Power Platform

A Step by Step Guide to Developing PCF Controls

Contents

Introduction

PowerApps Component Framework are used to create code components for model-driven apps and canvas apps (preview) in order to provide an enhanced user experience for users to view and work with data in forms, views and dashboards. PowerApps Component Framework controls can be used in the Unified Interface of Dynamics 365.

Prerequisites

The following steps are required in order to start developing PowerApps Component Framework custom controls.

Developer Command Prompt for Visual Studio 2017

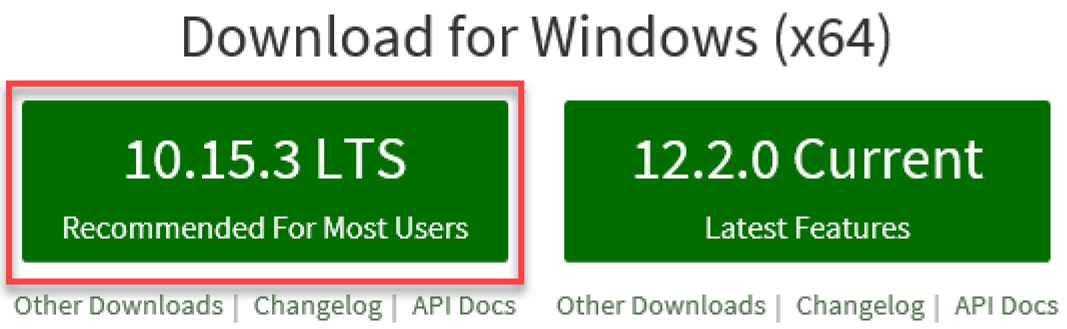
The Developer Command Prompt for Visual Studio 2017 is required to performs build tasks on your PowerApps Component Framework (PCF) custom control. For learning purposes you can download the Community Edition or a trial version of Visual Studio 2017, or if you already have Visual Studio 2017 Professional or Enterprise edition you can use that.

If you are using Visual Studio 2019, you can use either the Developer Command Prompt for Visual Studio 2019 or Developer PowerShell for Visual Studio 2019.

Download and Install Node.js

Node.js is an asynchronous even driven JavaScript runtime that is designed to build scalable network applications. In order to develop custom controls for Model-driven applications, we will need to use node.js in order to execute certain requests.

To download node.js, navigate to http://nodejs.org/en, and download the LTS version, which is the recommended version for most users, and what is required for this lab.


Download for Windows (x64)
10.15.3 LTS
Recommended For Most Users
12.2.0 Current
Latest Features
Other Downloads | Changelog | API Docs
Other Downloads | Changelog | API Docs

After you have downloaded installer file, go ahead and complete the installation of node.js.

PowerApps Command Line Interface

The Microsoft PowerApps CLI is a developer command line interface that enables developers to build custom components for PowerApps faster and more efficiently. You can learn more about the PowerApps CLI here, and then download it from nuget (https://www.nuget.org/packages/Microsoft.PowerApps.CLI).

If this is your first time installing or downloading Microsoft PowerApps CLI, you can download the msi file from the Microsoft web site, by following the link below:

https://aka.ms/PowerAppsCLI

After you have finished download the installation file, install the Command Line Interface.

Components

There are a few files that make up a PowerApps Component Framework custom control project. The two primary files that make up the control are the manifest file and the typescript file.

The manifest file is an Xml file which contains information about the namespace of the control, the properties of the control and the resources that make up the control which include the name of the typescript file, and any stylesheet or resource files that are included with part of the project.

The typescript file…

The stylesheet …

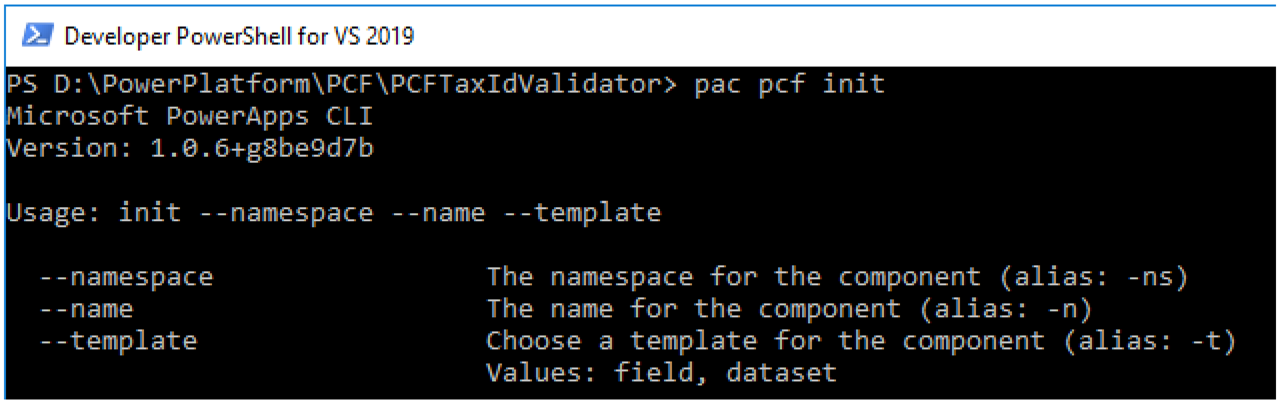
The resource file…

Manifest File

The manifest file is an Xml file which contains information about the namespace of the control, the properties of the control and the resources that make up the control which include the name of the typescript file, and any stylesheet or resource files that are included with part of the project.

The Control Element

The ***control*** element contains the namespace and the constructor which were provided when creating the component using the ***pac pcf init*** command.



```
PS D:\PowerPlatform\PCF\PCFTaxIdValidator> pac pcf init
Microsoft PowerApps CLI
Version: 1.0.6+g8be9d7b

Usage: init --namespace --name --template

  --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
```

The *display-name-key* is what will appear inside of properties of the control inside Dynamics form editor, when selecting the controls tab, and adding a different control to display on web, mobile or tablet.

The *description-key* is the description that will show up inside of the control properties within Dynamics form editor. The text below shown us the content of the control tag after adding the correct values to the display-name-key and description-key attributes.

```
<control namespace="PCFNamespace" constructor="PCFConstructor" version="0.0.1" display-name-key=
"PCFNamespace.PCFConstructor" description-key="Custom Control for PCF" control-type="standard">
```

The Property Element

By default, when a new PCF custom control project is created, only a single property is available. A PCF control can include multiple properties for different types of controls. The manifest file contains by default the following Xml for the property:

```
<property name="sampleProperty" display-name-key="Property_Display_Key" description-
key="Property_Desc_Key" of-type="SingleLine.Text" usage="bound" required="true" />
```

The following are the attributes of the property element:

- Name – this is the name of the property, which will be referred to from the typescript file in order to read and update content from the control.
- Display Name Key – this is the name of the property that would appear within Dynamics.
- Description Key – this is the description of the property that would appear within Dynamics
- Of Type – this attribute represents the data type of the property. When the usage of the ***of-type*** attribute is set to bound, then you bind it to the corresponding field within Dynamics. The of-type attribute can contains values such as Currency, DateAndTime, Decimal, Enum and more. The full list is available on the Microsoft Document site.
- Usage – this attribute can be either bound as previously stated, which means that this property is bound to a control within Dynamics or input which means that it will be used for read-only values.

The Resources Element

The final part is the resources. There are no changes required to the resources if your control will only be using the index.ts code file as it is already specified in the manifest file. The code element contains the location of the file where we will develop the custom control and build the HTML elements for it, as well as create the events of what happens to the control when interacted with such as on a click event or a change event.

The other two elements which are includes in the manifest file and are part of the parent resources element are css and resx. The css is the stylesheet that will be used for the control. This is not required, but if you want your control to blend in to the Unified Interface and look somewhat like other controls on the form, this is highly recommended.

You can also use existing styles that are available in Unified Interface and link them to the control itself. You will see this in the next sections. The resx is a file that contains localized content in order to display data for the control in additiona languages. For the purpose of this demo, we will not use it.

The unchanged resources element looks like this:

```
<resources>
   <code path="index.ts" order="1"/>
   <!-- UNCOMMENT TO ADD MORE RESOURCES
   <css path="css/filename.css" order="1" />
   <resx path="strings/filename.1033.resx" version="1.0.0" />
   -->
</resources>
```

Typescript File

Same as the automatic generation of the Manifest file, the index.ts file is also generated for us when we call the ***pac pcf init*** with the method signatures that we need to implement. Even if you don't know typescript, this is something that you should be able to grasp pretty quickly, especially for users with JavaScript experience.

The newly created component library (Typescript file) implements the following methods (which control the lifecycle of the code component): init, updateView, getOutputs, destroy. The getOutputs method is optional.

The init method

The init method is used to initialize the component instance. This is where the component can initialize particular actions or remote server calls. The init method accepts four parameters:

Parameter Name	Type	Required	Description
context	Context	yes	Includes the Input Properties containing the parameters, component metadata and interface functions
notifyOutputChanged	function	no	Notifies the framework that it has new output
state	Dictionary	no	State is saved from setControlState in previous session
container	HTMLDivElement	no	This is the div element to render

The updateView method

The updateView method is called whenever any value in the property bag is changed. This includes field values, datasets, global values (such as the container height and width), component metadata values (such as labels, visibility) and so on.

The updateView contains a single context parameter which contains the values that are mapped to the name that is defined in the manifest as well as in the utility functions.

The getOutputs method

The getOutput method is called by the framework before the component receives new data. It returns an object (as defined in the manifest). The object that is defined in the manifest file must be of type bound or output.

The destroy method

The destroy method is called when the component is removed from the DOM and is used for cleanup and release of any memory that the component is using.

Additional Resource Files

Add Content about Stylesheets and Resx files here.

Intro and Preparation

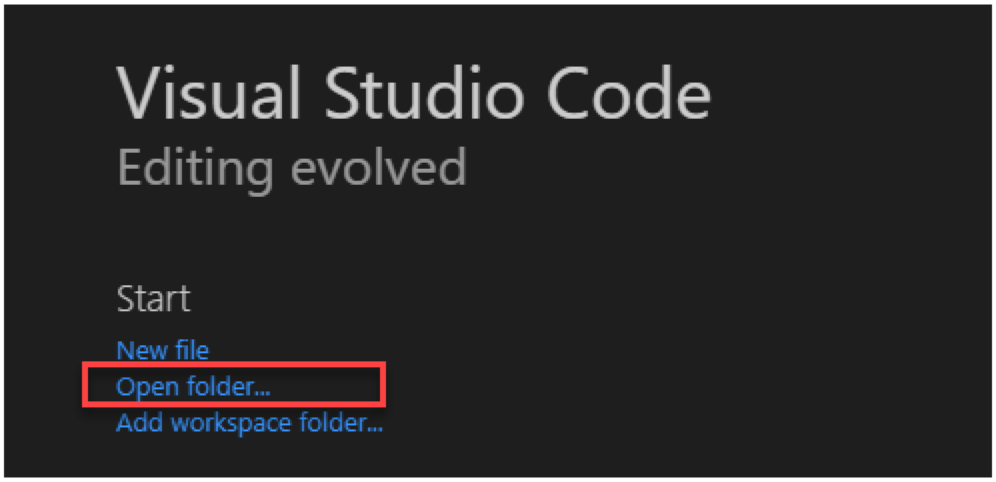
The next few chapters will go through the process from beginning to end of creating a custom control, testing it, creating a solution from the component that can be deployed into your Model-driven application, and finally customizing the form so that it will appear in the model driven application.

We will create in these next few chapters a credit card validation control, which will validate the user input on the CRM form, and will display the logo of the selected credit card or a validation error message if the credit card number is invalid.

Create the required files

To start this application, we will start by creating the required folders and files and calling the necessary commands to create the files. We will first start by creating the folder. You can create this anywhere on your local hard drive, but in our case, this was created in the following directory. We will use this location for later updating this solution.

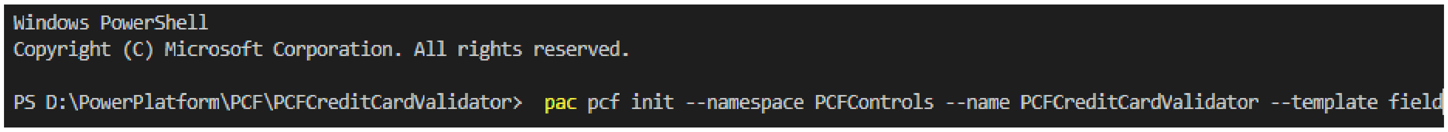
In our case we will use Visual Studio Code to perform that required action. Open Visual Studio Code and from the Home Page Start click on Open folder…



Select the location of the folder where you are going to start creating your project. In our case it will be D:\PowerPlatform\PCF\PCFCreditCardValidator.

Once we opened the folder, click on the View menu and select *Terminal*. This will allow us to execute PowerShell commands right from within Visual Studio Code.

We will use the command we mentioned earlier in previous chapters to initialize the project, by calling the ***pac pcf init*** command.

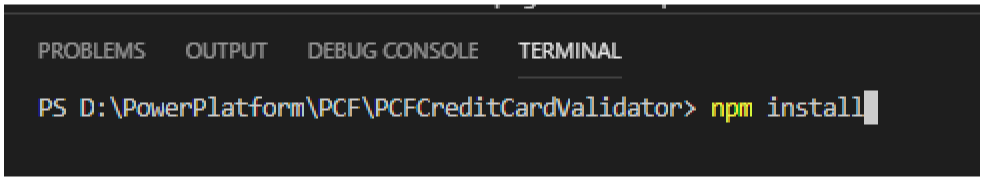


```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS D:\PowerPlatform\PCF\PCFCreditCardValidator>  pac pcf init --namespace PCFControls --name PCFCreditCardValidator --template field
```

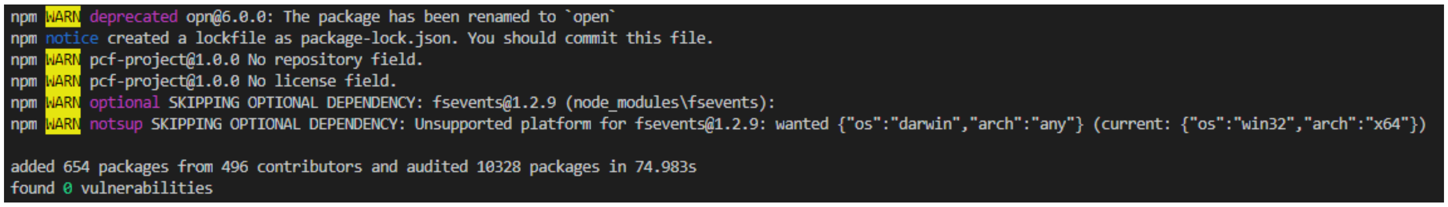
Install project dependencies

After the project has been created, you can navigate to the project directory, and you will notice that a subfolder with the name of the component has been created. The subfolder contains two files (ControlManifest.Input.xml and index.ts), which will be discussed later. A generated folder has also been created which includes the Manifest typescript file.

We can now go ahead and install all the required dependencies that are required. This steps can take a few minutes to complete. The command to install the dependencies is ***npm install***, as was shown in the screenshot from the previous task. While this task is executing you will see progression on your command prompt window, and you might see a few warnings or errors.



```
PS D:\PowerPlatform\PCF\PCFCreditCardValidator> npm install
```



```
npm WARN deprecated opn@6.0.0: The package has been renamed to `open`
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN pcf-project@1.0.0 No repository field.
npm WARN pcf-project@1.0.0 No license field.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.9 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.9: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

added 654 packages from 496 contributors and audited 10328 packages in 74.983s
found 0 vulnerabilities
```

You will notice in Visual Studio code that a node_modules folder was created with a lot of subfolders containing script files. This contains a large variety of available modules that can be added to your typescript project.

Update the Manifest File

After the solution has been initialized, the manifest file will look like the screenshot below. When working with the first component, we sometimes like to leave the comments in there, but as we start developing more PCF custom controls, we can remove them.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<manifest>
  <control namespace="PCFControls" constructor="PCFCreditCardValidator" version="0.0.1" displ
ay-name-key="PCFCreditCardValidator" description-
key="PCFCreditCardValidator description" control-type="standard">
    <!-
- property node identifies a specific, configurable piece of data that the control expects fr
om CDS -->
    <property name="sampleProperty" display-name-key="Property_Display_Key" description-
key="Property_Desc_Key" of-type="SingleLine.Text" usage="bound" required="true" />
    <!--
      Property node's of-type attribute can be of-type-group attribute.
      Example:
      <type-group name="numbers">
        <type>Whole.None</type>
        <type>Currency</type>
        <type>Decimal</type>
      </type-group>
      <property name="sampleProperty" display-name-key="Property_Display_Key" description-
key="Property_Desc_Key" of-type-group="numbers" usage="bound" required="true" />
    -->
    <resources>
      <code path="index.ts" order="1"/>
      <!-- UNCOMMENT TO ADD MORE RESOURCES
      <css path="css/PCFTaxIdValidator.css" order="1" />
      <resx path="strings/PCFTaxIdValidator.1033.resx" version="1.0.0" />
      -->
    </resources>
    <!-- UNCOMMENT TO ENABLE THE SPECIFIED API
    <feature-usage>
      <uses-feature name="Utility" required="true" />
      <uses-feature name="WebAPI" required="true" />
    </feature-usage>
    -->
  </control>
</manifest>
```

We will start by modifying the control element. The information here is based on what was entered in the ***pac pcf init*** command, so likely there is not much to modify, but we can change the description or other attributes as required. The final result here should look like this:

```
<control namespace="PCFControls" constructor="PCFCreditCardValidator" version="0.0.1" display
-name-key="PCFControls.PCFCreditCardValidator" description-
key="Credit Card Control for PCF" control-type="standard">
```

Next we will create a bound property for this control which will be based on a single line of text in our model driven application. We will provide the property name, display name key and description, the type of control, the usage (whether it is bound, input or output), and whether it is required. After making the changes it will look like this:

```
<property name="CreditCardNumber" display-name-
key="PCFCreditCardValidator_CreditCardNumber" description-key="Credit Card Number field" of-
type="SingleLine.Text" usage="bound" required="true" />
```

Finally, we will specify the resources that make up this project. In our case, we will be using a TypeScript file that will include the code elements, and a stylesheet that will define the look and feel of the control.

```
    <resources>
      <code path="index.ts" order="1"/>
      <css path="pcfcontrols.css" order="1" />
    </resources>
```

After making all the changes to the manifest file, and removing all of the comments, the manifest file will be short and clear.

```
<?xml version="1.0" encoding="utf-8" ?>
<manifest>
  <control namespace="PCFControls" constructor="PCFCreditCardValidator" version="0.0.1" displ
ay-name-key="PCFControls.PCFCreditCardValidator" description-
key="Credit Card Control for PCF" control-type="standard">
    <property name="CreditCardNumber" display-name-
key="PCFCreditCardValidator_CreditCardNumber" description-key="Credit Card Number field" of-
type="SingleLine.Text" usage="bound" required="true" />
    <resources>
      <code path="index.ts" order="1"/>
      <css path="pcfcontrols.css" order="1" />
    </resources>
  </control>
</manifest>
```

Code the Component File

We will start by looking at the file looks when it is only created. It will contain the four methods that we described earlier, and we will start by working with that file.

```
import {IInputs, IOutputs} from "./generated/ManifestTypes";

export class PCFCreditCardValidator implements ComponentFramework.StandardControl<IInputs, IO
utputs> {

    /**
     * Empty constructor.
     */
    constructor()
    {

    }

    /**
     * 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 con
tains values as set up by the customizer mapped to property names defined in the manifest, as
 well as utility functions.
     * @param notifyOutputChanged A callback method to alert the framework that the control h
as new outputs ready to be retrieved asynchronously.
     * @param state A piece of data that persists in one session for a single user. Can be se
t at any point in a controls life cycle by calling 'setControlState' in the Mode interface.
     * @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
    }


    /**
     * Called when any value in the property bag has changed. This includes field values, dat
a-
sets, global values such as container height and width, offline status, control metadata valu
es such as label, visible, etc.
     * @param context The entire property bag available to control via Context Object; It con
tains values as set up by the customizer mapped to names defined in the manifest, as well as
utility functions
     */
    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” or “output”
     */
    public getOutputs(): IOutputs
    {
        return {};
    }

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

The import and export portions of the file remain untouched, as well as the constructor. We will start by declaring variables above the construction that can be used as shared variables within the class. The list of global variables includes the PowerApps Component Framework required parameters (context, notify changes and the output HTML Div container), the HTML elements that will make up the control, and an event listener.

```
    // Required private parameters of the control
    private _context: ComponentFramework.Context<IInputs>;
    private _notifyOutputChanged: () => void;
    private _container: HTMLDivElement;

    // HTML Elements that will be used in the control
    private _creditCardNumberElement: HTMLInputElement;
    private _creditCardTypeElement: HTMLElement;
    private _creditCardErrorElement: HTMLElement;

    // String variable to store the credit card number
    private _creditCardNumber: string;

    // Event listener for changes in the credit card number
    private _creditCardNumberChanged: EventListenerOrEventListenerObject;
```

The init method

The init method contains all the initialization code for the typescript file This includes assigning values to global variables, binding control events to the delegate, skinning the control so that it looks like a control that is similar in look to the Unified Interface and finally creating the div element.

To assign the signature values to the public variables we will call the following lines of code. This will be similar in most of your projects. Within the init method we will start by adding the following code:

```
        // assigning environment variables.
        this._context = context;
        this._notifyOutputChanged = notifyOutputChanged;
        this._container = container;
```

Next we will add code to respond to the event of the control as shown in the function below.

```
        // Add control initialization code
        this._creditCardNumberChanged = this.creditCardChanged.bind(this);
```

Then, we will customize the control and add the attributes and an event to the new Input element and image element. The input element will be where the user will enter their credit card number on the CRM form, and the image element will be the image that will be displayed when the user enters the correct credit card number that corresponds to the rules of the card.

```
        // Add the textbox control, styling and event listener
        this._creditCardNumberElement = document.createElement("input");
        this._creditCardNumberElement.setAttribute("type", "text");
        this._creditCardNumberElement.setAttribute("class", "pcfinputcontrol");
        this._creditCardNumberElement.addEventListener("change", this._creditCardNumberChange
d);
```

```
        // Add the image control that will display the logo of the credit card
        this._creditCardTypeElement = document.createElement("img");
        this._creditCardTypeElement.setAttribute("class", "pcfimagecontrol");
        this._creditCardTypeElement.setAttribute("height", "24px");
```

We will also add HTML elements to display an error message under the HTML control in case the credit card number is invalid. The logic of the error uses similar styles as that of Dynamics 365 Unified Interface.

```
   // Add an error visual to show the error message when there is an invalid credit card
        this._creditCardErrorElement = document.createElement("div");
        this._creditCardErrorElement.setAttribute("class", "pcferrorcontroldiv");
        var creditCardErrorChild1 = document.createElement("label");
        creditCardErrorChild1.setAttribute("class", "pcferrorcontrolimage")
        creditCardErrorChild1.innerText = "?";

        var creditCardErrorChild2 = document.createElement("label");
        creditCardErrorChild2.setAttribute("class", "pcferrorcontrollabel")
        creditCardErrorChild2.innerText = "Invalid Credit Card Number entered.";

        this._creditCardErrorElement.appendChild(creditCardErrorChild1);
        this._creditCardErrorElement.appendChild(creditCardErrorChild2);
        this._creditCardErrorElement.style.display = "none";
```

Finally, we will add the three elements (text box, image and error HTML element) to the container that was defined and passed by the control.

```
        this._container.appendChild(this._creditCardNumberElement);
        this._container.appendChild(this._creditCardTypeElement);
        this._container.appendChild(this._creditCardErrorElement);
```

The updateView method

The updateView method is called when any value in the property bag changes, which includes field values, datasets, global values (such as container width and height), control metadata values (such as labels or visibility). The context parameter is the entire property bag that is available to the control via the context object. This parameter contains the values that are set up by the system customizer mapped to names defined in the manifest as well as utility functions.

For our credit card field, we will get the logical name of the field, and set the value to its formatted value.

```
        var crmCreditCardNumberAttribute = this._context.parameters.CreditCardNumber.attribut
es.LogicalName;

         // @ts-ignore
        Xrm.Page.getAttribute(crmCreditCardNumberAttribute).setValue(this._context.parameters
.CreditCardNumber.formatted);
```

Notice the @ts-ignore comment above the Xrm.Page.getAttribute call. When we use the Xrm namespace calls that are used in JavaScript in Dynamics, we have to tell the compiler to ignore these calls so that it does not error out.

We also add to the updateView method a notification in case there is a validation error. This is a notification that appears on the top of the form, and disappears after a few seconds.

```
        if (this._creditCardErrorElement.style.display != "none")
        {
            var message = "The credit card number is not valid.";
            var type = "ERROR";  //INFO, WARNING, ERROR
            var id = "9444";  //Notification Id
            var time = 4000;  //Display time in milliseconds

            // @ts-ignore
            Xrm.Page.ui.setFormNotification(message, type, id);

            //Wait the designated time and then remove the notification
            setTimeout( function () {
            // @ts-ignore
                Xrm.Page.ui.clearFormNotification(id);
            }, time );
```

You can modify this as needed.

The getOutputs method

The getOutputs method is called by the framework prior to the control receiving new data. It returns an object for a property that is marked as bound or output. The getOutput will return the Credit Card Number in this method class:

```
    public getOutputs(): IOutputs
    {
        return {
            CreditCardNumber: this._creditCardNumber
        };
    }
```

The event handler method

The event method handles the validation of the code and will call the notify output changed event at the end in order to handle the processing. In our particular case, we are looking the check that the credit card number is valid and that the length is correct for that type of credit card.

We check using regular expressions if this is a Visa, Mastercard, American Express or Discover card. If the card is one of the above, we will display the image of the card next to the text control. If the card is not valid, we will display an error message.

At the end of the method we call the notifyOutputChanged to let the notify the component that it needs to update the output.

```
    public creditCardChanged(evt: Event):void
    {
        // Variable declarations
        var _regEx: RegExp | null = null;
        var imageUrl = "";
        var isValid = false;

        var creditCardNumber = this._creditCardNumberElement.value;

        // Does Credit Card have a value
        if (creditCardNumber != null && creditCardNumber.length > 0)
        {
            // Is this a Visa card?
            _regEx = new RegExp('^4[0-9]{12}(?:[0-9]{3})?$');
            if (_regEx.test(creditCardNumber))
            {
                isValid = true;
                imageUrl = "https://xyz.blob.core.windows.net/shared/imgs/ico/visa24.png";
            }

            if (!isValid)
            {
                // Is this a Mastercard card?
                _regEx = new RegExp('^(?:5[1-5][0-9]{2}|222[1-9]|22[3-9][0-9]|2[3-6][0-
9]{2}|27[01][0-9]|2720)[0-9]{12}$');
                if (_regEx.test(creditCardNumber))
                {
                    isValid = true;
                    imageUrl = "https://xyz.blob.core.windows.net/shared/imgs/ico/mc24.png";
                }
            }
            if (!isValid)
            {
                // Is this an American Express card?
                _regEx = new RegExp('^3[47][0-9]{13}$');
                if (_regEx.test(creditCardNumber))
                {
                    isValid = true;
                    imageUrl = "https://xyz.blob.core.windows.net/shared/imgs/ico/amex24.png"
;
                }
            }

            if (!isValid)
            {
                // Is this a Discover card?
                _regEx = new RegExp('6(?:011|5[0-9]{2})[0-9]{12}');
                if (_regEx.test(creditCardNumber))
                {
                    isValid = true;
                    imageUrl = "https://xyz.blob.core.windows.net/shared/imgs/ico/disc24.png"
;
                }
            }


            // Is the card number entered valid?
            if (isValid)
            {
                this._creditCardTypeElement.setAttribute("src", imageUrl);
                this._creditCardNumber = creditCardNumber;
                this._creditCardErrorElement.style.display = "none";
            }
// Not valid
            else
            {
                this._creditCardTypeElement.setAttribute("src", "https://xyz.blob.core.window
s.net/shared/imgs/ico/redwarning.png");
                this._creditCardNumberElement.removeAttribute("src");
                this._creditCardNumber = "";
                this._creditCardErrorElement.style.display = "block";
            }
        }
        this._notifyOutputChanged();
    }
```

The destroy method

The destroyer method is used in many programming language to clean up control, remote calls or listeners that might still be consuming resources.

Below is the code for the destroy methods, which in our case is used only to remove the event listener change event from the bound control.

```
    public destroy(): void
    {
        // Add code to cleanup control if necessary
        this._creditCardNumberElement.removeEventListener("change", this._creditCardNumberCha
nged);
    }
```

Design the Stylesheet

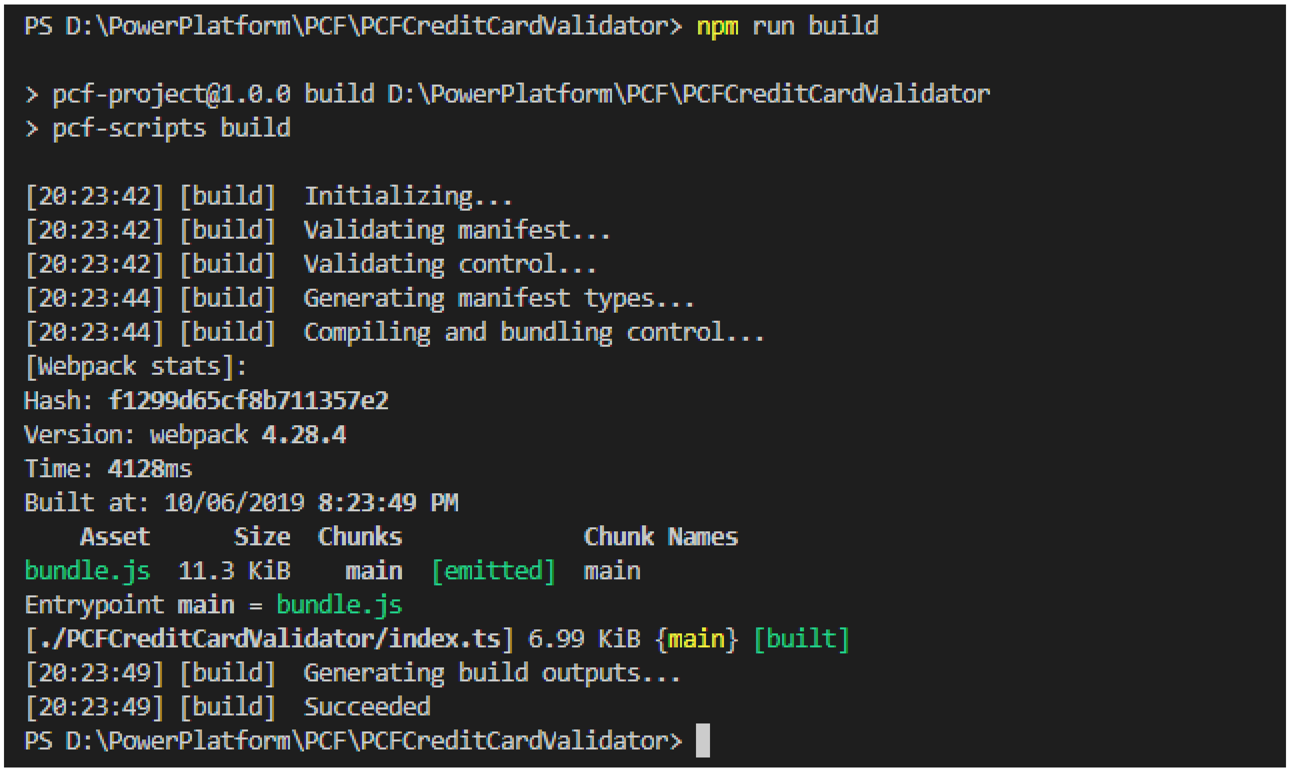
The following styles are used by the control, and set to display the control in a way that is similar to the way controls look in Model-Driven (Unified Interface) applications. This includes the control, image and error message.

```css
.pcfinputcontrol
{
  border-color: transparent;
  padding-right: 0.5rem;
  padding-left: 0.5rem;
  padding-bottom: 0px;
  padding-top: 0px;
  color: rgb(0,0,0);
  box-sizing: border-box;
  border-style: solid;
  border-width: 1px;
  line-height: 2.5rem;
  font-weight:600;
  font-size: 1rem;
  height: 2.5rem;
  margin-right: 0px;
  margin-left: 0px;
  text-overflow: ellipsis;
  width: 90%
}

.pcfinputcontrol:hover
{
  border-color: black;
}

.pcfimagecontrol
{
  padding-left: 5px;
  vertical-align: middle;
}

.pcferrorcontroldiv
{
  display: inline-flex;
  padding-top: 0.5rem;
  padding-bottom: 0.5rem;
  padding-left: 0.5rem;
  padding-right: 0.5rem;
  background-color: rgba(191, 9, 0, 0.075)
}

.pcferrorcontrolimage
{
  font-family:  "Dyn CRM Symbol", "Segoe MDL2 Assets";
  font-weight: 600;
  font-size: 1rem;
  line-height: 2rem;
  color: rgb(191, 9, 0);
  padding-right: 0.5rem;
}

.pcferrorcontrollabel
{
  display: initial;
  color: rgb(191, 9, 0);
  font-weight: 600;
  font-size: 1rem;
  font-family: SegoeUI-Semibold, "Segoe UI Semibold", "Segoe UI Regular", "Segoe UI";
  line-height: 2rem;
  word-break: break-word;
}
```

Build and Test the control

A PCF control is deployed into a Model-Driven app as a solution with its relevant components. The first step that needs to happen is for us to compile/build the code, and test it outside of Dynamics to see if it behaves properly.

We can still use the Terminal command ***npm run build*** to build the component and make it available for testing

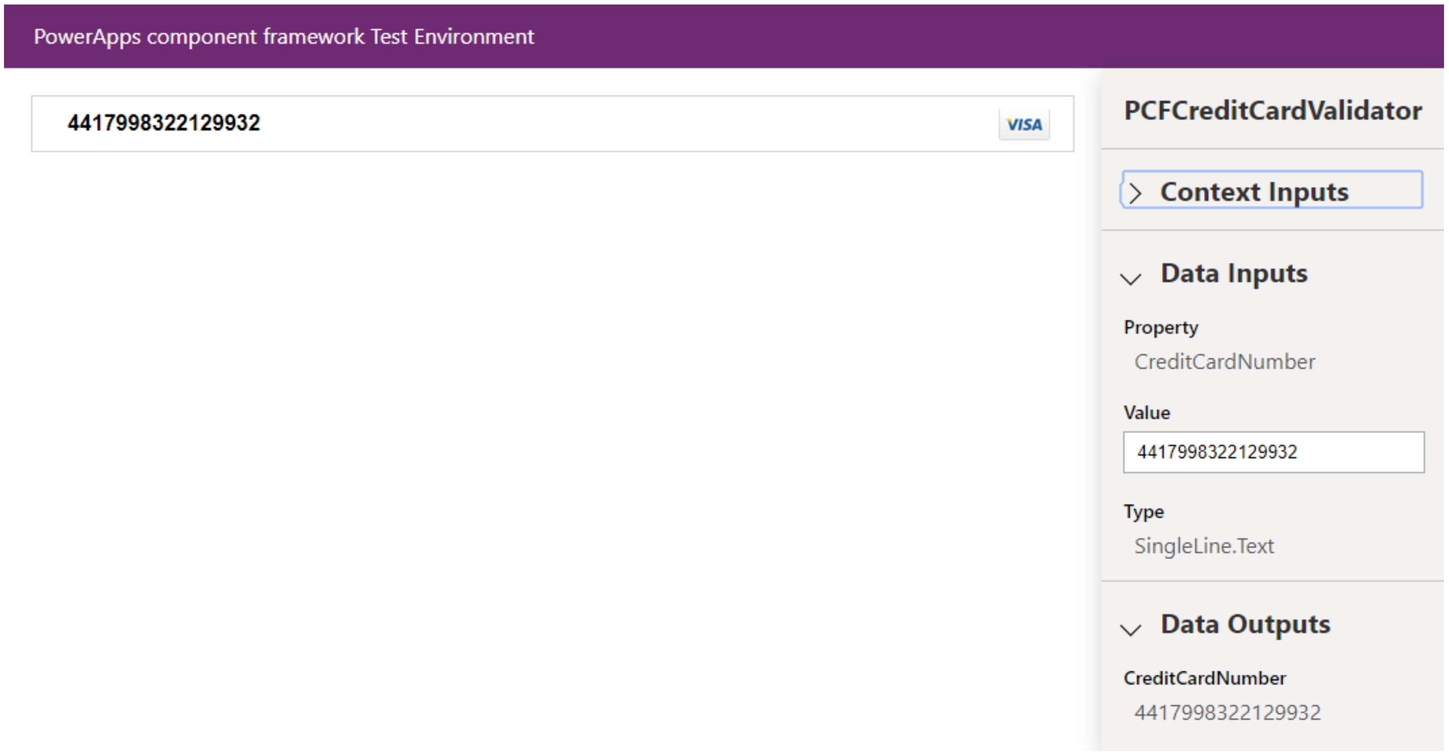


```
PS D:\PowerPlatform\PCF\PCFCreditCardValidator> npm run build

> pcf-project@1.0.0 build D:\PowerPlatform\PCF\PCFCreditCardValidator
> pcf-scripts build

[20:23:42] [build]  Initializing...
[20:23:42] [build]  Validating manifest...
[20:23:42] [build]  Validating control...
[20:23:44] [build]  Generating manifest types...
[20:23:44] [build]  Compiling and bundling control...
[Webpack stats]:
Hash: f1299d65cf8b711357e2
Version: webpack 4.28.4
Time: 4128ms
Built at: 10/06/2019 8:23:49 PM
    Asset      Size  Chunks             Chunk Names
bundle.js  11.3 KiB    main  [emitted]  main
Entrypoint main = bundle.js
[./PCFCreditCardValidator/index.ts] 6.99 KiB {main} [built]
[20:23:49] [build]  Generating build outputs...
[20:23:49] [build]  Succeeded
PS D:\PowerPlatform\PCF\PCFCreditCardValidator>
```

You will notice a **Succeeded** message at the end of the command if no errors are encountered.

Once the build has been completed, we can run the ***npm start*** command to start the testing the application in the PowerApps component framework Test Environment.

The first screenshot shows how the control will be display in the test environment when no errors occur.


PowerApps component framework Test Environment

4417998322129932

PCFCreditCardValidator

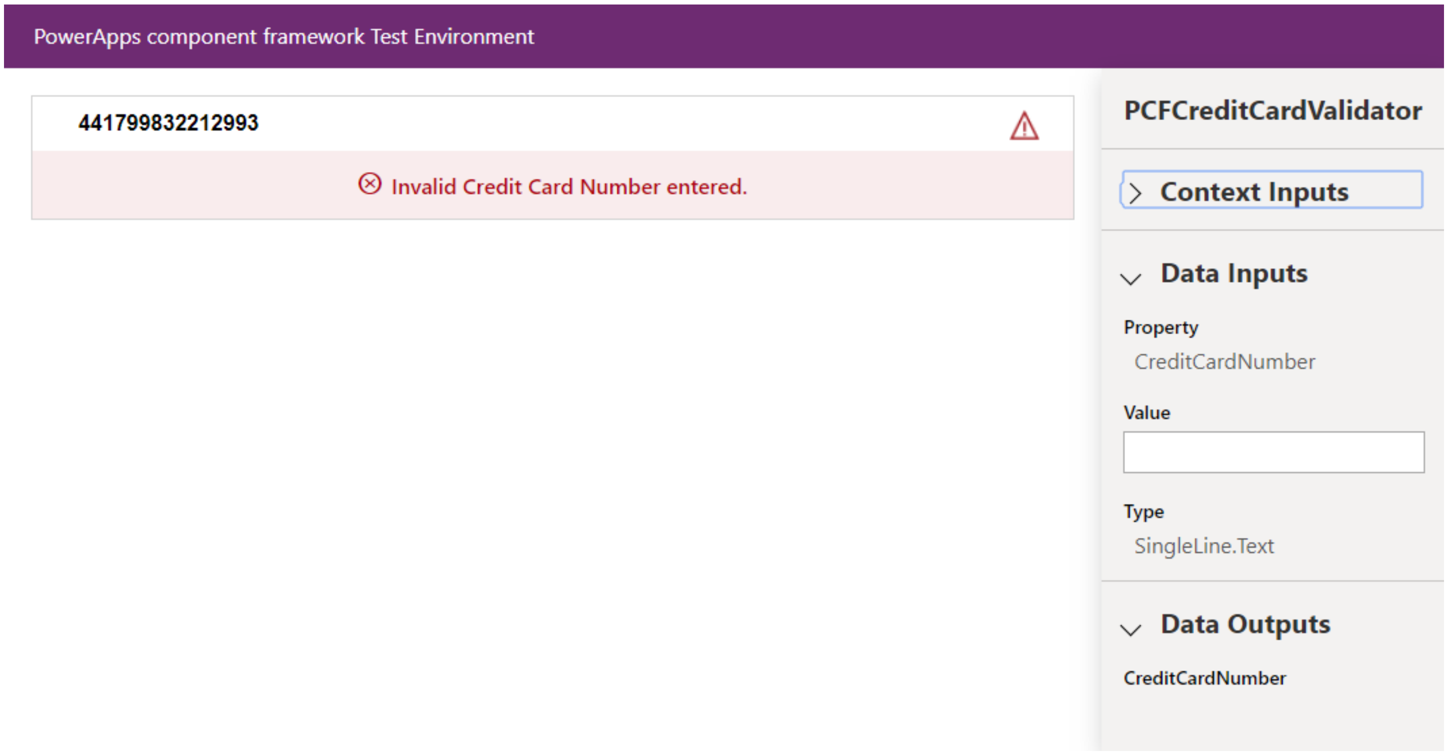
Property
CreditCardNumber

Value
4417998322129932

Type
SingleLine.Text

CreditCardNumber
4417998322129932

In case of errors, we will be able to see the way the control behaves:


PowerApps component framework Test Environment

441799832212993

Invalid Credit Card Number entered.

PCFCreditCardValidator

Property
CreditCardNumber

Value

Type
SingleLine.Text

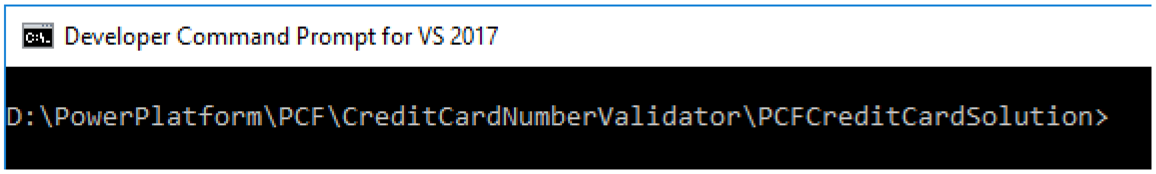
CreditCardNumber

The code that displays the error on the top of the Model-Driven app will only be displayed within the Model-Driven app and not in the PCF Test environment.

Build the solution

Now that the component has been built and tested, we are ready to prepare the solution that can be deployed into our Microsoft Dynamics 365 environment. The solution will be a zip file that can be installed via the import wizard into our Dynamics environment.

The first thing that we need to do is create a folder for the deployment of the solution. We will name the folder the same as the name we want to call our solution. In this case, I created a folder called PCFCreditCardSolution.



Developer Command Prompt for VS 2017

```
D:\PowerPlatform\PCF\CreditCardNumberValidator\PCFCreditCardSolution>
```

The following steps can still be done either from within Visual Studio Code or Developer Command Prompt/Developer PowerShell for Visual Studio.

Creating the solution files

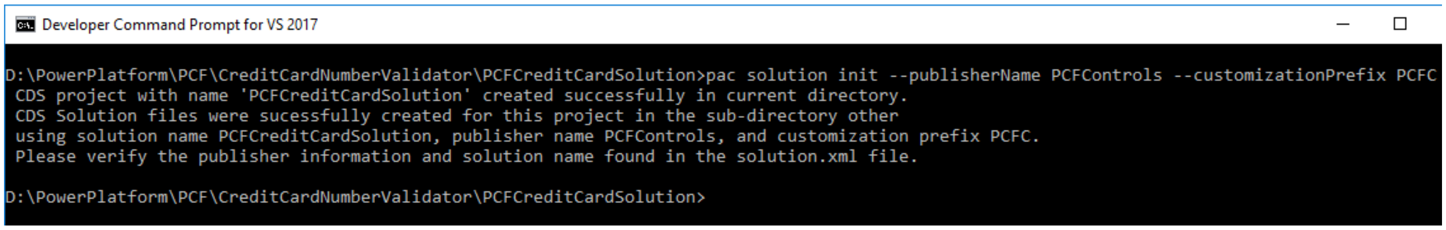
The ***pac solution init*** command will allow us to create the solution within our current folder. The command requires us to provide a publisher name and a customization prefix, same as with our CRM solutions. The full command is shown below:

pac solution init –-publisher-name [THE_NAME_OF_THE_PUBLISHER] –-publisher-prefix [PREFIX_TO_USE_WITH_PUBLISHER]

In our case I will created a publisher called PCFControls, and use the customizationPrefix PCFC:

pac solution init –-publisher-name PCFControls --customizationPrefix PCFC

The following screenshot shows the results after we run this command:

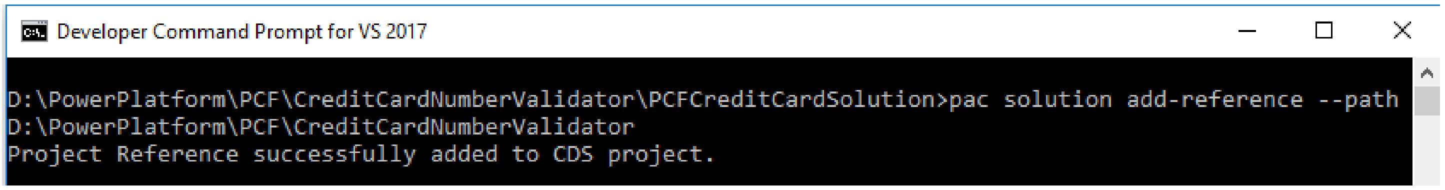


Developer Command Prompt for VS 2017

```
D:\PowerPlatform\PCF\CreditCardNumberValidator\PCFCreditCardSolution>pac solution init --publisherName PCFControls --customizationPrefix PCFC
 CDS project with name 'PCFCreditCardSolution' created successfully in current directory.
 CDS Solution files were sucessfully created for this project in the sub-directory other
 using solution name PCFCreditCardSolution, publisher name PCFControls, and customization prefix PCFC.
 Please verify the publisher information and solution name found in the solution.xml file.

D:\PowerPlatform\PCF\CreditCardNumberValidator\PCFCreditCardSolution>
```

Add the solution reference

After we have verified that the creation of the solution files was successful, we will need to add the custom control reference. We do this by calling the ***pac solution add-reference***, and providing the directory where the PowerApps Component Framework (PCF) project is located. In our case the directory of the D:\PowerPlatform\PCF\ CreditCardNumberValidator.



Developer Command Prompt for VS 2017

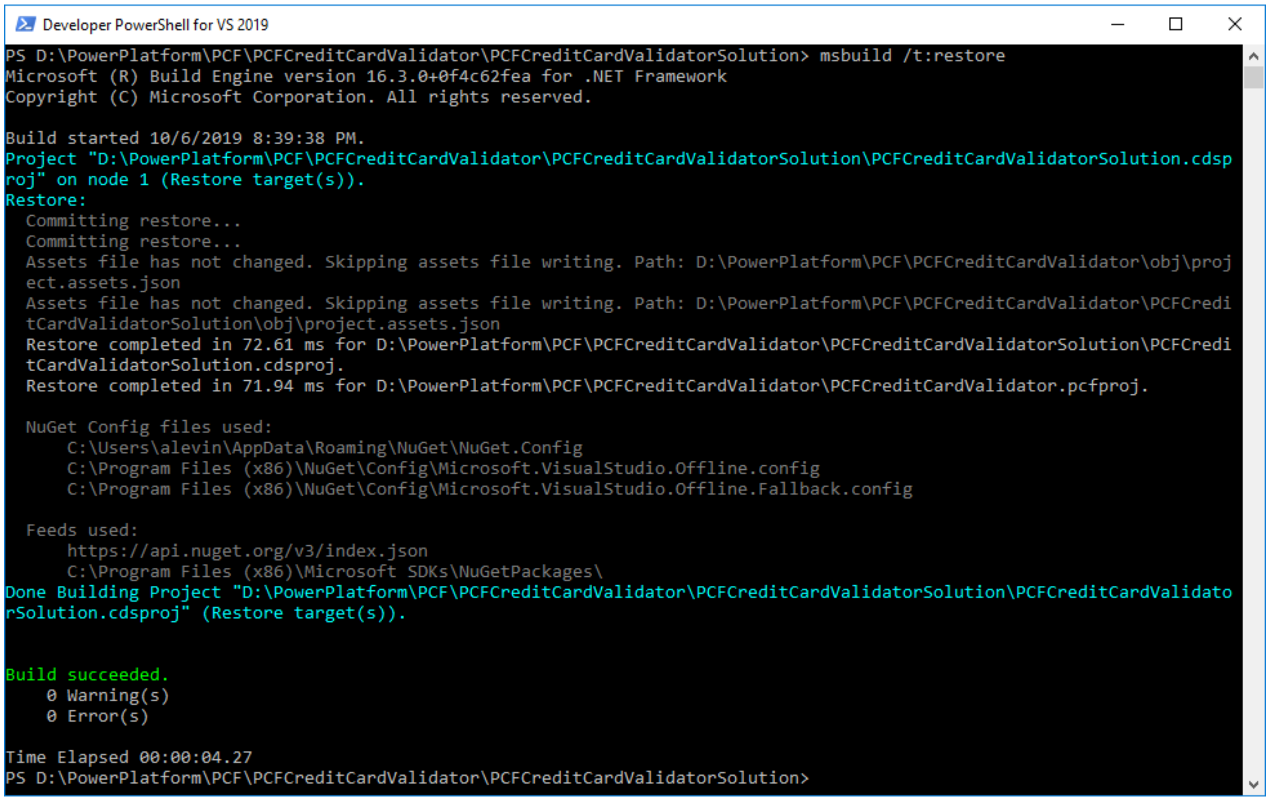
```
D:\PowerPlatform\PCF\CreditCardNumberValidator\PCFCreditCardSolution>pac solution add-reference --path
D:\PowerPlatform\PCF\CreditCardNumberValidator
Project Reference successfully added to CDS project.
```

You will receive a message that says: Project Reference successfully added to CDS project.

Running MSBuild

The final two steps to generate the solution zip files are to run the msbuild command. These steps can ONLY be run from within the Developer Command Prompt/Developer PowerShell for Visual Studio.

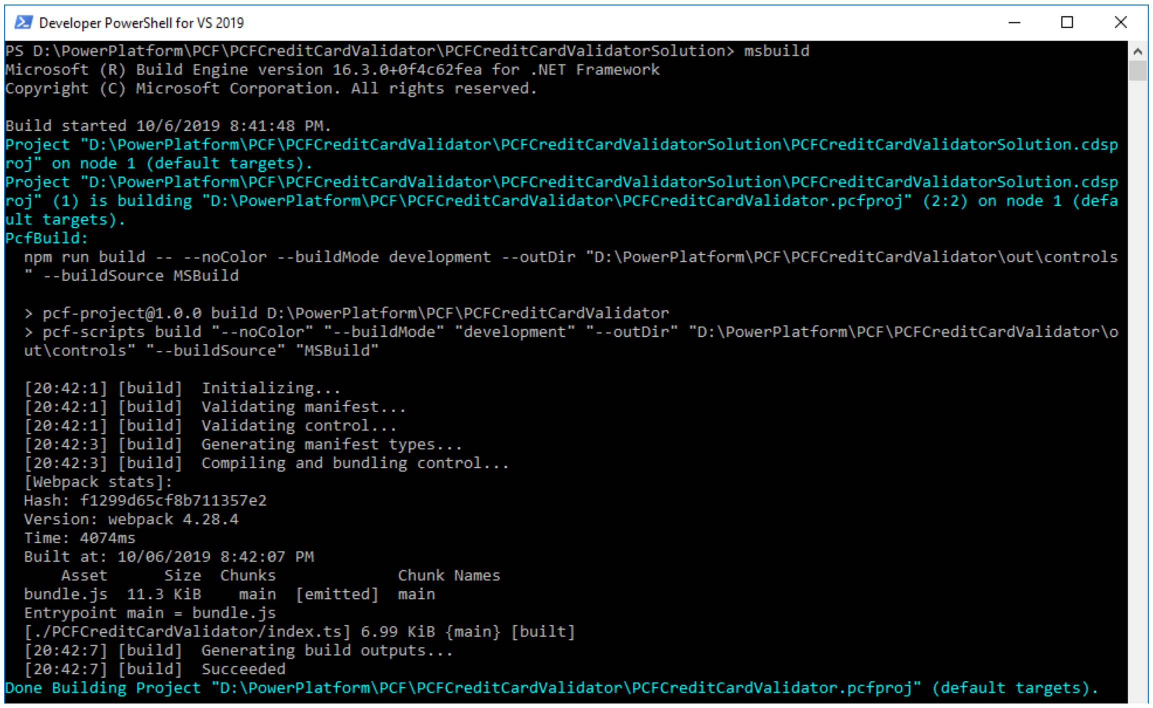
The first time we will run it with the /t:restore parameter, and then run it by itself.



Developer PowerShell for VS 2019

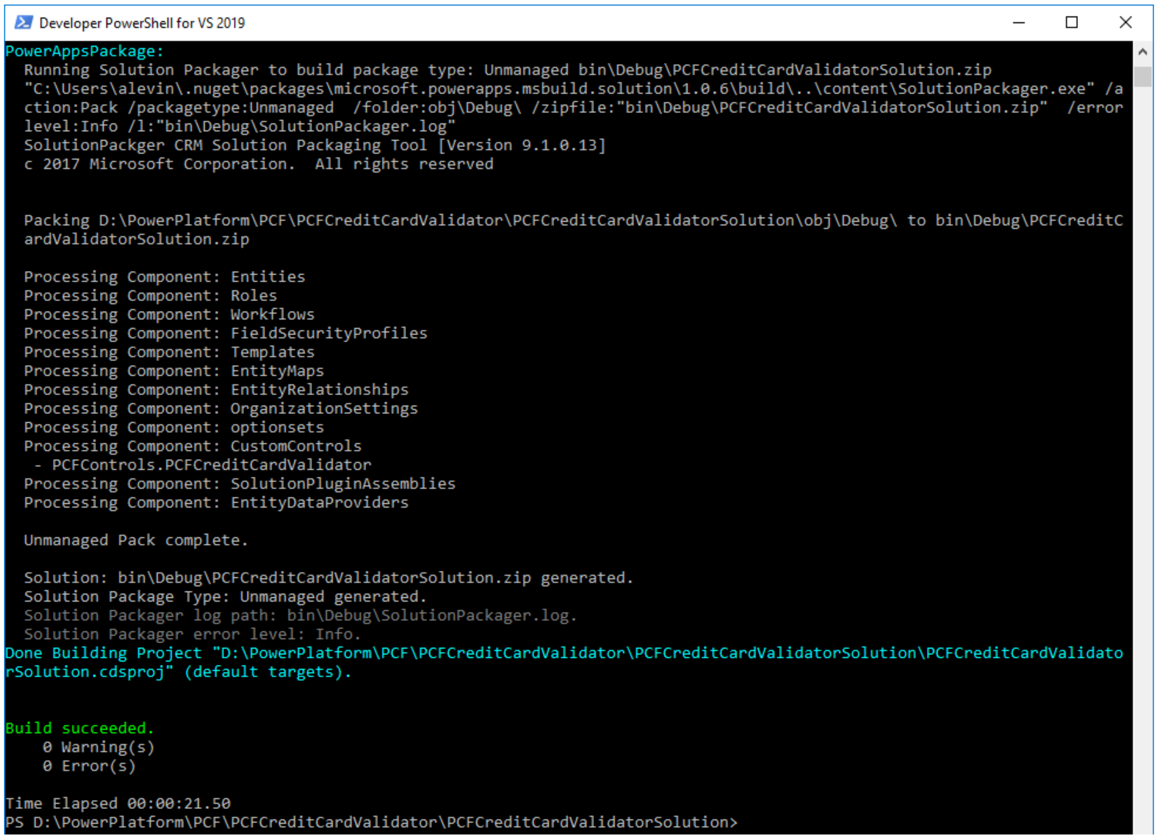
```
PS D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCreditCardValidatorSolution> msbuild /t:restore
Microsoft (R) Build Engine version 16.3.0+0f4c62fea for .NET Framework
Copyright (C) Microsoft Corporation. All rights reserved.

Build started 10/6/2019 8:39:38 PM.
Project "D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCreditCardValidatorSolution\PCFCreditCardValidatorSolution.cdsp
roj" on node 1 (Restore target(s)).
Restore:
  Committing restore...
  Committing restore...
  Assets file has not changed. Skipping assets file writing. Path: D:\PowerPlatform\PCF\PCFCreditCardValidator\obj\proj
  ect.assets.json
  Assets file has not changed. Skipping assets file writing. Path: D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCredi
  tCardValidatorSolution\obj\project.assets.json
  Restore completed in 72.61 ms for D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCreditCardValidatorSolution\PCFCredi
  tCardValidatorSolution.cdsproj.
  Restore completed in 71.94 ms for D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCreditCardValidator.pcfproj.

  NuGet Config files used:
      C:\Users\alevin\AppData\Roaming\NuGet\NuGet.Config
      C:\Program Files (x86)\NuGet\Config\Microsoft.VisualStudio.Offline.config
      C:\Program Files (x86)\NuGet\Config\Microsoft.VisualStudio.Offline.Fallback.config

  Feeds used:
      https://api.nuget.org/v3/index.json
      C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\
Done Building Project "D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCreditCardValidatorSolution\PCFCreditCardValidato
rSolution.cdsproj" (Restore target(s)).


Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:04.27
PS D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCreditCardValidatorSolution>
```

Next we will run msbuild in the same directory. We will be able to see the solution file required to import after that.



```
PS D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCreditCardValidatorSolution> msbuild
Microsoft (R) Build Engine version 16.3.0+0f4c62fea for .NET Framework
Copyright (C) Microsoft Corporation. All rights reserved.

Build started 10/6/2019 8:41:48 PM.
Project "D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCreditCardValidatorSolution\PCFCreditCardValidatorSolution.cdsp
roj" on node 1 (default targets).
Project "D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCreditCardValidatorSolution\PCFCreditCardValidatorSolution.cdsp
roj" (1) is building "D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCreditCardValidator.pcfproj" (2:2) on node 1 (defa
ult targets).
PcfBuild:
  npm run build -- --noColor --buildMode development --outDir "D:\PowerPlatform\PCF\PCFCreditCardValidator\out\controls
  " --buildSource MSBuild

  > pcf-project@1.0.0 build D:\PowerPlatform\PCF\PCFCreditCardValidator
  > pcf-scripts build "--noColor" "--buildMode" "development" "--outDir" "D:\PowerPlatform\PCF\PCFCreditCardValidator\o
  ut\controls" "--buildSource" "MSBuild"

  [20:42:1] [build]  Initializing...
  [20:42:1] [build]  Validating manifest...
  [20:42:1] [build]  Validating control...
  [20:42:3] [build]  Generating manifest types...
  [20:42:3] [build]  Compiling and bundling control...
  [Webpack stats]:
  Hash: f1299d65cf8b711357e2
  Version: webpack 4.28.4
  Time: 4074ms
  Built at: 10/06/2019 8:42:07 PM
      Asset      Size  Chunks             Chunk Names
  bundle.js  11.3 KiB    main  [emitted]  main
  Entrypoint main = bundle.js
  [./PCFCreditCardValidator/index.ts] 6.99 KiB {main} [built]
  [20:42:7] [build]  Generating build outputs...
  [20:42:7] [build]  Succeeded
Done Building Project "D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCreditCardValidator.pcfproj" (default targets).
```

The package will complete with a message of Build succeeded.



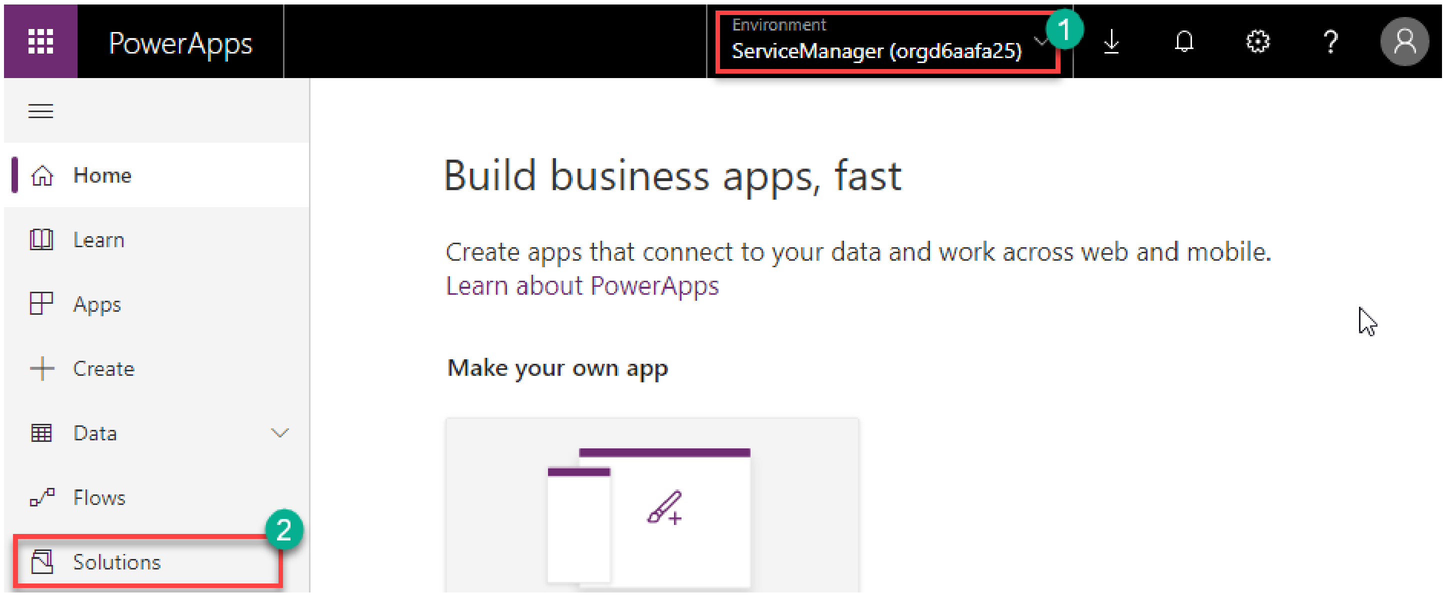
```
PowerAppsPackage:
  Running Solution Packager to build package type: Unmanaged bin\Debug\PCFCreditCardValidatorSolution.zip
  "C:\Users\alevin\.nuget\packages\microsoft.powerapps.msbuild.solution\1.0.6\build\..\content\SolutionPackager.exe" /a
  ction:Pack /packagetype:Unmanaged  /folder:obj\Debug\ /zipfile:"bin\Debug\PCFCreditCardValidatorSolution.zip"  /error
  level:Info /l:"bin\Debug\SolutionPackager.log"
  SolutionPackger CRM Solution Packaging Tool [Version 9.1.0.13]
  c 2017 Microsoft Corporation.  All rights reserved


  Packing D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCreditCardValidatorSolution\obj\Debug\ to bin\Debug\PCFCreditC
  ardValidatorSolution.zip

  Processing Component: Entities
  Processing Component: Roles
  Processing Component: Workflows
  Processing Component: FieldSecurityProfiles
  Processing Component: Templates
  Processing Component: EntityMaps
  Processing Component: EntityRelationships
  Processing Component: OrganizationSettings
  Processing Component: optionsets
  Processing Component: CustomControls
   - PCFControls.PCFCreditCardValidator
  Processing Component: SolutionPluginAssemblies
  Processing Component: EntityDataProviders

  Unmanaged Pack complete.

  Solution: bin\Debug\PCFCreditCardValidatorSolution.zip generated.
  Solution Package Type: Unmanaged generated.
  Solution Packager log path: bin\Debug\SolutionPackager.log.
  Solution Packager error level: Info.
Done Building Project "D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCreditCardValidatorSolution\PCFCreditCardValidato
rSolution.cdsproj" (default targets).


Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:21.50
PS D:\PowerPlatform\PCF\PCFCreditCardValidator\PCFCreditCardValidatorSolution>
```

Deploy the solution

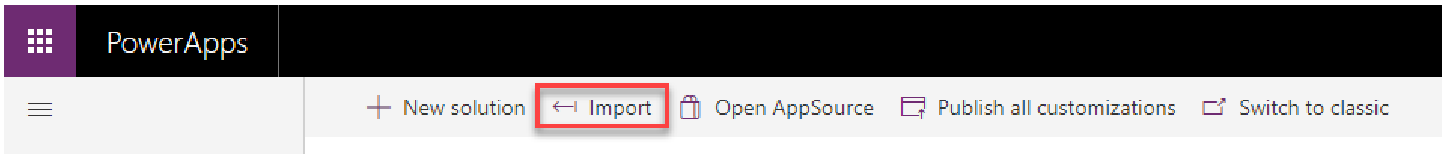
Now that the solution has been created we are ready to deploy it in our environment. We will first import the new solution into our environment, create the field inside our Model-driven application, and finally test it out.

Importing the solution

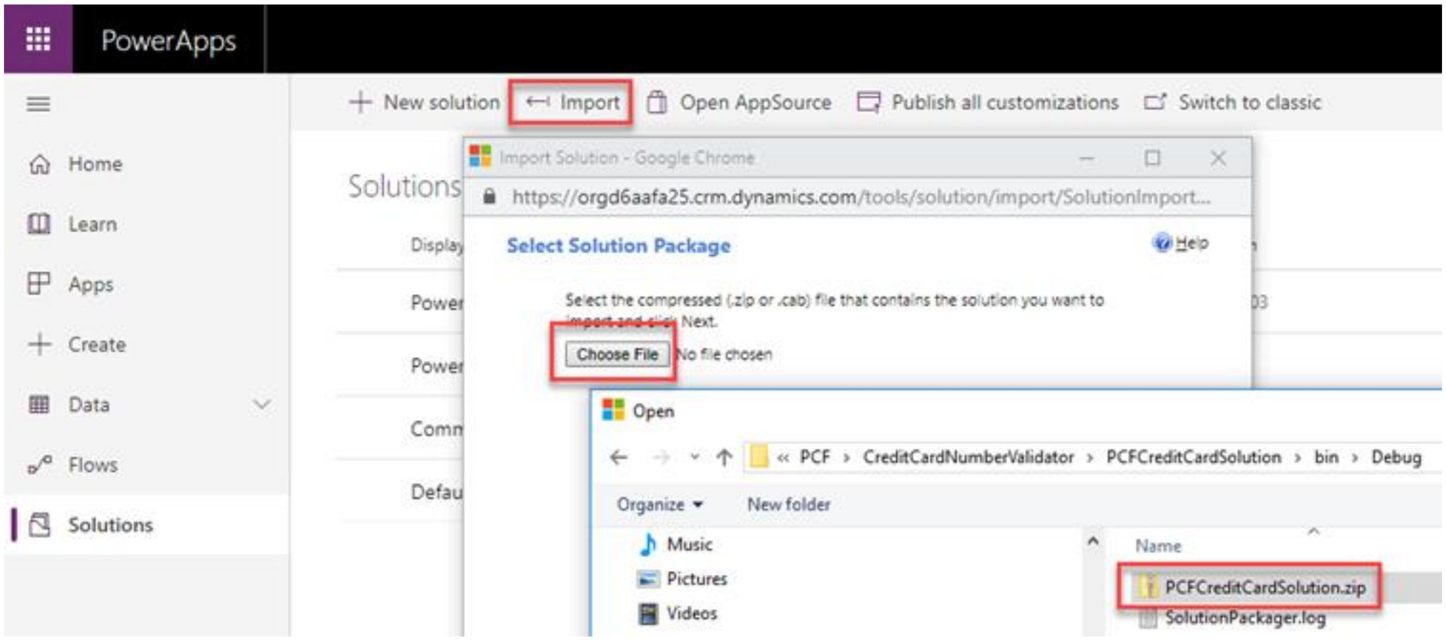
At the time of writing this guide, importing solutions is still using part of the old import functionality, but we will connect to our PowerApps application to implement this. Start by navigating to web.powerapps.com, and login with your admin credentials. Make sure that you are connected to the right environment, and then click on Solutions on the left side navigation.


PowerApps
Environment
ServiceManager (orgd6aafa25)
1
Home
Learn
Apps
Create
Data
Flows
Solutions
2
Build business apps, fast
Create apps that connect to your data and work across web and mobile.
Learn about PowerApps
Make your own app

Once you navigated to the solutions area, click on the Import button:

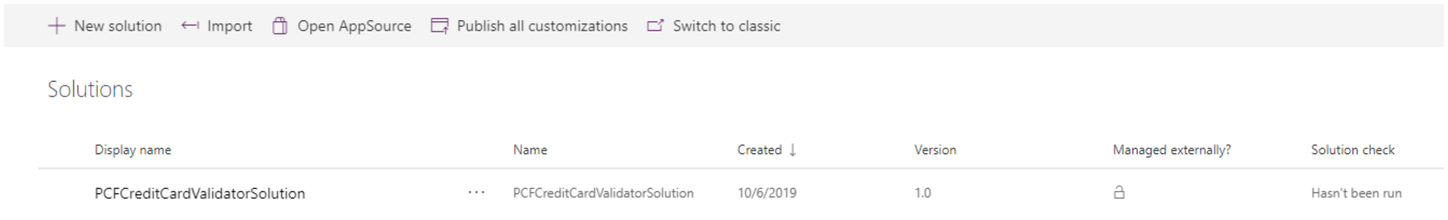


In the Select Solution package window, click on Choose File, and then navigate in the popup Open window to the directory containing your zip file. In our case this will be the bin/Debug folder under the PCFCreditCardSolution, as shown in the image below.


PowerApps
New solution
Import
Open AppSource
Publish all customizations
Switch to classic
Home
Learn
Apps
Create
Data
Flows
Solutions
Import Solution - Google Chrome
https://orgd6aafa25.crm.dynamics.com/tools/solution/import/SolutionImport...
Select Solution Package
Help
Select the compressed (.zip or .cab) file that contains the solution you want to import and click Next.
Choose File
No file chosen
Open
PCF › CreditCardNumberValidator › PCFCreditCardSolution › bin › Debug
Organize
New folder
Music
Pictures
Videos
Name
PCFCreditCardSolution.zip
SolutionPackager.log

Select the file and click Open in the Open window.

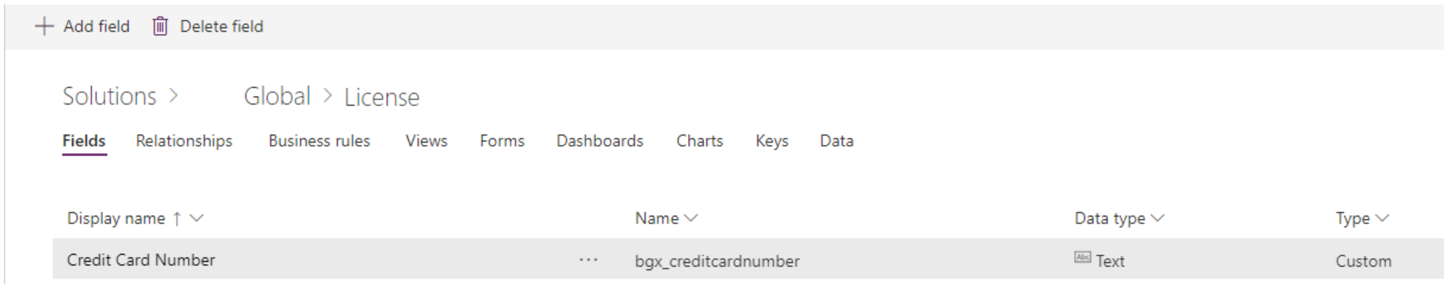
Click Next to see the solution information, and then click Import to start the Import process of the solution. Once the import has been completed, click on Publish All Customizations button to complete the process. When the publishing is done, click on Close.



Display name	Name	Created ↓	Version	Managed externally?	Solution check
PCFCreditCardValidatorSolution	PCFCreditCardValidatorSolution	10/6/2019	1.0	🔒	Hasn't been run

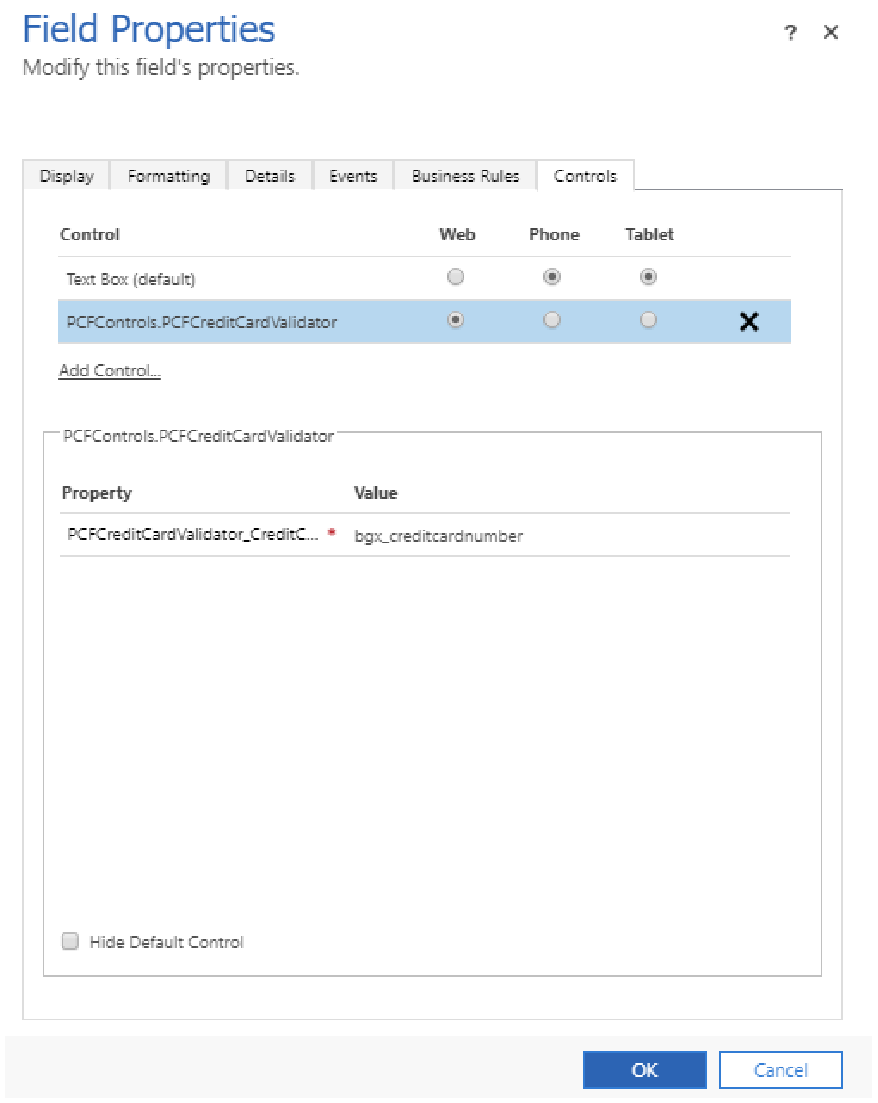
Customize the entity

Open a solution, and select the entity where you want to create the field to be used by the control. In our case we will create a field called CreditCardNumber of data type text.



Display name ↑	Name	Data type	Type
Credit Card Number	bgx_creditcardnumber	Text	Custom

Navigate to forms, and add the control to the form. Open the properties of the control, and in the Controls tab of the properties window, click on Add Control, select the control that you created and set it to display on Web, Phone and/or Tablet.



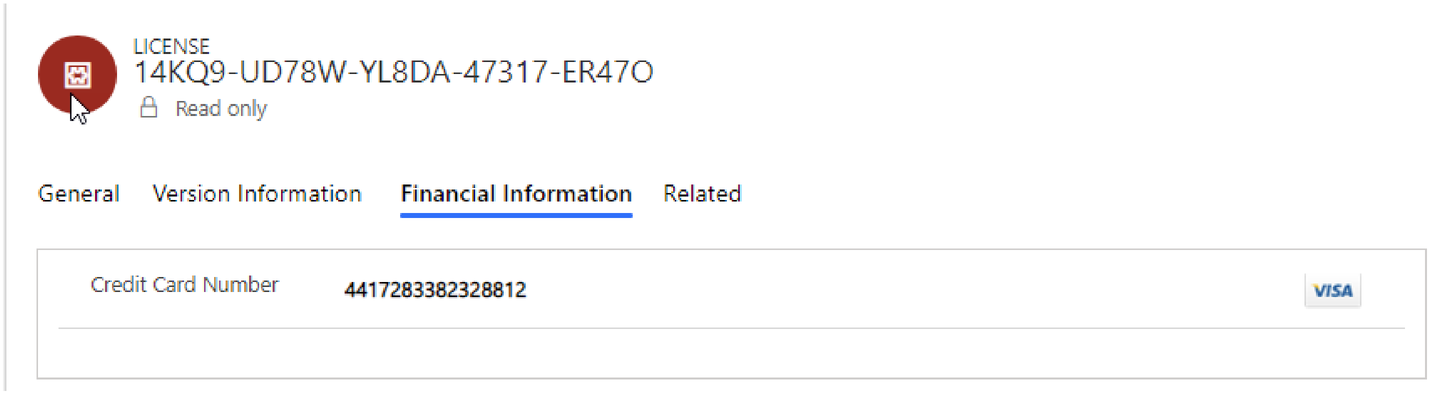
Control	Web	Phone	Tablet
Text Box (default)	○	●	●
PCFControls.PCFCreditCardValidator	●	○	○

Property	Value
PCFCreditCardValidator_CreditC... *	bgx_creditcardnumber

Test the deployed solution

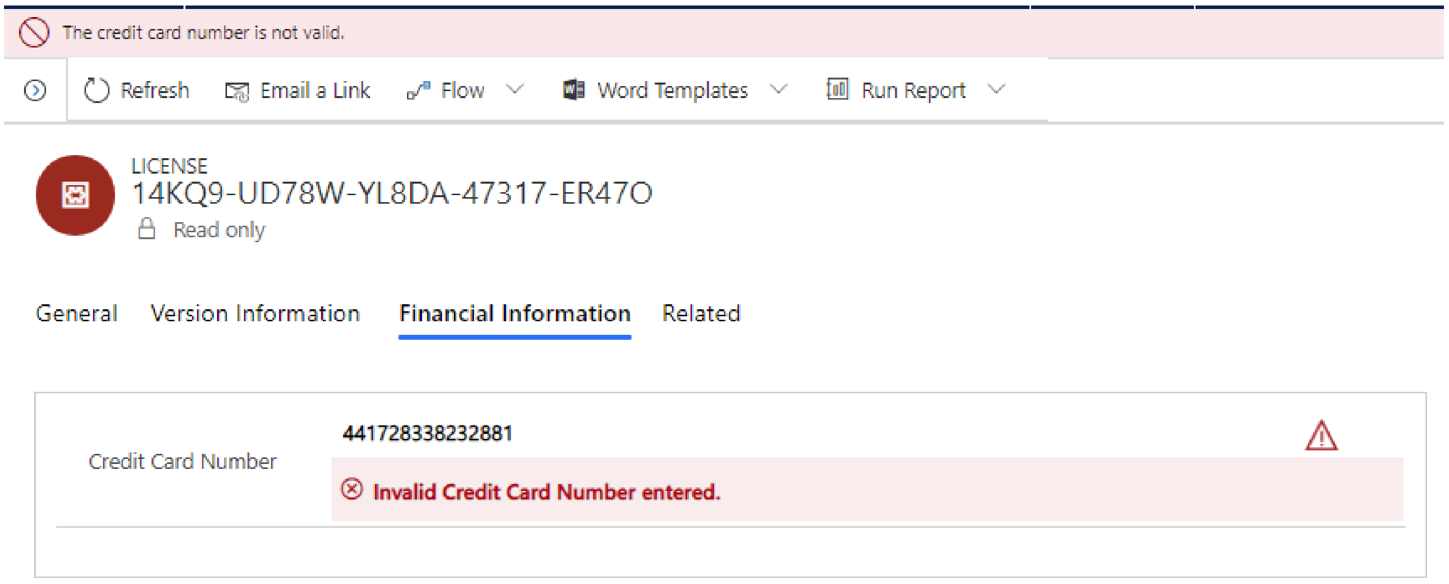
Now that the solution has been deployed, you can test it out.

Open the record type where you created the solution, and enter a valid credit card number. The control will display the correct credit card image.


LICENSE
14KQ9-UD78W-YL8DA-47317-ER47O
Read only

Credit Card Number 4417283382328812

Now change the credit card number, so that the number is no longer valid and you will see the error messages displayed in both the control and at the top of the form.


The credit card number is not valid.

LICENSE
14KQ9-UD78W-YL8DA-47317-ER47O
Read only

Credit Card Number 441728338232881

Invalid Credit Card Number entered.

Contributions

Author

Aric Levin, Business Applications MVP

MVP Contributions

Alex Shlega

Andrew Butenko

Andrew Ly

Guido Preite

Microsoft Contributions

Hemant Gaur

Resources

Community PCF Gallery

PowerApps Community PCF Framework Forum

PowerApps Component Framework Documentation