Create Custom Controls using PowerApp Component Framework

## **Prerequisite**

* Install **npm** (node js)
* If you don’t have Visual Studio 2017 then install **Visual Studio 2017 or higher version** or **Visual studio code**
* Download and Install **PowerApp CLI** on your machine (Currently PowerApps CLI is supported only on Windows 10)

## **Create & configure PowerApp Component Framework components**

1. Create or identify your working folder. In my case it is **“D:\PCFControls\PowerAppCustomControl”**
2. Open Visual Studio command prompt
3. Navigate to you recently created folder (step 1) using **cd** command. In my case it is

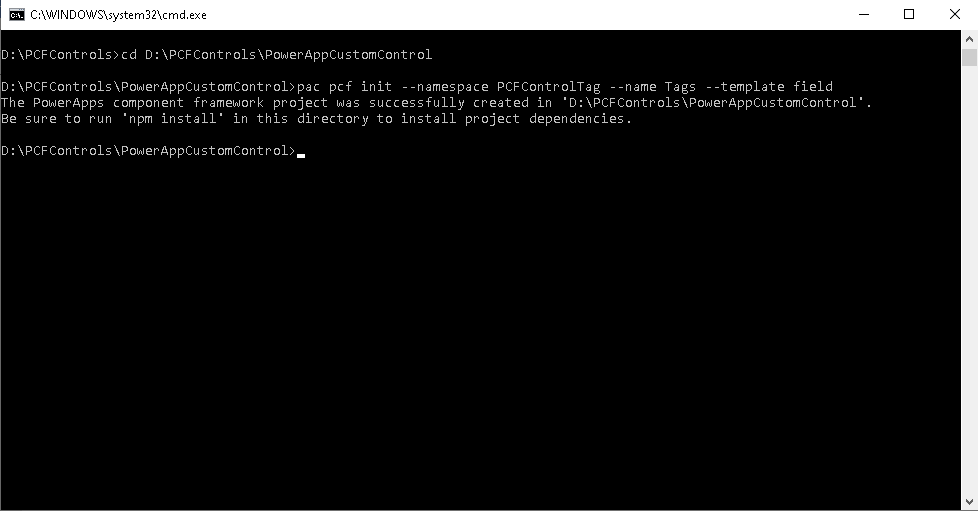
**cd D:\PCFControls\PowerAppCustomControl**

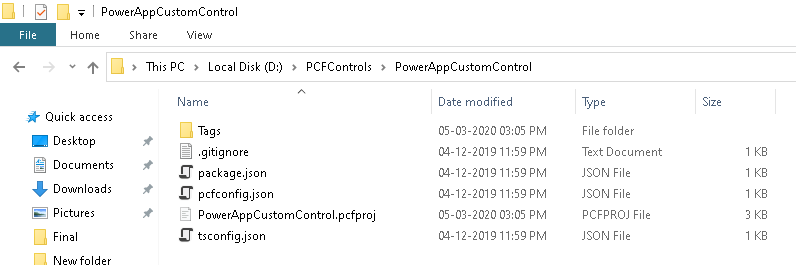
1. Run the **pac** command to create new component project. Command is as follows:

pac pcf init --namespace [Your Namespace] --name [Your Component Name] --template [Component Type]

In my case, following is the **pac** command:

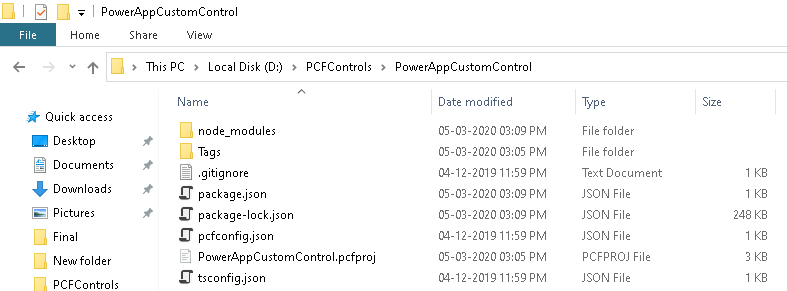
pac pcf init --namespace PCFControlTag --name Tags --template field





1. Now we want to retrieve all project dependencies. To do that we want to run the following command:

**npm install**

****

1. Now the project is ready; navigate in your project folder. You can use any development tool to implement your custom control. I have used Visual Studio Code.

*My final* ***ControlManifest.Input*** *file looks like below:*

<?xml version="1.0" encoding="utf-8" ?>

<manifest>

  <control namespace="PCFCustomControl" constructor="Tags" version="0.0.1" display-name-key="Tags" description-key="Tags description" control-type="standard">

  <!-- property node identifies a specific, configurable piece of data that the control expects from CDS -->

    <type-group name="forTags">

      <type>SingleLine.Text</type>

      <type>SingleLine.TextArea</type>

    </type-group>

    <property name="Tags" display-name-key="Tags" description-key="Tags" of-type-group="forTags" 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"/>

      <css path="css/index.css" order="1" />

      <!-- UNCOMMENT TO ADD MORE RESOURCES

      <css path="css/Tags.css" order="1" />

      <resx path="strings/Tags.1033.resx" version="1.0.0" />

      -->

    </resources>

  </control>

</manifest>

*My entire* ***index.ts*** *code file*

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

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

    /\*\*

     \* Variables for HTML element

     \*/

    private tagsElement: HTMLElement;

    private spaceElement: HTMLElement;

    private refreshButton: HTMLElement;

    private divElement: HTMLElement;

    /\*\*

     \* Variables for Properties

     \*/

    private tagsString: string;

    /\*\*

     \* Variables for Event Listener

     \*/

    private refreshClicked: EventListenerOrEventListenerObject;

    /\*\*

     \* Local Variables

     \*/

    private localContext: ComponentFramework.Context<IInputs>;

    private localNotifyOutputChanged: () => void;

    private localContainer: HTMLDivElement;

    /\*\*

     \* 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) {

        // Init local variables

        this.localContext = context;

        this.localNotifyOutputChanged = notifyOutputChanged;

        this.localContainer = container;

        // Register EventHandler

        this.refreshClicked = this.refreshClick.bind(this);

        // Refresh button

        this.refreshButton = document.createElement("button");

        this.refreshButton.setAttribute("type", "button");

        this.refreshButton.setAttribute("value", "Refresh");

        this.refreshButton.setAttribute("class", "btn btn-default btn-sm glyphicon glyphicon-refresh");

        this.refreshButton.addEventListener("click", this.refreshClick);

        // Add elements to container

        this.localContainer.appendChild(this.refreshButton);

        // @ts-ignore

        var crmTagStringsAttributeValue = crmTagStringsAttributeValue != null ? context.parameters.Tags.raw : "red,green,blue";

        var data = crmTagStringsAttributeValue.split(",");

        for (var i in data) {

            // Create controls

            // Tag element

            this.tagsElement = document.createElement("span");

            this.tagsElement.setAttribute("class", "badge badge-pill badge-primary");

            var ele = this.localContainer.appendChild(this.tagsElement);

            ele.innerHTML = data[i];

            // Space element

            this.spaceElement = document.createElement("span");

            var space = this.localContainer.appendChild(this.spaceElement);

            space.innerHTML = "  ";

        }

    }

    /\*\*

\* 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

        // @ts-ignore

        var crmTagStringsAttributeValue =  // @ts-ignore

            crmTagStringsAttributeValue = context.parameters.Tags.raw != null ? context.parameters.Tags.raw : "red,green,blue";

        var data = crmTagStringsAttributeValue.split(",");

        // Delete all elements first

        var tagCollection = this.localContainer.getElementsByTagName("span");

        var loopLength = tagCollection.length;

        for (var ti = 0; ti < loopLength; ti++) {

            this.localContainer.removeChild(tagCollection[0]);

        }

        // Add new tags

        for (var i in data) {

            // Create controls

            // Tag element

            this.tagsElement = document.createElement("span");

            this.tagsElement.setAttribute("class", "badge badge-pill badge-primary");

            var ele = this.localContainer.appendChild(this.tagsElement);

            ele.innerHTML = data[i];

            // Space element

            this.spaceElement = document.createElement("span");

            var space = this.localContainer.appendChild(this.spaceElement);

            space.innerHTML = "  ";

        }

    }

    /\*\*

     \* 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 {

            Tags: this.tagsString

        };

    }

    /\*\*

     \* 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 {

        // remove the event handlers.

        this.refreshButton.removeEventListener("click", this.refreshClick);

    }

    /\*\*

     \* Custom Event Handlers

     \*/

    public refreshClick(evnt: Event): void {

        this.localNotifyOutputChanged();

    }

}

*My* ***Index.css*** *file*

.badge-primary {

    color: #fff;

    background-color: #007bff;

}

.badge-pill {

    padding-right: 0.6em;

    padding-left: 0.6em;

    border-radius: 10rem;

}

.badge {

    display: inline-block;

    padding: 0.25em 0.4em;

    font-size: 75%;

    font-weight: bold;

    line-height: 1;

    color: #fff;

    text-align: center;

    white-space: nowrap;

    vertical-align: baseline;

    border-radius: 0.25rem;

}

button {

    text-rendering: auto;

    color: initial;

    letter-spacing: normal;

    word-spacing: normal;

    text-transform: none;

    text-indent: 0px;

    text-shadow: none;

    display: inline-block;

    text-align: start;

    margin: 0em;

    font: 400 13.3333px Arial;

}

.btn-sm {

    padding: 5px 10px;

    font-size: 12px;

    line-height: 1.5;

    border-radius: 3px;

}

.btn-default {

    color: #333;

    background-color: #fff;

    border-color: #ccc;

}

.btn {

    display: inline-block;

    padding: 6px 12px;

    margin-bottom: 0;

    font-size: 14px;

    font-weight: 400;

    line-height: 1.42857143;

    text-align: center;

    white-space: nowrap;

    vertical-align: middle;

    -ms-touch-action: manipulation;

    touch-action: manipulation;

    cursor: pointer;

    -webkit-user-select: none;

    -moz-user-select: none;

    -ms-user-select: none;

    user-select: none;

    background-image: none;

    border: 1px solid transparent;

    border-radius: 4px;

}

.glyphicon {

    position: relative;

    top: 1px;

    display: inline-block;

    font-family: 'Glyphicons Halflings';

    font-style: normal;

    font-weight: 400;

    line-height: 1;

    -webkit-font-smoothing: antialiased;

    -moz-osx-font-smoothing: grayscale;

}

.glyphicon-refresh:before {

    content: "\e031";

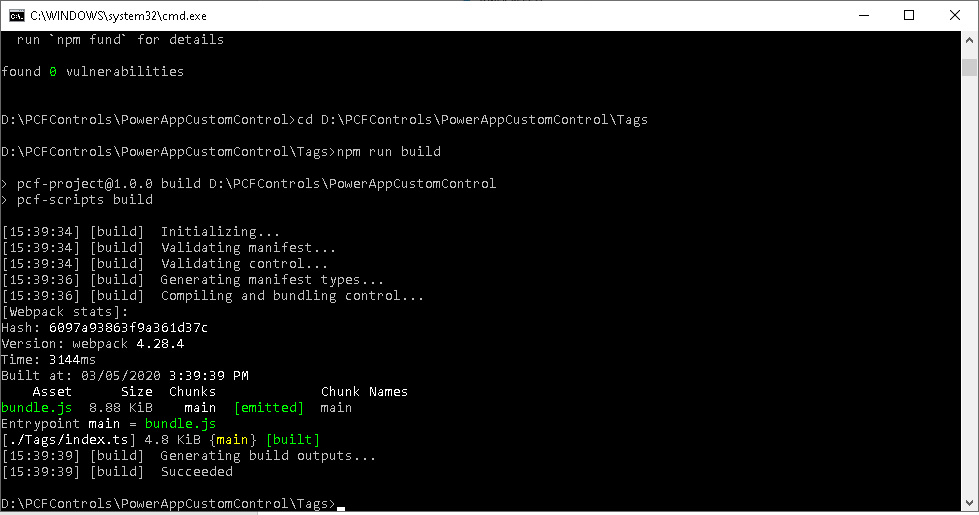
}

1. Build the project

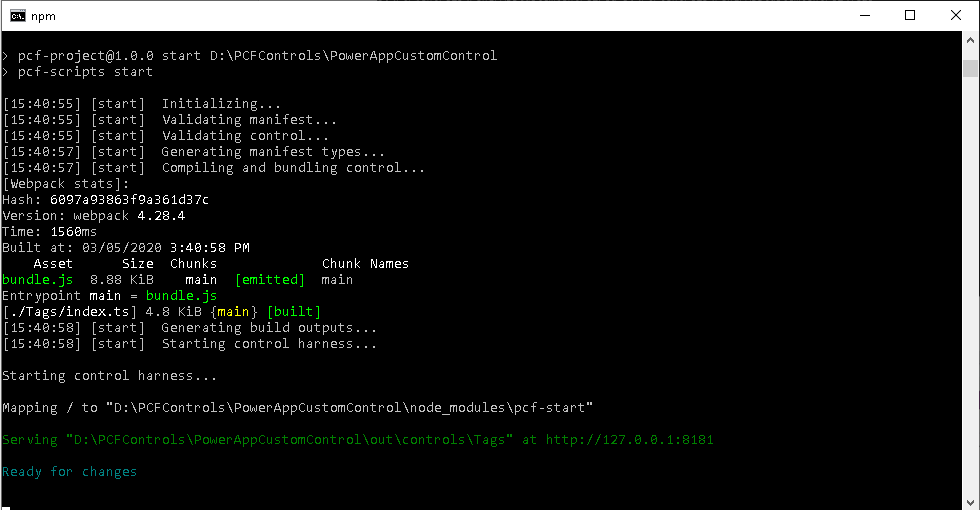
Again, open Visual Studio command prompt and navigate to the folder where the **index.ts** file resides using **cd** command.

**cd D:\PCFControls\PowerAppCustomControl\Tags**

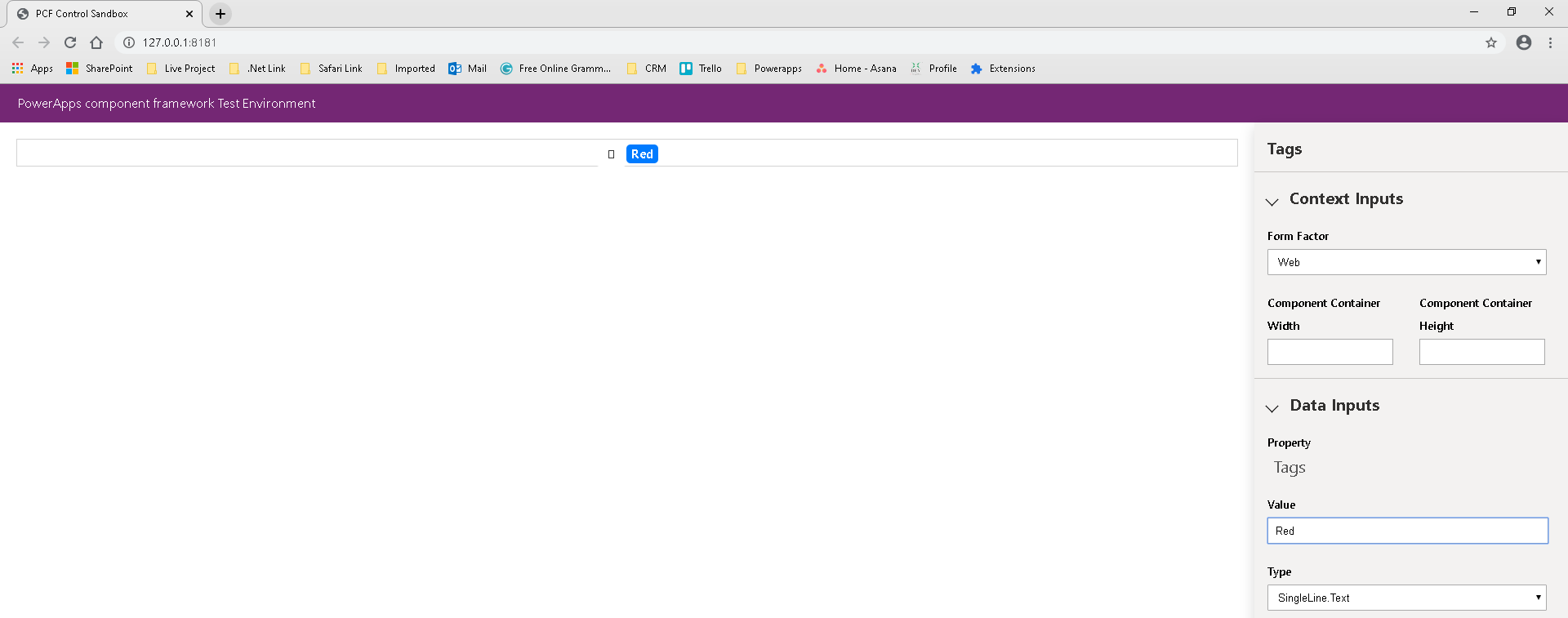
To build the package, run **npm run build** command as shown below.

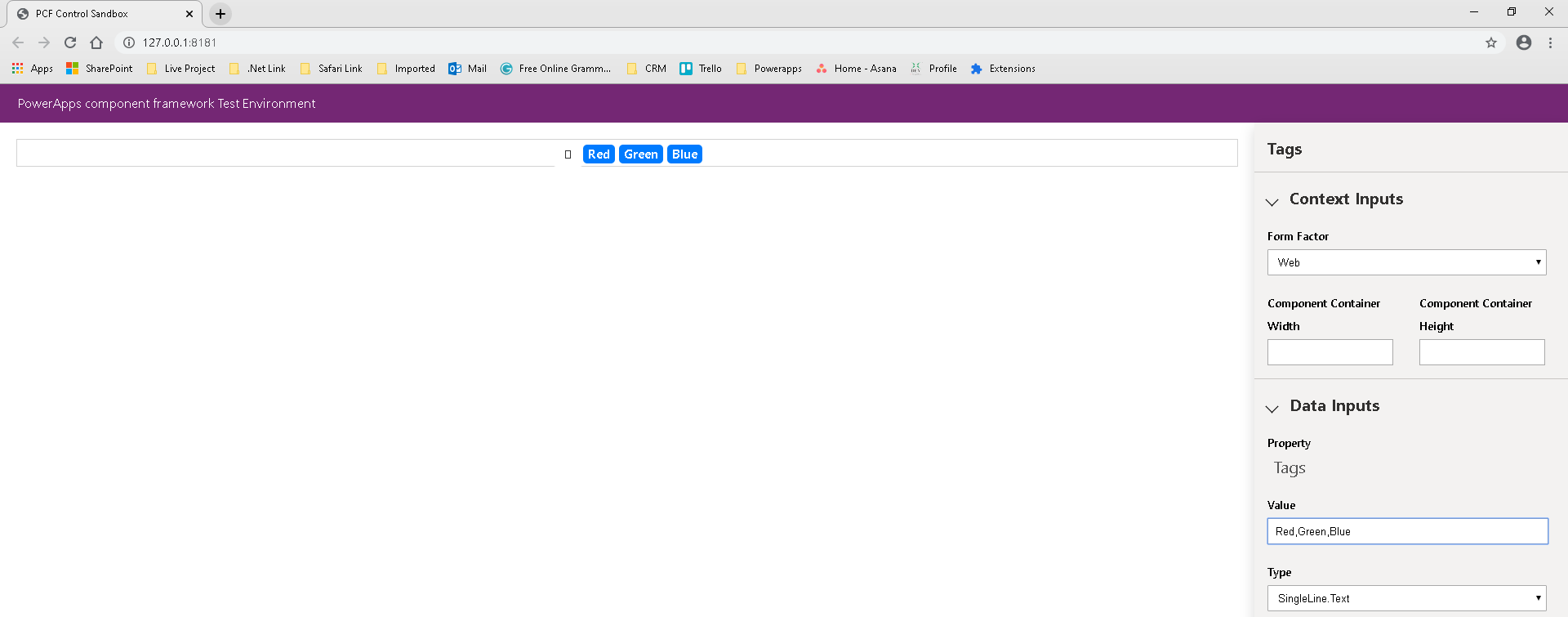


1. If the build is succeeded, then we can test it by running the control in a browser. To do so, you will need to use **npm start** command as shown below.

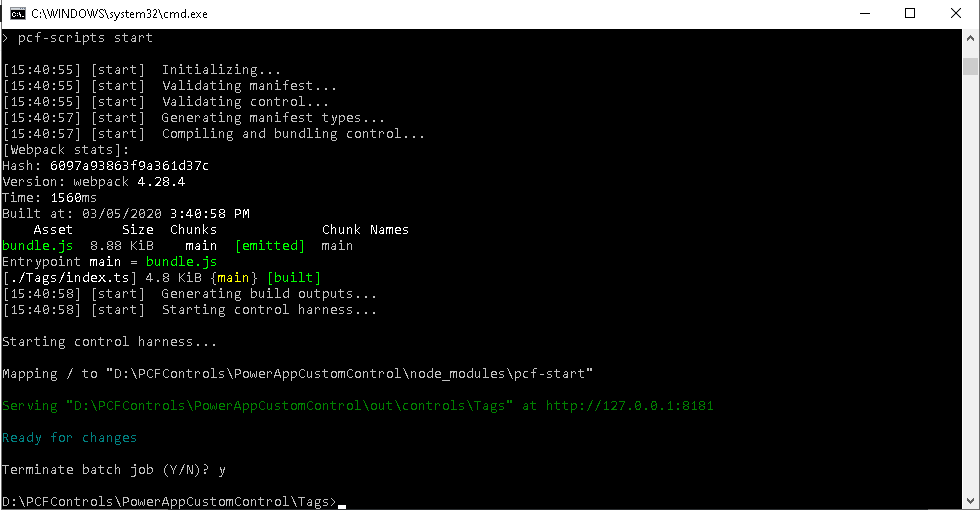


1. When testing in the browser; screen might look something like this. In my case as I do not have any output sent back I don’t see anything in there but if you have output sent back from the control you’ll see your output variables values as well.



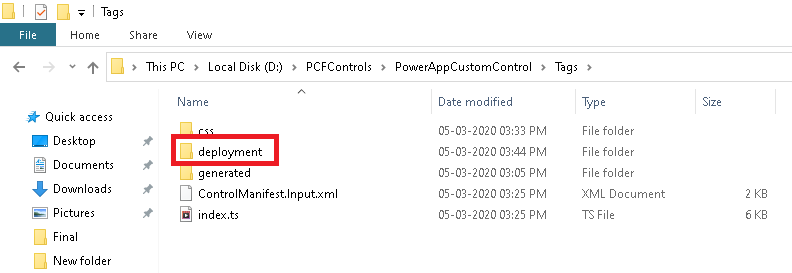


When we want to terminate the process press Ctrl + C



**Create a solution package for D365 CE**

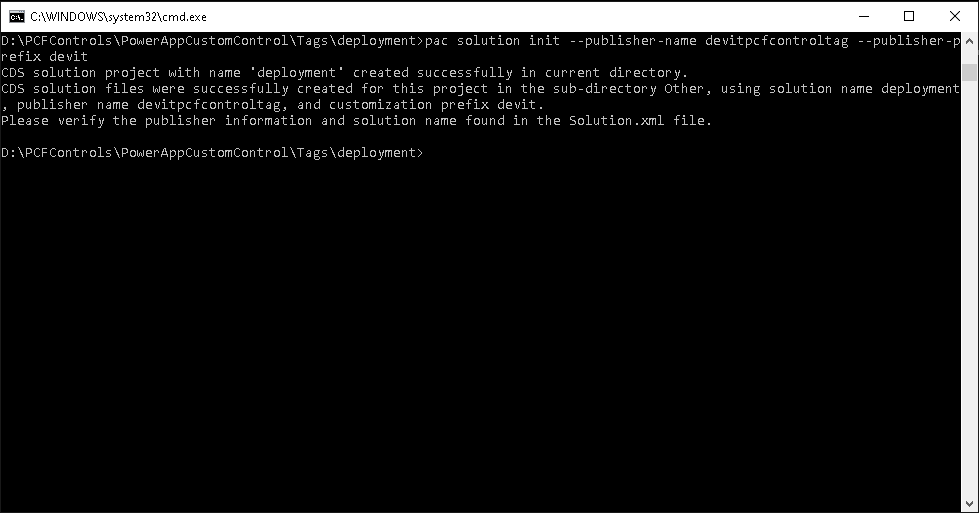
For this create a new directory inside your control folder. In my case I have created a folder named “deployment” inside the “Tags” folder.



Now, use **cd** command to navigate inside this newly created folder and run the following command to create a new solution project for D365 CE.

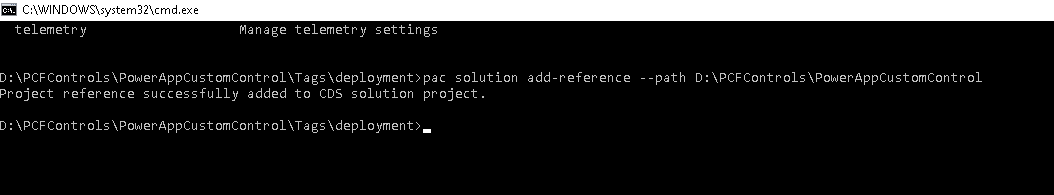
**cd D:\PCFControls\PowerAppCustomControl\Tags\deployment**

**pac solution init --publisher-name devitpcfcontroltag --publisher-prefix devit**



**pac solution add-reference --path D:\PCFControls\PowerAppCustomControl**

Once the solution project is created we need to add the component into this solution. To do this, we need to use the following command. The path needs to be where the project file (pcfproj) resides



Once the above command is executed, you’ll see deployment.cdsproj created.

We now have to run few more commands to create the .zip file we need for importing the solution in D365 CE. To do so, we need to execute **msbuild /t:restore** command followed by **msbuild** command.

Import this solution zip file in any of your favourite D365 CE instance and publish customization.