. Let's add Tic Tac Toe to a custom page:

A screenshot of the Power Apps canvas editor. A custom page titled 'sampleProperty' is displayed. On the page, there is a 'TresEnRaya1' control, which is a 3x3 grid for the game. The control has properties set in the right-hand panel: Position (X: 297, Y: 114), Size (Width: 450, Height: 494), and Visible (On). The tooltip for the control is 'Un Tres en Raya hecho con React'. The left sidebar shows the 'Insert' tab with various controls like Label, Text box, and the newly added 'Tres En Raya' component under 'Popular'.

Save the app and publish it:



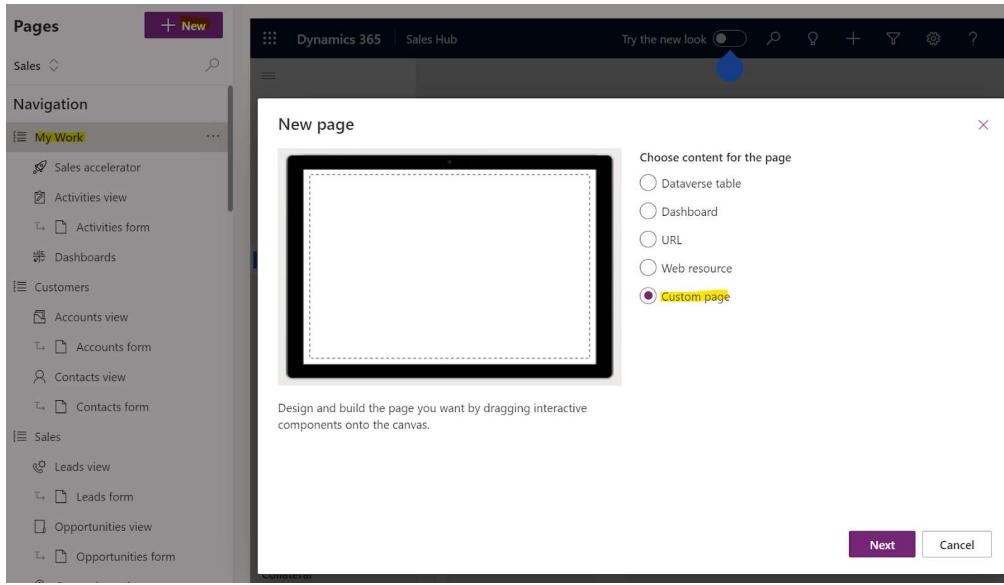
Now let's add the custom page to our model-driven app. We will add it to the site map. Let's edit the Sales Hub app:



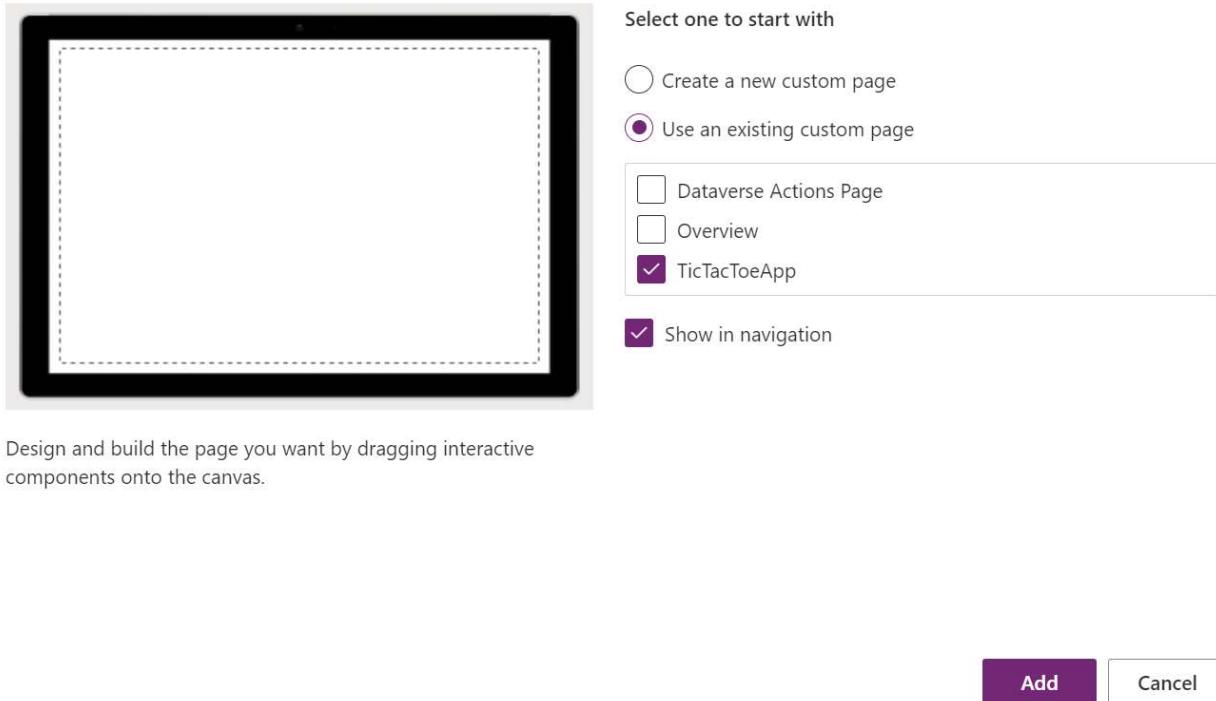
A list of applications in a Microsoft Dynamics 365 interface. The 'Sales Hub' item is selected and highlighted in grey. A context menu is open next to it, showing options: Edit (highlighted in yellow), Play, Share, Details, Settings, Turn off, and Delete. Other items in the list include: Omnichannel for Customer Service, Customer Service Hub, Customer Service workspace, Omnichannel admin center, Omnichannel Administration, Resource Scheduling, and Sales Hub.

App	Action
Omnichannel for Customer Service	
Customer Service Hub	
Customer Service workspace	
Omnichannel admin center	
Omnichannel Administration	
Resource Scheduling	
Sales Hub	⋮ 1 week ago

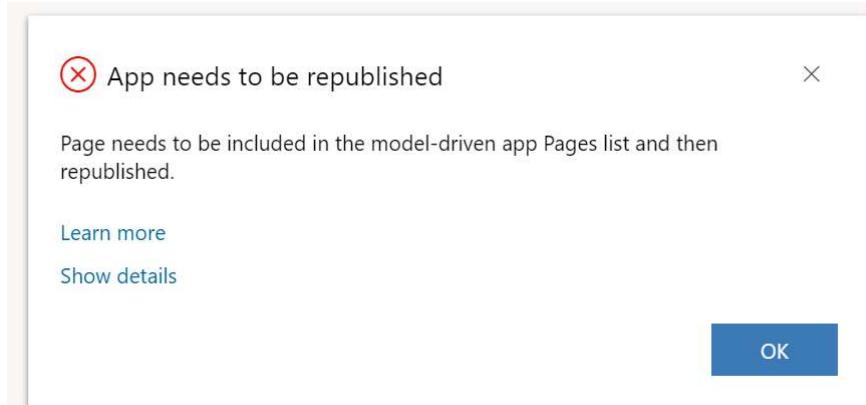
Under My Work, let's select New->Custom Page:



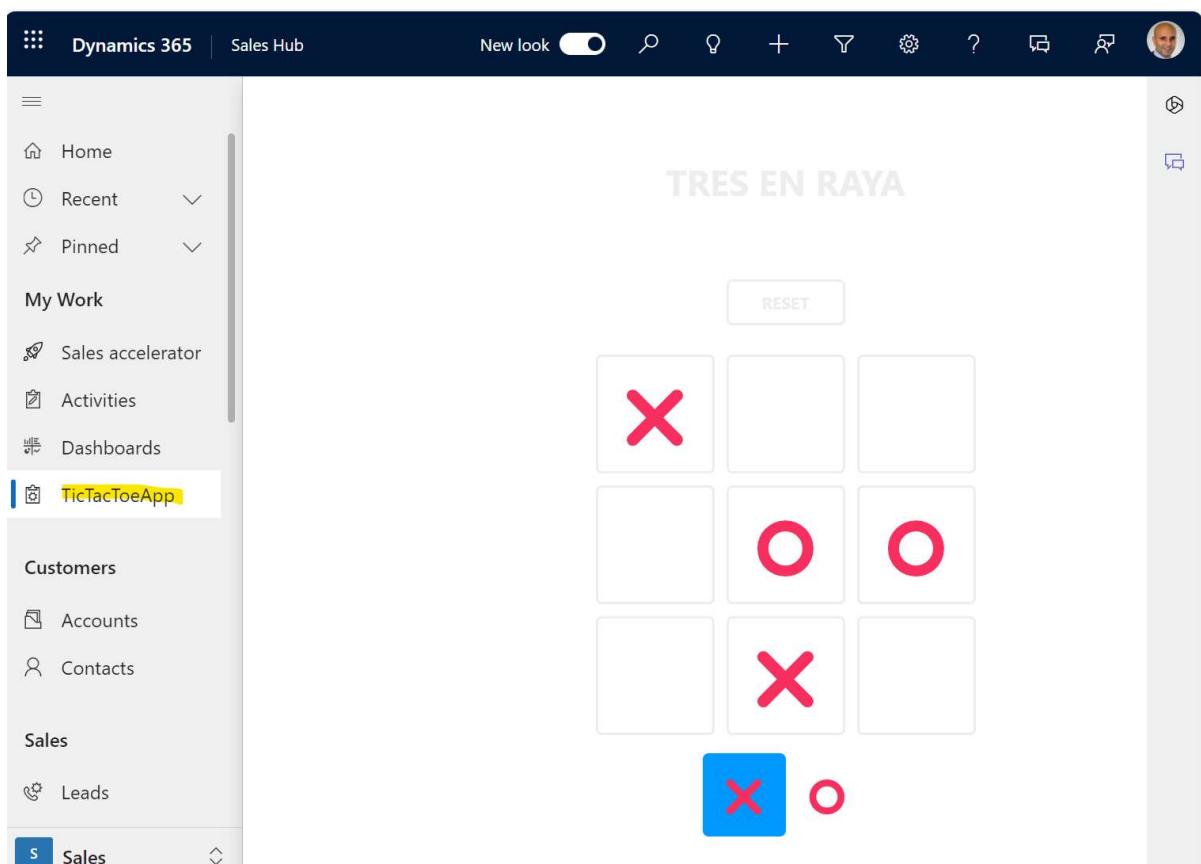
Select the Tic Tac Toe app and click Add:



You may see:



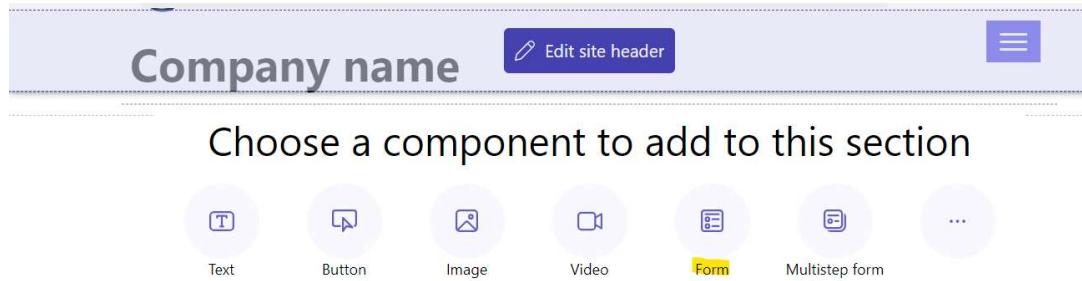
After saving and publishing, we see our custom page running our PCF control is available in the model-driven app:



# 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:

A screenshot of the "Describe a form to create it" interface. It shows a text input field with placeholder text "We'll create a form based on this description" and a character count of "0/250". Below the input field is a note: "Limit the collection of personal data to only what you need for a specific purpose. Make sure AI-generated content is accurate and appropriate before using it. [Learn more](#). [See preview terms](#)". At the bottom, there are three suggested form types: "User registration form", "Scholarship application form", and "Product customer support form".

Other ways to get started



And we will select our Account Dataverse form:

### Add a form

This form will collect information from your site visitors and store it in the table you choose. This form can be edited and reused.

**Form**

Data

On submit

CAPTCHA

Attachments

Choose a table \* ⓘ

Account

Select a form \*

Account

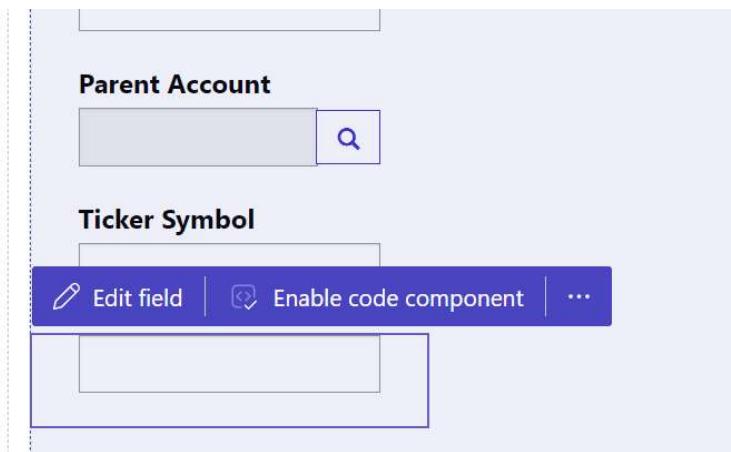
Name your copy of the selected form \* ⓘ

Account

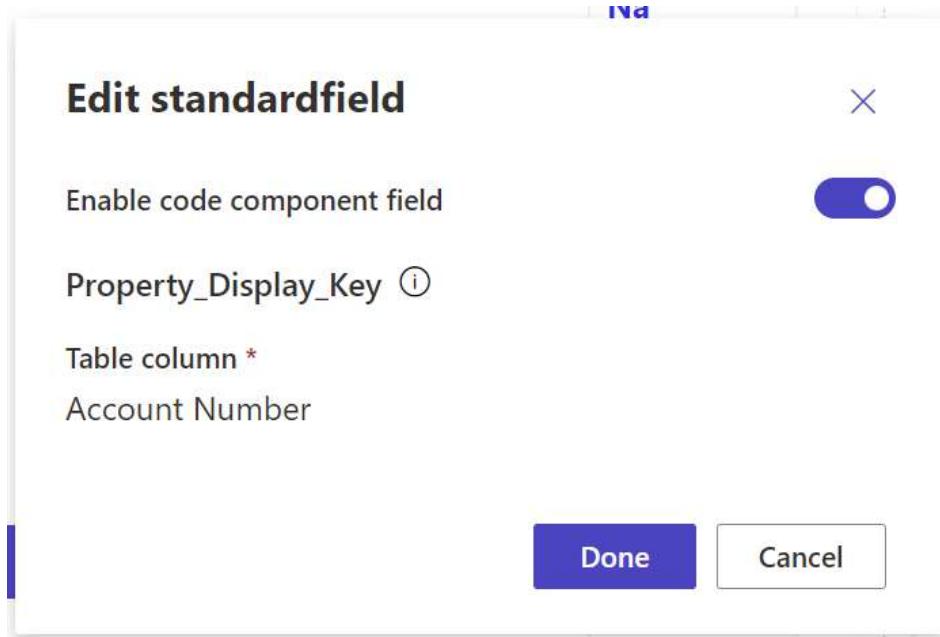
OK Cancel

The screenshot shows a modal dialog titled "Add a form". On the left, there's a sidebar with several tabs: "Form" (which is selected and highlighted in blue), "Data", "On submit", "CAPTCHA", and "Attachments". The main area contains descriptive text and input fields. It says, "This form will collect information from your site visitors and store it in the table you choose. This form can be edited and reused." Below this, under "Choose a table \* ⓘ", there's a search bar with the text "Account". Under "Select a form \*", there's a dropdown menu also showing "Account". Under "Name your copy of the selected form \* ⓘ", there's another text input field with "Account" typed into it. At the bottom right are two buttons: a blue "OK" button and a white "Cancel" button.

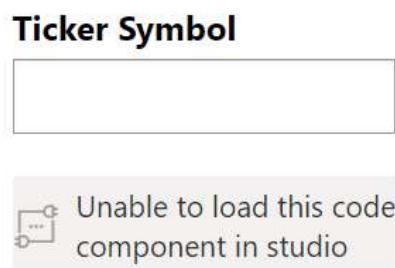
We see the field which contains our PCF code components, which was added previously to the form:



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):

## ACCOUNT INFORMATION

Account Name \*

Phone

  
Provide a telephone number

Fax

Website

Parent Account

Ticker Symbol

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.