Custom Connector Lab

Power Platform Workshop

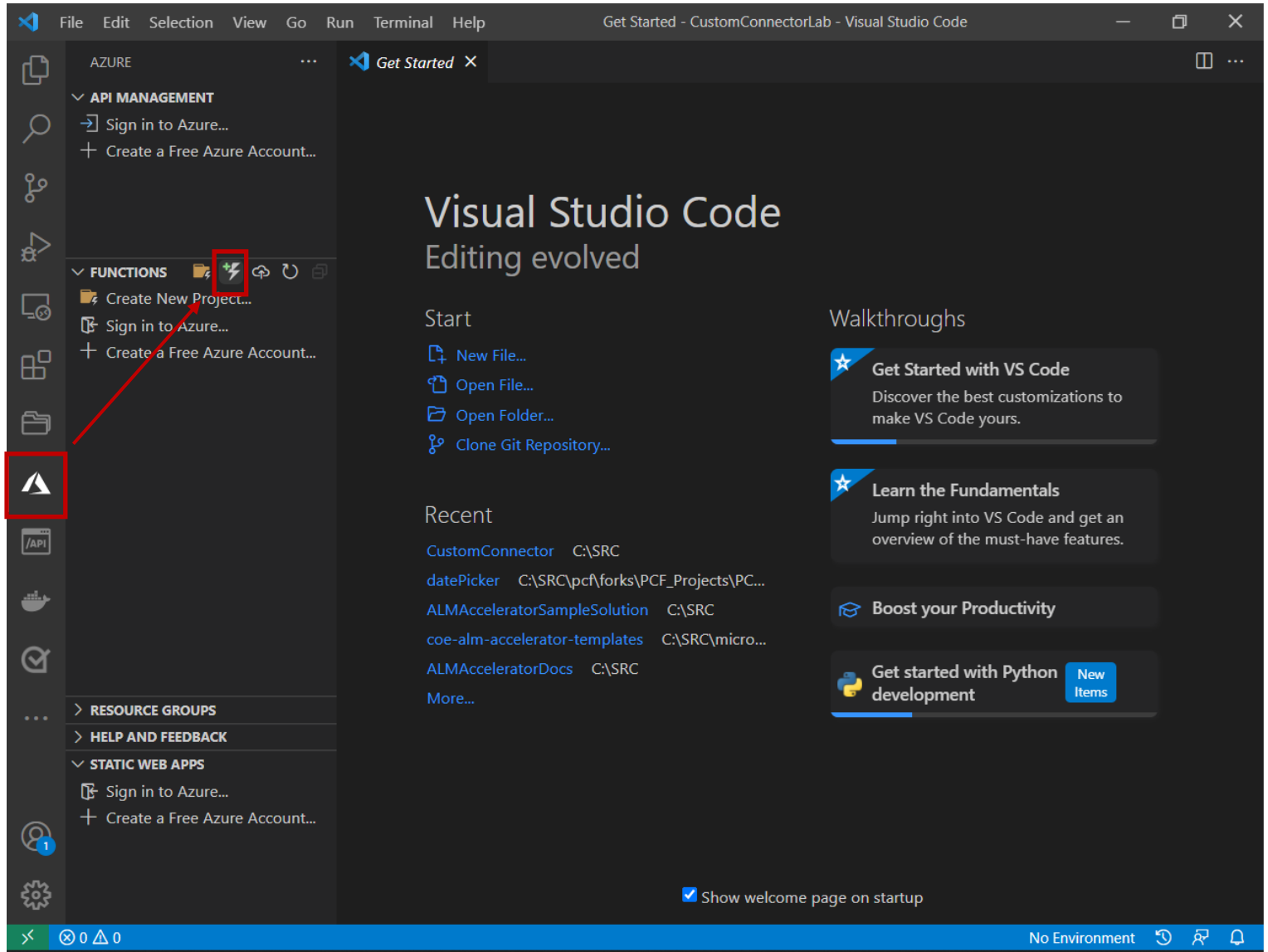
Step 1. Create a new Function using the Azure Functions Extension.

Select the [Azure Functions extension for Visual Studio Code] and add a new function using the lighting bolt icon.

Click yes if prompted to create a function project

Create your function project with the following properties

- Language: C#
- .NET runtime: .NET core 3
- Template: HTTP trigger with OpenAPI
- Name: GetWeather
- Namespace: CompanyName.Weather
- AccessRights: Anonymous*

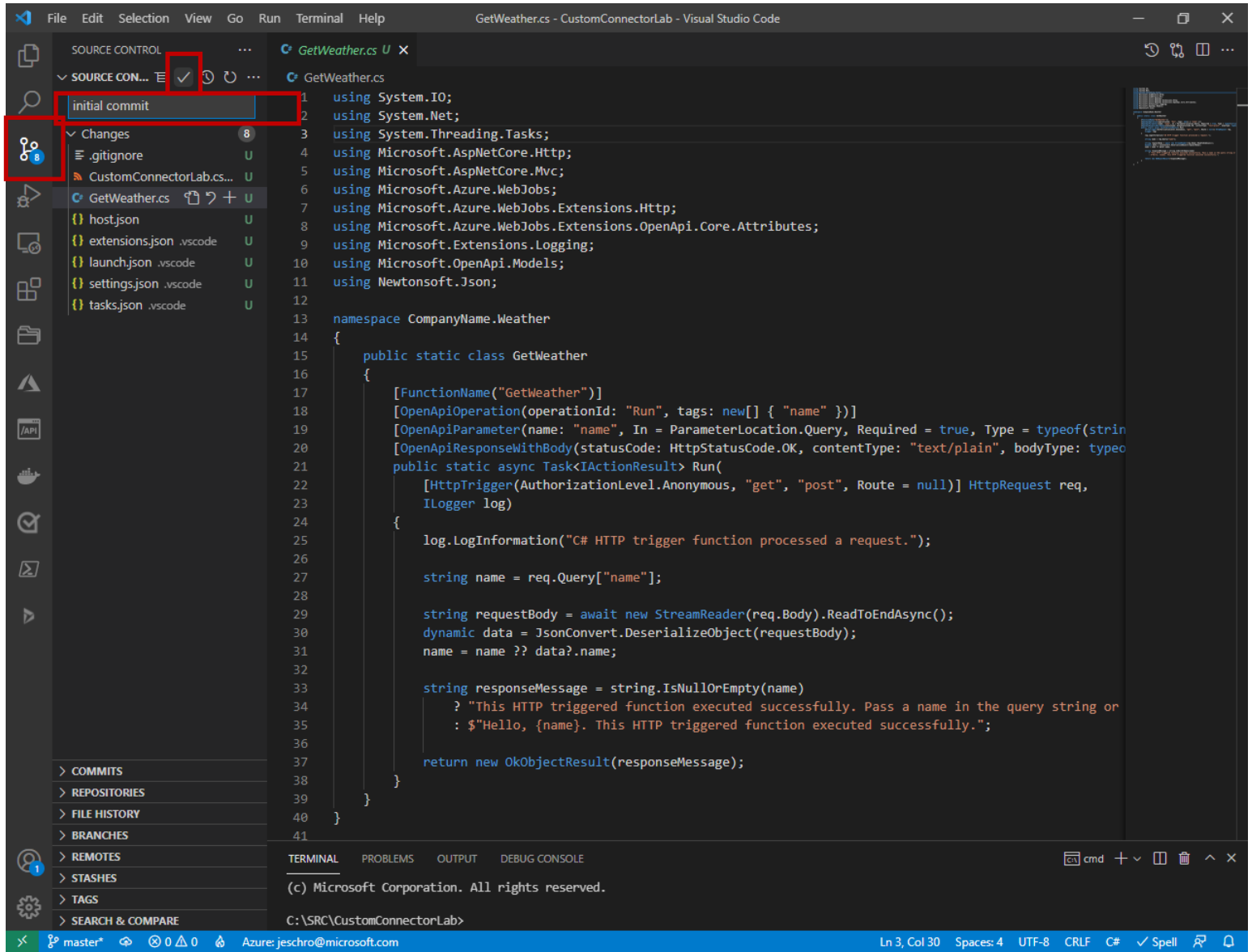


*For the purpose of this lab we will use anonymous access rights.
However, it is recommended to always secure your functions

Step 2. Check in the Function to version control.

Select the [GitLens extension]

Enter a commit message and click the commit (✓) button

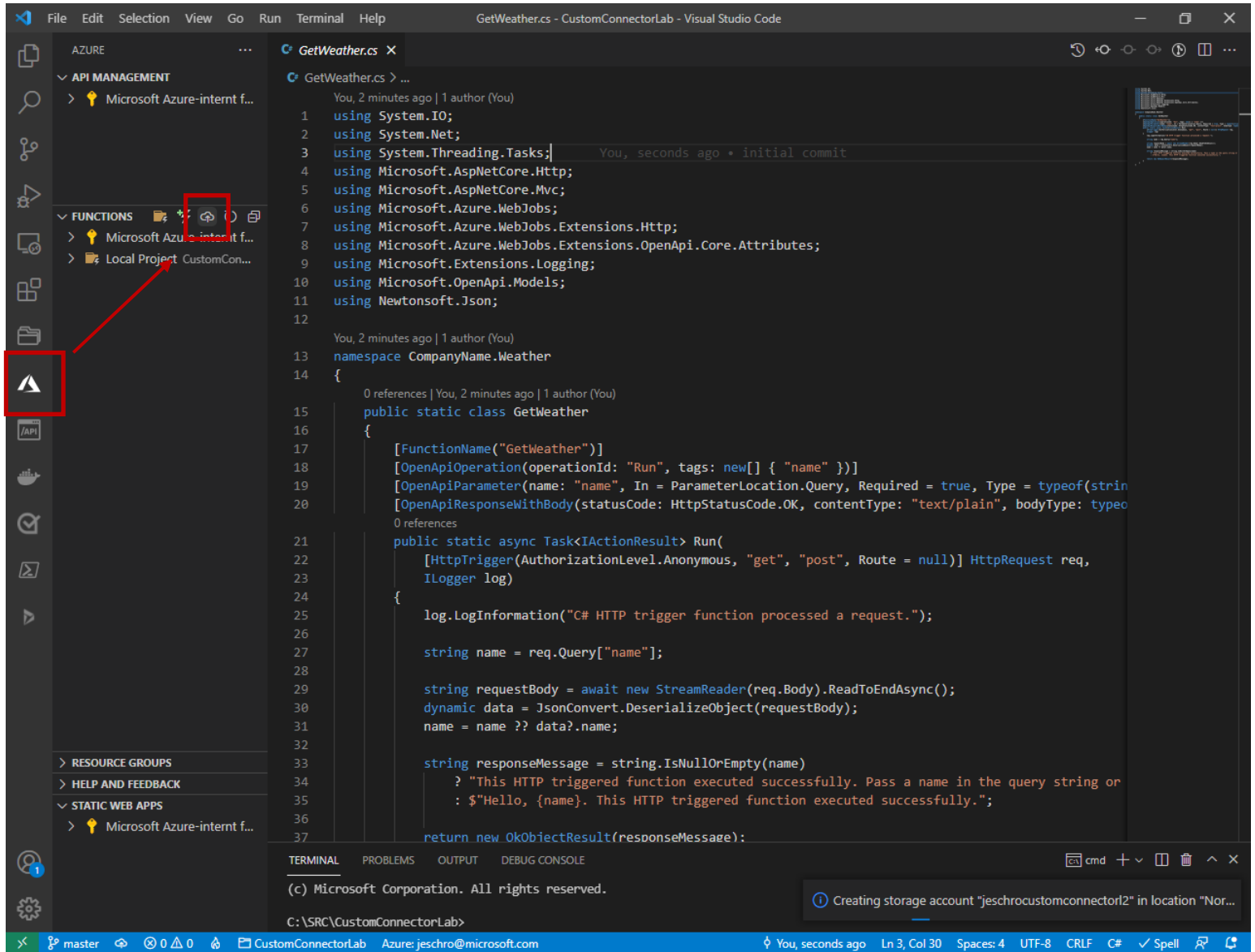


Step 3. Deploy this function into the Azure Subscription.

Select the Azure Functions extension for Visual Studio Code and deploy your function app to Azure by clicking the cloud icon.

Create new Function App in Azure with the following properties

- Name: ?(must be globally unique)
- Runtime: .NET core 3.1
- Location: North Europe (recommended)



Step 4. Run the Function interactively in a browser.

In this example the function was deployed to a function with the name “GetTemperature” therefore can be called using the following URL

https://jeschro-customconnectorlab.azurewebsites.net/api/GetWeather

This returned the following:

“This HTTP triggered function executed successfully. Pass a name in the query string or in the request body for a personalized response.”

If the query string parameter **name** is passed in like the following:

https://jeschro-customconnectorlab.azurewebsites.net/api/GetWeather?name=Jens

Will return the following:

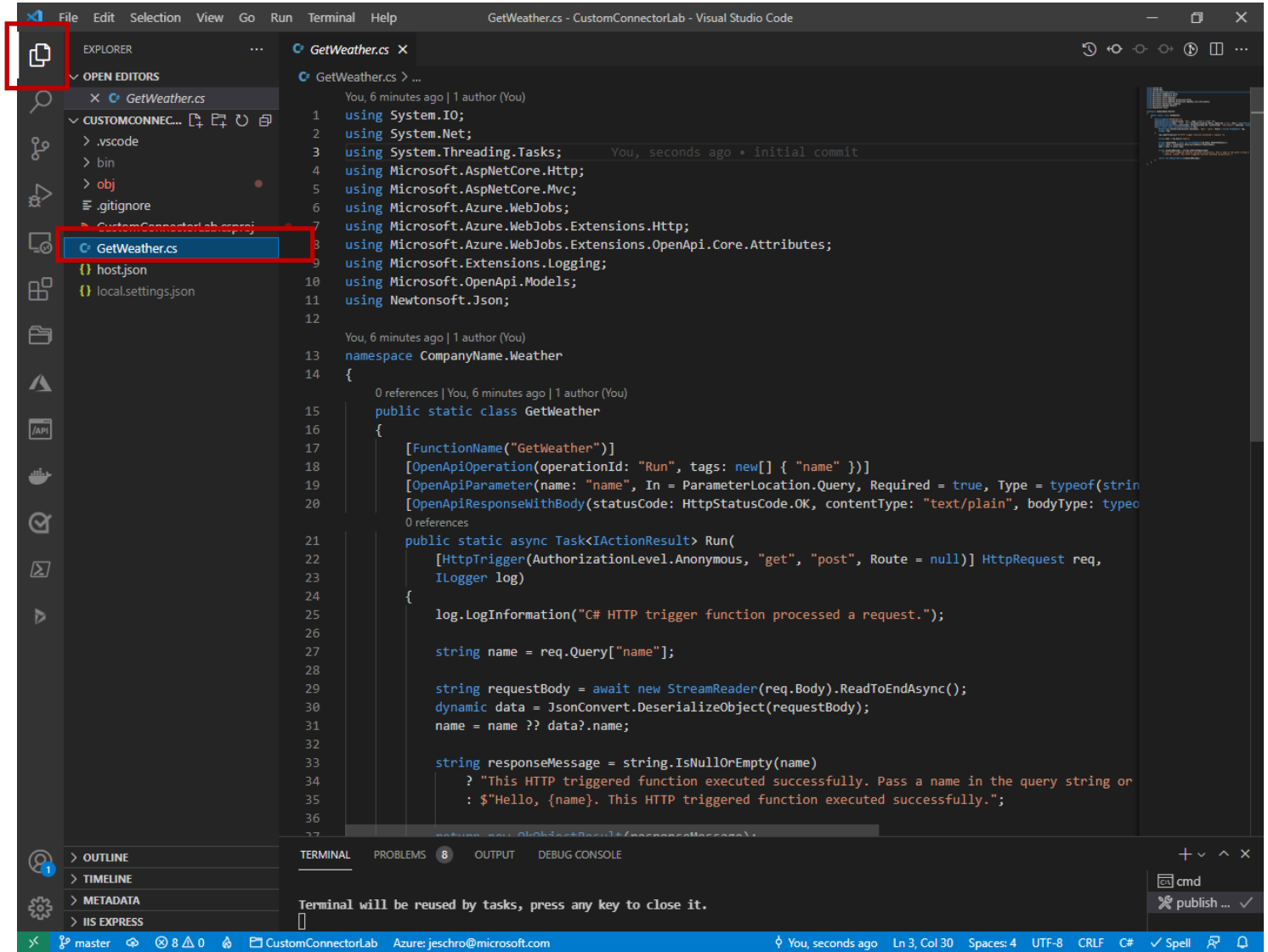
“Hello, Jens. This HTTP triggered function executed successfully.”

While Azure Functions extension for Visual Studio Code makes it easy to create an HTTP based function the default template returns a string, it does NOT return JSON…So this functions cannot be called from a Flow in Power Automate or Power Apps. Both requires connectors to return JSON

We will update the function to return JSON in the next steps

Step 5. Open the function implementation.

Click on the files view of Visual Studio code and the GetWeather.cs function file



Step 6. Add the System.Collections.Generic namespace

Add the following line of code at the end of the exisiting using statements in the top of you function code

We will be implementing logic using the List type in the System.Collections.Generic namespace in the following steps

```
12      using System.Collections.Generic;
```

Step 7. Add supporting classes

Add the weatherForecast and location classes to the Function App

Insert the code to the right inside the namespace but outside the static function class (before the last **}**)

```
56      public class weatherForecast {
57          public List<location> locations { get; set; }
58      }
59
60      public class location {
61          public string locationName { get; set; }
62          public string forecast { get; set; }
63      }
```

Step 8. Edit the function to weather forecast

Replace the following code (lines 33-37) :

```
34            string responseMessage = string.IsNullOrEmpty(name)
35                ? "This HTTP triggered function executed successfully. Pass a name in the query string or in the
request body for a personalized response."
36                : $"Hello, {name}. This HTTP triggered function executed successfully.";
37
38            return new OkObjectResult(responseMessage);
```

With this code :

```
34            // Weather Forecast Object
35            weatherForecast weatherForecastObject = new weatherForecast();
36            weatherForecastObject.locations = new List<location> {
37                new location { locationName = "Bergen", forecast = "Rain" },
38                new location { locationName = "Oslo", forecast = "Cloudy" },
39                new location { locationName = "Stavanger", forecast = "Sunny" }
40            };
41
42            // Find location matching querystring parameter name
43            var forecast = weatherForecastObject.locations.Find(x => x.locationName.ToLower() == name.ToLower());
44
45            // if location matching querystring parameter name is not found return error message
46            if(forecast == null) {
47                forecast = new location { locationName = name, forecast = $“Location '{name}’ not found" };
48            }
49            // return weather forecast for location
50            return new OkObjectResult(jsonForecast);
```

Step 9. Edit the function to return JSON continued

You code should look like the code in the following pages

Full code page 1 of 2

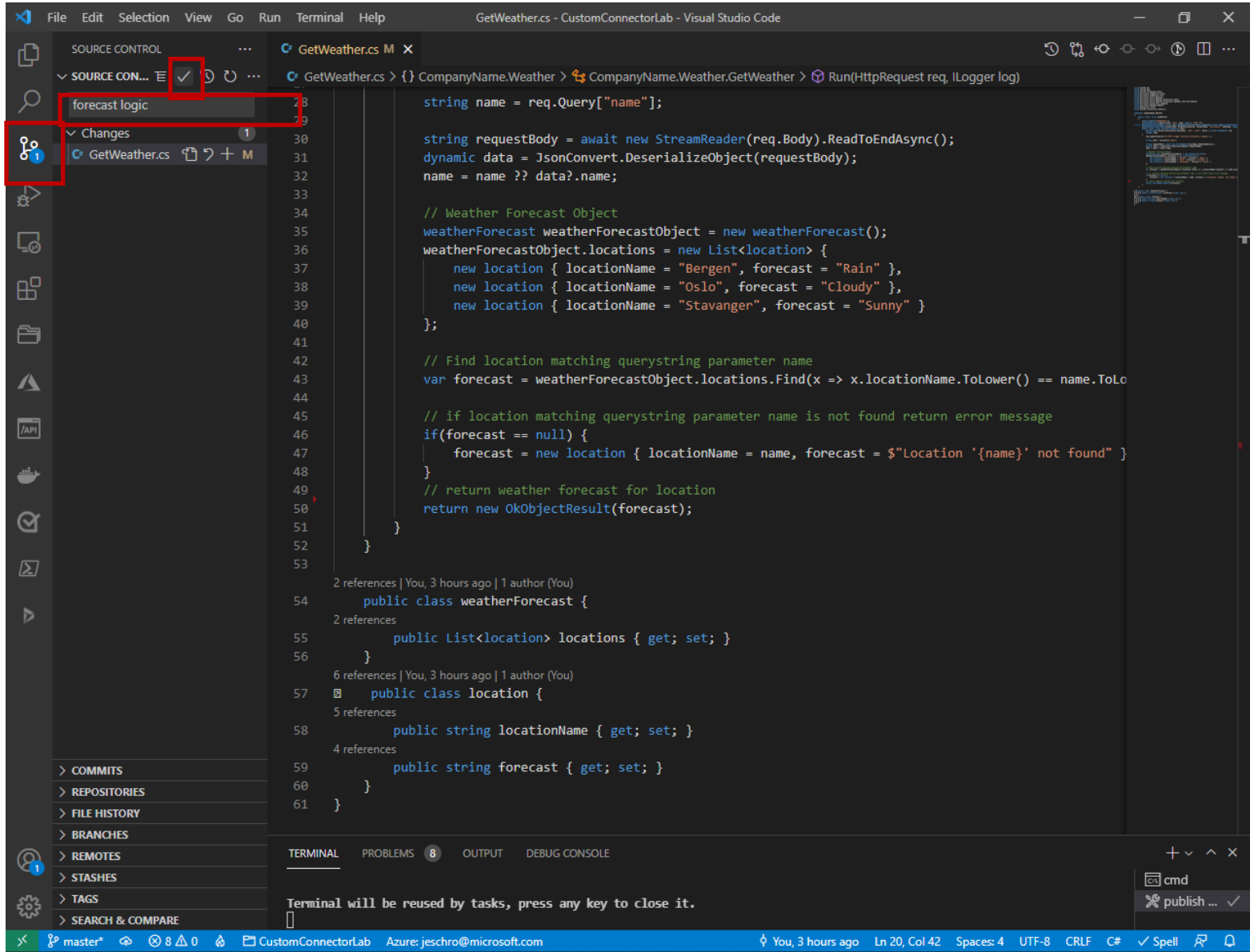
```
using System.IO;
using System.Net;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Extensions.OpenApi.Core.Attributes;
using Microsoft.Extensions.Logging;
using Microsoft.OpenApi.Models;
using Newtonsoft.Json;
using System.Collections.Generic;

namespace Equinor.CustomConn
{
    public static class HttpTrigger
    {
        [FunctionName("HttpTrigger")]
        [OpenApiOperation(operationId: "Run", tags: new[] { "name" })]
        [OpenApiParameter(name: "name", In = ParameterLocation.Query, Required = true, Type = typeof(string), Description = "The **Name** parameter")]
        [OpenApiResponseWithBody(statusCode: HttpStatusCode.OK, contentType: "text/plain", bodyType: typeof(string), Description = "The OK response")]
        public static async Task<IActionResult> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)] HttpRequest req,
            ILogger log)
        {
            log.LogInformation("C# HTTP trigger function processed a request.");

            string name = req.Query["name"];

            string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
            dynamic data = JsonConvert.DeserializeObject(requestBody);
            name = name ?? data?.name;
```

Full code page 2 of 2

```
            // Weather Forecast Object
            weatherForecast weatherForecastObject = new weatherForecast();
            weatherForecastObject.locations = new List<location> {
                new location { locationName = "Bergen", forecast = "Rain" },
                new location { locationName = "Oslo", forecast = "Cloudy" },
                new location { locationName = "Stavanger", forecast = "Sunny" }
            };

            // Find location matching querystring parameter name
            var forecast = weatherForecastObject.locations.Find(x => x.locationName.ToLower() == name.ToLower());

            // if location matching querystring parameter name is not found return error message
            if(forecast == null) {
                forecast = new location { locationName = name, forecast = $"Location '{name}' not found" };
            }
            // return weather forecast for location
            return new OkObjectResult(forecast);
        }
    }

    public class weatherForecast {
        public List<location> locations { get; set; }
    }

   public class location {
        public string locationName { get; set; }
        public string forecast { get; set; }
    }
}
```

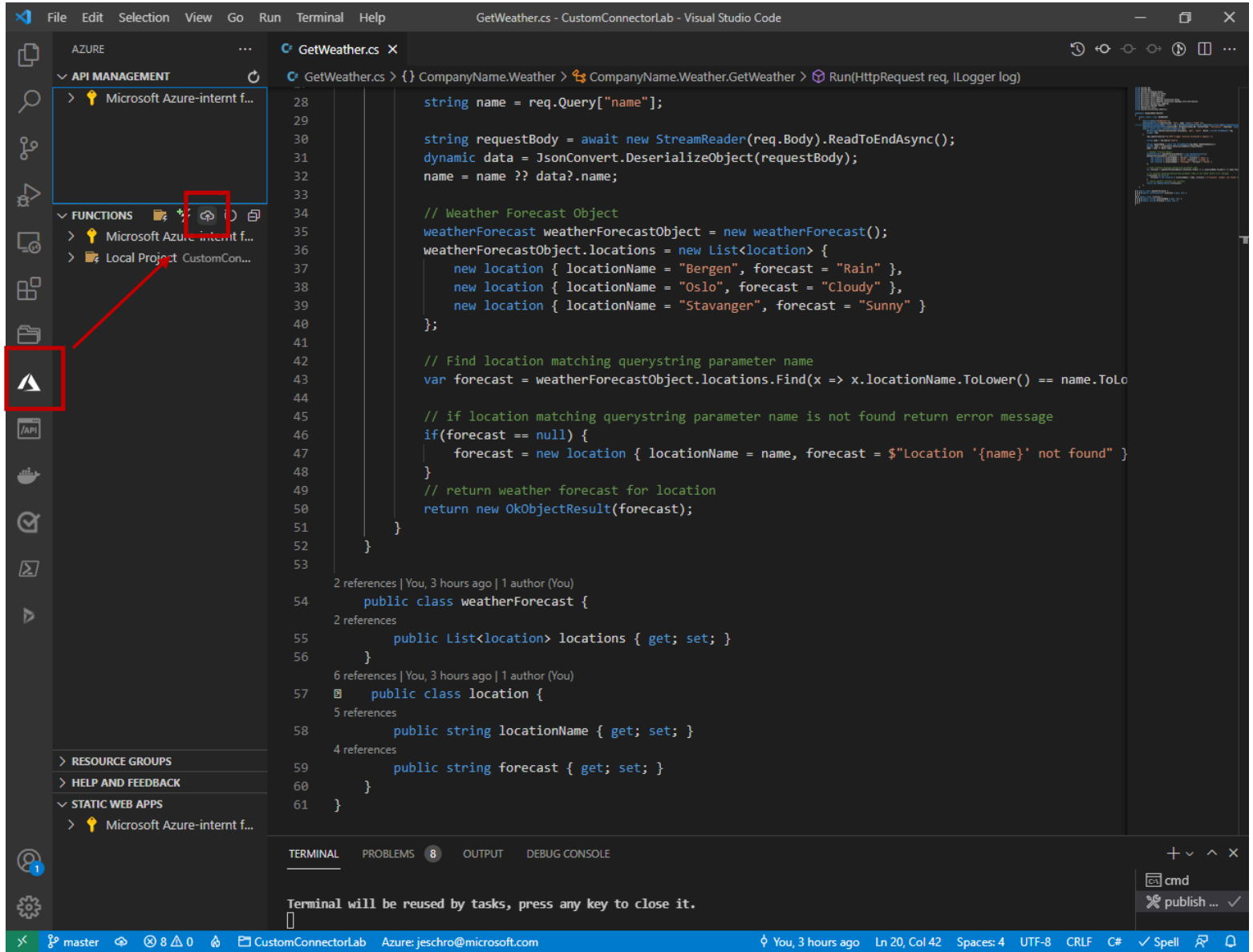
Step 10. Check in this update.

Save and commit your changes



Step 11. Deploy these updates:

Deploy your changes to the Azure Function App you created earlier.

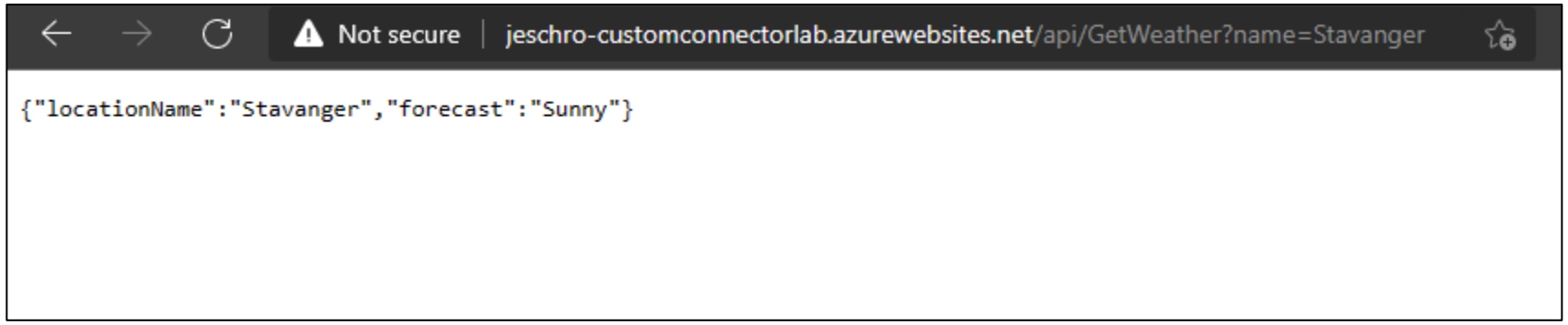


Step 12. Run the function interactively from the Browser.

Running the updated browser will now return the following JSON

Note, the name parameter is required for the function to work properly. We will handle this in the Custom Connector implementation

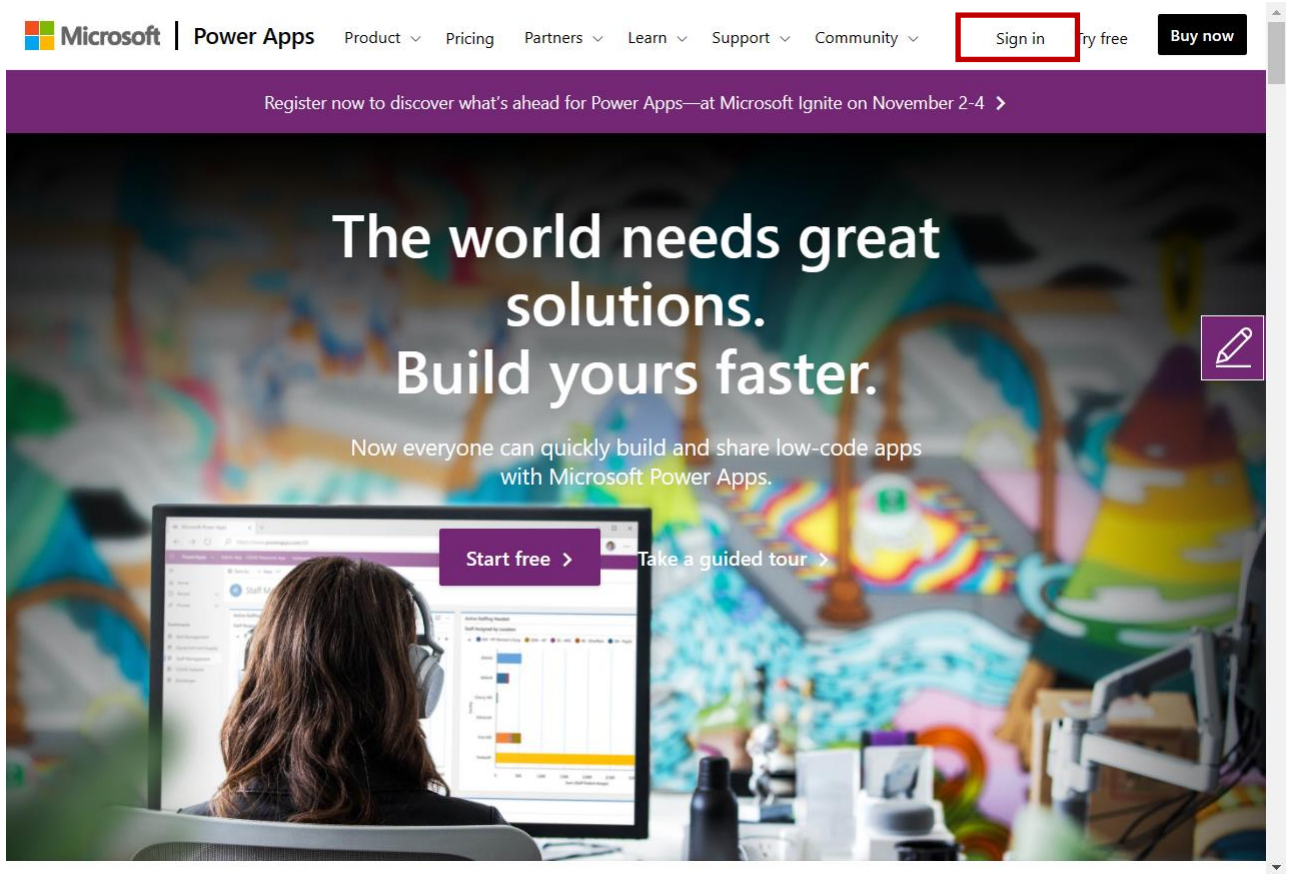
Congratulations your Azure function is now ready to be called by Power Apps!



Creating a Power Platform custom connector to wrap the Azure Function

Step 13. Log in to Power Apps.

Got to https://powerapps.microsoft.com/ and sign in with your organization account



Step 14. Create a solution

Once signed in click **Solutions**.

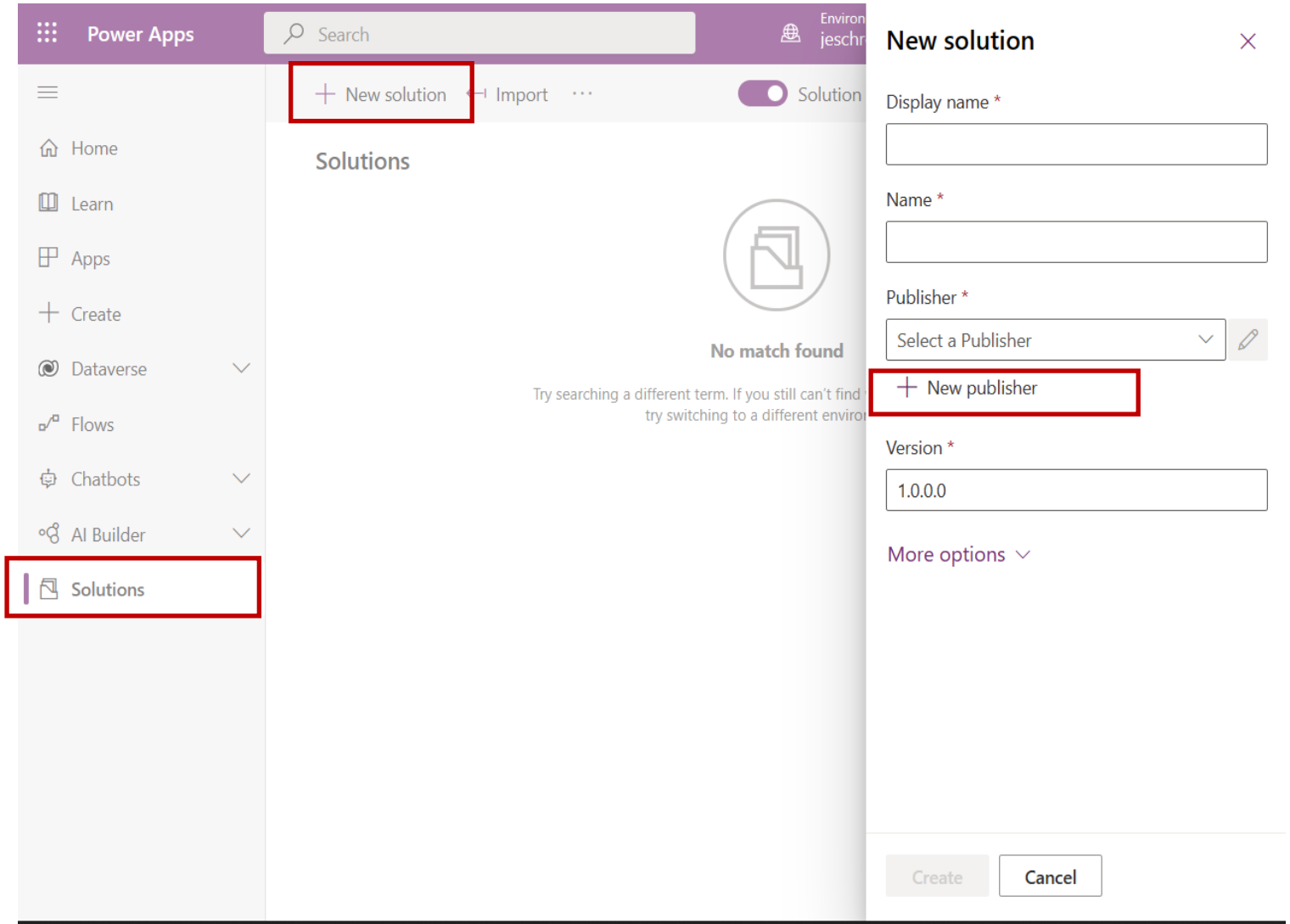
Then click **+ New solution**

In the New solution pane click **+ New publisher**

Note, a publisher enables us to have multiple components with same name as their schema name will always be prefixed with the publisher prefix.

It also helps us identify components created by different publishers

When creating a solution you must select a publisher.

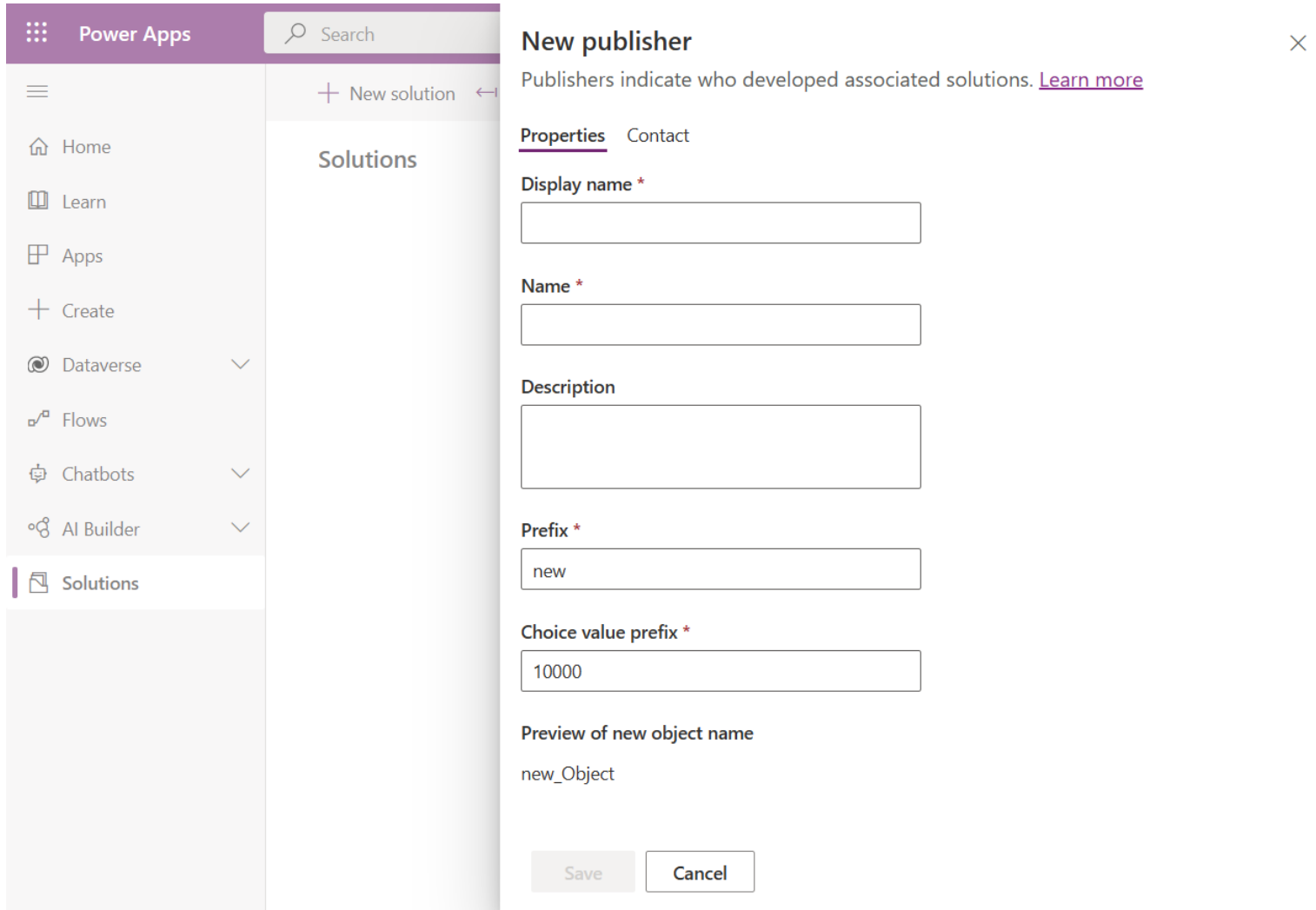


Step 15. Create a Publisher

Enter the following details:

- Display name : your name
- Name : your name with no white spaces
- Prefix: your alias
- Choise value prefix: leave as is

Click Save

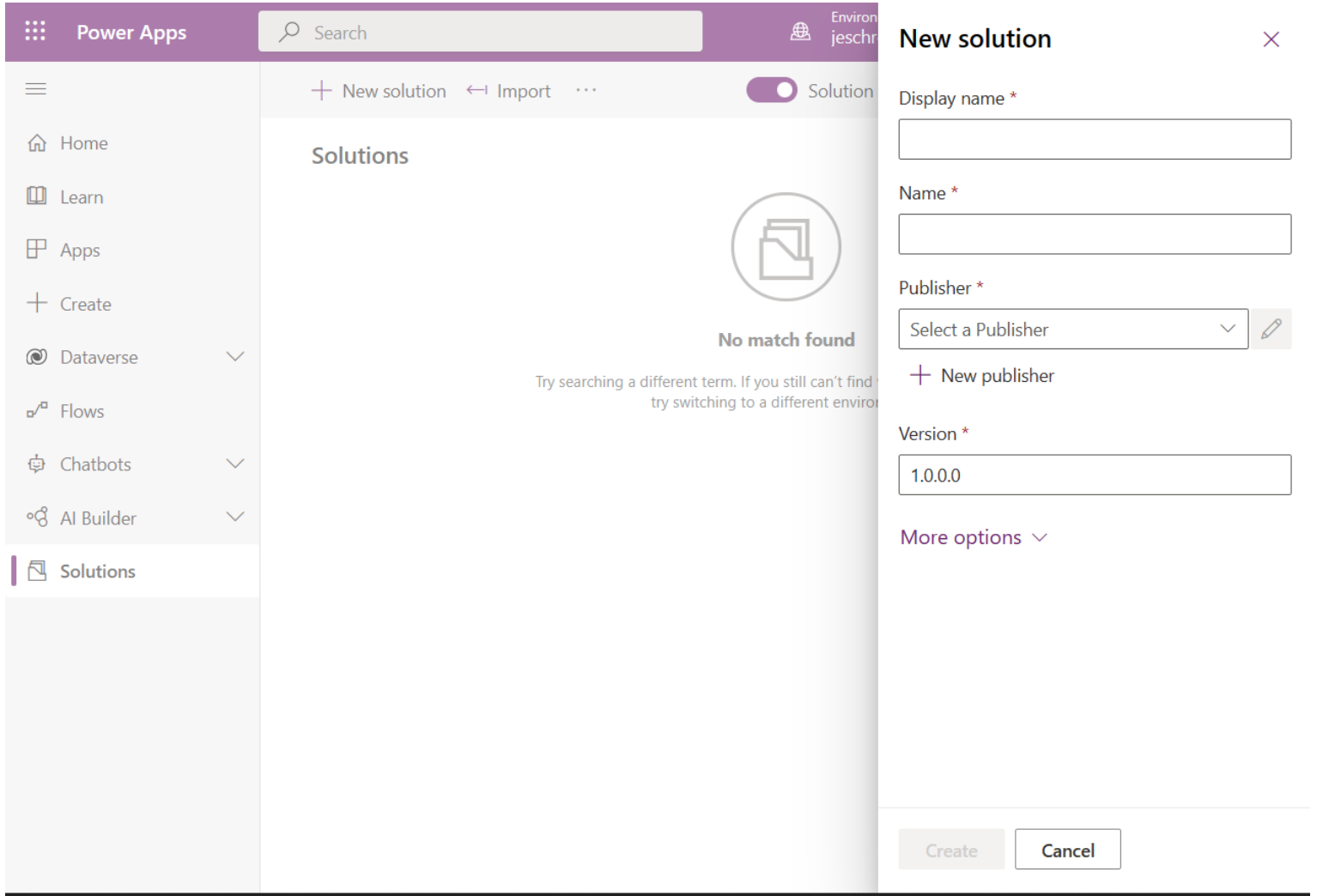


Step 16. Complete the solution creation

Back at the create solution pane, enter the following details:

- Display name: name you solution. Include your alias
- Name: leave as it is generated
- Publisher: select the publisher you just created

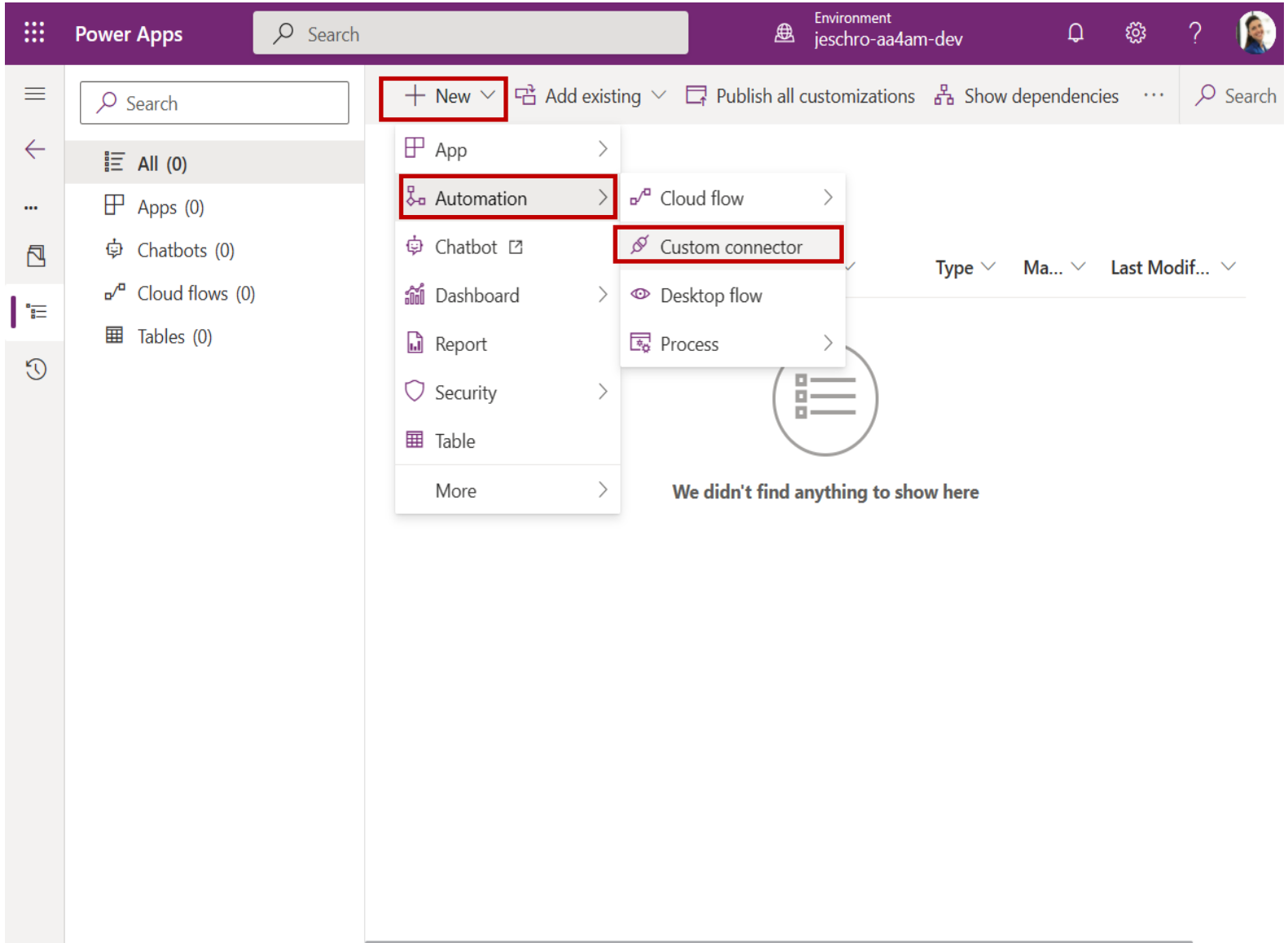
Click Create and click you newly created solution



Step 17. Add a custom connector to your solution

Open the solution you just created.

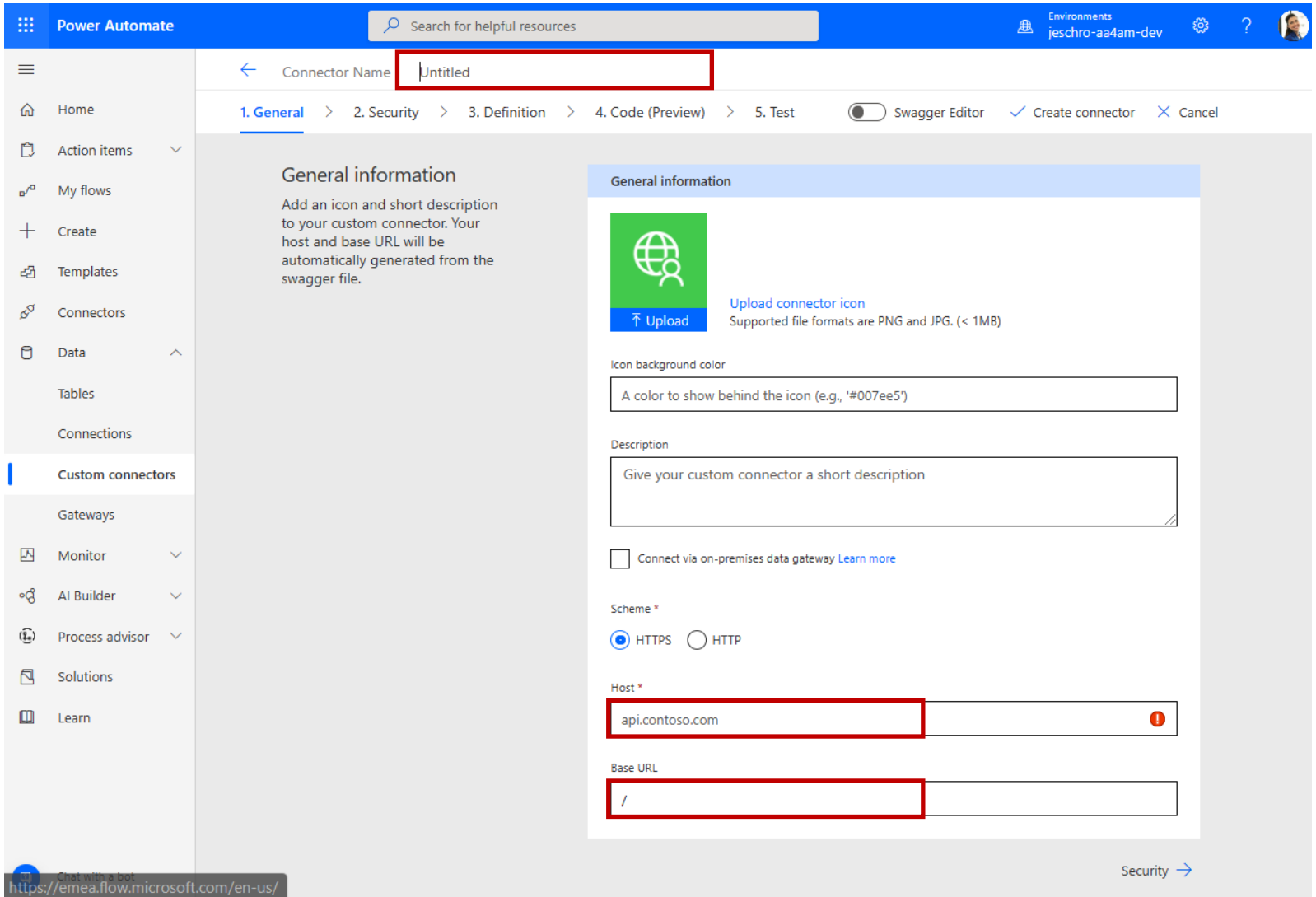
Click New => Automation => Custom connector



Step 18. Set the General properties of you custom connector

Enter the following details:

- Connector Name: alias + WeatherConnector
- Host: [your function app name].azurewebsites.net
- Base URL: /api

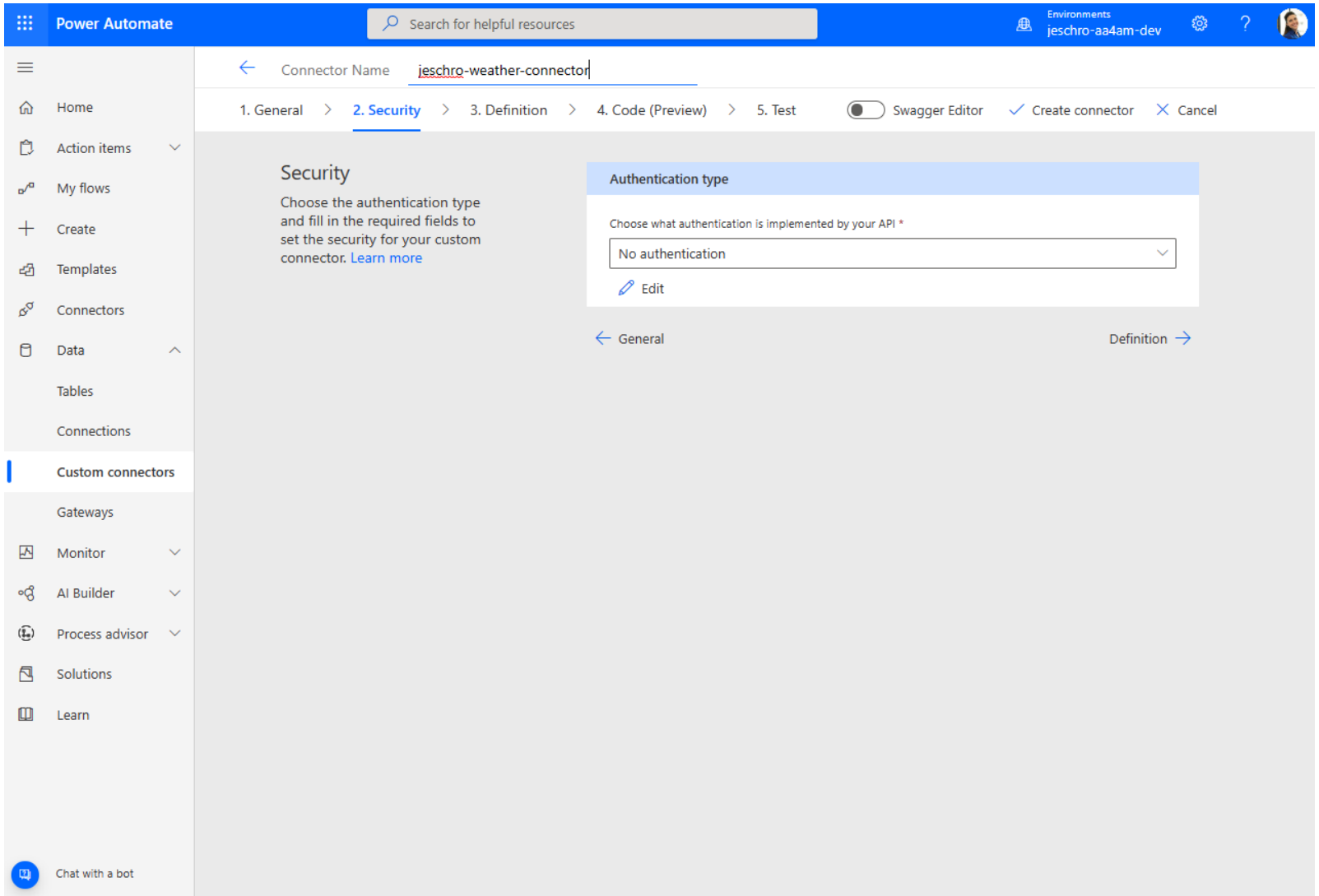


Step 19. Security properties

In this lab we will not configure any security settings.

However, have a look at the different authentication methods that are supported.

For Azure services it is common to use Oauth 2 with Azure AD Identity Provider.

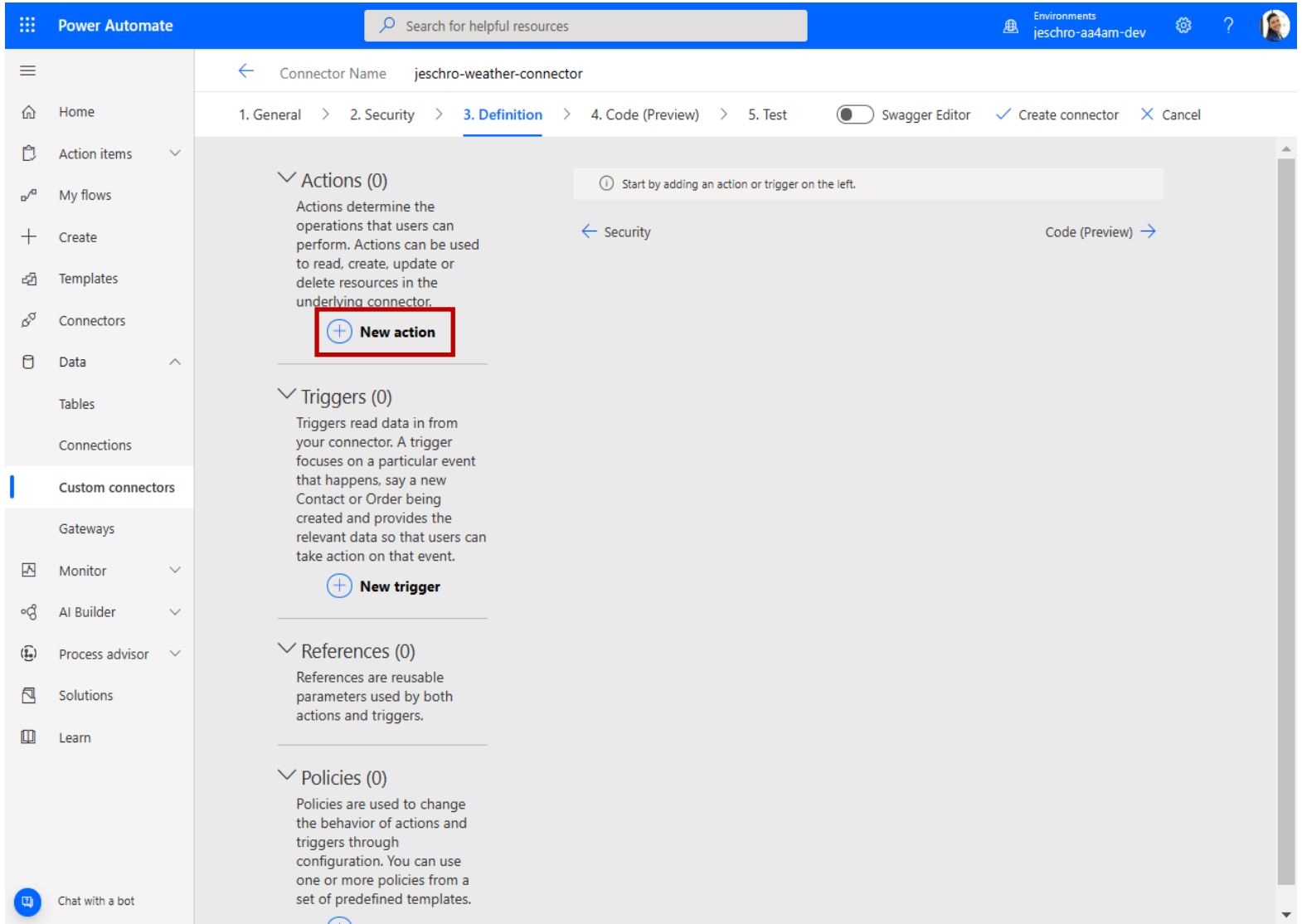


Step 20. Definition properties

This is where we define the functionality of our connector

We must create an action to retrieve the weather forecast

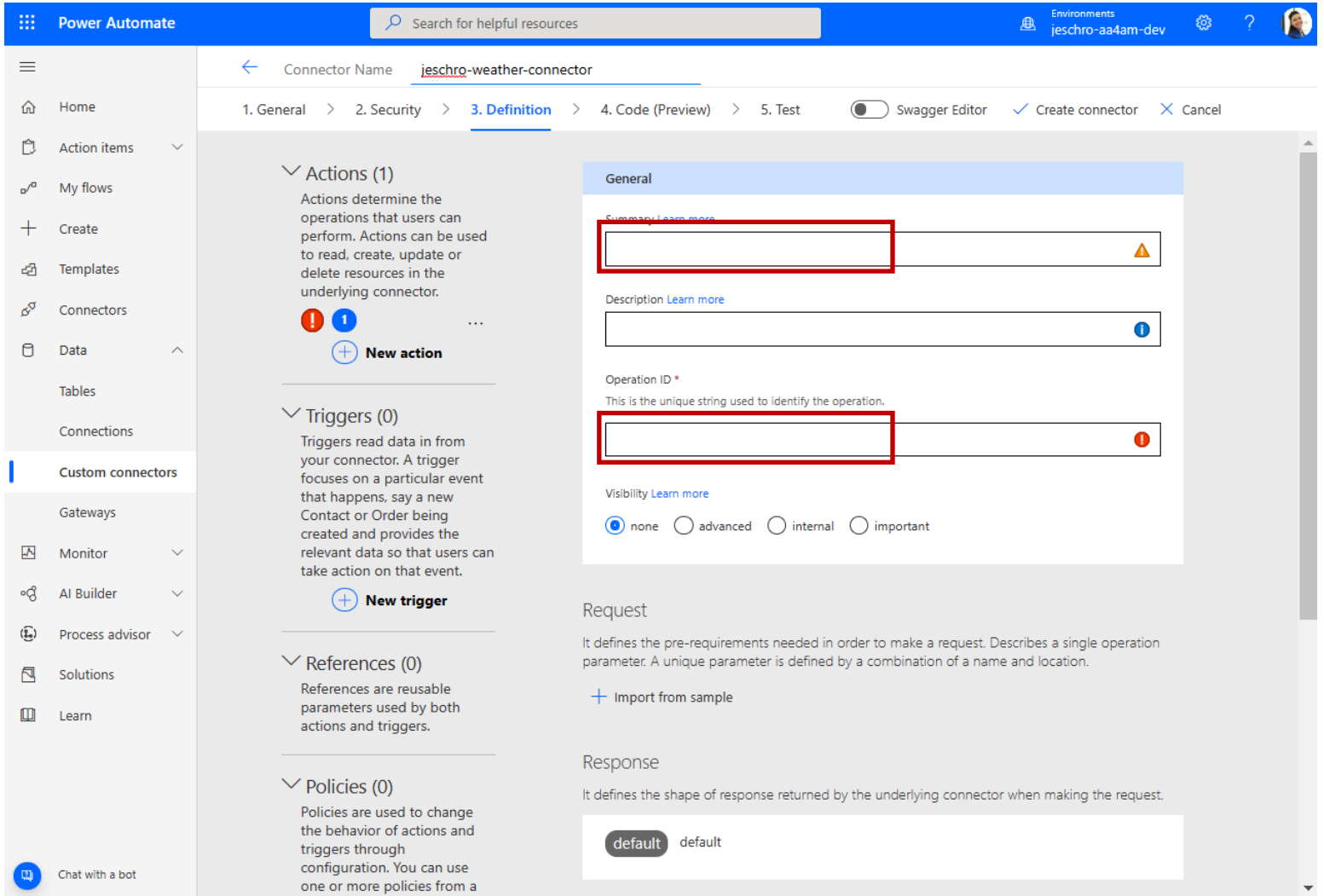
Start by clicking **+New action**



Step 21. Define the General properties of your new action

Enter the following details in the General section:

- Summary: Get weather by location
- Operation ID: GetWeatherByLocation

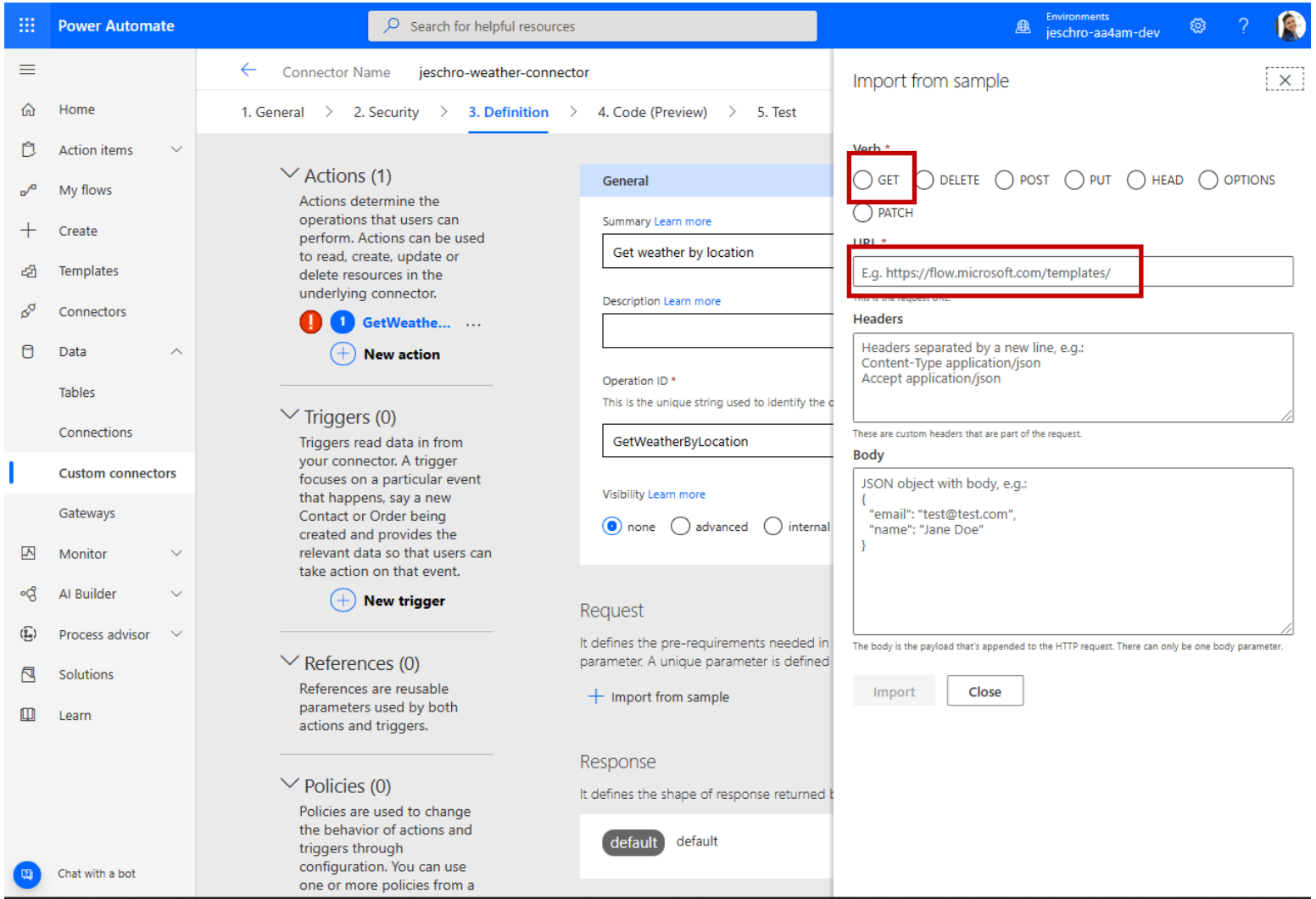


Step 22. Now define the Request schema of the action

Click **+import from sample** in the Request section and fill in the following details:

- Verb: Get
- Url: GetWeather?name={Location}

Click **Import**



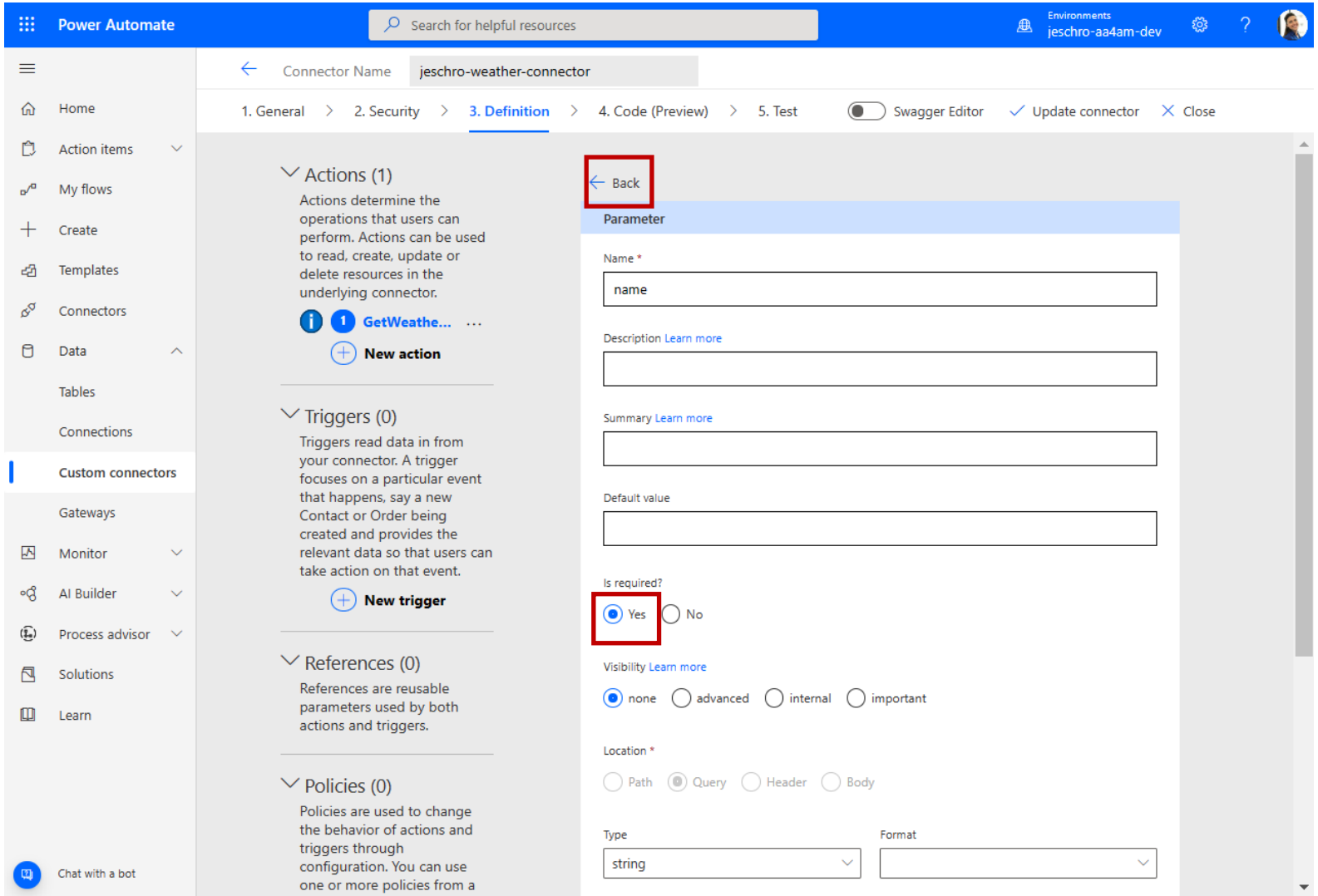
Step 23. Now define the Request schema of the action - Continued

Since we have not implemented code to handle empty/null values in our name parameter we should set the name property to required

Click the **name => Edit** in the Query section of the Request section.

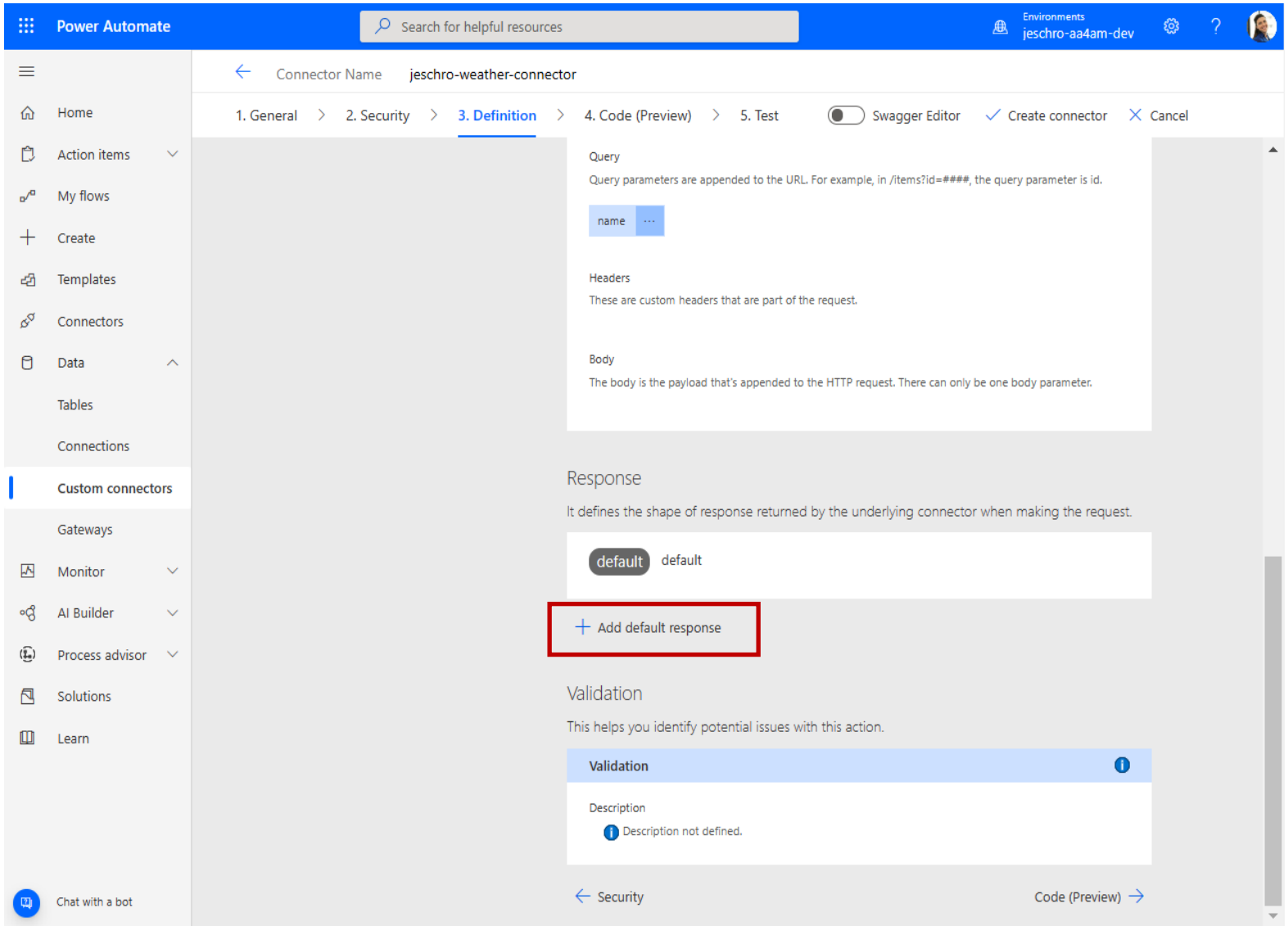
Set **Is required?** to **Yes**

Click <- **Back** just above the Parameter section



Step 24. Now define the Response object

Scroll down to the Response section and click **+ Add default response**

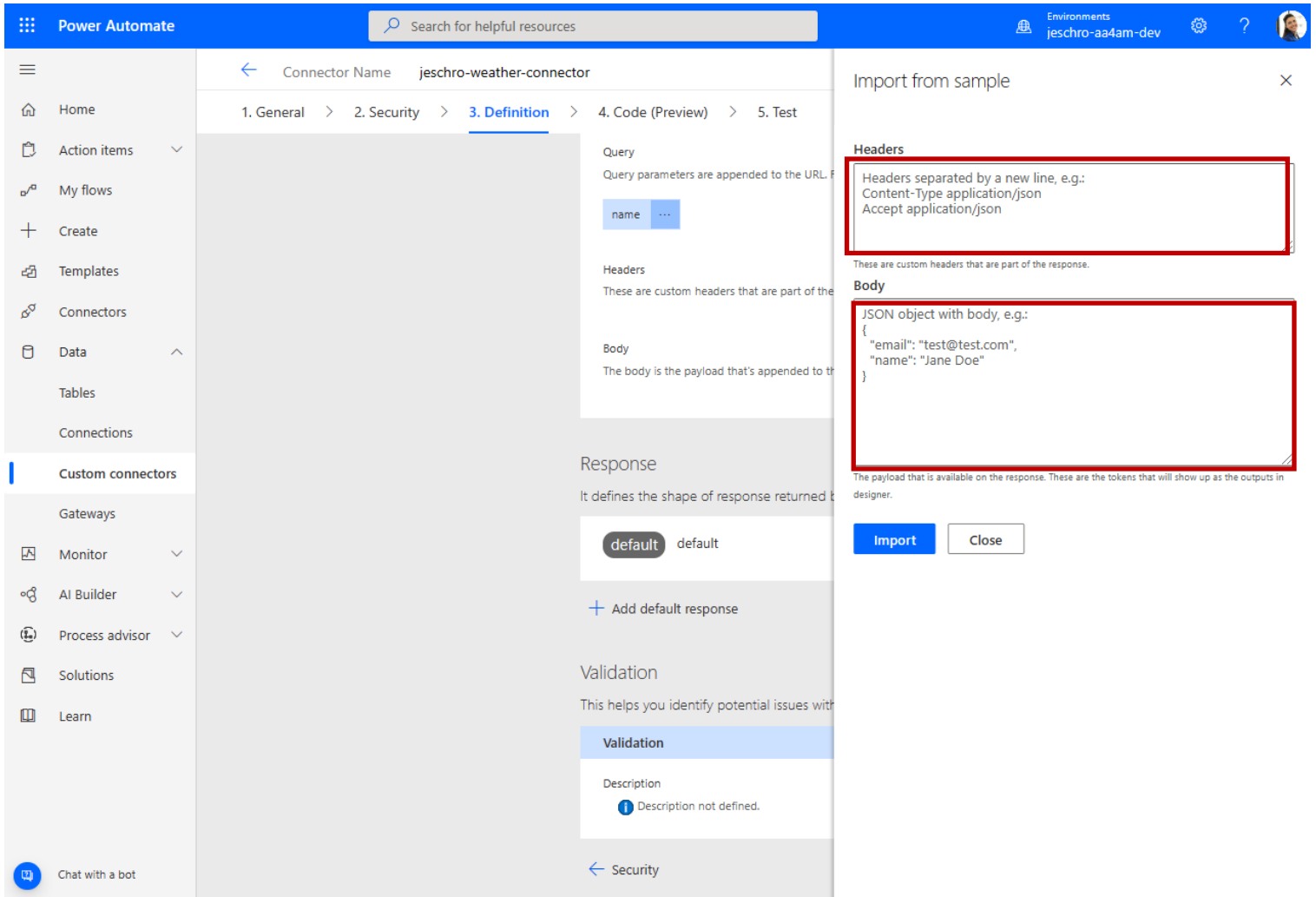


Step 25. Continue defining the Response object

Enter the following details:

- Headers: Content-Type application/json
- Body: Paste in the output of your function as a sample:
  {"name":"Oslo","forecast":"Cloudy"}

Click **Import**

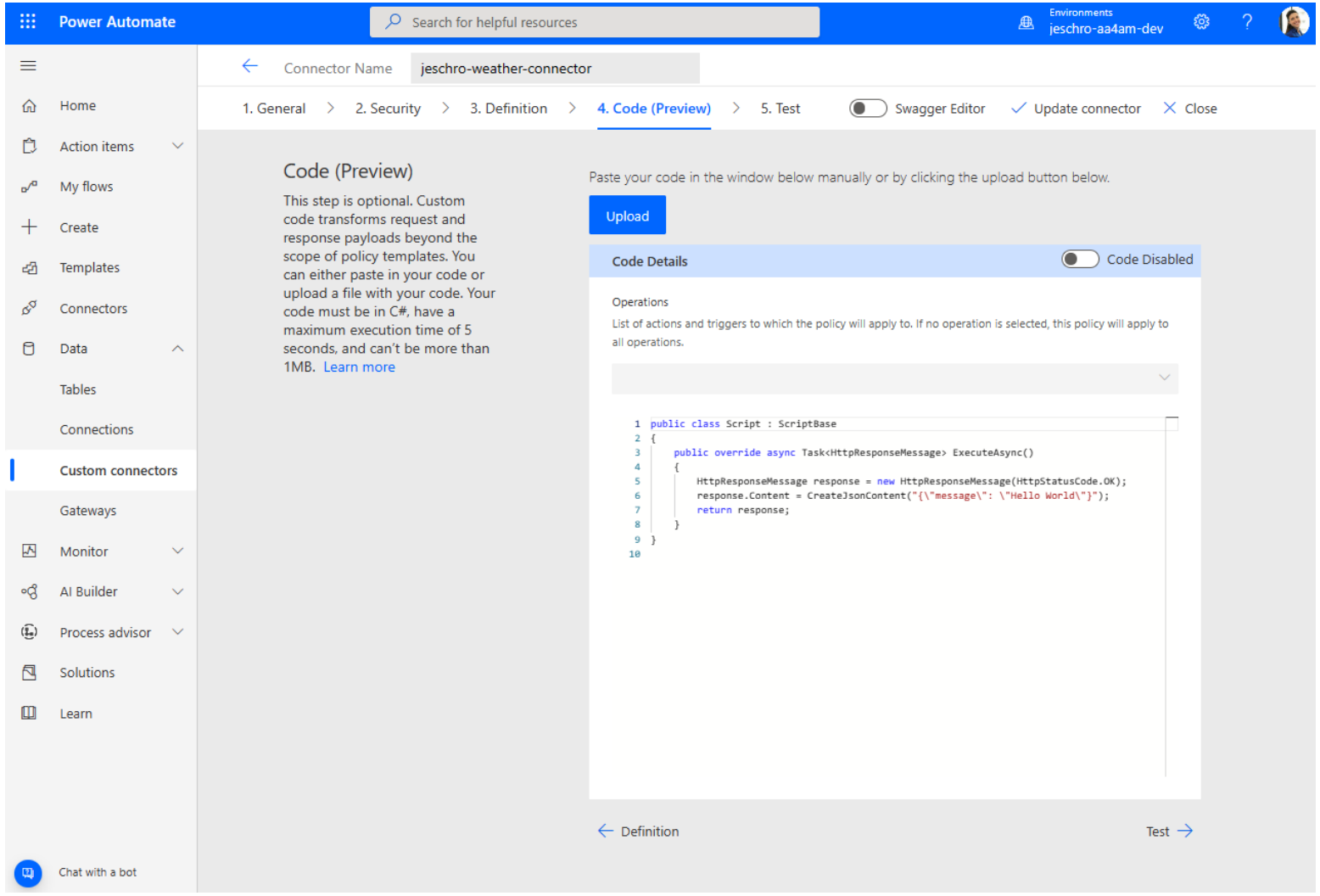


Step 26. Continue past the Code (Preview) configuration

In this Lab we will not implement Code in the connector

Custom Connector Code is a Preview feature that allows you to implement advanced logic via C# code

[Write code in a custom connector | Microsoft Docs]



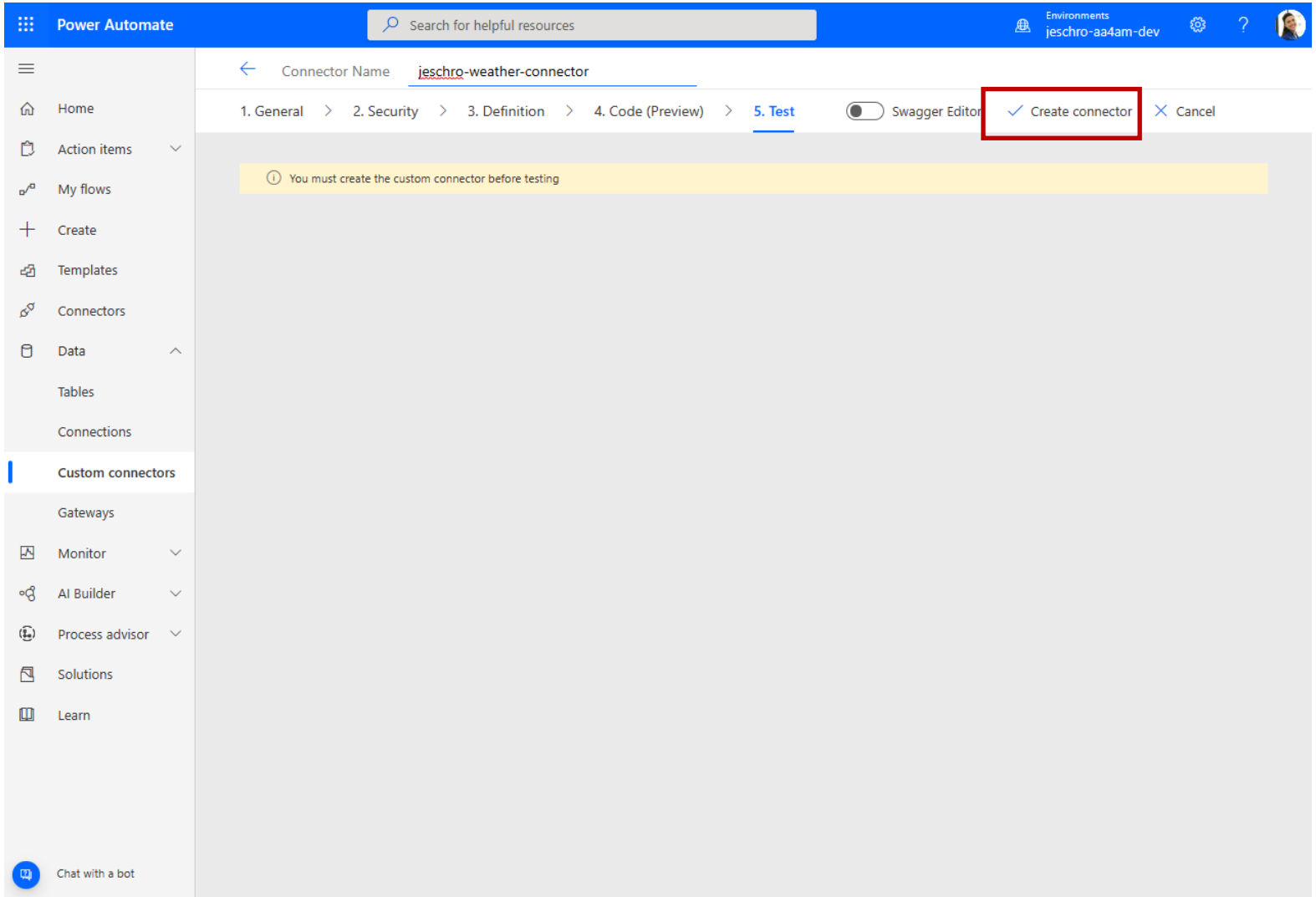
Step 27. Create/Update your Custom Connector

Before we can test the connector it must be created first.

Click **Create connector**

If you have already created you connector make sure you update it with the latest changes you have made.

Note, it will take a short period of time for the connector to be provisioned in the supporting infrastructure



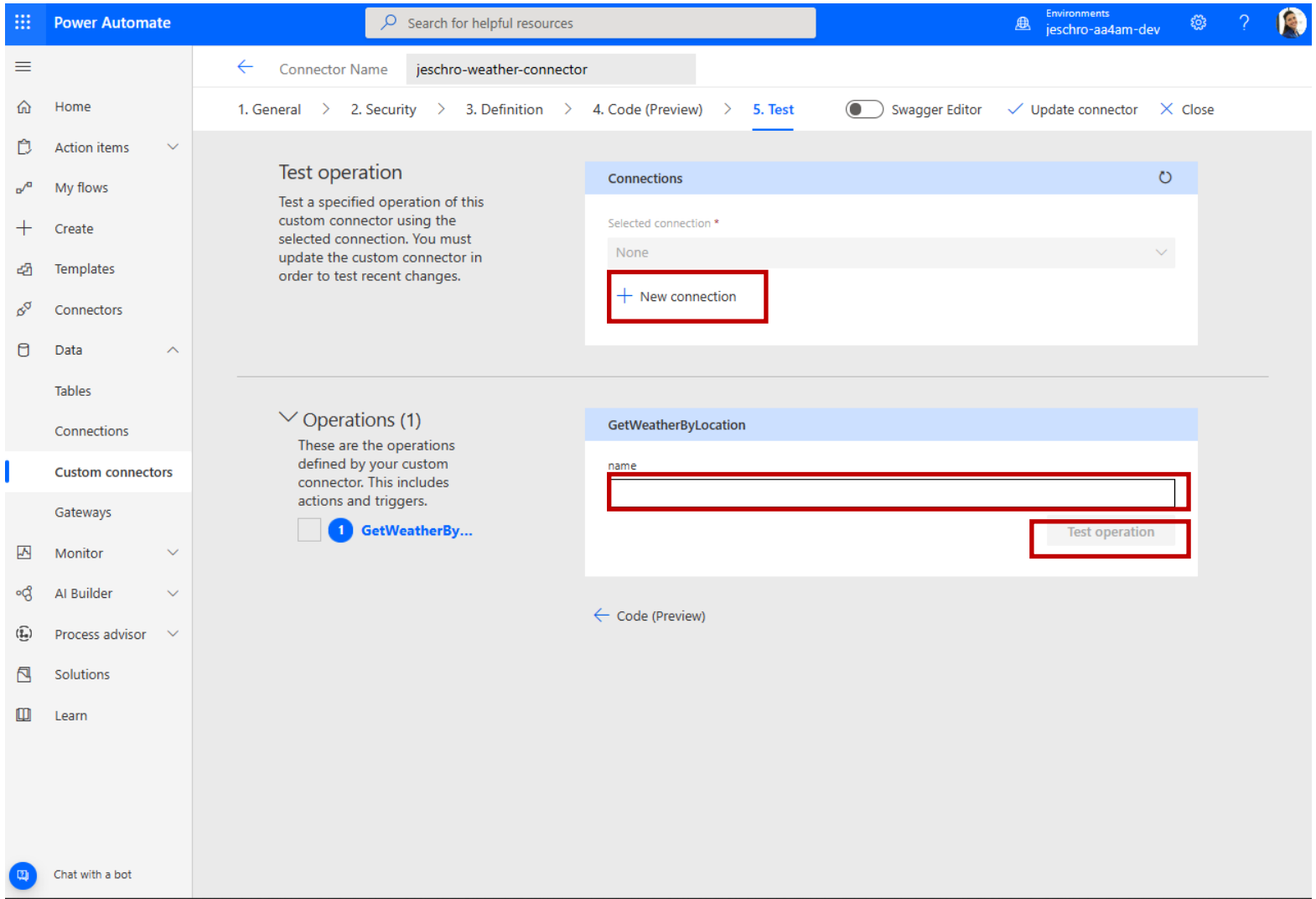
Step 28. Test your Custom Connector

In order to use your customer connector you must first create a connection.

Click **+ New connection**

Note, because we havent implemented any authentication mechanism the connection will be created without user interaction. If authentication was implemented the user would be required to authenticate (username + password, Oauth sign in or API Key) to create a connection.

Enter a value in the name property and click **Test operation**



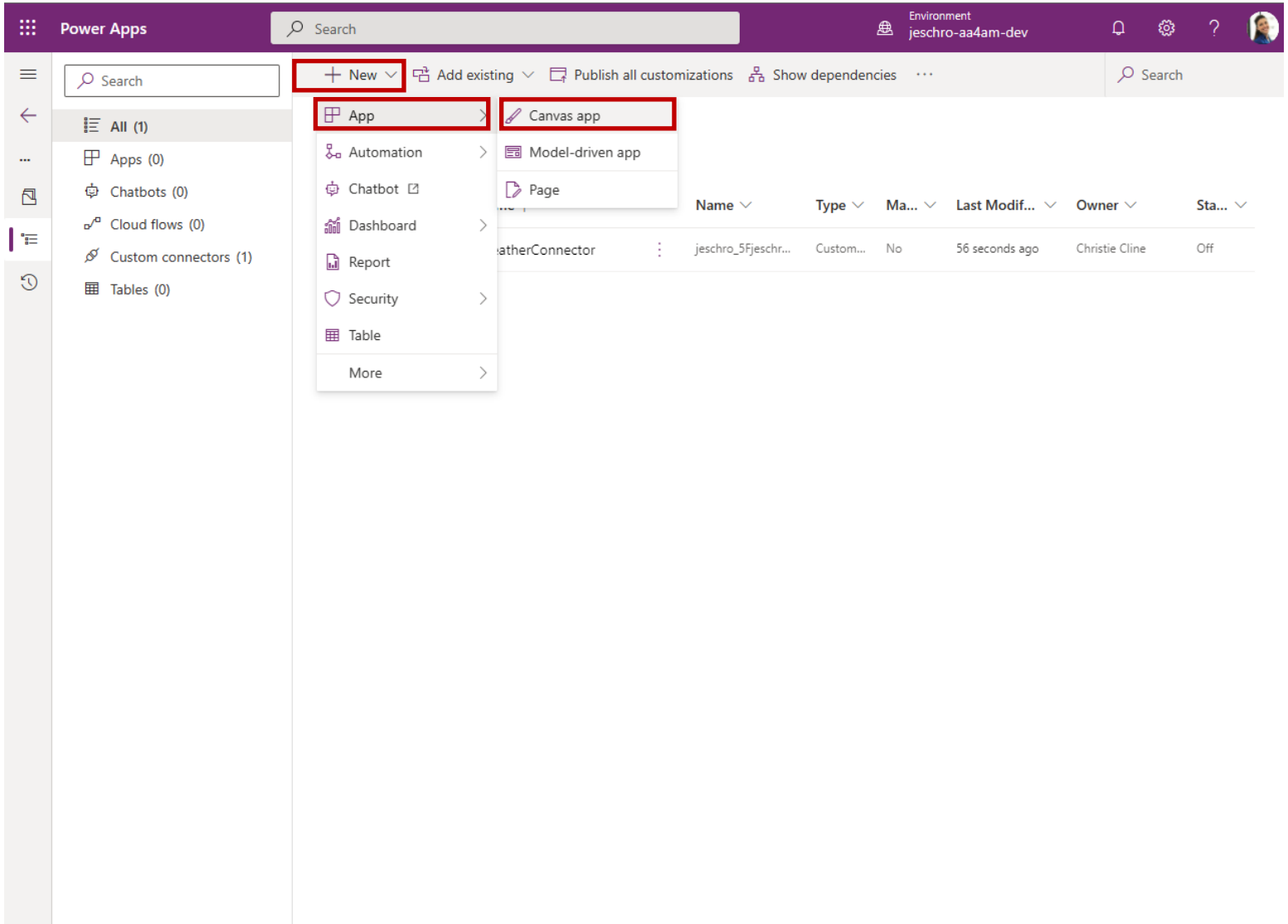
Using a Power Platform custom connector in a Power App

Step 29. Create a new Canvas App

Navigate to the solution you created earlier.

Notice that the custom connector has been added

Click **+New => App => Canvas app**



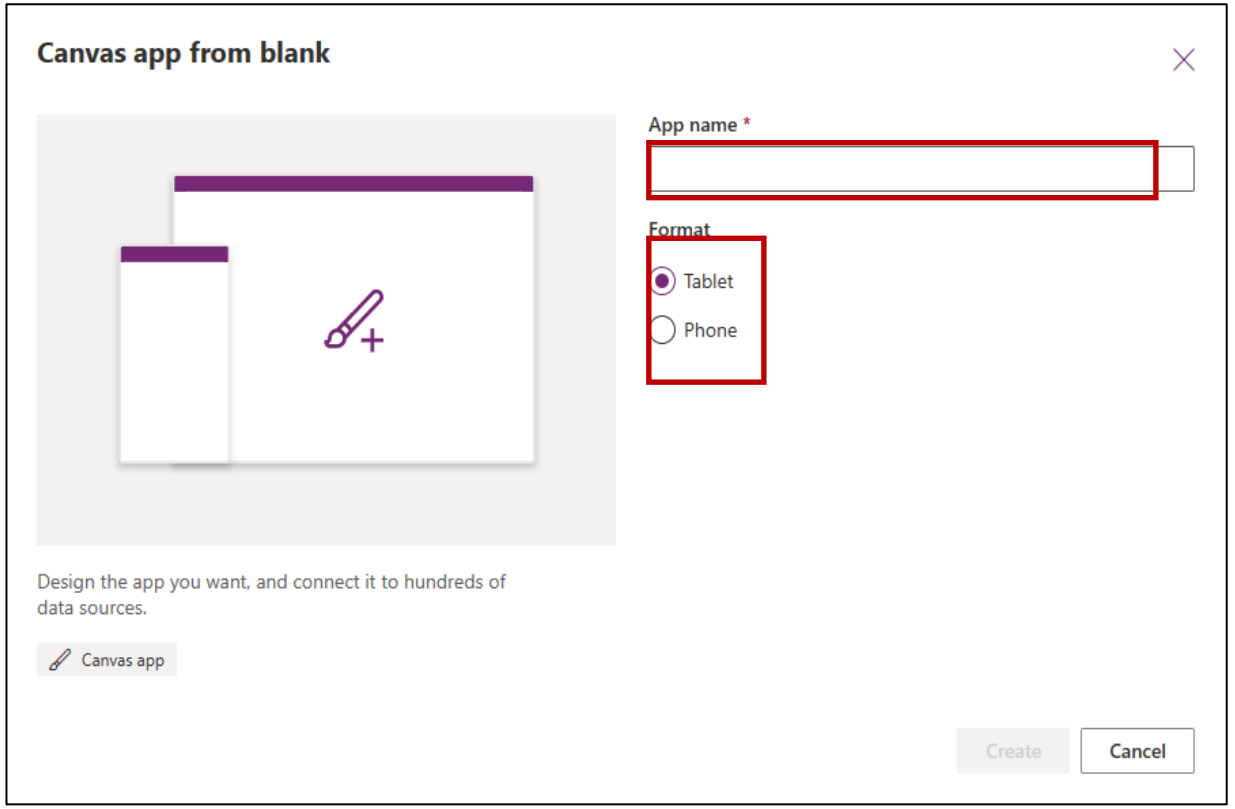
Step 30. Name you app and select format

Give your app a name and select the format of your app

Name: Your alias + WeatherForecast

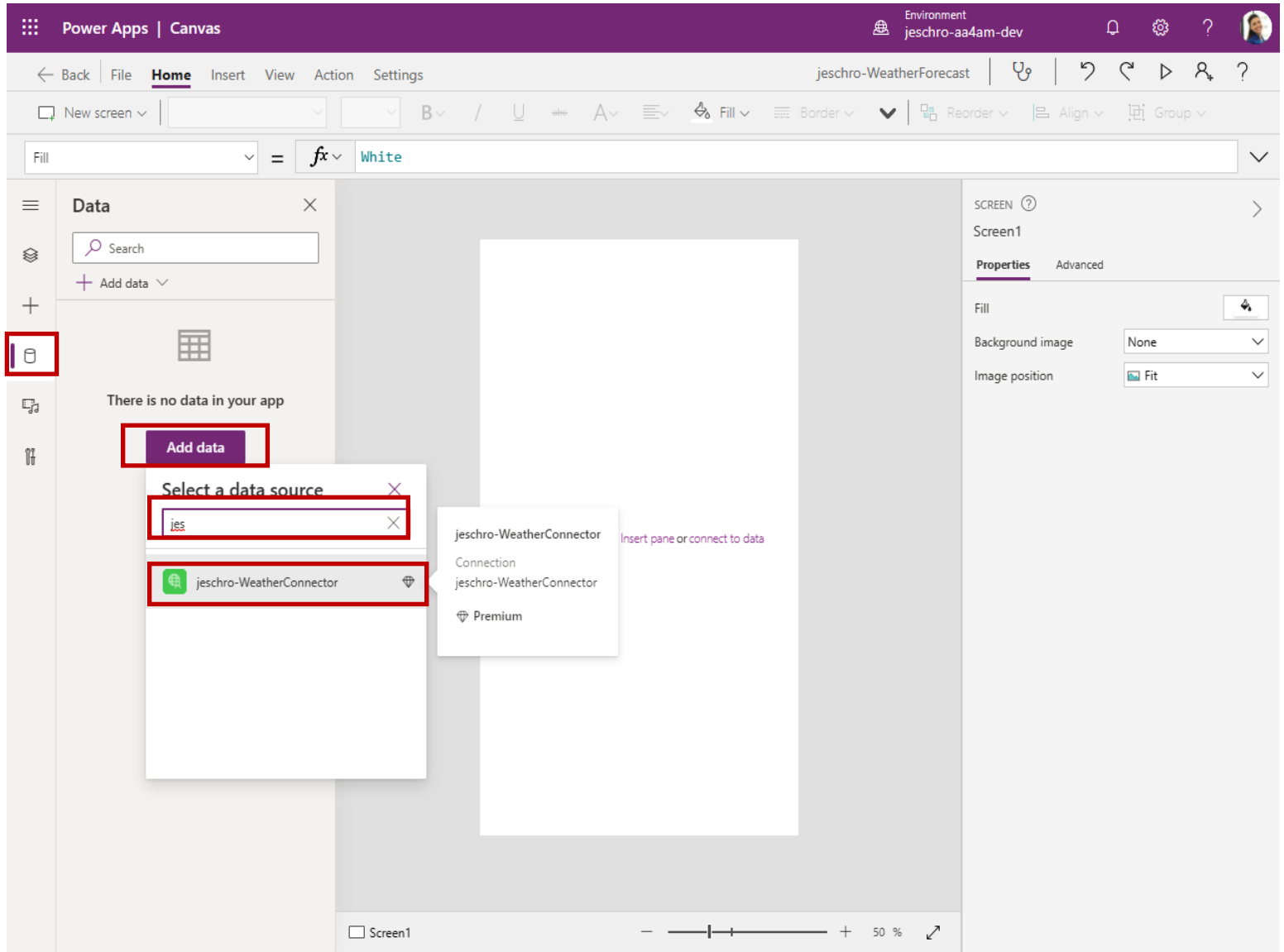
Format: Phone

Click **Create**



Step 31. Add your custom connector as a datasource

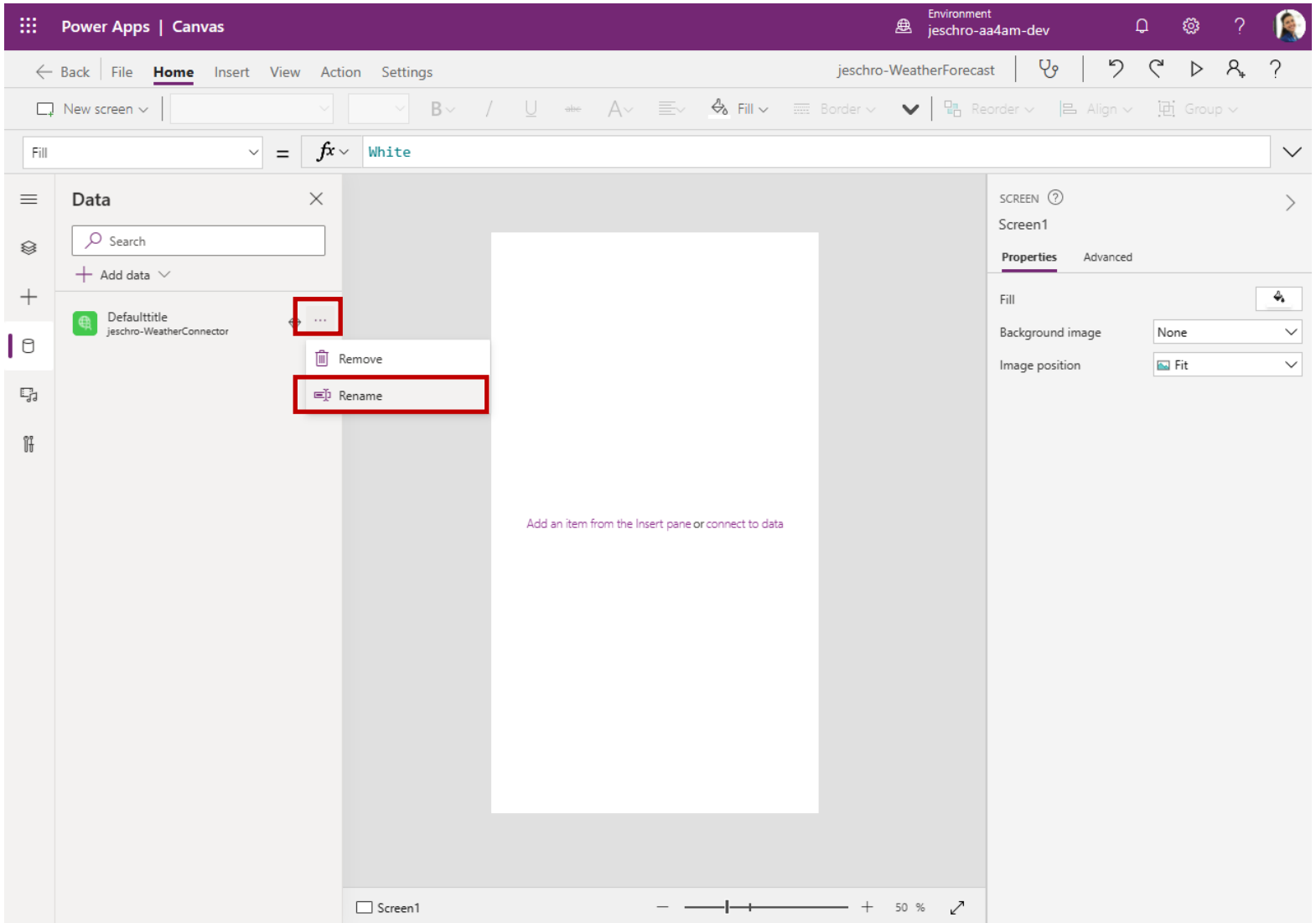
1. Click the database icon on the left navigation bar
2. Click **Add data**
3. Search for your alias to find your connector
4. Click your connector and click it again



Step 32. Rename the reference to your connector

Your connector is added with Defaulttitle as name.

Change this to WeatherConnector

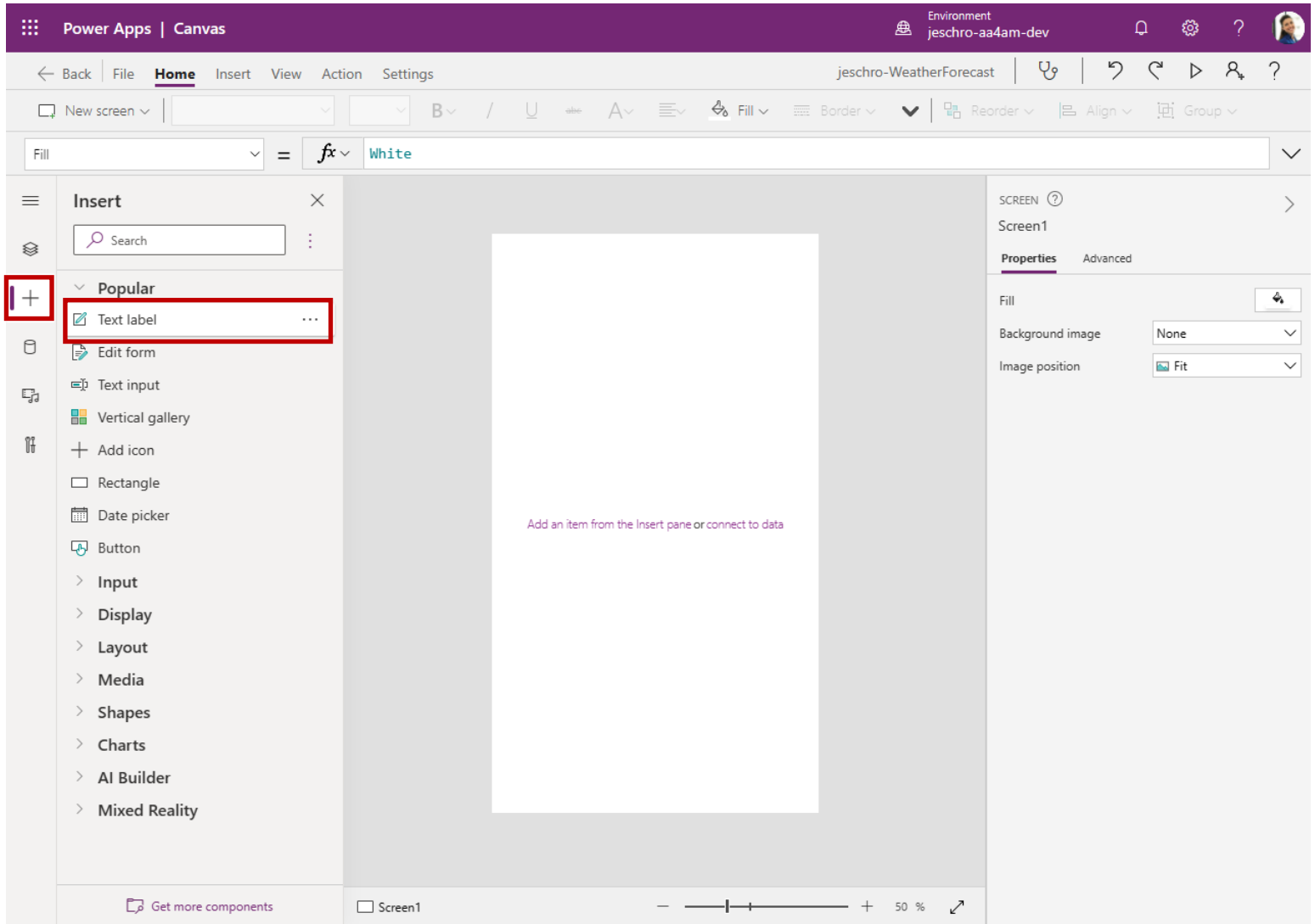


Step 33. Add controls to your app

Add a label to your app

Click the + icon in the left Navigation bar

Click Text label



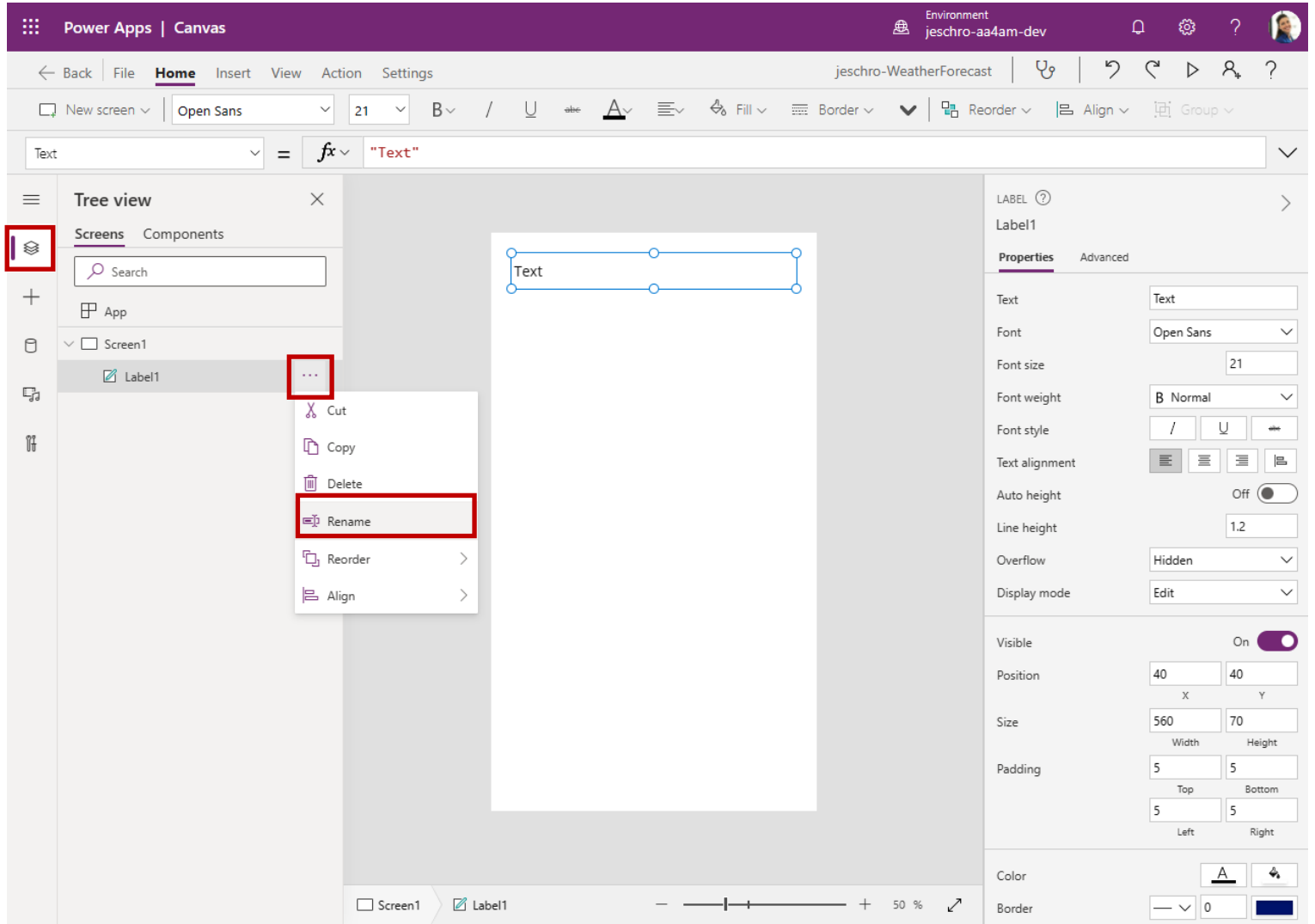
Step 34. Set the label control properties

It is best practice to use meaningful names for your controls

Click the Treeview button ( ) in the left Navigation bar.

Click the **…** next to your label control and click **Rename**

A good name could be **lblLocation**. This shows it is a label and refer to its purpose

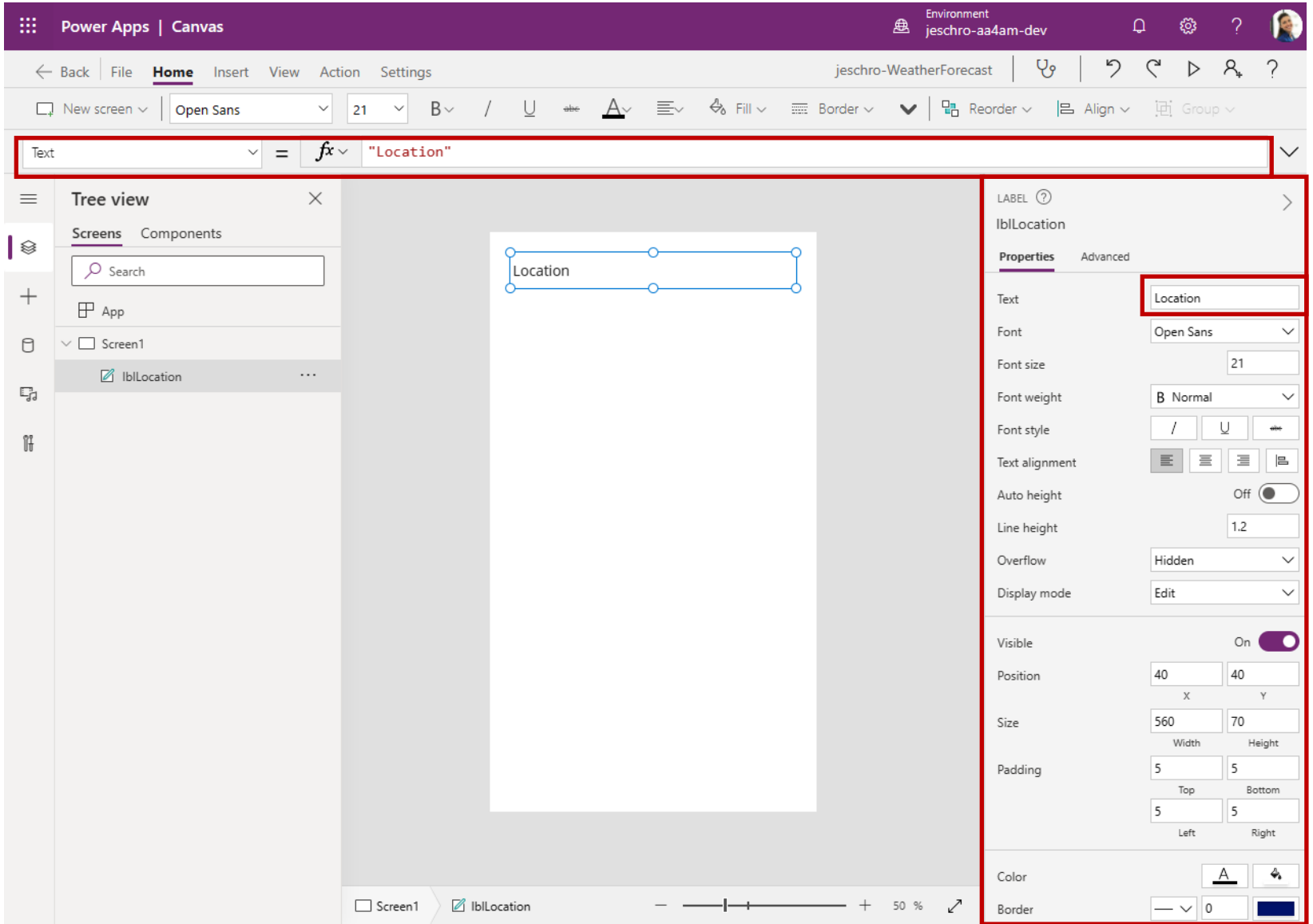


Step 35. Set the label control properties continued

Next set the Text property of the label.

You can do this in multiple ways. Most common are via the Properties pane to the right or the Properties dropdown in top left.

Set the Text property to **Location**



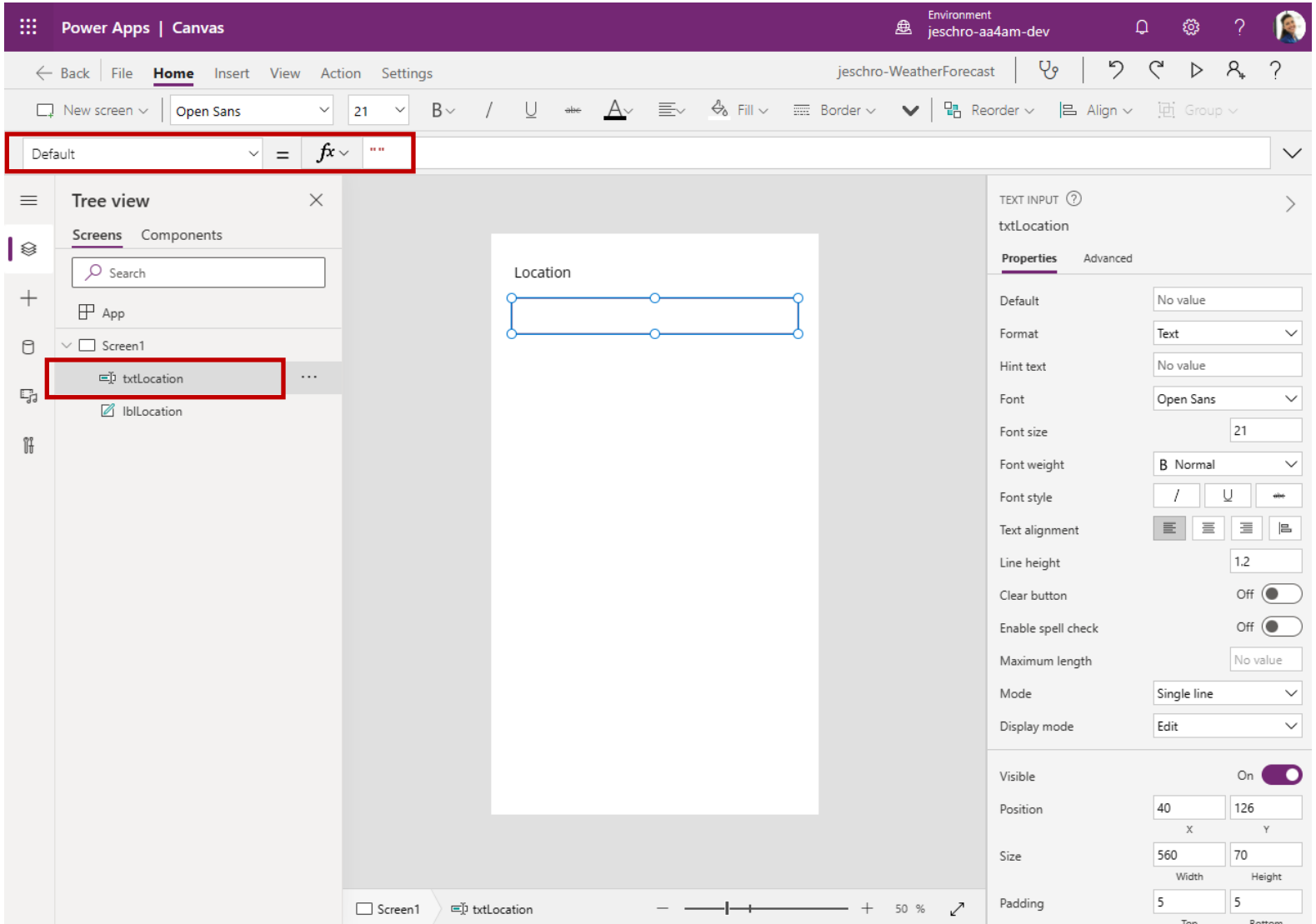
Step 36. Add Text input control to your app

Set the following properties for the Text Input control

Name: txtLocation

Default: ””

Drag the control to just below the lblLocation label on the screen



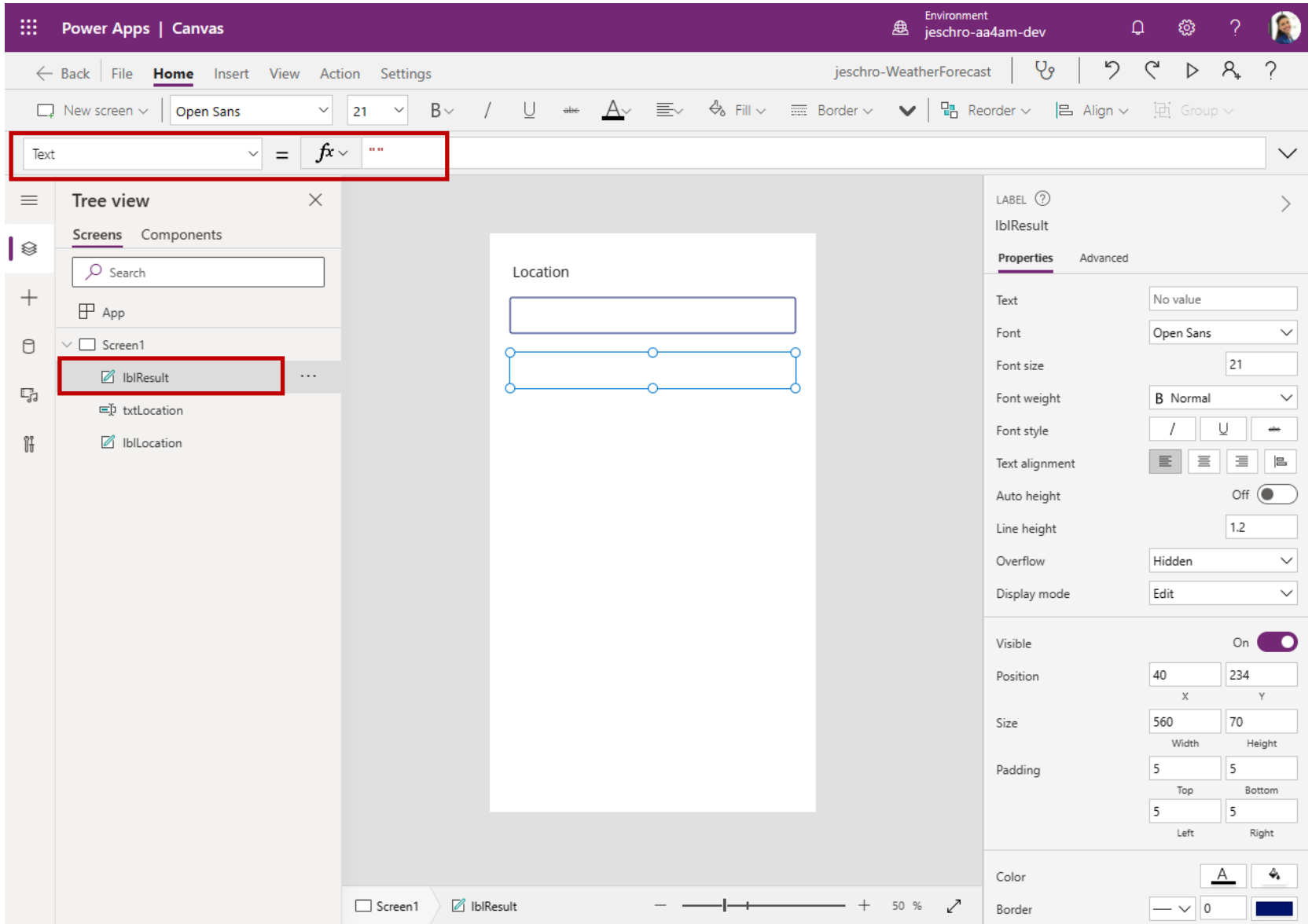
Step 37. Add label to display the forecast result

Set the following properties for the label control

Name: lblResult

Text: ””

Drag the control to just below the txtLocation input on the screen



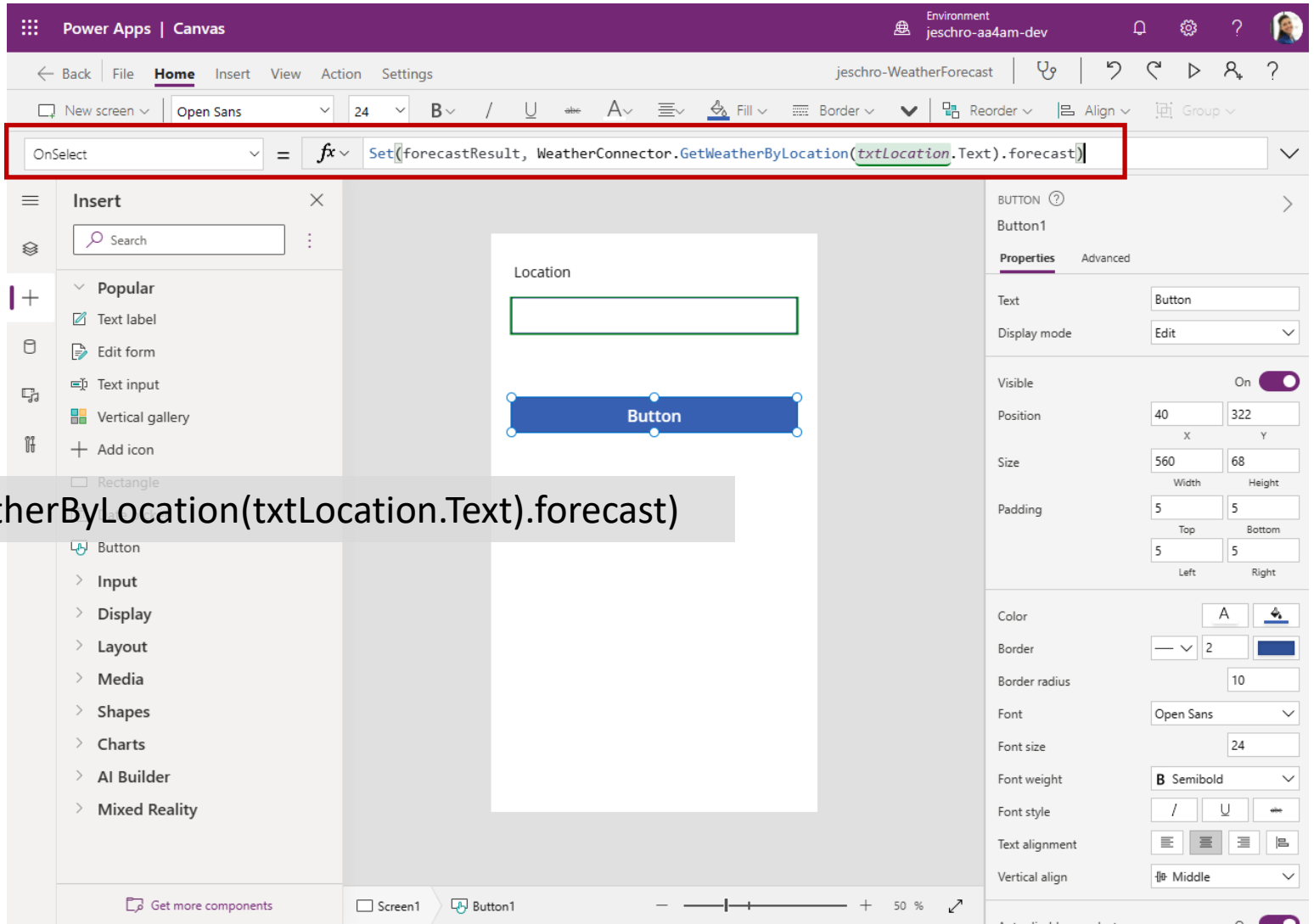
Step 38. Add a button to call our custom connector

Add a button and drag it below the lblResult label on the screen

Select the **OnSelect** property in the property dropdown and enter the following formula:

```
Set(forecastResult, WeatherConnector.GetWeatherByLocation(txtLocation.Text).forecast)
```

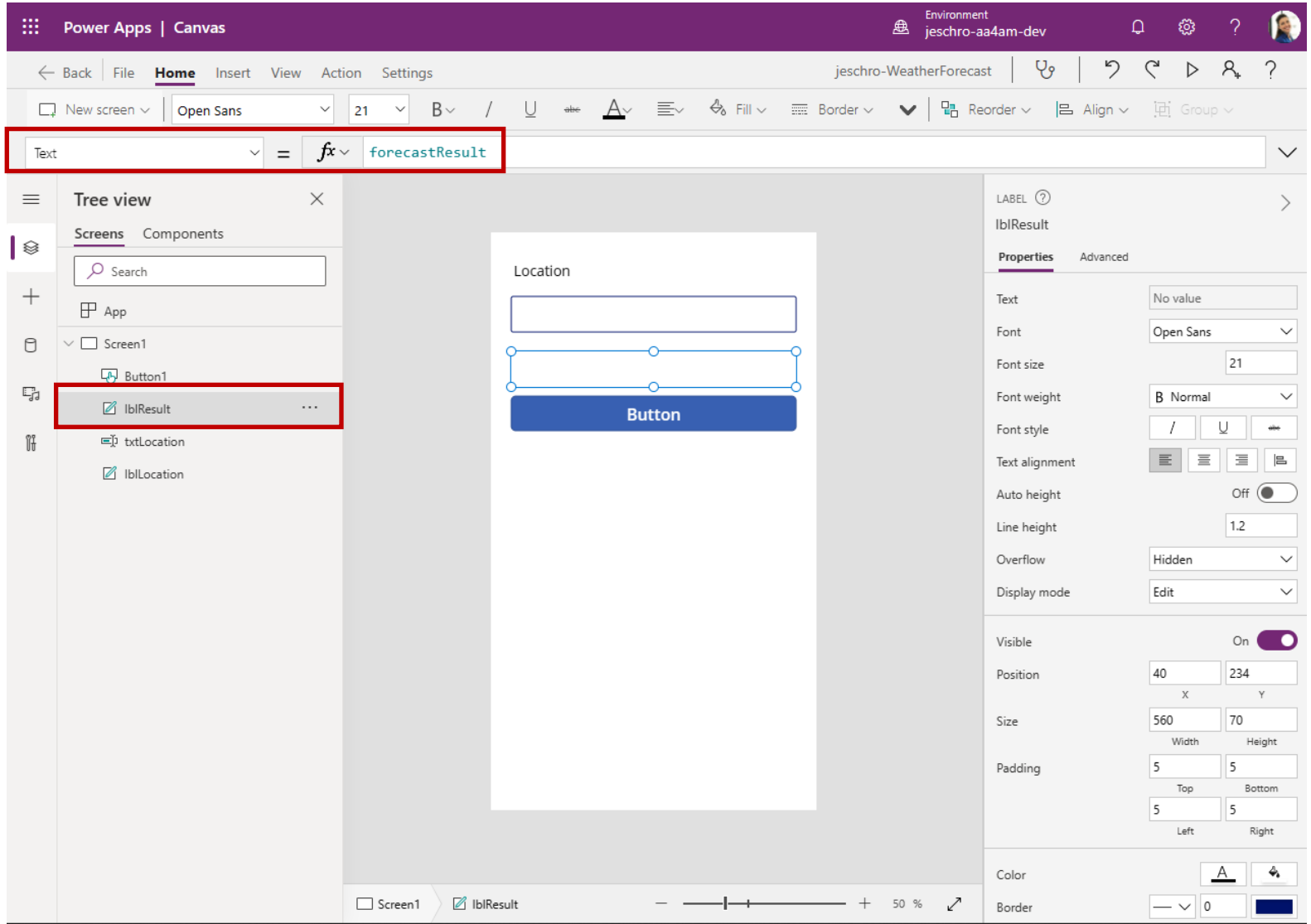
This will set a variable called forecastResult to the forecast property of the object returned by the WeatherConnector



Step 39. Display the result in the result label

Select the lblResult label in the Tree view and set its **Text** property to **forecastResult**

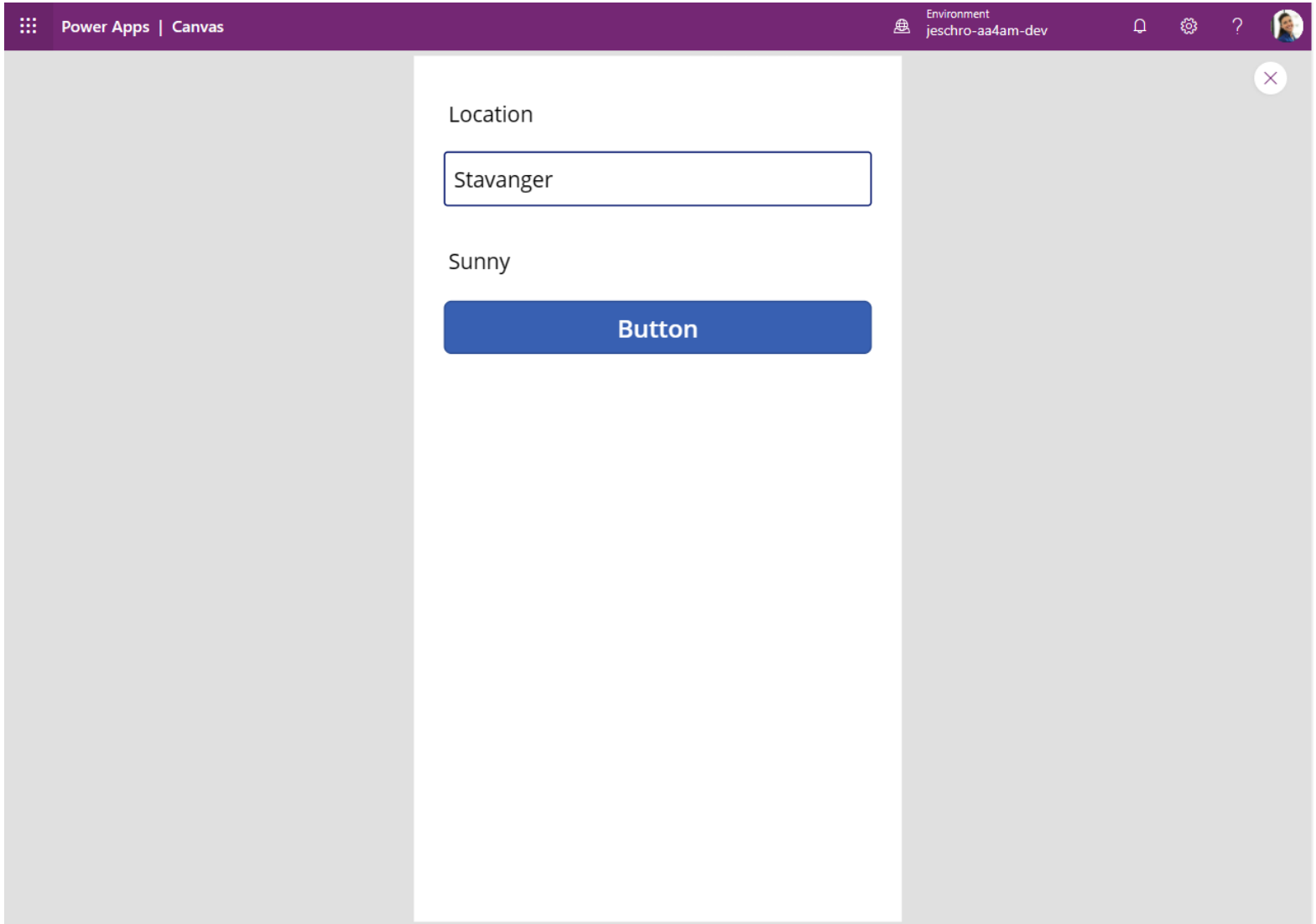
The label will show the value of the forecastResult variable



Step 40. Test your app

Play your app by clicking the play button (▷) in the top right corner or pressing F5

Enter a location and press the button



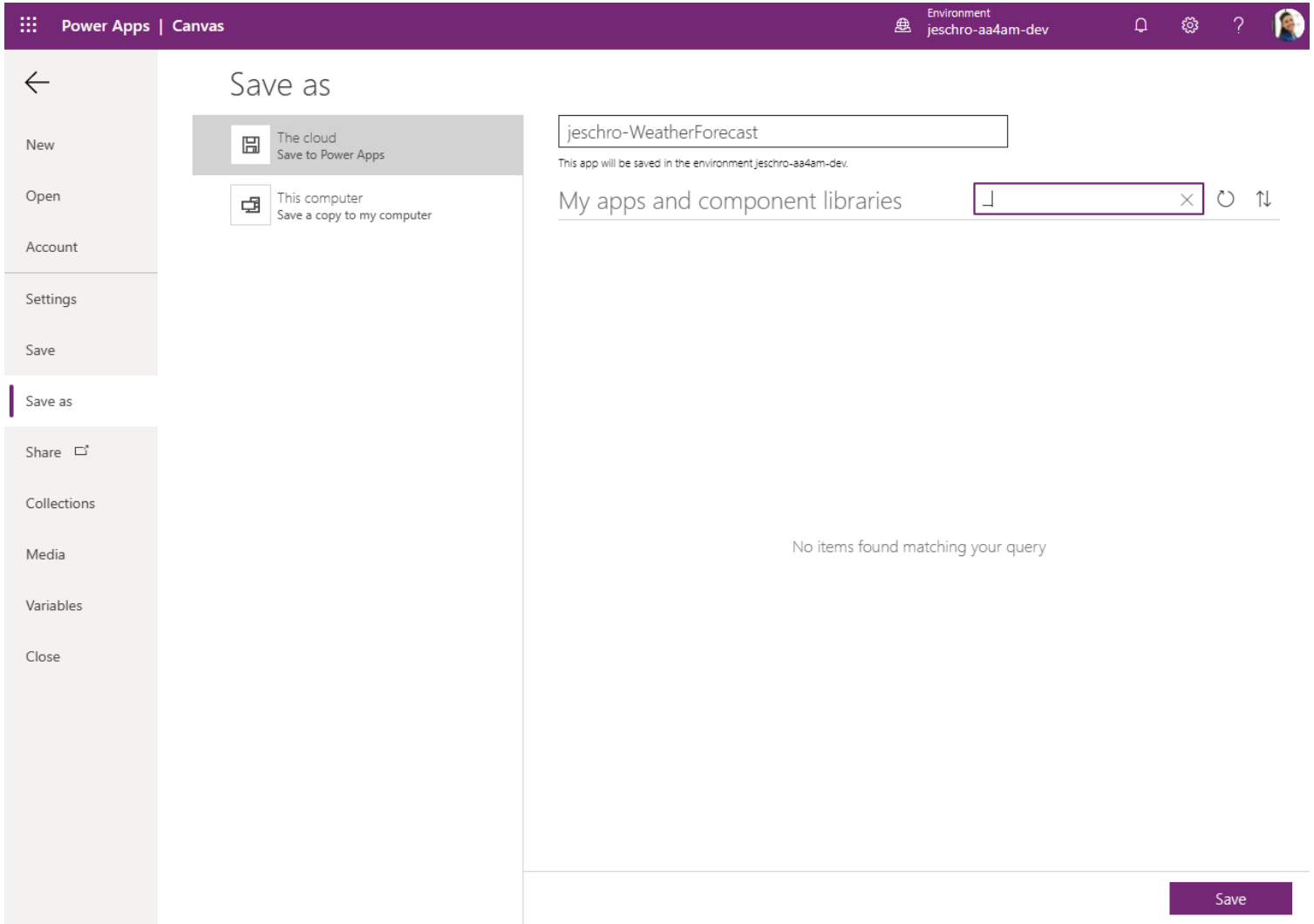
Step 41. Save your app

Close the app player in top right corner.

Click **File => Save**

to save your app

You app calling an Azure Function via custom connector is now ready to be used via mobile phone, browser, teams client or embedded in other web technologies.

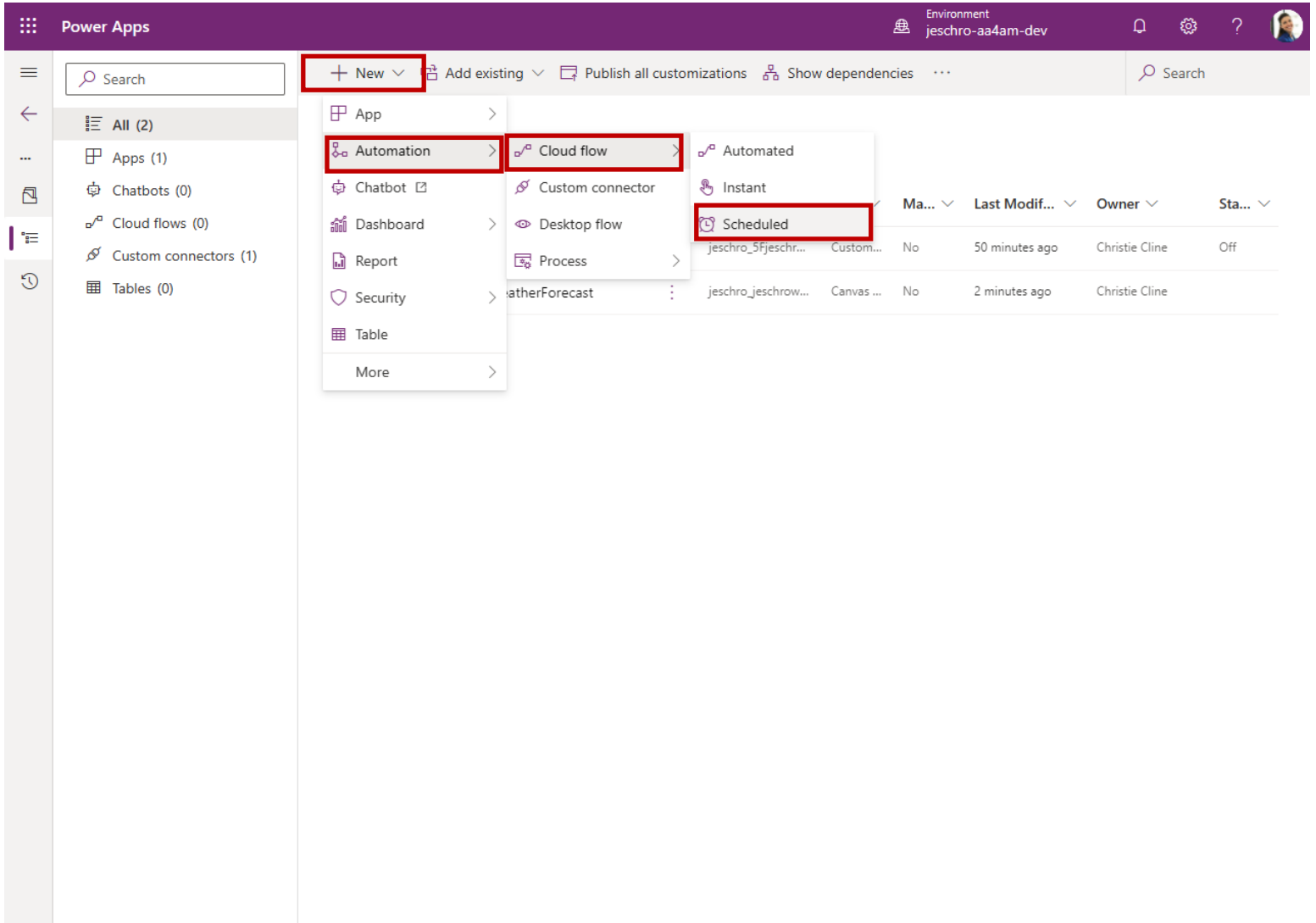


Using a Power Platform custom connector in a Power Automate Flow

Step 42. Add a Scheduled Cloud flow to your solution

Navigate to your solution

Click **+New => Cloud flow =>Scheduled**

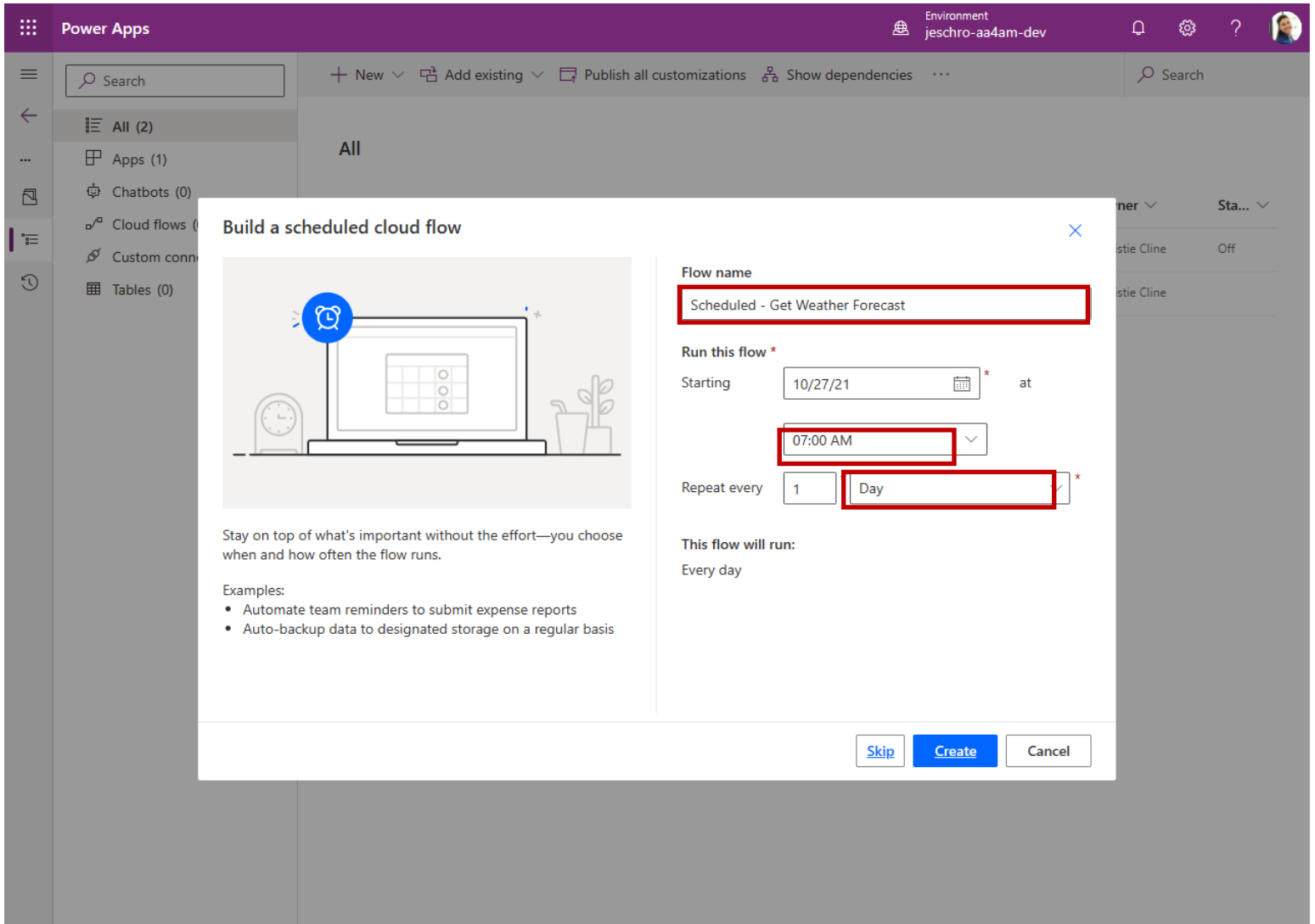


Step 43. Configure Schedule and create cloud flow

Create your cloud flow with the following properties:

- Flow name: Scheduled – Get Weather Forecast
- Starting: [today] at 7:00 AM
- Repeat every: 1 Day

Click **Create**



Step 44. Add a new step to your flow

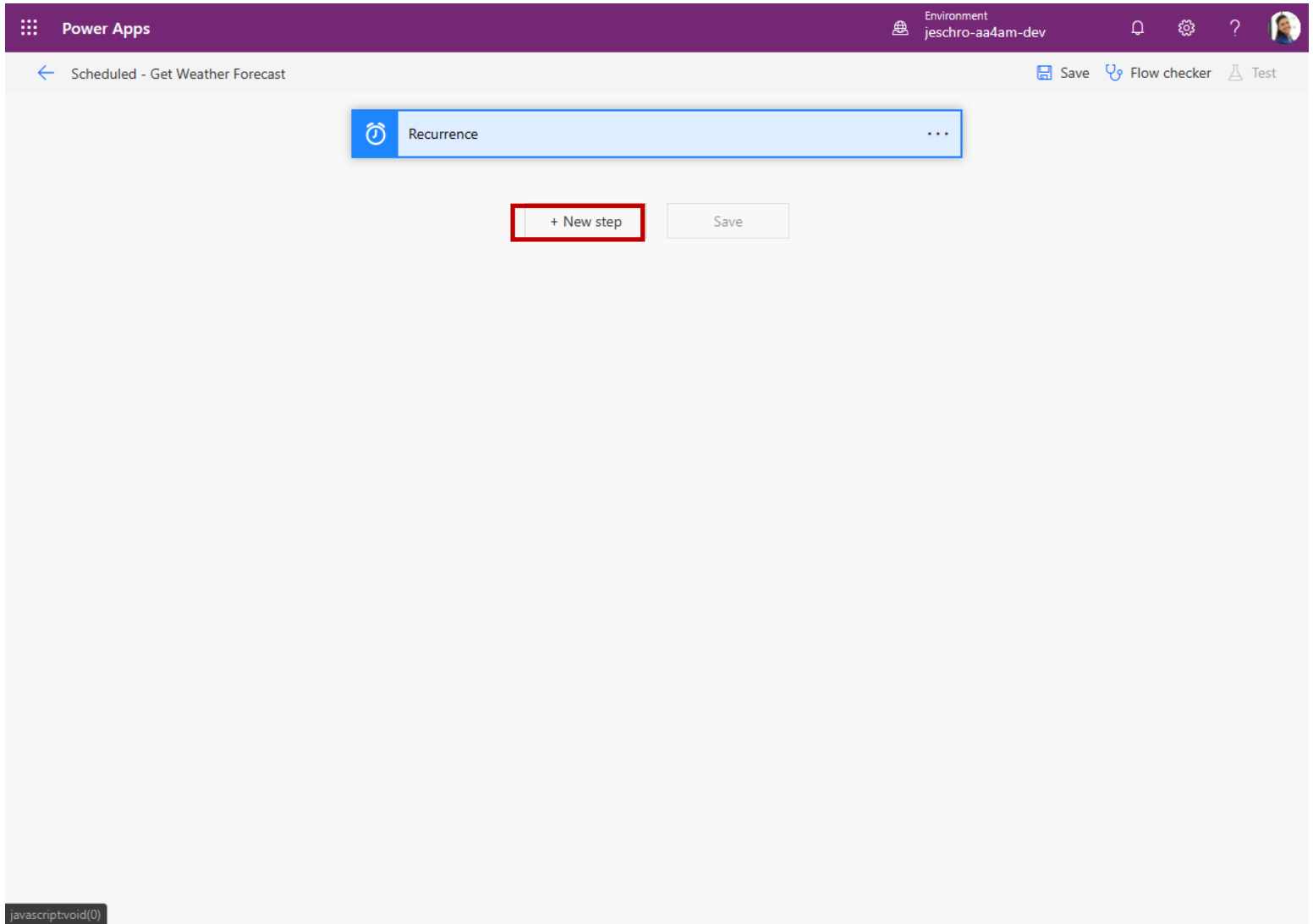
You flow has been created with only a trigger.

The trigger is what starts the flow.

As defined when the flow was created the trigger is a Recurring trigger with a schedule.

We now must add at least one action to the flow.

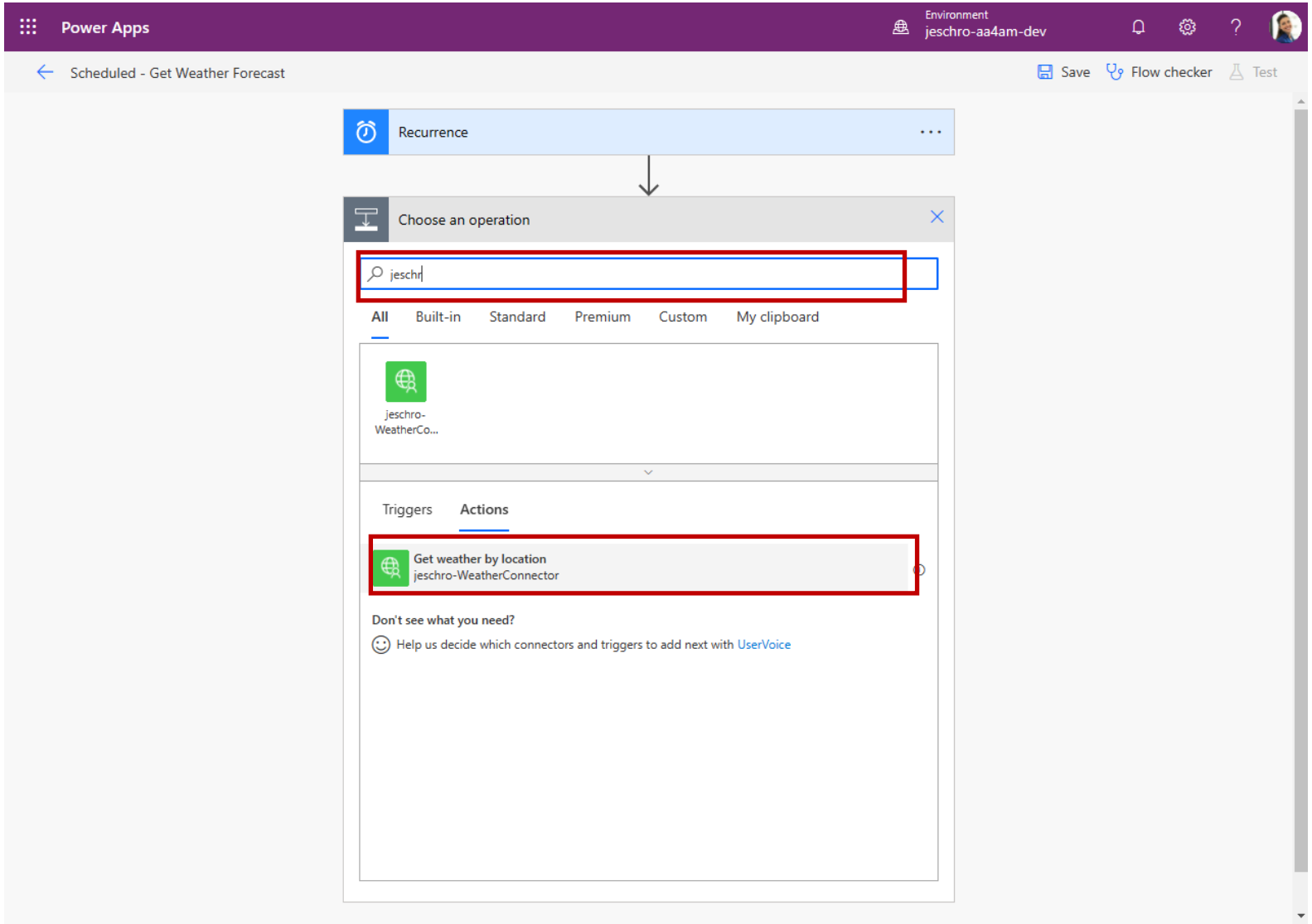
Click **+New step**



Step 45. Add the Get weather by location action

Search for your alias to find your custom connector

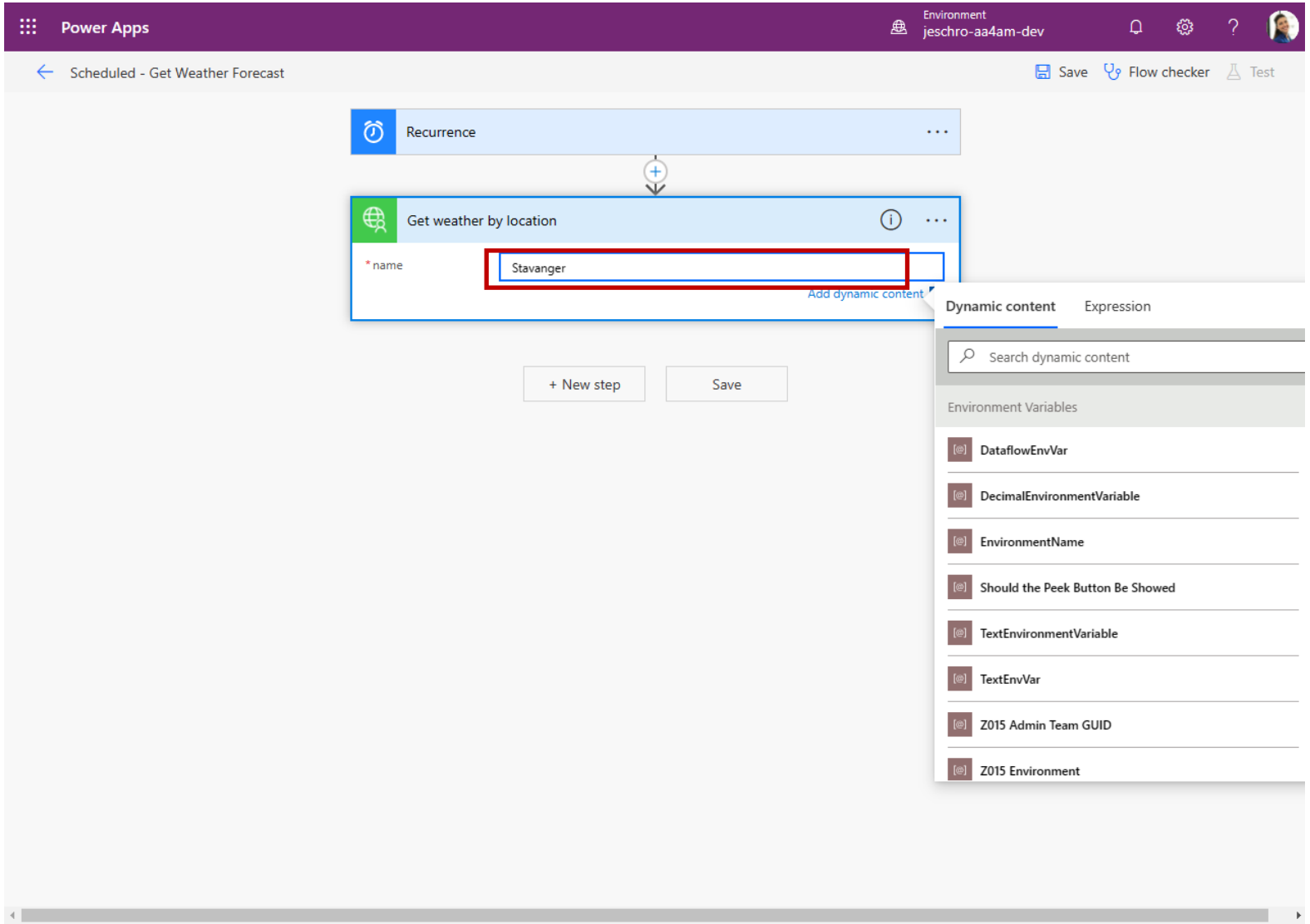
Click the **Get weather by location** action



Step 46. Configure the Get weather by location Action

For the purpose of the lab we will hardcode in a location value.

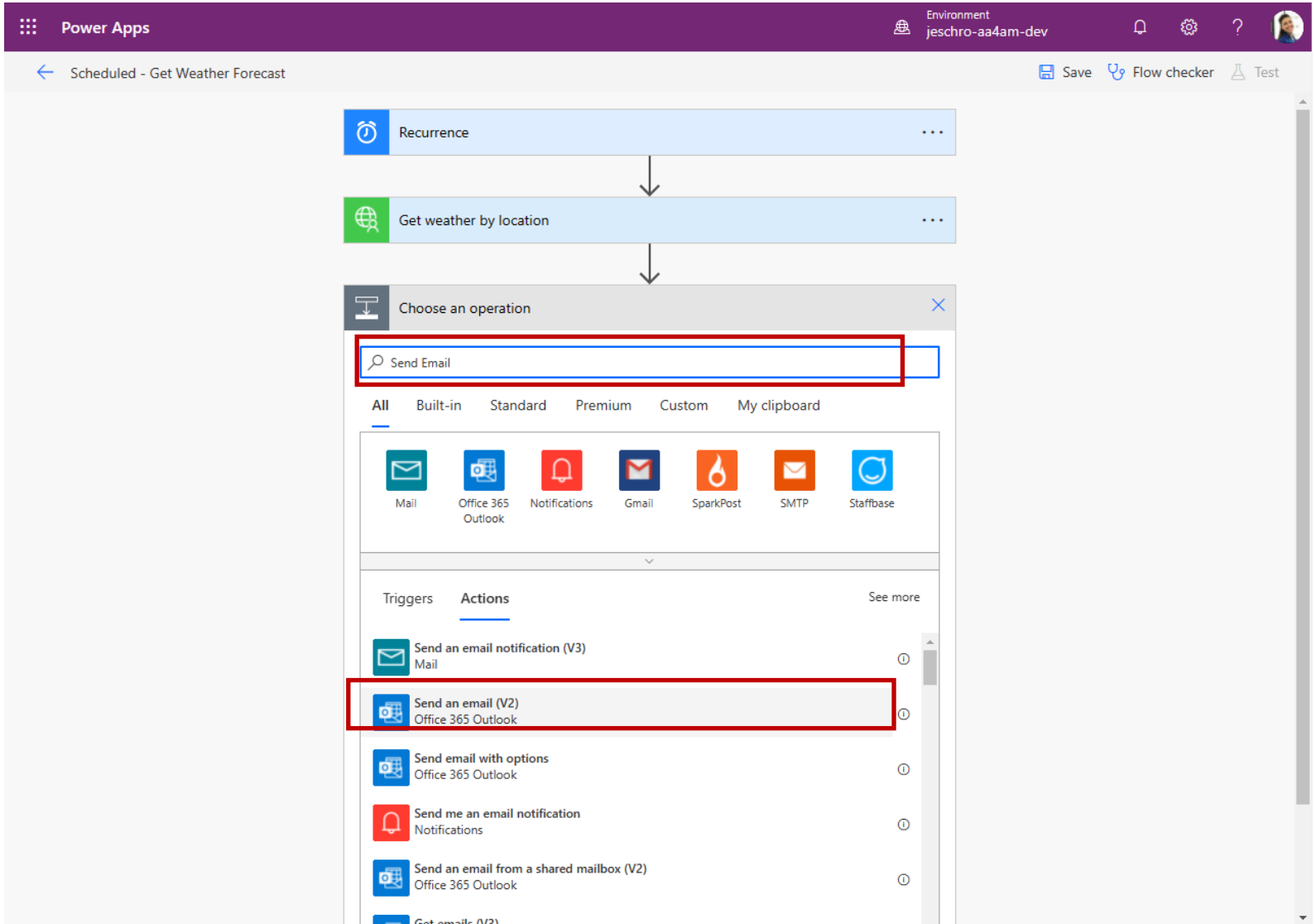
Enter a location (Bergen, Oslo, Stavanger) in the **name** property.



Step 47. Add the Send an email (v2) action

Lets complete the cloud flow by sending the forecast via email

Click **+ New step** and select the Send an email (v2) action



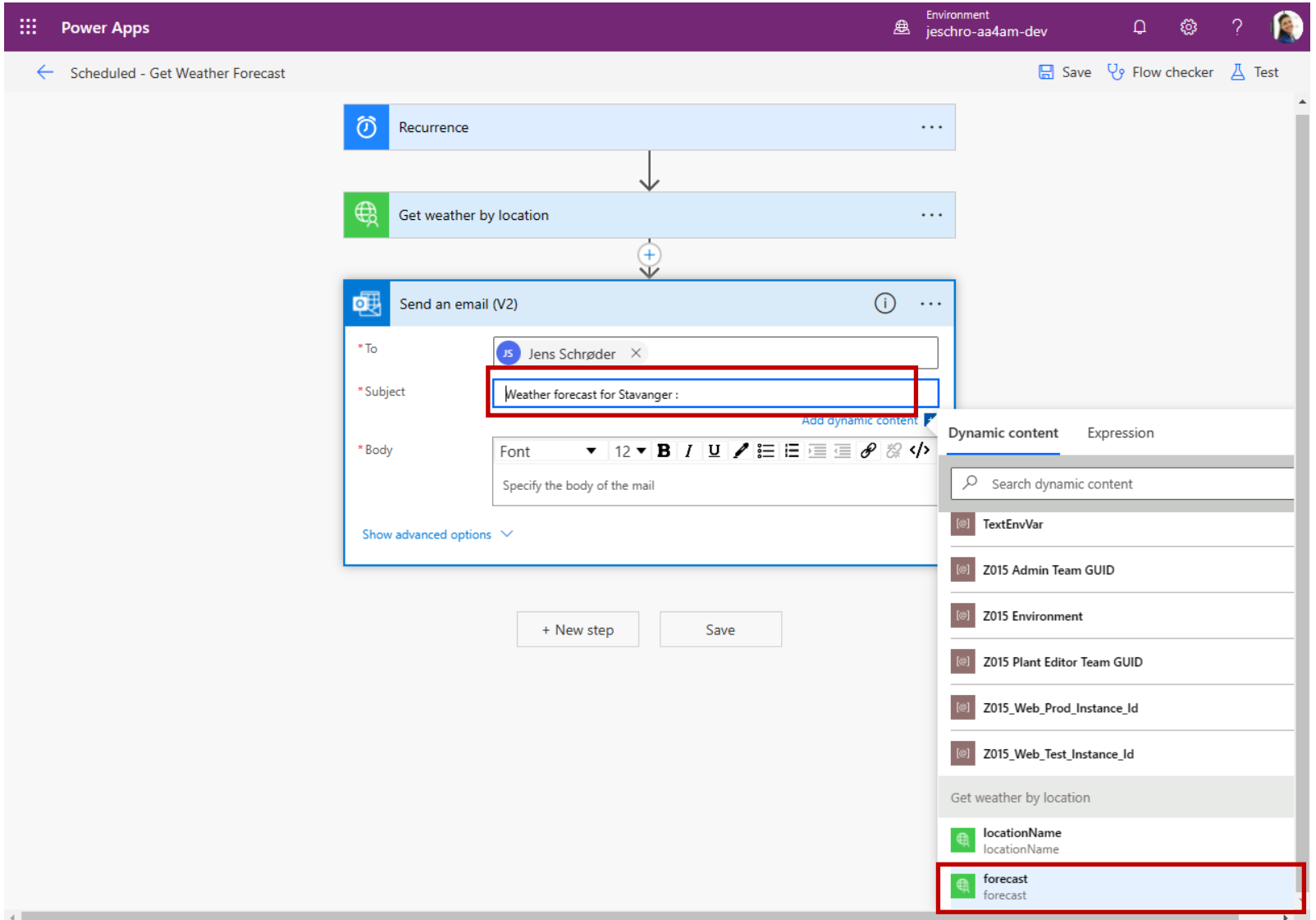
Step 48. Configure the Send an Email (v2) action

Configure the following properties of the Send an email (v2) action

- To: your email address
- Subject : ”Weather forecast for Stavanger : ”

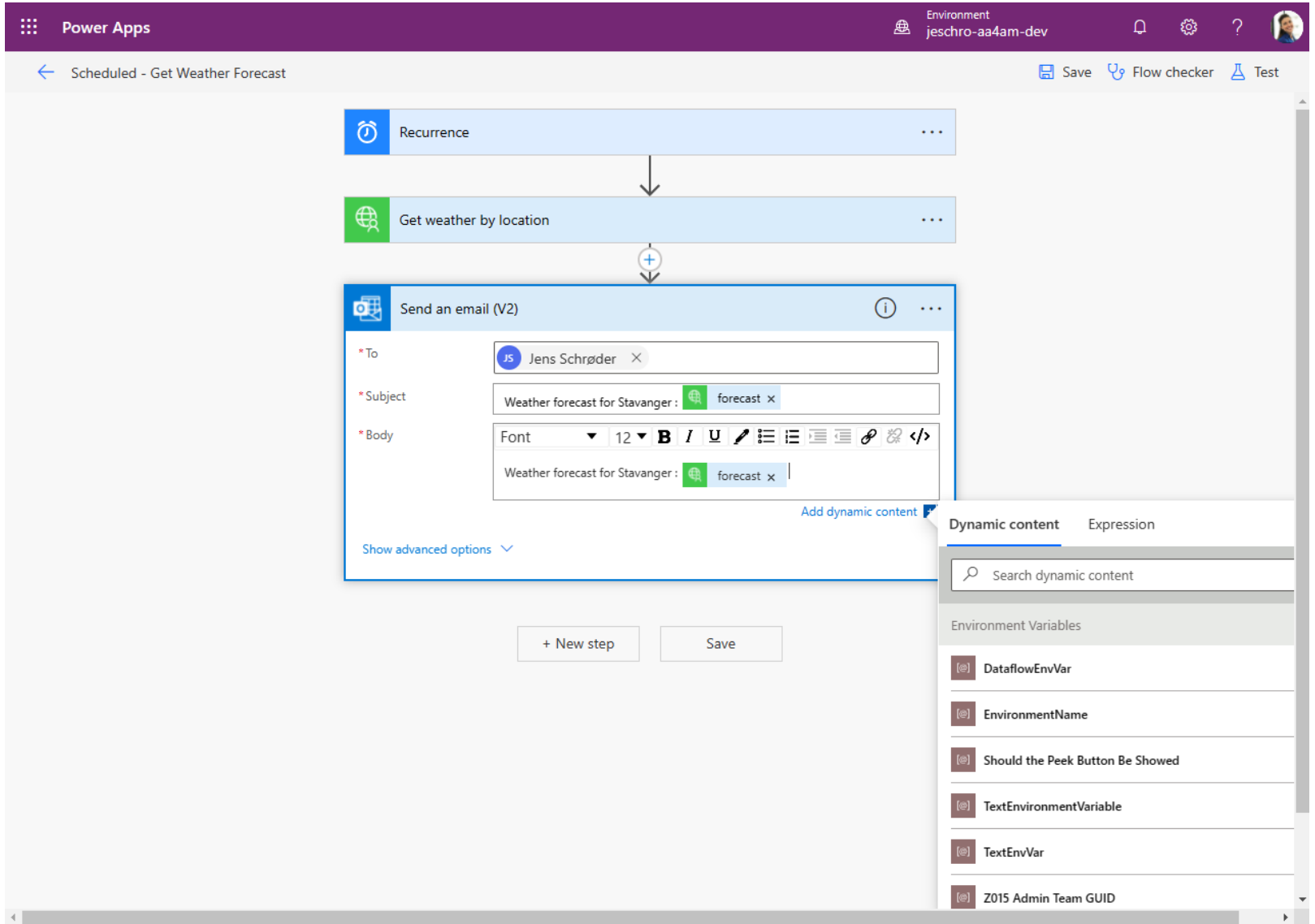
While in the Subject property scroll down in the Dynamics content pop up and click **forecast**.

This will add a reference to forecast property in the output of the Get weather by location action



Step 49. Configure the Send an Email (v2) action continued

Copy and paste the content from the Subject property to the Body property

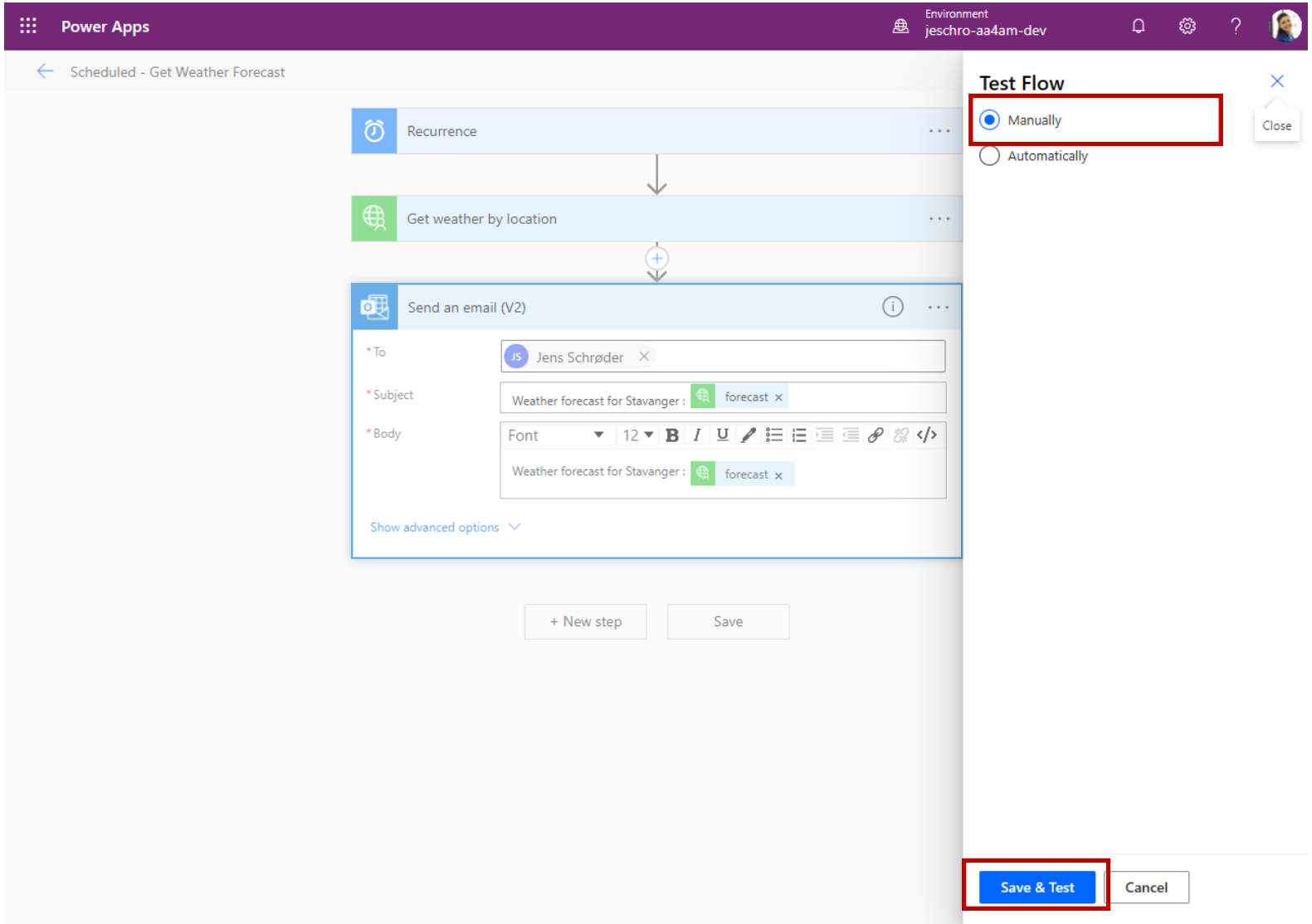


Step 50. Save and Test

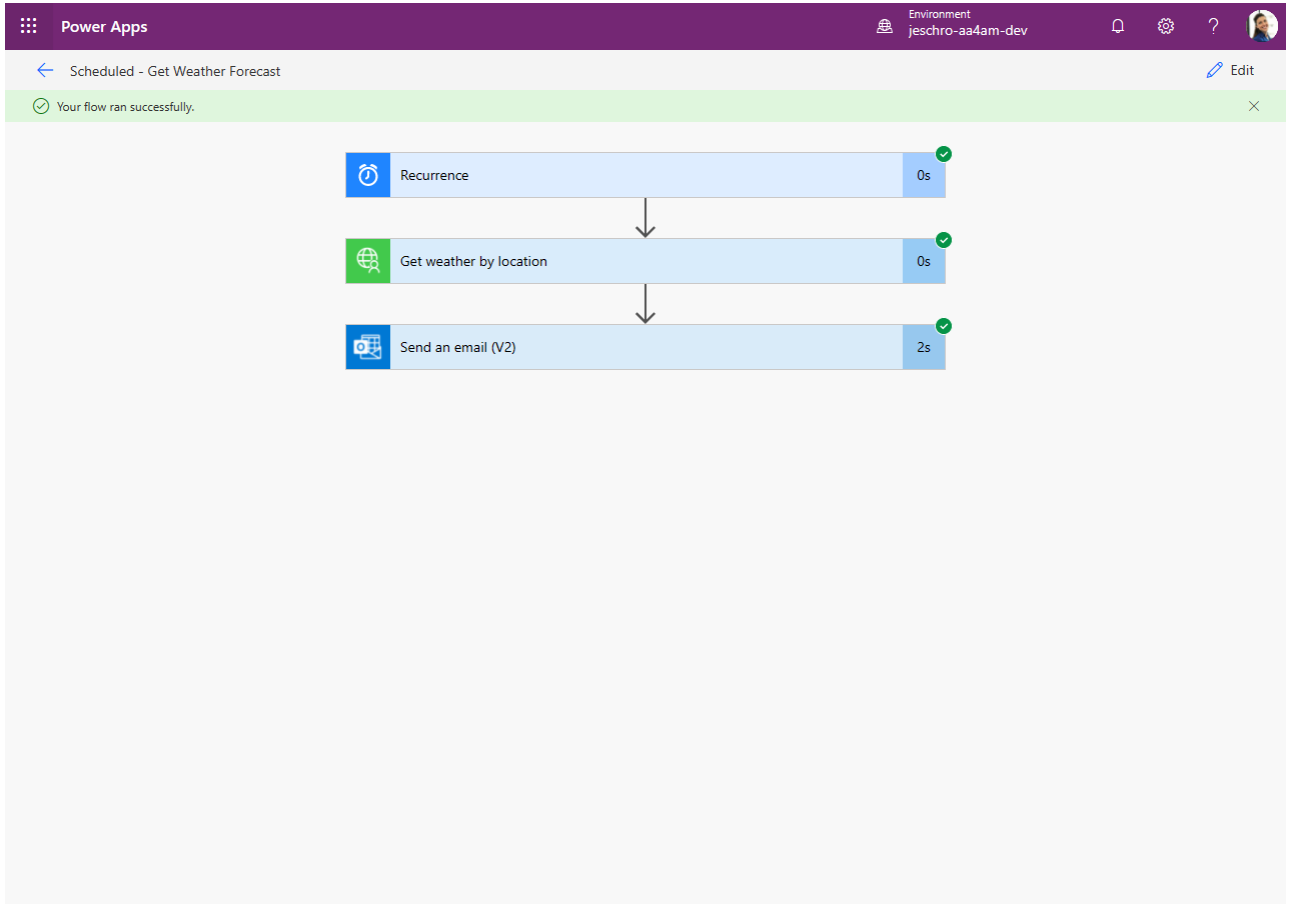
Click **Test** in the top right corner to test the cloud flow.

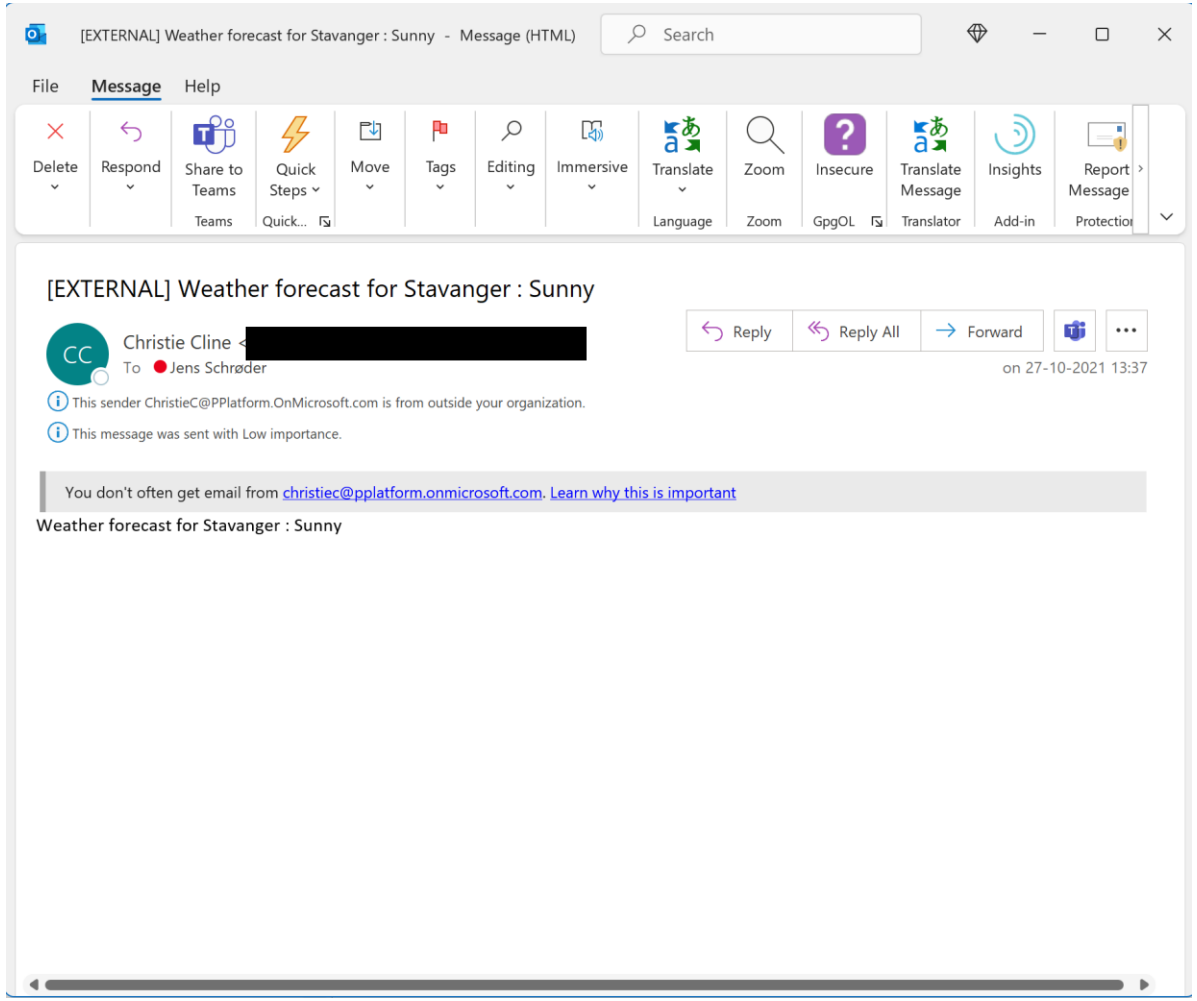
Select **Manually** and click **Save & Test**

Then click **Run flow**



Step 51. Verify the run is successful





Create a custom connector for Azure AD protected Azure Functions | Microsoft Docs

How To: Use Swagger in Azure Functions - Cloudkasten