Power Platform in a Day

Module: PowerApps Component Framework

Hands-on Lab Step-by-Step

May 2019

Contents

PCF Custom Control

Lab Prerequisites

The following pre-requisite steps are required in order to develop our first PowerApps Component Framework Custom Control. The list below explains the steps described in the document: **00-AppInADay Lab Overview.pdf**, that is included in the lab package. Before beginning this lab, confirm that you have provisioned an environment where you still save your apps, flows and database entities and solutions.

IMPORTANT: Do not proceed ahead, before going through the pre-requisite steps

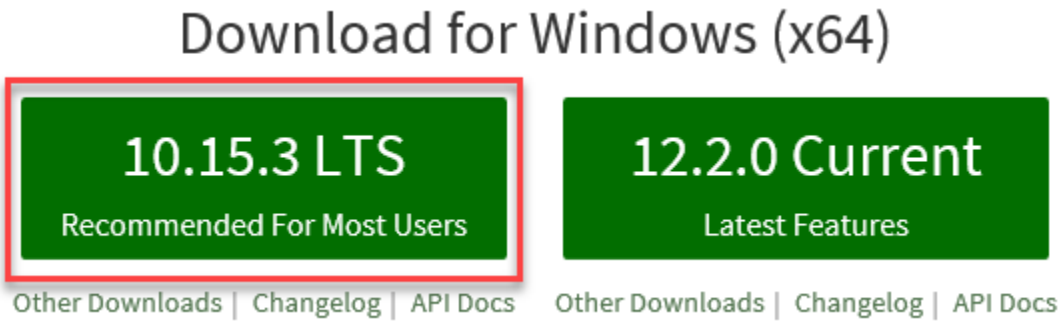
Task 1: Download Visual Studio 2017

The **Developer Command Prompt for VS2017** is required to perform the required tasks in this document. You can download the Community Edition or a trial version of Visual Studio Professional or Enterprise (https://visualstudio.microsoft.com) to continue with this document.

Task 2: 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.



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

Task 3: Download and Install PowerApps Command Line Interface (CLI)

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 nugget (https://www.nuget.org/packages/Microsoft.PowerApps.CLI) or by navigating directly to the link below:

http://download.microsoft.com/download/D/B/E/DBE69906-B4DA-471C-8960-092AB955C681/powerapps-cli-0.1.51.msi

After the download is complete, install the PowerApps Command Line Interface.

Introduction: Credit Card PCF Demonstration and Lab

The purpose of this lab is to demonstrate how to create a custom control and embed it into a Dynamics 365 Unified Interface form. In this demonstration and lab, we will create a custom credit card validator control, build it, test it and deploy it as a solution to our Dynamics 365 CE or CDS environment. Once the control has been deployed we will test it within our model-driven application to make sure that the functionality is correct.

Creating your PowerApps Component Framework project

In order to start creating your PowerApps Component Framework project, we need to use the Developer Command Prompt for VS 2017. You can open it by typing the name in the Cortana Search Bar or the Start button (under the Visual Studio 2017 menu).

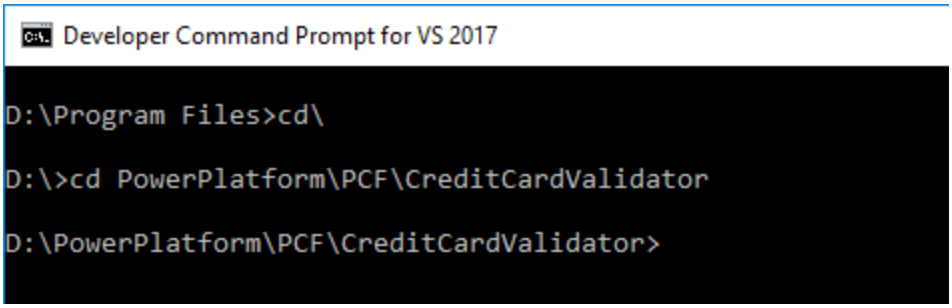
The project that we will be creating is a Credit Card Validation control where the user will enter a Credit Card number, and the system will determine type of credit card the user entered based on number rules.

Task 1: Create your Project Directory

Using Command Prompt or File Explorer, create the directory where you want to develop and build your Custom Control project. For the purpose of this training, let's create the following directory structure in your local drive:

C:\PowerPlatform\PCF\CreditCardValidator or
D:\PowerPlatform\PCF\CreditCardValidator (if you want to use a different drive)

Within Visual Studio navigate to the directory that you created:



Task 2: Create your Project

In order to create the project, we will need to use the pcf init command using the Command Prompt. The init command has three parameters:

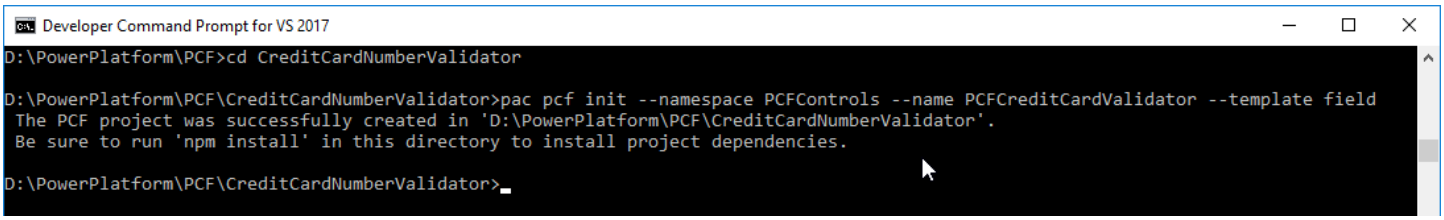
- Namespace – this is the namespace that you will use to provide for the controls that you create for this project. You can use your company name, or a simple name that will represent the type of components that you are creating.
- Name – this is the name of the component that you are creating. This can be any name you want, but notice that there are some character restrictions that need to be adhered to, so use Alphanumeric characters only.

- Template – this is the type of project template we will be creating. The two available options for this are **field** and **dataset**.

For our purpose, we will use the following parameters:

Namespace: PCFControls, Name: PCFCreditCardValidator, Template: field

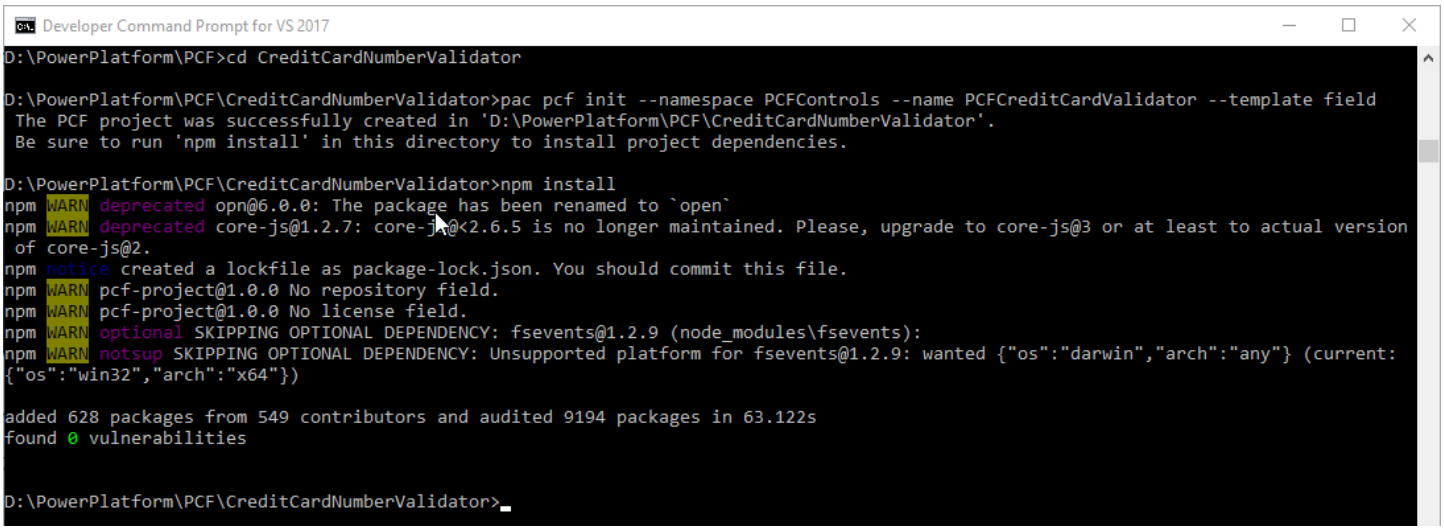
pac pcf init --namespace PCFControls --name PCFCreditCardValidator --template field



Task 3: Install the 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.



Upon completion you will see that a large number of packages has been added to your project directory. You can browser your project directory, and you will notice that the node_modules folder has been created. This contains a large variety of available modules that can be added to your typescript project.

Configure the manifest file

The manifest file defines the properties of the field and includes the data types for the control which is being developed.

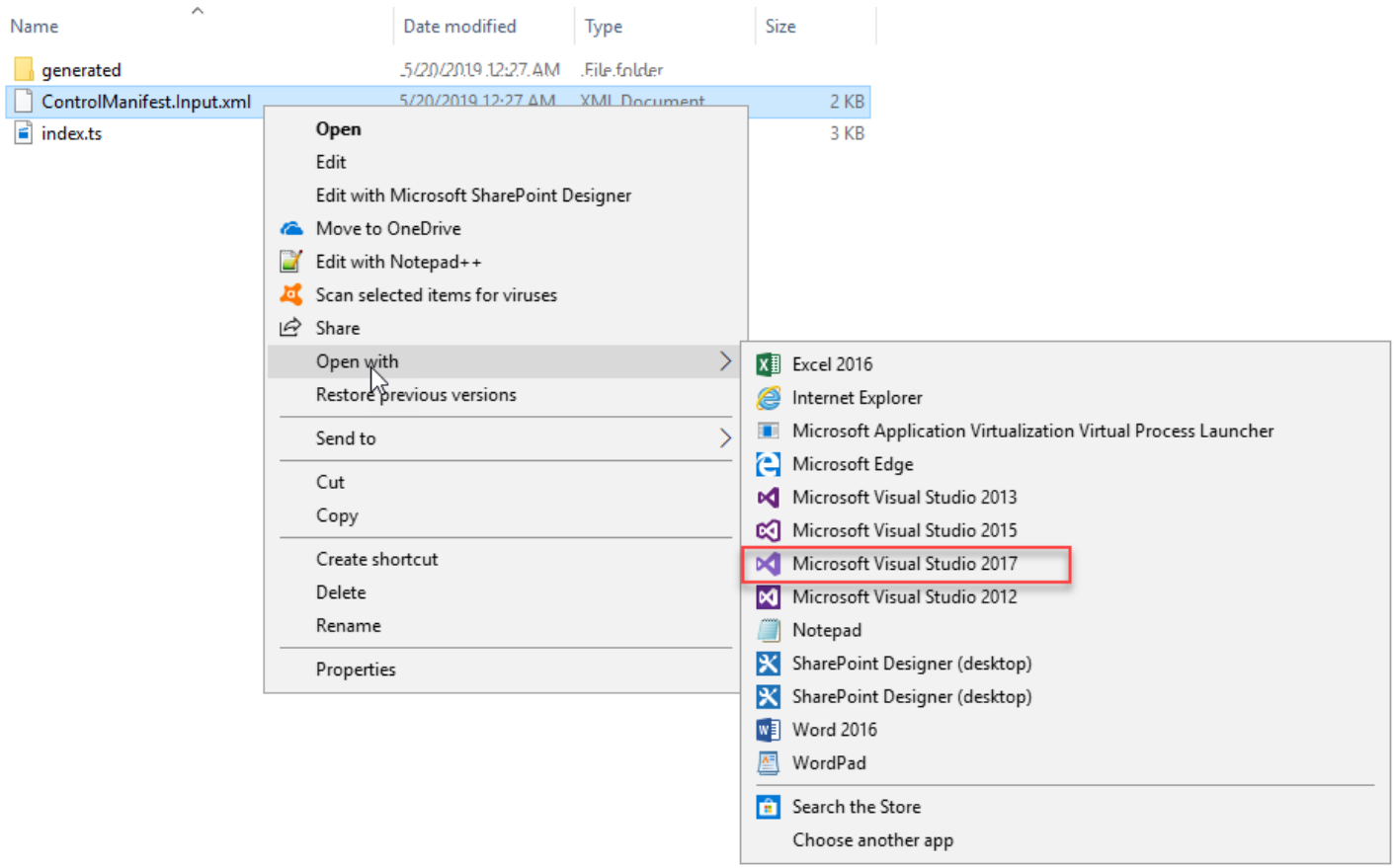
Add more info about manifest file...

Task 1: Open the Manifest file

The manifest file is an Xml file, which can be opened in Visual Studio, Notepad++ or any other Xml or Text editor. It is pretty easy to configure in either Visual Studio or Notepad++.

Navigate to the directory containing the manifest file, which is named ControlManifest.Input.xml. This will be within the directory of your project that is called the name of the component or ***PCFCreditCardValidator*** in our case.

Right-click on the file, select Open with and select Microsoft Visual Studio 2017 in order to open the file with Visual Studio.



When you open the Control Manifest Xml file it will contain a control element, a property element and a resources area. We will review each one separately.

The manifest file is an Xml file, which can be opened in Visual Studio, Notepad++ or any other Xml or Text editor. It is pretty easy to configure in either Visual Studio or Notepad++.

Task 2: Review and Edit 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. 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="PCFControls" constructor="PCFCreditCardValidator" version="0.0.1"
 display-name-key="PCFControls.PCFCreditCardValidator"
 description-key="Credit Card Control for PCF"  control-type="standard">
```

Task 3: Review and Edit the property element(s)

The next part is reviewing the property element. Our solution is using a single property element for storing a credit card number, but can contain multiple property elements based on the type of the control that you are creating. 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.

For the purpose of this lab, we will modify the property as follows:

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

Task 4: Final view of the Manifest file

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/PCFCreditCardValidator.css" order="1" />
   <resx path="strings/PCFCreditCardValidator.1033.resx" version="1.0.0" />
   -->
</resources>
```

The final version of the Control Manifest file that we will be using for this demo will look like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<manifest>
  <control namespace="PCFControls" constructor="PCFCreditCardValidator" version="0.0.1"
           display-name-key="PCFControls.PCFCreditCardValidator" description-key="Credit Carc Control for PCF" control-type="standard">
    <!-- property node identifies a specific, configurable piece of data that the control expects from CDS -->
    <property name="CreditCardNumber" display-name-key="PCFCreditCardValidator_CreditCardNumber"
              description-key="Credit Card Number field" 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>FP</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/PCFCreditCardValidator.css" order="1" />
      <resx path="strings/PCFCreditCardValidator.1033.resx" version="1.0.0" />
      -->
    </resources>
  </control>
</manifest>
```

Building the index.ts file

Same as the automatic generation of the Manifest file, the index.ts file is also generated for us 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.

Similar to a C# application, the typescript file contains header import such as in the empty file:

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

The class declaration is also somewhat similar to C# code for a class the implements a control:

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

A full copy of the startup file is displayed as an image next.

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

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

    /**
     * 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 contains 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 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' in the Mode interface.
     * @param container If a control is marked control-type='starndard', 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, 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
     */
    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 call for cleanup.
     * i.e. cancelling any pending remote calls, removing listeners, etc.
     */
    public destroy(): void
    {
        // Add code to cleanup control if necessary
    }
}
```

There are X steps required when starting to build the control.

First we need to define the variables for the HTML Elements, variables for the control properties and event variables for any event that will be executed based on control interaction. Then we need to define the variables to hold the control context, a delegate to handle the notifyOutputChanged event and an HTML element which will contain the user interface elements that are bound to the Unified Interface.

Task 1: Declare the variables for your index file

Since the control that we are building is a Credit Card Control, there are two HTML elements that make up the control, an Input control and an additional HTML element which will hold the image. We will declare the HTML elements as follows:

```
private _creditCardNumberElement: HTMLInputElement;
private _creditCardTypeElement: HTMLElement;
```

Next we will declare the properties for the input control. Since the input control only has a single property which is the Credit Card number, we will declare a single string variable:

```
private _creditCardNumber: string;
```

Next we will declare variable for the event handlers. We only have a single event handler which is called when the Credit Card number is changed, so we will declare that event handler variable.

```
private _creditCardNumberChanged: EventListenerOrEventListenerObject;
```

Finally, we will declare a variable to hold the control context that contains the information about the control, a delegate to respond to the notifyOutputChanged event which gets triggered when the control receives new outputs, and finally an HTML Div element which will contain the UI elements that are bound to the User Interface.

```
private _context: ComponentFramework.Context<IInputs>;
private _notifyOutputChanged: () => void;
private _container: HTMLDivElement;
```

Task 2: 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.

```
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.

```
this._creditCardNumberChanged = this.creditCardChanged.bind(this);
```

Finally, 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 enter the correct credit card number that corresponds to the rules of the card.

```
// input control
this._creditCardNumberElement=document.createElement("input");
this._creditCardNumberElement.setAttribute("type", "text");
this._creditCardNumberElement.setAttribute("class", "CreditCardBox");
this._creditCardNumberElement.setAttribute("style", "width: 90%");
this._creditCardNumberElement.addEventListener("change", this._creditCardNumberChanged);

// image control
this._creditCardTypeElement = document.createElement("img");
this._creditCardTypeElement.setAttribute("height", "24px");
```

After creating the HTML elements we will add them to the HTMLDivElement container.

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

The final init method is shown below:

```
public init(context: ComponentFramework.Context<IInputs>, notifyOutputChanged: () => void, state: ComponentFramework.Dictionary, container:HTMLDivElement)
{
    // assigning environment variables.
    this._context = context;
    this._notifyOutputChanged = notifyOutputChanged;
    this._container = container;

    // Set the control event binding
    this._creditCardNumberChanged = this.creditCardChanged.bind(this);

    // Set textbox input element
    this._creditCardNumberElement=document.createElement("input");
    this._creditCardNumberElement.setAttribute("type", "text");
    this._creditCardNumberElement.setAttribute("class", "gp cs mk x ml fa ez hv kd ly jn jo bu bv kk kl mm mn mo mp mq mr ms mt aq mu mv mw mx my mz na nb cu nc nd");
    this._creditCardNumberElement.setAttribute("style", "width: 90%");
    this._creditCardNumberElement.addEventListener("change", this._creditCardNumberChanged);

    // Set image element
    this._creditCardTypeElement = document.createElement("img");
    this._creditCardTypeElement.setAttribute("height", "24px");

    // Add input and image elements to the Div Container element
    this._container.appendChild(this._creditCardNumberElement);
    this._container.appendChild(this._creditCardTypeElement);
}
```

Task 3: 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 crmCCNAttr = this._context.parameters.CreditCardNumber.attributes.LogicalName;
Xrm.Page.getAttribute(crmCCNAttr).setValue(this._context.parameters.CreditCardNumber.formatted);
```

The final function is shown below:

```
public updateView(context: ComponentFramework.Context<IInputs>): void
{
    debugger

    // CRM attributes bound to control properties
    // @ts-ignore
    var crmCreditCardNumberAttribute = this._context.parameters.CreditCardNumber.attributes.LogicalName;

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

Note the @ts-ignore above the lines that call Xrm.Page class. This is used in order to suppress any errors in the following offending lines of code.

Task 4: 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:

```
return {
            CreditCardNumber: this._creditCardNumber
       };
```

The final getOutput method will look like this:

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

Task 5: Creating the event

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 start the event by getting the value that was entered in the field, and verifying that it is not empty:

```
var creditCardNumber = this._creditCardNumberElement.value;
if (creditCardNumber != null && creditCardNumber.length > 0) …
```

When the provide the conditions to check the type of credit card, and based on the conditions assign the src attribute to the creditCardType image element. Currently the images are stored in Azure Blob container. This contain exists within the if statement above:

```
if (creditCardNumber.length == 15 && (creditCardNumber.substring(0, 2) === "34" ||
creditCardNumber.substring(0, 2) === "37"))
  this._creditCardTypeElement.setAttribute("src", "https://bgx.blob.core.windows.net/ imgs/amex24.png");
else if (creditCardNumber.substring(0, 1) === "4" && (creditCardNumber.length == 13 ||
creditCardNumber.length == 16))
  this._creditCardTypeElement.setAttribute("src", "https://bgx.blob.core.windows.net/ imgs/visa24.png");
else if (creditCardNumber.substring(0, 4) === "6011" && creditCardNumber.length == 16)
  this._creditCardTypeElement.setAttribute("src", "https://bgx.blob.core.windows.net /imgs/disc24.png");
else if (parseInt(creditCardNumber.substring(0, 2)) > 50 && parseInt(creditCardNumber.substring(0, 2))
< 56 && creditCardNumber.length == 16)
  this._creditCardTypeElement.setAttribute("src", "https://bgx.blob.core.windows.net/imgs/ico/mc24.png");
else
  this._creditCardTypeElement.setAttribute("src", "https://bgx.blob.core.windows.net/
imgs/redwar.png");

this._creditCardNumber = creditCardNumber;
```

We then add a condition for the case that the value is not valid to remove the src attribute:

```
this._creditCardNumberElement.removeAttribute("src");
this._creditCardNumber = "";
```

After adding both all conditions, we call the notifyOutputChanged callback method that the control has new outputs that are ready:

```
this._notifyOutputChanged();
```

The final code for the event is displayed below:

```
public creditCardChanged(evt: Event):void
{
    debugger

    var creditCardNumber = this._creditCardNumberElement.value;
    if (creditCardNumber != null && creditCardNumber.length > 0)
    {
        if (creditCardNumber.length == 15 && (creditCardNumber.substring(0, 2) === "34" || creditCardNumber.substring(0, 2) === "37"))
            this._creditCardTypeElement.setAttribute("src", "https://[redacted].blob.core.windows.net/shared/imgs/ico/amex24.png");
        else if (creditCardNumber.substring(0, 1) === "4" && (creditCardNumber.length == 13 || creditCardNumber.length == 16))
            this._creditCardTypeElement.setAttribute("src", "https://[redacted].blob.core.windows.net/shared/imgs/ico/visa24.png");
        else if (creditCardNumber.substring(0, 4) === "6011" && creditCardNumber.length == 16)
            this._creditCardTypeElement.setAttribute("src", "https://[redacted].blob.core.windows.net/shared/imgs/ico/disc24.png");
        else if (parseInt(creditCardNumber.substring(0, 2)) > 50 && parseInt(creditCardNumber.substring(0, 2)) < 56 && creditCardNumber.length == 16)
            this._creditCardTypeElement.setAttribute("src", "https://[redacted].blob.core.windows.net/shared/imgs/ico/mc24.png");
        else
            this._creditCardTypeElement.setAttribute("src", "https://[redacted].blob.core.windows.net/shared/imgs/ico/redwarning.png");
        this._creditCardNumber = creditCardNumber;
    }
    else
    {
        this._creditCardNumberElement.removeAttribute("src");
        this._creditCardNumber = "";
    }
    this._notifyOutputChanged();
}
```

Task 6: The constructor and destroyer methods

The constructor and destroyer methods are used in many programming language to initialize certain logic or variables in a class and to clean up control, remote calls or listeners that might still be consuming resources.

Below is the code for the constructor and destroy methods. In our case for the Credit card PCF control, there is no constructor logic, and the destroy method is used only to remove the event listener change event from the bound control.

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

}

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

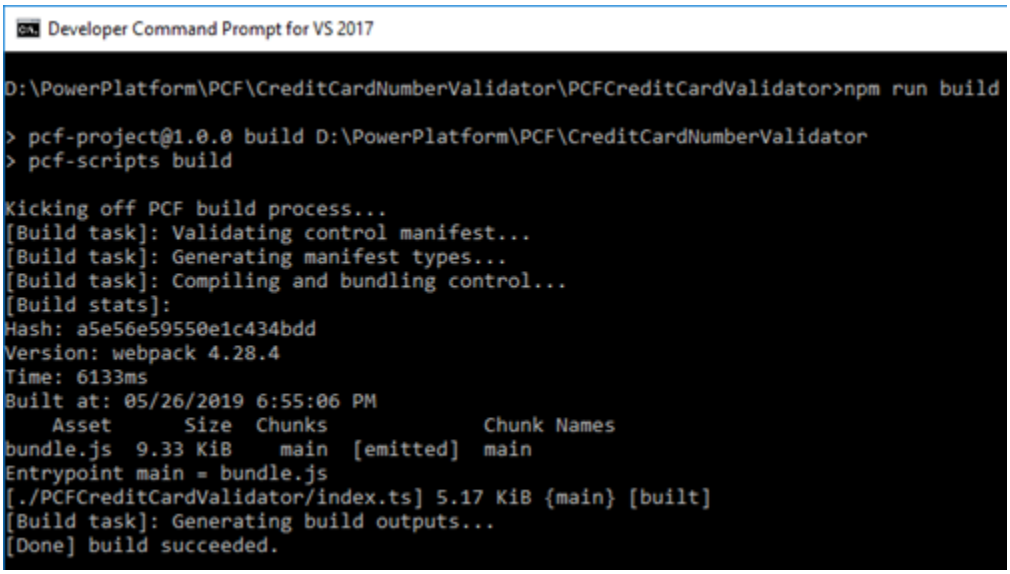
Build and Test the Credit Card PCF Control

A PCF control is deployed into Microsoft Dynamics 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.

In order to build the application, we will need to use the Developer Command Prompt for VS 2017 once again. Once we open the command prompt, we need to navigate to the directory containing our source code files, or the directory that contains the Manifest and Index typescript files (as well as the css or resx files).

Task 1: Build the control

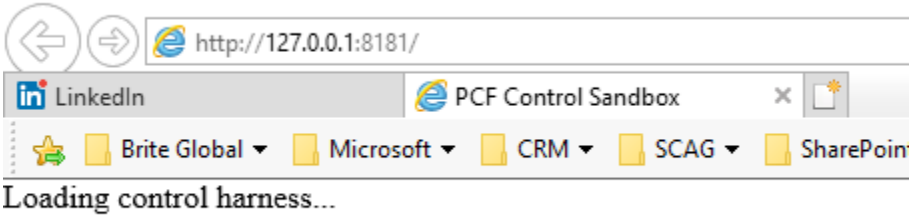
To build the component, within the directory type ***npm run build***. This will show up a progress screen and if there are no errors, will display at the end a build succeeded message in the command prompt as shown below:



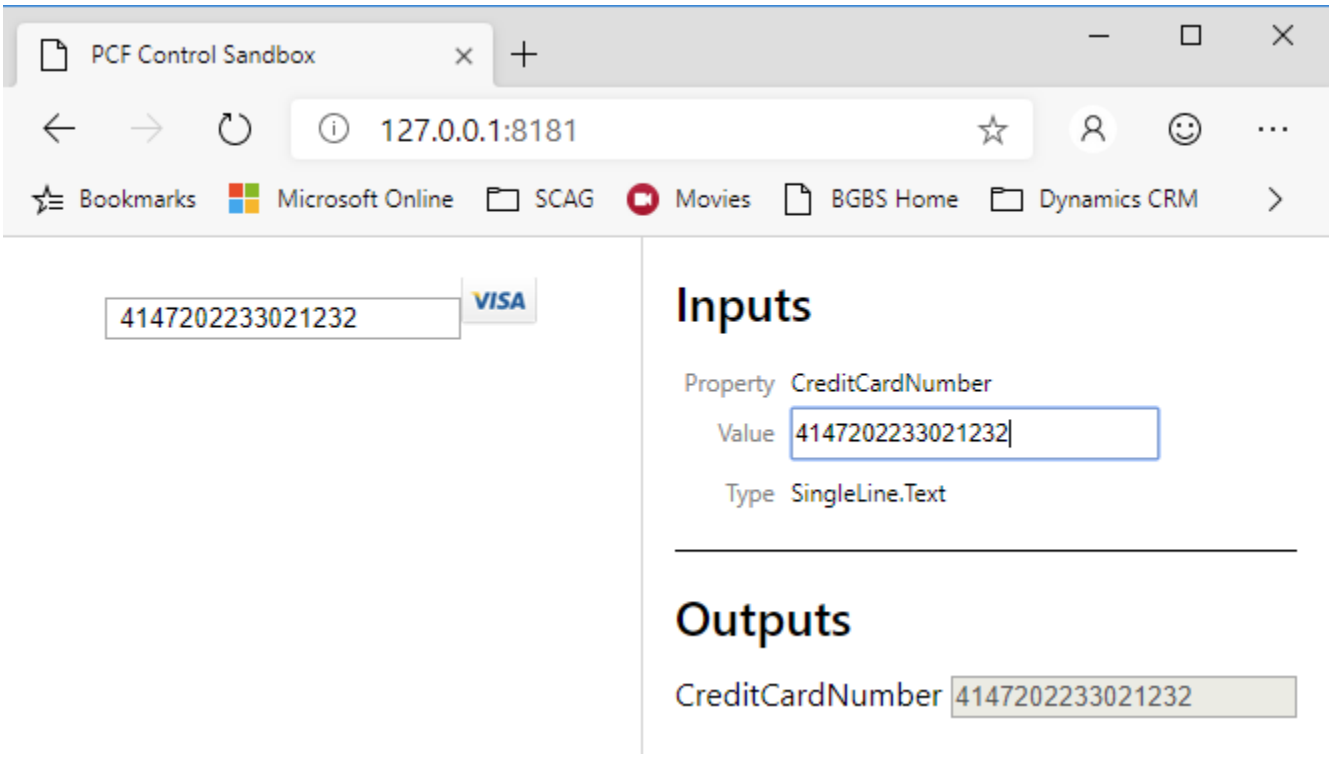
If any errors were found during build recheck your manifest and typescript files and make sure that there are no issues with the Xml or typescript code. Now that the validation has been completed we can go ahead and test the control.

Task 2: Test the control

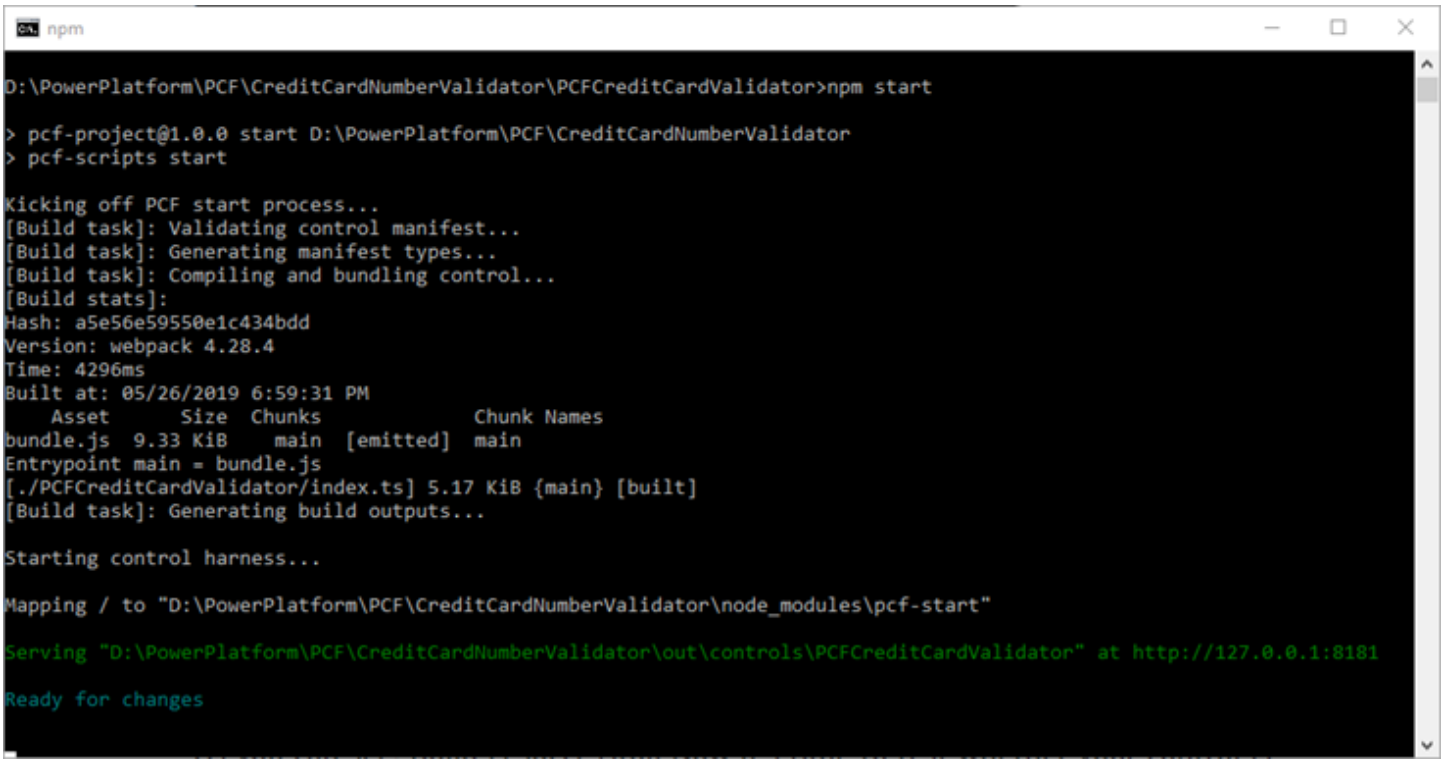
To test the component, within the directory type ***npm start***. This will allow you to view your control within a browser window so that you can verify the functionality. By default the control will open within Internet Explorer. Internet Explorer does not always render the controls properly, as shown below, so I would recommend using a different browser.



Copy the url, and use a different browser (such as Chrome or Edge)



Below your will see the npm start command, where at the end it shows Ready for Changes.

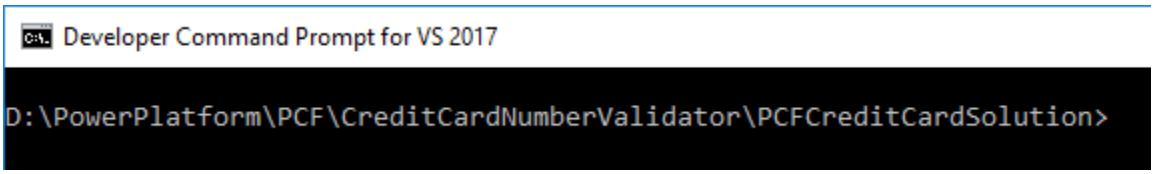


Once you have done testing your control, press on the keyboard Ctrl+C. This will display a message to Terminate the job (Y/N). Type in Y and press enter. This will stop the control testing process.

Creating 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.



Task 1: Creating the solution files

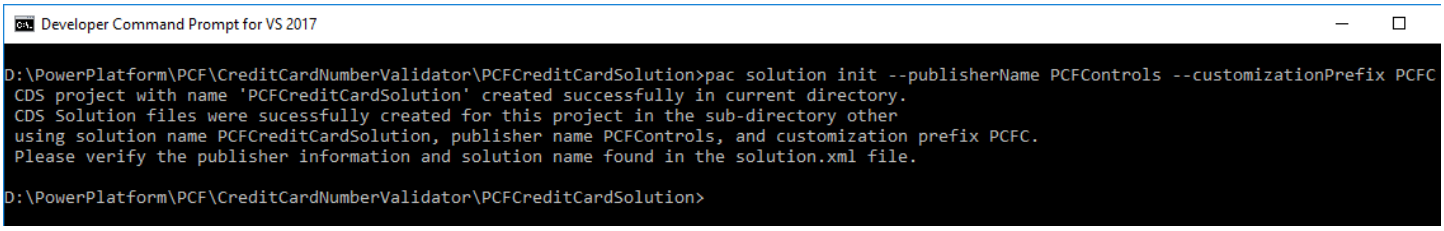
The ***pac solution init*** command will allow us to create the solution within our current folder. The command requires us to provide a publisher name and a customization prefix, same as with our CRM solutions. The full command is shown below:

pac solution init –publisherName [THE_NAME_OF_THE_PUBLISHER] --customizationPrefix [PREFIX_TO_USE_WITH_PUBLISHER]

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

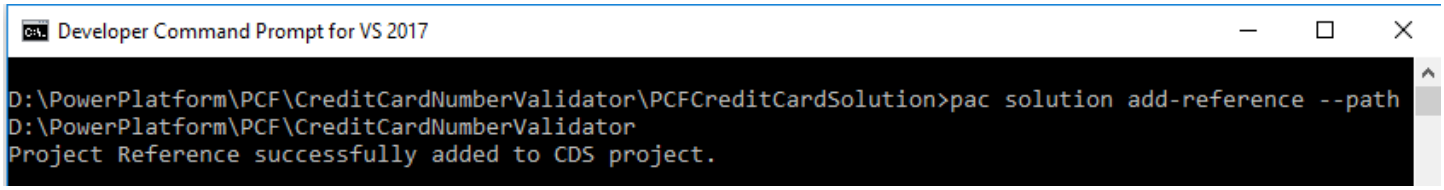
pac solution init –publisherName PCFControls --customizationPrefix PCFC

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



Task 2: Adding the solution reference

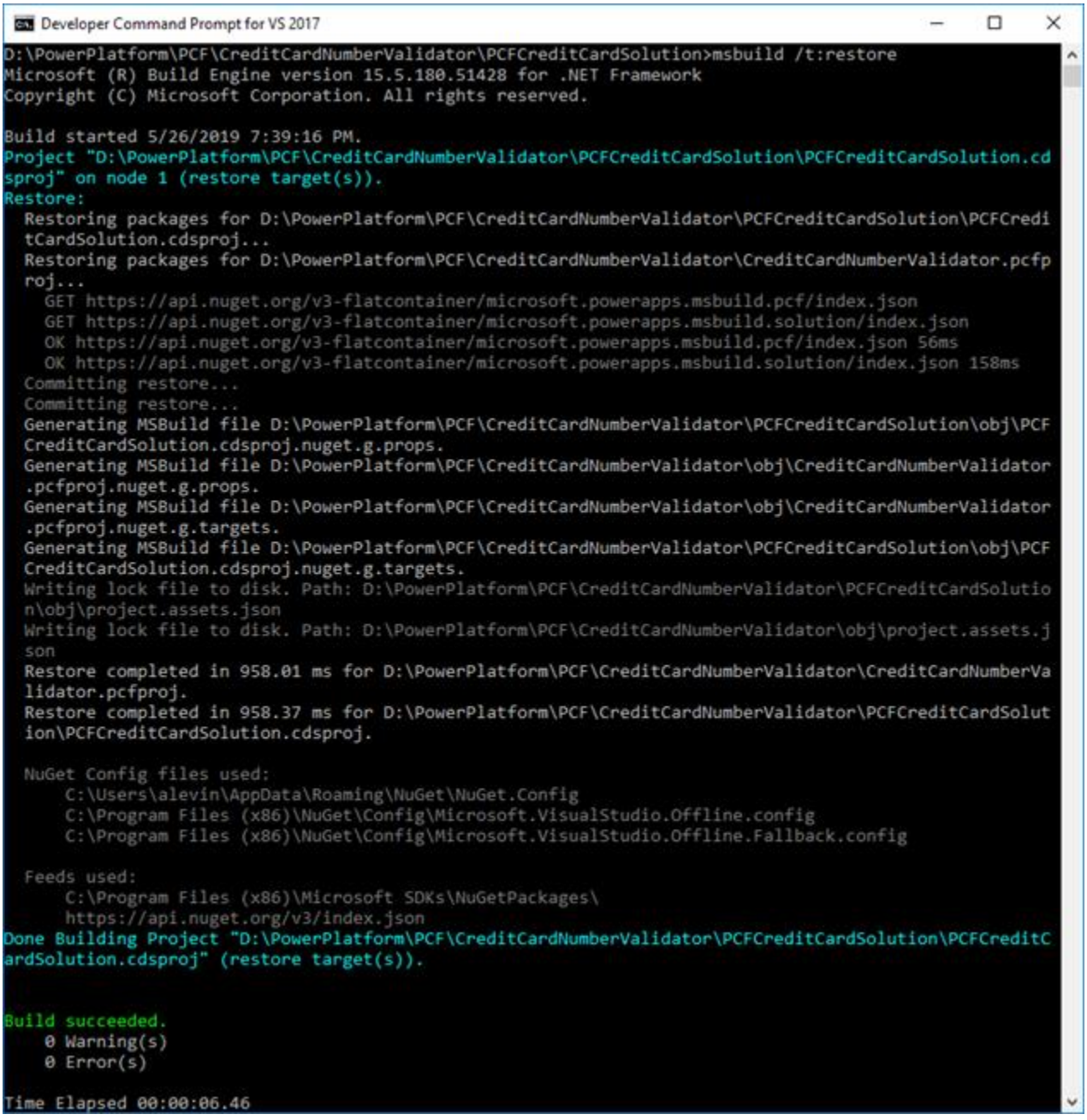
After we have verified that the creation of the solution files was successful, we will need to add the custom control reference. We do this by calling the ***pac solution add-reference***, and providing the directory where the PowerApps Component Framework (PCF) project is located. In our case the directory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



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

Task 3: Running MSBuild

The final two steps to generate the solution zip files are to run the msbuild command. The first time we will run it with the /t:restore parameter, and then run it by itself.

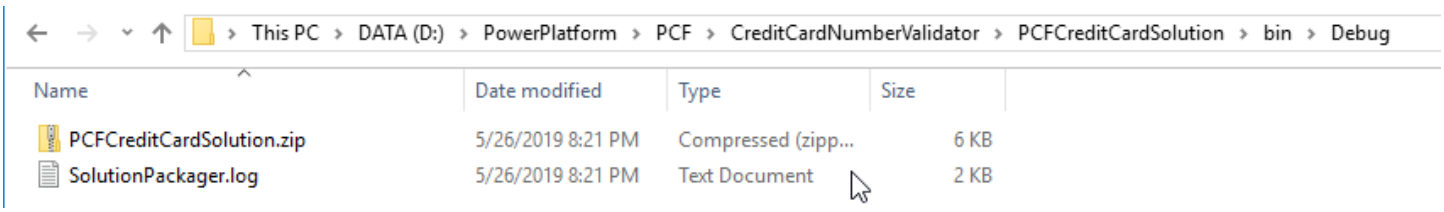


If the restore succeeded, you will see a Build succeeded message at the end of the process. Finally, we will now run the msbuild command to complete the build and prepare the solution.

You can see the execution and end of the build command results below:

```
Developer Command Prompt for VS 2017

D:\PowerPlatform\PCF\CreditCardNumberValidator\PCFCreditCardSolution>msbuild
Microsoft (R) Build Engine version 15.5.180.51428 for .NET Framework
Copyright (C) Microsoft Corporation. All rights reserved.

Build started 5/26/2019 8:21:06 PM.
Project "D:\PowerPlatform\PCF\CreditCardNumberValidator\PCFCreditCardSolution\PCFCreditCardSolution.cd
sproj" on node 1 (default targets).
Project "D:\PowerPlatform\PCF\CreditCardNumberValidator\PCFCreditCardSolution\PCFCreditCardSolution.cd
sproj" (1) is building "D:\PowerPlatform\PCF\CreditCardNumberValidator\CreditCardNumberValidator.pcfpr
oj" (2) on node 1 (default targets).
Build:
  npm run build -- --buildmode development --outDir D:\PowerPlatform\PCF\CreditCardNumberValidator\out
  \controls\\
  Packing D:\PowerPlatform\PCF\CreditCardNumberValidator\PCFCreditCardSolution\obj\Debug\ to bin\Debug
  \\PCFCreditCardSolution.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: EntityDataProviders

  Unmanaged Pack complete.

  Solution: bin\Debug\\PCFCreditCardSolution.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\CreditCardNumberValidator\PCFCreditCardSolution\PCFCreditC
ardSolution.cdsproj" (default targets).


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

Time Elapsed 00:00:15.58
```

If we look at the bin/debug directory we will now see the zip file over there:



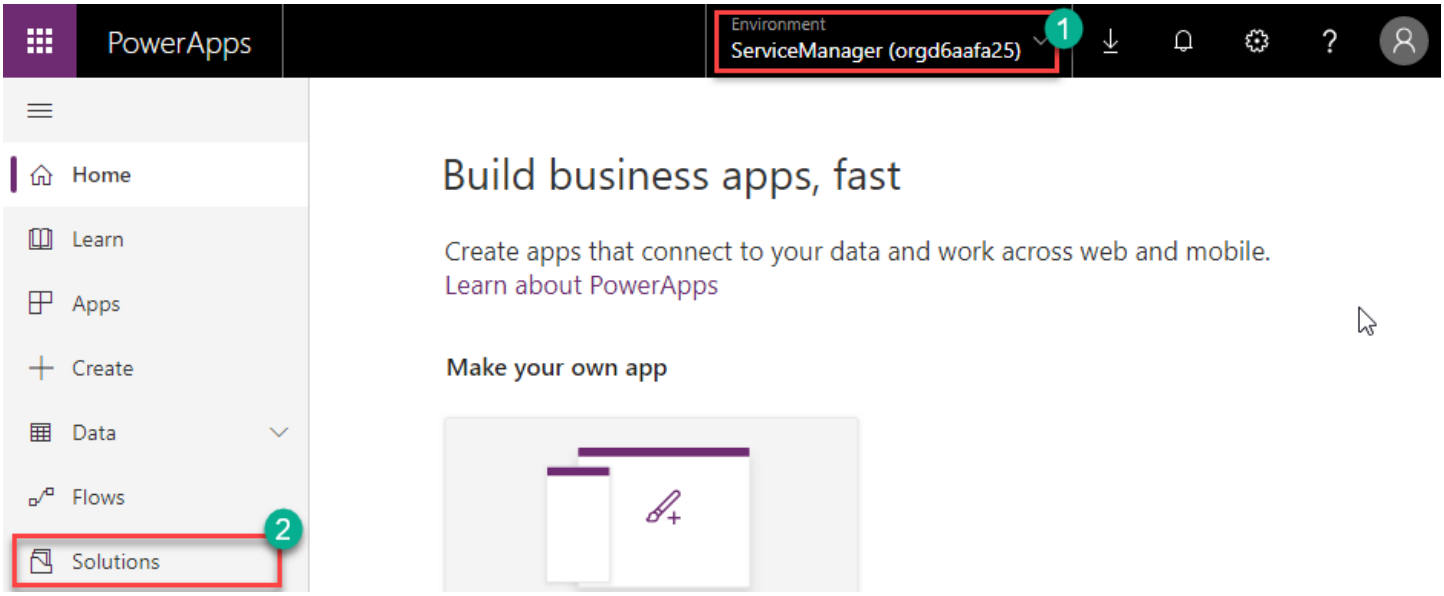
We are not done with the creation of the package to be deployed into Dynamics. In the next section we will deploy the solution into Microsoft Dynamics 365 environment and see it in progress.

Deploying the Solution and using the control on the form

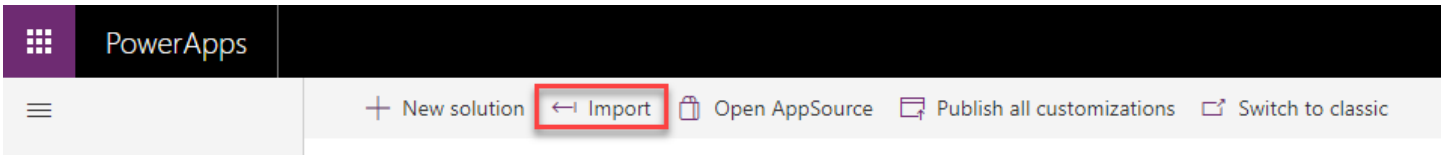
Now that the solution has been created we are ready to deploy it in our environment. We will first import the new solution into our environment, create the field inside our Model-driven application, and finally test it out.

Task 1: Importing the solution

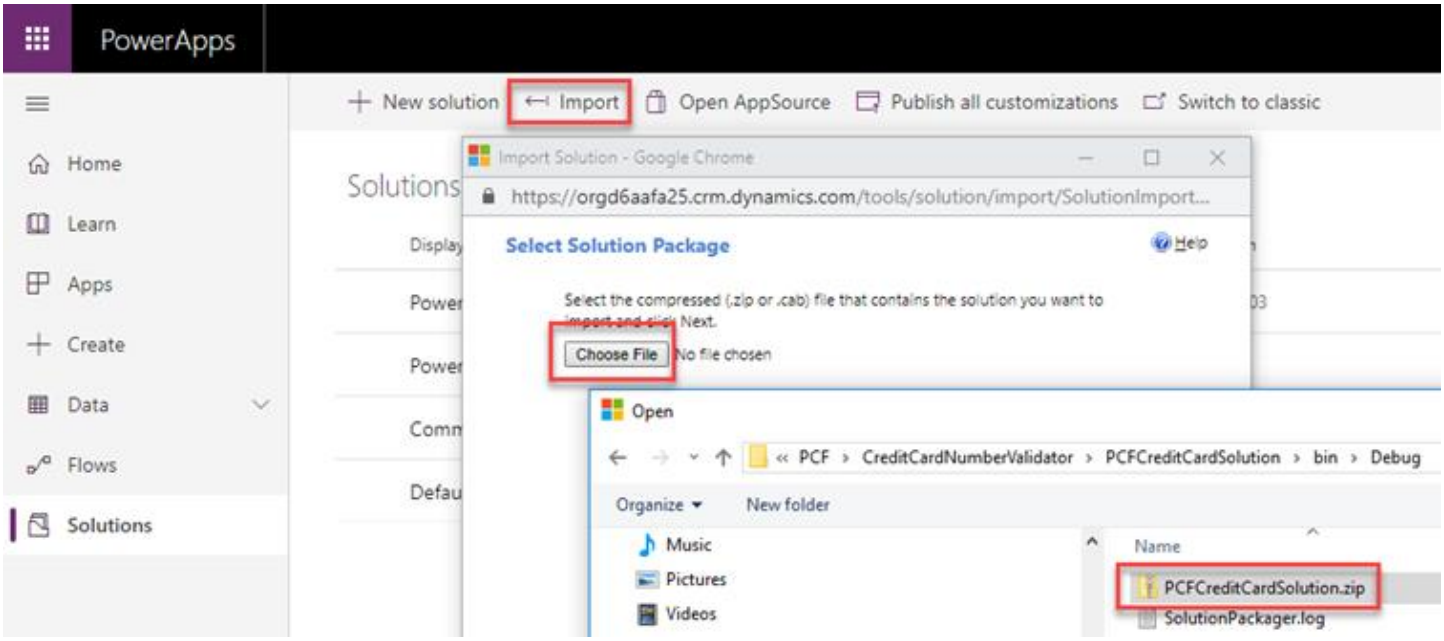
At the time of writing this guide, importing solutions is still using part of the old import functionality, but we will connect to our PowerApps application to implement this. Start by navigating to web.powerapps.com, and login with your admin credentials. Make sure that you are connected to the right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



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.

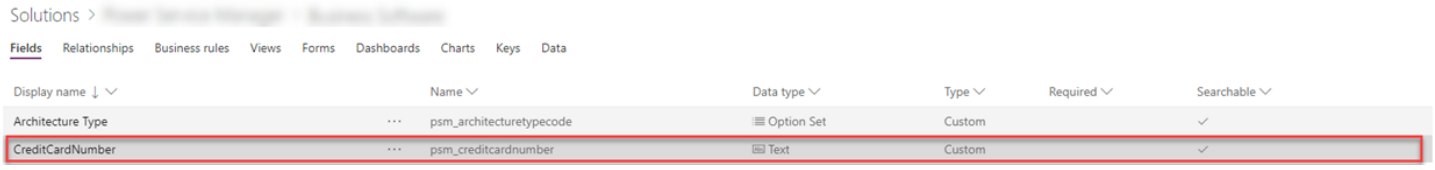


Select the file and click Open in the Open window.

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.

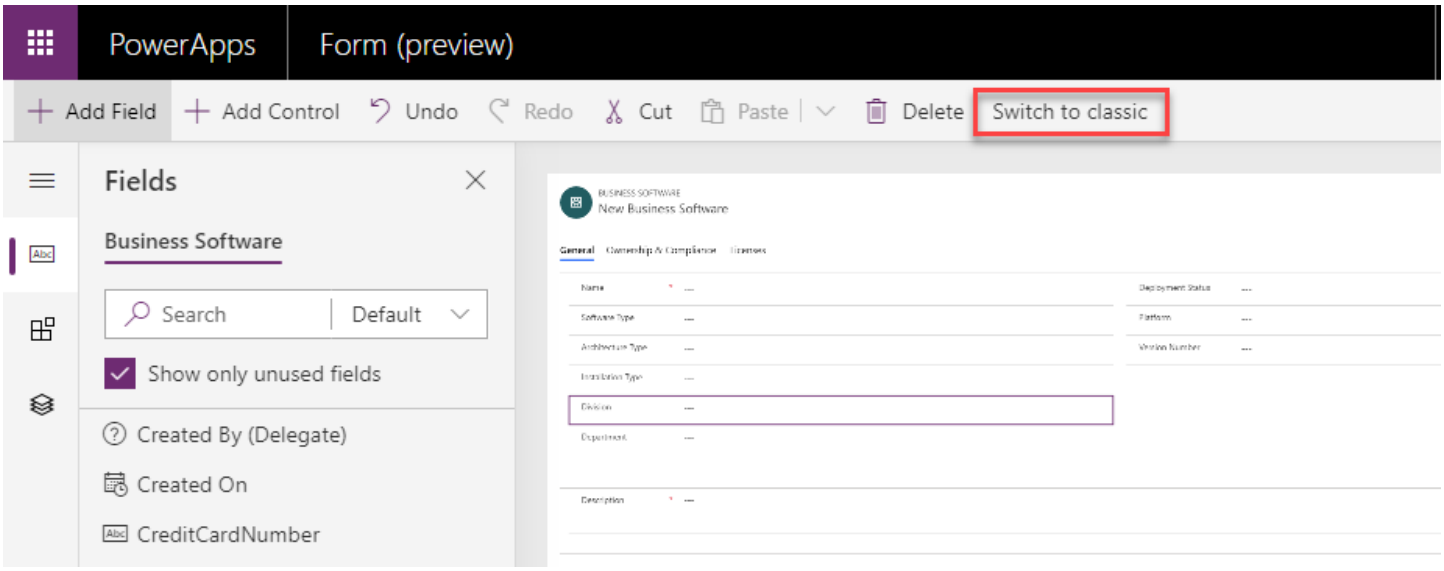
Task 2: Add the required fields

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.



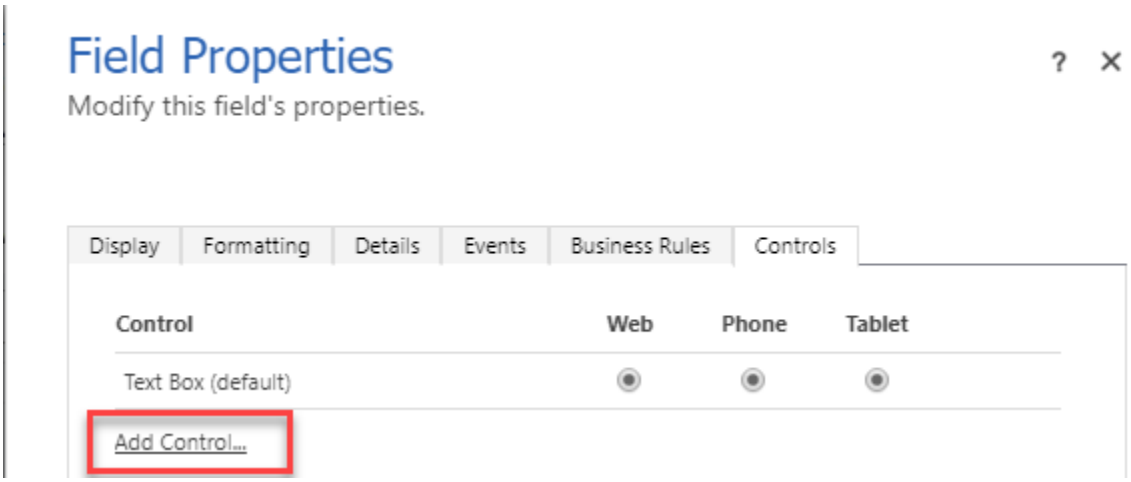
Task 3: Add the control to the form

After adding the field and publishing the entity, Click on forms, and select the Main form type. This will open the form in a new tab. Since the new Forms designer does not yet include all the features that are available in the Classic form, we will have to switch to the Classic form designer to make the changes. In general, it is a good idea to get used to making all changes in the new User Interface of PowerApps.



After clicking on ***Switch to classic***, we will see the old form designer. Add the Credit Card Number field to the form, and double click on it or select it and then click on Change Properties on the ribbon. The properties window will open.

Click on the Controls tab, and then click on Add Control link.



In the Add Control popup window, select the PCFControls.PCFCreditCardValidator control and then click on Add. In the Field properties screen, change the Web display from Text Box (default) to PCFControls.PCFCreditCardValidator. Click OK to complete this operation.

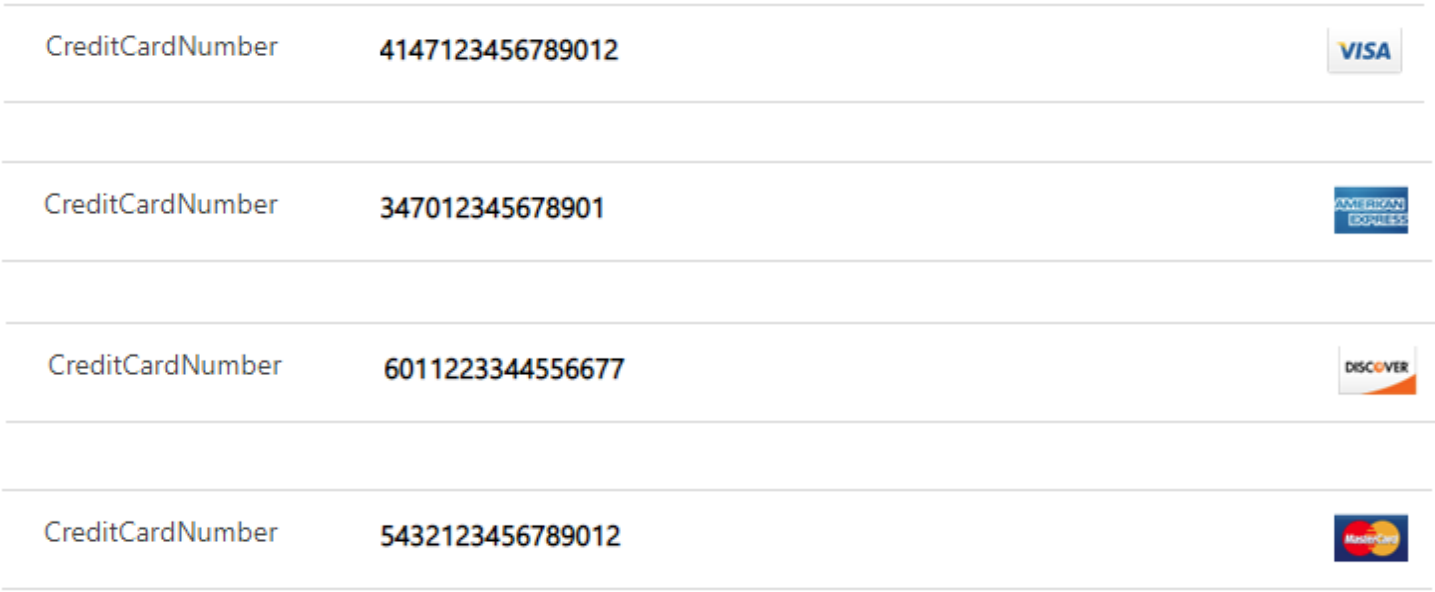
In the Classic form designer Click on the Save button, and then click Publish.

Task 4: Test the control to the form

Now that we had added the control to the form, we can test this out and make sure that it works. We will run the following tests with these test credit card numbers and show the results below:

Credit Card Number	Credit Card Type
4147123456789012	VISA
347012345678901	AMEX
6011223344556677	DISCOVER
5432123456789012	MASTERCARD

The results are shown in the images below:



References

This module uses terminology from PowerApps, CDS and the PowerApps component framework

PowerApps

- Website | Blog | Documentation | Community | Suggest Ideas | Webinars -
- Guided Learning | YouTube playlist

Microsoft Flow

- Website | Blog | Documentation | Community | Suggest Ideas | Webinars
- Guided Learning | YouTube Playlist

Common Data Service (CDS)

- Common Data Service documentation portal

PowerApps Component Framework

- https://docs.microsoft.com/en-us/powerapps/developer/component-framework/overview
- Debajit's Dynamic CRM Blog