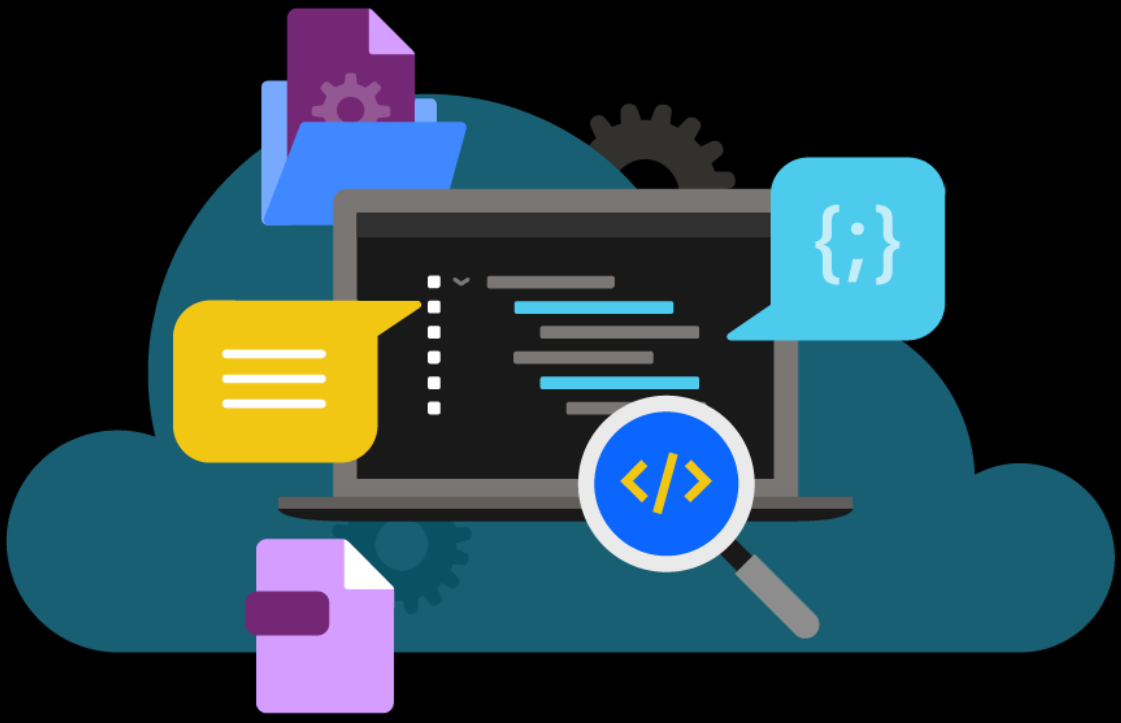
Microsoft

Fusion development approach to building apps using Power Apps

Bring your pro devs and business together to build apps fast



Shayne Boyer

Version 1.0

THE FUSION DEVELOPMENT APPROACH TO BUILDING POWER APPS:

USE POWER APPS TO BRING YOUR BUSINESS AND PRO DEVS TOGETHER TO BUILD APPS FAST

EDITION v1.0

PUBLISHED BY

Microsoft Developer Relations, and Power Apps product teams

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Authors:

Shayne Boyer – Principal Developer Advocate - Microsoft

John Sharp – Principal Technologist – Content Master

Alistair Matthews – Principal Technologist – Content Master

Phil Stollery – Principal Technologist – Content Master

Editors and Reviewers:

Greg Hurlman - Sr. Software Engineer - Microsoft

Matt Soucoup - Sr. Developer Advocate – Microsoft

April Dunnam - Sr. Developer Advocate – Microsoft

TABLE OF CONTENTS

Microsoft Power Apps is a great way to build business apps quickly. Power Apps Studio enables a *citizen developer*—who may be more familiar with understanding and solving business problems than the technical nuances associated with writing code—to be actively involved in creating business solutions. The low code tooling allows a non-technical user to quickly flesh out the design of an app and specify how it should function. Power Apps supports connectors that can integrate an app with a wide range of data sources and services. The citizen developer can work with a *professional developer* who implements connectors to these services. In turn, the services implement the more intricate parts of the system that require data access and complex processing. The citizen developer can then *plug* these connectors into the app. The result is an accelerated process for designing, building, and deploying business applications.

The purpose of this guide is to summarize the way in which citizen and professional developers can work together, following a *fusion development approach*. As you progress through this guide, you'll play the role of the different participants in this process to build a complex, fully functional solution that combines Power Apps with Azure services.

PREQUISITES AND SETUP

To perform the steps described in this guide, you require the following licenses and software:

- A Git client, such as Git for Windows or the Git command line tools, available at https://git-scm.com.
- A **Power Apps** account. If your organization has an Office 365 account, you may be able to create a free Power Apps account. See the Microsoft Power Apps page at https://powerapps.microsoft.com/ to get started.
- An **Azure subscription**. The apps built by the steps in this guide create and use resources in Azure. If you don't have a subscription, you can sign up on the Microsoft Azure page at https://azure.microsoft.com.
- The **Azure CLI**, available at https://aka.ms/AAbvfzf.
- **.NET 5.0**, available on the **Download .NET** page at https://aka.ms/AAbvu77.
- **Visual Studio Code**, available at https://aka.ms/AAbvu79. You'll also need the following extensions for Visual Studio Code:
    - The **C# extension for Visual Studio Code**. This extension is available in the Visual Studio Code Marketplace at https://aka.ms/AAbvu72.

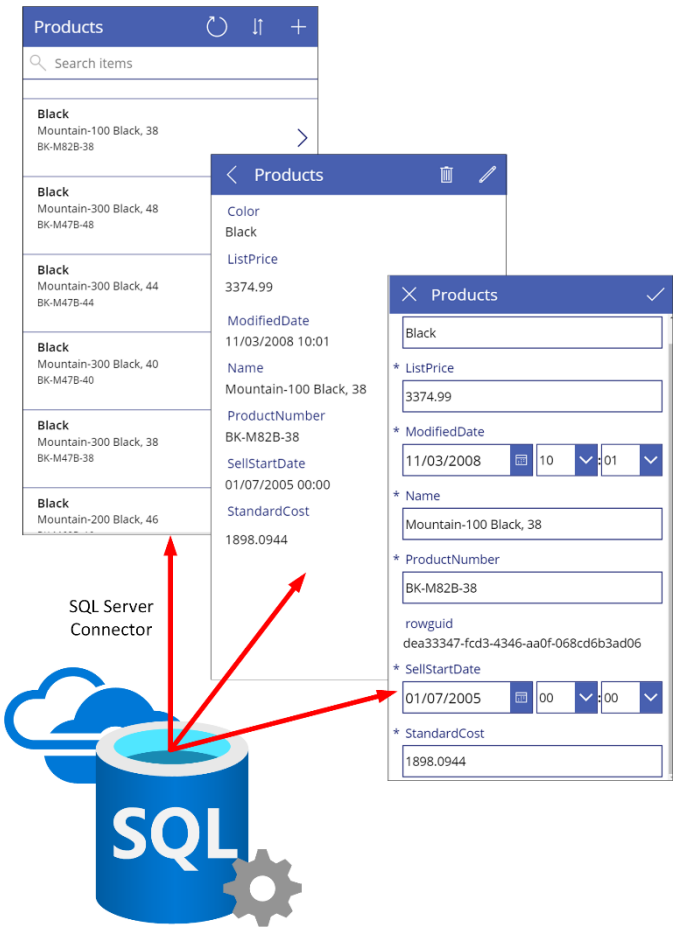
- The **Azure App Service** extension, at https://aka.ms/AAbvgm8.
- The **Azure Account** extension, at https://aka.ms/AAbvgm7.
- The **SQL Server extension**, at https://aka.ms/AAbvgm5.

The complete code for the Web API and the app is available in GitHub at https://github.com/microsoft/fusion-dev-ebook

Clone this repository locally on your computer and read the README.md file carefully. Before continuing with this chapter, make sure you have created the Azure SQL Database Server and databases required by the app using the instructions in the README.md file.

CHAPTER 1: WHAT IS THE FUSION DEVELOPMENT APPROACH?

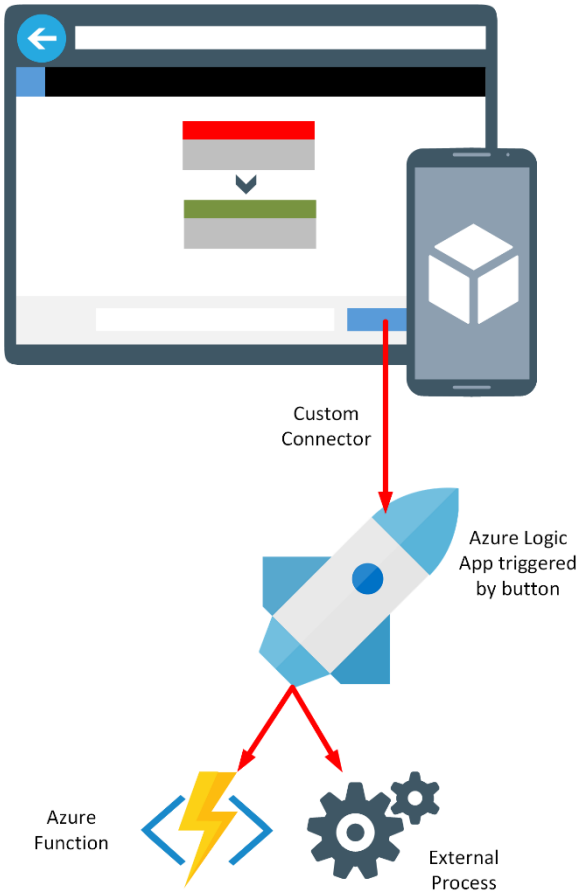
Effective application development depends on accurate and efficient communication of business requirements, and ideas for addressing these requirements. Many software engineering strategies promote the concept of the end users of applications being highly involved throughout the lifetime of the development process. However, there is frequently a *glass wall* between the end users and the software creators. Both parties can see each other and talk through their perspectives of how a new system should work, but the terminology spoken by one party might be different from that understood by the other. The need to translate language and ideas into a grammar that all members involved in the development process can agree on is fundamental to success. Additionally, in a rapidly changing business environment, time is of the essence. Failure to be agile enough to exploit a narrow window of opportunity can be costly. By building Microsoft Power Apps you can create and deploy working solutions that meet users' needs very quickly.



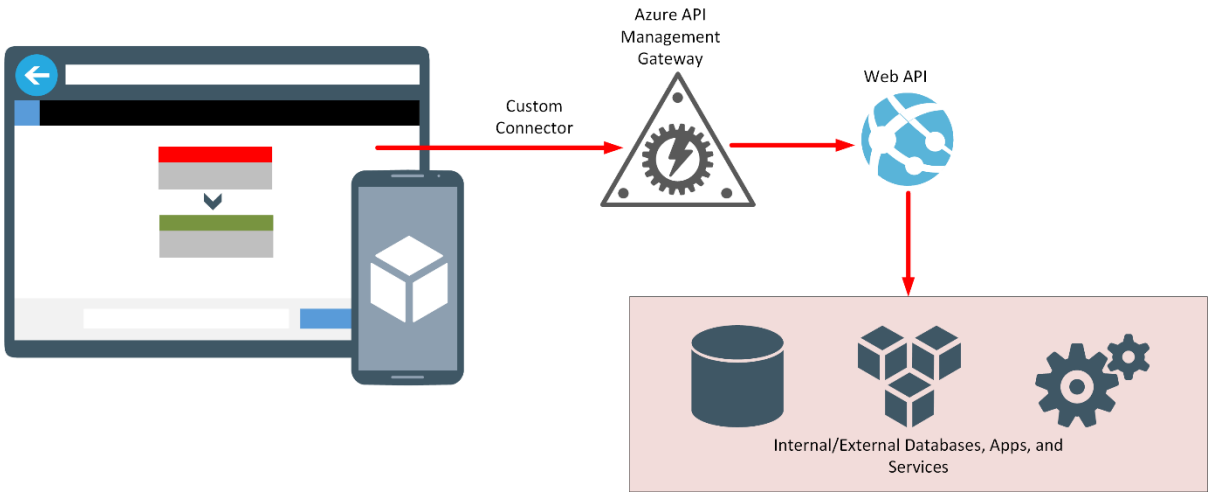
Power Apps enables a business user to quickly innovate and experiment with ways to improve their business processes. Using Power Apps, *citizen developers* who understand the business requirements can quickly put together the basics of a solution, with the minimum of coding effort. A citizen developer uses the graphical tooling provided by Power Apps Studio to create the business user's interface to a new system and some elemental logic that describes the functionality—typically involving data entry forms, displays, and reports. It's relatively easy to generate a working app from the data connectors that are supplied with Power Apps. These connectors enable the user interface to connect to many data sources, such as Microsoft SQL Server, Microsoft SharePoint, Oracle, Microsoft Excel, Twitter, Microsoft Dynamics, and several hundred others.

Note: You can find a full list of connectors on the **Connector Reference Overview** page at https://aka.ms/AAbvfzh

Many Power Apps built like this may fulfill an immediate business need quickly and cheaply, but there will always be situations where more complexity is involved that cannot be satisfied in this way. For example, your organization might have existing systems and databases with which the app needs to interact, and for which no connector is currently available. There may be additional business logic that needs to be enforced to ensure that data remains consistent. An app might need to implement a complex, dynamic business flow. This is where *professional developers* come into play. After a citizen developer has produced the front-end prototype for a system, the professional developer can work with them to create any appropriate custom connectors that they may require. A custom connector doesn't just provide a path to a data source; a professional developer can create custom connectors that give access to other services, such as Azure Logic Apps, which in turn can invoke Azure Functions. Connectors such as these enable the citizen developer to incorporate complex business logic into their apps without requiring that they understand how it's implemented.



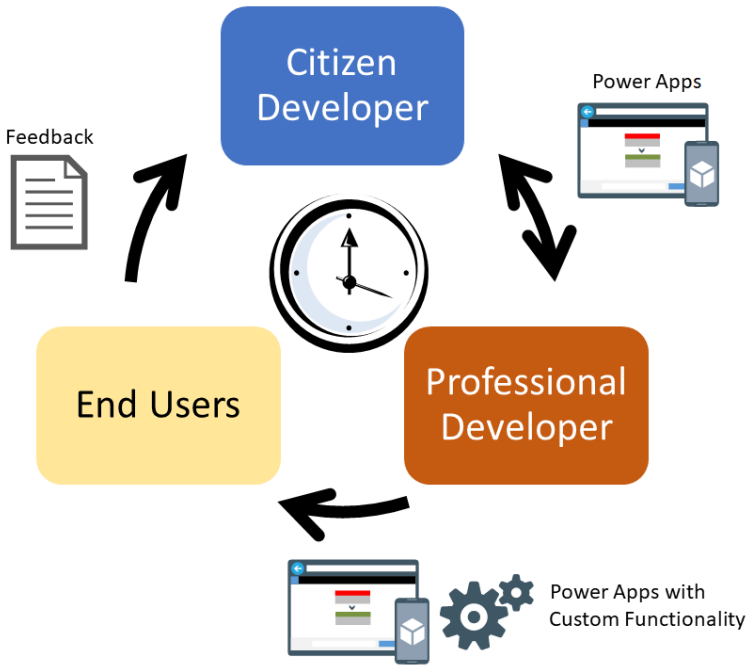
A common use case for a custom connector is to enable an app to access other systems and services inside and outside of an organization. A professional developer can create a Web API that wraps the operations exposed by these systems and services, host the Web API as an Azure Web App, and then make this web app available to a custom connector through Azure API Management (APIM).



Note: Other parts of the Microsoft Power Platform can benefit from a similar approach. For example, a citizen developer might implement business logic in a low code manner through Microsoft Power Automate and Robotic Process Automation, then utilize Web APIs to integrate other services into this logic. You can also build chatbots with Microsoft Power Virtual Agents that combine AI capabilities with data and services exposed through Web APIs.

Fusion Development with Power Apps is about combining the worlds of the citizen developer, the professional developer, and the other parties instrumental in building and using applications to further the objectives of the business. A citizen developer can express the business need quickly by building an app, and work with a professional developer to *fill in the gaps*. End users can provide feedback on missing functionality and any changes required. The whole process is highly iterative, perhaps more so than many other Agile processes, with the velocity of possibly several iterations a day.

Note: Gartner describes *digital fusion teams* as "distributed and multidisciplinary digital business teams that blend technology and other types of domain expertise. At least 84% of companies and 59% of government entities have fusion teams." (Source: 2019 Gartner Digital Business Teams Survey)



Note: For a more detailed introduction to the Fusion Development process, and how it can accelerate development times, read **Citizen developers use Microsoft Power Apps to build an intelligent launch assistant** at https://aka.ms/AAbvfzj

CHAPTER 2: INTRODUCTION TO THE SAMPLE SCENARIO

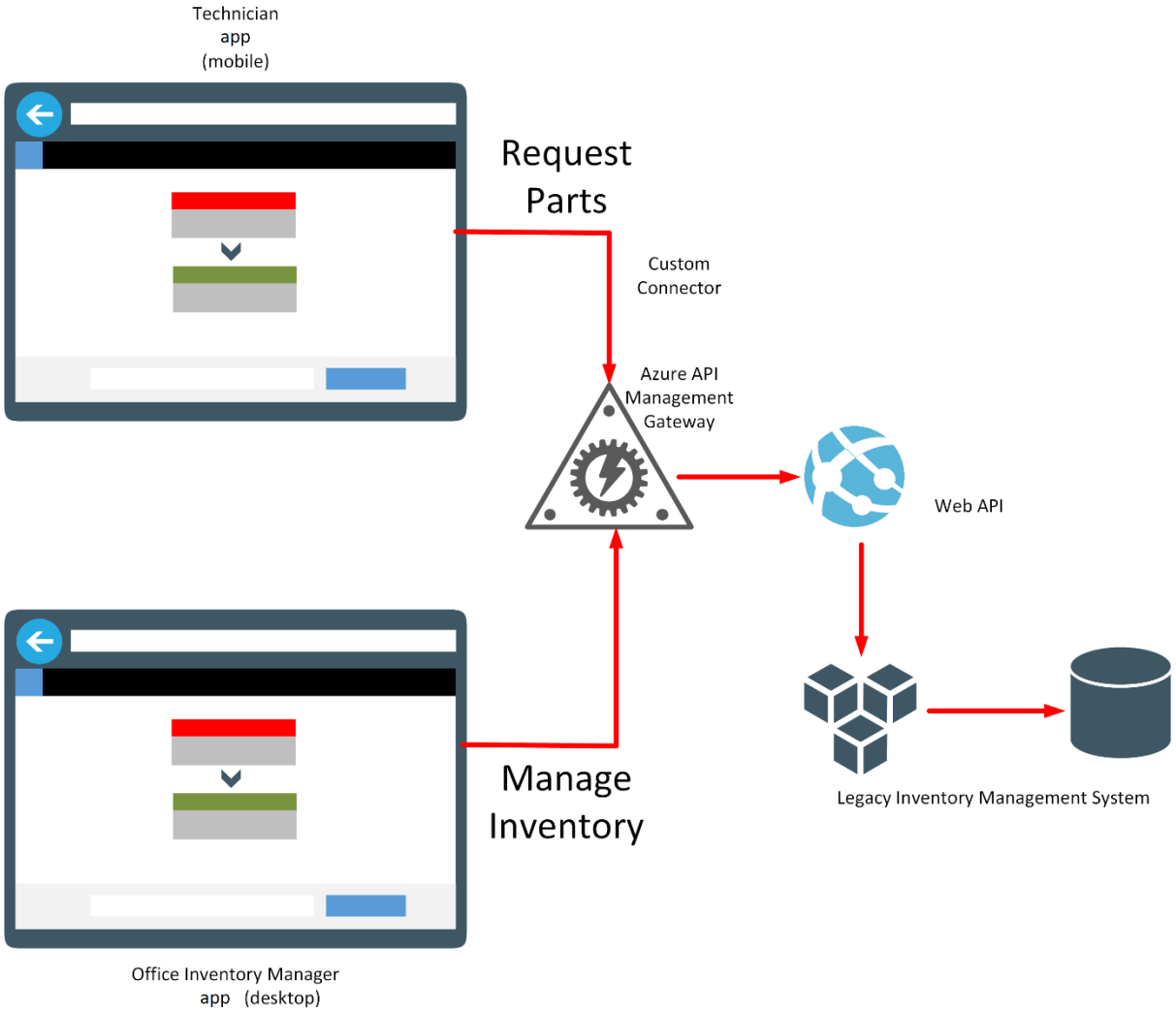
VanArsdel Heating and Air Conditioning is the world's leading company in furnace and air conditioning installation and repair. The business sends field technicians to customers' houses to install and repair all brands of heating and air conditioning equipment. The business has grown exponentially over the past year. While still a small company, VanArsdel relied a lot on manual and paper-intensive work processes. However, as the company has grown, it has experienced some friction in scaling the core business applications, as outlined by the use cases described in the following sections:

FIELD INVENTORY MANAGEMENT

When a technician arrives at a customer's home and finds they don't have a part on the truck needed for the repair, they sometimes travel back to the shop and grab it from the warehouse. They fill out a piece of paper stating the part has been removed. If the part is not stocked, the technician requests that it's needed. An office inventory manager then spends part of their day placing orders with a legacy system making sure the warehouse is appropriately stocked. This pattern of work results in the following inefficiencies:

- The technician must make a round trip to grab a needed part. That could result in a wasted trip if the part isn't in stock.
- The office inventory manager needs to check a spreadsheet of parts in stock several times a day to order new supplies.
- Because mistakes do happen, the office inventory manager must audit the spreadsheet against the inventory.

A solution is to create an app that allows the field technician to check inventory from the field and place an order immediately if that's necessary. The app will interface with a Web API running in Azure, that provides controlled access to the legacy inventory management system. The office inventory manager can connect to the same legacy system through a desktop app running on-premises. The desktop app enables the office inventory manager to see what parts are currently in stock, and when to place orders to replenish areas that are running low.

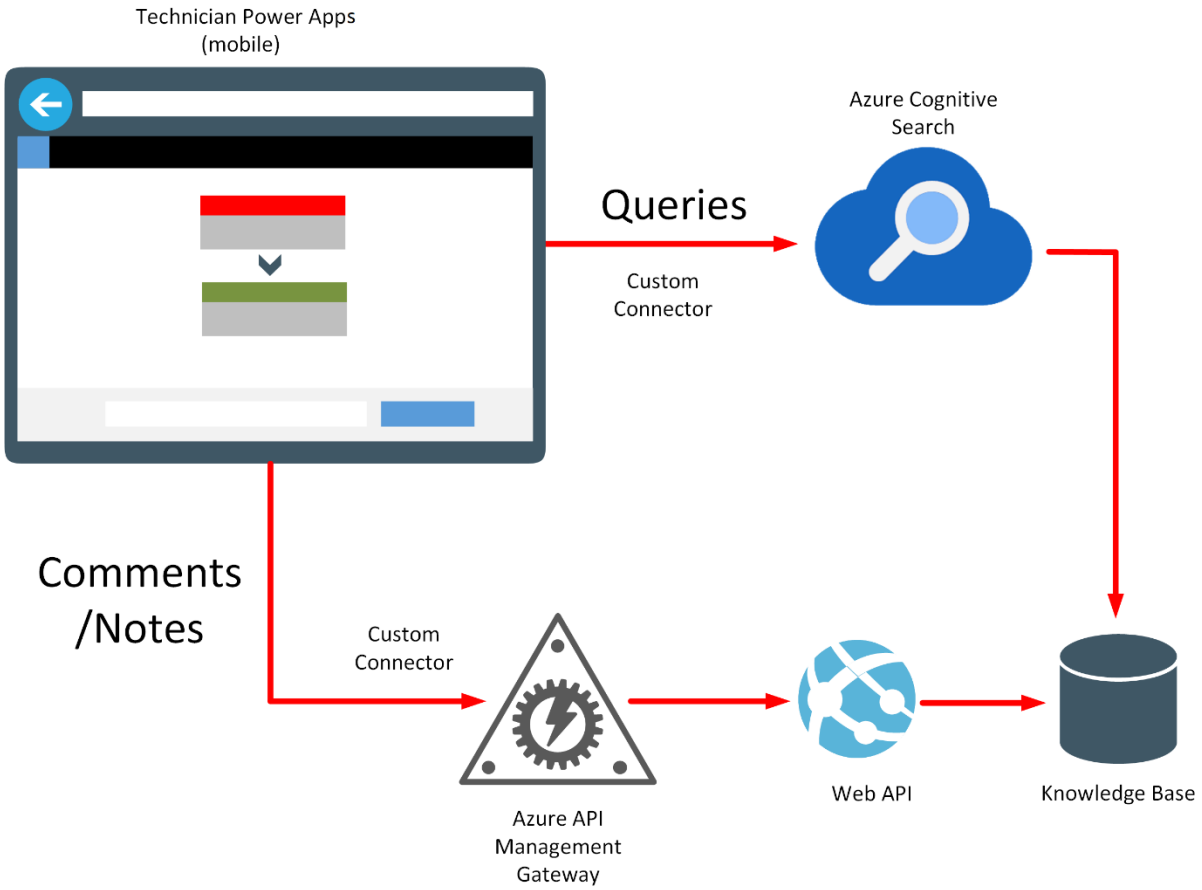


FIELD KNOWLEDGEBASE

It's impossible for one single technician to know everything about every model of furnace or air conditioner they may encounter in the field. However, with the knowledge of a great team of technicians, there's always somebody who has solved a problem before. To tap into this wealth of knowledge, an individual technician might have to play *phone tag* with several other colleagues while they track down the one person who has solved the problem they are currently facing. This approach has several issues, including:

- Making several phone calls to find the one person who has solved a problem is a time intensive process.
- The person with the answer could be busy, causing the first technician to wait.
- Knowledge is subject to ebb and flow with technician turnover. Important information can be easily lost or misremembered unless it is recorded.

A solution is to capture the information about furnaces and air conditioners, problems that have occurred, and how they were fixed, in a knowledge base. An app could allow a technician to record comments about a job and the repairs performed while still at the customer's premises. The same app could provide an interface that allows the technician to query the knowledge base about useful information that other technicians may have addressed on similar jobs. The knowledge base itself could be implemented as a database with Azure Cognitive Search providing the lookup facility, based on one or more key words.



FIELD SCHEDULING AND NOTES

Customers contact the VanArsdel office to make appointments. Throughout the day things change. Customers cancel visits and emergencies take priority over other events. Customers may provide additional information about the job. The office receptionist stores this information in a legacy customer database.

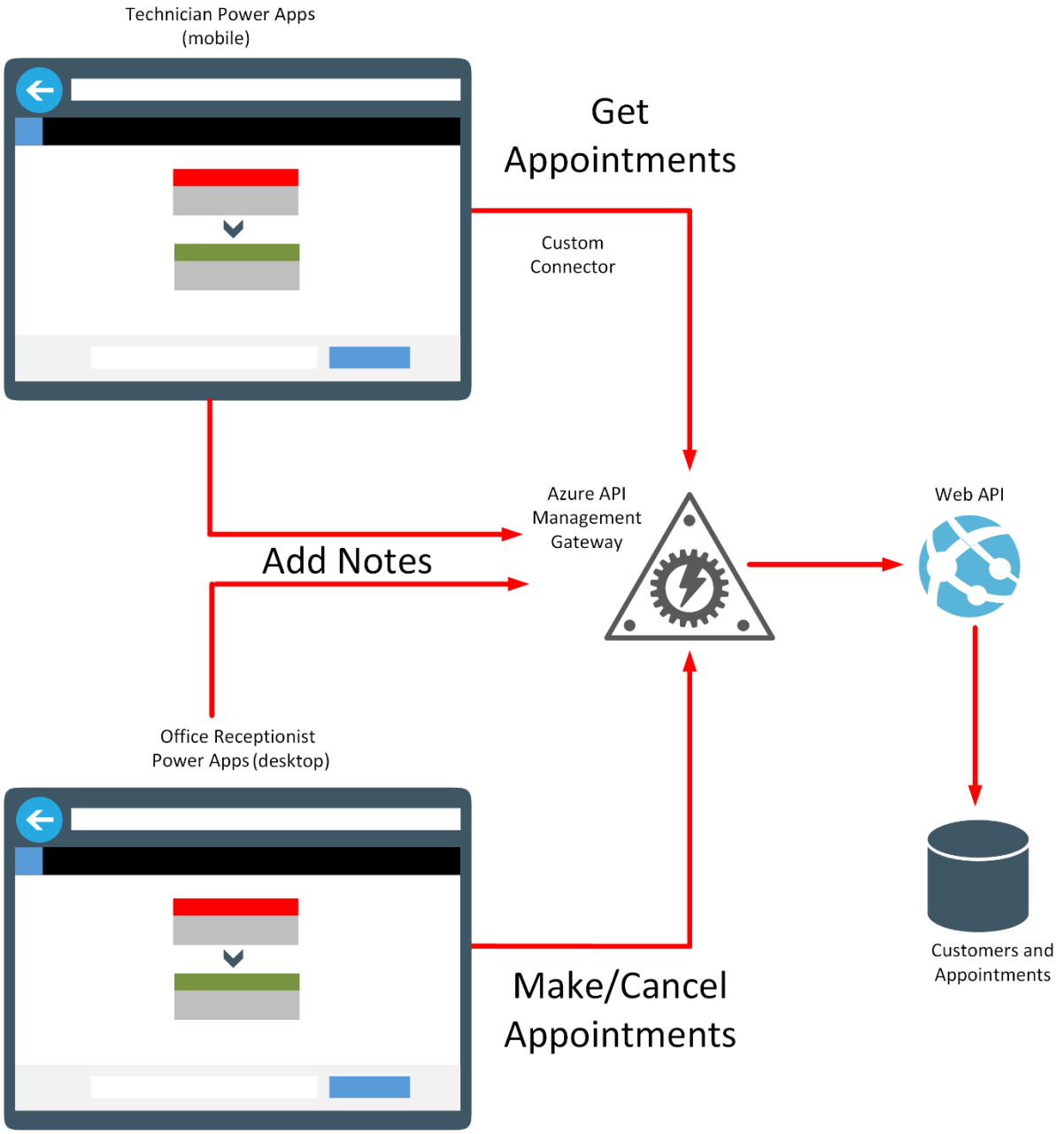
Each morning, technicians receive their schedule for customer visits for the day at the office, in the form of a printout from the legacy system, before heading into the field. This schedule contains information about customers and jobs. If this information changes during the day, the office receptionist must try to manually call the technician in the field to pass on any updates.

While in the field, the technicians take notes and will manually update the same customer information database when they return to the office at the end of the day.

There are several obvious drawbacks to the current scheduling strategy:

- If a customer cancels a visit and the office is unable to reach the technician, the technician will make an unnecessary stop. The technician might also miss a chance to be rescheduled to a new customer.
- The technician might not go to the most important jobs.
- The technician spends a lot of time at the end of the day updating customer notes when they would rather go home.

VanArsdel could use an app that acts as a front end to the legacy system. It would enable the office receptionist staff to record appointments and cancellations, and add any additional notes to customer records. An app that's available to technicians can provide access to their appointments schedule in real time and see any changes. The same app should enable technicians to enter notes about a finished job and save this information back to the legacy system.



FUSION DEVELOPMENT TEAM MEMBERS

VanArsdel Heating and Air Conditioning have started a Fusion Development Team to design and build solutions that solve the problems and inefficiencies highlighted in the previous sections. The team members are:

- **Kiana Anderson: Professional developer**. Kiana is a full-stack developer and software architect specializing in C# and .NET. She has written and designed many of VanArsdel's applications but is getting stretched very thin by all the new requests. Kiana is familiar with Power Apps at a high level but is skeptical of having *non-developers* create applications.
- **Maria Zelaya: Inventory management**. Maria makes sure VanArsdel runs like a well-oiled machine. She verifies the warehouse has enough parts and, if not, orders more using a legacy system that Kiana wrote. But more than that, Maria performs audits on the inventory, checks with vendors for the best prices, and does other inventory supply management tasks.
- **Caleb Foster: Field technician/engineer**. Caleb is VanArsdel's lead field technician. He is very knowledgeable and is on the phone a lot, mentoring junior technicians. Caleb's time is very valuable and VanArsdel wants to make sure he visits as many of their most valuable customers on a daily basis.

- **Malik Barden: Office receptionist**. Malik is the heart of the VanArsdel office. He answers all customer inquiries, schedules appointments, and even helps technicians find answers when they need to. In other words, he's overworked and needs to automate some of his repetitive tasks to provide even better customer service.
- **Preeti Rajdan: IT Operations**. Preeti is responsible for making sure the IT systems are up and running. She worries a lot about security and applications that might accidentally leave *backdoors* open. She also is stretched thin and needs to be sure any new apps are easy to govern and administer.

CHAPTER 3: BUILDING A LOW-CODE PROTOTYPE

Note: The previous chapter referenced the mobile app used by the field technicians and engineers, and desktop Power Apps used by on-premises staff. The following chapters focus on the design, implementation, and rollout of the mobile Power Apps. The desktop Power Apps are left as an exercise for the reader.

Kiana is skeptical of low-code solutions and Power Apps, but she and Maria decide to build an app together to help the field technicians check inventory (and order parts if necessary), query the knowledge base, and check their next appointment while out of the office, on service calls. Kiana and Maria plan to use this experience to explore how to add controls and use formulas in Power Apps.

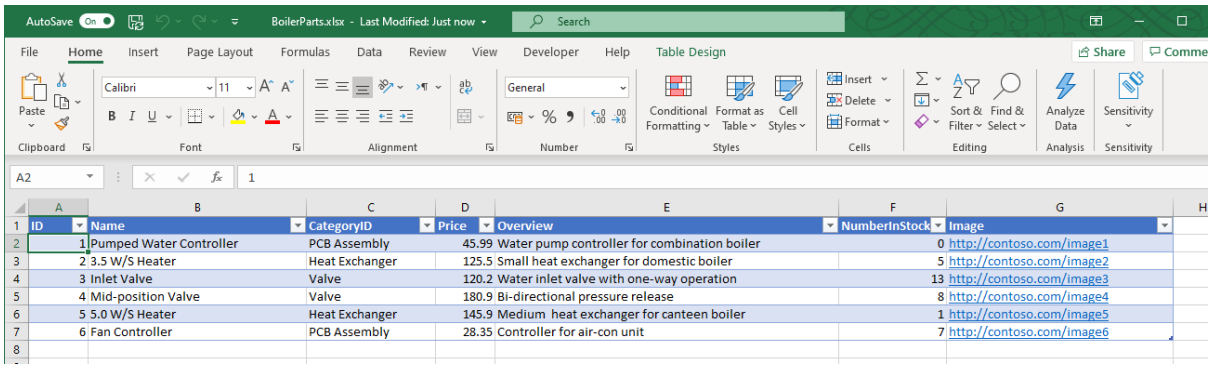
Although building an initial low-code prototype is primarily a typical citizen developer task, Kiana decides to pay attention to the process to ensure that she understands how the app is constructed. She needs this information to enable her to help Maria integrate the real-world data sources, Web APIs, and other services required into the app.

ITEM 1: FIELD INVENTORY MANAGEMENT

Maria's initial aim is to build a canvas app that displays a list of parts and enables the user to view the details of any part. The user should also be able to order a part. However, this initial version of the app will simply be a prototype and won't be hooked up to a back end yet. After she has obtained feedback from Caleb, the lead field technician, Maria will work with Kiana on integrating the canvas app with the inventory system running on-premises.

Maria is very familiar with the existing inventory management system and understands the information that it contains. She starts by creating an Excel spreadsheet that contains tables holding mock data with the details of some sample parts. The fields in the table are **ID**, **Name**, **CategoryID**, **Price**, **Overview**, **NumberInStock**, and **Image** (a URL that references an image of the part). She can use this spreadsheet to build and test the canvas app, to ensure that it displays the required data correctly. She stores this spreadsheet in her OneDrive account with the name **BoilerParts.xlsx**:

Note: You can find a copy of this spreadsheet in the **Assets** folder in the Git repository for this guide.



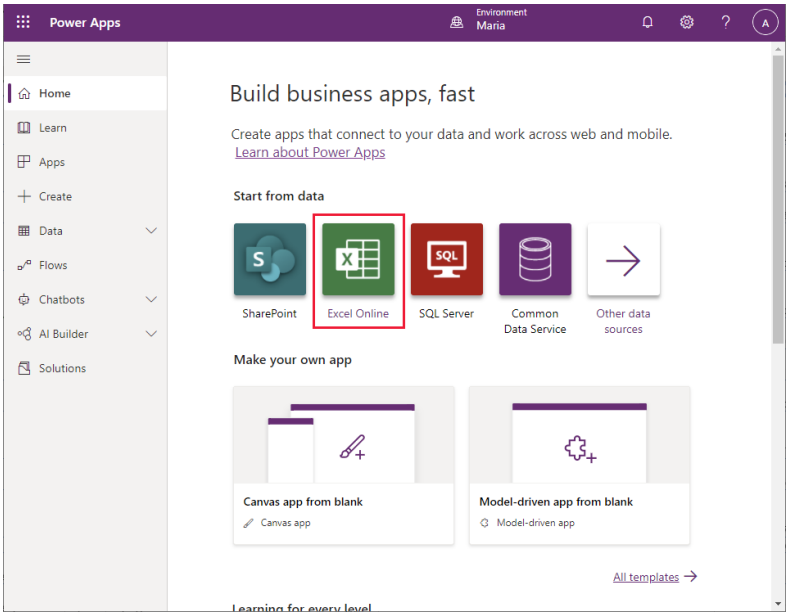
	A	B	C	D	E	F	G
1	ID	Name	CategoryID	Price	Overview	NumberInStock	Image
2	1	Pumped Water Controller	PCB Assembly	45.99	Water pump controller for combination boiler	0	http://contoso.com/image1
3	2	3.5 W/S Heater	Heat Exchanger	125.5	Small heat exchanger for domestic boiler	5	http://contoso.com/image2
4	3	Inlet Valve	Valve	120.2	Water inlet valve with one-way operation	13	http://contoso.com/image3
5	4	Mid-position Valve	Valve	180.9	Bi-directional pressure release	8	http://contoso.com/image4
6	5	5.0 W/S Heater	Heat Exchanger	145.9	Medium heat exchanger for canteen boiler	1	http://contoso.com/image5
7	6	Fan Controller	PCB Assembly	28.35	Controller for air-con unit	7	http://contoso.com/image6
8							

Notes: if you are a relational database designer, you'll notice that the Excel spreadsheet presents a denormalized view of the data. For example, in a relational database, **CategoryID** would most likely be a numeric identifier that references a separate table containing the details of the category, including the name.

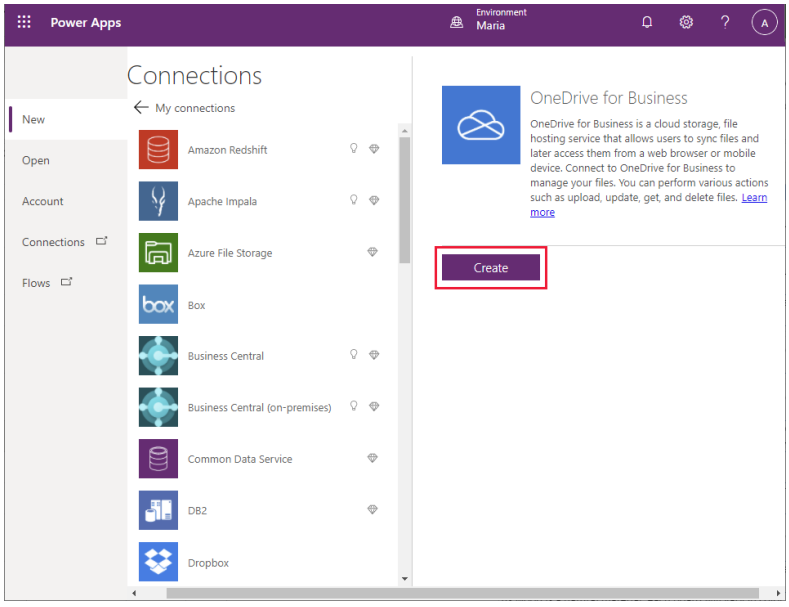
The URLs in the Image column are currently just placeholders. In the completed app, these URLs will be replaced with the addresses of real image files.

Follow these steps to create the app:

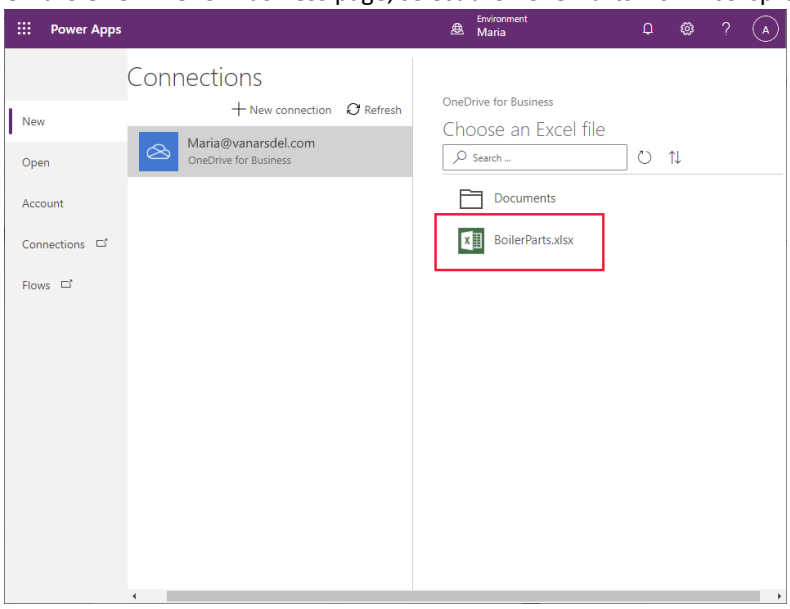
1. Sign in to Power Apps Studio at http://make.powerapps.com.
2. On the **Home** page, under **Start from data**, select **Excel Online**:



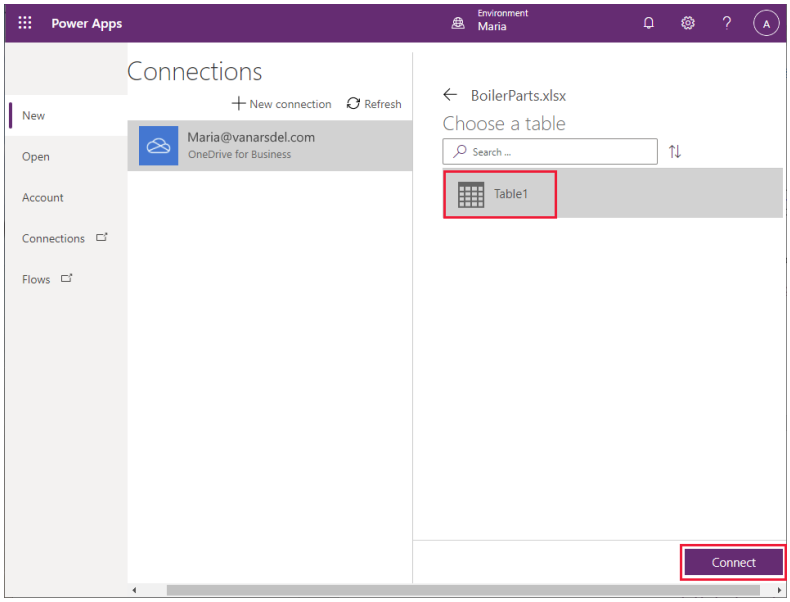
3. On the **Connections** page, select **OneDrive for Business**, and then select **Create**:



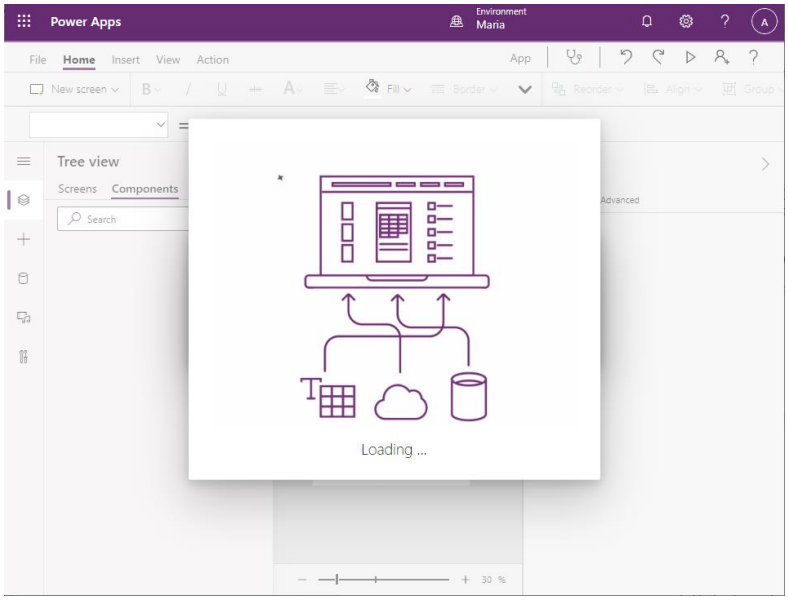
4. On the **OneDrive for Business** page, select the **BoilerParts.xlsx** Excel spreadsheet:



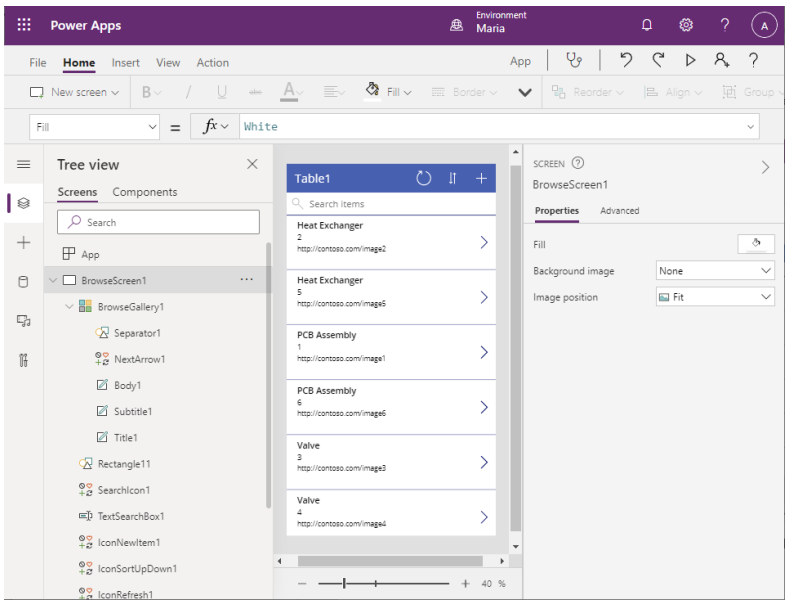
5. Select the table in the Excel sheet (Maria created the table with the default name, **Table1**), and then select **Connect**:



6. Wait while Power Apps generates the app:

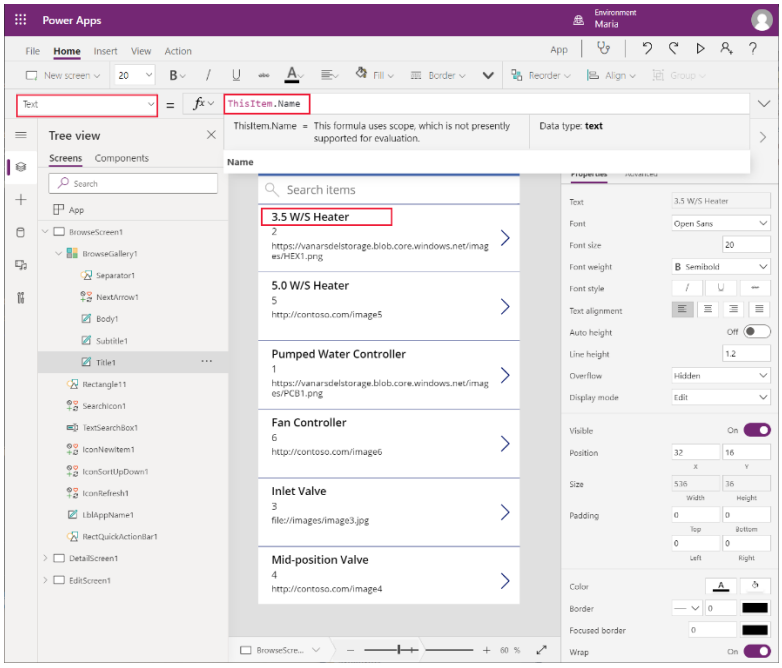


7. When the app has been generated, you should see the **Browse** screen, displaying the **CategoryID**, **ID**, and **Image** fields from each row of the parts table in the spreadsheet:

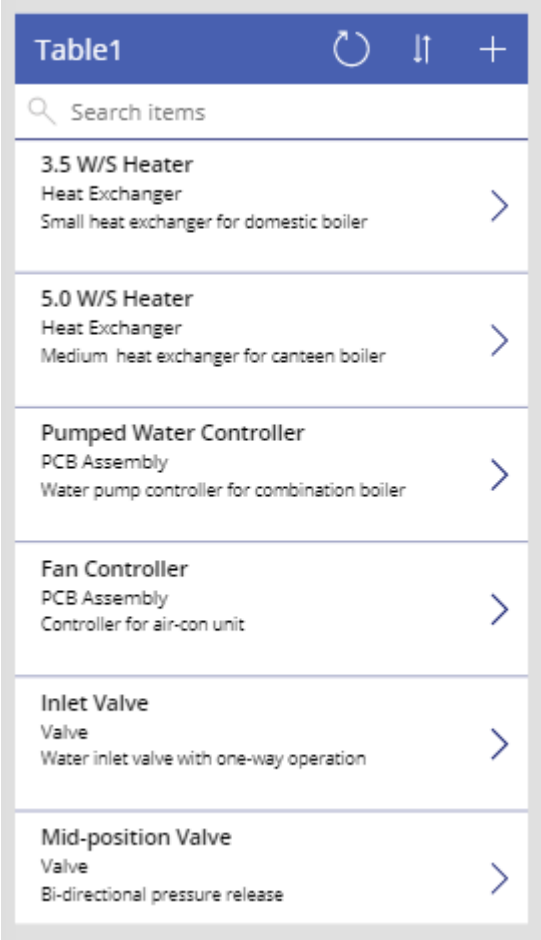


8. The fields currently displayed are not that useful for helping an engineer to select a specific product. In the pane displaying the **Browse** screen, select the **Heat Exchanger** label in the first row of data. In the formula bar select the **Text** property from the drop-down list. Change the value of this property to **ThisItem.Name**. The text in the first field of each row will switch to display the part name.

> **Note**: In the image below, the **Heat Exchanger** label originally displayed on the form has changed to the product name, **3.5 W/S Heater**.

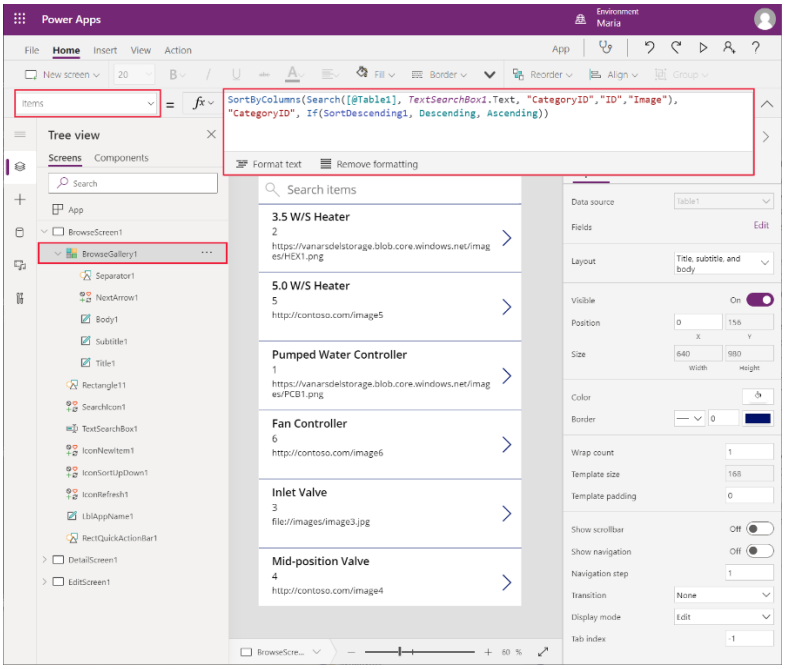


9. Repeat the previous step for the **ID** and **Image** labels. Change the **Text** property of the **ID** field to **CategoryID**, and the **Text** property of the **Image** field to **Overview**. The **Browse** screen should now look like this. The field engineer should find this information more useful for selecting parts:



10. The search feature of the **Browse** screen defaults to searching using the fields that were initially selected when the screen was generated—in this case, the **CategoryID**, **ID**, and **Image** fields. The results are sorted

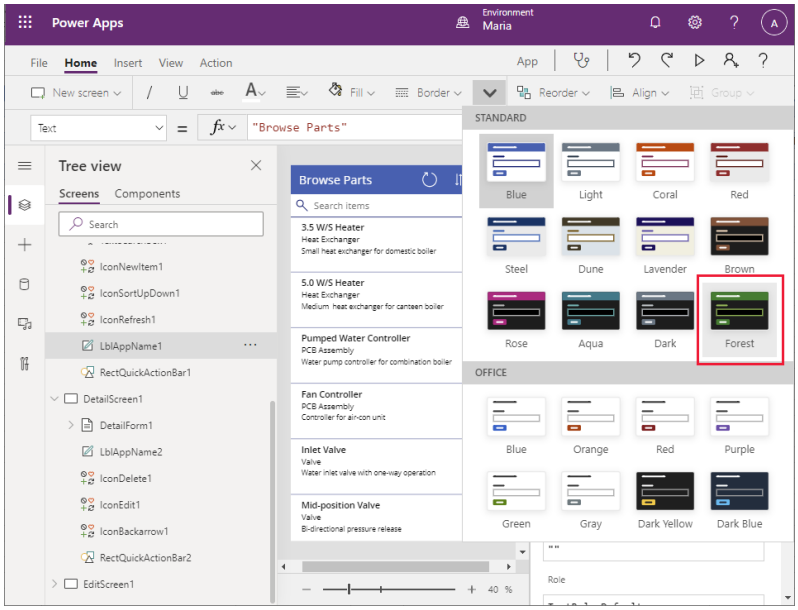
by **CategoryID**. It makes sense to switch this to the **Name, CategoryID**, and **Overview** fields, with the results sorted by **Name**. Select the **BrowseGallery1** control in the **Tree view** pane. In the drop-down list by the formula bar select the **Items** property. Drag the lower edge of the formula bar down so the formula is completely visible. The formula contains the expression **SortByColumns(Search([@Table1], TextSearchBox1.Text, "CategoryID","ID","Image"), "CategoryID", If(SortDescending1, Descending, Ascending))**:



11. Change the **Search** expression to reference the **Name**, **CategoryID**, and **Overview** fields: **SortByColumns(Search([@Table1], TextSearchBox1.Text, "Name", "CategoryID", "Overview"), "Name", If(SortDescending1, Descending, Ascending))**.

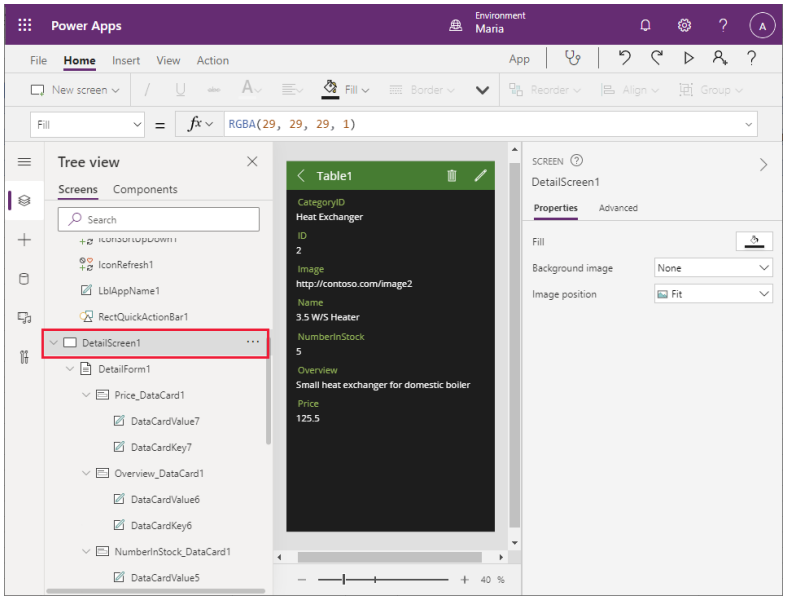
12. The title in the form header isn't helpful, and the default theme doesn't match the corporate look and feel. In the **Browse** screen, select the **Table1** label, and on the **Formula** bar change the **Text** property of the label to **"Browse Parts"** (include the double quotes in the value):

13. In the toolbar, select **Theme** (you might have to click the drop-down arrow displaying more items for the toolbar), and then select the **Forest** theme. The colors and styling for the **Browse** screen should change to match this theme:



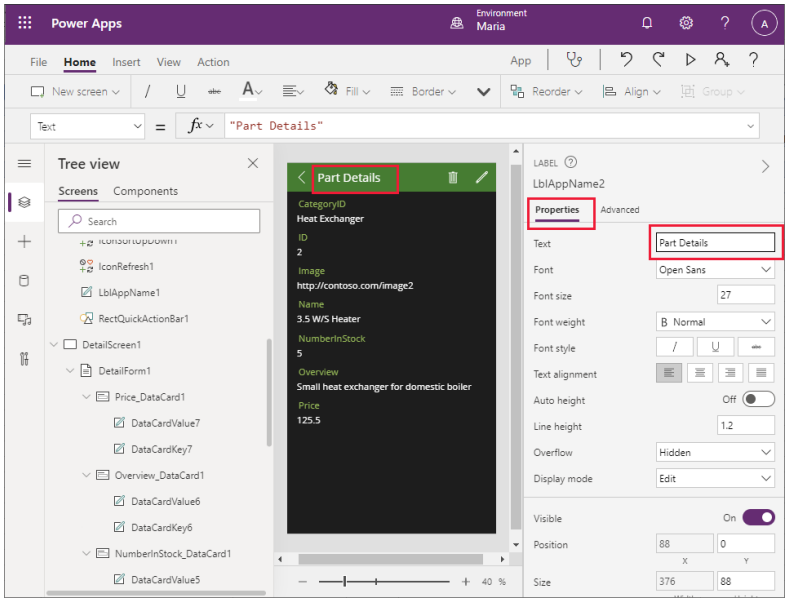
You have created the basic app and modified the **Browse** screen to display the fields that an engineer can use to identify a part. The app also contains a **Details** screen which shows all the information for a selected part. The fields on this screen aren't currently displayed in a logical sequence, so you should rearrange them. You can also delete the Part ID from this screen as this information is irrelevant to an engineer. The following steps describe this process:

1. In the **Tree view** pane on the left, scroll down and select the **DetailScreen1** screen. This screen appears when the user selects an item in the **Browse** screen. It displays the details of the selected part:



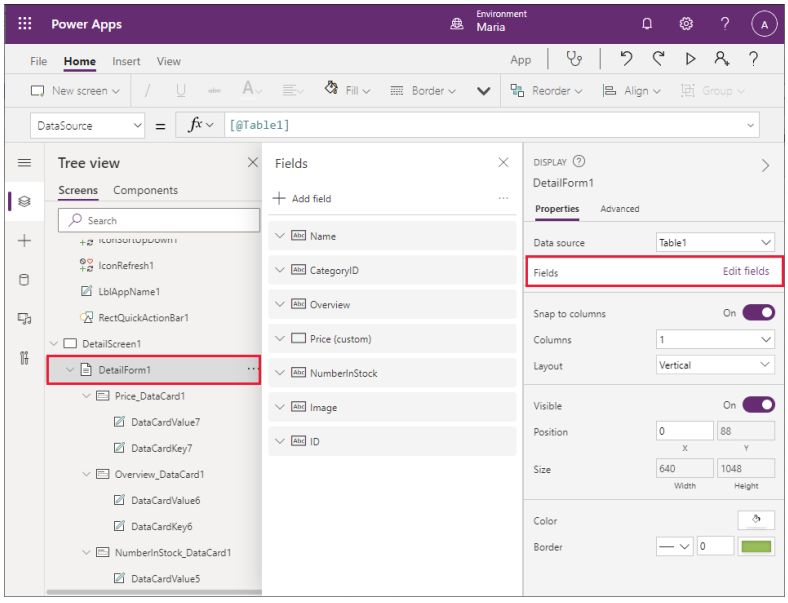
2. In the header of the **Details** screen in the middle pane, select the **Table1** label. On the **Properties** tab in the right pane, change the **Text** property to **Part Details**.

> **Note**: In many cases, the **Properties** pane is an alternative to using the formula bar; the results are the same. However, there are some properties which are only available through the **Properties** pane.

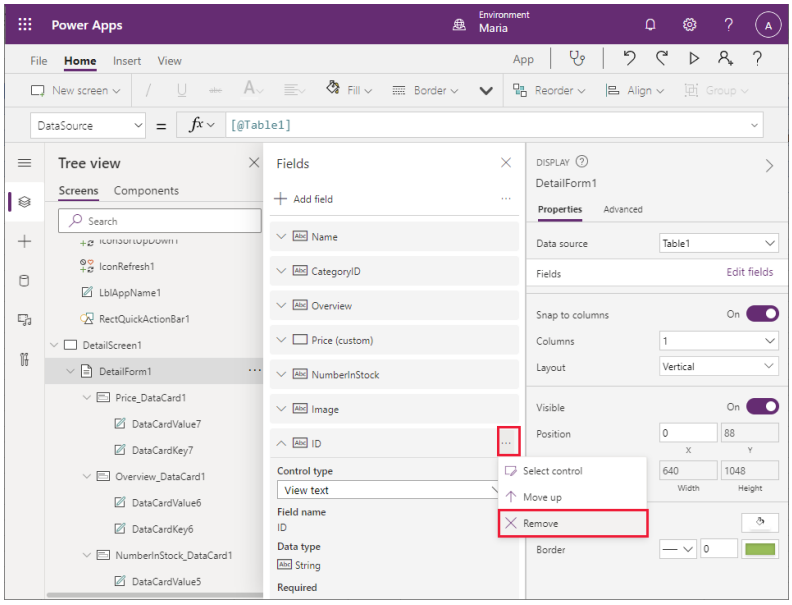


3. In the **Tree view** pane on the left, scroll down and select the **DetailForm1** form under the **DetailScreen1** screen. In the right pane, on the **Properties** tab, in the **Fields** field, select **Edit fields**. In the middle pane, click and drag each field so that they are displayed in the following order, from top to bottom:

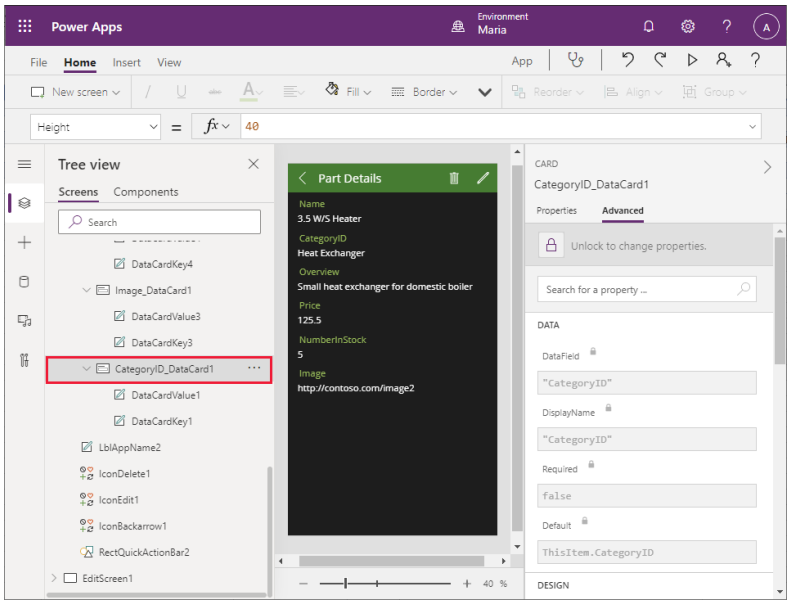
    - Name
    - CategoryID
    - Overview
    - Price
    - NumberInStock
    - Image
    - ID



4. Select the **ID** field, click the ellipsis that appears on the right of the field, and then select **Remove** from the drop-down menu that appears. This action removes the ID field from the form:

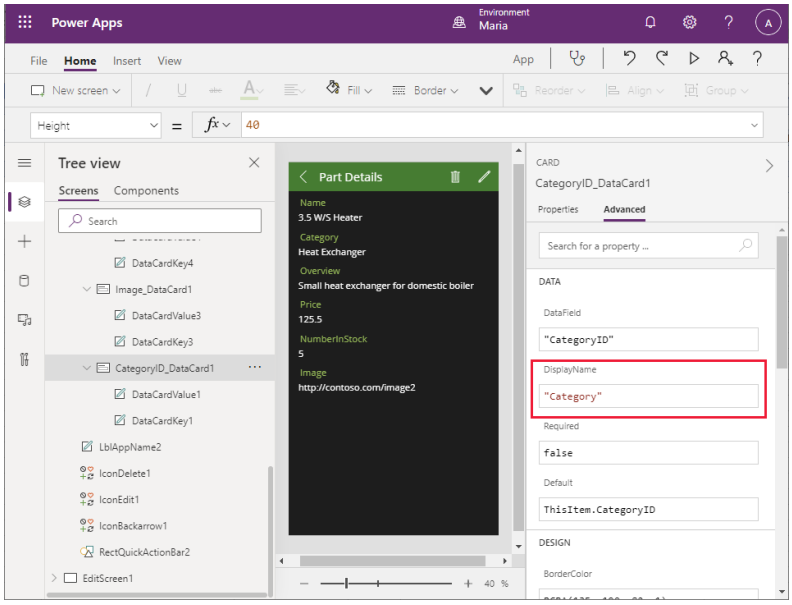


5. In the **Tree view** pane on the left, scroll down and select the **CategoryID_DataCard1** element under the **DetailForm1** form. This element is a **DataCard** control that displays the name of a field (called the **key**) and its value.



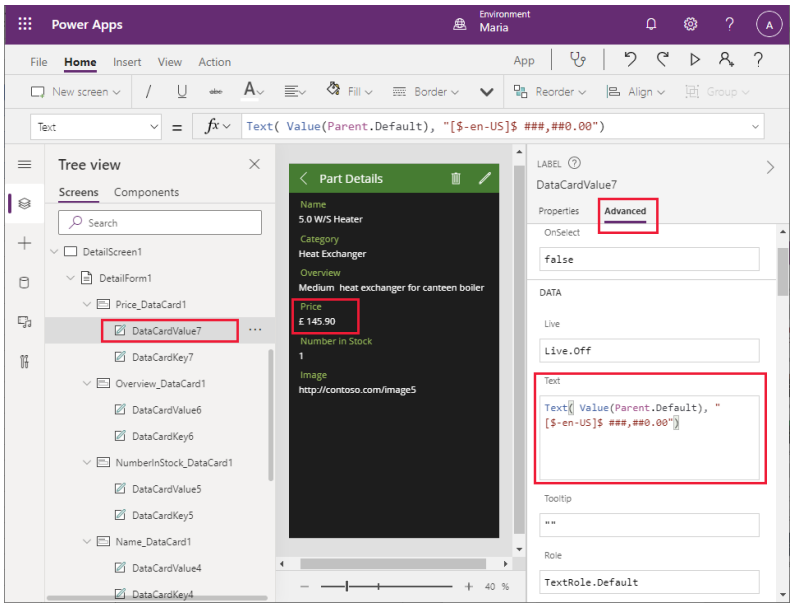
In the right pane, on the **Advanced** tab, select **Unlock to change properties**. In the **Data** section, change the **DisplayName** field to **"Category"** (including the quotes).

> **Note:** As with the **Properties** tab, many of the properties on the **Advanced** tab are also accessible through the formula bar. In these situations, you can use the formula bar if you prefer.



6. Repeat this process to change the key for the **NumberInStock_DataCard1** DataCard to **"Number in Stock"**.

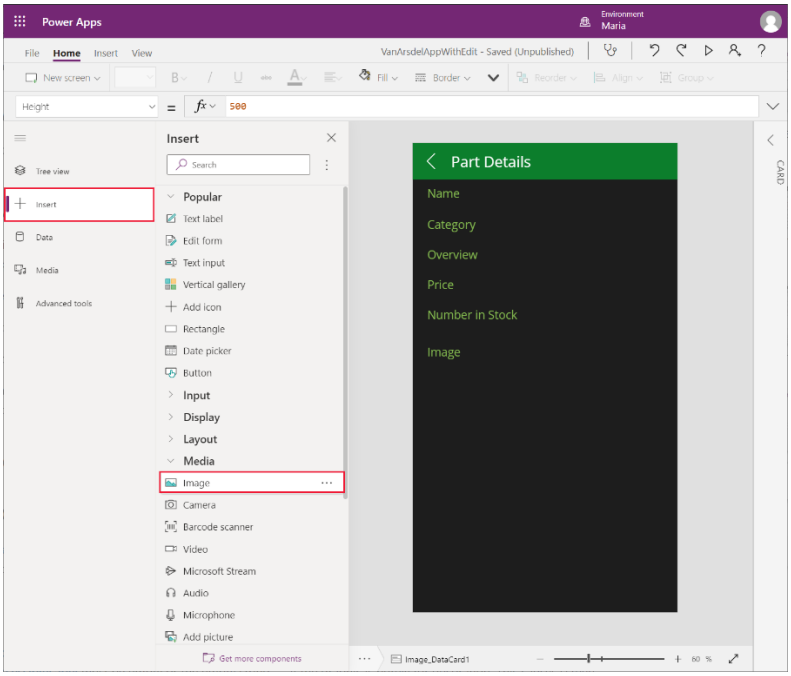
7. The formatting of the **Price** field should be adjusted to display the data as a currency value. In the **Tree view** pane, locate the **Price_DataCard1** element under the **DetailForm1** form and select the **DataCardValue7** element. This is the field that displays the value of the **Price** field. In the **DataCardValue7** pane on the right, on the **Advanced** tab, change the **Text** property to **Text(Value(Parent.Default), "[$-en-US]$ ###,##0.00")**:



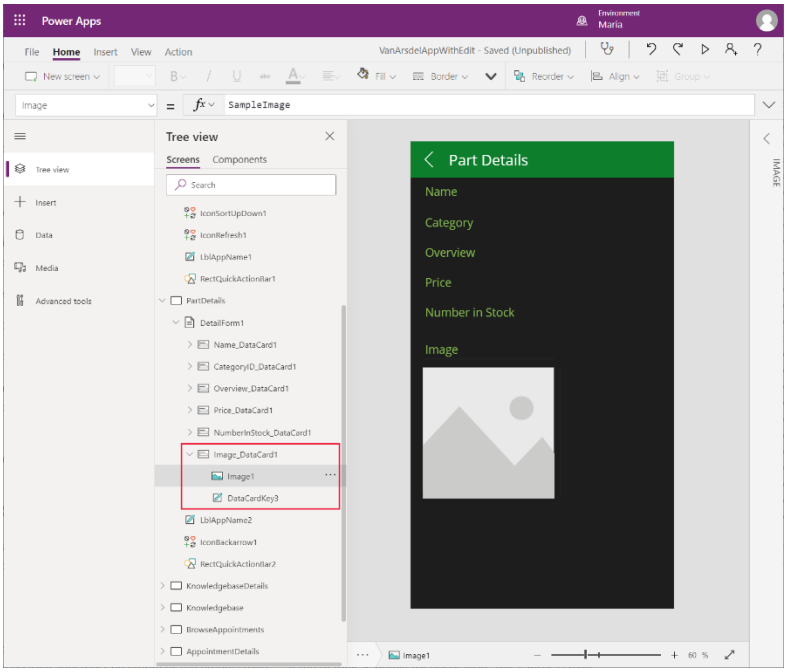
   The expression **Parent.Default** refers to the data item to which the parent control (the **DataCard**) is bound; in this case, the **Price** column. The **Text** function reformats this value using the format specified as the second argument; local currency with two decimal places, in this example.

8. The Image **DataCard** should display an image of the part rather than the URL of the image file. In the **Tree view** pane, locate the **Image_DataCard1** element under the **DetailForm1** form, select the **DataCardValue3** element, and then press **Delete** to remove this control.

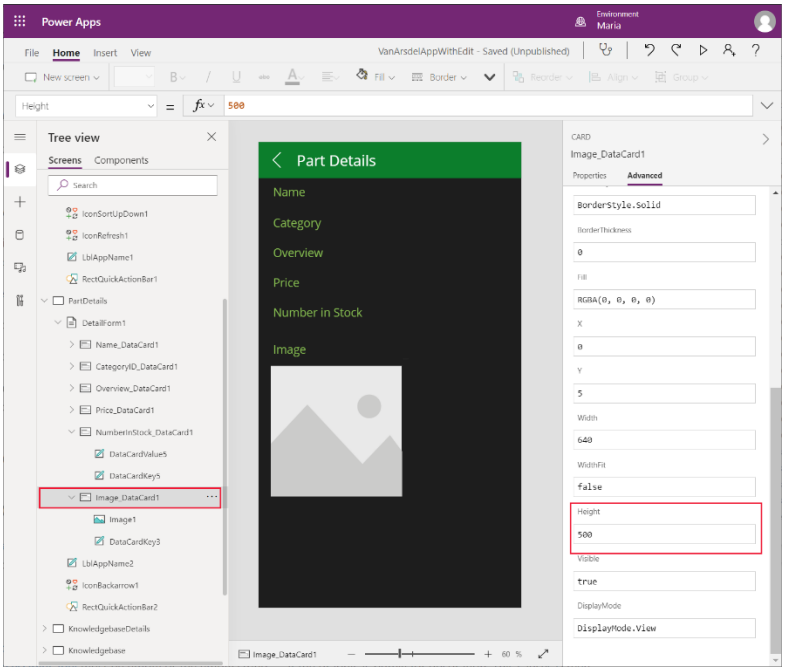
9. Select the **Image_DataCard1** element. In the left pane, select **+ Insert**. In the **Insert** pane, expand the **Media** subtree, and select the **Image** control:



10. Return to the **Tree view** pane, and verify that the HTML text control (**Image1**) has been added to the **Image_DataCard1** control:



11. In the **Tree view** pane, select the **Image_DataCard1** control. Change the **Height** property to **500**, to allow sufficient space for an image to be displayed:



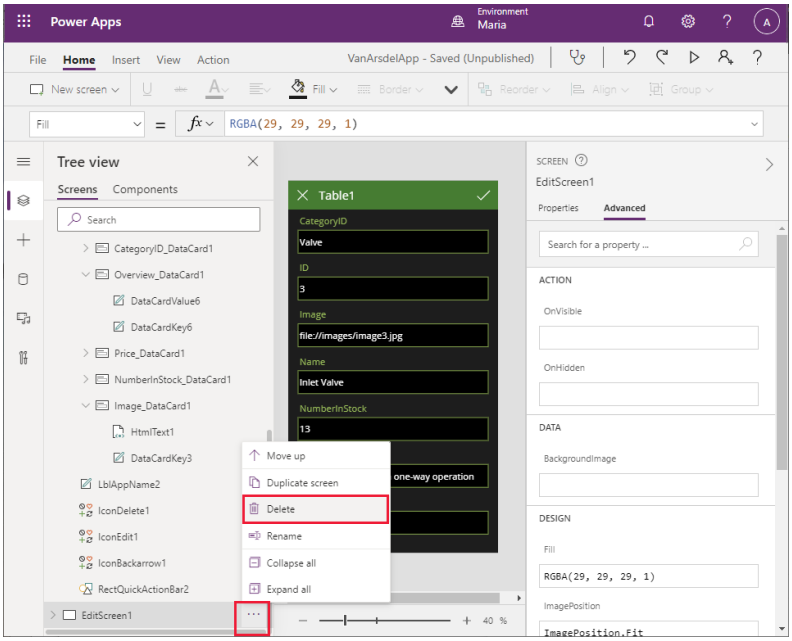
12. In the **Tree view** pane, select the **Image1** control. Set the following properties:

    - Image: **Parent.Default**
    - ImagePosition: **ImagePosition.Fit**
    - Width: **550**
    - Height: **550**

> **Note:** The image displayed is currently empty because the URL in the Excel spreadsheet is just a placeholder. You'll address this issue, and fetch a real URL, when you bind the app to a Web API in a later chapter.

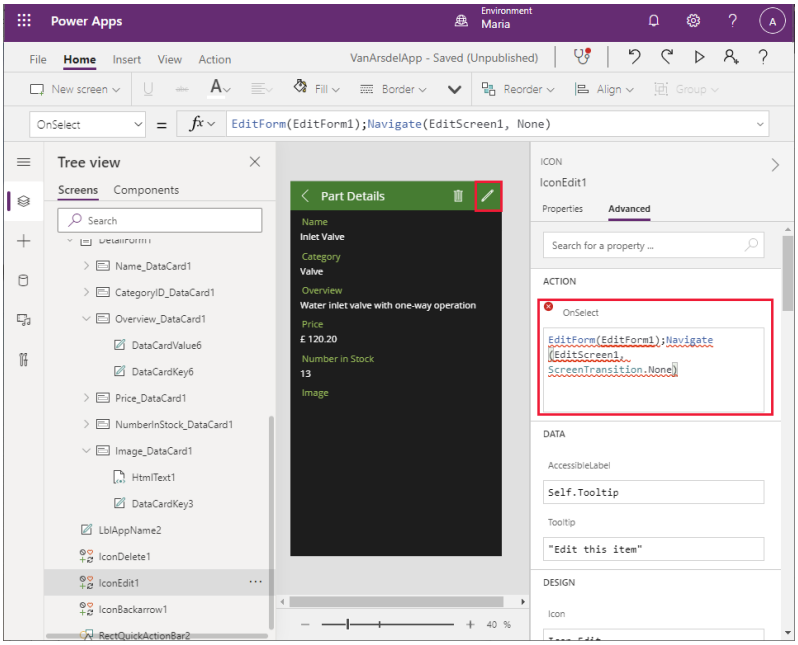
The app also contains an **Edit** screen which enables a user to change the information for a part. An engineer shouldn't be able to change the details of a part, create new parts, or delete parts from the catalog. Remove this functionality from the app as follows:

1. In the **Tree view** pane on the left, scroll down and select the **EditScreen1** screen. Select the ellipsis button, and then select **Delete** to remove this screen:



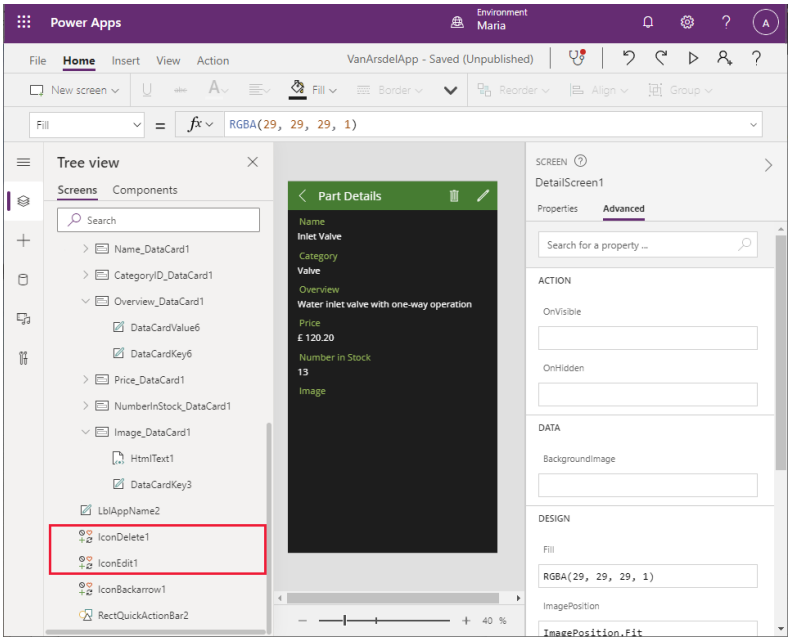
2. In the **Tree view** pane on the left, scroll down and select the **DetailsScreen1** screen. Notice that Power Apps Studio is displaying an error message for this screen. This error occurs because the **DetailsScreen1** contains expressions that reference the **EditScreen1** screen, which no longer exists:

3. In the header of the **DetailsScreen1**, select the pencil (**IconEdit1**) icon. The **OnSelect** property for this control contains the expression **EditForm(EditForm1);Navigate(EditScreen1, ScreenTransition.None)**. When the **Edit** icon is selected, this expression runs and attempts to move to the **EditScreen1** screen:

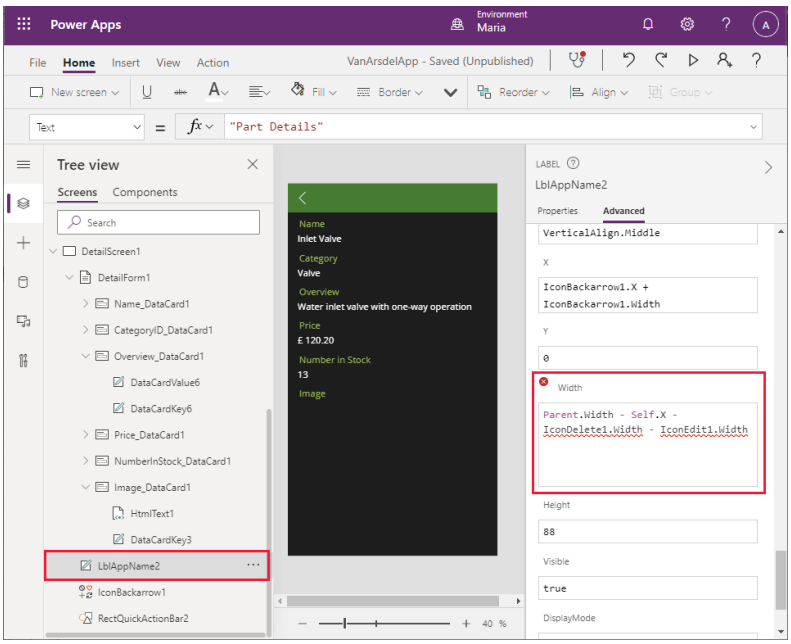


4. In the **Tree view** pane, select the **IconEdit1** control and press **Delete**. This icon is no longer required.

5. Select the **IconDelete1** control and press **Delete**. This icon is used to delete the current part, and is also not required:

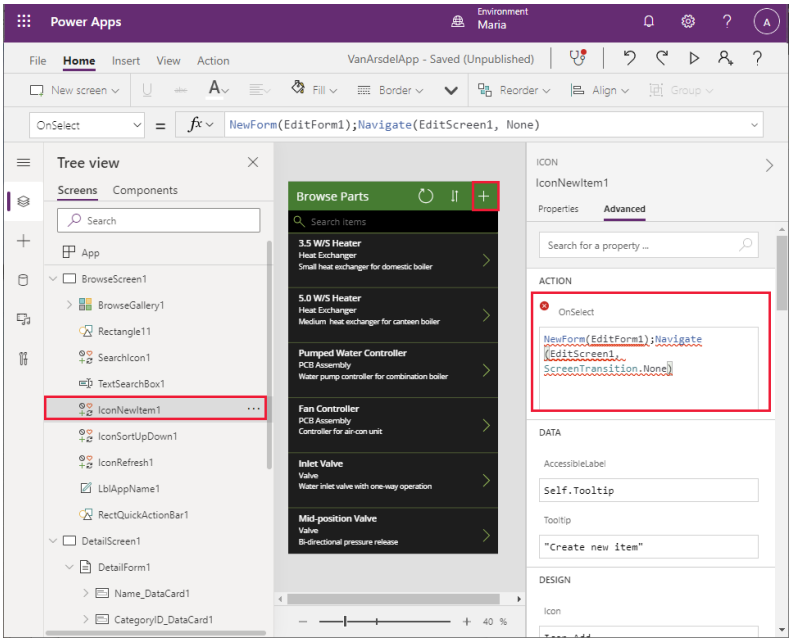


6. Notice that the **Part Details** text has disappeared from the screen header, and instead Power Apps Studio is displaying an error message. This has happened because the width of the label control displaying the text is calculated. In the **Tree view** pane, select the **LblAppName2** label control. Examine the **Width** property. The value of this property is the result of the expression **Parent.Width - Self.X - IconDelete1.Width - IconEdit1.Width**:



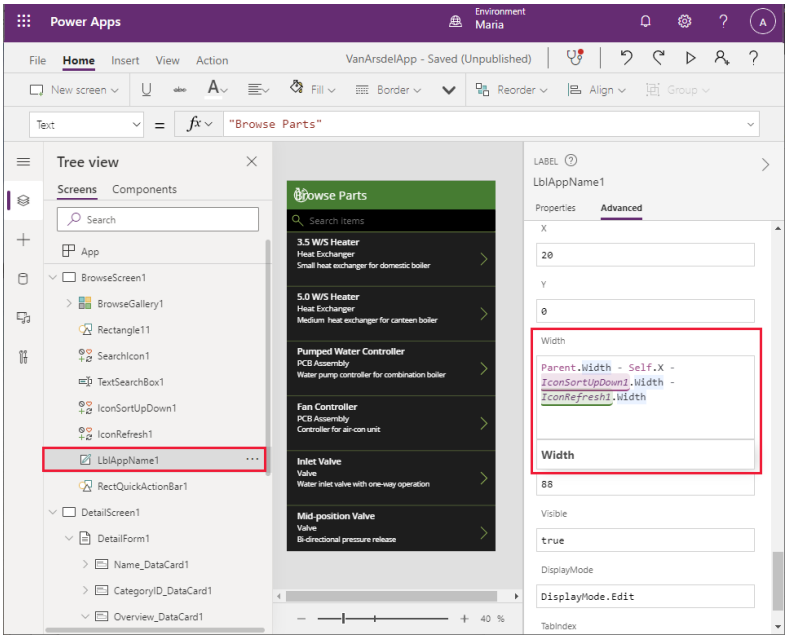
7. Change the expression for the **Width** property to **Parent.Width - Self.X**. The error should disappear, and the **Part Details** text should reappear in the screen header.

8. In the **Tree view** pane, scroll up and select the **BrowseScreen1** screen. This screen will also display an error message. The **+** icon in the toolbar (**IconNewItem1**) enables the user to add a new part. The **OnSelect** property for this icon references the **EditScreen1** screen.

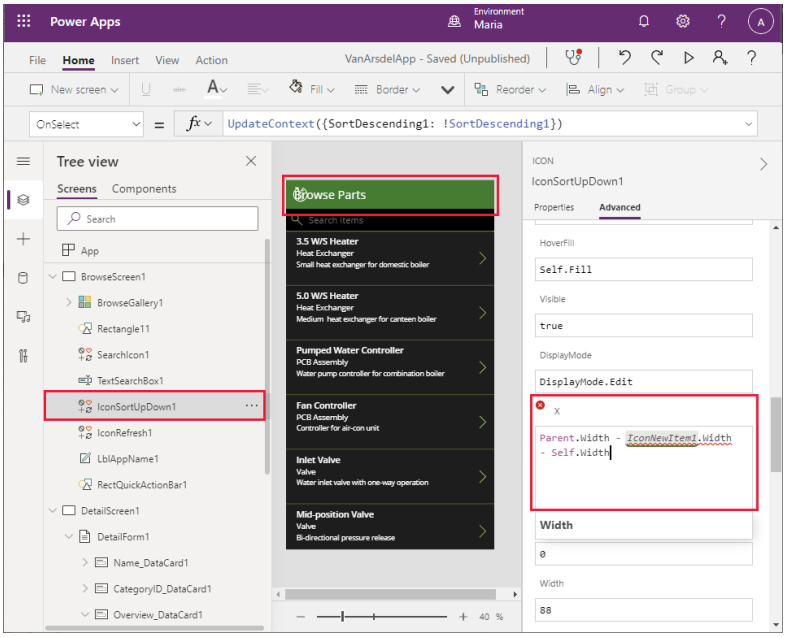


9. Select the **IconNewItem1** icon, and then press **Delete**. As before, the text in the header for the screen disappears with an error message, and for the same reason.

10. In the **Tree view** pane, select the **LblAppName1** label control under **BrowseScreen1**. Modify the expression for the **Width** property and remove the reference to **IconNewItem1.Width**. The new expression should be **Parent.Width - Self.X - IconSortUpDown1.Width - IconRefresh1.Width**:



11. There is still a problem with the header. Although the **Browse Parts** text has reappeared, there's an error and the refresh and sort icons are in the wrong place. Select the **IconSortUpDown1** control in the **Tree view** pane. Find the **X** property for this control. This property determines the horizontal position of the icon in the header. It is currently calculated based on the width of the **IconNewItem1** control:



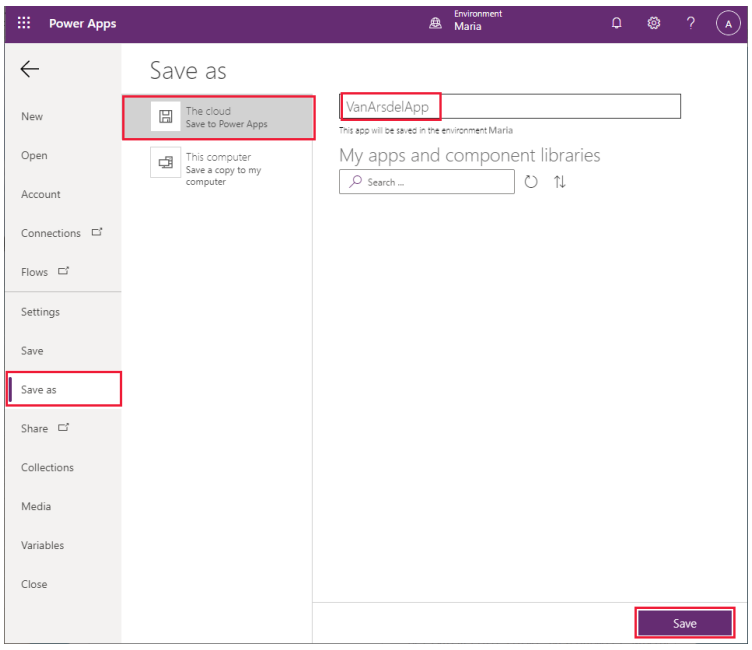
12. Change the value of the expression for the **X** property to **Parent.Width - Self.Width**.

13. In the **Tree view** pane, select the **IconRefresh1** control. Change the **X** property for this control to **Parent.Width - IconSortUpDown1.Width - Self.Width**. The errors should now have all disappeared.

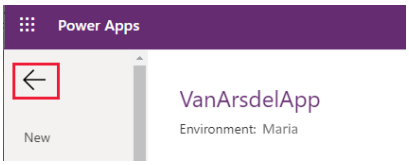
You can save and test the application:

1. Select the **File** menu in the Power Apps Studio toolbar, and then select **Save as**.

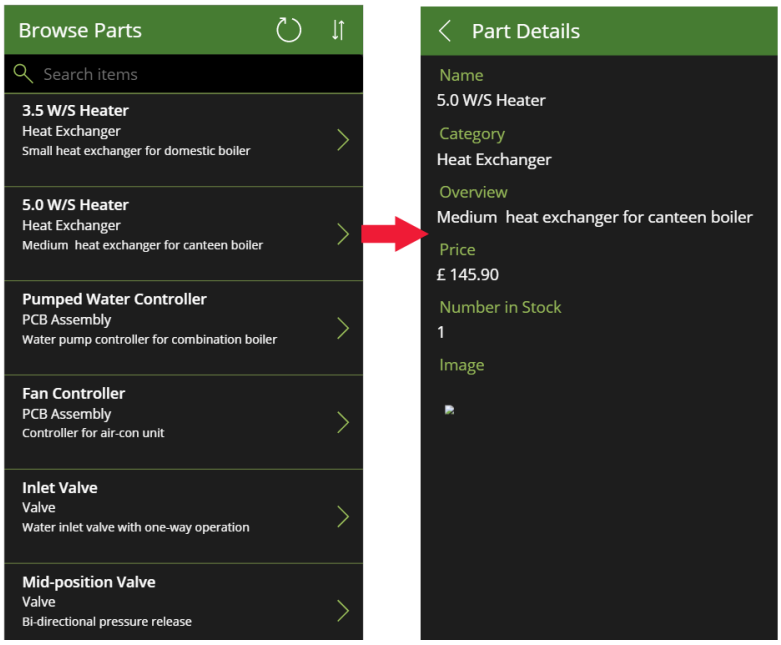
2. Under **Save as**, select **The cloud** as the destination, name the app **VanArsdelApp**, and then select **Save**.



3. Press the back arrow icon in the Power Apps Studio toolbar to return to the **Home** screen:

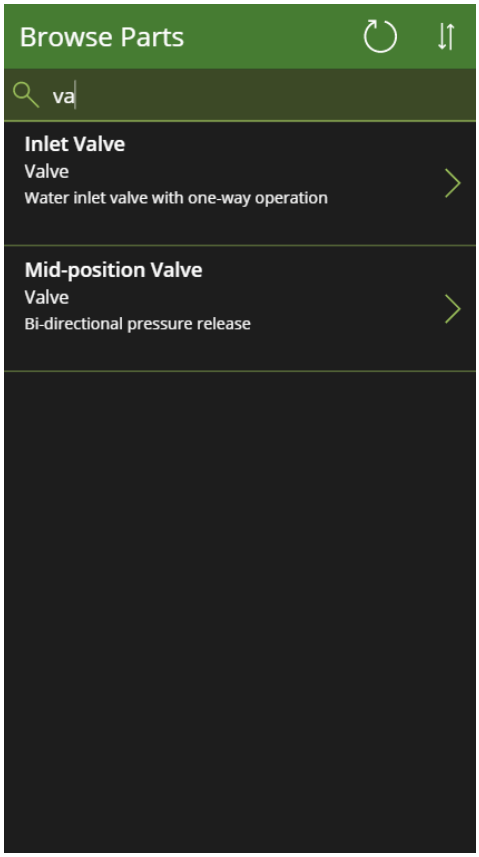


4. Press **F5** to preview the app. On the **Browse Parts** page, select the **>** symbol for any part. The **Details** screen for the part should appear:



5. Click the **<** icon in the header in the **Details** screen to return to the **Browse** screen.

6. On the **Browse** screen, enter some text in the **Search** box. As you type, the items will be filtered to only those that have matching text in the **Name**, **CategoryID**, or **Overview** fields:



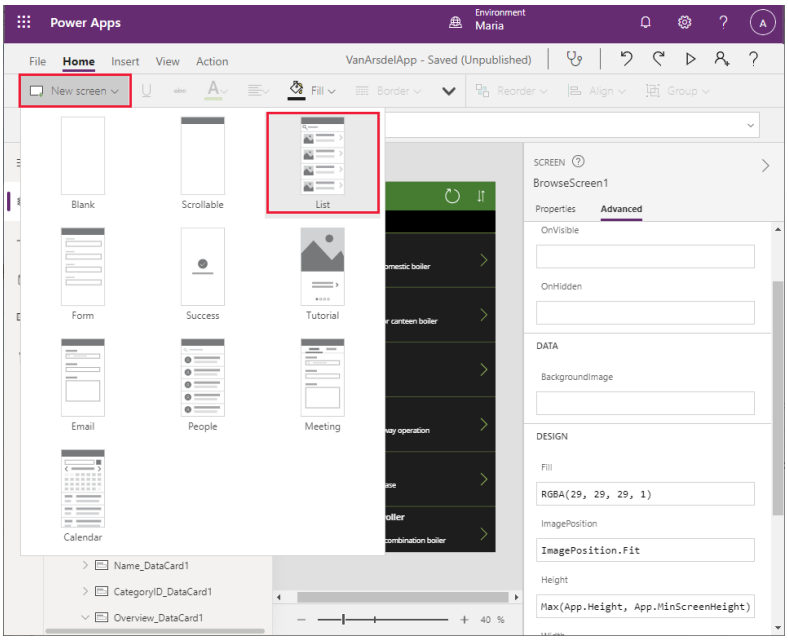
7. Close the preview window and return to Power Apps Studio.

ITEM 2: FIELD KNOWLEDGE BASE

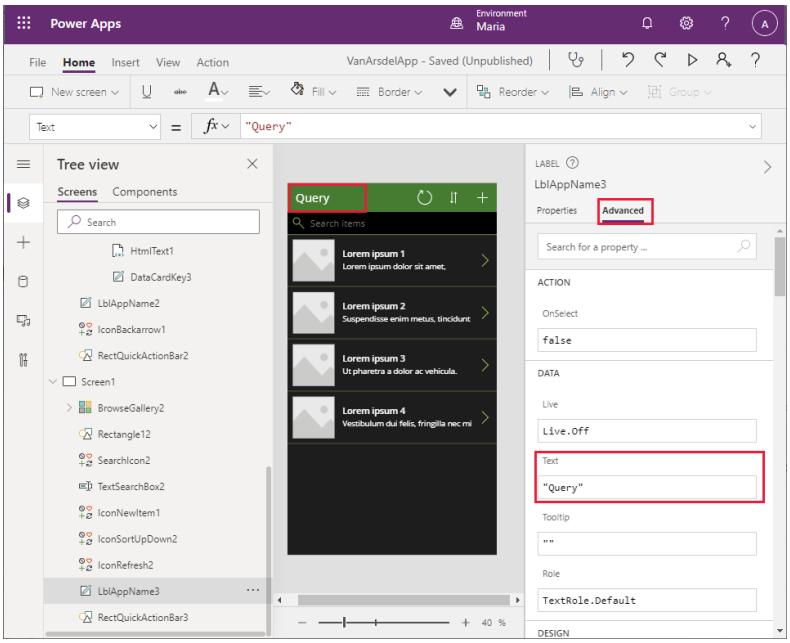
For access to the knowledge base, Maria and Caleb (the technician) envisage a simple interface that enables the user to enter a search term, and for the app to display all knowledge base articles that mention the term. Maria knows that this process is going to involve Azure Cognitive Search, but doesn't need, or want, to understand how it works. Therefore, Maria decides to just provide the basic user interface and will work with Kiana later to add the actual functionality.

Maria decides to create a new screen based on the **List** template available in Power Apps Studio, as follows:

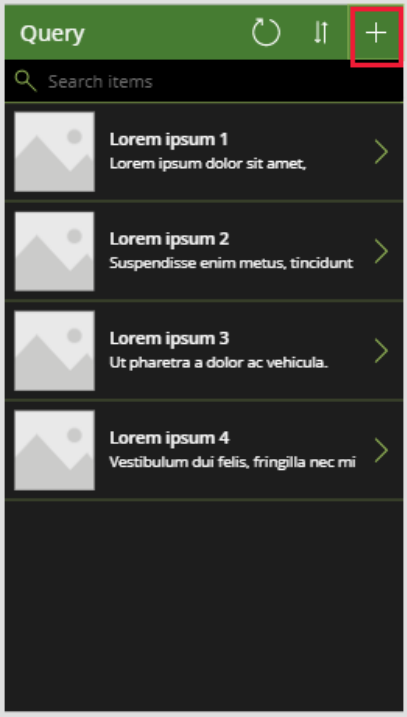
1. In Power Apps Studio, on the **Home** screen, in the toolbar, select **New Screen**, and then select the **List** template:



2. In the screen header, select the label displaying the text **[Title]**. Change the **Text** property to **"Query"** (including the quotes):

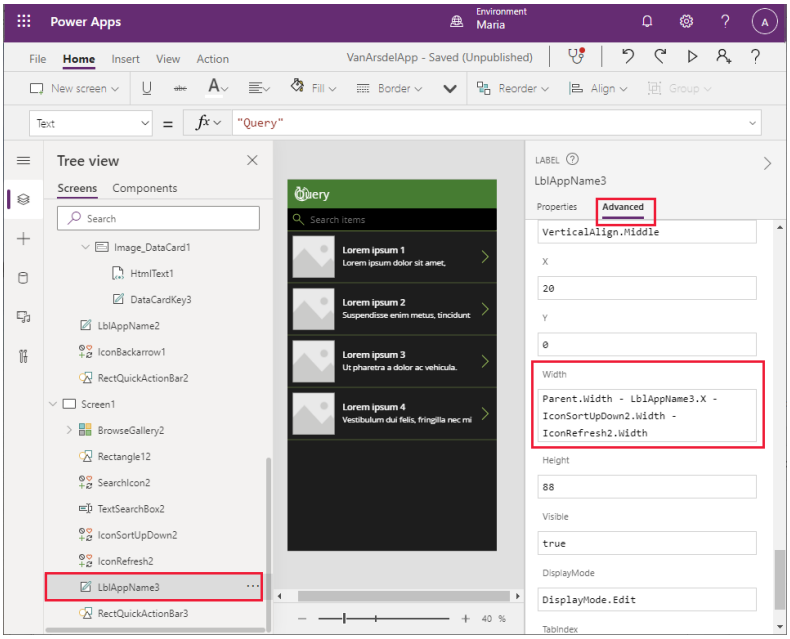


3. In the screen header, select the **+** icon, and then press **Delete**. The + icon is intended to enable the user to add more items to the list. The knowledge base is query only, so this feature isn't needed.



Notice that removing this icon causes an error in the header due to the way in which the location and widths of the other items are calculated. You saw this earlier with the Inventory Management screen, and the solution is the same, as described in the following steps.

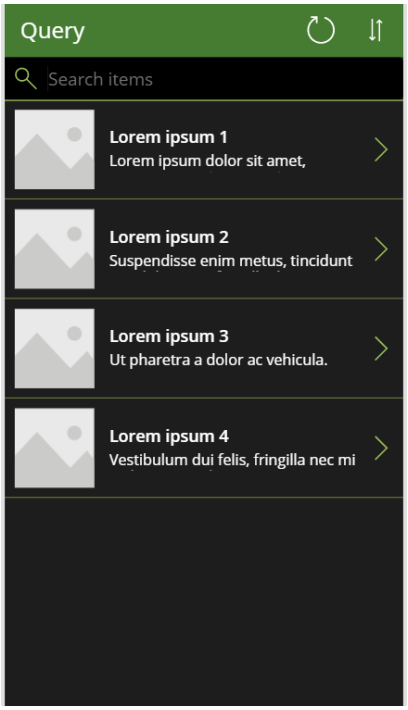
4. In the **Tree view** pane, scroll down to the **Screen1** screen, and select the **LblAppName3** control. Change the **Width** property to the formula **Parent.Width - LblAppName3.X - IconSortUpDown2.Width - IconRefresh2.Width**.



5. In the **Tree view** pane, select the **IconSortUpDown2** control. Modify the **X** property to the formula **Parent.Width - IconSortUpDown2.Width**.

6. In the **Tree view** pane, select the **IconRefresh2** control. Modify the **X** property to **Parent.Width - IconSortUpDown2.Width - IconRefresh2.Width**. This should resolve all the errors with the screen.

7. Select the **File** menu in the Power Apps Studio toolbar, and then select **Save**.
8. In the **Version note** box, enter the text **Added Knowledgebase UI**, and then select **Save**.
9. Return to the **Home** screen and press **F5** to preview the new screen. It should look like this:



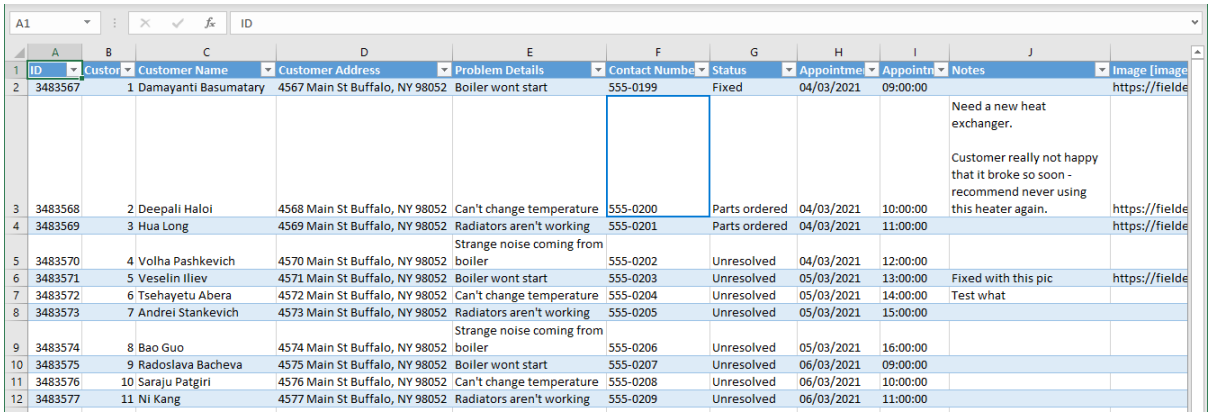
   Note that if you press the **>** icon by any of the dummy entries, the details functionality doesn't currently work. You'll address this later when you integrate Azure Cognitive Search into the app.

10. Close the preview window and return to Power Apps Studio.

ITEM 3: FIELD SCHEDULING AND NOTES

Maria works with Malik, the office receptionist, to design the interface for the field scheduling and appointments part of the app. Malik provides an Excel spreadsheet with some sample data that Maria can use to build the appointments screen. The spreadsheet contains a table with the following columns:

- ID (the appointment ID)
- Customer ID (a unique identifier for the customer)
- Customer Name
- Customer Address
- Problem Details (a text description of the problem the customer is experiencing)
- Contact Number
- Status
- Appointment Date
- Appointment Time
- Notes (a text description with any notes added by the engineer)
- Image (a photograph of the appliance, either in working condition after repair, or as a supplementary picture for the engineer's notes)



ID	Customer	Customer Name	Customer Address	Problem Details	Contact Number	Status	Appointment	Appointment	Notes	Image [image
3483567	1	Damayanti Basumatary	4567 Main St Buffalo, NY 98052	Boiler wont start	555-0199	Fixed	04/03/2021	09:00:00		https://fielde
3483568	2	Deepali Haloi	4568 Main St Buffalo, NY 98052	Can't change temperature	555-0200	Parts ordered	04/03/2021	10:00:00	Need a new heat exchanger. Customer really not happy that it broke so soon - recommend never using this heater again.	https://fielde
3483569	3	Hua Long	4569 Main St Buffalo, NY 98052	Radiators aren't working	555-0201	Parts ordered	04/03/2021	11:00:00		https://fielde
3483570	4	Volha Pashkevich	4570 Main St Buffalo, NY 98052	Strange noise coming from boiler	555-0202	Unresolved	04/03/2021	12:00:00		
3483571	5	Veselin Iliev	4571 Main St Buffalo, NY 98052	Boiler wont start	555-0203	Unresolved	05/03/2021	13:00:00	Fixed with this pic	https://fielde
3483572	6	Tsehayetu Abera	4572 Main St Buffalo, NY 98052	Can't change temperature	555-0204	Unresolved	05/03/2021	14:00:00	Test what	
3483573	7	Andrei Stankevich	4573 Main St Buffalo, NY 98052	Radiators aren't working	555-0205	Unresolved	05/03/2021	15:00:00		
3483574	8	Bao Guo	4574 Main St Buffalo, NY 98052	Strange noise coming from boiler	555-0206	Unresolved	05/03/2021	16:00:00		
3483575	9	Radoslava Bacheva	4575 Main St Buffalo, NY 98052	Boiler wont start	555-0207	Unresolved	06/03/2021	09:00:00		
3483576	10	Saraju Patgiri	4576 Main St Buffalo, NY 98052	Can't change temperature	555-0208	Unresolved	06/03/2021	10:00:00		
3483577	11	Ni Kang	4577 Main St Buffalo, NY 98052	Radiators aren't working	555-0209	Unresolved	06/03/2021	11:00:00		

Note: As with the Field Inventory Management data, this spreadsheet represents a denormalized view of the data. In the existing appointments system, this data is stored in separate tables holding appointment data and customer data.

Maria stores this spreadsheet in her OneDrive account with the name **Appointments.xlsx**. Having remembered that she previously used the default name for the table in the spreadsheet, and had to change the title in the various screens that were generated, she also names the table in the spreadsheet to **Appointments**.

Note: This spreadsheet is available in the **Assets** folder of the Git repository for this guide.

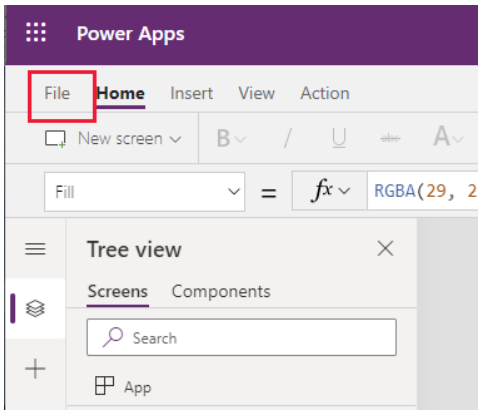
Maria wants to build the appointments part of the app directly from the Excel spreadsheet. She decides to follow a similar approach to that of the Field Inventory Management functionality, except that this time the engineer will be allowed to create and edit appointments.

Maria decides to build the appointments screens, initially as a separate app. This way, she can use Power Apps Studio to generate much of the app automatically. Power Apps Studio doesn't currently let you generate additional screens from a data connection in an existing app. When Maria has created and tested the screens, she will copy them to the Field Inventory and Knowledgebase app.

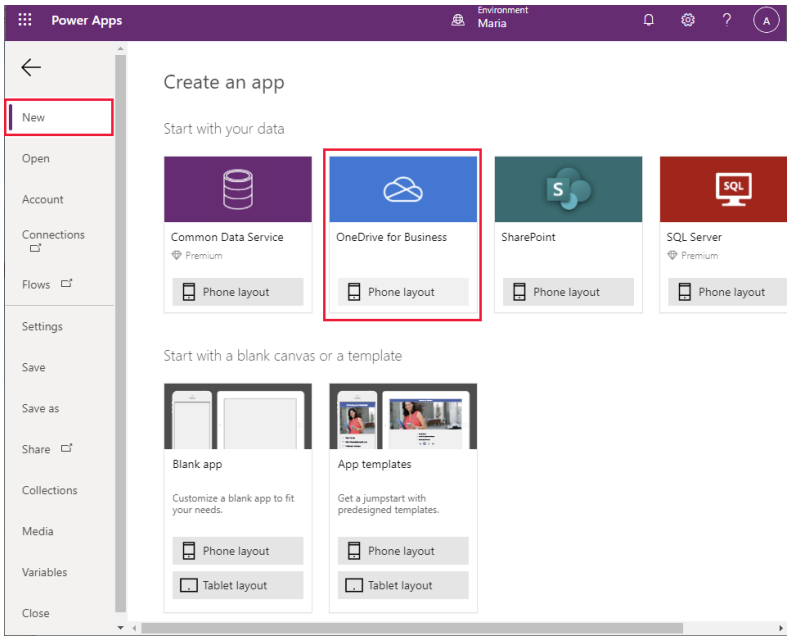
Note: An alternative approach is to add the **Appointments** table in the Excel spreadsheet as a second data source to the existing app and then hand-craft the screens for appointments. Maria opted to generate the new screens from the spreadsheet and copy the screens; she is currently more familiar with the concepts of *copy and paste* than building screens manually, although she will gradually learn how to create screens from scratch as the process of building this app progresses.

You can follow these steps to create the Appointments app:

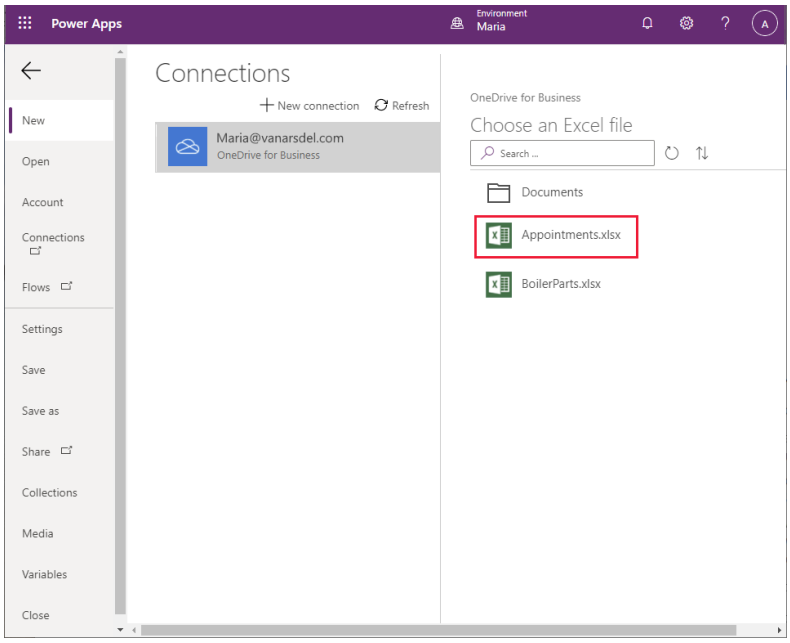
1. In Azure Power Apps Studio, select the **File** menu in the toolbar.



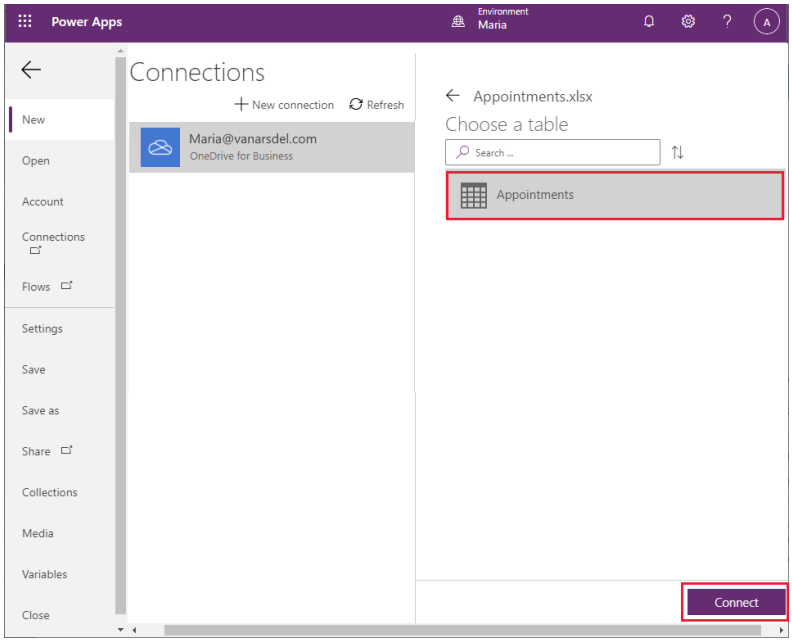
2. On the **File** page, in the left pane, select **New**. In the main pane, select **Phone layout** in the **OneDrive for Business** box:



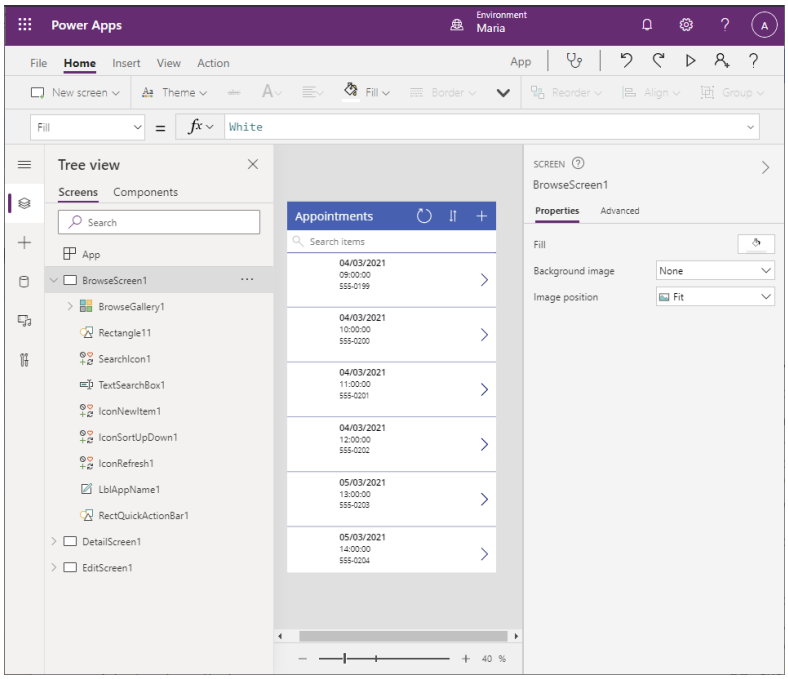
3. In the **Connections** pane, select the **Appointments.xlsx** spreadsheet:



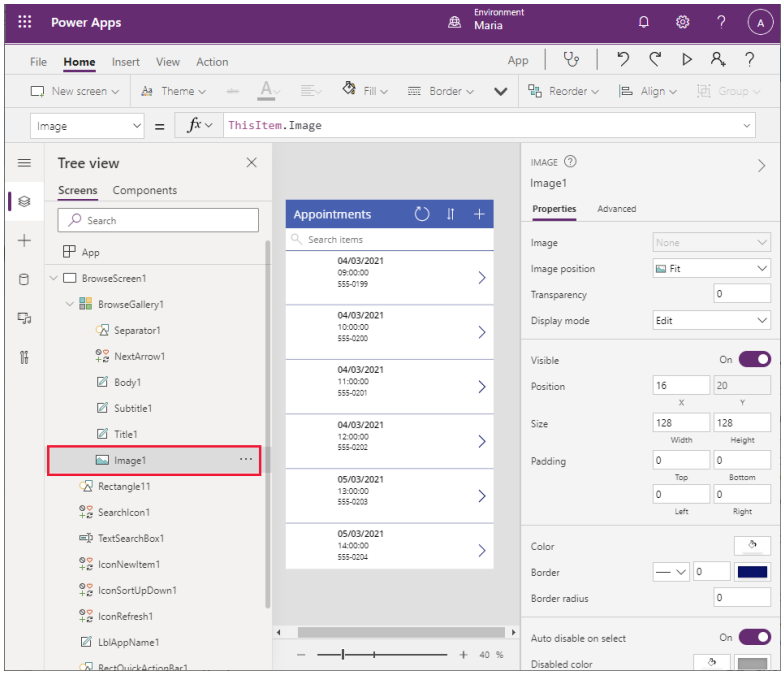
4. Select the **Appointments** table in the Excel Sheet, and then select **Connect**:



5. Wait while the app is generated. When the new app appears, it'll contain a **Browse** screen, a **Details** screen, and an **Edit** screen, using the default theme:



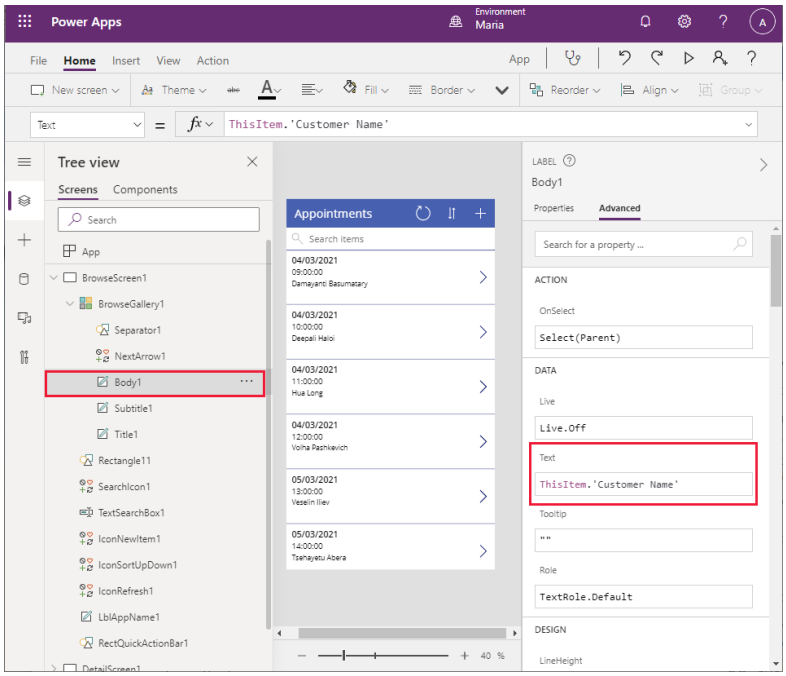
6. In the **Tree view** pane, under the **BrowseGallery1** control in the **BrowseScreen1** screen, select the **Image1** control and press **Delete**. The **Browse** screen should just list appointments and not any images associated with them:



Notice that removing the **Image1** control causes several errors on the screen because the widths and location of the **Title1** control reference the **Image** control. You'll fix these problems in the next step.

7. In the **Tree view** pane, select the **Title1** control under **BrowseGallery1**. Change the value in the **X** property to **16**. Change the formula in the **Width** property to **Parent.TemplateWidth – 104**. This should resolve the errors for the screen.

8. In the **Tree view** pane, select the **Body1** control under **BrowseGallery1**. This control currently displays the contact telephone details for the customer. Change the value in the **Text** property to **ThisItem.'Customer Name'** (including the single quotes).



The details on the **Browse** screen name now display the customer name.

9. In the **Tree view** pane, select the **BrowseGallery1** control. Using the formula bar examine the expression in the **Items** property. The control searches for appointments using the appointment date, time, and contact number. Change this formula to search the customer name rather than the contact number:

```
SortByColumns(Search([@Appointments], TextSearchBox1.Text,
"Appointment_x0020_Date","Appointment_x0020_Time","Customer_x0020_Name")
, "Appointment_x0020_Date", If(SortDescending1, Descending, Ascending)).
```

   Notice that the appointments are ordered by date and then time (the first two fields displayed).

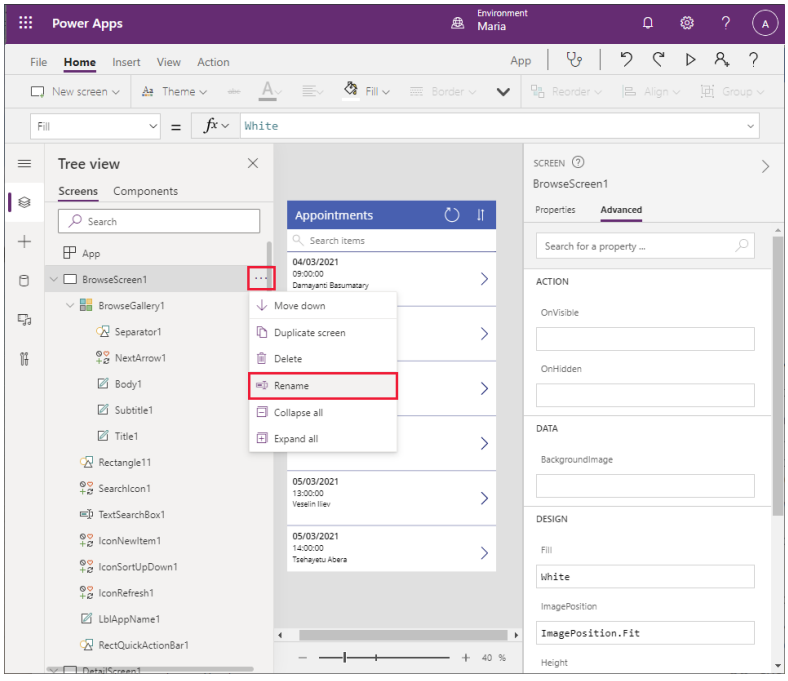
10. In the **Tree view** pane, delete the **IconNewItem1** icon. Only on-premises staff can book new appointments for engineers and technicians. Notice that this action results in errors in the form because the width and position of other controls in the header reference the icon you just removed.

11. In the **Tree view** pane, select the **LblAppName1** control. Change the formula for the **Width** property. to **Parent.Width - Self.X - IconSortUpDown1.Width - IconRefresh1.Width**.

12. In the **Tree view** pane, select the **IconRefresh1** icon. Change the value for the **X** property to **Parent.Width - IconSortUpDown1.Width - Self.Width**.

13. In the **Tree view** pane, select the **iconSortUpDown1** icon. Change the value for the **X** property to **Parent.Width - Self.Width**.

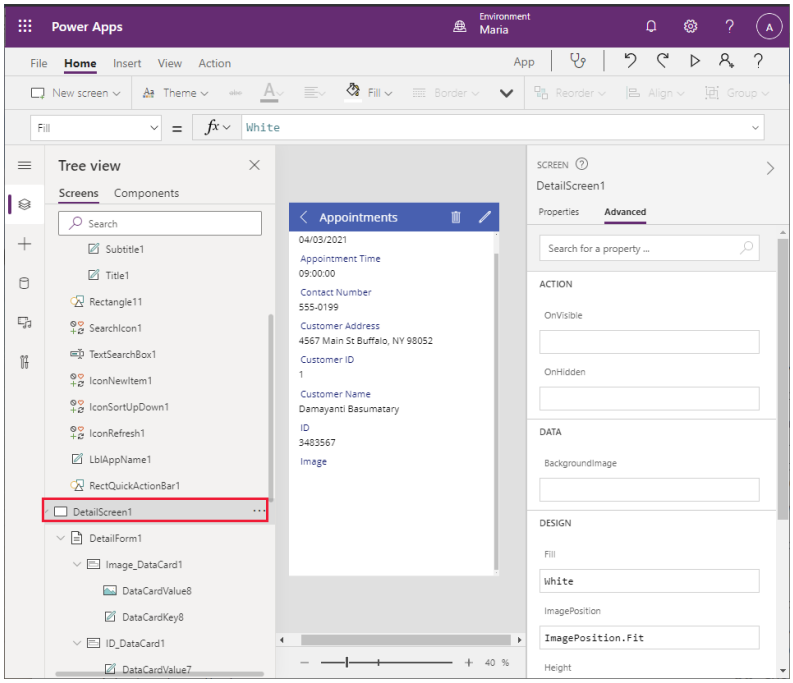
14. In the **Tree view** pane, select the **BrowseScreen** screen, and then select the ellipsis button. On the drop-down menu that appears, select **Rename** and change the name of the screen to **BrowseAppointments**.



15. Using the same technique, change the name of the **BrowseGallery1** control to **BrowseAppointmentsGallery**.

That completes the **Browse** screen. You can now turn your attention to the **Details** screen.

1. In the **Tree view** pane, scroll down to the **DetailsScreen1** screen. You can see that the details are presented in alphabetical order of the field names, and not all the useful bits of information, such as the **Notes** field, are displayed:

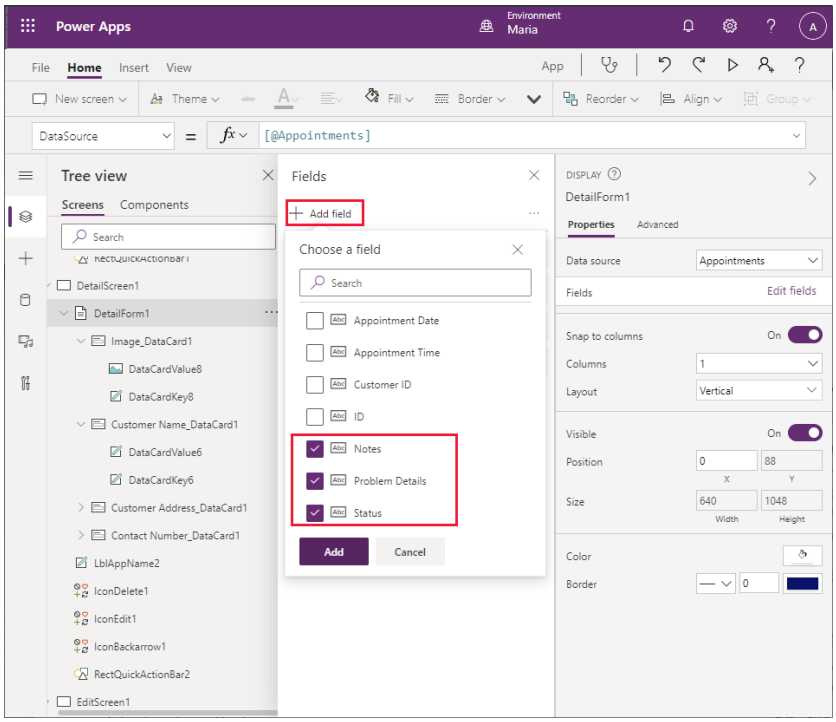


2. In the **Tree view** pane, select the **DetailForm1** control. In the right pane, on the **Properties** tab, in the **Fields** field, select **Edit fields.** In the middle pane, select each of the following fields, and then press **Delete**:

   - Appointment Date
   - Appointment Time
   - Customer ID
   - ID

3. In the middle pane, select **+ Add field**, and add the following fields:

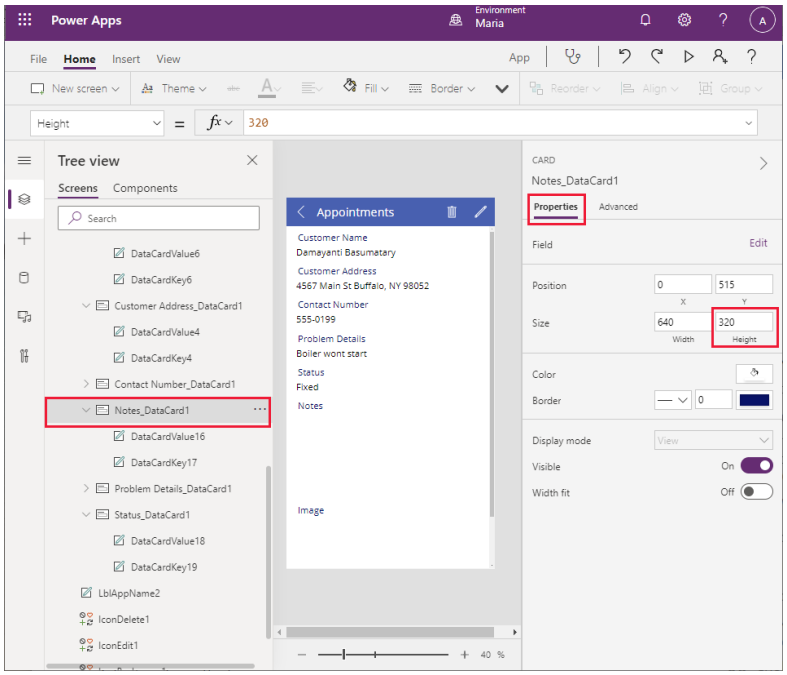
   - Notes
   - Problem Details
   - Status



4. In the middle pane, click and drag each field so that they're displayed in the following order, from top to bottom:

    - Customer Name
    - Customer Address
    - Contact Number
    - Problem Details
    - Status
    - Notes
    - Image

5. In the **Tree view** pane, select the **Notes_DataCard1** control. Change the **Height** property to **320**:



6. In the **Tree view** pane, delete the **IconDelete1** icon. Engineers shouldn't be able to remove appointments from the system.

7. Select the **LblAppName2** control. Update the **Width** property of this control to **Parent.Width - Self.X - IconEdit1.Width**.

8. Using the technique described earlier, change the name of **DetailsScreen1** to **AppointmentDetails**.

The final screen to look at for now is the **Edit** screen.

1. In the **Tree view** pane, scroll down and select the **EditScreen1** screen.

2. In the **Tree view** pane, select the **EditForm1** control in the **EditScreen1** screen. In the right pane, on the **Properties** tab, in the **Fields** field, select **Edit fields.**

3. Remove the following fields from the **EditForm1** control:

    - Customer Address
    - ID
    - Customer ID
    - Appointment Date
    - Appointment Time

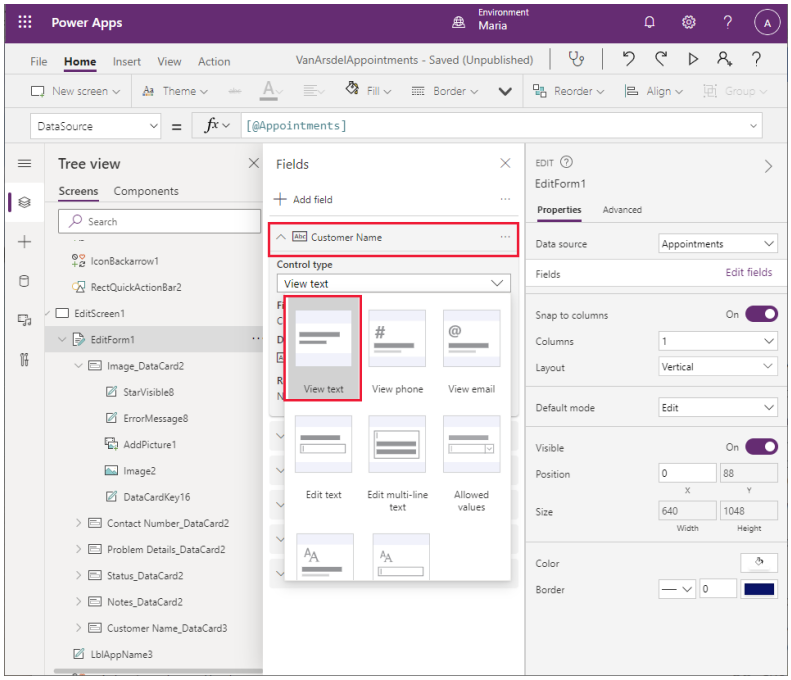
4. Add the following fields to the form:

- Problem Details
- Status
- Notes

5. Reorganize the fields so that they're in the following sequence:

   - Contact Name
   - Customer Number
   - Problem Details
   - Status
   - Notes
   - Image

6. Select the **Customer Name** field and click the drop-down arrow to view its properties. Change the **Control type** to **View text**. This change makes the control read-only; an engineer shouldn't be able to change the customer's name, although it is useful to see it on the **Edit** screen:

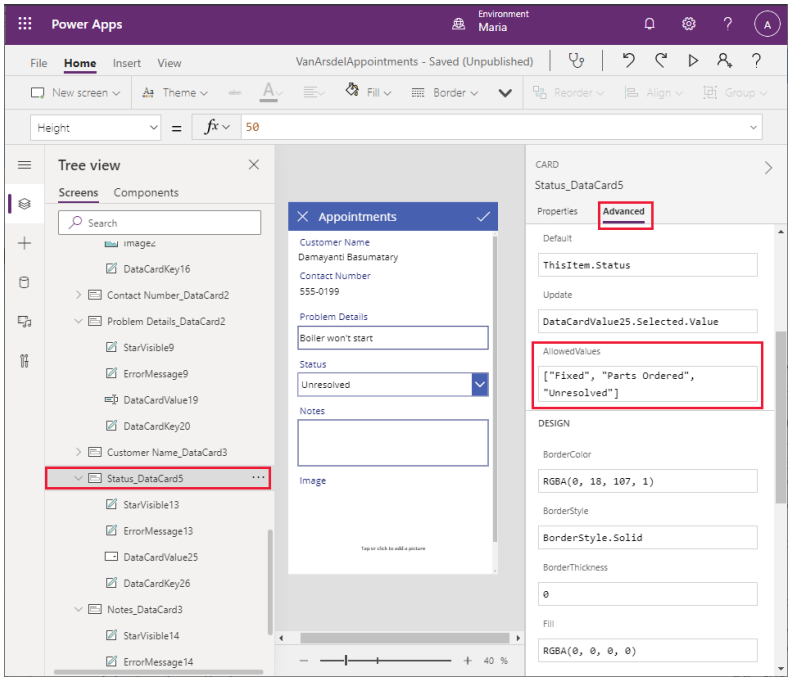


7. Select the **Contact Number** field and click the drop-down arrow to view its properties. Change the **Control type** to **View text**. This field should also be read only.

8. Select the **Notes** field, click the drop-down arrow to view its properties, and change the **Control type** to **Edit multi-line text**. This setting enables the engineer to add detailed notes that can span several lines.

9. Select the **Status** field, click the drop-down arrow to view its properties, and change the **Control type** to **Allowed Values**.

10. In the **Tree view** pane, select the **Status_DataCard5** control. In the right pane, on the **Properties** tab, select **Unlock to change properties**. Scroll down to the **AllowedValues** property, and change the text to **["Fixed", "Parts Ordered", "Unresolved"]** (including the square brackets). The field engineer can only set the **Status** to one of these defined values.



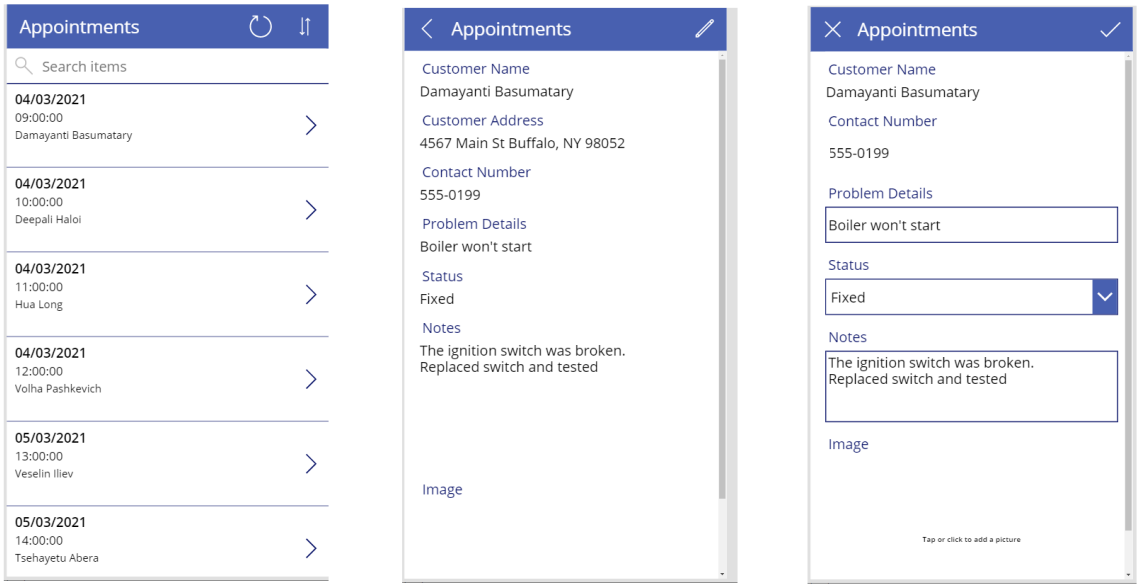
11. In the **Tree view** pane, rename **EditScreen1** as **EditAppointment**.

You can now save and test the app.

1. Select the **File** menu in the Power Apps Studio toolbar, and then select **Save as**.
2. Under **Save as**, select **The cloud** as the destination, name the app **VanArsdelAppointments**, and then select **Save**.
3. Press the back arrow icon in the Power Apps Studio toolbar to return to the **Home** screen.
4. Press **F5** to preview the app. On the **Appointments** page, select the **>** symbol for any appointment. The **Details** screen for the appointment should appear. Select the **Edit** button in the header to update the appointment. Verify the following:

    - The customer name and contact number fields are read-only.
    - The status field is limited to the values in the drop-down list box.
    - You can enter notes over several lines.
    - You can upload an image file to the image field.

> **Note**: An enhancement that you will add later allows you to take a picture with your phone from within the app, and add it to the image field.

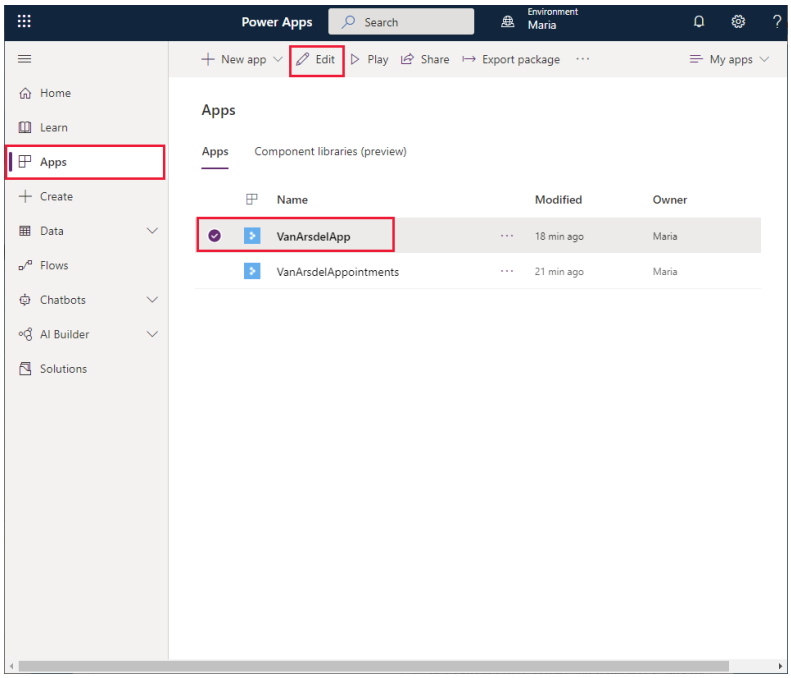


COMBINING THE SCREENS INTO A SINGLE APP

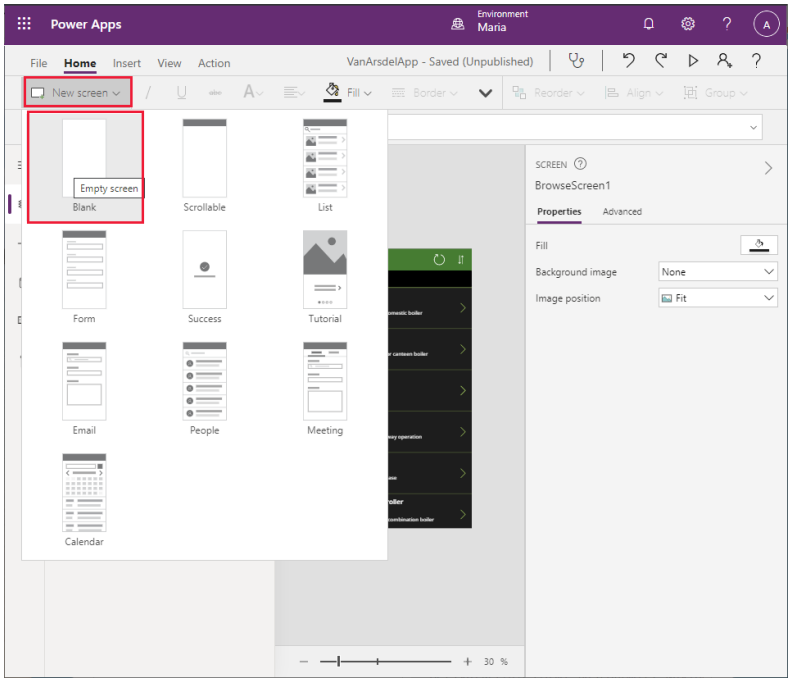
Maria has built two apps, but she wants to combine them into a single app. To do this, she copies the screens for the Appointments app into the Field Inventory Management and Knowledgebase app, as follows:

1. Open a new browser window and sign in to Power Apps Studio with your account details.

2. In the left pane, select **Apps**, select the **VanArdselApp** app, and then select **Edit.**



3. In the toolbar, select **New screen**, and then select **Blank**. This action will add a new screen to the app into which you will copy the controls for the **Browse** screen for the **VanArsdelAppointments** app.

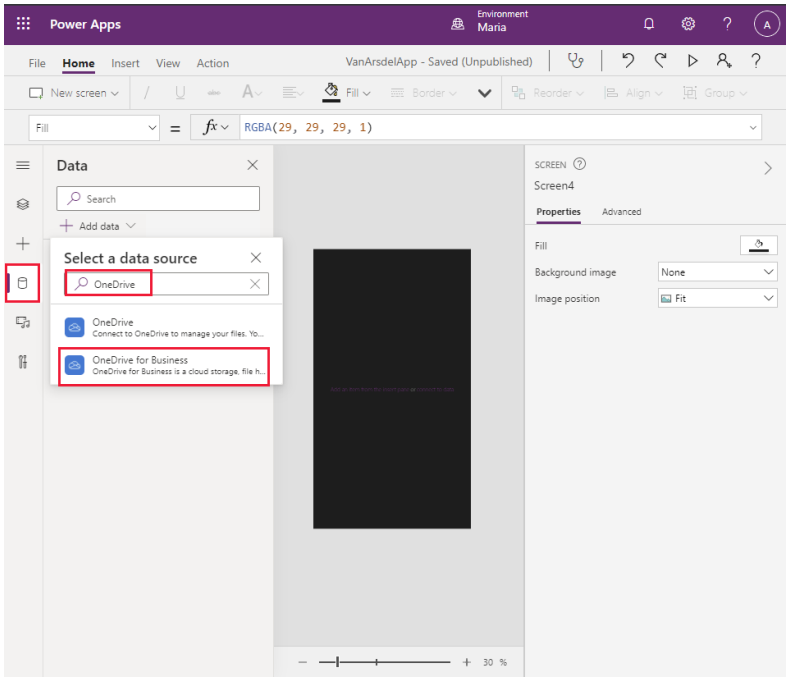


4. The new screen will be called **Screen2**. In the **Tree view** pane, rename it as **BrowseAppointments**.

5. Repeat the previous step twice more, to add two more blank screens (**Screen3** and **Screen4**).

6. Rename **Screen3** as **AppointmentDetails**, and rename **Screen4** as **AppointmentDetails**.

7. In the left toolbar of Power Apps Studio, select the **Data** icon. In the **Data** pane, select **Add data**. In the **Select a data source** drop-down list box, in the **Search** field, type **OneDrive**, and select **OneDrive for Business**.



8. Select the **Appointments.xlsx** Excel spreadsheet, and the **Appointments** table, and then select **Connect**.

9. Switch to the browser window with the **VanArsdelAppointments** app.

10. In the toolbar, select **Theme** (you might have to click the drop-down arrow displaying more items for the toolbar), and then select the **Forest** theme. This is the same theme used by the **VanArsdel** app.
11. In the left toolbar, select the **Tree view** icon, select the **BrowseAppointments** screen, and then press **CTRL A**. This action selects all the controls in the screen.
12. Press **CTRL C** to copy these controls to the clipboard.
13. Return to the browser window with the **VanArsdelApp** app.
14. In the left toolbar, select the **Tree view** icon, and then select the **BrowseAppointments** screen.
15. Press **CTRL V** to paste the controls into the screen.

> **Note:** Sometimes the screen header appears slightly too low down. To fix this problem, select the **IconSortUpDOwn1_1**, **IconRefresh1_1**, **LblAppName1_1**, and **RectQuickActionBar1_1** controls in the **Tree view** pane (use **Shift-click** to select more than one control at a time), and then use the mouse or arrow keys to move them up in the design view pane.

16. Switch back to the browser window with the **VanArsdelAppointments** app, then select and copy the controls in the **AppointmentDetails** screen to the clipboard (**CTRL A** followed by **CTRL C**).
17. Return to the browser window with the **VanArsdelApp** app, select the **AppointmentDetails** screen, and paste the controls (**CTRL V**). Adjust the position of the controls in the screen header if necessary.

> **Note**: You will see an error reported in the header of the **AppointmentDetails** screen. This error occurs because the screen references controls in the **EditAppointment** screen which haven't been copied yet. The next steps should resolve this error.

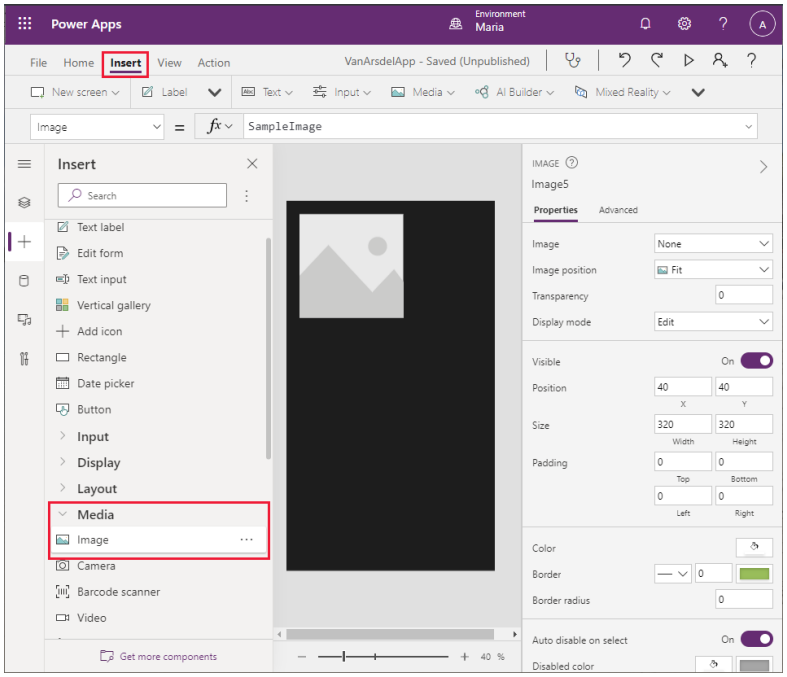
18. Switch back to the browser window with the **VanArsdelAppointments** app, then select and copy the controls in the **EditAppointment** screen to the clipboard.
19. Return to the browser window with the **VanArsdelApp** app, select the **EditAppointment** screen, and paste the controls. Again, move the controls in the screen header if necessary.
20. Select the **AppointmentDetails** screen in the **Tree view** menu. Verify that the error indicated previously has now disappeared.
21. In the **Tree view** menu, select the **BrowseScreen1** screen. Change the name of this screen to **BrowseParts**.
22. Change the name of the **DetailsScreen1** screen to **PartDetails**.
23. Change the name of the **Screen1** screen to **Knowledgebase**.

> **Note**: It is good practice to rename screens to reflect their function rather than using the default names generated by Power Apps Studio. This is especially important if an app contains several screens. It can help to avoid confusion later if the app is modified by another developer.

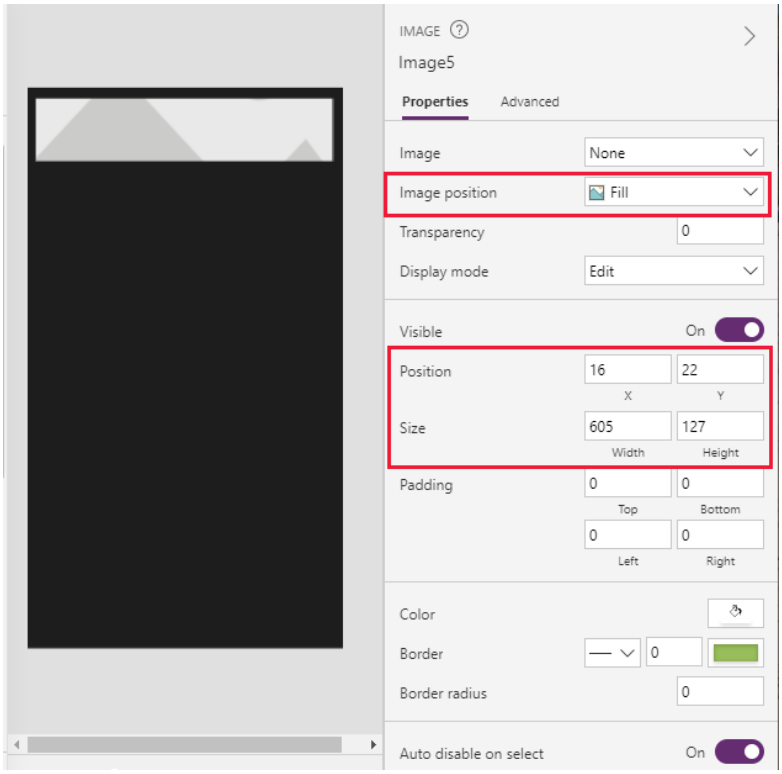
ADDING A HOME SCREEN TO THE APP

The final stage is to add a **Home** screen to the app. The **Home** screen will enable the engineer to move between the different parts of the app (inventory management, knowledge base, and appointments).

1. In the **VanArsdelApp** app, in the toolbar, select **New screen**, and then select **Blank**.
2. In the **Tree view** pane, change the name of the screen (**Screen2**) to **Home**.
3. In the toolbar, select **Insert**. In the list of controls, expand **Media**, and select the **Image** control. The control will be added to the screen:

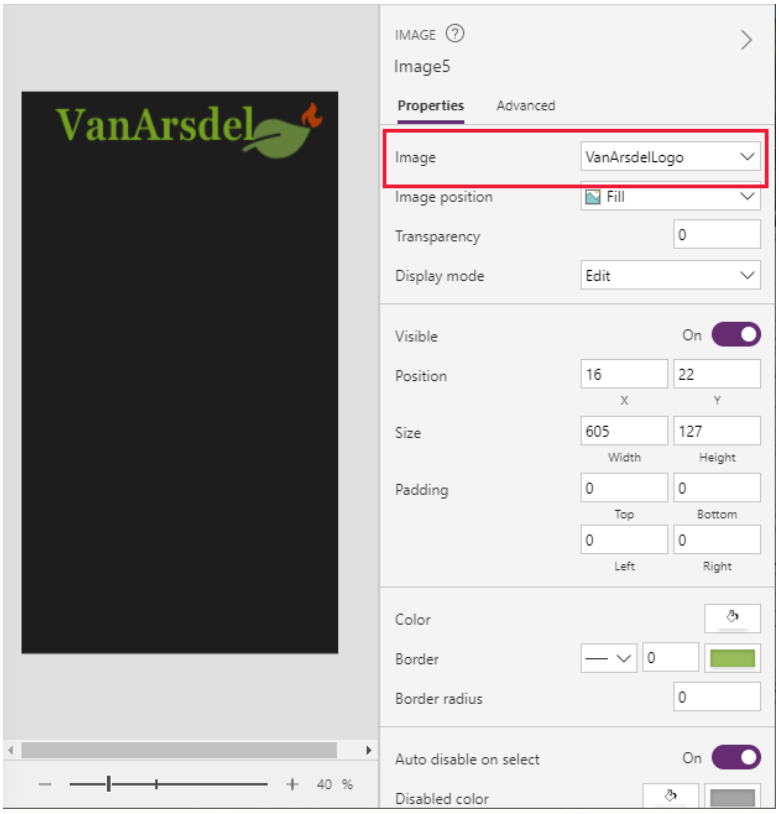


4. Set the **X** position of the control to **16**, and the **Y** position to **22**. Change the **Width** to **605**, and the **Height** to **127**. Change the **Image position** to **Fill**:

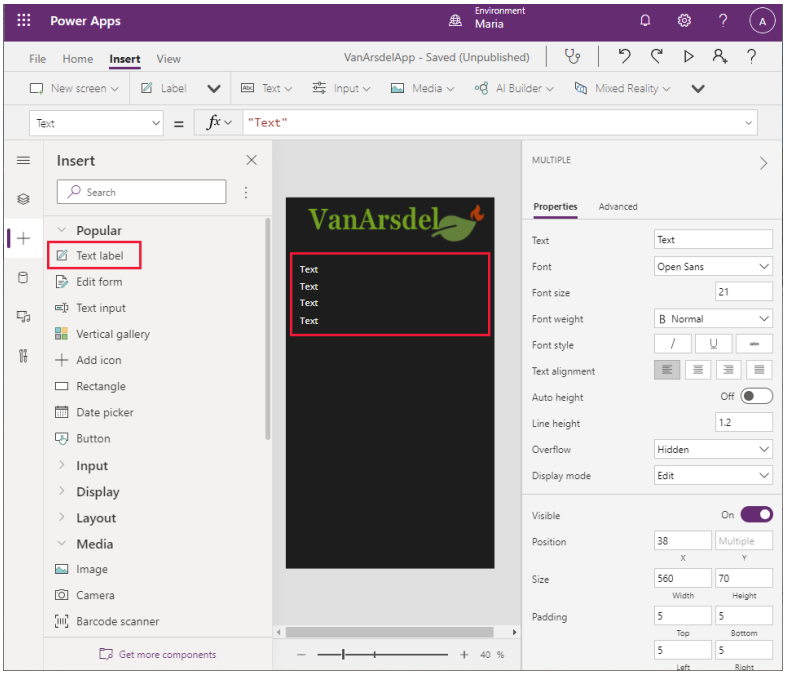


5. Still on the **Properties** tab, in the **Image** drop-down list box, select **+ Add an image file**, and upload the **VanArsdelLogo.png** image to the control:

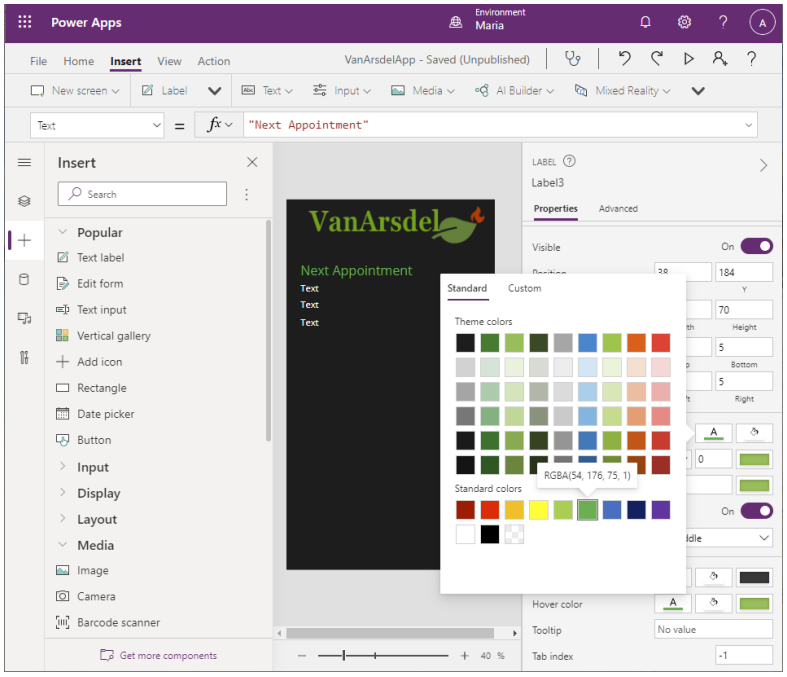
> **Note**: The image file is available in the **Assets** folder in the Git repository for this guide.



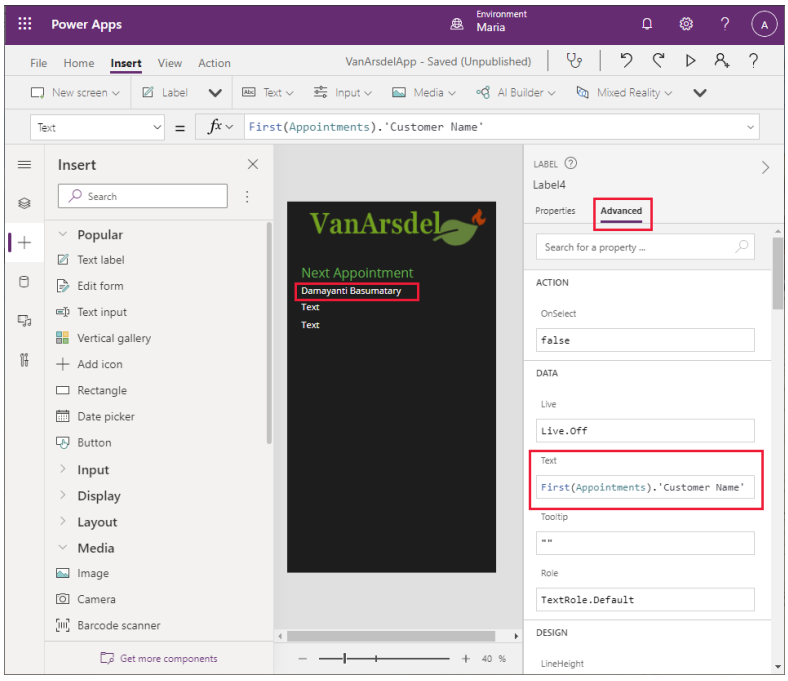
6. From the list of controls, add four **Text label** controls to the form and position them as shown in the image below:



7. Select the uppermost **Text label** control. In the right pane, on the **Properties** tab, set the **Text** property to **Next Appointment**. Set the **Font Size** to 30, and use the color picker to set the font color to Green (to match the logo):



8. Select the second **Text label** control. Change the value of the **Text** property to **First(Appointments).'Customer Name'** (including the quotes). This formula retrieves the customer name from the first row in the appointments table:



> **Note**: Currently, this formula just acts as a placeholder. You will modify this label later to retrieve the next appointment for the engineer rather than always displaying the first one.

9. Select the third **Text label** control and set the **Text** property to **First(Appointments).'Appointment Date**.

10. Set the **Text** property of the fourth label control to **First(Appointments).'Appointment Time**. Set the **Font size** property to 30.

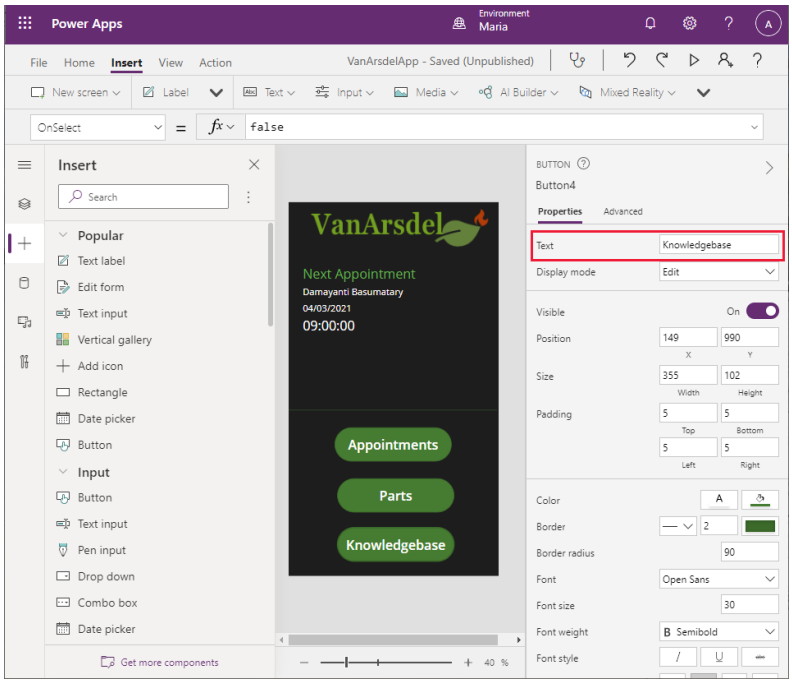
11. From the list of controls, add a **Rectangle** control. Set the following properties for this control:

    - Display mode: **View**

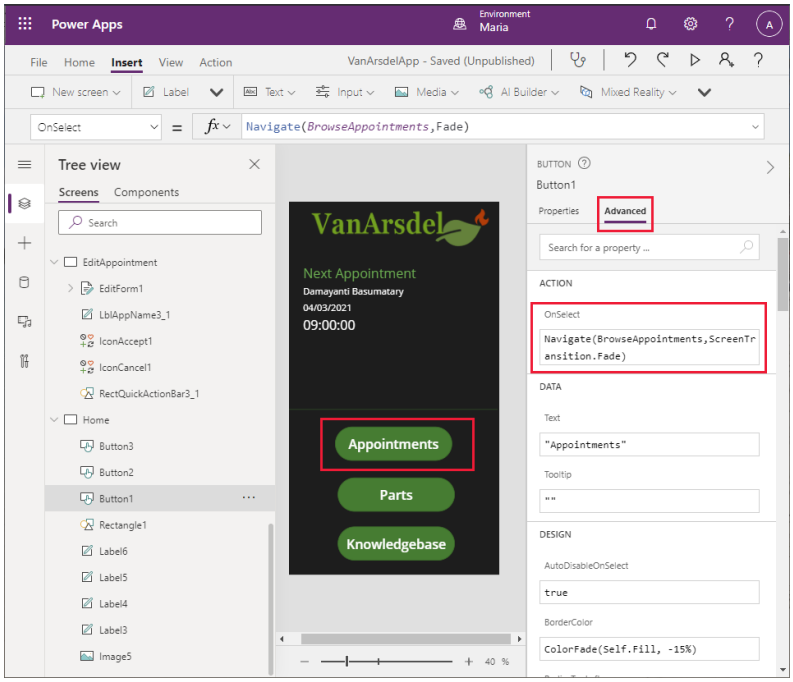
- X: **0**
- Y: **632**
- Width: **635**
- Height: **1**

This control acts as a visual separator across the middle of the screen.

12. Add three **Button** controls to the screen, arranged vertically and spaced evenly below the separator. Set the **Text** property for the top button to **Appointments**, the **Text** property for the middle button to **Parts**, and the **Text** property for the lower button to **Knowledgebase**.



13. Select the **Appointments** button. Change the expression in the **OnSelect** action to the formula **Navigate(BrowseAppointments ,ScreenTransition.Fade)**. This action switches the display to the appointments screen when the user selects the button:



14. Set the **OnSelect** action for the **Parts** button to **Navigate(BrowseParts, ScreenTransition.Fade)**.

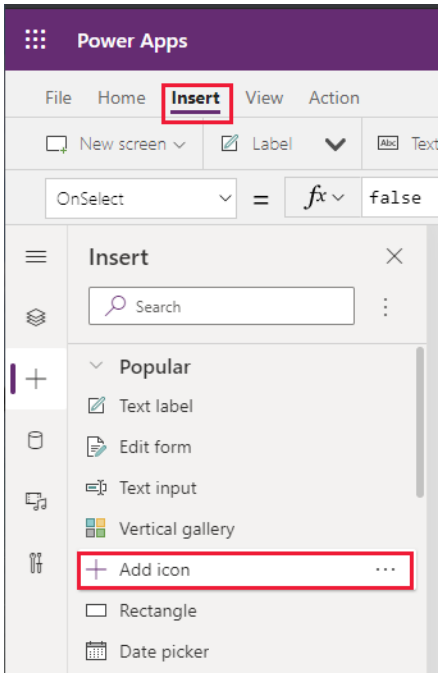
15. Set the **OnSelect** action for the **Knowledgebase** button to **Navigate(Knowledgebase, ScreenTransition.Fade)**.

As well as navigating from the **Home** screen to the other screens in the system, the appointments, parts, and knowledge base screens need a way to enable the user to return to the **Home** screen. Maria decides to add *back* buttons to these screens.

1. In the **Tree view** pane, select the **BrowseParts** screen.

2. Select the **RectQuickActionBar1** control to give it the focus.

3. Select the **Insert** menu, and select **Add icon.** Move the icon to the left of the **RectQuickActionBar1** control. Note that the icon will obscure part of the **Browse Parts** label.

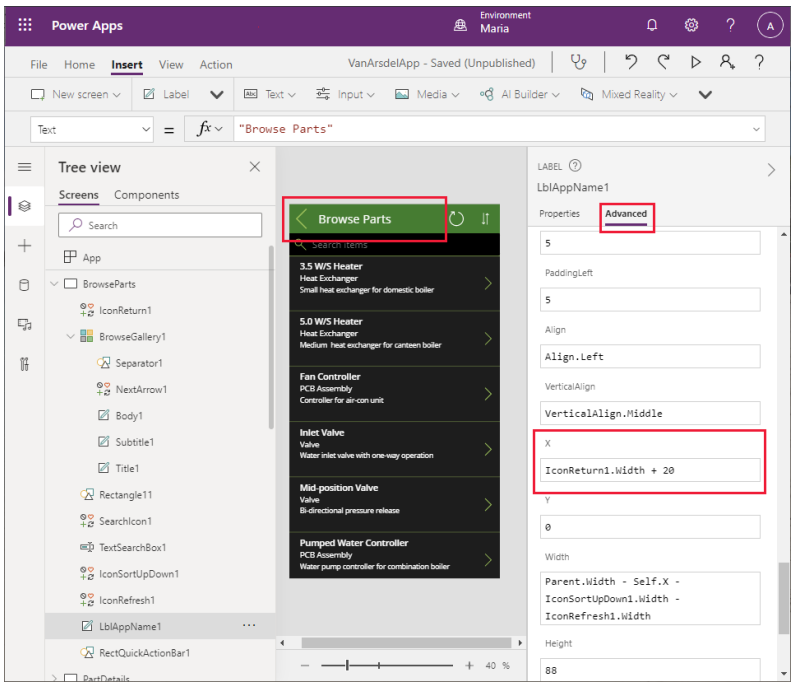


4. In the **Tree view** menu, change the name of the new icon control to **IconReturn1**.

5. Change the formula for the **OnSelect** action to the expression **Back(ScreenTransition.Fade)**. The **Back** function navigates the user to the previous screen they visited.

6. On the **Properties** tab, change the Icon property to **< Left**.

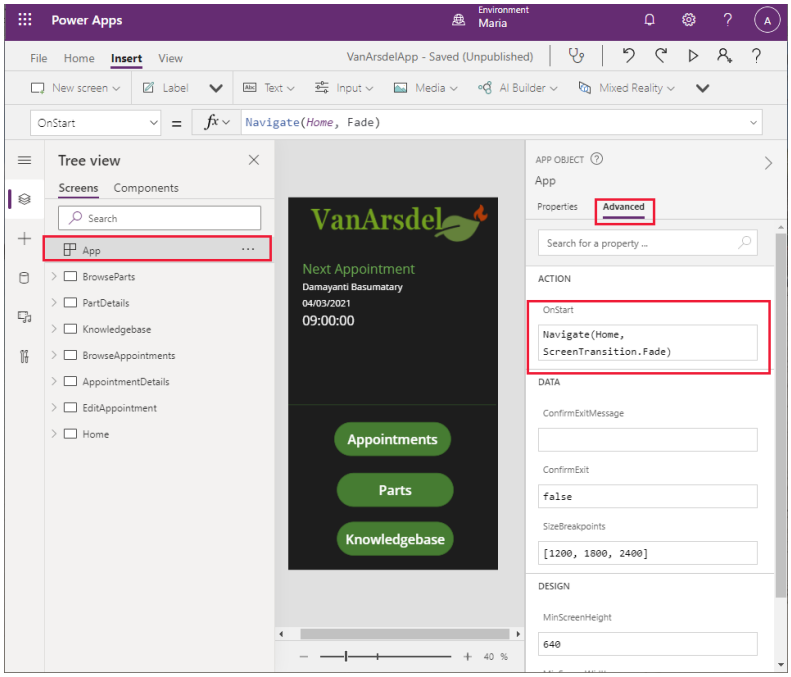
7. In the screen header, select the **Browse Parts** label. Change the **X** property to **IconReturn1.Width + 20.** The **Browse parts** label should no longer be partially obscured.



8. Following the process described in steps 16 to 22, add an icon named **IconReturn2** to the **RectQuickActionBar3** control in the **Knowledgebase** screen.

9. Similarly, add an icon named **IconReturn3** to the **RectQuickActionBar1_1** control in the **BrowseAppointments** screen.

10. In the **Tree view** pane, select the **App** object. Change the **OnStart** action property to the expression **Navigate(Home, ScreenTransition.Fade)**. This action ensures that the **Home** screen is displayed whenever the app starts:



Note: If you don't specify which screen should be displayed when the app starts, the screen that appears at the top of the **Tree view** pane will be used. To move a screen to the start of the list, right-click the screen in the **Tree view** pane and select **Move up** until it is at the top.

Finally, you can test the app.

1. On the **File** menu, on the **Save** tab, enter the text **Complete version with Home screen** in the **Version note** box, and select **Save**.
2. Select the back arrow icon to return to the **Home** screen and press **F5** to run the app.
3. Verify that the **Home** screen for the app appears.
4. Select **Appointments**. The appointments browser screen should appear.
5. Click the back arrow icon in the screen header to return to the **Home** screen.
6. Select **Parts**. The parts browser should appear.
7. Click the back arrow icon in the screen header to return to the **Home** screen.
8. Select **Knowledgebase**. The knowledge base query screen should appear.
9. Click the back arrow icon in the screen header to return to the **Home** screen.
10. Close the preview window and return to Power Apps Studio.

The prototype app is now complete.

CHAPTER 4: USING MICROSOFT DATAVERSE AS THE DATA SOURCE

Maria has built a prototype app using test data held in Excel spreadsheets. She can now consider how to connect the app to data sources that will provide real-world data. She has heard about Microsoft Dataverse as an option for doing this, but she wants to know more about it.

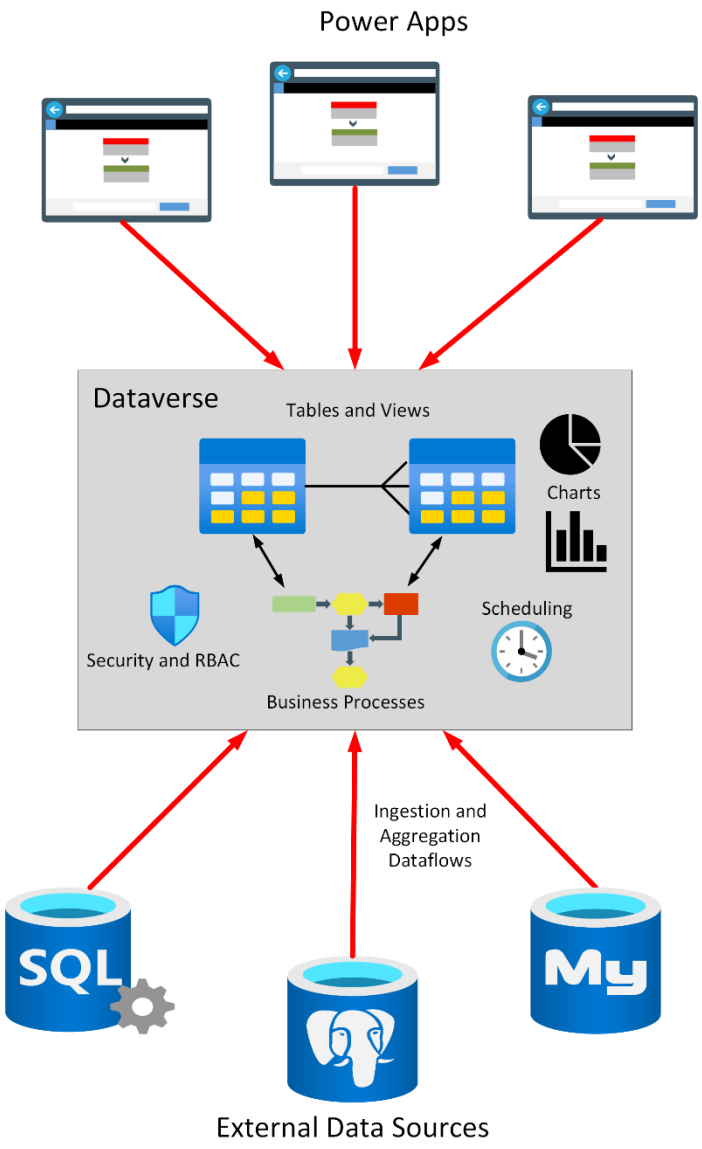
WHAT IS MICROSOFT DATAVERSE?

Microsoft Dataverse is a data store with a set of standard tables. It enables you to store business information, manage business rules, and define business dataflows. In many ways it acts like a database, except that it holds more than just data. You can use it to record elements of business logic for your solutions, and share this logic across Power Apps. Dataverse includes scheduling capabilities that enable you to automate processing and workflows. Additionally, you can add charts and associate them with your data; Power Apps can reference these charts directly from Dataverse.

Dataverse follows the *low-code* approach of Power Apps, enabling a business user to create business entities and workflows. Additionally, Dataverse is a scalable, reliable, and secure system, implemented in Azure. Azure Role-Based Access Control (RBAC) limits the type of access to different users in your organization; users can only see or manipulate the entities to which they are granted access.

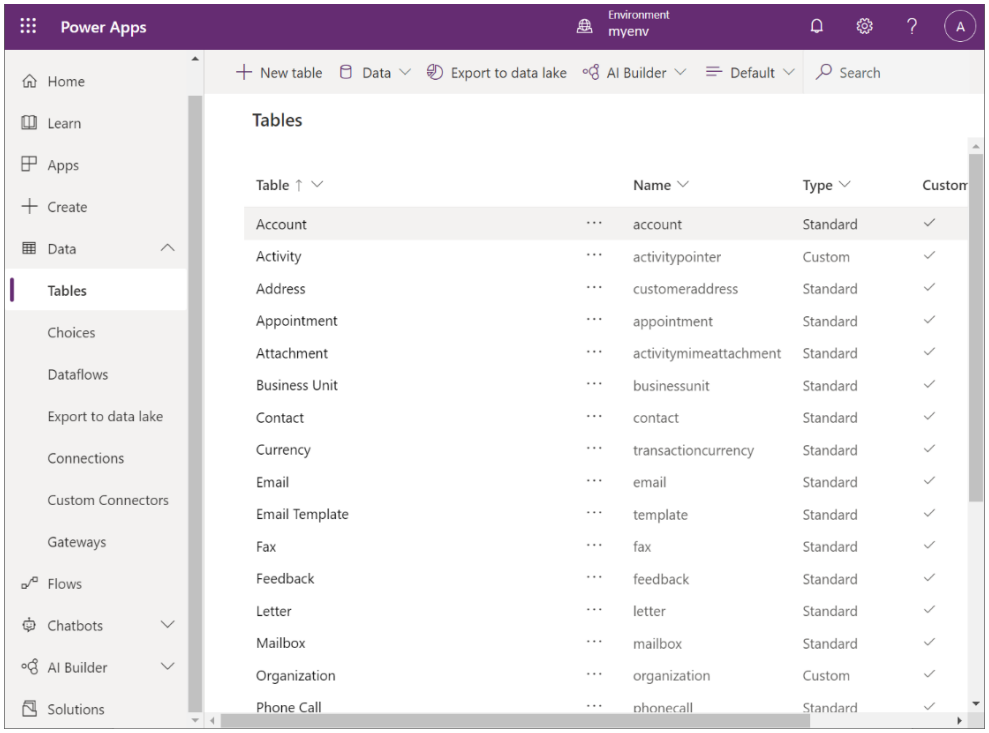
> **Note:** The definitions of Power Apps and users are also stored in Dataverse. Power Apps Studio uses this information for creating, editing, and publishing Power Apps.

Dataverse enables you to unify data held in disparate databases into a single repository. You can create dataflows that periodically ingest data held in one or more databases into the tables in Dataverse to create aggregated datasets.



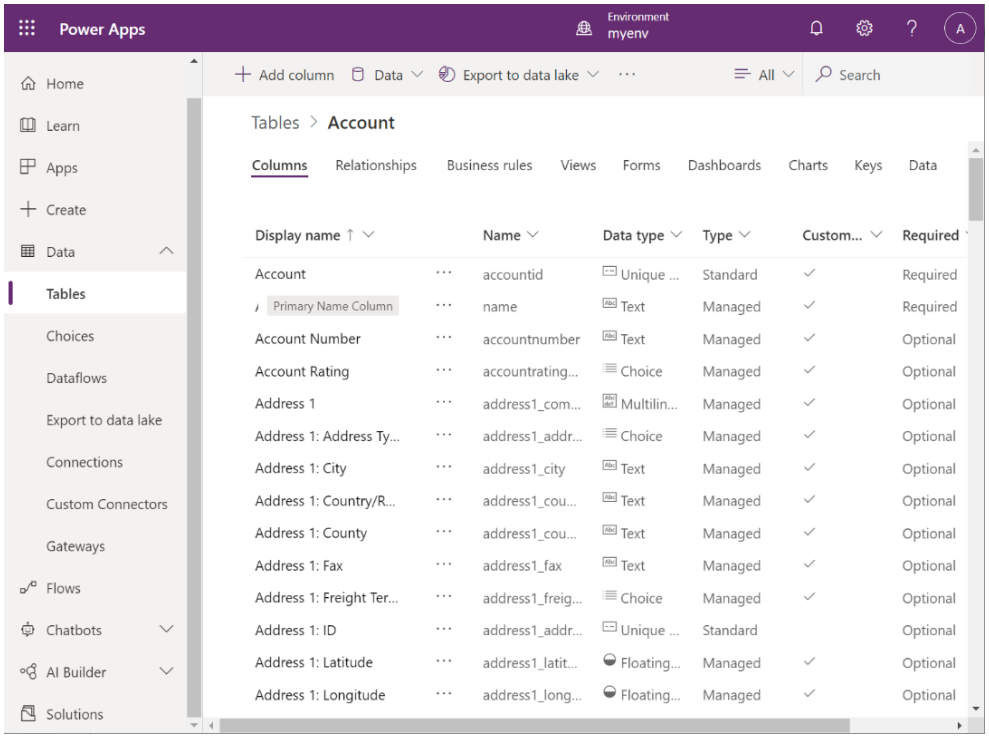
DEFINING ENTITIES AND RELATIONSHIPS IN DATAVERSE

Dataverse contains a collection of open-sourced, standardized, extensible data entities and relationships that Microsoft and its partners have published in the industry-wide *Open Data Initiative*. The data for these entities is stored in a set of tables. Dataverse defines entities for many common business objects, such as Accounts, Addresses, Contacts, Organizations, Teams, and Users. You can view the tables in Dataverse from the *Tables* tab under Data in Power Apps Studio. You can add your own custom tables to Dataverse if necessary, but it's good practice to use the existing tables wherever possible as this will help to ensure the portability of your Power Apps. Tables that are part of the default Dataverse have the *Type* designated as *Standard*, but your own tables are marked as *Custom*:



In Dataverse each table contains a default set of columns also defined by the Open Data Initiative. You can view the definition of a table using the *Edit* command for that table in the list of tables. You can extend a table with your own columns, but as before, it's good practice to use existing columns wherever possible. The example below shows the default definition of the *Account* table.

Note: You can modify the display name of tables and columns without changing their names. The display names are the default labels that appear on forms in Power Apps.



Dataverse supports a rich set of datatypes for columns, ranging from simple text and numeric values, through to abstractions with specified formatting constraints such as *Email*, *URL*, *Phone*, and *Ticker Symbol*. Other

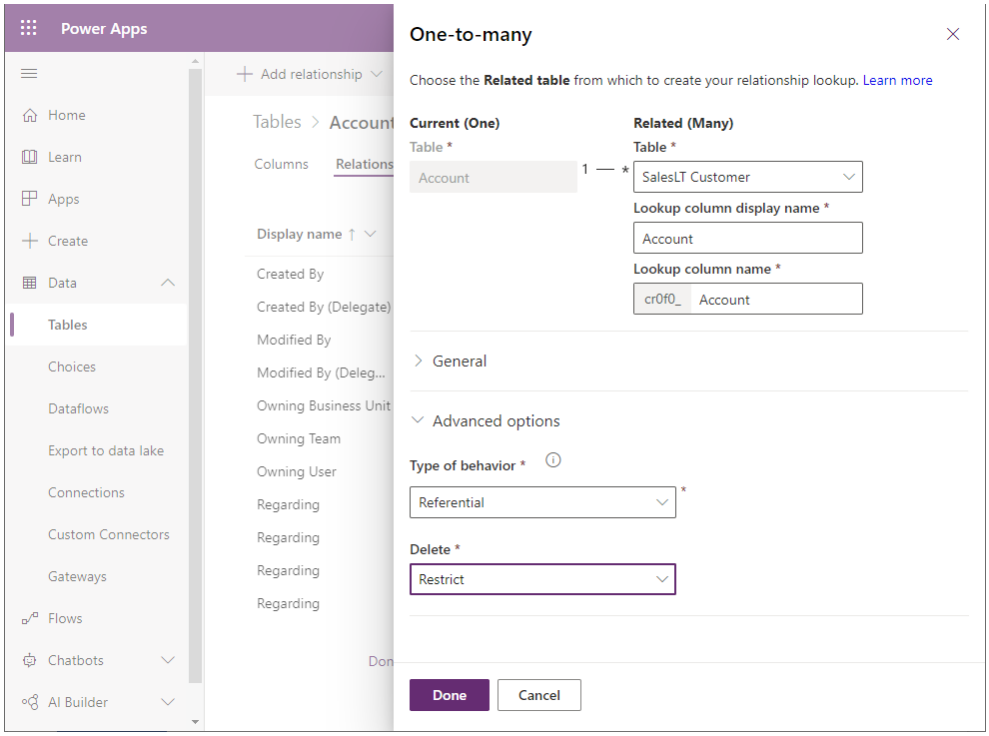
types, such as *Choice* and *Lookup*, enable you to restrict the values entered in a column to a fixed domain, or data retrieved from a column in a related table. The *File* and *Image* types allow you to store unstructured data and images in a table. Images have a maximum size of 30 MB, but files can be up to 128 MB.

Note: Power Apps Studio enables you to define your own custom choices for use by *Choice* columns.

You can also define relationships between tables. These relationships can be *many-to-one*, *one-to-many*, or *many-to-many*. In addition, you specify the behavior of the related entities as part of the relationship. The behavior can be:

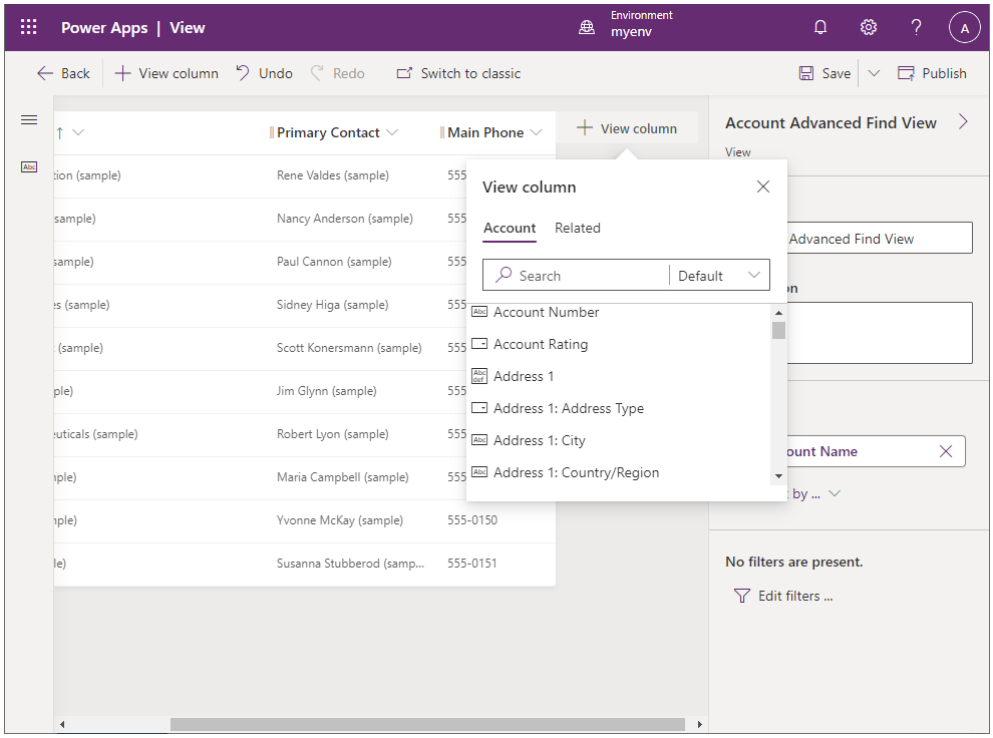
- *Referential,* with or without restricted delete. Restricted delete prevents a row in a related table from being removed if it's referenced by another row in the same, or a different table.
- *Parental,* in which case any action performed on a row is also applied to any rows that it references.
- *Custom,* which enables you to specify how referenced rows are affected by an action performed on the referencing row.

The example below shows how to add a one-to-many relationship from the *Account* table to a custom table named *SalesLT Customer*. The behavior prevents a customer from being deleted if it's referenced by a row in the *Account* table:

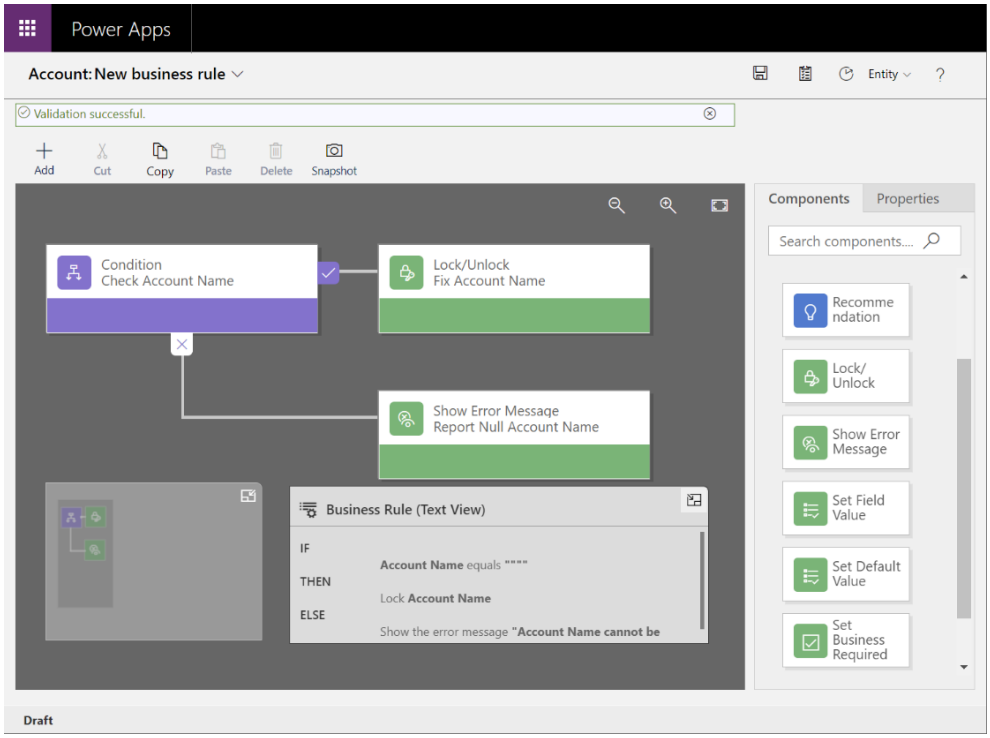


ADDING VIEWS AND BUSINESS RULES

A view provides access to specified columns and rows in one or more related tables. You can think of a view as being a query but with a name that allows you to treat it as a table. A view contains selected columns from a table but can include columns from related tables. Additionally, a view can filter rows to only contain rows that match specified criteria. You can also stipulate the default sort order for the rows presented by a view. Note that a view provides a dynamic window onto the underlying data; if the data changes in the tables behind a view, so does the information represented by the view. You can display data through views in model-driven apps. The following image shows the view designer. The user is adding a new column to a view based on the *Account* table.



Business rules enable you to define validations and automate the flow of control when data is added, modified, or deleted in a table. A business rule comprises a condition that can test for certain conditions in the affected table, such as whether the data in a column matches or breaks a given rule. The business rules designer in Power Apps Studio provides a graphical user interface for defining business rules, as illustrated below.



The business rules designer supports the following actions:

- Set column values.
- Clear column values.

- Set column requirement levels.
- Show or hide columns (for model-driven apps only).
- Enable or disable columns (for model-driven apps only).
- Validate data and show error messages.
- Create business recommendations based on business intelligence (for model-driven apps only).

> **Note**: Business rules are best suited to model-driven apps. Not all business rule actions are supported by canvas apps.

DEFINING BUSINESS ACTIVITIES

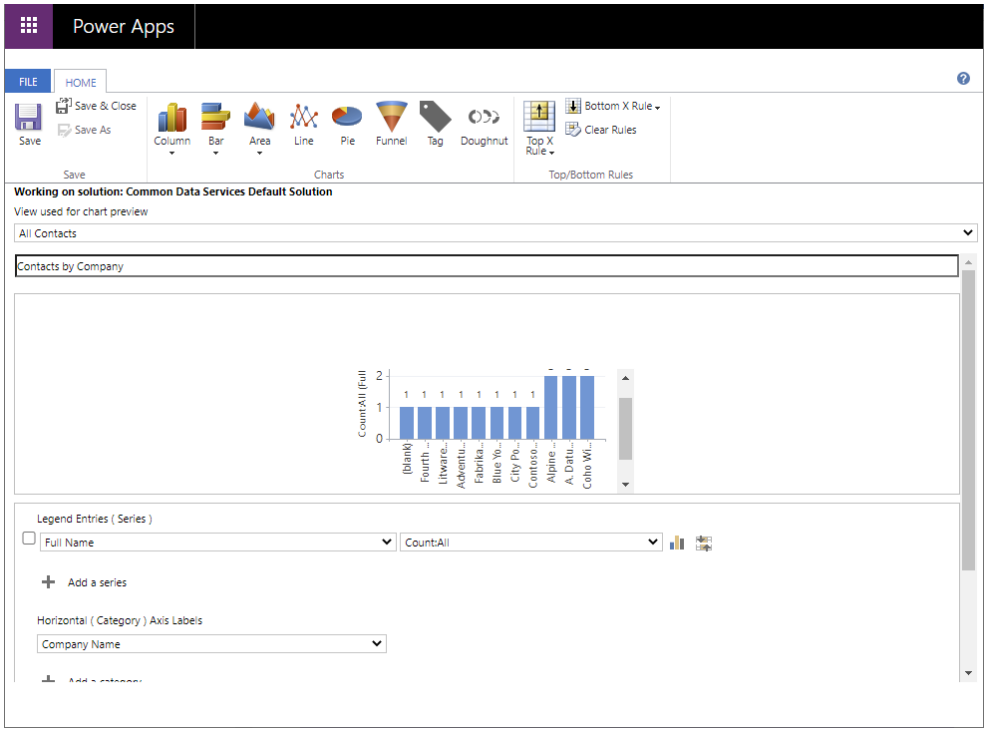
There are two fundamental types of table in Dataverse—*Standard* tables (including custom tables), which contain data, and *Activity* tables, which represent business actions and workflows that can be scheduled to run by Dataverse. An activity table contains references to the data entities involved in the activity (such as customers or salespeople), a series of states through which the activity can go, the current state, and other information used by Dataverse to schedule operations when appropriate.

Dataverse contains built-in activities for managing meetings, scheduling business processes, marketing, managing the sales process, creating recurring appointments, and handling customer service incidents. You can find more detail about these built-in activity entities on the **Activity entities** page at https://aka.ms/AAbvfzk.

You implement the actual business logic using custom actions, or your own code if you require additional control not directly available in Power Apps. The details of this process are beyond the scope of this guide, but for more information, read **Create a custom action** at https://aka.ms/AAbvfzm.

ADDING GRAPHICAL DISPLAY ELEMENTS

As well as the data structure and logic associated with a business entity, Dataverse can also store layouts for forms, charts, and dashboards associated with an entity. When you create a model-driven app, you can use these forms for data entry and display, while the charts and dashboards enable a user to visualize the data more easily than by looking at basic data values.



WHY VANARSDEL DIDN'T USE DATAVERSE

Dataverse is an excellent choice of repository for many situations. You should seriously consider it for Power Apps development based on new systems and services, especially if you're creating model-driven apps. However, VanArsdel's current operations are heavily dependent on existing legacy systems and databases. For the time being, VanArsdel want to focus on getting the Power Apps out into the field rather than spend time migrating to Dataverse. Kiana and Maria, the professional and citizen developers, want to integrate their existing systems and processes into a Power Apps solution for Caleb, the field technician, as quickly as possible. They also want to minimize disruption to the critical operations performed by the receptionist Malik, or compromise the security and IT operations managed by Preeti. To achieve this, they will create a Web API around their existing systems and connect to this Web API from Power Apps. They can then integrate the Web API into their canvas app. The following chapters walk through this process.

CHAPTER 5: CREATING AND PUBLISHING A WEB API IN AZURE

Having established that the data for the technicians' app should be sourced from the existing systems through a Web API, Maria and Kiana work together to determine exactly which information is needed, and in what format. Kiana will then create a web app that exposes the appropriate Web API and arrange for it to be hosted in Azure. The app can connect to Azure from anywhere there is a wireless connection.

DEFINING THE WEB API OPERATIONS: FIELD INVENTORY MANAGEMENT

The **Browse** screen of the Field Inventory Management section of the app displays a list of parts for boilers and air conditioning systems (referred to simply as *boiler parts*). The **Details** screen enables the technician to view more information about a selected part.

In the existing inventory database (named **InventoryDB**), information about parts is held in a single table named **BoilerParts**. Kiana determines that the Web API should support the following requests:

- Get all boiler parts.
- Get the details of a part given the part ID.

DEFINING THE WEB API OPERATIONS: FIELD KNOWLEDGEBASE

In the existing system, the knowledge base database (named **KnowledgeDB)** contains three tables that record and manage the relationships between tips, engineers, and parts:

- **Tips**, which contains the details of a tip. Each tip comprises a single line summary identifying a particular problem (the *subject*), and a more detailed explanation describing how to solve the problem (the *body*). Each tip also references a part, and the engineer who recorded the tip.
- **BoilerParts**, which contains a list of the parts referenced by tips. The details of the parts themselves are stored in the **BoilerParts** table in the **InventoryDB** database.
- **Engineers**, which lists the technicians who have authored each tip.

The knowledge base part of the app currently just contains a placeholder **Browser** screen. Maria wants to implement the following functionality:

- The technician specifies a search term on the **Browse** screen to find all matching tips. The match could be in the name of the part to which the tip refers, the text in the subject or body of the tip, or the name of a technician who is an expert with a specific piece of equipment.
- When all matching tips have been found, the technician can select a tip to view its details.
- A technician can also add new tips to the knowledge base, as well as add notes and comments to existing tips.

The knowledge base is large and growing, and querying across multiple tables and columns can involve complex logic that requires significant compute power. To reduce the load on the Web API, Kiana decides to use Azure Search to provide the search functionality, as described earlier. To support the app, Kiana decides that the following operations are required from the Web API:

- Find the details of a specified knowledge base tip from the **Tips** table.
- Update an existing knowledge base tip in the **Tips** table.
- Add a new knowledge base tip to the **Tips** table, which might also involve adding rows to the **BoilerParts** and **Engineers** tables if the specified part or engineer currently have no tips recorded against them. The routine that actually performs the logic behind adding a new tip will be implemented as an Azure Logic app called from the app.

DEFINING THE WEB API OPERATIONS: FIELD SCHEDULING

Scheduling technician appointments requires not only querying, adding, and removing appointments, but also recording information about customers. The existing appointments system records this data in three tables in the **SchedulesDB** database:

- **Appointments**, which contains the details of each appointment, including the date, time, problem, notes, and technician assigned to the task.
- **Customers**, which holds the details of each customer, including their name, address, and contact details.
- **Engineers**, which lists each technician attending appointments.

> **Note**: The database actually contains a fourth table named **AppointmentsStatus**. This table contains a list of valid appointment statuses and is simply a lookup used by other parts of the existing appointments system.

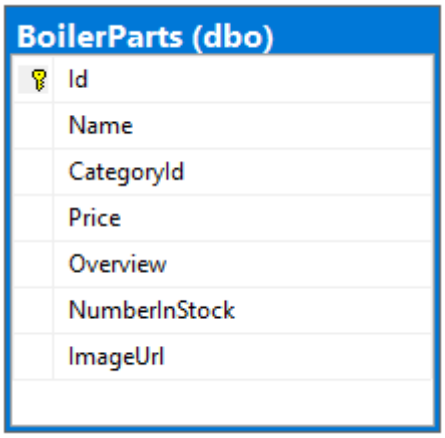
Kiana decides that the following operations would be useful for the Field Scheduling part of the app:

- Find all appointments for a specified technician.
- Find all appointments for the current day for a specified technician.
- Find the next scheduled appointment for a specified technician.
- Update the details of an appointment, such as adding notes and a photograph.
- Find the details of a customer.

BUILDING THE WEB API: FIELD INVENTORY MANAGEMENT

The existing systems store data using Azure SQL Database. Kiana decides to build the Web API using the Entity Framework Core, because this approach can generate a lot of the code that queries, inserts, and updates data automatically. The Web API template provided by Microsoft can also create the Swagger descriptions that describe each operation in the API. These descriptions are useful for testing the API operations. Many tools can use this information to integrate the API with other services, such as Azure API Management.

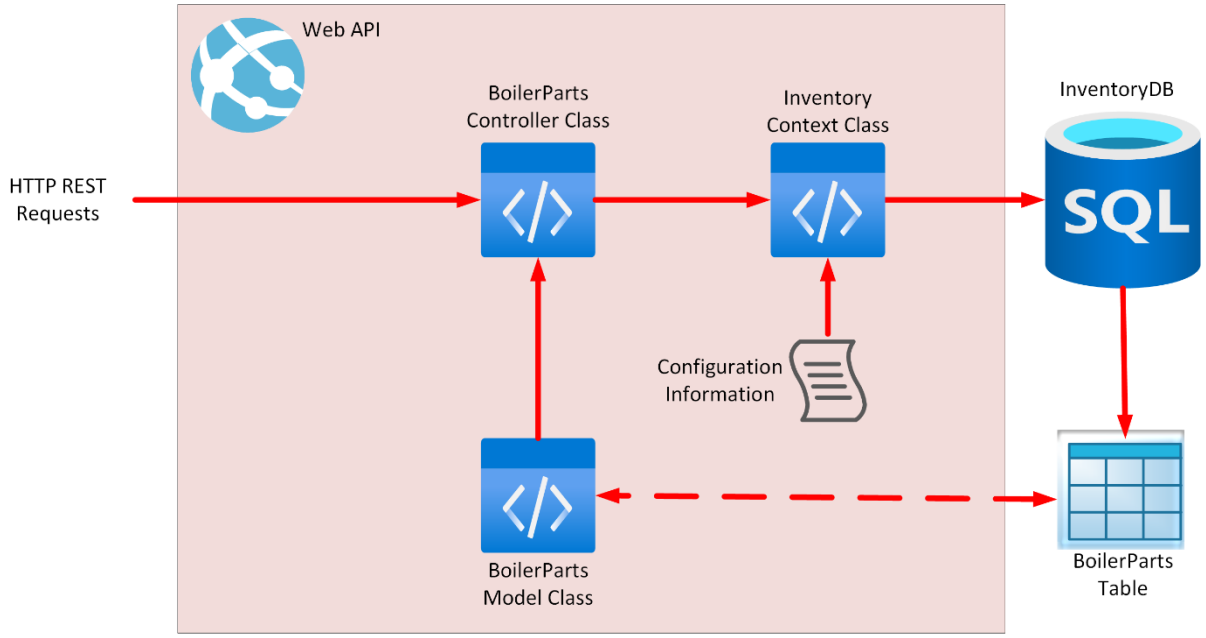
Kiana started with the Field Inventory functionality because this is the most straightforward part. The Field Inventory operations in the Web API query a single table, **BoilerParts**, in the **InventoryDB** database. This table contains the columns shown below:



Kiana took the *Code First* approach to building the Web API. With this strategy, she:

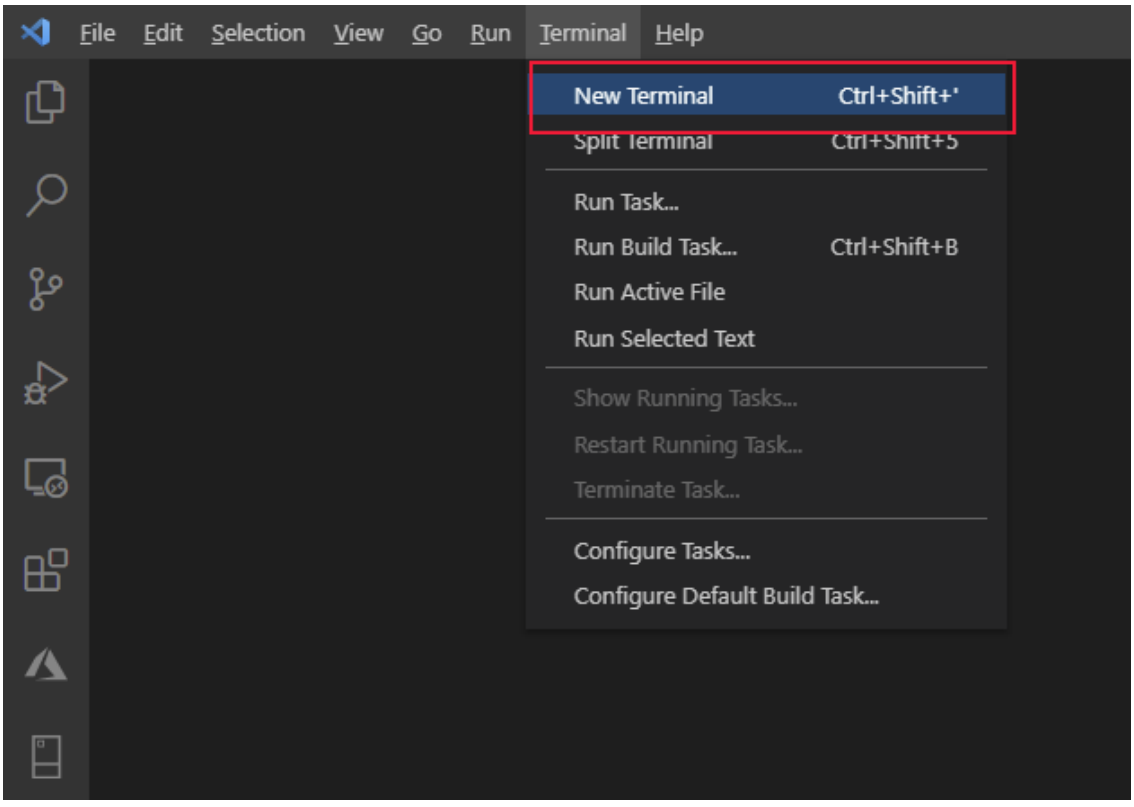
1. Defined her own C# **model** class that mirrored the structure of the **BoilerParts** table in the **InventoryDB** database.
2. Created an Entity Framework **context** class that the Web API uses to connect to the database, to perform queries.
3. Configured the context class to connect to the **InventoryDB** database in Azure.
4. Used the Entity Framework command-line tools to generate a Web API **controller** class that implements HTTP REST requests for each of the operations that can be performed against the **BoilerParts** table.

5. Used the Swagger API to test the Web API.



Kiana used the following procedure to create the Web API using .NET 5.0 command-line tools and Visual Studio Code:

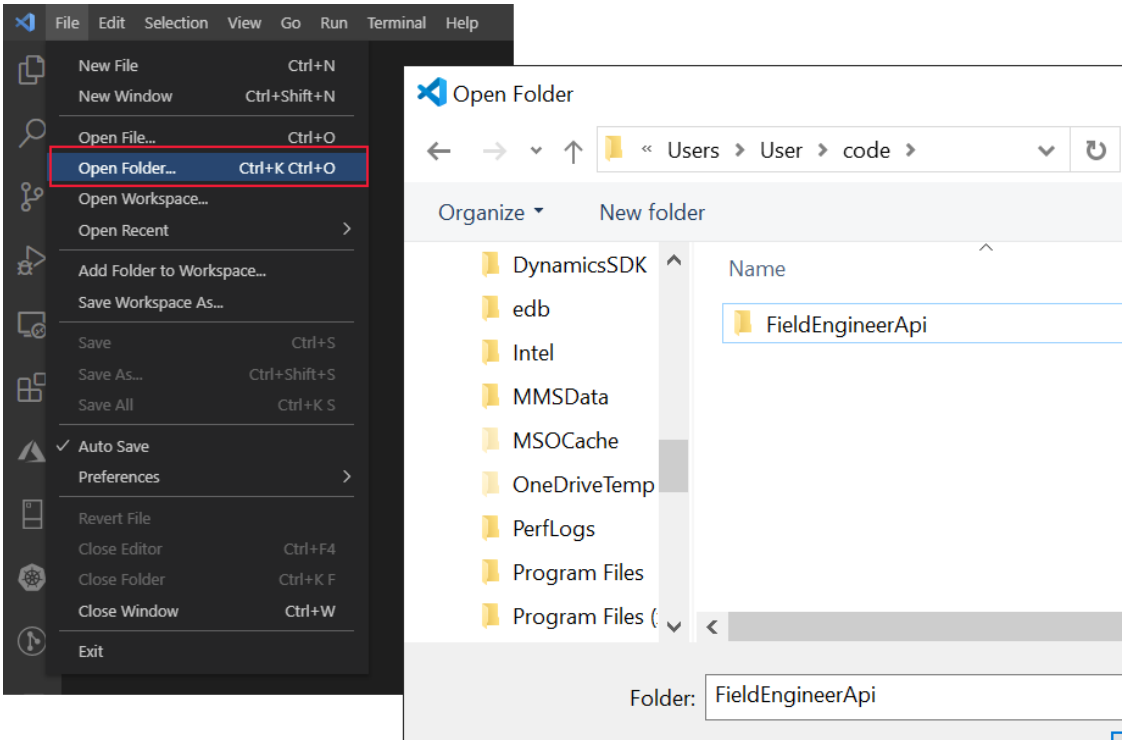
1. Open a terminal window in Visual Studio Code.



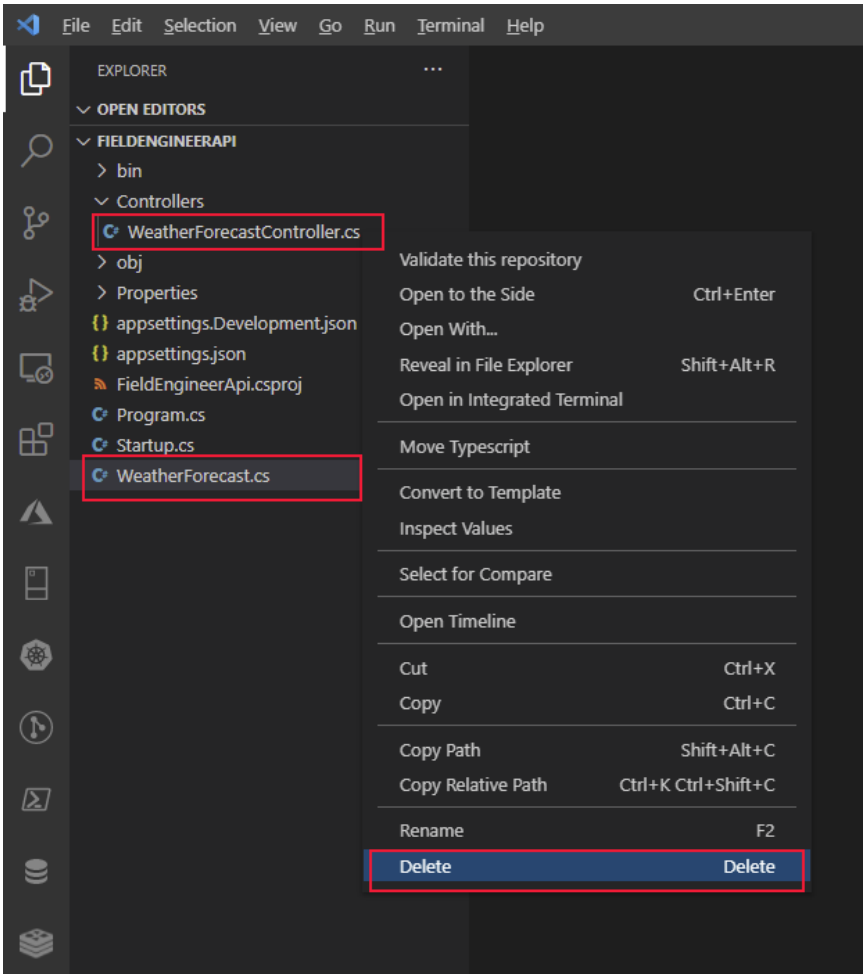
2. Run the following command to create a new Web API project named **FieldEngineerApi**:

```
dotnet new webapi -o FieldEngineerApi
```

3. Open the **FieldEngineerApi** folder:



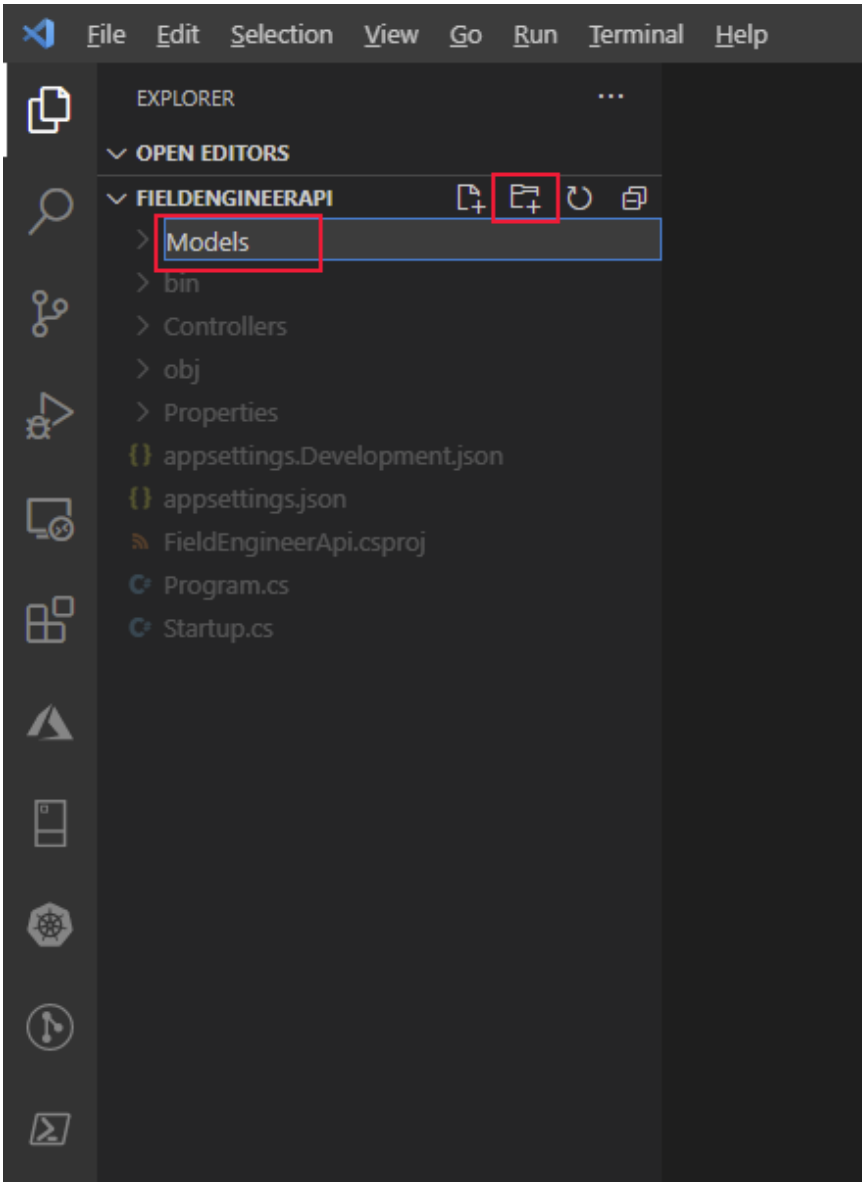
4. Remove the example **WeatherForecastController.cs** controller and **WeatherForecast.cs** class file that was created by the Web API template:



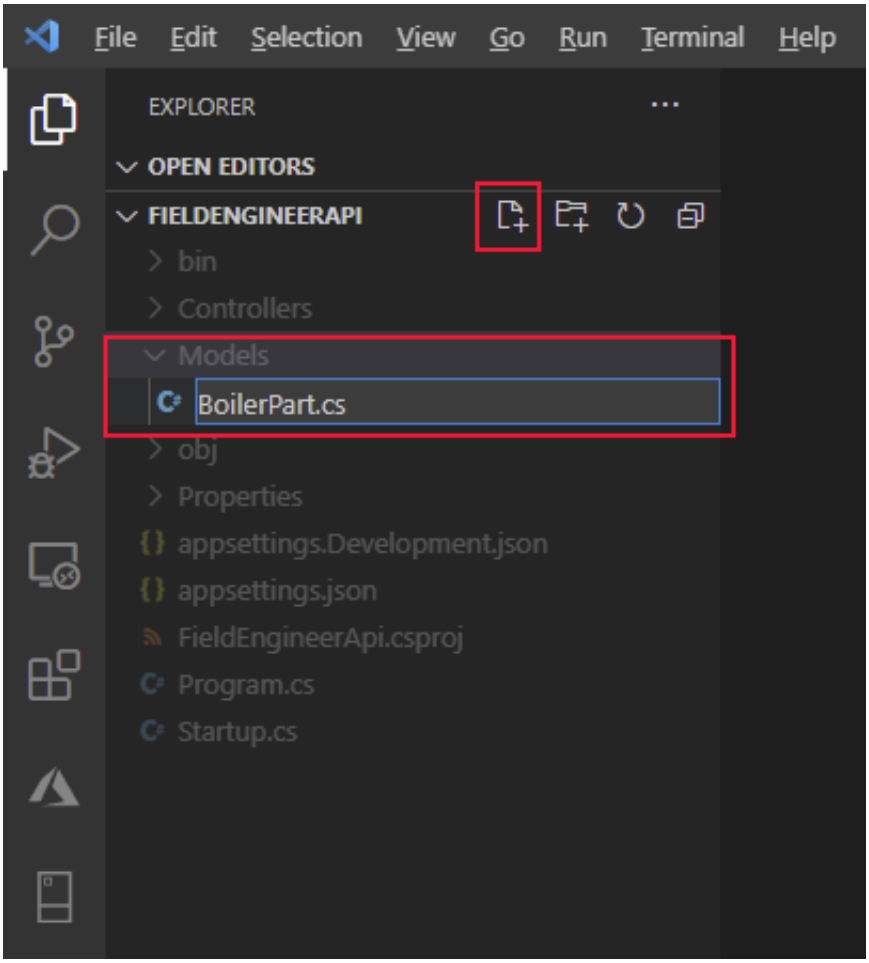
5. In the **Terminal** window, add the following Entity Framework packages and tools, together with support for using SQL Server, to the project:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Mvc.NewtonsoftJson
dotnet tool install --global dotnet-ef
dotnet tool install --global dotnet-aspnet-codegenerator
```

6. In the **FieldEngineerApi** folder, create a new folder called **Models**:



7. In the **Models** folder, create a C# code file named **BoilerPart.cs**.



8. In this file, add the properties and fields shown below. These properties and fields mirror the structure of the **BoilerParts** table in the **InventoryDB** database:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace FieldEngineerApi.Models
{
    public class BoilerPart
    {
        [Key]
        public long Id { get; set; }

        public string Name { get; set; }

        public string CategoryId { get; set; }

        [Column(TypeName = "money")]
        public decimal Price { get; set; }

        public string Overview { get; set; }

        public int NumberInStock { get; set; }

        public string ImageUrl { get; set; }
    }
}
```

9. In the **Models** folder, create another C# code file named **InventoryContext.cs**. Add the code shown below to this class. The class provides the connection between the controller (to be created next), and the database.

```
using Microsoft.EntityFrameworkCore;

namespace FieldEngineerApi.Models
{

    public class InventoryContext : DbContext
    {
        public InventoryContext(DbContextOptions<InventoryContext>
options)
            : base(options)
        {

        }

        public DbSet<BoilerPart> BoilerParts { get; set; }
    }
}
```

10. Edit the **appsettings.Development.json** file for the project, and add a **ConnectionStrings** section with the **InventoryDB** connection string highlighted in bold below. Replace *<server name>* with the name of the Azure SQL Database server you created to hold the **InventoryDB** database.

```
{
  "ConnectionStrings": {
    "InventoryDB": "Server=tcp:<server
name>.database.windows.net,1433;Initial Catalog=InventoryDB;Persist
Security Info=False;User
ID=sqladmin;Password=Pa55w.rd;MultipleActiveResultSets=False;Encrypt=Tru
e;TrustServerCertificate=False;Connection Timeout=30;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

Important: For the purposes of this guide only, the connection string contains the user ID and password for the database. In a production system, you should never store these items in clear text in a configuration file.

11. Edit the **Startup.cs** file and add the following **using** directives to the list at the start of the file:

```
using FieldEngineerApi.Models;
using Microsoft.EntityFrameworkCore;
```

12. In the **Startup** class, find the **ConfigureServices** method. Add the statement shown in bold below to this method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<InventoryContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("InventoryDB")));

    services.AddControllers();
```

```
    ...
}
```

13. Modify the **Configure** method, and enable the Swagger UI even when the app is running in production mode, as shown below in bold (this change involves relocating the two **app.UseSwagger** method calls outside of the **if** statement):

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseSwagger();
    app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json",
"FieldEngineerApi v1"));

    ...
}
```

> **Important:** This change enables the Swagger endpoint to be exposed for Azure API Management integration. Once APIM has been configured, you should move this code back inside the **if** statement and redeploy the Web API. **Never leave the Swagger endpoint open in a production system**.

14. In the **Terminal** window, run the following command to generate the **BoilerParts** controller from the **BoilerPart** model class and the **InventoryContext** context class:

```
dotnet aspnet-codegenerator controller ^
  -name BoilerPartsController -async -api ^
  -m BoilerPart -dc InventoryContext -outDir Controllers
```

The **BoilerParts** controller should be created in the **Controllers** folder.

> **Note:** The line terminator character, **^**, is only recognized by Windows. If you are running Visual Studio Code on a Linux system, use the **** character instead.

15. Open the **BoilerParts.cs** file in the **Controllers** folder and review its contents. The **BoilerPartsController** class exposes the following REST methods:

- **GetBoilerParts()**, which returns a list of all the **BoilerPart** objects from the database.
- **GetBoilerPart(long id)**, which retrieves the details of the specified boiler part.
- **PutBoilerPart(long id, BoilerPart boilerPart)**, which updates a boiler part in the database with the details in the **BoilerPart** object specified as a parameter.
- **PostBoilerPart(BoilerPart boilerPart)**, which creates a new boiler part.
- **DeleteBoilerPart(long id)**, which removes the specified boiler part from the database.

> **Note**: The technician's app only requires the two **Get** methods, but the others are useful for the desktop inventory management app (not covered in this guide).

16. Compile and build the Web API:

```
dotnet build
```

The Web API should build without reporting any errors or warnings.

DEPLOYING THE WEB API TO AZURE: FIELD INVENTORY MANAGEMENT

Kiana deployed and tested the Web API, by performing the following tasks:

1. Using the Azure Account extension in Visual Studio Code, sign in to your Azure subscription.
2. From the Terminal window in Visual Studio Code, create a new resource group called **webapi_rg** in your Azure subscription. In the command below, replace *<location>* with your nearest Azure region:

```
az group create ^
  --name webapi_rg ^
  --location <location>
```

3. Create an Azure Appservice Plan to provide the resources for hosting the Web API:

```
az appservice plan create  ^
  --name webapi_plan ^
  --resource-group webapi_rg ^
  --sku F1
```

> **Note**: **F!** is the free SKU for Appservice plans. It provides limited throughput and capacity, and is only suitable for development purposes.

4. Create an Azure Web App using the Appservice Plan. Replace *<webapp name>* with a unique name for the web app:

```
az webapp create ^
  --name <webapp name> ^
  --resource-group webapi_rg ^
  --plan webapi_plan
```

5. In Visual Studio Code, edit the **appSettings.json** file, and add the same connection string that you previously wrote to the **appSettings.Development.json** file. Remember to replace *<server name>* with the name of the Azure SQL Database server you created to hold the **InventoryDB** database.

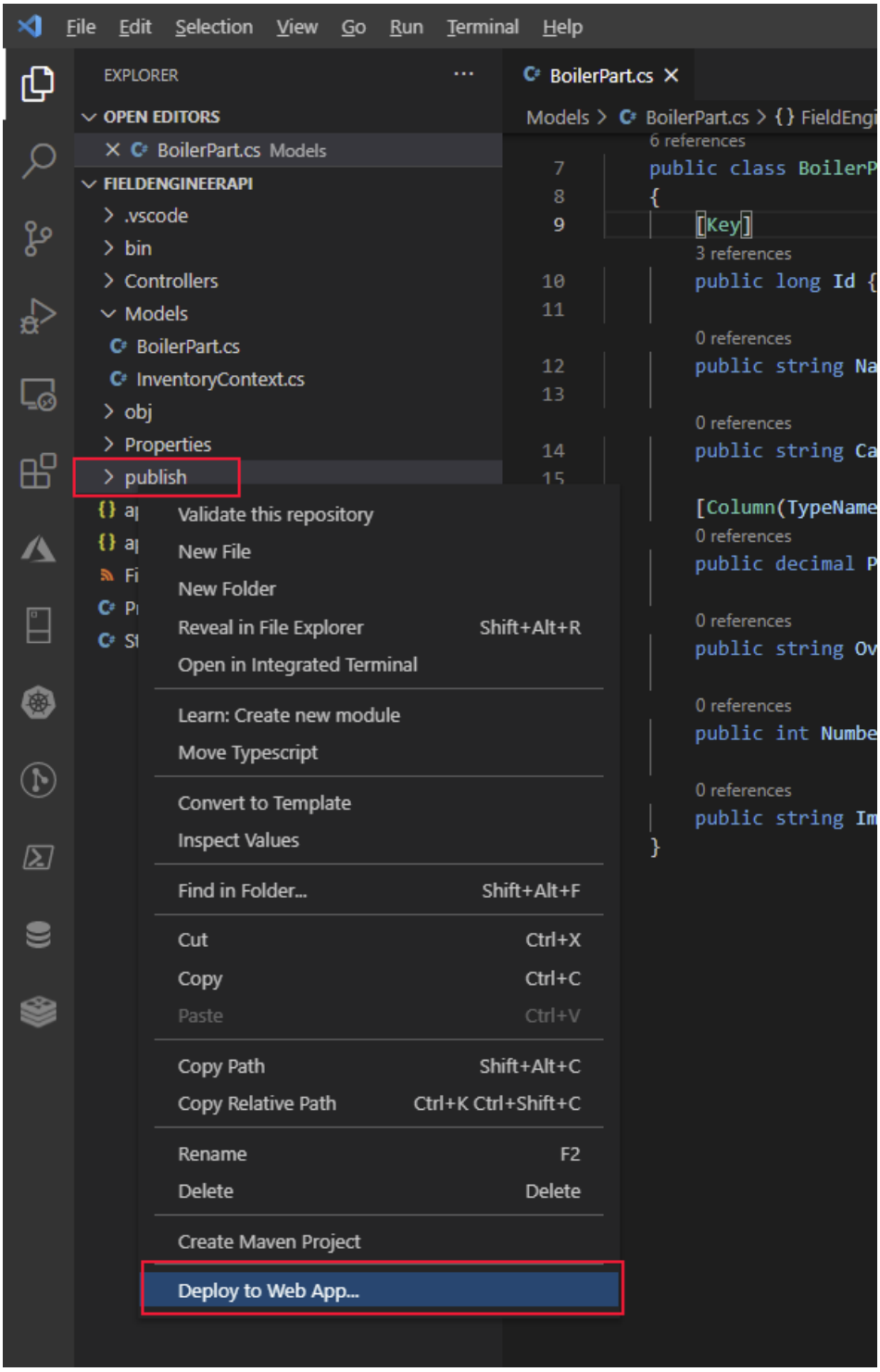
```
{
  "ConnectionStrings": {
    "InventoryDB": "Server=tcp:<server
name>.database.windows.net,1433;Initial Catalog=InventoryDB;Persist
Security Info=False;User
ID=sqladmin;Password=Pa55w.rd;MultipleActiveResultSets=False;Encrypt=Tru
e;TrustServerCertificate=False;Connection Timeout=30;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

6. In the Terminal window, package the Web API ready for deployment to Azure:

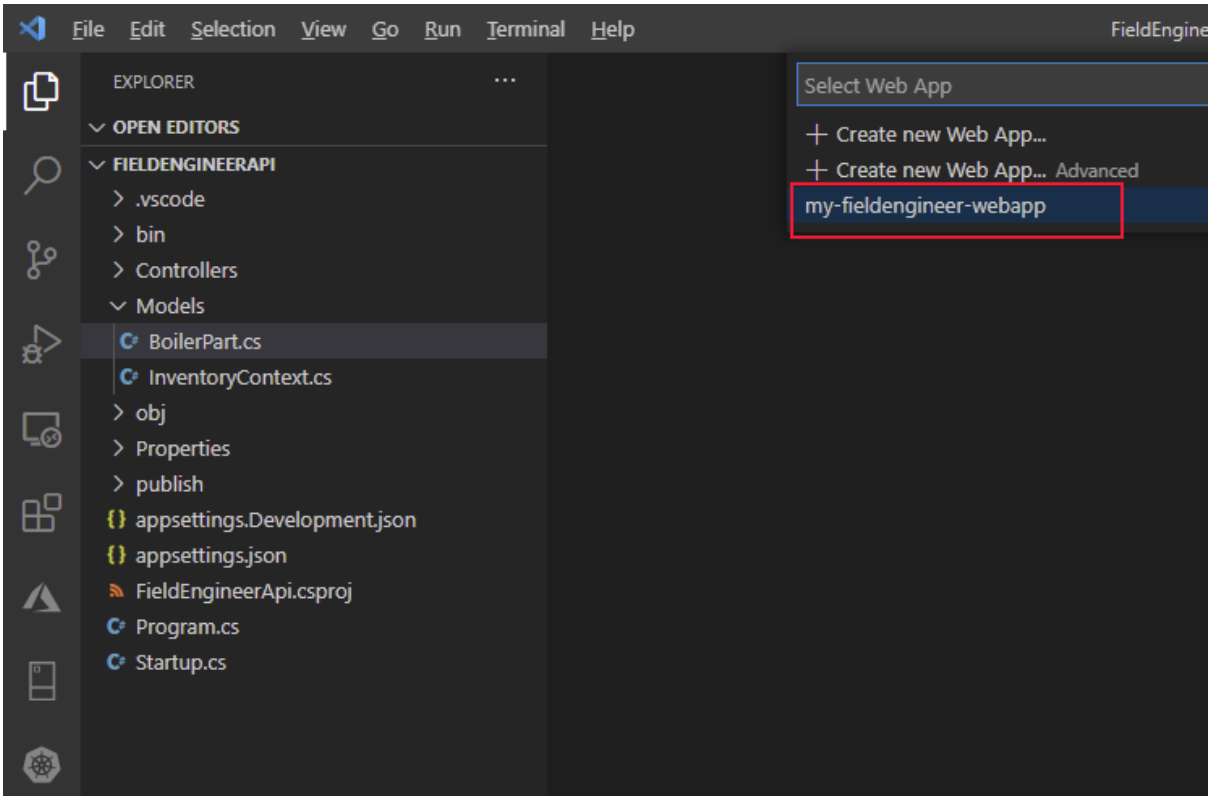
```
dotnet publish -c Release -o ./publish
```

   This command saves the packaged files to a folder named **publish**.

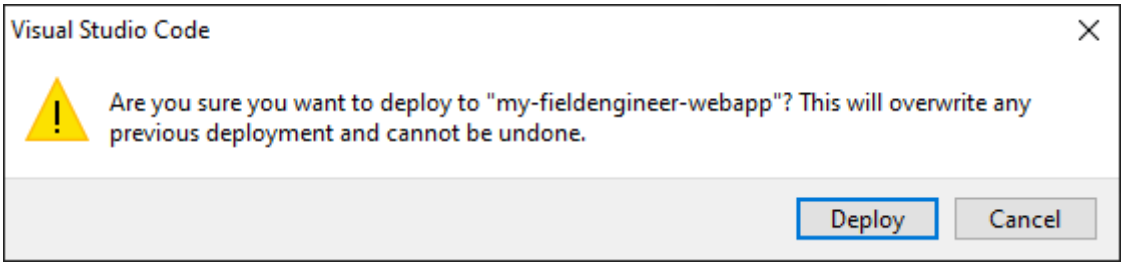
7. In Visual Studio Code, right-click the **publish** folder, and then select **Deploy to Web App**:



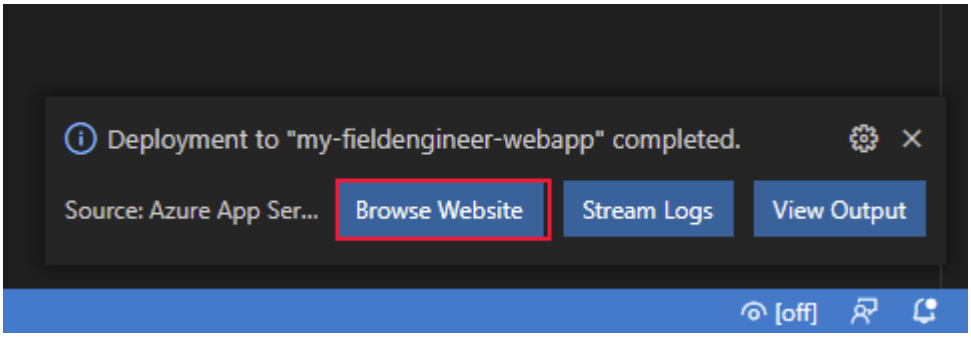
8. Select the name of the web app you created in Step 4 above (*<webapp name>*). In the example below, the web app is called **my-fieldengineer-webapp**:



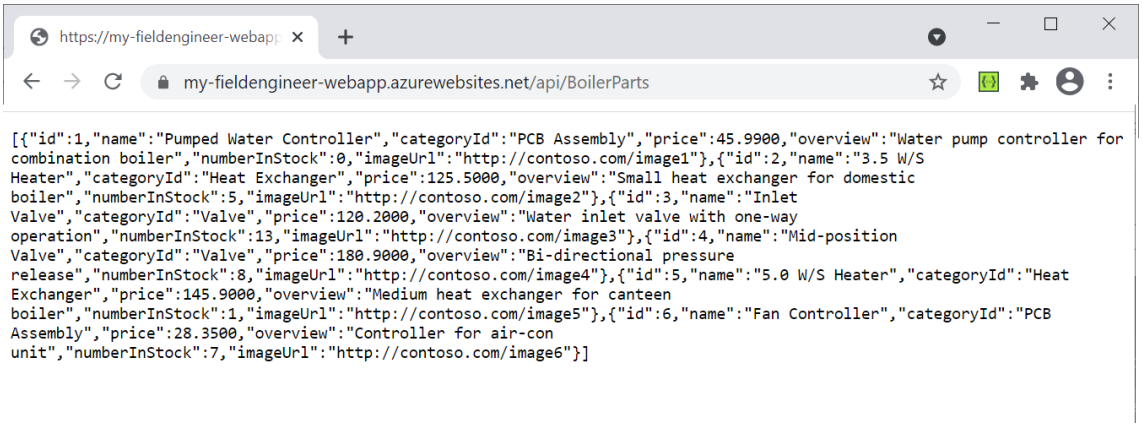
9. At the prompt in the Visual Studio Code dialog box, select **Deploy** to accept the warning and deploy the web app:



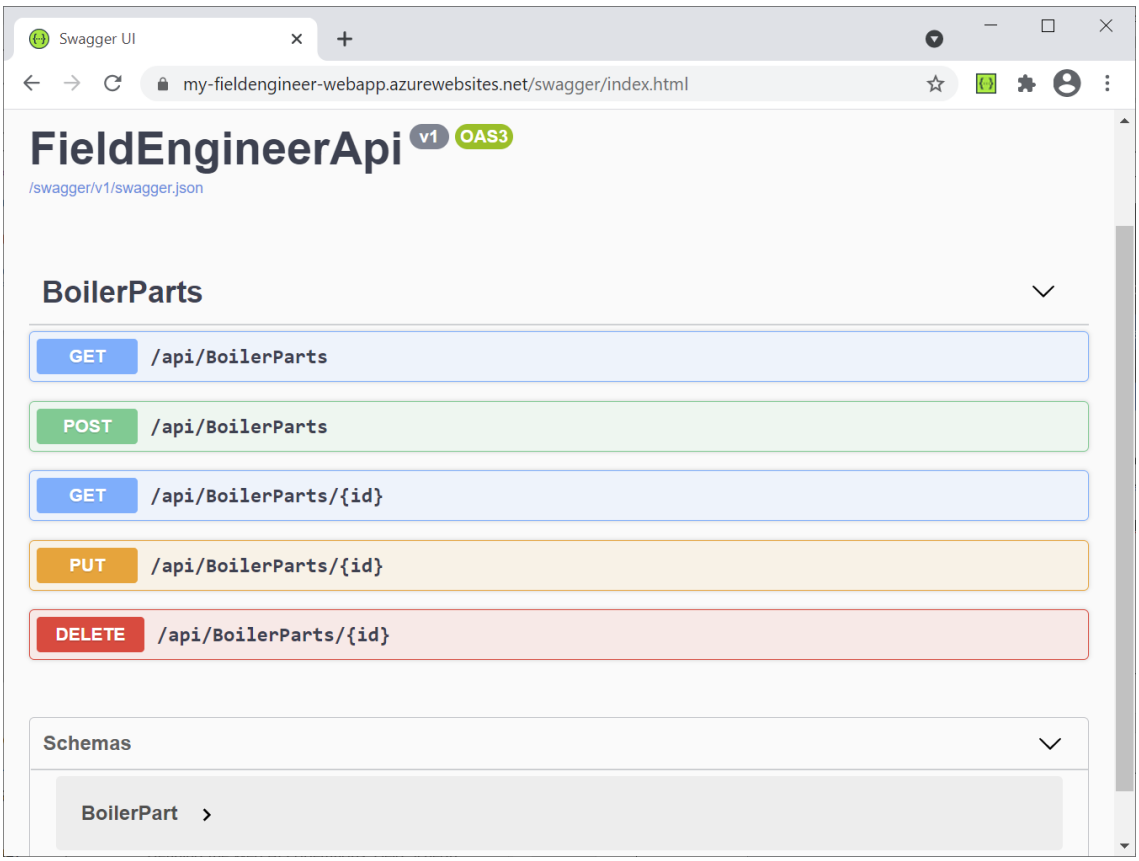
10. Verify that the web app deploys successfully, and then browse to the website:



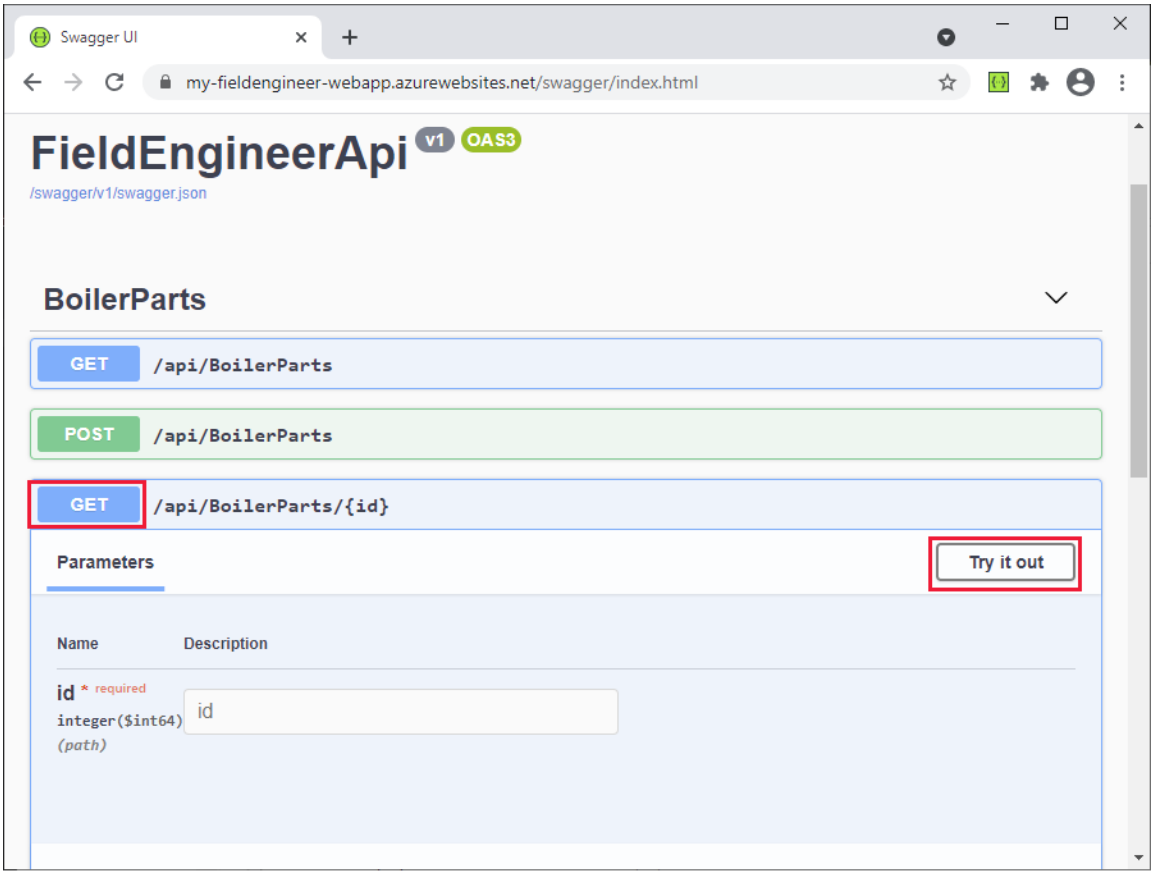
11. The website will open in a new browser window, but will display an HTTP 404 error (not found). This is because the Web API operations are available through the **api** endpoint rather than the root of the website. Change the URL to **https://*<webapp name>*.azurewebsites.net/api/BoilerParts**. This URI invokes the **GetBoilerParts** method in the **BoilerParts** controller. The Web API should respond with a JSON document that lists all the boiler parts in the **InventoryDB** database:



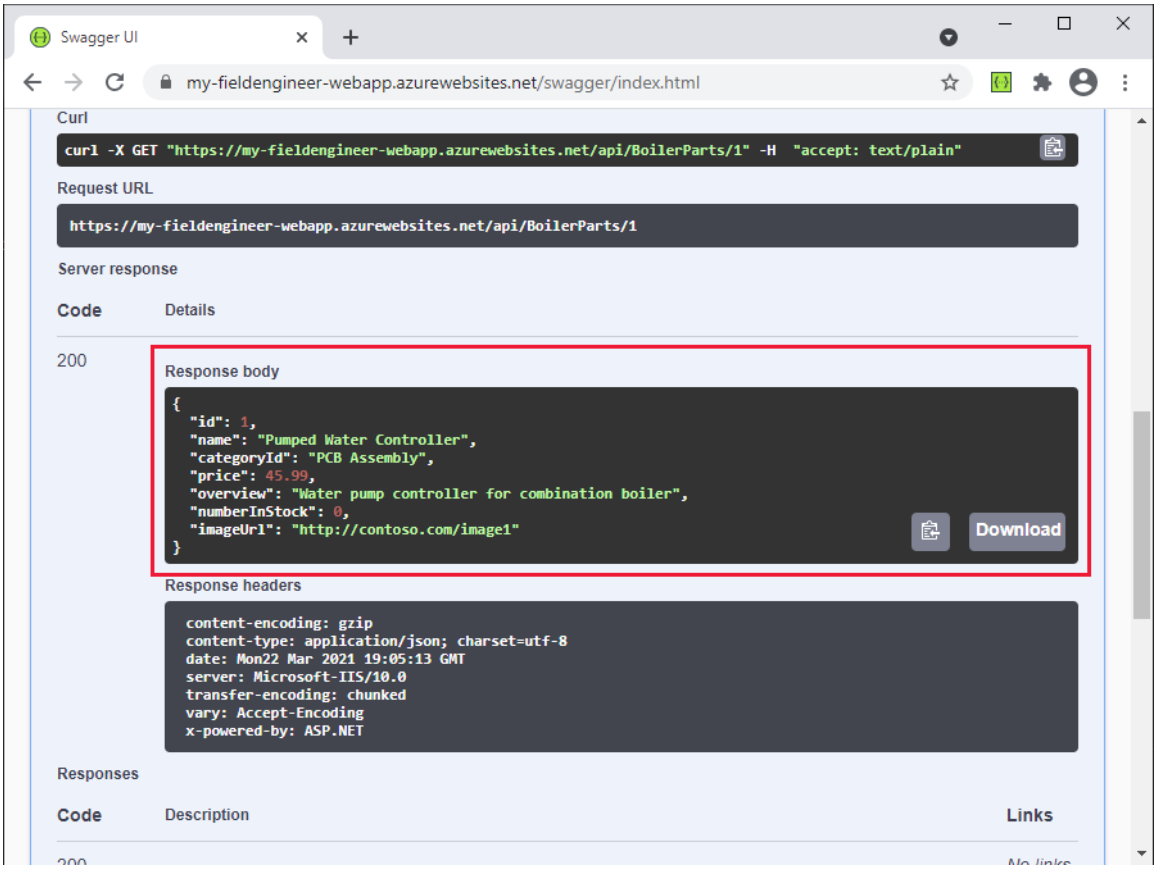
12. Change the URL in the browser to **https://*<webapp name>*.azurewebsites.net/swagger**. The Swagger API should appear. This is a graphical user interface that enables a developer to verify and test each of the operations in a Web API. It also acts as a useful documentation tool:



13. Select **GET** adjacent to the **/api/BoilerParts/{id}** endpoint, and then select **Try it out**.



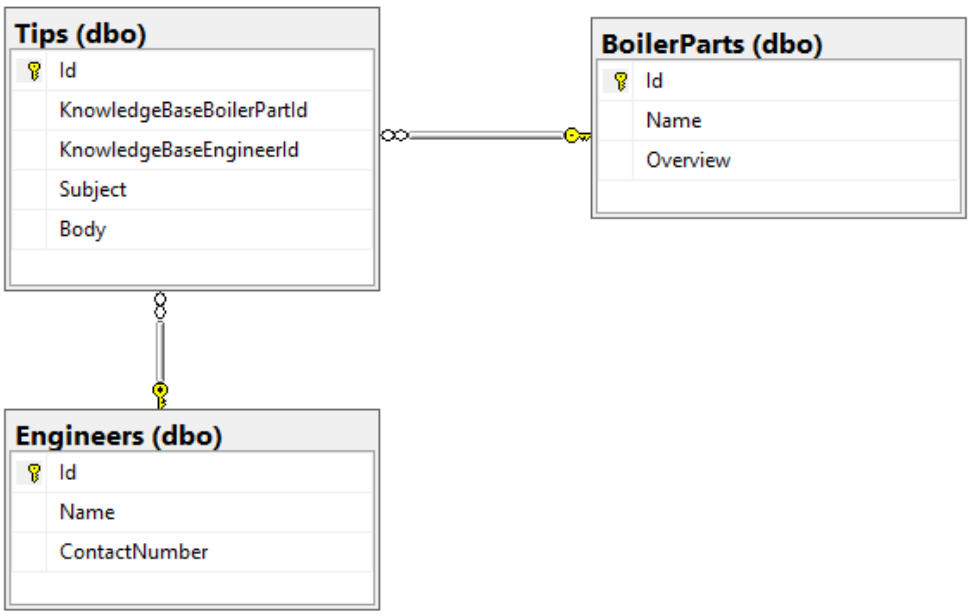
14. In the **id** field, enter the ID of a part, and then select **Execute**. This action calls the **GetBoilerPart(long id)** method in the **BoilerParts** controller. It'll return a JSON document with the details of the part, or an HTTP 404 error if no matching part is found in the database:



15. Close the web browser and return to Visual Studio Code.

BUILDING AND DEPLOYING THE WEB API: FIELD KNOWLEDGEBASE

The Field Knowledgebase operations in the Web API work on three tables in the **KnowledgeDB** database: **Tips**, **BoilerParts**, and **Engineers**. The diagram below shows the relationships between these tables and the columns they contain:



Kiana adopted a similar approach for the Field Knowledgebase database that she used for the Field Inventory Management database. She performed the following tasks:

1. Create C# model classes that mirror the structure of the **Tips**, **BoilerParts**, and **Engineers** table in the **KnowledgeDB** database. The code for each of these classes is shown below:

> **Note**: The **BoilerParts** table in the **KnowledgeDB** database is distinct from the **BoilerParts** table in the **InventoryDB** database. To avoid a name clash, the model classes for tables in the **KnowledgeDB** database have the **KnowledgeBase** prefix.

```
// KnowledgeBaseTips.cs

using System.ComponentModel.DataAnnotations;

namespace FieldEngineerApi.Models
{
    public class KnowledgeBaseTip {
        [Key]
        public long Id { get; set; }
        public long KnowledgeBaseBoilerPartId { get; set; }
        public virtual KnowledgeBaseBoilerPart KnowledgeBaseBoilerPart
{ get; set; }
        public string KnowledgeBaseEngineerId { get; set; }
        public virtual KnowledgeBaseEngineer KnowledgeBaseEngineer
{ get; set; }
        public string Subject { get; set; }
        public string Body { get; set; }
    }
}
```

> **Note**: The engineer **Id** is a string, not a number. This is because the existing systems use GUIDs to identity technicians and other users.

```
// KnowledgeBaseBoilerPart.cs

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace FieldEngineerApi.Models
{
    public class KnowledgeBaseBoilerPart
    {
        [Key]
        public long Id { get; set; }
        public string Name { get; set; }
        public string Overview { get; set; }
        public virtual ICollection<KnowledgeBaseTip> KnowledgeBaseTips
{ get; set; }
    }
}
```

```
// KnowledgeBaseEngineer.cs
```

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace FieldEngineerApi.Models
{

    public class KnowledgeBaseEngineer
    {
        [Key]
        public string Id { get; set; }
        [Required]
        public string Name { get; set; }
        public string ContactNumber { get; set; }
        public virtual ICollection<KnowledgeBaseTip> KnowledgeBaseTips
{ get; set; }
    }
}
```

2. Create another Entity Framework **context** class that the Web API uses to connect to the **KnowledgeDB** database:

```
// KnowledgeBaseContext.cs

using Microsoft.EntityFrameworkCore;

namespace FieldEngineerApi.Models
{

    public class KnowledgeBaseContext : DbContext
    {
        public
KnowledgeBaseContext(DbContextOptions<KnowledgeBaseContext> options)
            : base(options)
        {

        }

        public DbSet<KnowledgeBaseBoilerPart> BoilerParts { get; set; }
        public DbSet<KnowledgeBaseEngineer> Engineers { get; set; }
        public DbSet<KnowledgeBaseTip> Tips { get; set; }
    }
}
```

3. Edit the **appsettings.Development.json** file for the project, and add the **KnowledgDB** connection string highlighted in bold below to the **ConnectionStrings** section. Replace *<server name>* with the name of the Azure SQL Database server you created to hold the **KnowledgeDB** database.

```
{
  "ConnectionStrings": {
    "InventoryDB": "Server=tcp:...",
    "KnowledgeDB": "Server=tcp:<server
name>.database.windows.net,1433;Initial Catalog=KnowledgeDB;Persist
Security Info=False;User
ID=sqladmin;Password=Pa55w.rd;MultipleActiveResultSets=False;Encrypt=Tru
e;TrustServerCertificate=False;Connection Timeout=30;"
  },
  "Logging": {
    ...
    }
  }
```

```
}
```

> **Important**: For the purposes of this guide only, the connection string contains the user ID and password for the database. In a production system, you should never store these items in clear text in a configuration file.

4. Edit the **Startup.cs** file and, in the **ConfigureServices** method, add the statements shown in bold below:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<InventoryContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("InventoryDB")));
    services.AddDbContext<KnowledgeBaseContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("KnowledgeDB")));

    services.AddControllers().AddNewtonsoftJson(
        options => options.SerializerSettings.ReferenceLoopHandling =
Newtonsoft.Json.ReferenceLoopHandling.Ignore
    );

    services.AddControllers();
    ...
}
```

The second statement controls the way in which data is serialized when it s retrieved. Some of the model classes have references to other model classes, which in turn can reference further model classes. Some of these references can result in recursive loops (Entity A references Entity B, which references back to Entity A, which references Entity B again, and so on). The **ReferenceLoopHandling** option causes the serializer to ignore such loops in the data, and only return an entity and the objects that it immediately references, but no more.

5. In the **Terminal** window, run the following command to generate controllers from the **KnowledgeBaseBoilerTip**, **KnowledgeBaseBoilerPart**, and **KnowledgeBaseEngineer** model classes and the **KnowledgeBaseContext** context class:

```
dotnet aspnet-codegenerator controller ^
  -name KnowledgeBaseTipController -async -api ^
  -m KnowledgeBaseTip ^
  -dc KnowledgeBaseContext -outDir Controllers

dotnet aspnet-codegenerator controller ^
  -name KnowledgeBaseBoilerPartController -async -api ^
  -m KnowledgeBaseBoilerPart ^
  -dc KnowledgeBaseContext -outDir Controllers

dotnet aspnet-codegenerator controller ^
  -name KnowledgeBaseEngineerController -async -api ^
  -m KnowledgeBaseEngineer ^
  -dc KnowledgeBaseContext -outDir Controllers
```

All three controllers should be created in the **Controllers** folder.

6. Edit the **KnowledgeBaseBoilerPartController.cs** file. This file contains the code for the **KnowledgeBaseBoilerPart** controller. It should follow the same pattern as the **BoilerPartsController** class created earlier, exposing REST methods that enable a client to list, query, insert, update, and delete entities**.** Add the **GetTipsForPart** method shown below in bold to the controller:

```
[Route("api/[controller]")]
[ApiController]
public class KnowledgeBaseBoilerPartController : ControllerBase
```

```
{
    private readonly KnowledgeBaseContext _context;

    public KnowledgeBaseBoilerPartController(KnowledgeBaseContext
context)
    {
        _context = context;
    }

    // GET: api/KnowledgeBaseBoilerPart/5/Tips
    [HttpGet("{id}/Tips")]
    public async Task<ActionResult<IEnumerable<KnowledgeBaseTip>>>
GetTipsForPart(long id)
    {
        return await _context.Tips.Where(
            t => t.KnowledgeBaseBoilerPartId == id).ToListAsync();
    }

    ...
}
```

This method returns all the knowledge base tips that reference a specified part. It queries the **Tips** table in the database through the **KnowledgeBaseContext** object to find this information.

7. Edit the **KnowledgeBaseEngineerController.cs** file and add the method shown below in bold to the **KnowledgeBaseEngineerController** class:

```
[Route("api/[controller]")]
[ApiController]
public class KnowledgeBaseEngineerController : ControllerBase
{
    private readonly KnowledgeBaseContext _context;

    public KnowledgeBaseEngineerController(KnowledgeBaseContext context)
    {
        _context = context;
    }

    // GET: api/KnowledgeBaseEngineer/5/Tips
    [HttpGet("{id}/Tips")]
    public async Task<ActionResult<IEnumerable<KnowledgeBaseTip>>>
GetTipsForEngineer(string id)
    {
        return await _context.Tips.Where(
            t => t.KnowledgeBaseEngineerId == id).ToListAsync();
    }

    ...
}
```

The **GetTipsForEngineer** method finds all knowledge base tips posted by the specified engineer.

8. In the **Terminal** window, compile and build the Web API:

```
dotnet build
```

The Web API should build without reporting any errors or warnings.

9. Edit the **appSettings.json** file and add the connection string for the **KnowledgeDB** database. This string should be the same that you previously wrote to the **appSettings.Development.json** file:

```
{
```

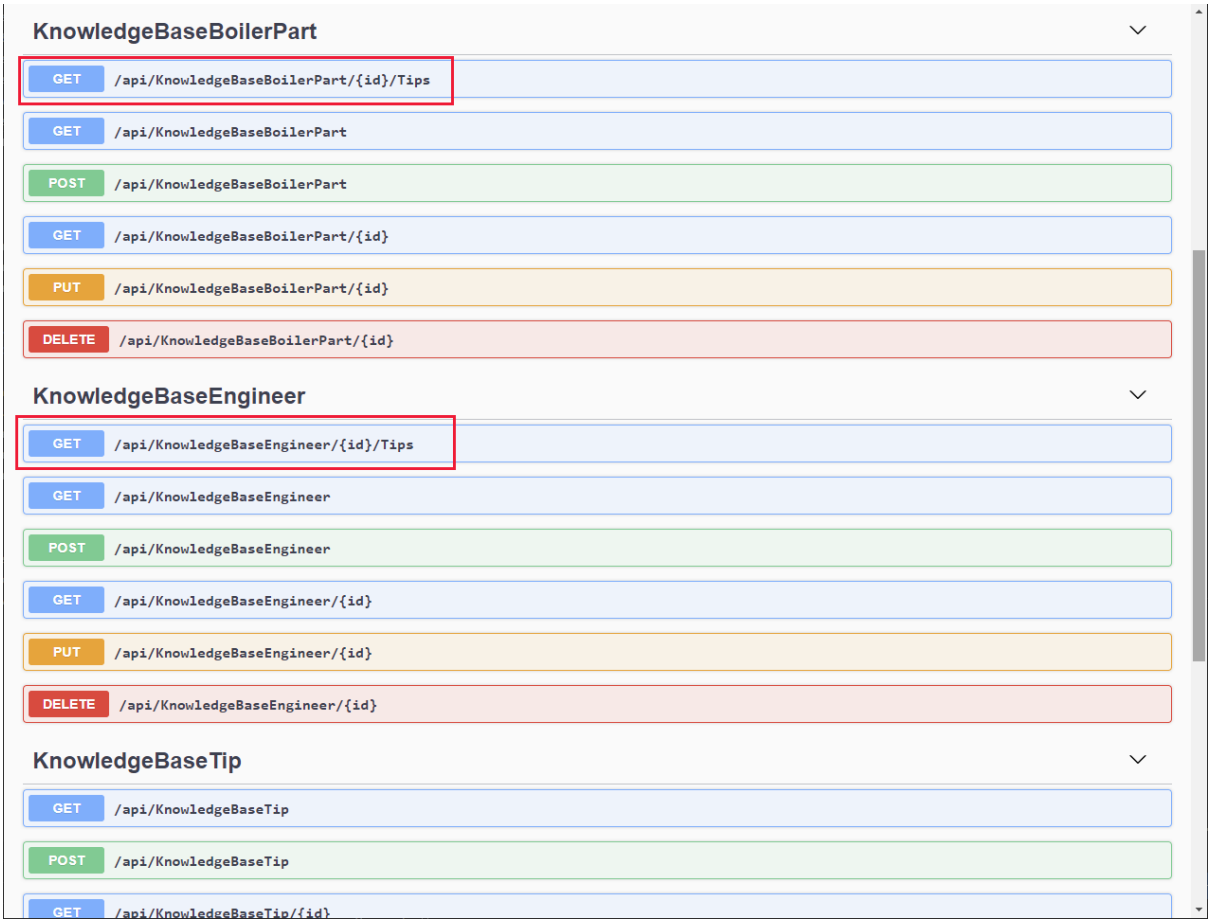
```
  "ConnectionStrings": {
    "InventoryDB": ...,
    "KnowledgeDB": "Server=tcp:<server
name>.database.windows.net,1433;Initial Catalog=KnowledgeDB;Persist
Security Info=False;User
ID=sqladmin;Password=Pa55w.rd;MultipleActiveResultSets=False;Encrypt=Tru
e;TrustServerCertificate=False;Connection Timeout=30;"
  },
  "Logging": {
    ...
  },
  "AllowedHosts": "*"
}
```

10. In the **Terminal** window, package the Web API ready for deployment to Azure:

```
dotnet publish -c Release -o ./publish
```

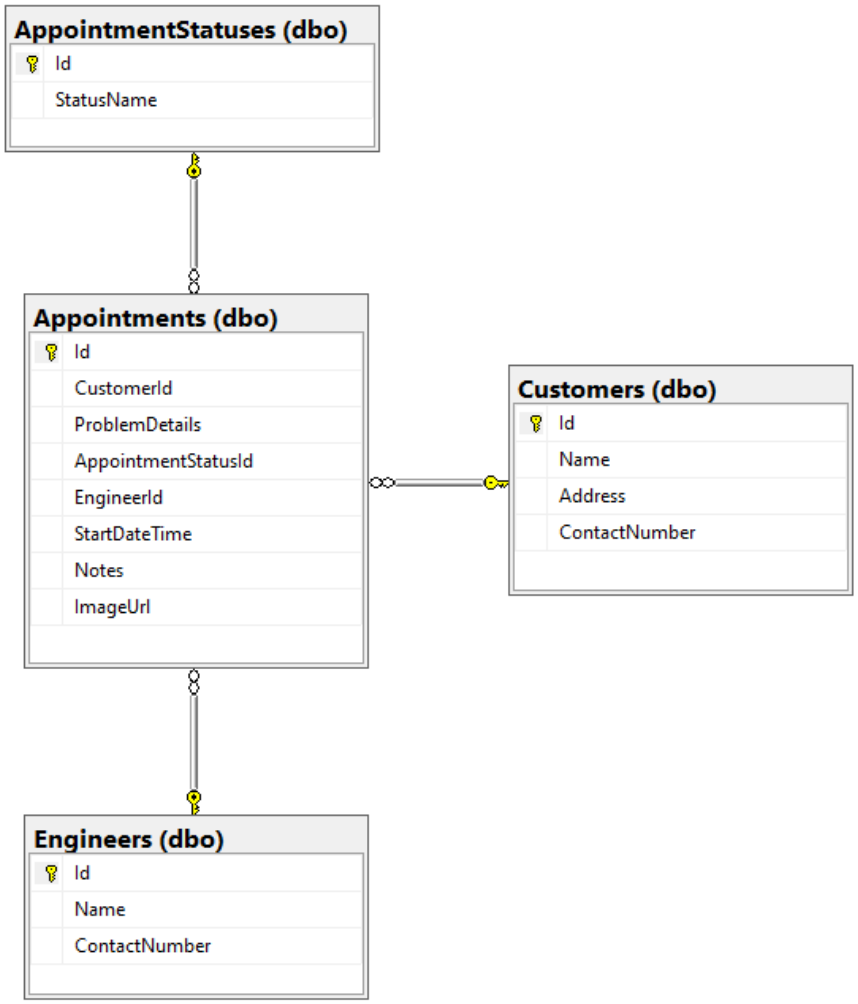
11. In Visual Studio Code, right-click the **publish** folder, and then select **Deploy to Web App**. Deploy to the same Azure Web app that you created previously. Allow the wizard to overwrite the existing web app with the new code.

12. When deployment is complete, browse to the website but change the URL in the browser to **https://*<webapp name>*.azurewebsites.net/swagger**. The operations for the **KnowledgeBaseBoilerPart**, **KnowledgeBaseEngineer**, and **KnowldgeBaseTip** controllers should be listed, as well as the existing **BoilerParts** operations. Verify that the **KnowledgeBaseBoilerPart** operations include a **GET** operation for the URI **/api/KnowledgeBaseBoilerPart/{id}/Tips**, and the **KnowledgeBaseEngineer** operations include a **GET** operation for the URI **/api/KnowledgeBaseEngineer/{id}/Tips**:



BUILDING AND DEPLOYING THE WEB API: FIELD SCHEDULING

The Field Scheduling operations use the tables **Appointments**, **AppointmentStatuses** (this is a simple lookup table that lists the valid appointment status values), **Customers**, and **Engineers**, shown in the diagram below. These tables are stored in the **SchedulesDB** database:



To create the Web API operations for the Field Scheduling part of the system, Kiana performed the following tasks:

1. Create C# model classes that mirror the structure of the **AppointmentStatus**, **Appointments**, **Customers**, and **Engineers** table in the **SchedulesDB** database. The code for each of these classes is shown below:

> **Note**: The model class for **Engineers** table is named **ScheduleEngineer** to distinguish it from the model for the **Engineers** table in the **InventoryDB** database.

```
// AppointmentStatus.cs

using Newtonsoft.Json;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace FieldEngineerApi.Models
{
    public class AppointmentStatus {
```

```
        [Key]
        public long Id { get; set; }
        public string StatusName { get; set; }

        [JsonIgnore]
        public virtual ICollection<Appointment> Appointments { get;
set; }
    }
}
```

// Appointment.cs

```
using System;
using System.ComponentModel.DataAnnotations;

namespace FieldEngineerApi.Models
{

    public class Appointment
    {
        [Key]
        public long Id { get; set; }
        [Required]
        public long CustomerId { get; set; }
        public virtual Customer Customer { get; set; }
        public string ProblemDetails { get; set; }
        [Required]
        public long AppointmentStatusId { get; set; }
        public virtual AppointmentStatus AppointmentStatus { get; set; }
        public string EngineerId { get; set; }
        public virtual ScheduleEngineer Engineer { get ; set; }
        [Display(Name = "StartTime")]
        [DataType(DataType.DateTime)]
        [DisplayFormat(DataFormatString = "{0:MM/dd/yyyy H:mm:ss}")]
        public DateTime StartDateTime { get; set; }
        public string Notes { get; set; }
        public string ImageUrl { get; set; }
    }
}
```

// Customer.cs

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace FieldEngineerApi.Models
{

    public class Customer
    {
        [Key]
        public long Id { get; set; }
        [Required]
        public string Name { get; set; }
        public string Address { get; set; }
        public string ContactNumber { get; set; }
        public virtual ICollection<Appointment> Appointments { get;
set; }
```

```
    }
}
```

```
// ScheduleEngineer.cs

using Newtonsoft.Json;
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;

namespace FieldEngineerApi.Models
{

    public class ScheduleEngineer
    {
        [Key]
        public string Id { get; set; }

        [Required]
        public string Name { get; set; }
        public string ContactNumber { get; set; }

        [JsonIgnore]
        public virtual ICollection<Appointment> Appointments { get;
set; }
    }

}
```

2. Create an Entity Framework **context** class that the Web API uses to connect to the **SchedulesDB** database:

```
// ScheduleContext.cs

using System;
using Microsoft.EntityFrameworkCore;

namespace FieldEngineerApi.Models
{

    public class ScheduleContext : DbContext
    {
        public ScheduleContext(DbContextOptions<ScheduleContext>
options)
            : base(options)
        {

        }

        public DbSet<Appointment> Appointments { get; set; }
        public DbSet<AppointmentStatus> AppointmentStatuses { get;
set; }
        public DbSet<Customer> Customers { get; set; }
        public DbSet<ScheduleEngineer> Engineers { get; set; }
    }
}
```

3. Edit the **appsettings.Development.json** file for the project, and add the **SchedulesDB** connection string highlighted in bold below to the **ConnectionStrings** section. Replace *<server name>* with the name of the Azure SQL Database server you created to hold the **KnowledgeDB** database.

```
{
```

```
  "ConnectionStrings": {
    "InventoryDB": "Server=tcp:...",
    "KnowledgeDB": "Server=tcp;... ",
    "SchedulesDB": "Server=tcp:<server
name>.database.windows.net,1433;Initial Catalog=SchedulesDB;Persist
Security Info=False;User
ID=sqladmin;Password=Pa55w.rd;MultipleActiveResultSets=False;Encrypt=Tru
e;TrustServerCertificate=False;Connection Timeout=30;"
  },
  "Logging": {
    ...
    }
  }
}
```

Important: For the purposes of this guide only, the connection string contains the user ID and password for the database. In a production system, you should never store these items in clear text in a configuration file.

4. Edit the **Startup.cs** file and in the **ConfigureServices** method, add the statement shown in bold below:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<InventoryContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("InventoryDB")));
    services.AddDbContext<KnowledgeBaseContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("KnowledgeDB")));
    services.AddDbContext<ScheduleContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("SchedulesDB")));

    services.AddControllers().AddNewtonsoftJson(
        options => options.SerializerSettings.ReferenceLoopHandling =
Newtonsoft.Json.ReferenceLoopHandling.Ignore
    );

    services.AddControllers();
    ...
}
```

5. In the **Terminal** window, run the following command to generate controllers from the **Appointment**, **Customer** and **ScheduleEngineer** model classes, and the **ScheduleContext** context class:

Note: Don't create a separate controller for the **AppointmentStatus** model.

```
dotnet aspnet-codegenerator controller ^
  -name AppointmentsController -async -api ^
  -m Appointment ^
  -dc ScheduleContext -outDir Controllers

dotnet aspnet-codegenerator controller ^
  -name CustomerController -async -api ^
  -m Customer ^
  -dc ScheduleContext -outDir Controllers

dotnet aspnet-codegenerator controller ^
  -name ScheduleEngineerController -async -api ^
  -m ScheduleEngineer ^
  -dc ScheduleContext -outDir Controllers
```

6. Edit the **AppointmentsController.cs** file. In the **AppointmentsController** class, find the **GetAppointments** method. Modify the **return** statement as shown below. This change ensures that the **Customer**, **Engineer**, and **AppointmentStatus** information is retrieved as part of the **GET** operation; these fields reference other entities that would otherwise be left null due to the lazy loading mechanism of the Entity Framework.

```
public class AppointmentsController : ControllerBase
{
    private readonly ScheduleContext _context;

    public AppointmentsController(ScheduleContext context)
    {
        _context = context;
    }

    // GET: api/Appointments
    [HttpGet]
    public async Task<ActionResult<IEnumerable<Appointment>>>
GetAppointments()
    {
        return await _context.Appointments
            .Include(c => c.Customer)
            .Include(e => e.Engineer)
            .Include(s => s.AppointmentStatus)
            .ToListAsync();
    }
    ...
}
```

7. In the same file, modify the **GetAppointment(long id)** method as shown below:

```
// GET: api/Appointments/5
[HttpGet("{id}")]
public async Task<ActionResult<Appointment>> GetAppointment(long id)
{
    var appointment = _context.Appointments
        .Where(a => a.Id == id)
        .Include(c => c.Customer)
        .Include(e => e.Engineer)
        .Include(s => s.AppointmentStatus);

    var appData = await appointment.FirstOrDefaultAsync();

    if (appData == null)
    {
        return NotFound();
    }

    return appData;
}
```

This version of the method populates the **Customer**, **Engineer**, and **AppointmentStatus** fields of an appointment when it is retrieved (lazy loading will leave these fields empty otherwise).

8. Find the **PutAppointment** method, and replace it with the code shown below:

This version of the **PutAppointment** method takes the fields in an appointment that a user can modify in the app rather than a complete **Appointment** object:

```
[HttpPut("{id}")]
public async Task<IActionResult> PutAppointment(long id,
        string problemDetails, string statusName,
```

```
        string notes, string imageUrl)
{
    var statusId = _context.AppointmentStatuses.First(s => s.StatusName
== statusName).Id;
    var appointment =  _context.Appointments.First(e => e.Id == id);

    if (appointment == null)
    {
        return BadRequest();
    }

    appointment.ProblemDetails = problemDetails;
    appointment.AppointmentStatusId = statusId;
    appointment.Notes = notes;
    appointment.ImageUrl = imageUrl;
    _context.Entry(appointment).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!AppointmentExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

Note: As a general rule, PUT operations should only modify data that a user should be allowed to update, not necessarily every field in an entity.

9. Open the **ScheduleEngineerController.cs** file and add the **GetScheduleEngineerAppointments** method highlighted below to the **ScheduleEngineerController** class:

```
[Route("api/[controller]")]
[ApiController]
public class ScheduleEngineerController : ControllerBase
{
    private readonly ScheduleContext _context;

    public ScheduleEngineerController(ScheduleContext context)
    {
        _context = context;
    }

    // GET: api/ScheduleEngineer/5/Appointments
    [HttpGet("{id}/Appointments")]
    public async Task<ActionResult<IEnumerable<Appointment>>>
GetScheduleEngineerAppointments(string id)
    {
        return await _context.Appointments
```

```
            .Where(a => a.EngineerId == id)
            .OrderByDescending(a => a.StartDateTime)
            .Include(c => c.Customer)
            .Include(e => e.Engineer)
            .Include(s => s.AppointmentStatus)
            .ToListAsync();
    }
    ...
}
```

These methods retrieve the appointments for the specified technician.

10. Edit the **CustomerController.cs** file and add the **GetAppointments** and **GetNotes** methods, highlighted below, to the **CustomerController** class:

```
[Route("api/[controller]")]
[ApiController]
public class CustomerController : ControllerBase
{
    private readonly ScheduleContext _context;

    public CustomerController(ScheduleContext context)
    {
        _context = context;
    }

    //GET: api/Customers/5/Appointments
    [HttpGet("{id}/Appointments")]
    public async Task<ActionResult<IEnumerable<Appointment>>>
GetAppointments(long id)
    {
        return await _context.Appointments
            .Where(a => a.CustomerId == id)
            .OrderByDescending(a => a.StartDateTime)
            .ToListAsync();
    }

    //GET: api/Customers/5/Notes
    [HttpGet("{id}/Notes")]
    public async Task<ActionResult<IEnumerable<object>>> GetNotes(long
id)
    {
        return await _context.Appointments
            .Where(a => a.CustomerId == id)
            .OrderByDescending(a => a.StartDateTime)
            .Select(a =>
                new {a.StartDateTime, a.ProblemDetails, a.Notes})
            .ToListAsync();
    }
    ...
}
```

The **GetAppointments** method finds all appointments for the specified customer. The **GetNotes** method retrieves all the technician's notes made on previous visits to the customer.

11. Edit the **appSettings.json** file and add the connection string for the **KnowledgeDB** database. This string should be the same that you previously wrote to the **appSettings.Development.json** file:

```
{
  "ConnectionStrings": {
    "InventoryDB": ...,
    "KnowledgeDB": ...,
```

```
    "SchedulesDB": "Server=tcp:<server
name>.database.windows.net,1433;Initial Catalog=SchedulesDB;Persist
Security Info=False;User
ID=sqladmin;Password=Pa55w.rd;MultipleActiveResultSets=False;Encrypt=Tru
e;TrustServerCertificate=False;Connection Timeout=30;"

  },
  "Logging": {
    ...
  },
  "AllowedHosts": "*"
}
```

12. In the **Terminal** window, compile and build the Web API:

```
dotnet build
```

    The Web API should build without reporting any errors or warnings.

13. In the **Terminal** window, package the Web API ready for deployment to Azure:

```
dotnet publish -c Release -o ./publish
```

14. In Visual Studio Code, right-click the **publish** folder, and then select **Deploy to Web App**. Deploy to the same Azure Web app that you created previously. Allow the wizard to overwrite the existing web app with the new code.

13. When deployment is complete, browse to the website but change the URL in the browser to **https://*<webapp name>*.azurewebsites.net/swagger**. Verify that the operations for the **Appointments**, **Customer**, and **ScheduleEngineer** controllers are now available.

The Web API is now ready to be incorporated into the app.

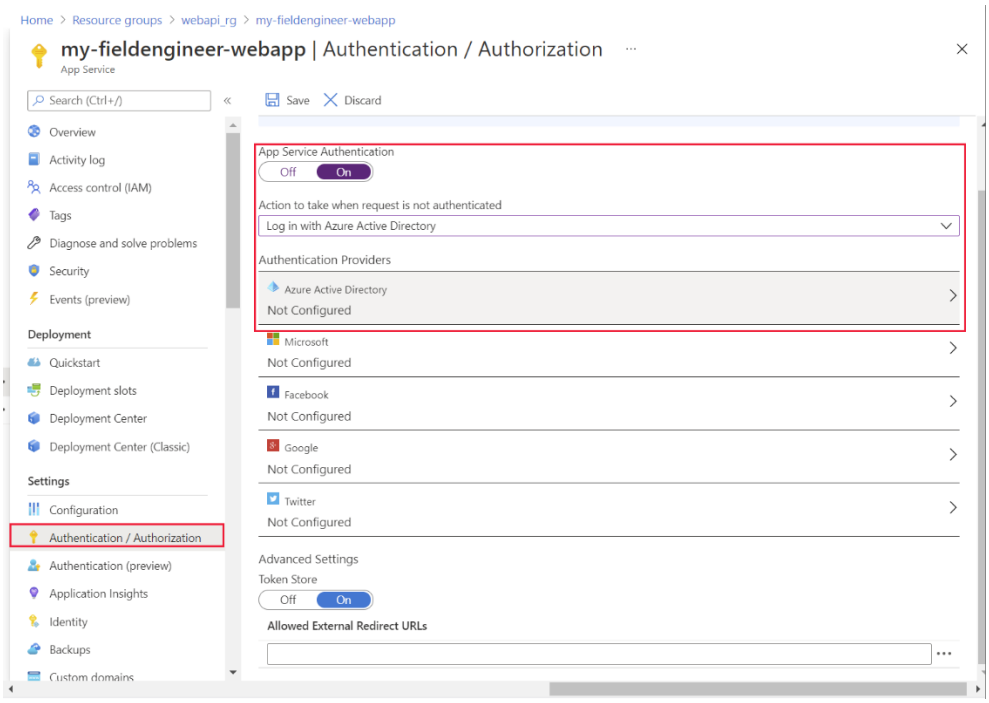
CHAPTER 6: USING THE WEB API IN POWER APPS

Maria and Kiana are ready to combine the app with the Web API. However, before proceeding, they decide to consult with Preeti, the IT Operations Manager.

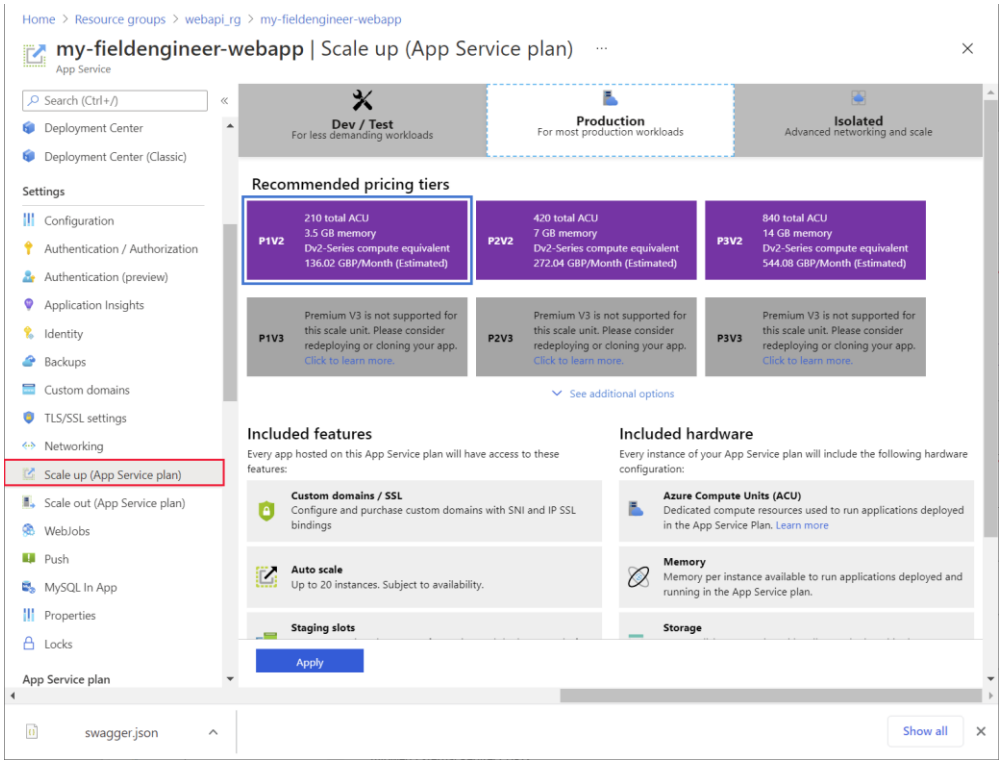
UNDERSTANDING THE IT OPERATIONS MANAGEMENT REQUIREMENTS FOR THE WEB API

Preeti is concerned that the app and the Web API must be secure because they provide access to sensitive data stored in the various databases. She wants assurances that she will be able to include authentication and authorization, to prevent unwarranted access to information. Preeti is also aware that the company is rapidly expanding, and the volume of data involved in managing customers, appointments, parts, and the knowledge base is likely to increase exponentially in the near term. Consequently, she wants the solution to be scalable.

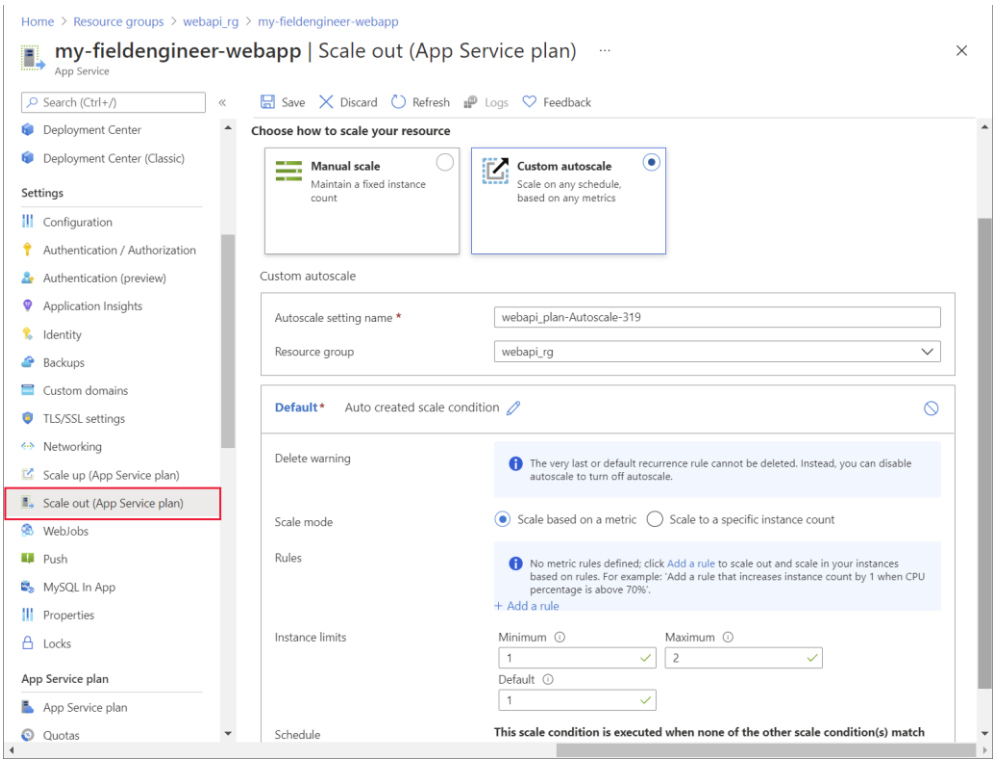
Kiana explains to Preeti that the Web API is currently implemented as an Azure App Service. This service supports a number of authentication providers, which Preeti can configure using the Azure portal. Preeti is especially interested in Azure Active Directory because VanArsdel are looking to roll out this form of authentication to many of their other corporate systems in the near future:



Azure App Service also provides horizontal and vertical scalability. If needed, Preeti can scale up the resources available to the Web API by upgrading the App Service Plan for the web app:



She can also arrange for the system to scale out by configuring autoscaling. Azure App Service enables an operations manager to define autoscale rules that determine the conditions under which the system should scale out across more instances when the load increases, or back in again as demand drops. She can also configure pre-emptive autoscaling to occur according to a schedule:



A key part of the role of an IT Operations Manager is to have an eye for how systems might evolve, and to ensure that the underlying support structures will handle future expansion and changes. Preeti knows that the Web API developed by Kiana might be extended, and reused by other VanArsdel systems in the future. She needs to be able to manage and control the way in which developers request use of the Web API, protect it as

a valuable resource, and monitor its use. Therefore Preeti decides to protect the Web API behind the Azure API Management service (APIM).

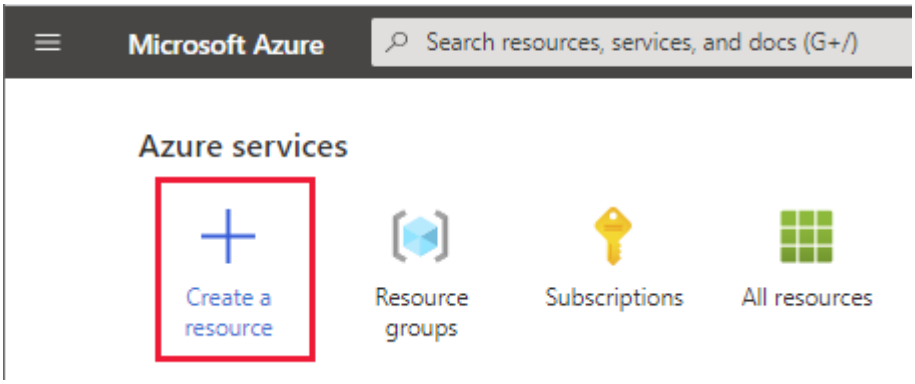
APIM provides an extra layer of security to a Web API, as well as enabling detailed monitoring and control over which clients can access which operations. Using APIM, Preeti can manage resource utilization, and throttle the performance of low priority clients to ensure that critical higher priority apps are serviced more quickly.

> **Note**: For more details on the services that APIM provides, read **About API Management** at https://aka.ms/AAbvfzn.

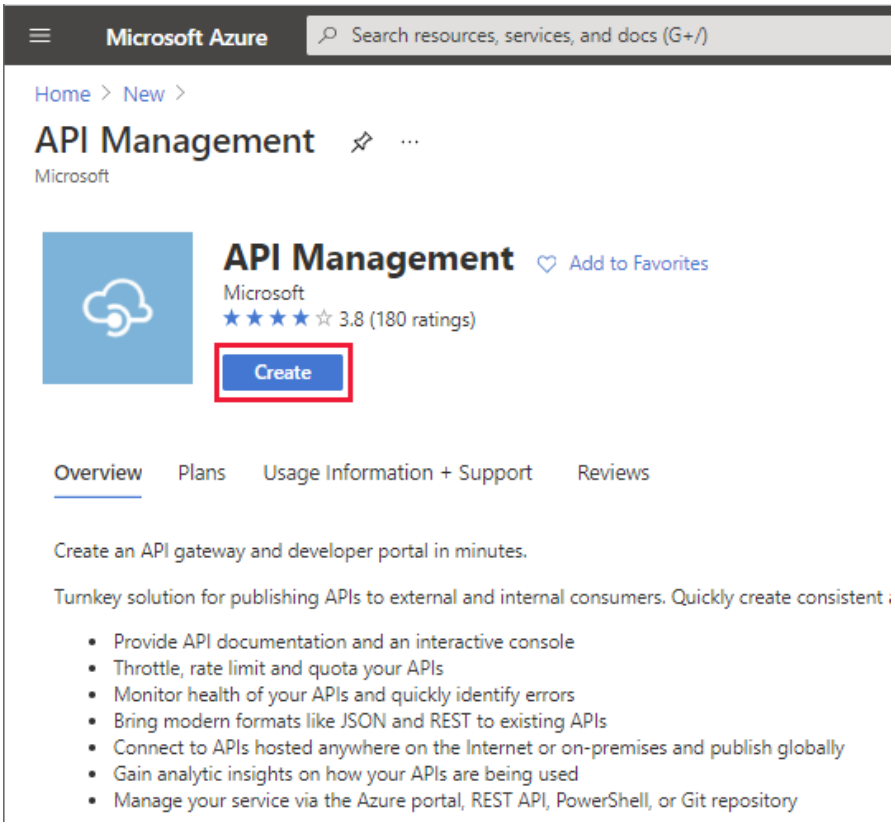
CREATING AN AZURE API MANAGEMENT SERVICE

Preeti created the API Management service through the Azure portal, using the following steps:

1. Sign in to the Azure portal and, on the **Home** page, select **Create a resource**.



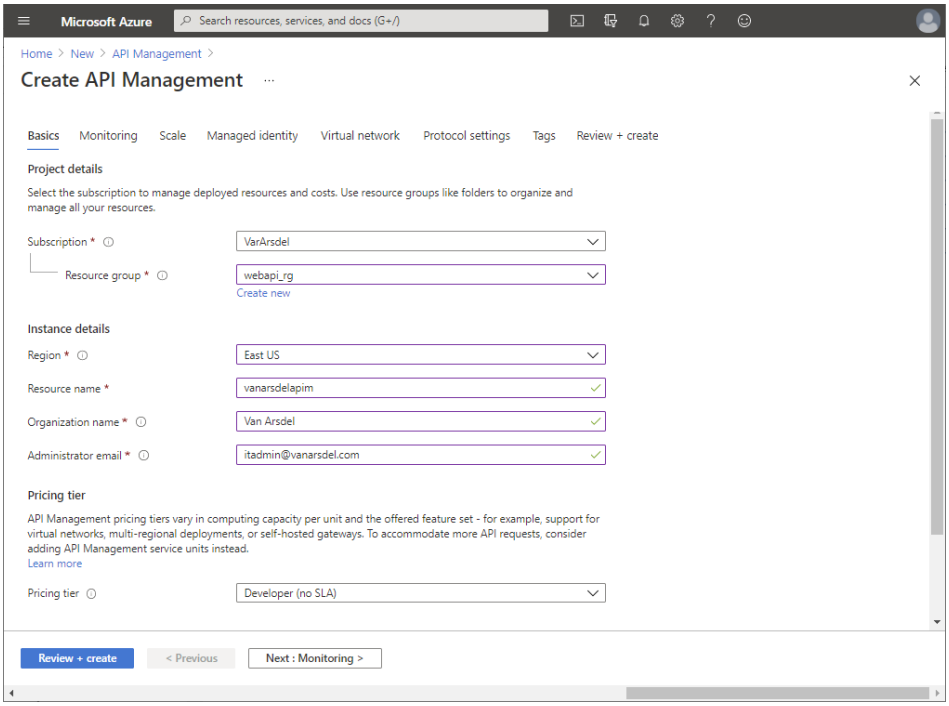
2. In the **Search the MarketPlace** text box, enter **API Management**, and press **Enter**.
3. On the **API Management** page, select **Create**.



4. On the **Create API Management** page, enter the following values, and then select **Review + create**:

   - Subscription: Select your subscription
   - Resource group: **webapi_rg** (this is the same resource group that you created for the App Service)
   - Region: Select your nearest region
   - Resource name: Enter a unique name for the service
   - Organization name: **VanArsdel**
   - Administrator email: **itadmin@vanarsdel.com**
   - Pricing tier: **Developer (no SLA)**

> **Note**: Don't use the **Developer** pricing tier for a production system.



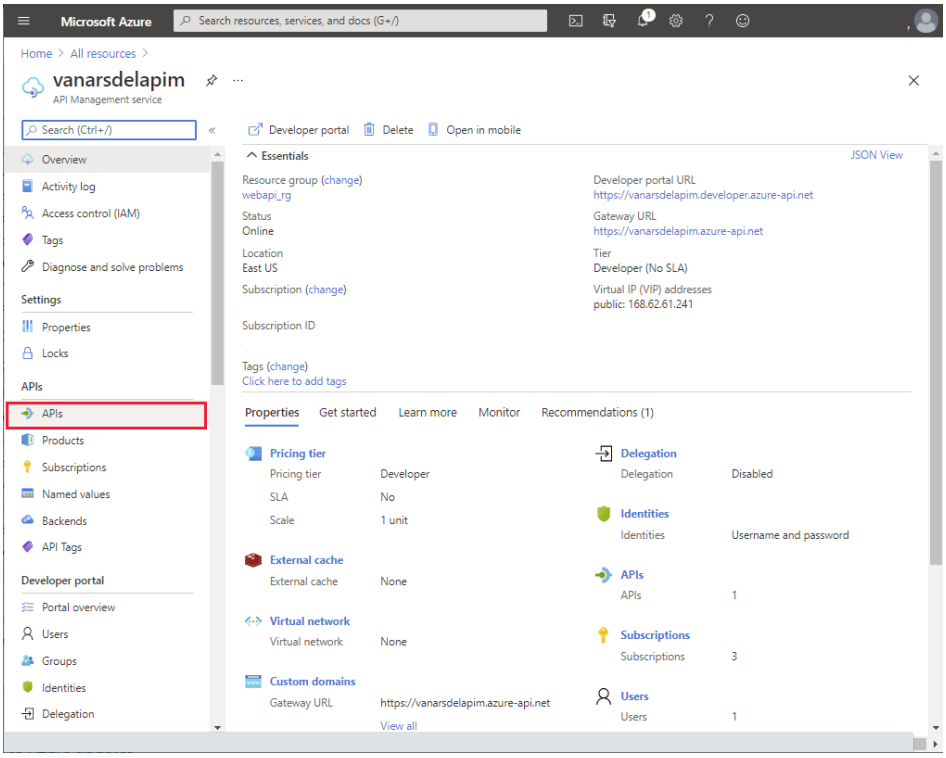
5. On the validation page, select **Create**, and wait while the API Management service is created.

> **Note**: It can take 30 minutes or more for the API Management service to be provisioned, so be patient.

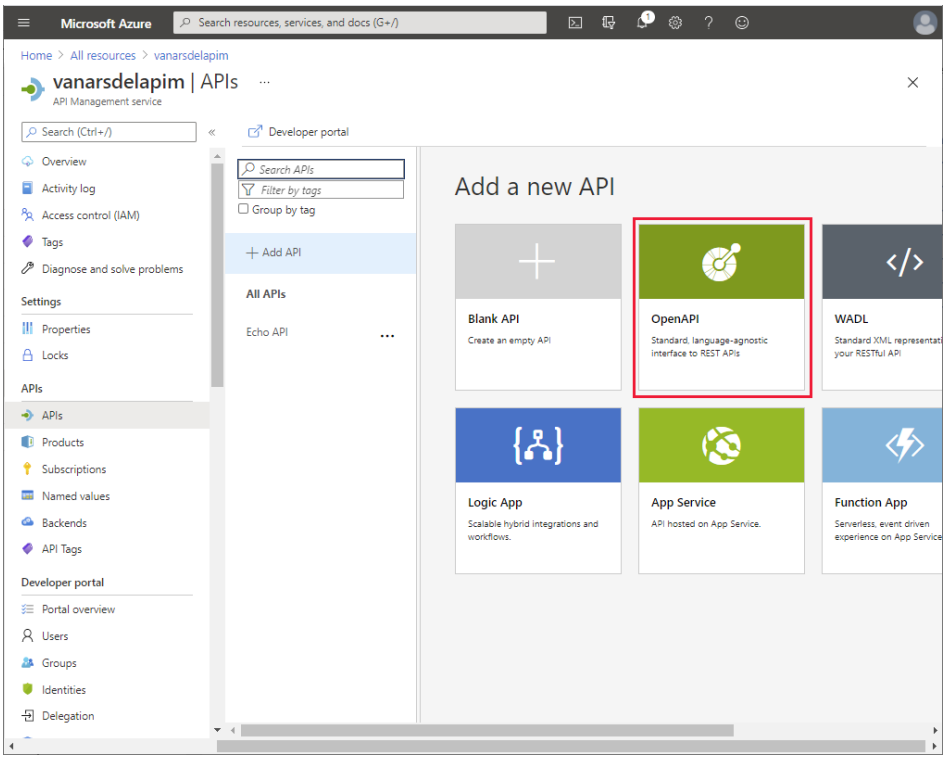
PUBLISHING THE WEB API THROUGH APIM

After the APIM service was created, Preeti published the Web API to make it accessible to other services and applications. She used the following steps:

1. In the Azure portal, go to the APIM service.
2. On the **API Management service** page, in the left pane, under **APIs**, select **APIs**:



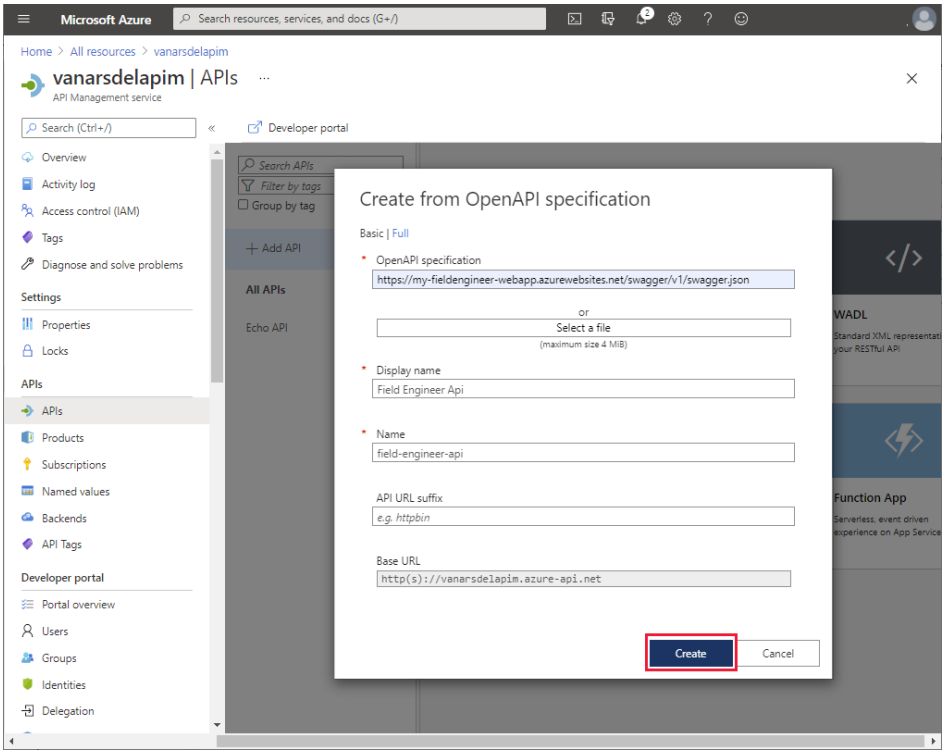
3. In the **Add a new API** pane, select **OpenAPI**:



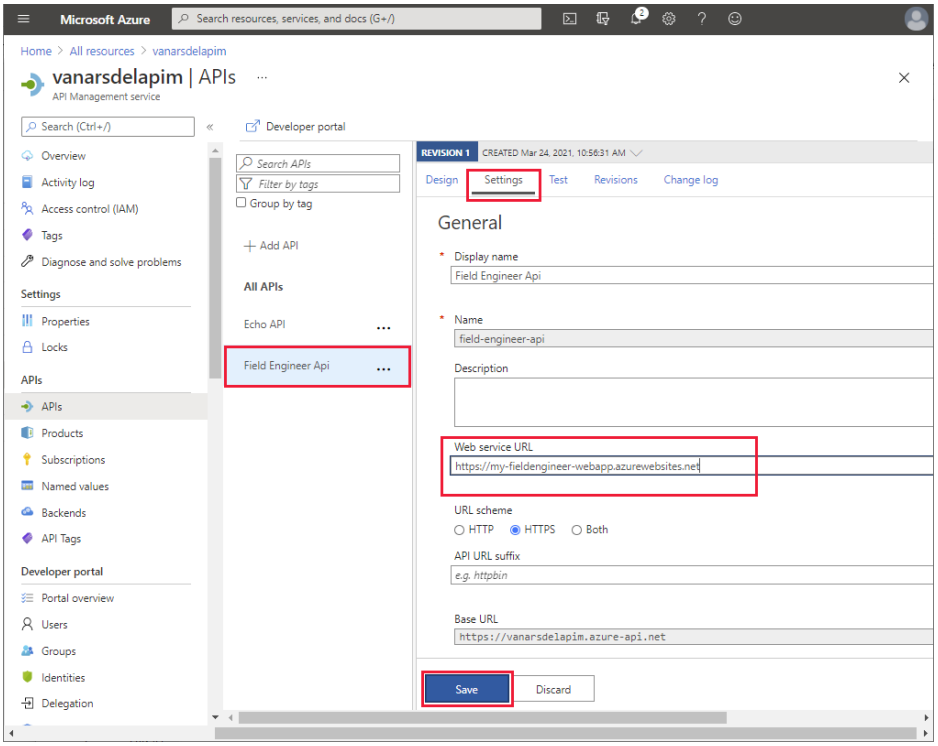
4. In the **Create from OpenAPI specification** dialog box, enter the following values, and then select **Create**:

   - OpenAPI specification: **https://*<webapp name>*.azurewebsites.net/swagger/v1/swagger.json**, where *<webapp name>* is the name of the App Service hosting your Web API
   - Display name: **Field Engineer API**
   - Name: **field-engineer-api**
   - API URL suffix: Leave empty

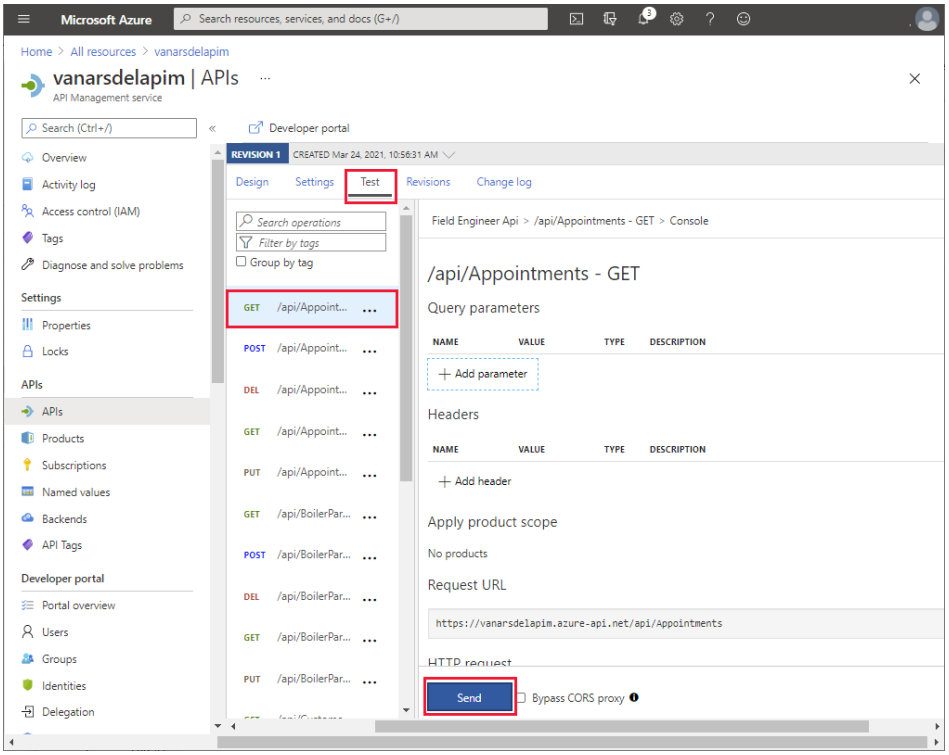
- Base URL: Use the default URL



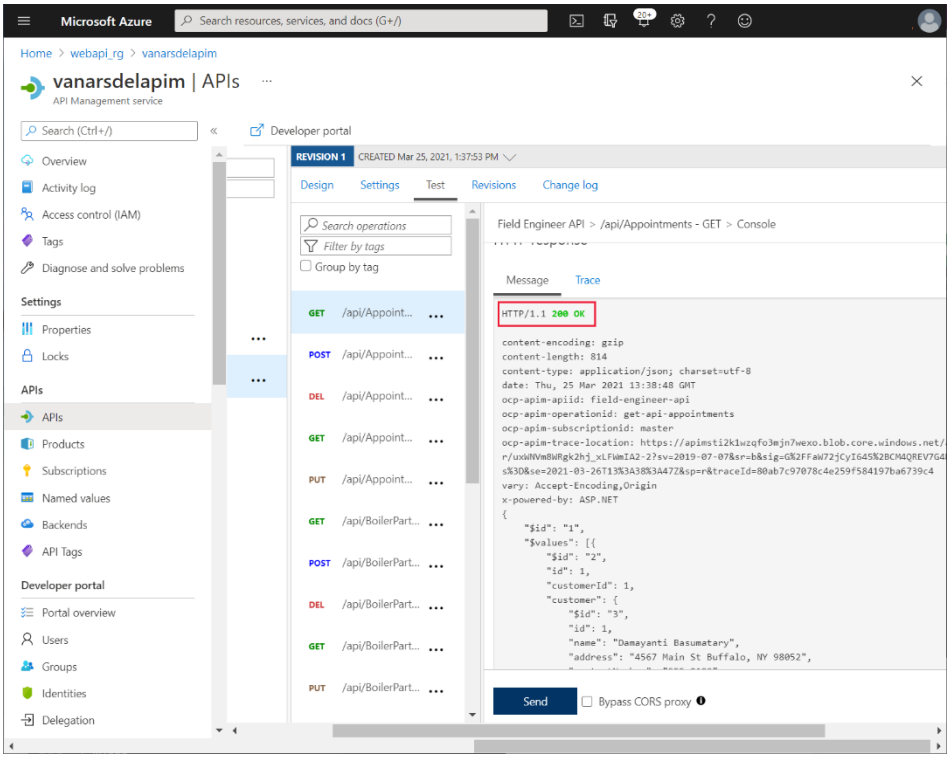
5. When the Field Engineer API has been created, select the **Settings** tab for the API, set the **Web Service URL** to **https://*<webapp name>*.azurewebsites.net**, and then select **Save**:



6. On the **Test** tab, select the **GET api/Appointments URI**, and then select **Send**:



7. Verify that the request is successful (the HTTP return code is **200 OK**), and that it returns a result containing a list of appointments in the response body:

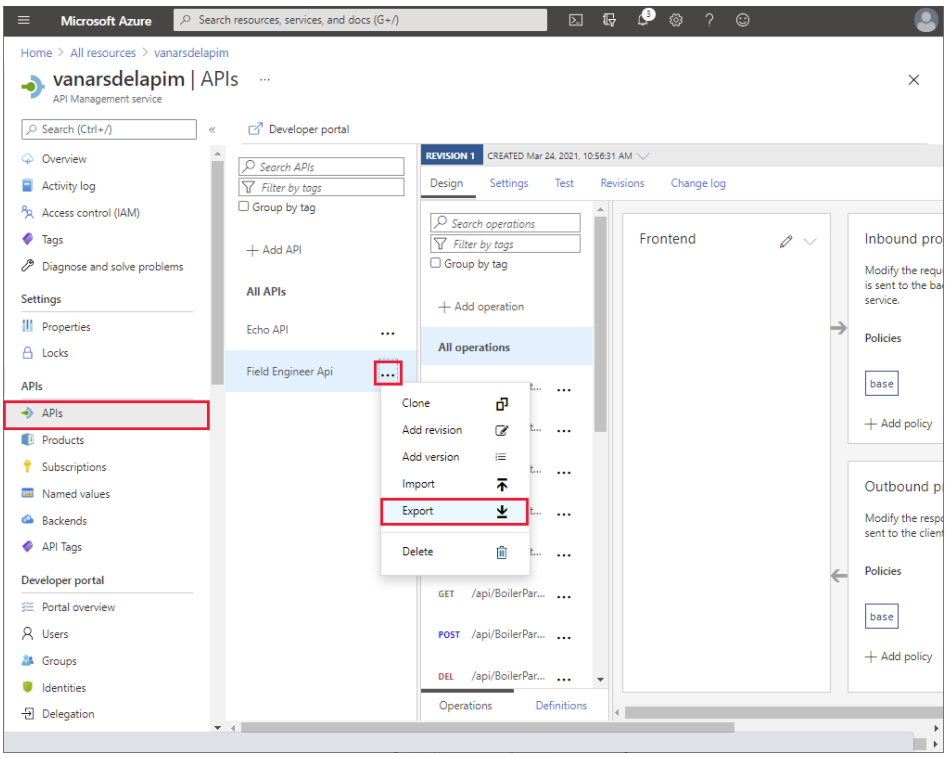


CONNECTING TO APIM FROM THE APP

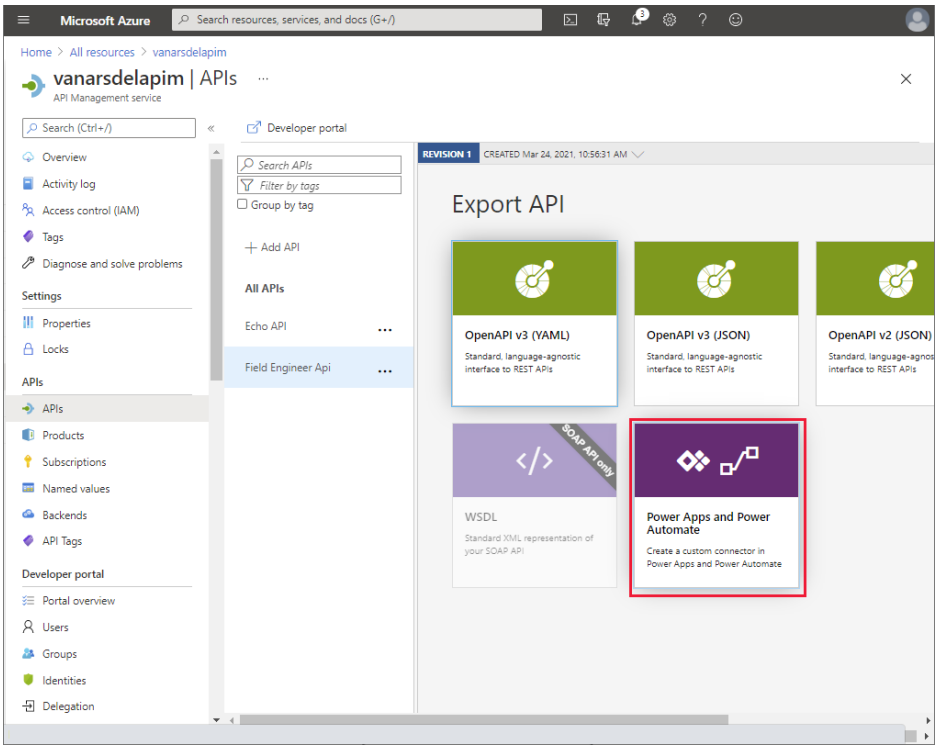
Kiana and Maria can now work together to connect the app to the Web API through the APIM service.

The first task is to create a custom connector that's used by the app to communicate with APIM. This involves exporting the API to the Power Apps environment used to create the app, which Kiana does as follows:

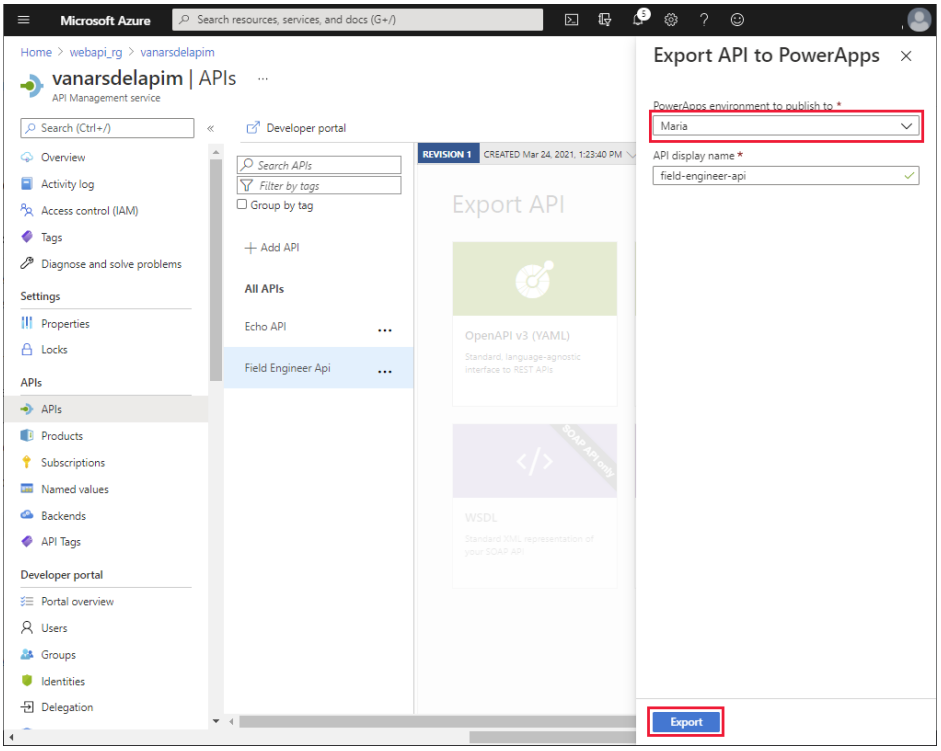
1. In the Azure portal, go to the page for the APIM service that Preeti created.
2. In the left pane, under **APIs**, select **APIs**.
3. Click the ellipsis button for the **Field Engineer Api**, and then select **Export**:



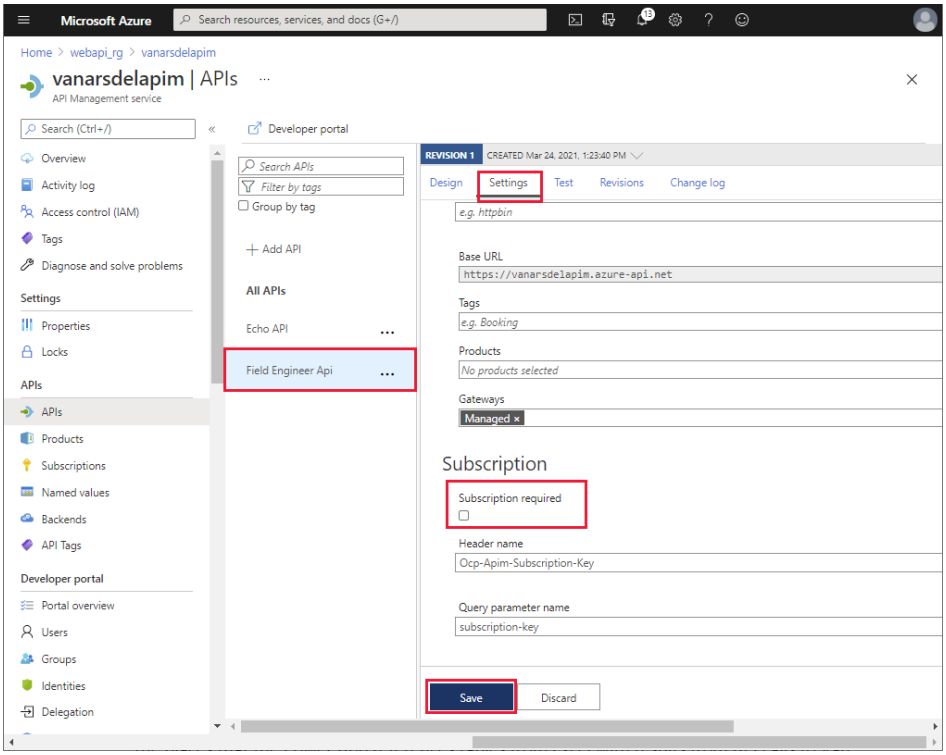
4. In the **Export API** pane, select **Power Apps and Power Automate**:



5. In the **Export API to PowerApps** pane, select the PowerApps environment in which you created the prototype app (**Maria** in the image shown below), and then select **Export**:



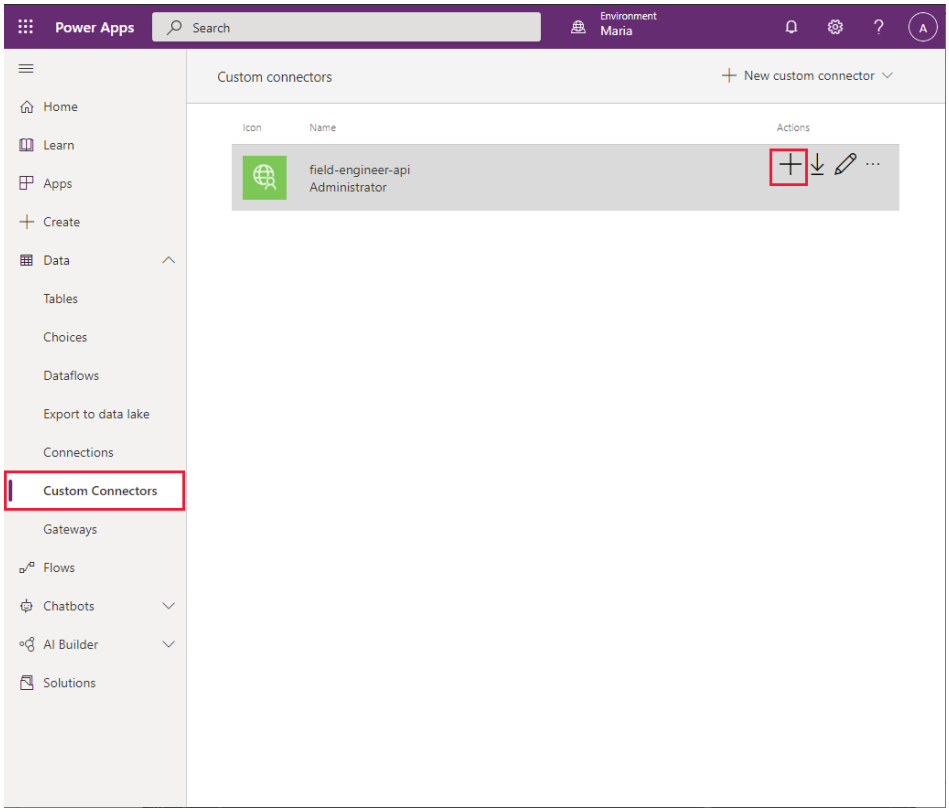
6. After the API has been exported, select the **Field Engineer API**. On the **Settings** page, scroll down to the **Subscriptions** section, clear **Subscription required**, and then select **Save**:



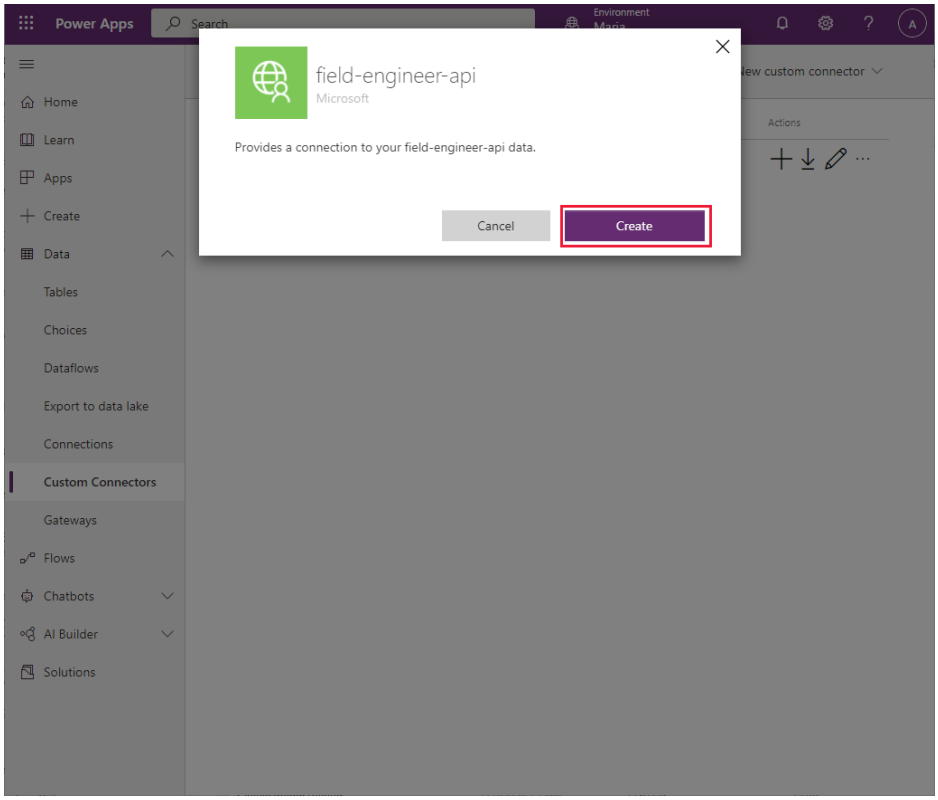
The prototype app used Excel spreadsheets for the data sources. Now the custom connector for the Web API is available, Maria performs the following steps to add the connector to the app:

1. Sign in to Power Apps Studio at [http://make.powerapps.com.](http://make.powerapps.com)

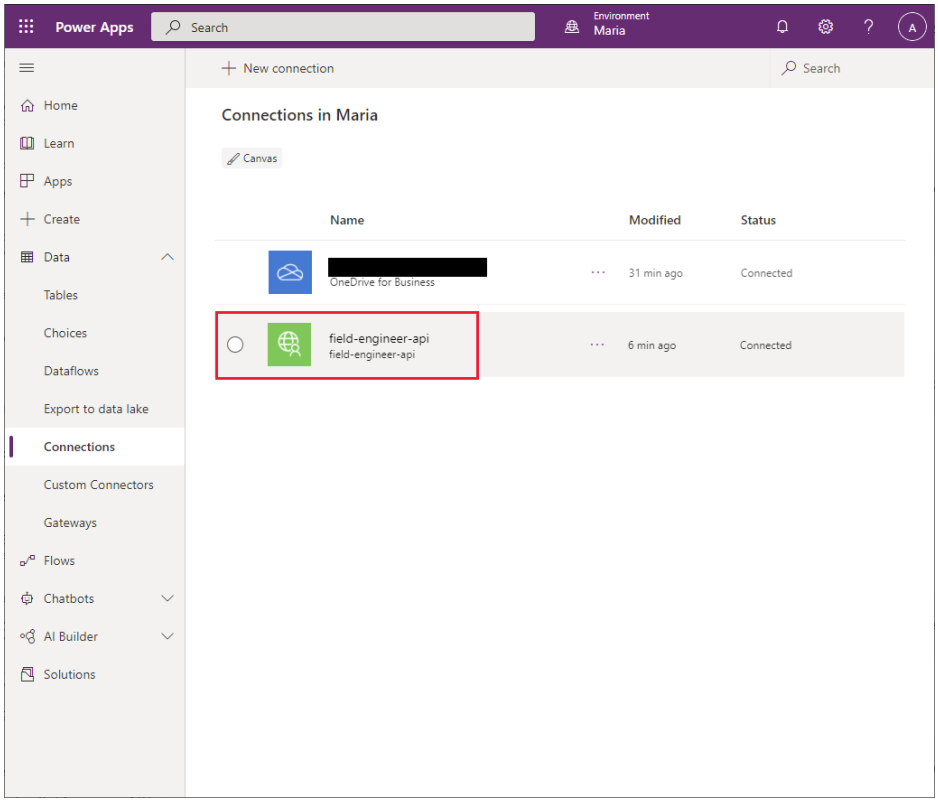
2. In the left pane, expand **Data**, and select **Custom Connectors**. The **field-engineer-api** custom connector should be listed. Select **Create connection**:



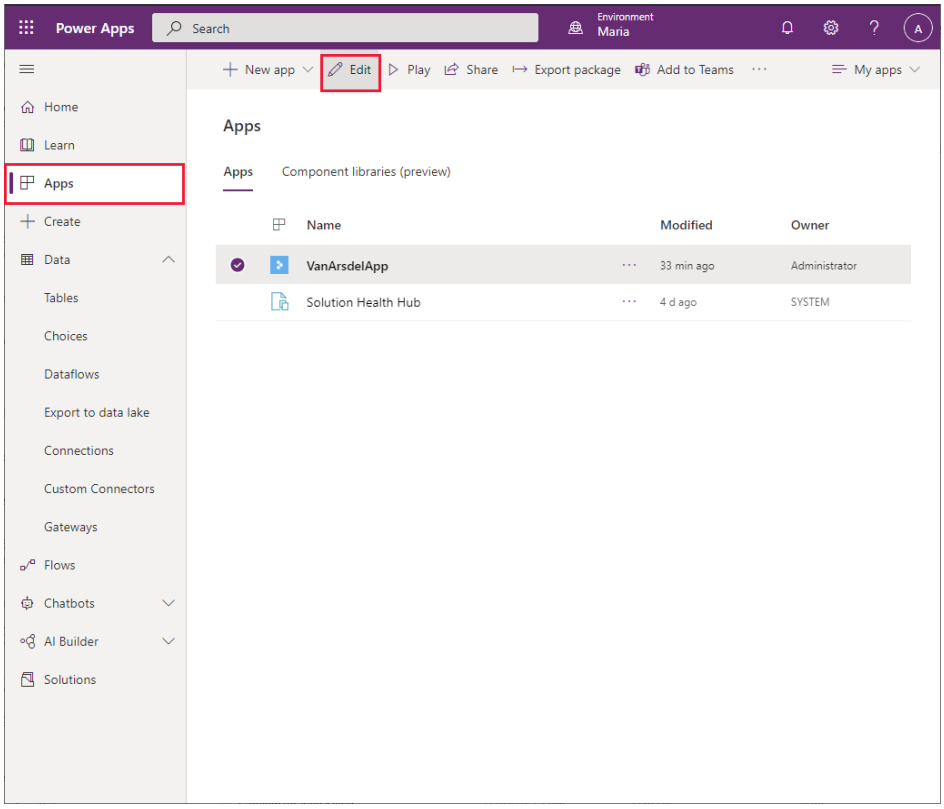
3. In the **field-engineer-api** dialog box, select **Create**:



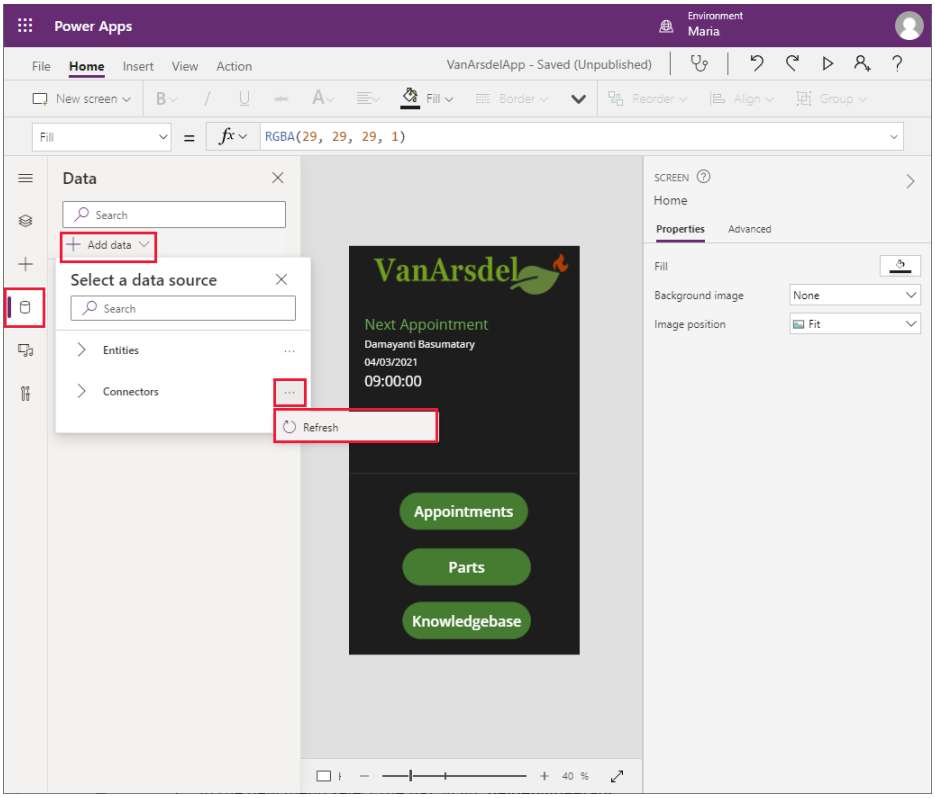
4. When the connection has been created, verify that it appears in the list of available connections:



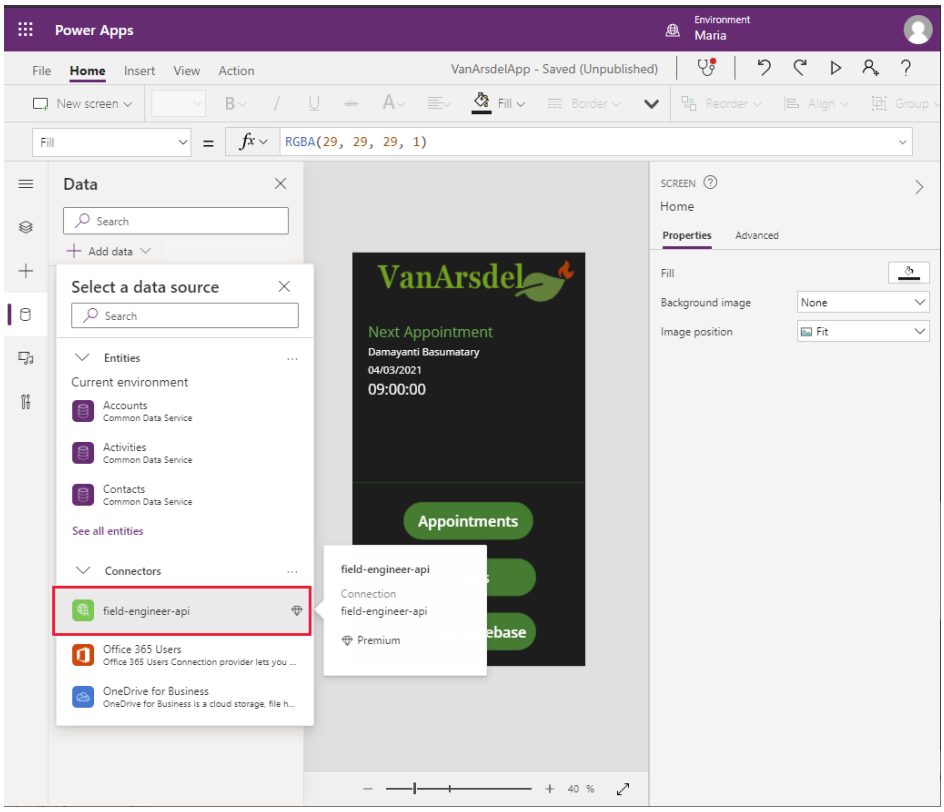
5. In the left pane, select **Apps**, select **VanArsdelApp**, and then select **Edit**:



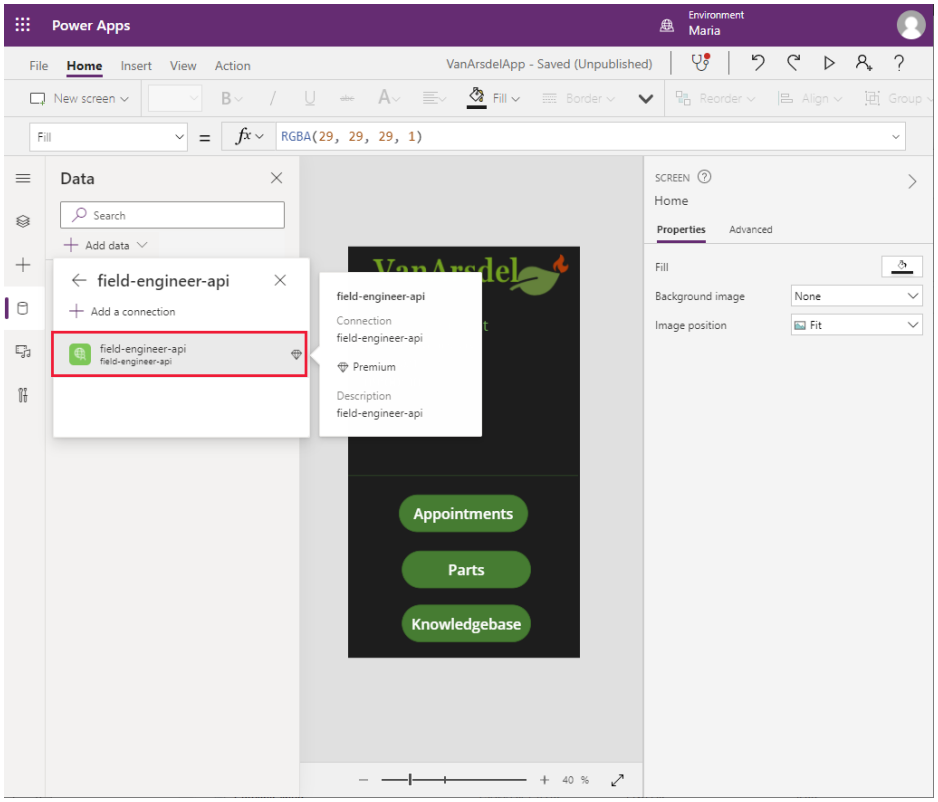
6. In the left pane, select the **Data** tab. Select **Add data**, select the ellipsis button for **Connectors**, and then select **Refresh**:



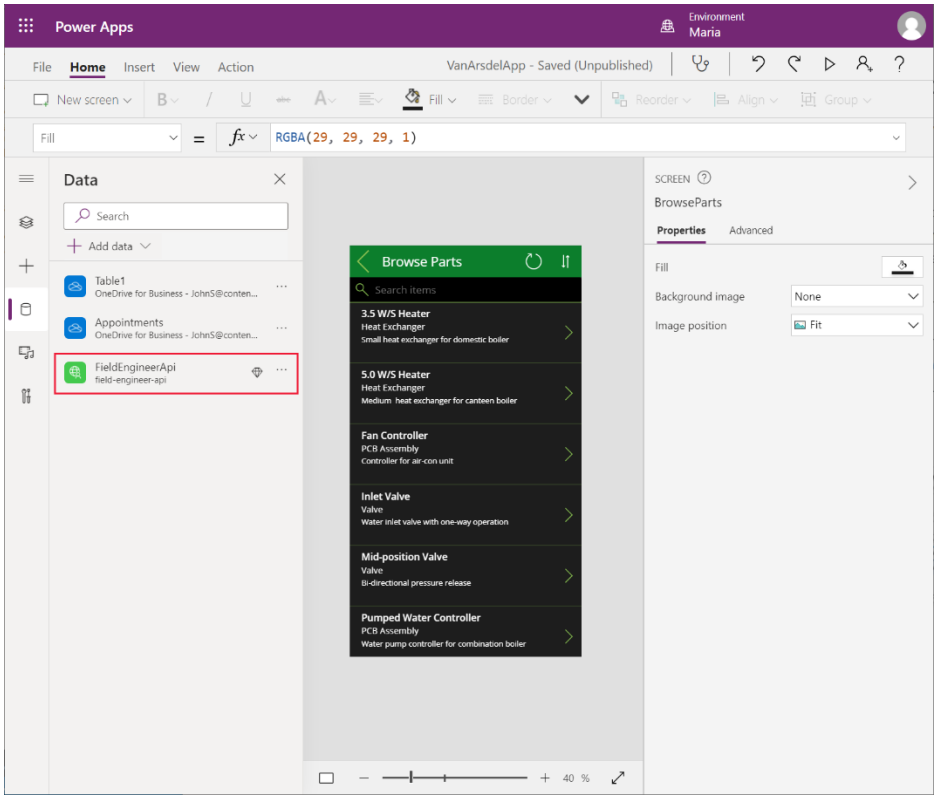
7. In the list of connectors, select the **field-engineer-api** connector:



8. In the **field-engineer-api** dialog box, select the **field-engineer-api** connector:



9. In the **Data** pane, verify that the **FieldEngineerApi** connector is listed:



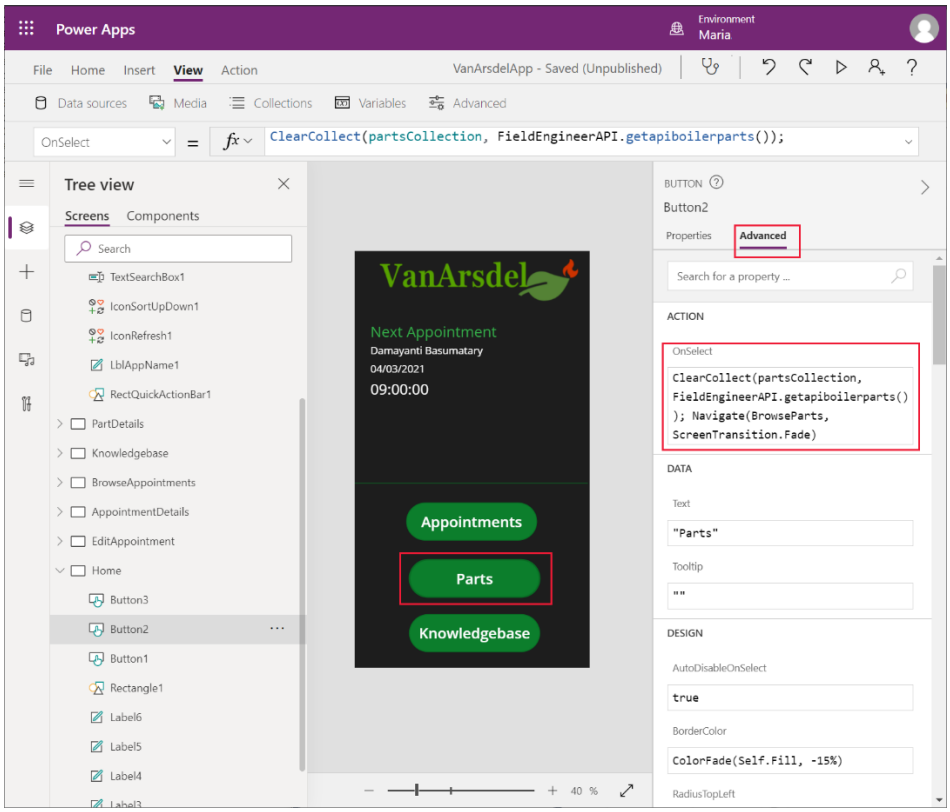
UPDATING THE APP TO USE THE CONNECTOR: FIELD INVENTORY MANAGEMENT

Now that the connection has been added to the app, Maria can modify the screens to use it to replace the Excel spreadsheets. This involves working through each screen methodically and changing the data source. No other changes should be necessary. She starts with the **BrowseParts** and **PartDetails** screens:

1. On the **Home** screen of the app, select the **Parts** button. Set the **OnSelect** action property to the following formula:

```
ClearCollect(partsCollection, FieldEngineerAPI.getapiboilerparts());

Navigate(BrowseParts, ScreenTransition.Fade)
```

The **ClearCollect** function creates a new collection named **partsCollection**, and populates it with the data that results from calling the **getboilerparts** operation in the **FieldEngineerAPI** connection.



> **Note**: It's good practice to retrieve the data into a collection and reference that collection from any screens that need the information. This approach can save different screens from repeatedly running the same query and fetching the same data.

2. Press **F5** to preview the app.

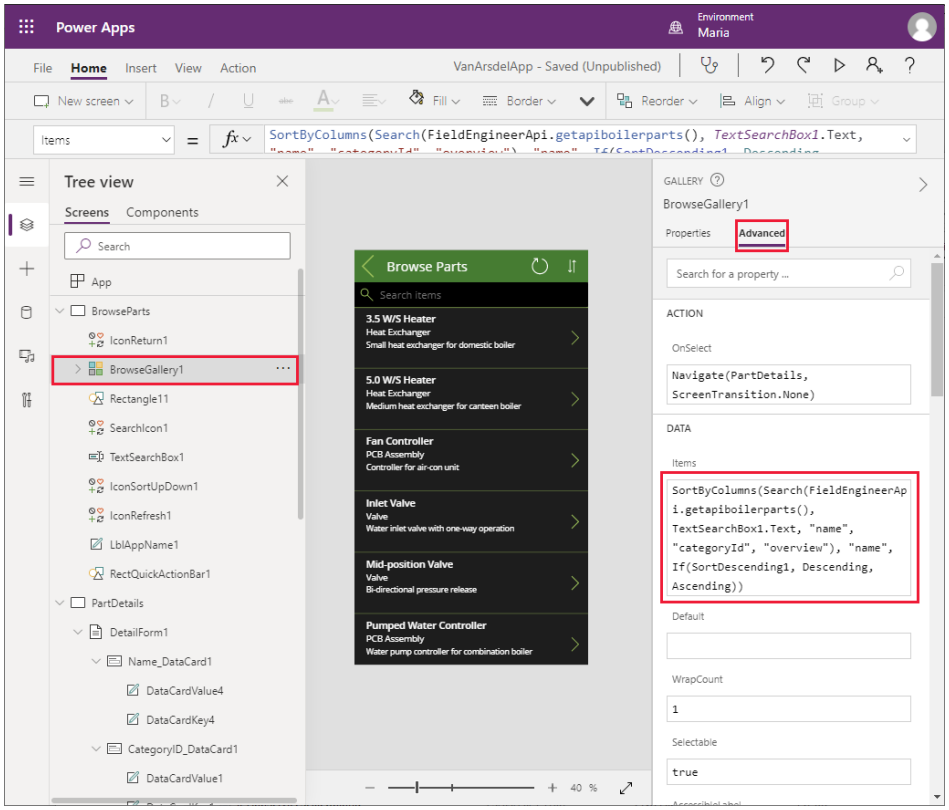
3. On the **Home** screen, select **Parts**. This action will create the **partsCollection** collection. Close the preview window and return to Power Apps Studio.

> **Note**: The purpose of this step is to enable you to see the data while you edit the **BrowseParts** screen in the following steps.

4. Select the **BrowseGallery1** control in the **BrowseParts** screen. In the formula for the **Items** property, replace the reference to the **[@Table1]** data source to **partsCollection**.

This change will result in some errors. This is because the field names in the original Excel spreadsheet used capitalization (**Name**, **CategoryID**, and **Overview**), whereas the properties returned in the body of the Web API response are named in lowercase. Change these references to use lowercase. The formula should look like this:

```
SortByColumns(Search(FieldEngineerApi.getapiboilerparts(),
TextSearchBox1.Text, "name", "categoryId", "overview"), "name",
If(SortDescending1, Descending, Ascending))
```



5. In the **Tree view** pane, select the **IconRefresh1** control. Change the **OnSelect** action to the formula **ClearCollect(partsCollection, FieldEngineerAPI.getapiboilerparts())**.

> **Note**: The original formula for this action calls the **Refresh** function to repopulate the data using the connection to the original data source. You can't use **Refresh** with a connection that runs a function to retrieve the data, so it won't work with **FieldEngineerApi.getapiboilerparts()**. The solution in this step repopulates the **partsCollection** collection with the latest data.

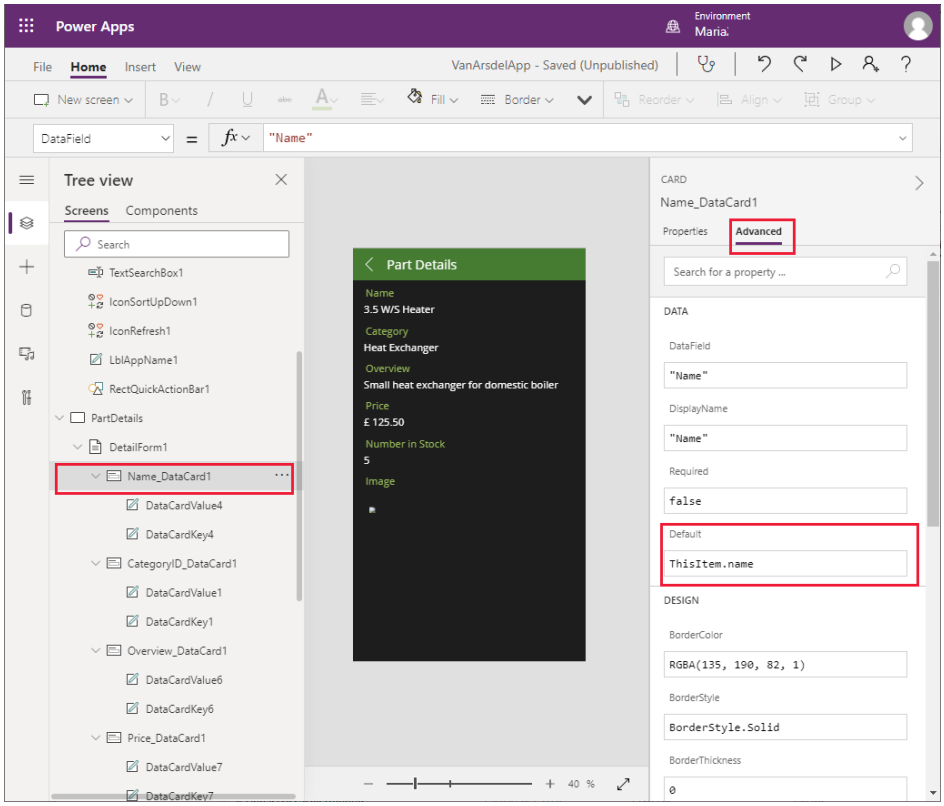
6. In the **Tree view** pane, expand the **BrowseGallery1** control, and select the **Body1** control. Change the **Text** property to **ThisItem.overview**.

7. In the **Tree view** pane, select the **Subtitle1** control. Change the **Text** property to **ThisItem.categoryId**.

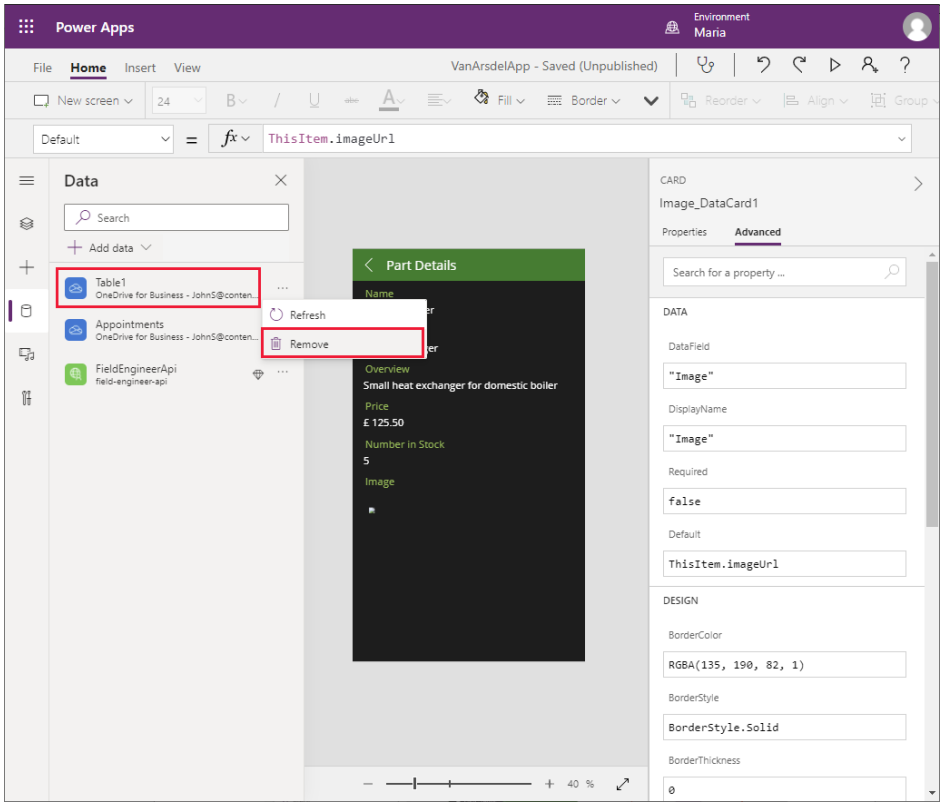
8. In the **Tree view** pane, select the **Title** control. Change the **Text** property to **ThisItem.name**.

9. In the **Tree view** pane, select the **DetailForm1** control in the **PartDetails** screen. Change the **DataSource** property from **[@Table1]** to **partsCollection**.

10. In the **Tree view** pane, select the **Name_DataCard1** control under **DetailForm1**. Change the **Default** property to **ThisItem.name**.



11. Change the **Default** property of the **CategoryID_DataCard1** control to **ThisItem.categoryId**.
12. Change the **Default** property of the **Overview_DataCard1** control to **ThisItem.overview**.
13. Change the **Default** property of the **Price_DataCard1** control to **ThisItem.price**.
14. Change the **Default** property of the **NumberInStock_DataCard1** control to **ThisItem.numberInStock**.
15. Change the **Default** property of the **Image_DataCard1** control to **ThisItem.imageUrl**.
16. In the left pane, on the **Data** tab, right-click the **Table1** data connection, and then select **Remove** to delete it from the app. This connection is no longer required.



17. Save the app.

> **Note**: You can quickly save the app without using the **File** menu by pressing **Ctrl + S**

18. Press **F5** to preview the app. The **Browse Parts** and **Part Details** screens should operate exactly as before, except this time they are retrieving data from the **InventoryDB** Azure SQL database through the Web API, rather than from a local spreadsheet.

19. Close the preview window and return to Power Apps Studio.

UPDATING THE APP TO USE THE CONNECTOR: FIELD SCHEDULING AND NOTES

Maria continues with the **BrowseAppointments**, **AppointmentDetails**, and **EditAppointment** screens. The data presented by these screens currently originates from the **Appointments** table in another Excel spreadsheet.

1. On the **Home** screen of the app, set the **OnVisible** action to the following formula:

```
ClearCollect(appointmentsCollection,
Sort(Filter(FieldEngineerAPI.getapiappointments(), DateDiff(Today(),
startDateTime) >= 0), startDateTime))
```

   This formula retrieves appointments data into the **appointmentsCollection** collection. The appointments are filtered to retrieve visits scheduled on or after the current date.

2. Select the label control that displays the time of the next appointment. Set the **Text** property to **Text(First(appointmentsCollection).startDateTime, ShortTime24)**.

3. Select the label control that displays the date for the next appointment. Set the **Text** property to **Text(First(appointmentsCollection).startDateTime, LongDate)**.

4. Select the label control that displays the date for the next appointment. Set the **Text** property to **First(appointmentsCollection).customer.name**.

5. Press **F5** to preview the app. On the **Home** screen, select **Appointments**. This action will create the **appointmentsCollection** collection. Close the preview window and return to Power Apps Studio.

6. In the **Tree view** pane, select the **BrowseAppointmentsGallery** control in the **BrowseAppointments** screen. Change the formula in the **Items** property to the formula shown below:

```
Sort(Filter(appointmentsCollection, StartsWith(customer.name,
TextSearchBox1_1.Text)), startDateTime)
```

   This formula filters the data displayed on the screen by customer name, enabling the user to enter the name of a customer. The appointments are displayed in date and time order.

7. In the **Tree view** pane, expand the **BrowseAppointmentsGallery** control, and select the **Title1_1** control. Change the **Text** property to:

```
Text(ThisItem.startDateTime, LongDate)
```

   This formula displays the date part of the **startDateTime** field for the appointment.

8. In the **Tree view** pane, expand the **BrowseAppointmentsGallery** control, and select the **Subtitle1_1** control. Change the **Text** property to:

```
Text(ThisItem.startDateTime, ShortTime24)
```

   This formula displays the time element of the **startDateTime** field.

9. In the **Tree view** pane, expand the **BrowseAppointmentsGallery** control, and select the **Body1_1** control. Change the **Text** property to:

```
ThisItem.customer.name
```

10. In the **Tree view** pane, select the **IconRefresh1_1** control on the **BrowseAppointments** screen. Set the **OnSelect** action to the following formula:

```
ClearCollect(appointmentsCollection,
Sort(Filter(FieldEngineerAPI.getapiappointments(), DateDiff(Today(),
startDateTime) >= 0), startDateTime));
```

11. In the **Tree view** pane, expand the **AppointmentDetails** screen, and select the **DetailForm1_1** control. Set the **DataSource** property to **appointmentsCollection**.

12. In the **Tree view** pane, select the **IconEdit1** control. Modify the formula in the **DisplayMode** property to test the **appoinmentsCollection** collection:

```
If(DataSourceInfo(appointmentsCollection,
DataSourceInfo.EditPermission), DisplayMode.Edit, DisplayMode.Disabled)
```

13. In the **Tree view** pane, expand the **DetailForm1_1** screen, and select the **Customer Name_DataCard1** control. Change the **Default** property to **ThisItem.customer.name**.

14. Change the **Default** properties of the remaining data cards as follows:

    - Customer Address_DataCard1: **ThisItem.customer.address**
    - Contact Number_DataCard1: **ThisItem.customer.contactNumber**
    - Problem Details_DataCard1: **ThisItem.problemDetails**
    - Status_DataCard1: **ThisItem.appointmentStatus.statusName**
    - Notes_DataCard1: **ThisItem.notes**
    - Image_DataCard1_1: **ThisItem.imageUrl**

15. In the **Tree view** pane, expand the **EditAppointment** screen, and select the **EditForm1** control. Set the **DataSource** property to **appointmentsCollection**.

16. In the **Tree view** pane, expand the **EditForm1** control, and select the **Customer Name_DataCard3** control. Change the **Default** property to **ThisItem.customer.name**.
17. Change the **Default** properties of the remaining data cards as follows:
    - Contact Number_DataCard2: **ThisItem.customer.contactNumber**; additionally, change the **MaxLength** property to **20**
    - Problem Details_DataCard2: **ThisItem.problemDetails**
    - Status_DataCard5: **ThisItem.appointmentStatus.statusName**
    - Notes_DataCard3: **ThisItem.notes**
    - Image_DataCard2: **ThisItem.imageUrl**
18. In the **Tree view** pane, expand the **Problem Details_Card2** control. Rename the **DataCardValue*X*** (*X* will be a number) field under this control to **ProblemDetailsValue**. Repeat this process for the **DataCardValue*X*** controls in the following data cards:
    - Status_DataCard5: **StatusValue**
    - Notes_DataCard3: **NotesValue**

> **Note**: The Image control will be addressed in the next chapter.

19. Select the **ProblemDetailsValue**, and set the **MaxLength** property to **100**.
20. In the **Tree view** pane, select the **IconAccept1** control on the **EditAppointment** screen. Set the **OnSelect** action property to the following formula:

```
FieldEngineerAPI.putapiappointmentsid(BrowseAppointmentsGallery.Selected
.id, {problemDetails:ProblemDetailsValue.Text,
statusName:StatusValue.Selected.Value, notes:NotesValue.Text,
imageUrl:""});

Remove(appointmentsCollection, First(Filter(appointmentsCollection,
id=BrowseAppointmentsGallery.Selected.id)));

Set(appointmentRec,
FieldEngineerAPI.getapiappointmentsid(BrowseAppointmentsGallery.Selected
.id));

Collect(appointmentsCollection, appointmentRec);

Navigate(AppointmentDetails, ScreenTransition.None);
```

This formula calls the **PUT** operation for the Appointments controller in the Web API. It passes the appointment ID for the current appointment as the first parameter, followed by the details that the user might have modified on the screen. The details are passed as a JSON object. The Remove, Set, and Collect statements update the **appointmentsCollection** collection with the data saved to the database.

> **Note**: Don't use the **ClearCollect** function to delete and refresh the entire collection in situations such as this because it's wasteful if, for example, only one record has changed.

21. In the **Tree view** pane, select the **IconAccept1** control on the **EditAppointment** screen. Set the **OnSelect** action property to:

```
ResetForm(EditForm1);

Navigate(AppointmentDetails, ScreenTransition.None);
```

22. In the left pane, on the **Data** tab, right-click the **Appointments** data connection, and then select **Remove** to delete it from the app.

23. Save the app.

24. Press **F5** to preview the app. From the **Home** screen, go to the **Appointments** screen, select and edit an appointment, then save the changes. Verify that the appointment is updated.

25. Close the preview window and return to Power Apps Studio.

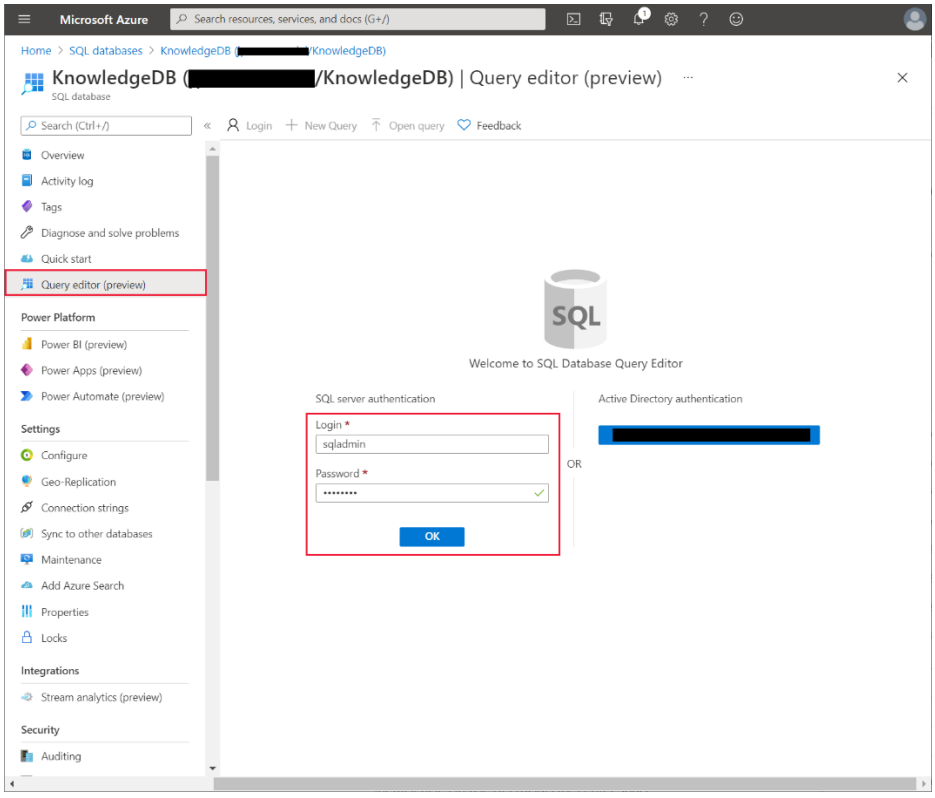
CREATING THE AZURE COGNITIVE SEARCH SERVICE FOR THE FIELD KNOWLEDGEBASE

The Knowledgebase screen in the app is not currently attached to any data source. The Web API includes operations for querying and updating the **Tips**, **BoilerParts**, and **Engineers** tables in the **KnowledgeDB** database. However, the purpose of the **Query** screen in the app is to support searches through all of these tables. The volume of data in these tables is likely to increase quickly, so Maria, Kiana, and Preeti decide to deploy Azure Cognitive Search to support this feature. An app can submit queries and receive results from Azure Cognitive Search through a custom connector.

Azure Cognitive Search works best if the data to be searched is contained in a single database entity. Kiana creates a view in the **KnowledgeDB** database that presents a unified view of the **Tips**, **BoilerParts**, and **Engineers** tables, as follows:

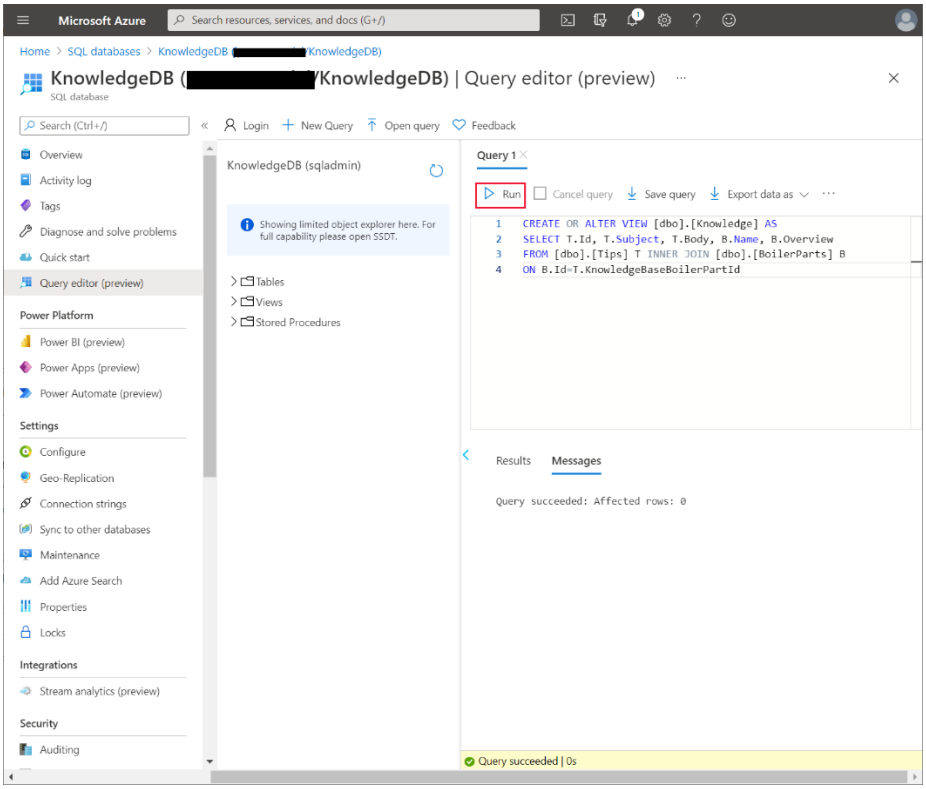
1. In the Azure portal, go to the **KnowledgeDB SQL Database** page.

2. In the left pane, select **Query Editor** and sign in to the database as **sqladmin**, using the password **Pa55w.rd**.



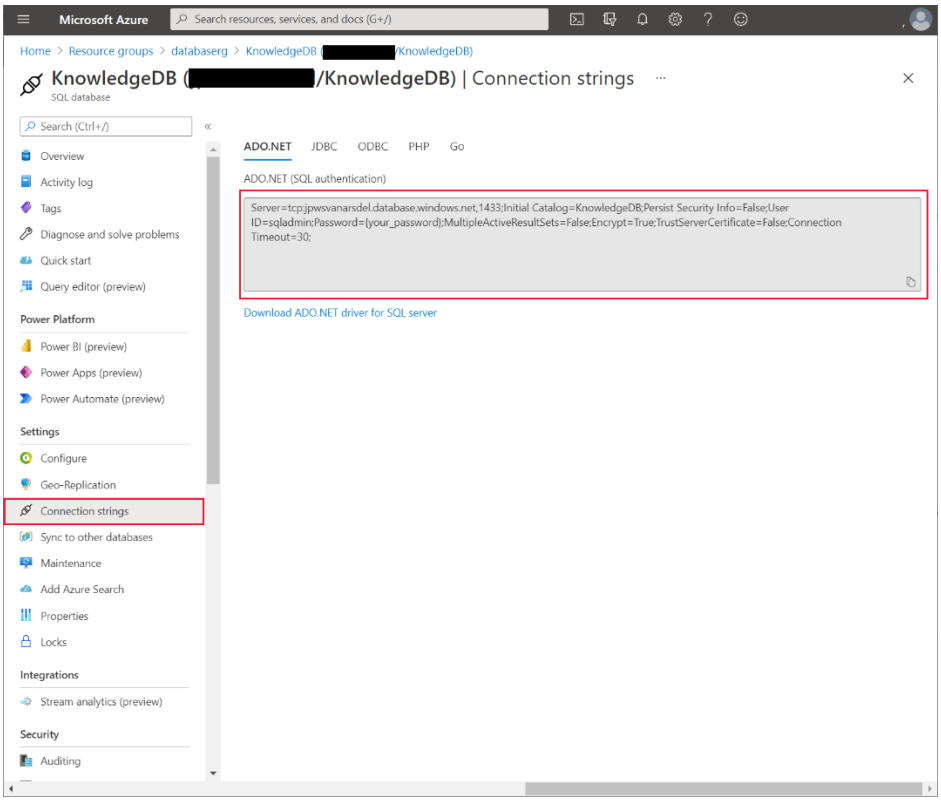
3. In the query window, enter the following statement, and then select **Run**:

```
CREATE OR ALTER VIEW [dbo].[Knowledge] AS
SELECT T.Id, T.Subject, T.Body, B.Name, B.Overview
FROM [dbo].[Tips] T INNER JOIN [dbo].[BoilerParts] B
ON B.Id=T.KnowledgeBaseBoilerPartId
```



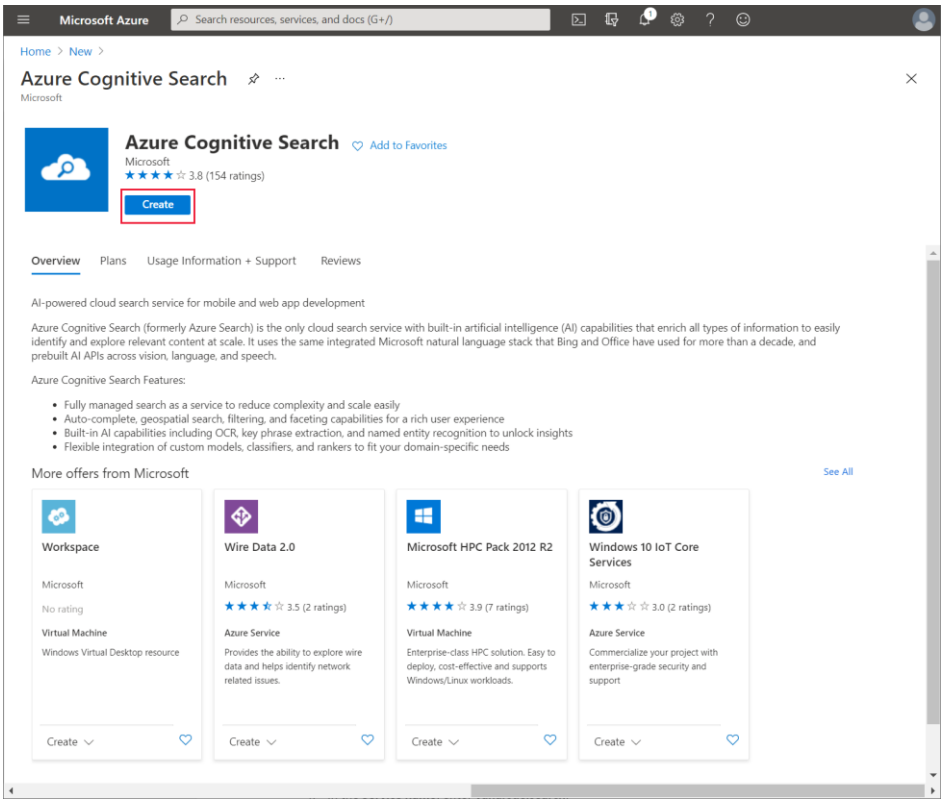
Verify that the view, **Knowledge**, is created successfully.

4. In the left pane, select **Connection strings**. Make a note of the **ADO.NET** connection string; you'll need it when you configure **Azure Cognitive Search**:



Working with Kiana, Preeti configures a new instance of the Azure Cognitive Search service to perform searches on rows in the **Knowledge** view:

1. On the **Home** page, in the Azure portal, select **+ Create a resource**, type **Azure Cognitive Search**, press **Enter**, and then select **Create**:

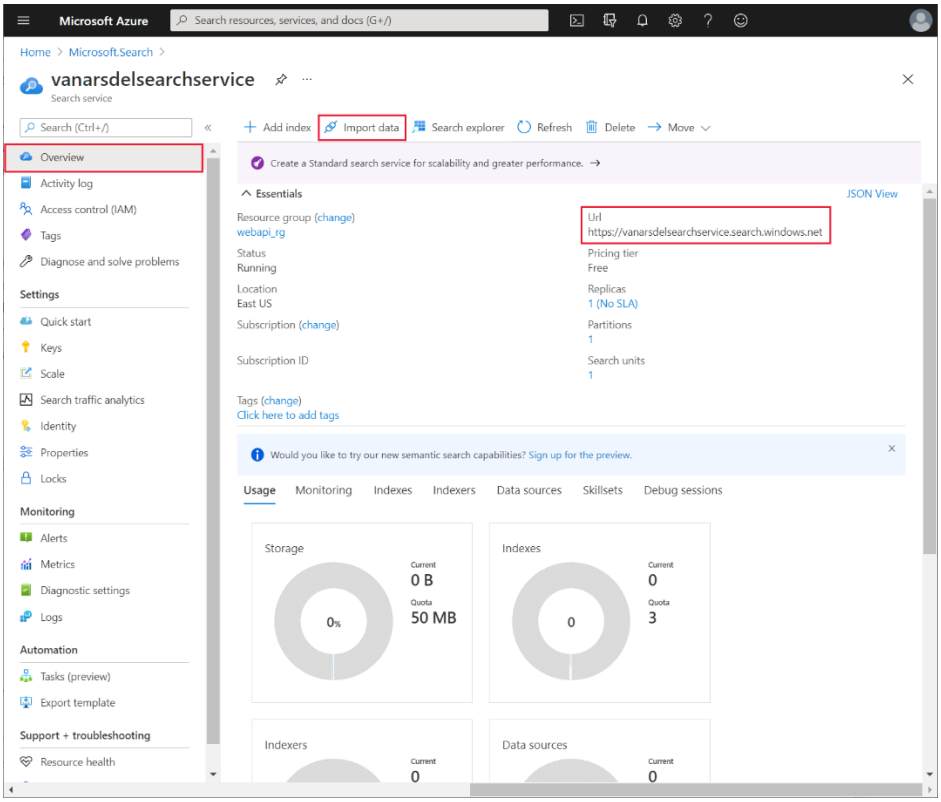


2. On the **New Search Service** page, enter the following settings, and then select **Review + create**:

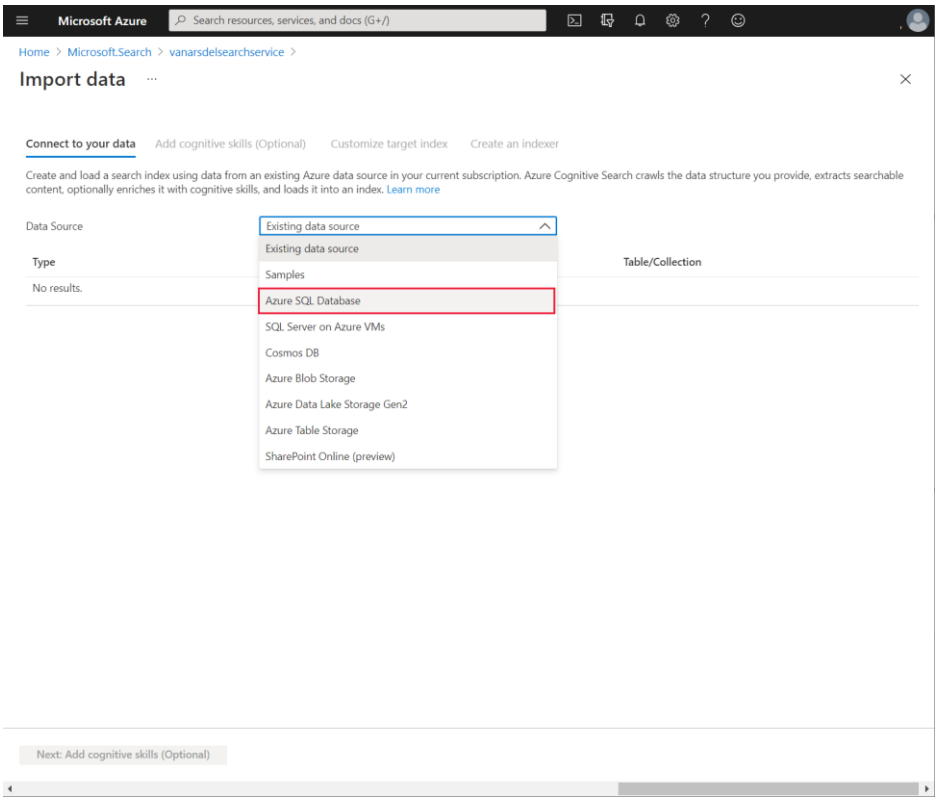
   - Subscription: Select your Azure subscription
   - Resource group: **webapi_rg**
   - Service name: Enter a unique name for the service
   - Location name: Select your nearest region
   - Pricing tier: **Free**

3. On the validation page, select **Create**, and wait while the service is provisioned.

4. Go to the page for the new search service, select **Overview**, make a note of the **Url** (you'll need this later when you create the custom connector for Power Apps), and then select **Import Data**:



5. On the **Import data** page, in the Data Source drop-down list box, select **Azure SQL Database**:

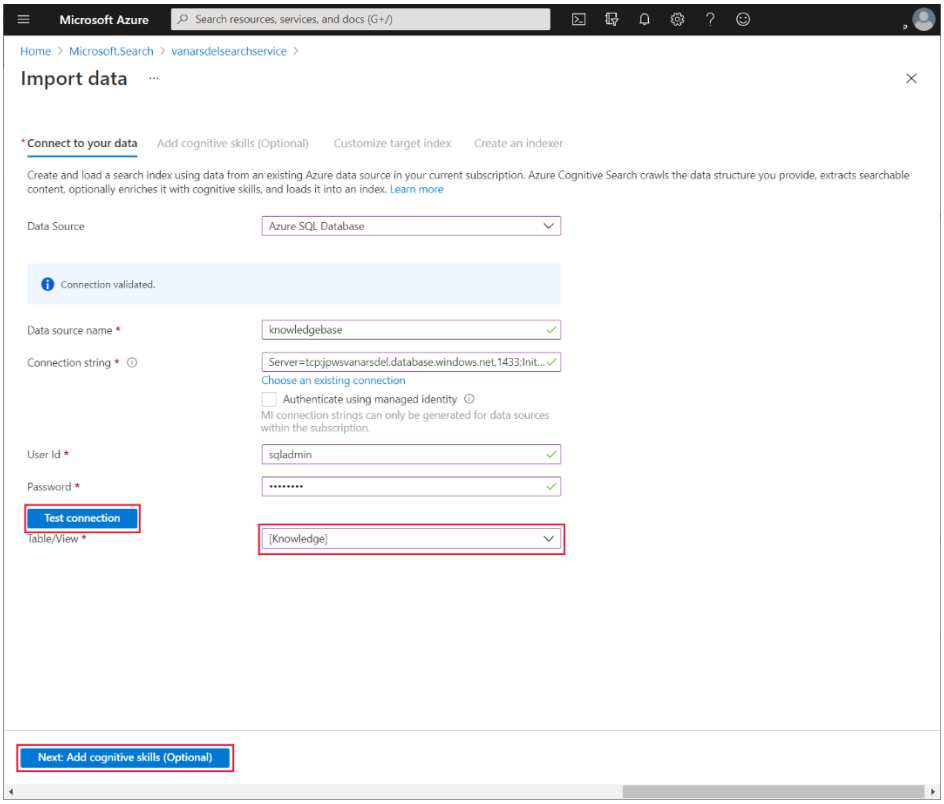


6. On the **Connect to your data** page, specify the following settings:

   - Data Source: **Azure SQL Database**
   - Data source name: **knowledgebase**

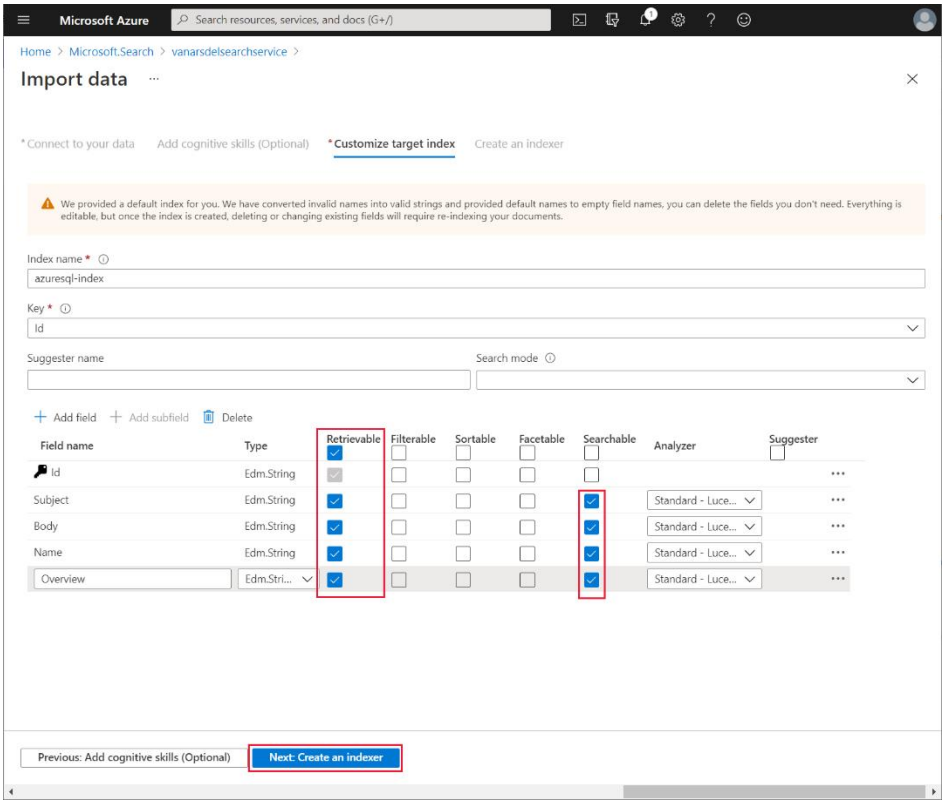
- Connection string: Enter the Azure SQL Database connection string for the **KnowledgDB** database that you recorded earlier; in this string, make sure to set the password to **Pa55w.rd**
- Leave the **User Id** and **Password** fields at their default values; these items are retrieved from the connection string

7. Select **Test connection**. Ensure that the test is successful, select the **[Knowledge]** view in the **Table/View** drop-down list box, and then select **Next: Add cognitive skills (Optional)**:

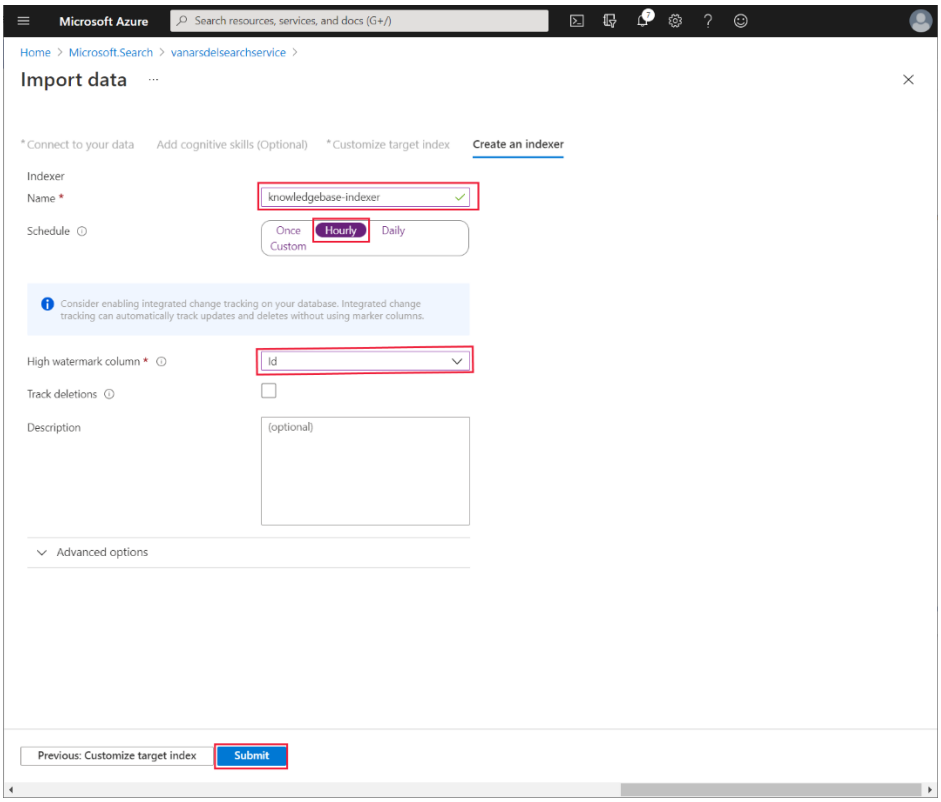


8. On the **Add cognitive skills (Optional)** page, select **Skip to: Customize target index**.

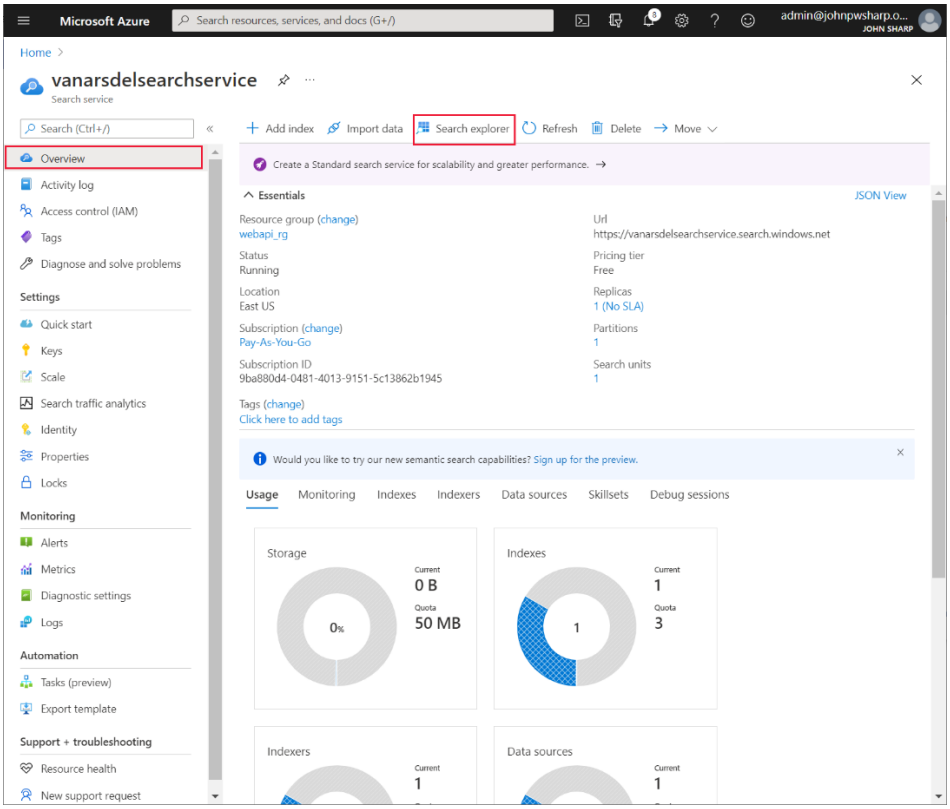
9. On the **Customize target index** page, select **Retrievable** for all columns, and **Searchable** for **Subject**, **Body**, **Name**, and **Overview**. Select **Next: Create an indexer**:



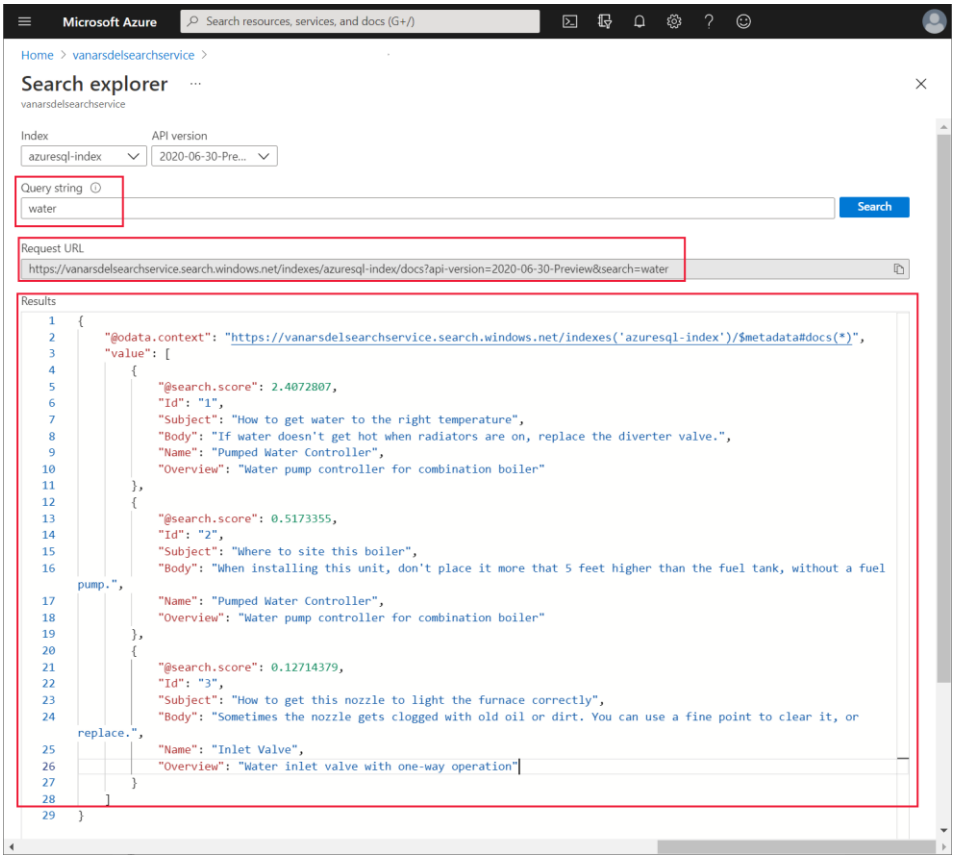
10. On the **Create an indexer** page, change the indexer **Name** to **knowledgebase-indexer**. For the **Schedule**, select **Hourly**, set the **High watermark column** to **Id**, and then select **Submit**:



11. To test the indexer, on the **Overview** page for the search service, select **Search Explorer**:



12. In the **Query string** field, enter a word to search for in the knowledge base, and then select **Search**. The search service should generate a list of documents with a match in the **Subject**, **Body**, **Name**, or **Overview** fields, and display them in the **Results** pane. Make a note of the **Request URL** and the sample **Results**; you'll need these items later as an example request and response when you set up the Power Apps custom connector:

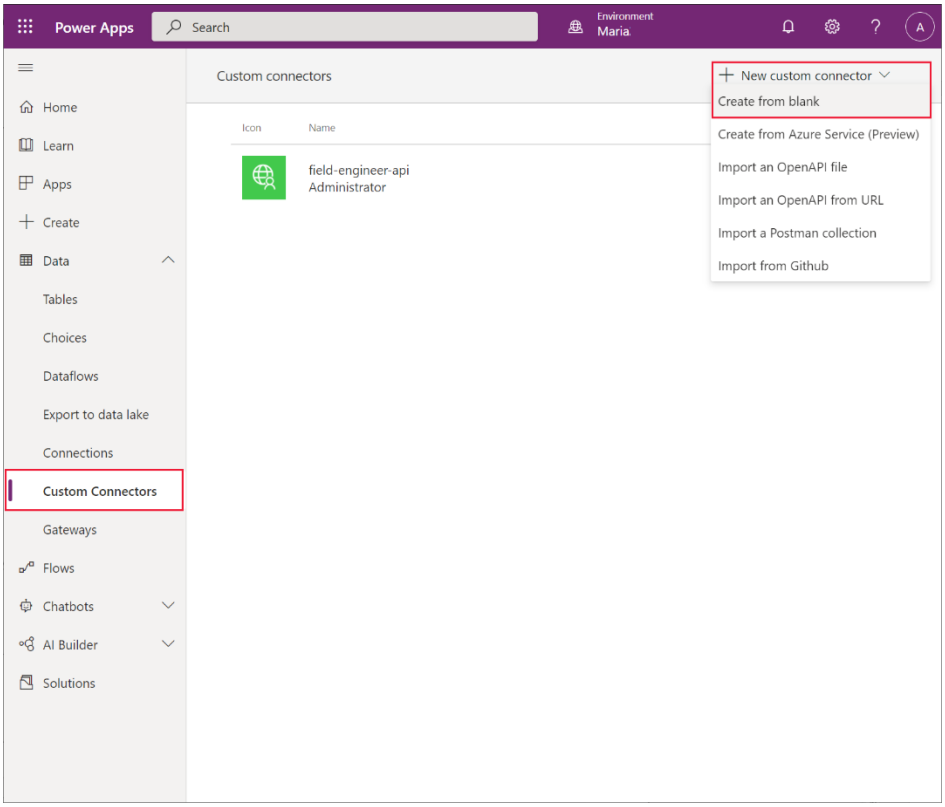


CREATING THE CUSTOM CONNECTOR FOR THE AZURE COGNITIVE SEARCH SERVICE

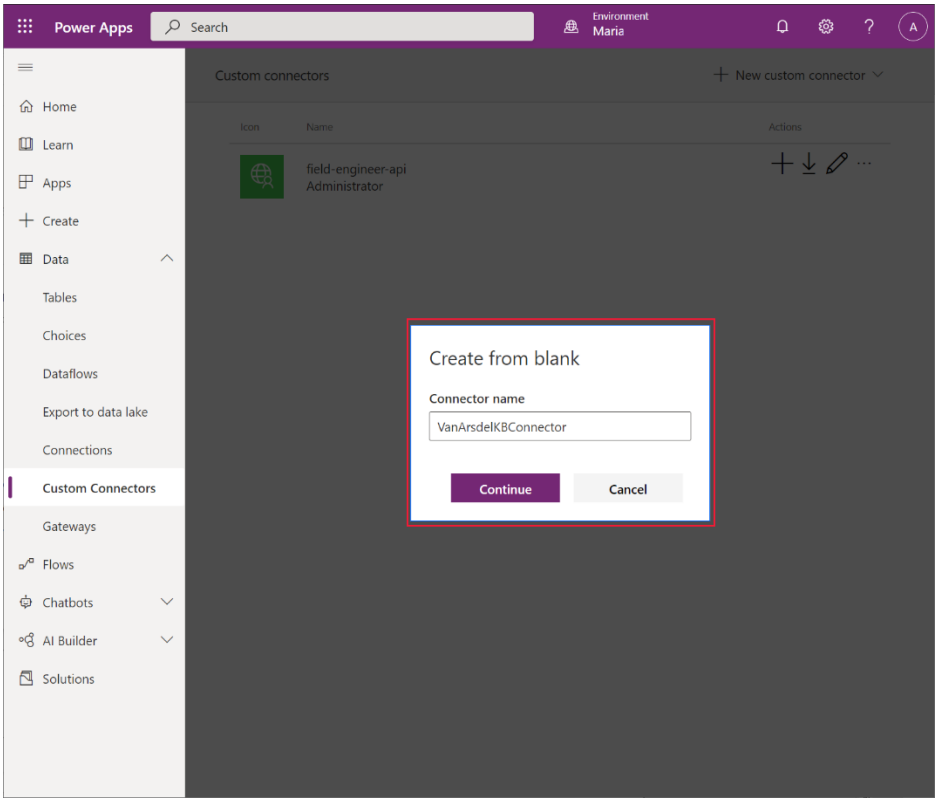
Kiana can now create a custom connector that Power Apps uses to send search requests to the search service. She does this using Power Apps Studio:

1. Sign in to Power Apps Studio at http://make.powerapps.com.

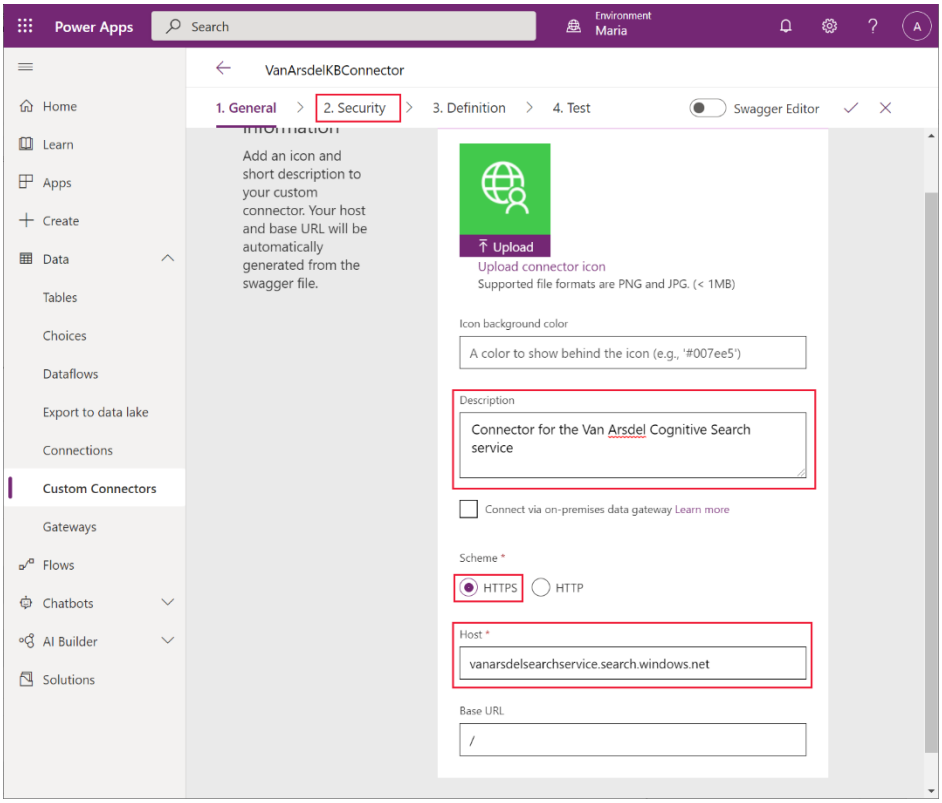
2. In the left pane, expand **Data**, and select **Custom Connectors**. In the right pane, select **+ New custom connector**, and then select **Create from blank**:



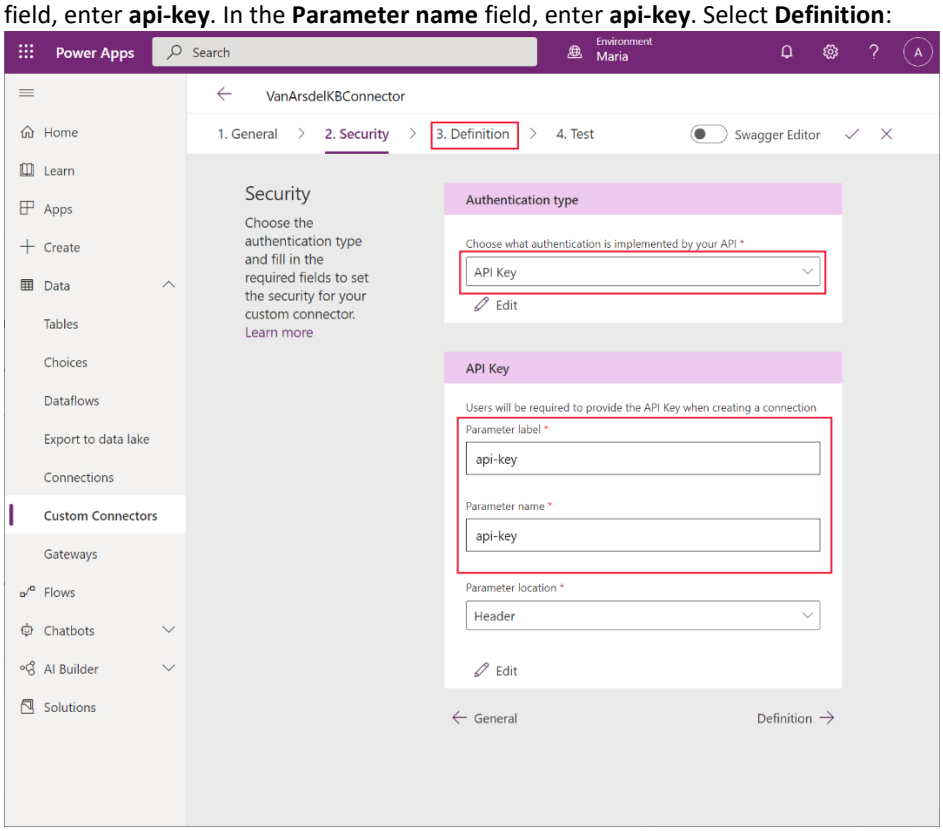
3. In the **Create from blank** dialog box, set the new connector name to **VanArsdelKBConnector**, and then select **Continue**:



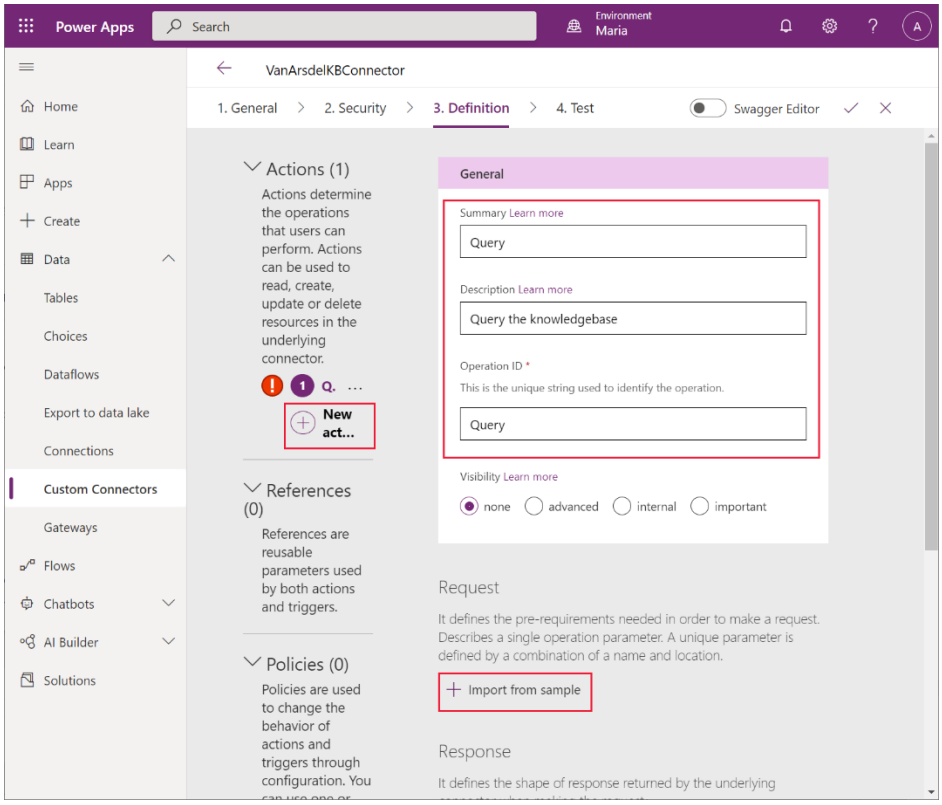
4. On the **General information** page, enter a description and set the **Scheme** to **HTTPS**. In the **Hosts** box, enter the URL for your search service (you noted this URL earlier) but without the **https://** prefix, and then select **Security**:



5. On the **Security** page, in the **Authentication** drop-down list box, select **API Key**. In the **Parameter label** field, enter **api-key**. In the **Parameter name** field, enter **api-key**. Select **Definition**:

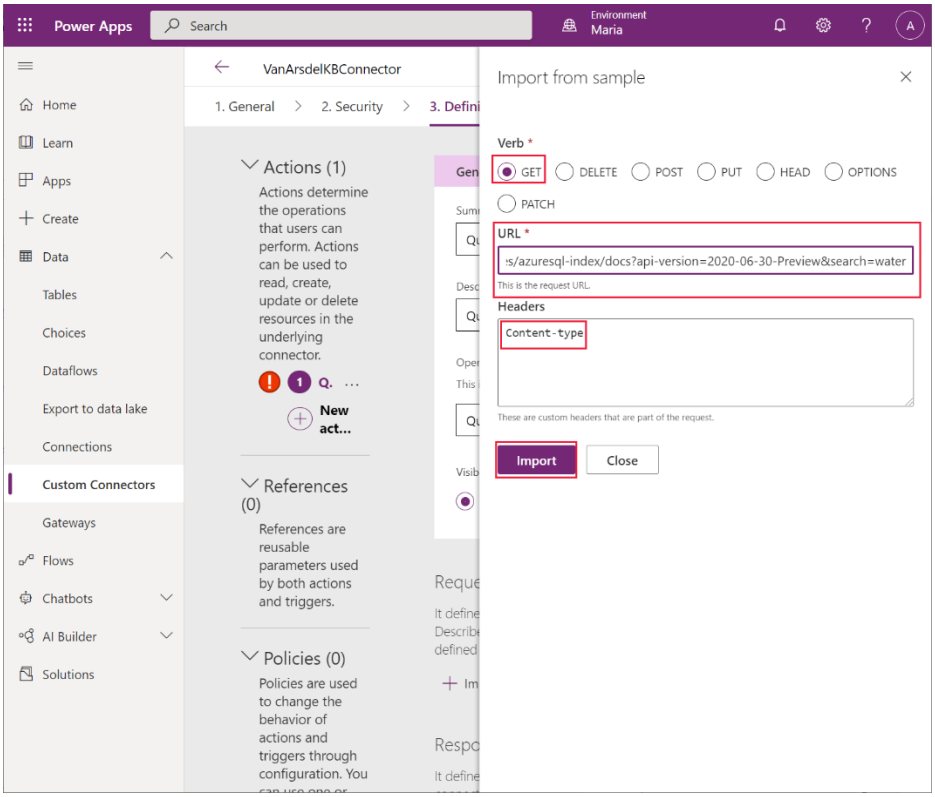


6. On the **Definition** page, select **New action**. In the **Summary** field, enter **Query**. In the **Description** field, enter **Query the knowledgebase**. In the **Operation ID** field, enter **Query**. Under **Request**, select **+ Import from sample**:

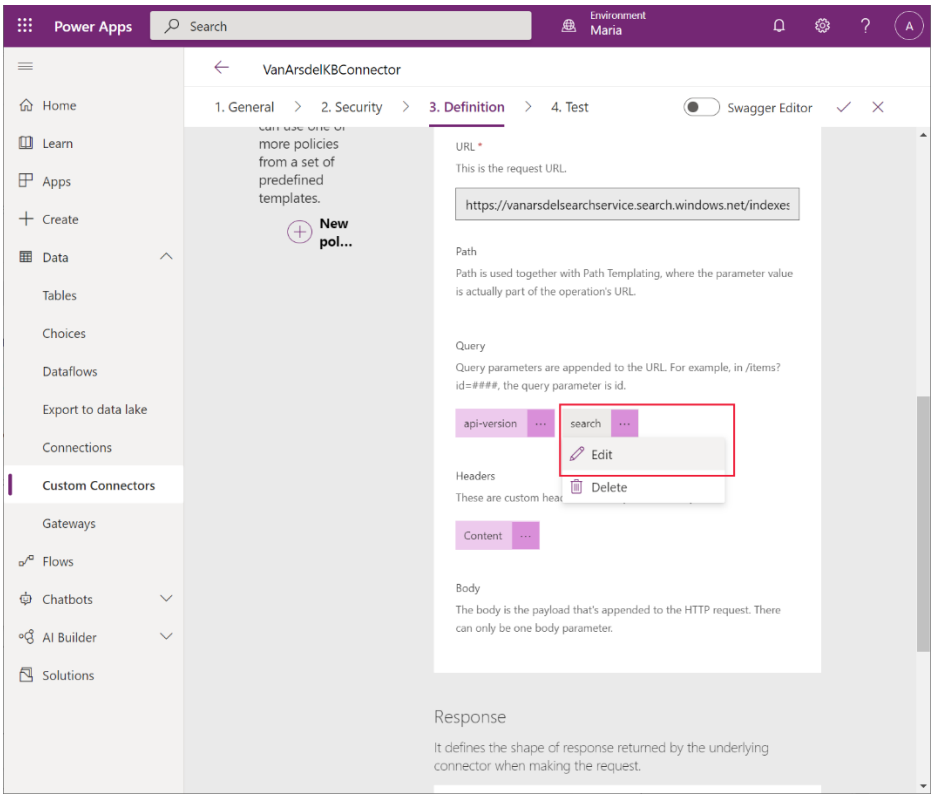


7. In the **Import from sample** dialog box, enter the following values, and then select **Import**:

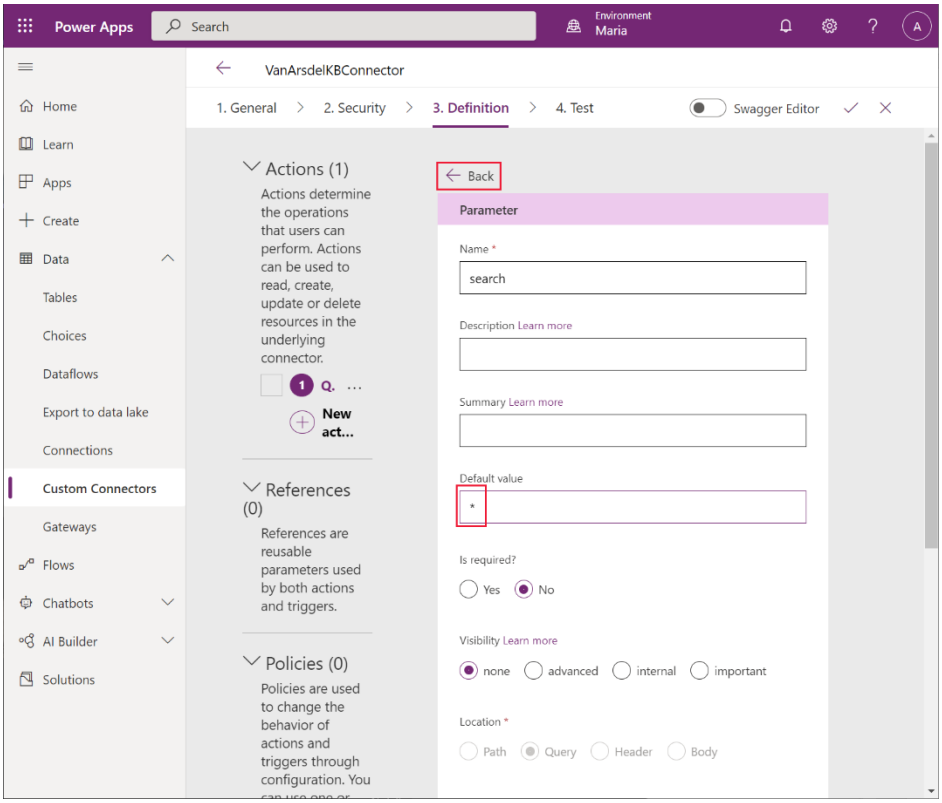
   - Verb: **GET**
   - URL: Provide the example request URL that you noted when you tested the search service in Search Explorer earlier
   - Headers: **Content-type**



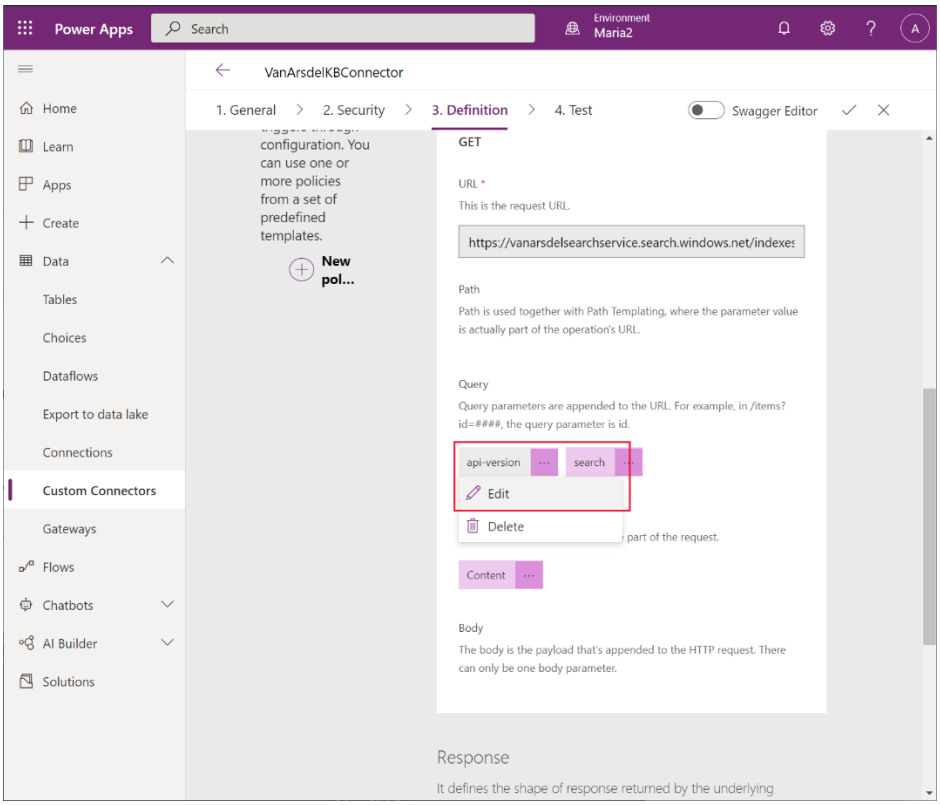
8. Back on the **Definition** page, scroll down to the **Query** section, select the ellipsis button next to **search**, and then select **Edit**:



9. On the edit screen, in the **Parameters** section, in the **Default value** field, enter an asterisk (*). Leave the other fields at their default values, and select **Back**:

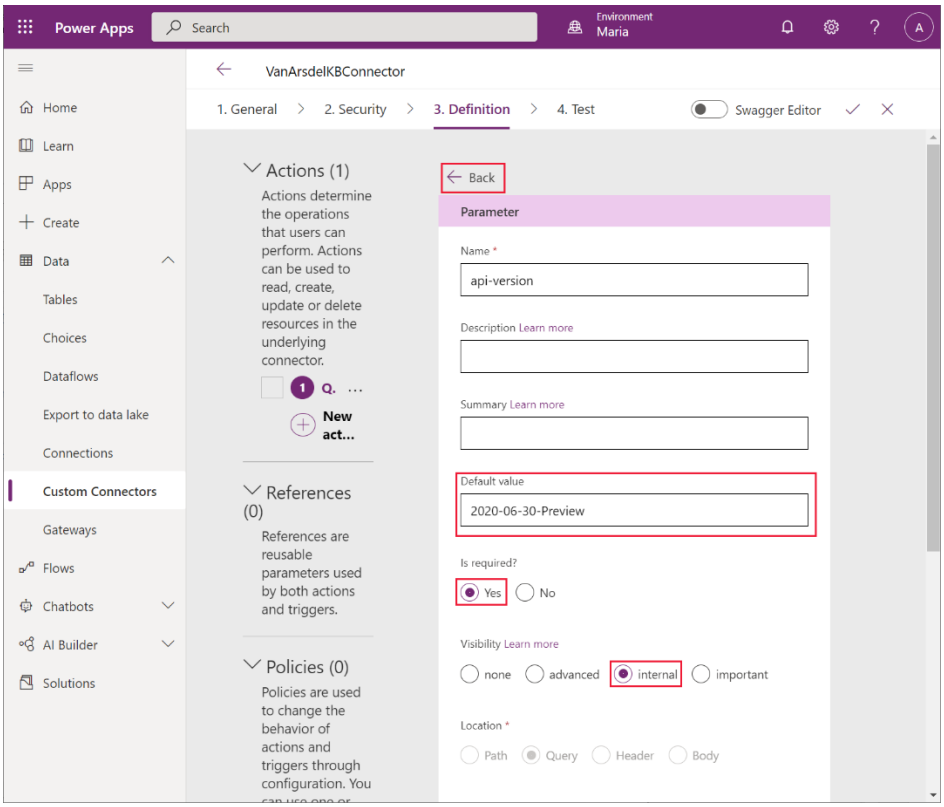


10. On the **Definition** page, in the **Query** section, select the ellipsis button next to **api-version**, and then select **Edit**:

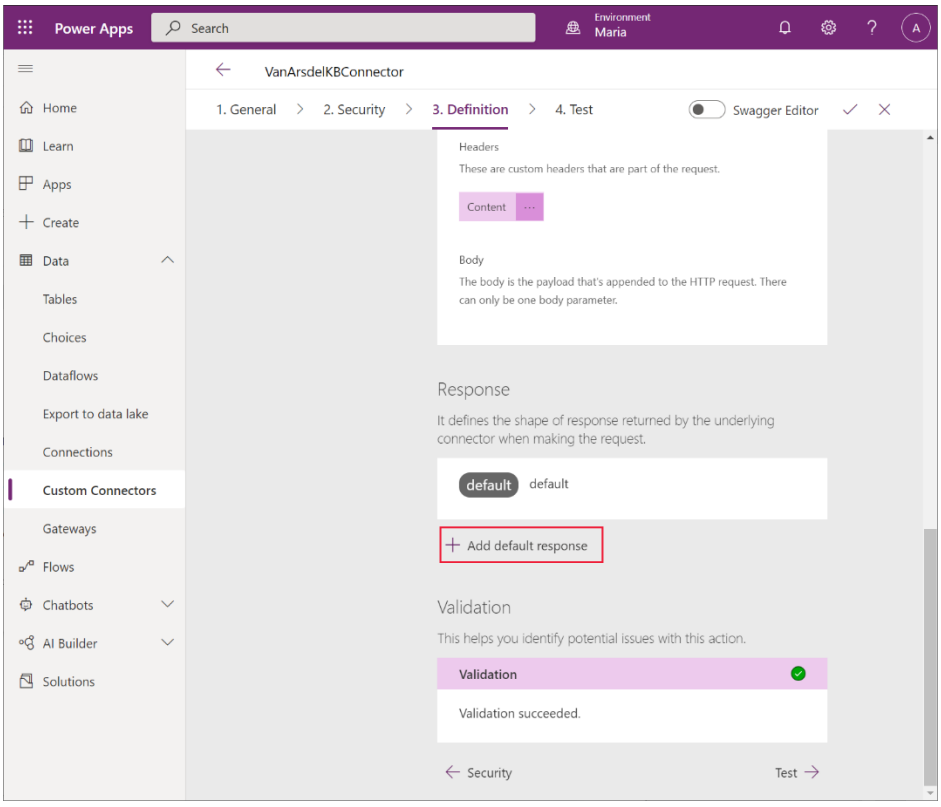


11. On the edit screen, in the **Parameters** section, in the **Default value** field, enter **2020-06-30-Preview** (this is the version associated with the current version of Azure Cognitive Search; you can see the version in the

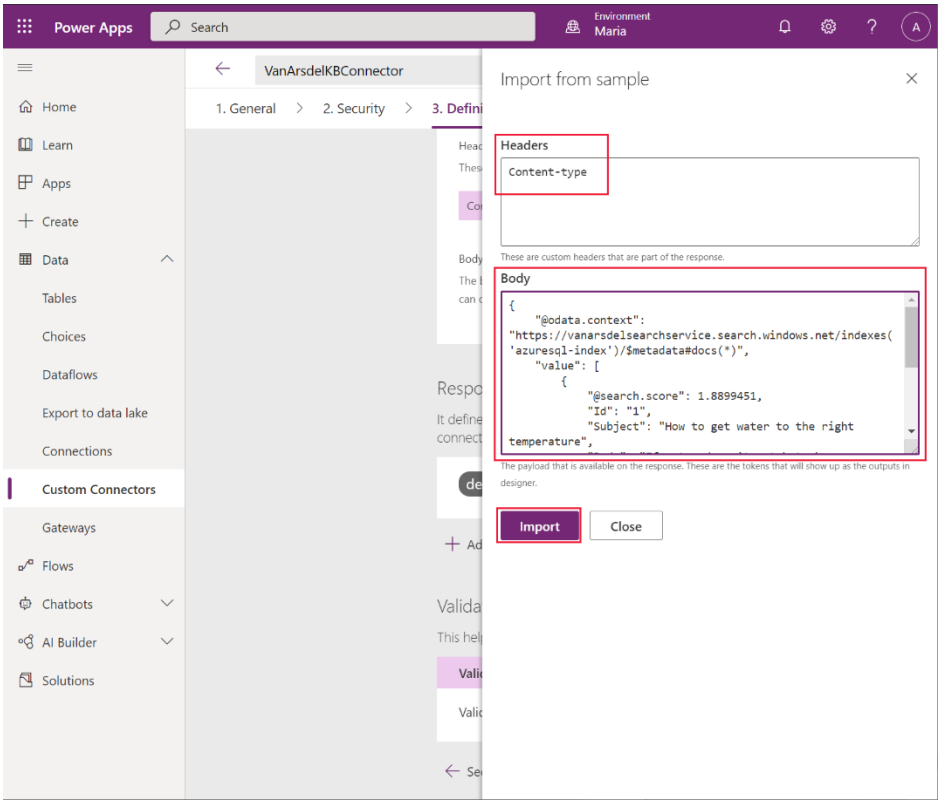
request URL that you noted earlier). Set **Is required** to **Yes**, and set **Visibility** to **internal**. Leave the other fields at their default values, and then select **Back**:



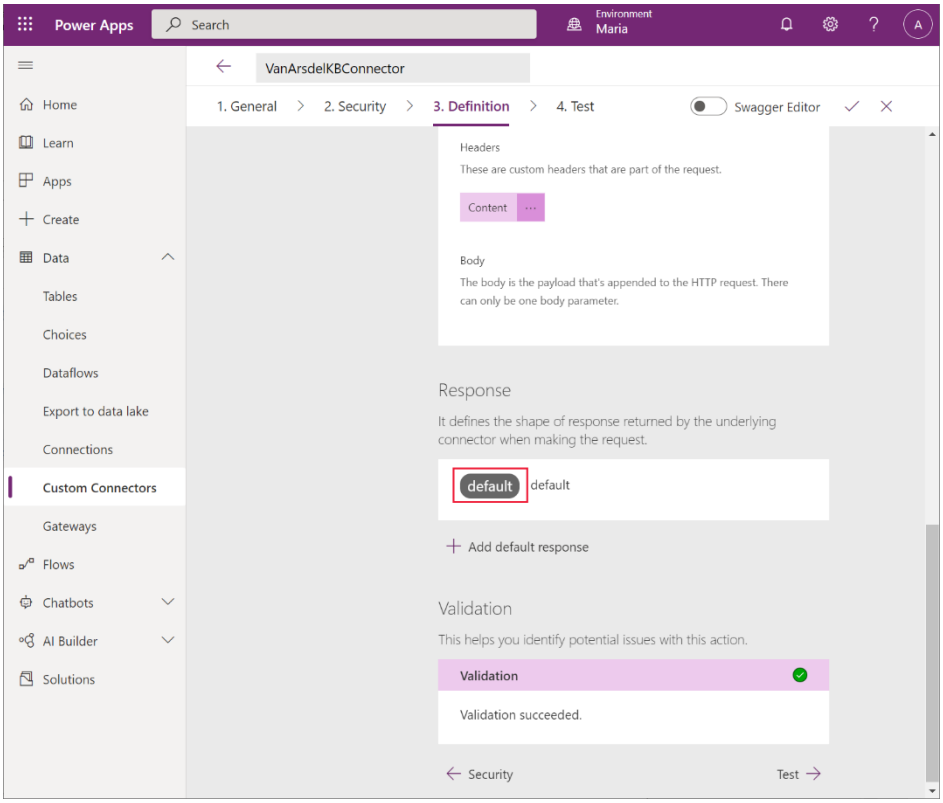
12. On the **Definition** page, scroll down to the **Response** section, and select **+ Add default response**:



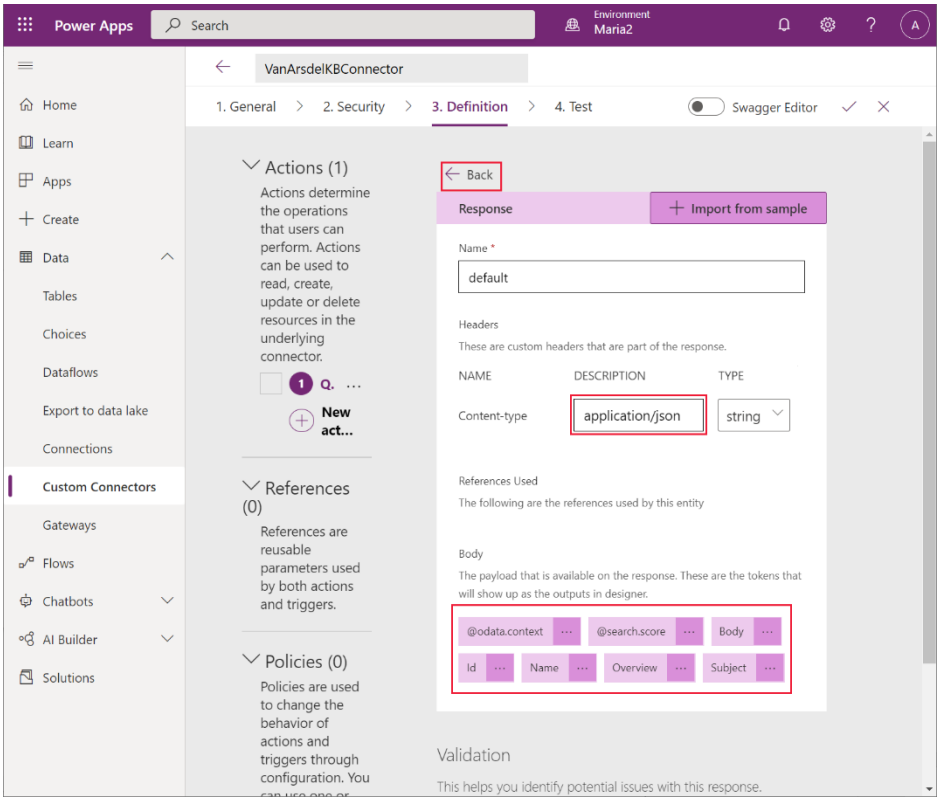
13. In the **Import from sample** dialog box, in the **Headers** field, enter the text **Content-type**. In the **Body** field, enter the example results that you recorded when testing the search service, and then select **Import**:



14. On the **Definition** page, select the **default** response:

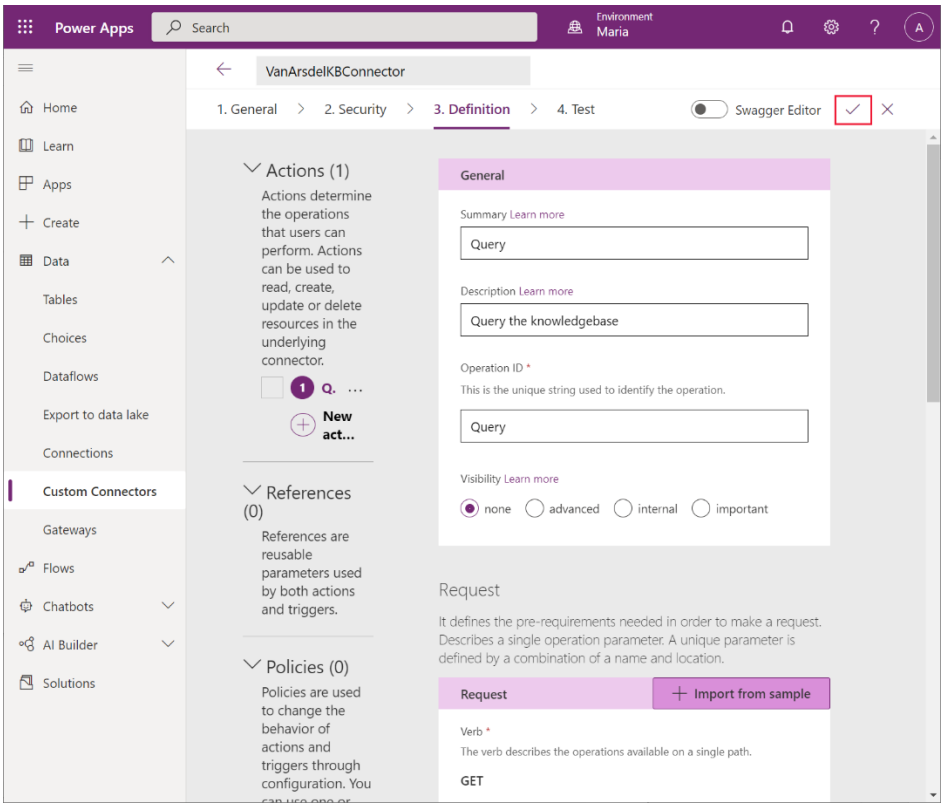


15. In the **Description** field of the **Content-type** response, enter **application/json**, and then select **Back**:



Note: The **Body** section on this page should display the fields of the response, such as **Body**, **Id**, **Name**, **Overview**, and **Subject** if it has been parsed successfully.

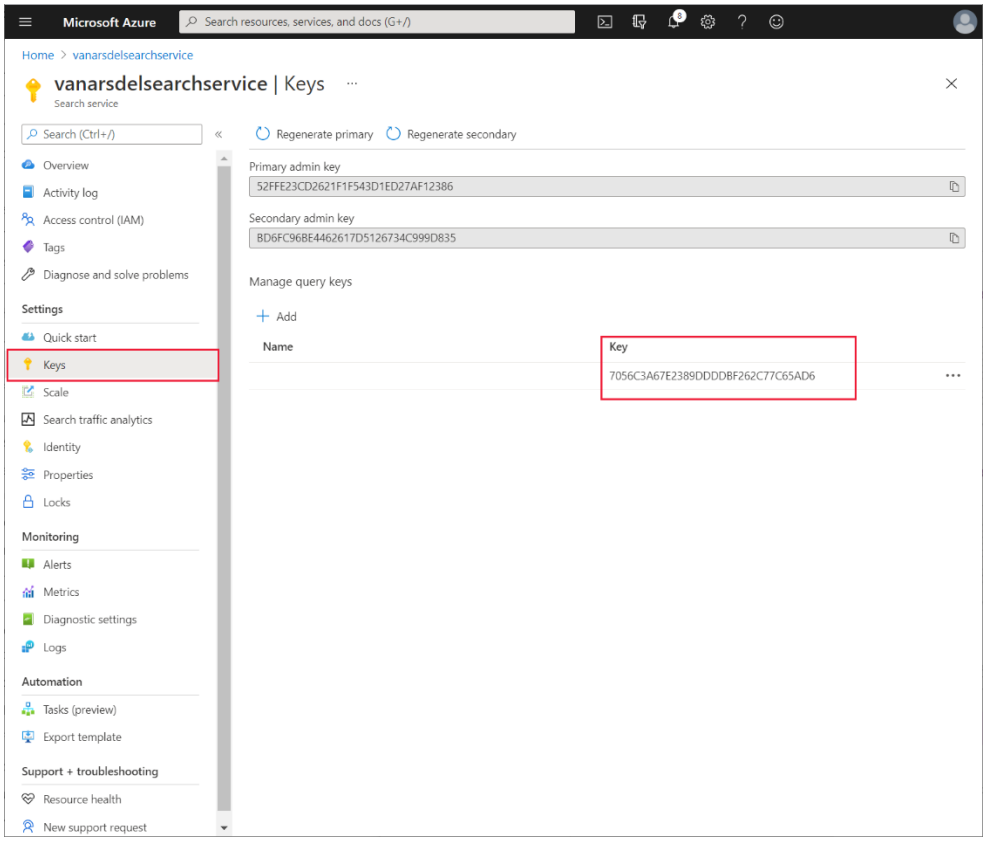
16. Select **Create connector**:



The connector should be created without reporting any errors or warnings.

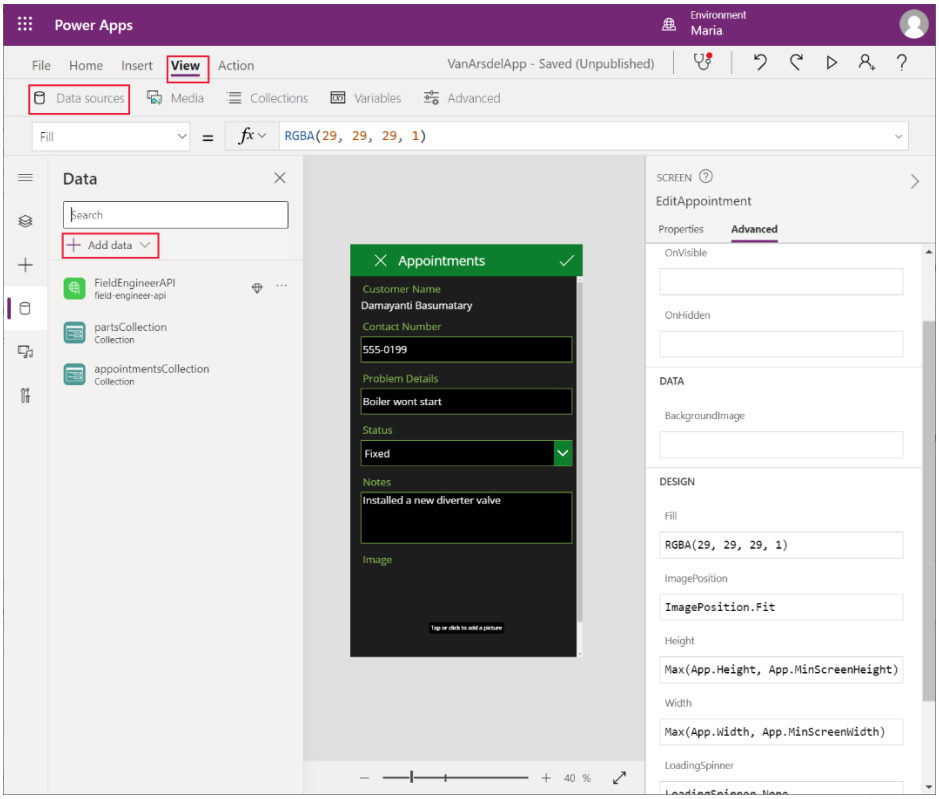
UPDATING THE APP TO USE AZURE COGNITIVE SEARCH: FIELD KNOWLEDGEBASE

Maria can now use the custom connector in the app. But first, she requires a key that grants her the privileges required to connect to the Azure Cognitive Search service. Preeti obtains the key from the **Keys** page for the service in the Azure portal, and gives it to Maria:

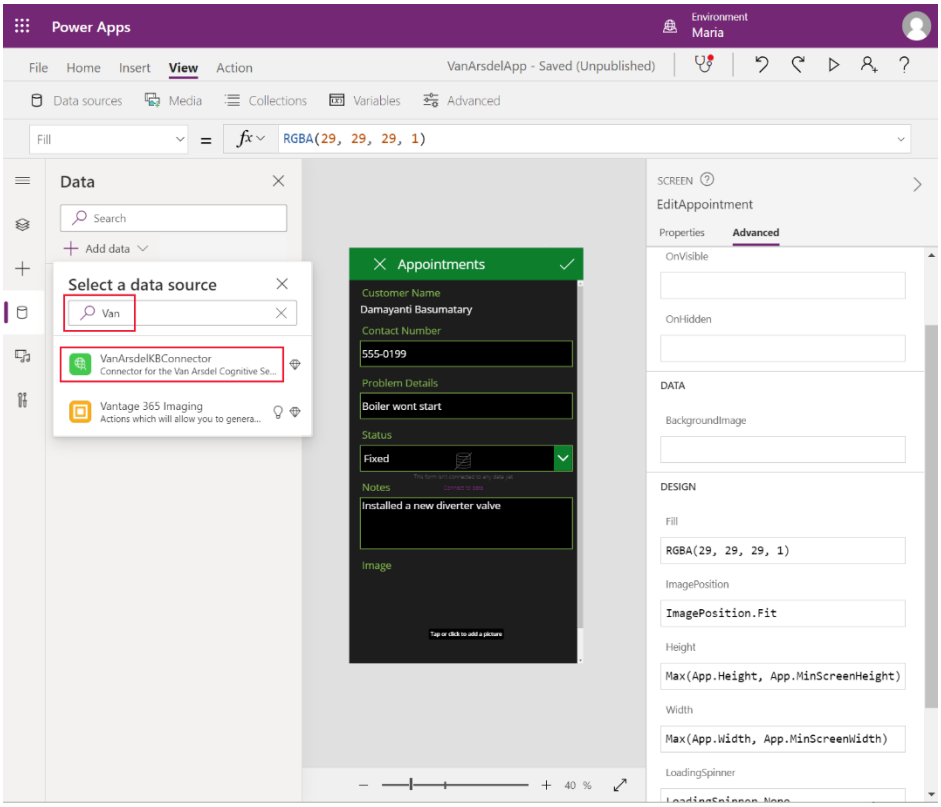


Maria edits the app in Power Apps Studio and performs the following tasks:

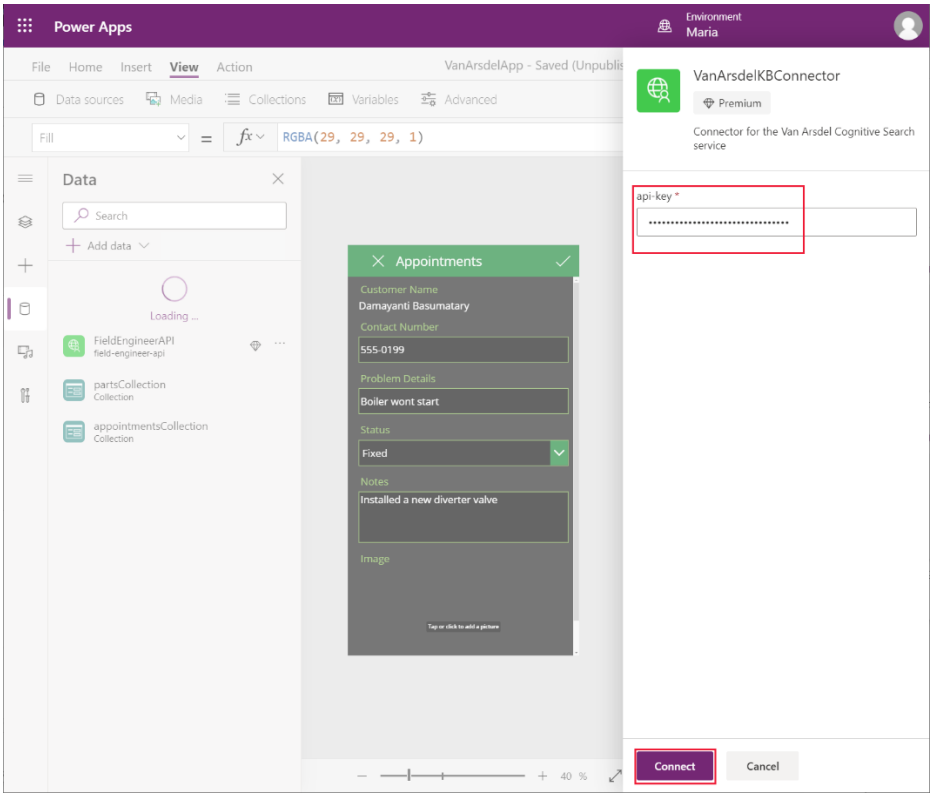
1. Open the **VanArsdelApp** app for editing.
2. On the **View** menu, select **Data sources**, and then select **Add data**:



3. In the **Search** box, under **Select a data source**, enter **Van**. The **VanArdelKBConnector** connector should be listed:



4. Select the **VanArdelKBConnector** connector. In the **VanArdelKBConnector** pane, enter the key that Preeti provided for the search service, and then select **Connect**:



5. On the **File** menu, save and close the app, and then open it again. You may be prompted to authorize use of the custom connector when the app reopens.

> **Note**: This step is necessary to enable the custom connector.

6. In the **Tree view** pane, expand the **Knowledgebase** screen, and select the **TextSearchBox2** control. Enter the following formula for the **OnChange** action:

```
If(!IsBlank(TextSearchBox2.Text), ClearCollect(azResult, VanArsdelKBConnector.Query({search: TextSearchBox2.Text}).value))
```

   This formula calls the **Query** operation of the custom connector searching for items that match the term the user types into the search box. The results are stored in a collection named **azResult**.

7. In the **Tree view** pane, under the **Knowledgebase** screen, select the **BrowseGallery2** control. Set the **Items** property to **azResult**.

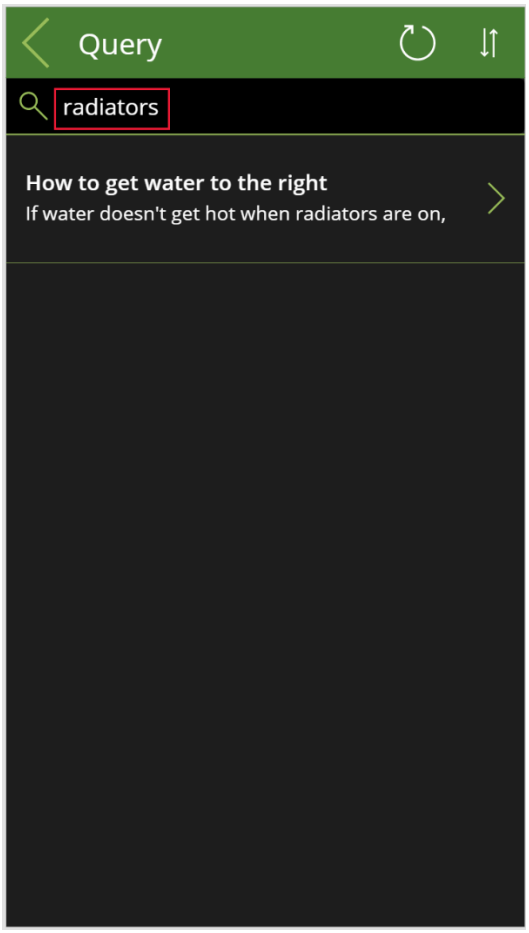
8. Expand the **BrowseGallery2** control and remove the **Image4** control.

9. Select the **Title2** control. Set the following properties to the values specified:

   - Text: **ThisItem.Subject**
   - X: **24**
   - Width: **Parent.TemplateWidth – 104**

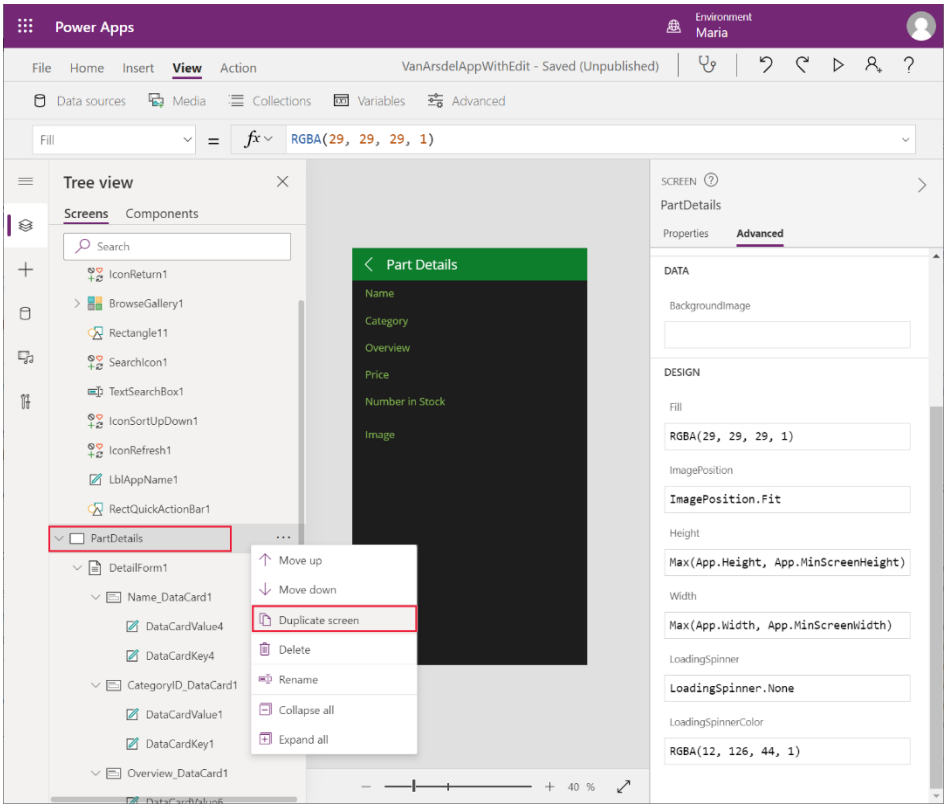
10. Select the **Subtitle2** control. Set the **Text** property to **ThisItem.Body**.

11. Press **F5** to preview the app. On the **Knowledgebase** screen, enter a search term, and press **Enter**. Matching articles from the knowledge base should be displayed:



Note: The details screen hasn't been created yet, so clicking the **>** icon next to an article doesn't work.

12. Close the preview window and return to Power Apps Studio.
13. In the **Tree view** pane, right-click the **PartDetails** screen, and select **Duplicate screen**. This action will add another screen to the app, named **PartDetails_1**:



14. In the **Tree view** pane, rename the **PartDetails_1** screen as **KnowledgebaseDetails**.

    - Select the **LblAppName*X*** control on the screen; set the **Text** property to **"Article Details"** (including the quotes)

15. In the **Tree view** pane, select the **DetailFormX** control on the screen. Set the following properties:

    - DataSource: **azResult**
    - Item: **BrowseGallery2.Selected**

    > **Note**: **BrowseGallery2** is the browse gallery on the **Knowledgebase** screen. In your application, this gallery may have a different name.

16. In the **Tree view** pane, expand the **DetailForm*X*** form, then change the names of the following data card controls:

    - Name_DataCard1_1: **Name_DataCard**
    - CategoryID_DataCard1_1: **Subject_DataCard**
    - Overview_DataCard1_1: **Overview_DataCard**
    - Price_DataCard1_1: **Body_DataCard**

17. Delete the **NumberInStock_DataCard1_1**, and **Image_DataCard1_1** controls.

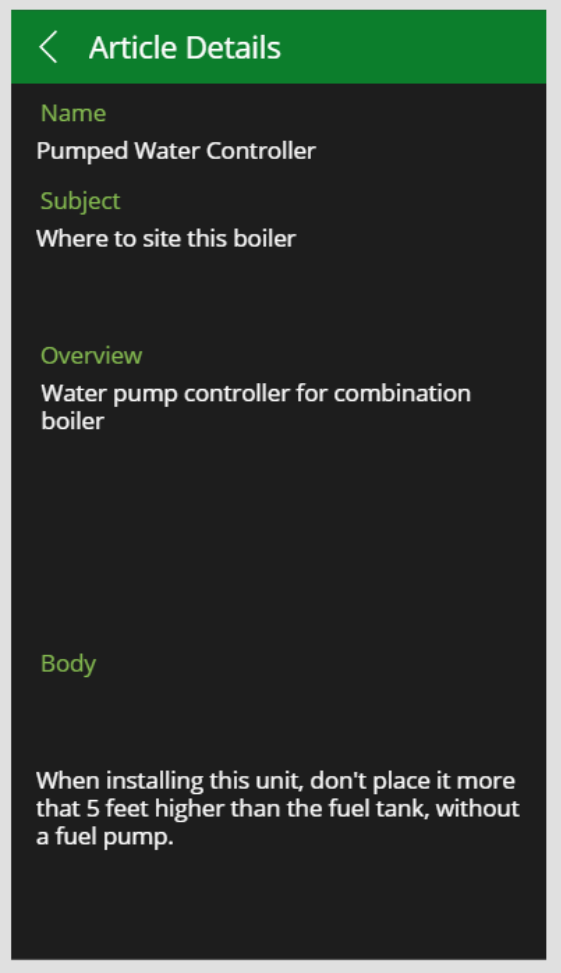
18. Select the **Name_DataCard** control. Set the **Default** property to **ThisItem.Name**.

19. Select the **Subject_DataCard** control. Set the following properties:

    - DataField: **"Subject"**
    - DisplayName: **"Subject"**

- Default: **ThisItem.Subject**

20. Select the **Overview_DataCard** control. Set the **Default** property to **ThisItem.Overview**.
21. Select the **Body_DataCard** control. Set the following properties:
    - DataField: **"Body"**
    - DisplayName: **"Body"**
    - Default: **ThisItem.Body**
22. Select the **DataCardValue*X*** control in the **Body_DataCard** control. Set the **Text** property to **Parent.Default**.
23. Resize each of the data card controls to spread them out down the screen:



24. Select the back arrow icon in the screen header. Change the **OnSelect** action property to **Navigate(Knowledgebase, ScreenTransition.None)**.
25. In the **Tree view** pane, select the **Knowledgebase** screen, and then select the **BrowseGalleryX** control. Change the **OnSelect** action property to **Navigate(KnowledgebaseDetails, ScreenTransition.None)**. This action displays the details screen for the knowledge base article when the user selects the **>** icon for an entry in the browse screen.
26. Save the app.

27. Press **F5** to preview the app. On the **Knowledgebase** screen, enter a search term and press **Enter**. Select an article and verify that its details are displayed. Verify that the **Back** icon returns the user to the browse screen.

28. Close the preview window and return to Power Apps Studio.

Maria, Kiana, and Preeti have successfully incorporated the Web API and Azure Cognitive Search into the app.

CHAPTER 7: ADDING FUNCTIONALITY TO THE APP

Kiana and Maria are excited to show the inventory management app to Caleb, the field technician. He likes it, but suggests adding some extra user interface functionality to make it easier to use. Specifically, Caleb would like to be able to:

- Add a photograph of the work done on a boiler or air conditioning unit, and add it to the appointment details on the **Edit Appointment** screen. This image could prove useful as documentary evidence of repairs performed. The **Edit Appointment** screen currently enables the user to add an image to the appointment, but the image isn't saved as this feature hasn't been fully implemented yet. The reason for this omission is that Kiana and Preeti need to determine the best place to store image data. Caleb would like this functionality added as soon as possible.
- View a complete appointment history for a customer, to track repairs requested for that customer and monitor any ongoing issues that may require repeated callouts.
- Order parts from the **Part Details** screen.

Additionally, the image control on the **Part Details** screen displays the images stored at a specified URL. Currently the URLs in the data are simply placeholders. Like the photographs for the appointment screen, Kiana and Preeti need to determine the best place to store images so they're available to the app.

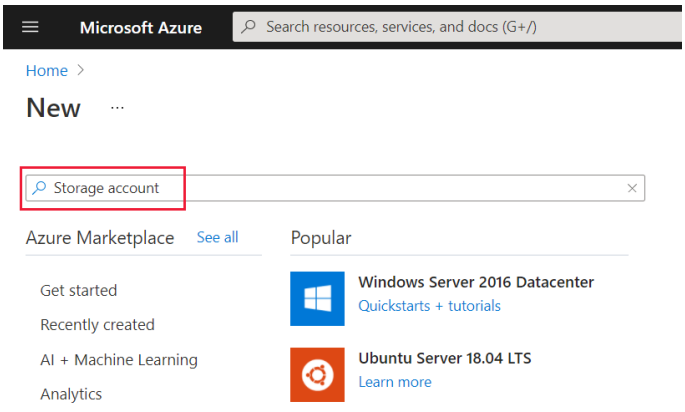
ADDING A PHOTOGRAPH TO AN APPOINTMENT

Photographs need to be stored somewhere accessible by the app. For performance and security reasons, Preeti doesn't want photographs to be saved in OneDrive or in Azure SQL Database. Instead, she and Kiana decide to use Azure Blob Storage. Blob Storage is optimized for holding large binary objects, and is robust, with built-in security. Power Apps has a connector that allows access to Blob Storage. Maria suggests adding a new picture-taking screen, improving the user experience for Caleb.

> **Note**: For detailed information, visit the **Azure Blob Storage** page at https://azure.microsoft.com/services/storage/blobs/.

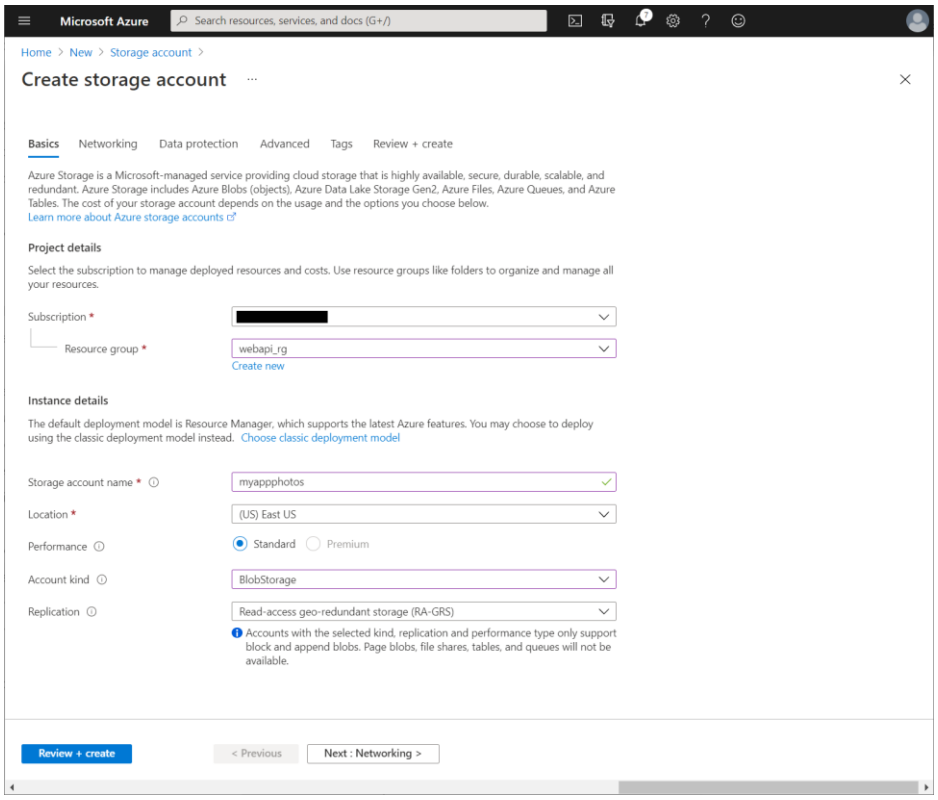
Preeti creates the Blob Storage account from the Azure portal:

1. In the Azure portal, on the **Home** page, select **+ Create a resource**. In the **Search the Marketplace** box, enter **Storage account** and press **Enter**.

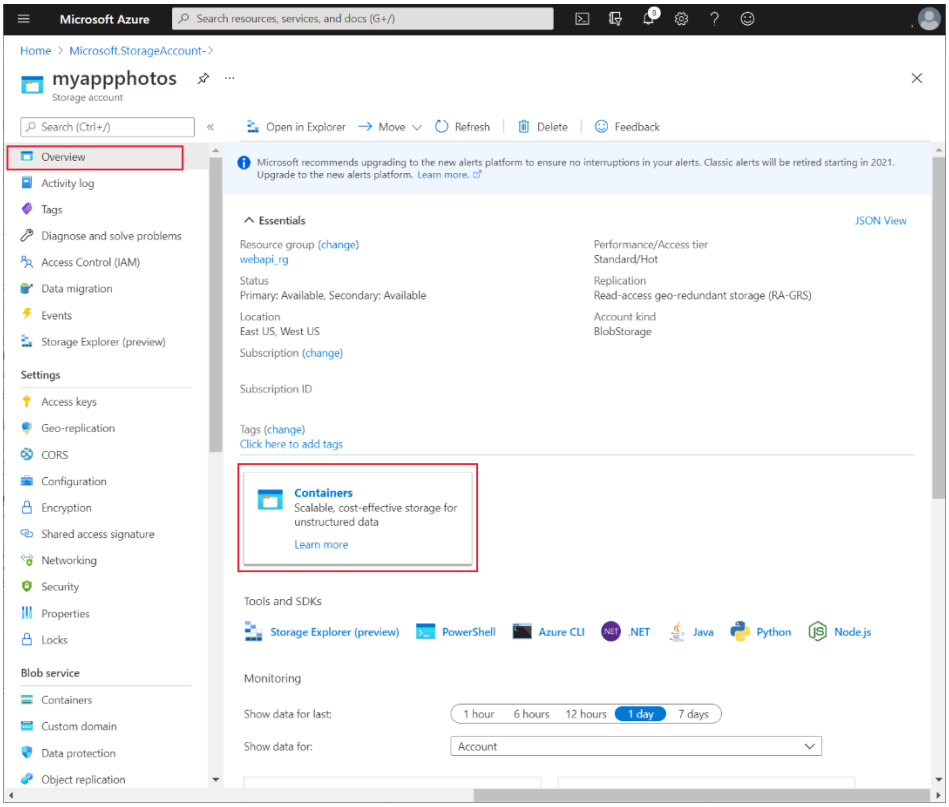


2. On the **Storage account** page, select **Create**.
3. On the **Create storage account** page, enter the following details, and then select **Review + create**:
   - Subscription: Select your subscription
   - Resource group: **webapi_rg**

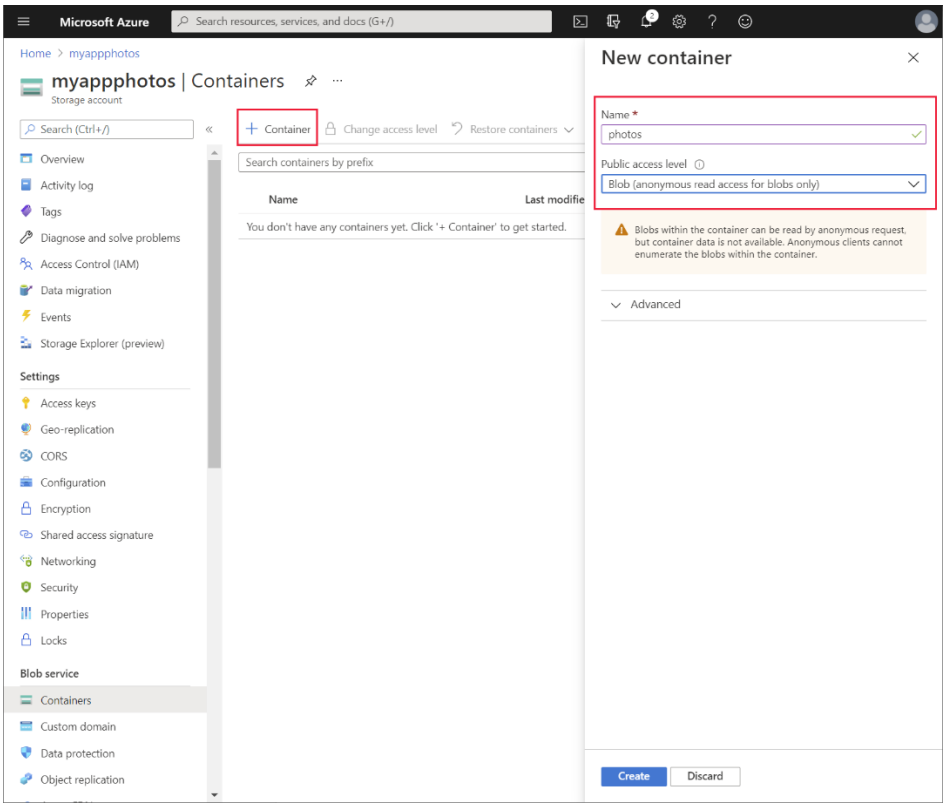
- Storage account name: Provide a globally unique name and make a note of it for later
- Location: Select your nearest location
- Performance: **Standard**
- Account kind: **BlobStorage**
- Replication: **RA-GRS**



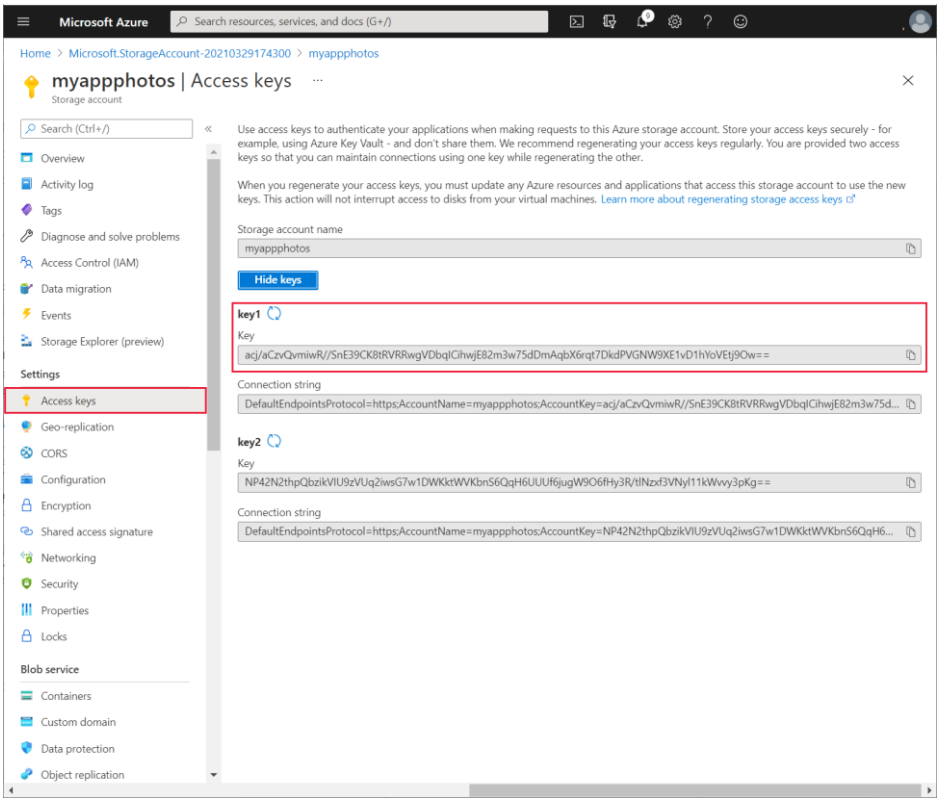
4. On the validation page, select **Create** and wait for the storage account to be provisioned.
5. Go to the page for the new storage account.
6. On the **Overview** page, select **Containers**:



7. On the **Containers** page, select **+ Container**. Create a new container named **photos**, and then select **Create**. Change the **Public access level** to **Blob**:



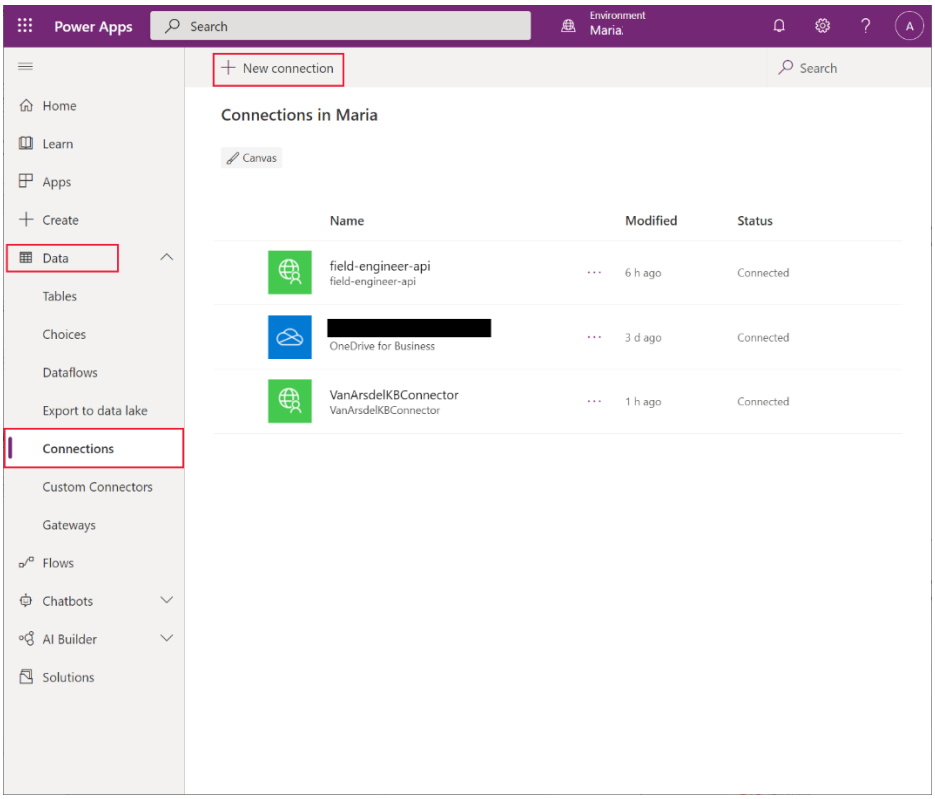
8. Back on the **Overview** page for the storage account, under settings, select **Access keys**. On the **Access keys** page, select **Show keys.** Make a note of the value of the key for **key1**:



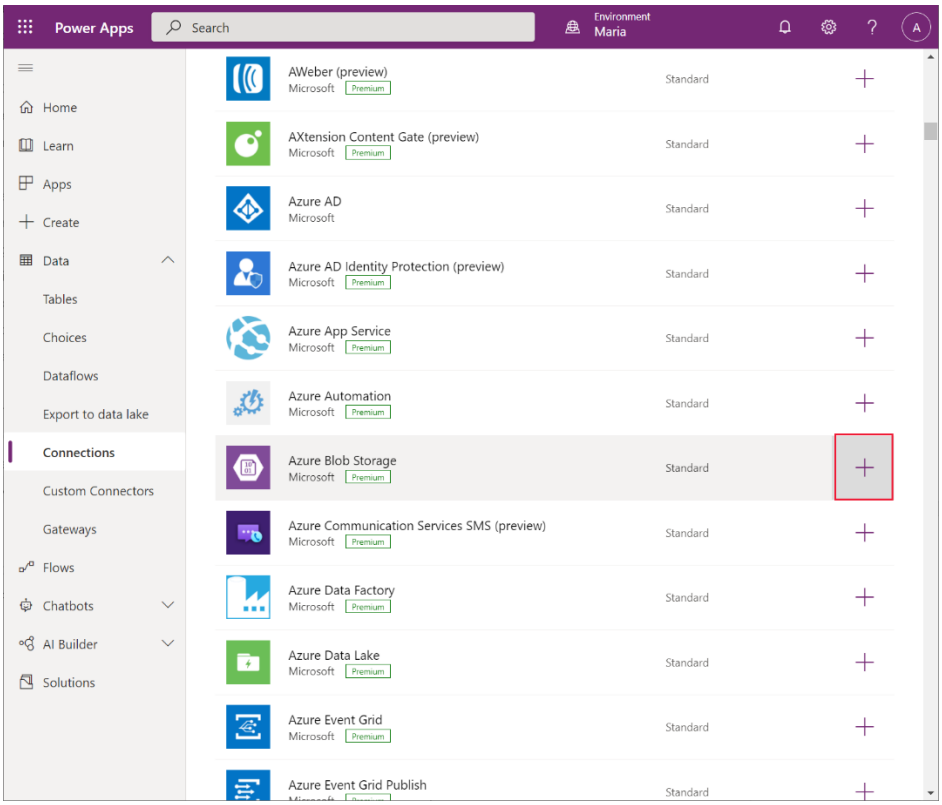
Preeti gives the storage account name and key to Kiana, who uses this information to create a custom connector for the app:

1. Sign in to Power Apps Studio at [http://make.powerapps.com.](http://make.powerapps.com)

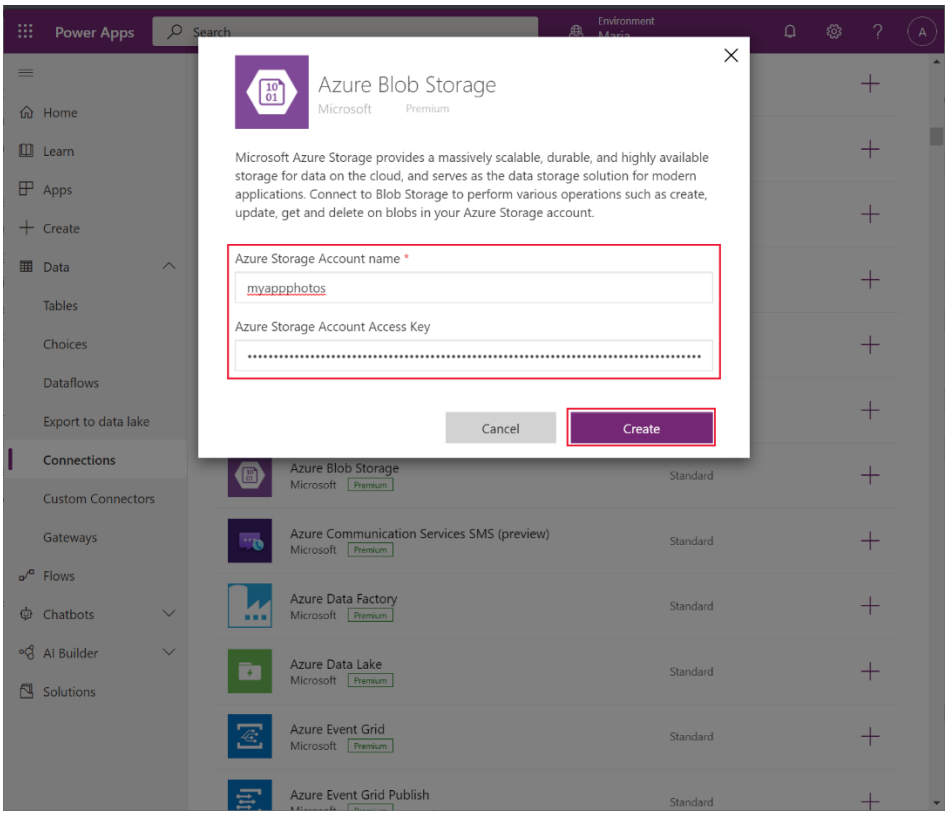
2. In the left pane, expand **Data**, and select **Connections**. The existing connections used by the app should be listed. Select **+ New connection**:



3. On the **New connection** page, scroll down, select **Connections**, select **Azure Blob** Storage, and then select **Create**:



4. In the **Azure Blob Storage** dialog box, enter the storage account name and access key that Preeti provided, and then select **Create**:

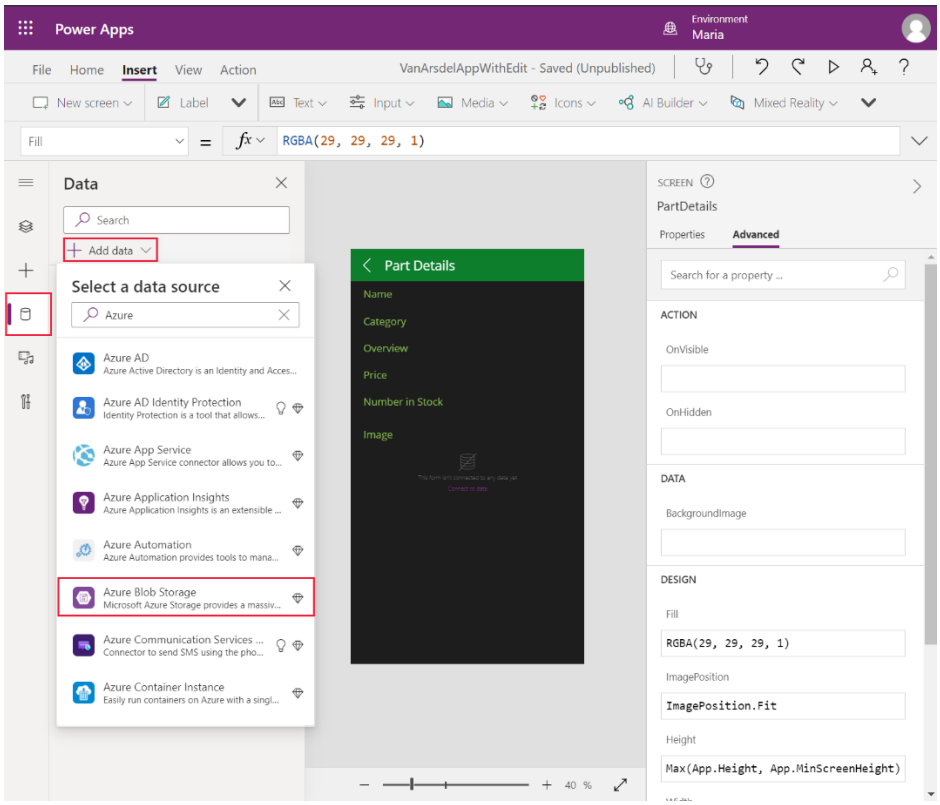


5. Wait while the new connection is created. It should appear on the list of connections.

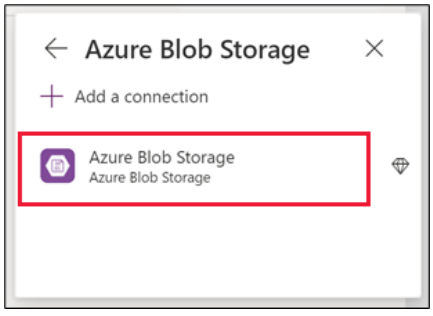
Maria can use this connection to Azure Blob Storage in the app to save and retrieve photographic images. Her first task is to add the connection to the app:

1. Open the **VanArsdelApp** app for editing in Power Apps Studio.

2. In the **Data** pane, select **Add data**, search for the **Azure Blob Storage** connector, and then select this connector:

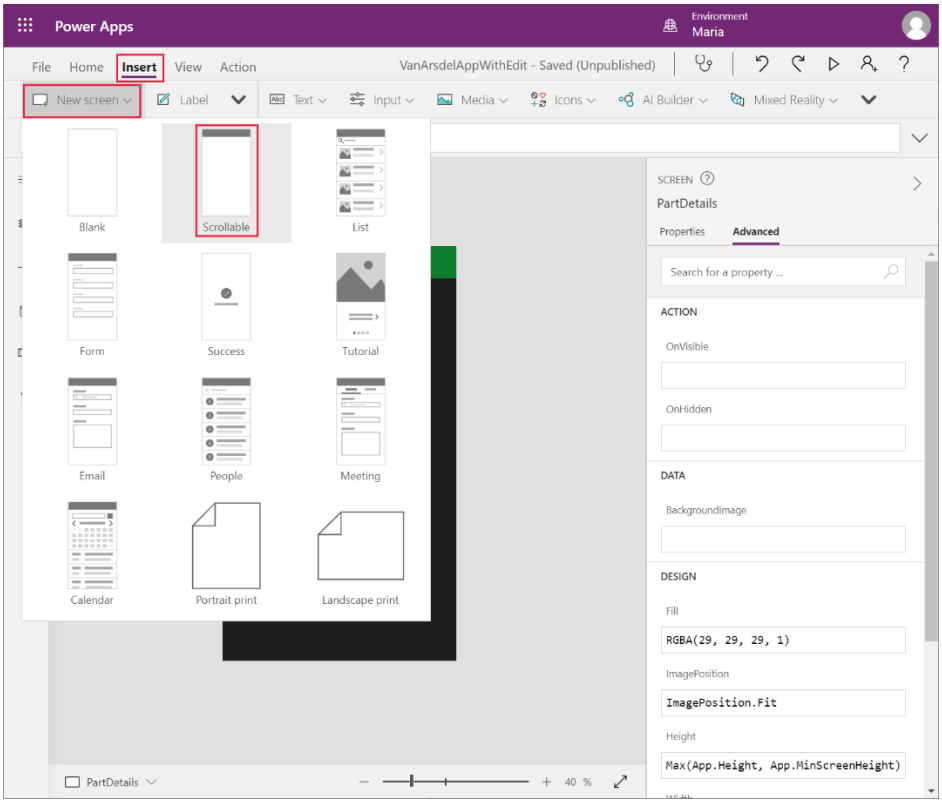


3. In the **Azure Blob Storage** dialog box, select the **Azure Blob Storage** connector to add it to your app:

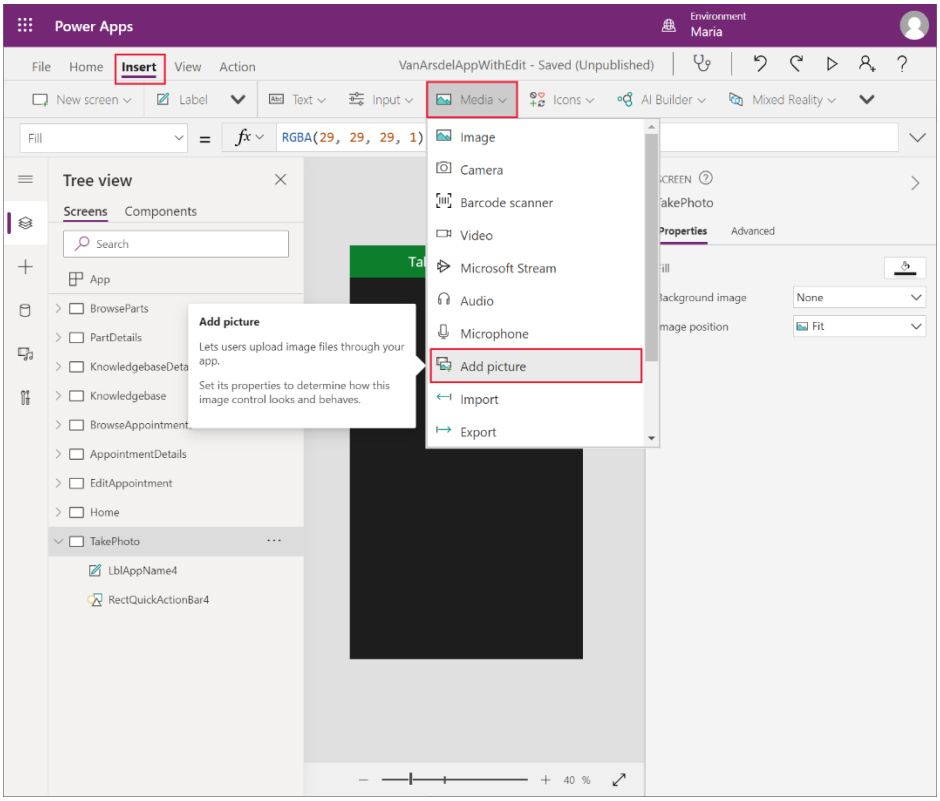


Maria's next task is to add a screen that enables a technician or engineer to save a photograph. Maria decides to add a new screen with a Picture control. When the app is run on a mobile device, this control can integrate with the camera to enable the technician to take a photograph. On other devices, this control prompts the user to upload an image file instead. She adds a link to this new screen from the **EditAppointment** screen:

1. On the **Insert** menu, select **New screen**, and then select the **Scrollable** template:



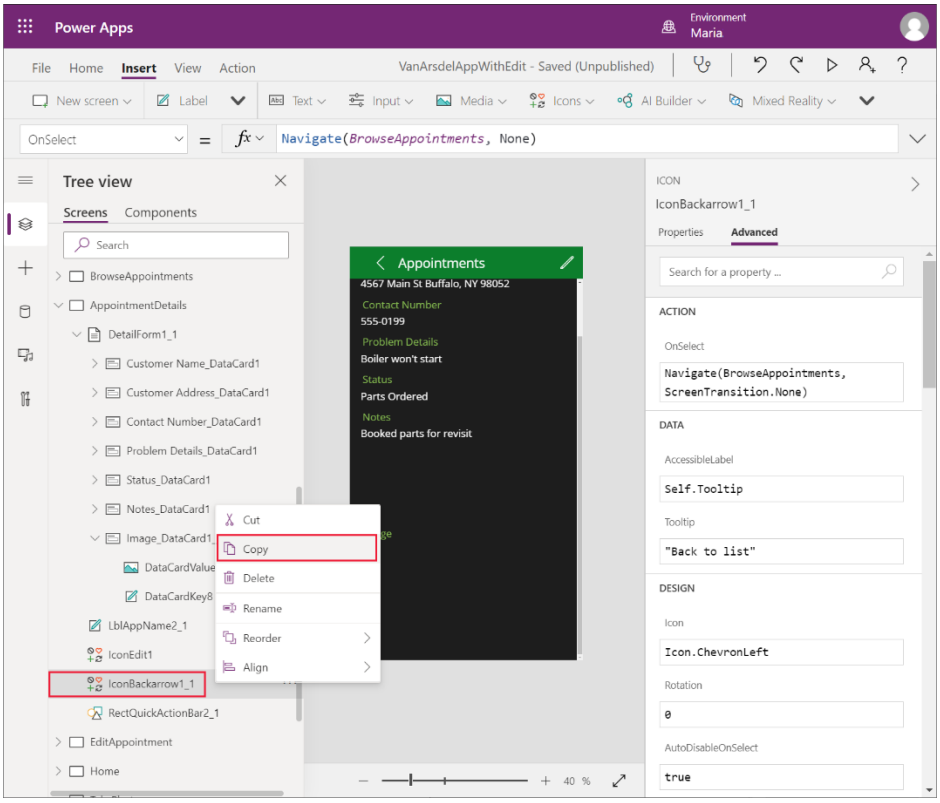
2. In the **Tree view** pane, select the new screen and rename it as **TakePhoto**.
3. Change the **Text** property of the **LblAppName*X*** control on this screen to **Take a photograph**.
4. Delete the **Canvas*X*** control from the screen.
5. In the **Insert** menu, from the **Media** drop-down list, select **Add picture** to create a new picture control:



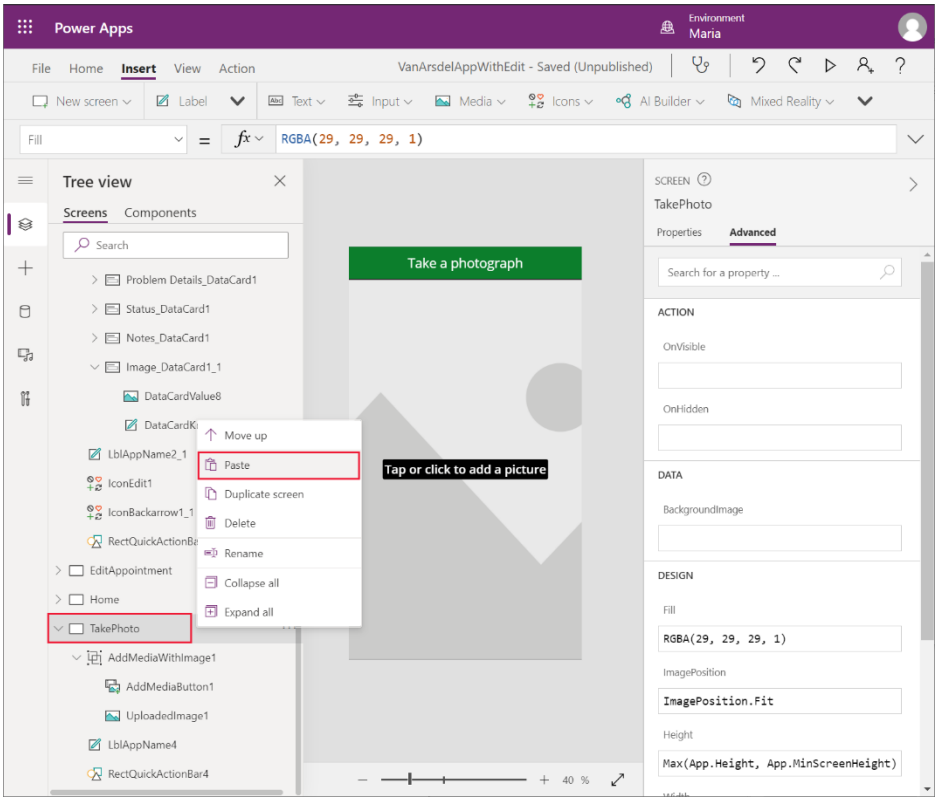
> **Note**: The picture control is actually a composite custom component that enables the user to add a picture to the screen and display the results.

6. Resize and reposition the picture control to occupy the body of the screen.

7. In the **Tree view** pane, select the **IconBackarrow*X*** control on the **AppointmentDetails** screen, and select **Copy**:



8. In the **Tree view** menu, right-click the **TakePhoto** screen, and then select **Paste**. The **IconBackArrow*X*** control will be added to the screen:

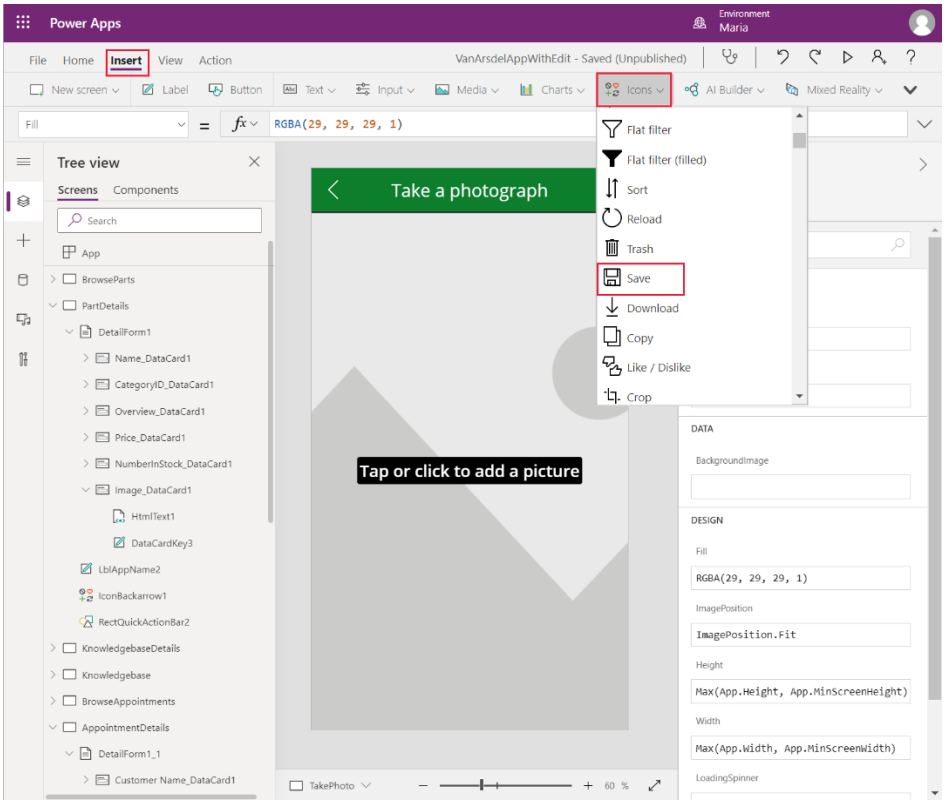


9. Move the **IconBackArrow*X*** control to the top left of the header bar.

10. In the **Tree view** pane, select the **IconBackArrow*X*** control on the **TakePhoto** screen. In the right pane, on the **Advanced** tab, modify the **OnSelect** action property to **Navigate(EditAppointment, ScreenTransition.None)**.

11. Add a new **Save** icon control to the top right of the header bar. Set the **Visible** property of this control to **If(IsBlank(AddMediaButton1.Media), false, true)**.

    This setting makes the **Save** icon invisible if the user hasn't selected an image.



12. Change the formula in the **OnSelect** action property of the **Save** icon control to:

```
Set(ImageID, GUID() & ".jpg");

AzureBlobStorage.CreateFile("photos", ImageID, AddMediaButton1.Media);

Patch(appointmentsCollection,
LookUp(appointmentsCollection,id=BrowseAppointmentsGallery.Selected.id),
{imageUrl:"https://myappphotos.blob.core.windows.net/photos/" &
ImageID});

Navigate(EditAppointment,ScreenTransition.Cover);
```

Replace **<storage account name>** with the name of the Azure storage account that Preeti created.

This code uploads the image to the **photos** container in Azure Blob Storage. Each image is given a unique filename. The **Patch** function updates the **imageUrl** property in the appointments record with the URL of the image in Blob Storage.

13. In the **Tree view** pane, expand the **AddMediaWithImageX** control. Modify the **Image** property of the **UploadedImage*X*** control, and set it to **AppointmentImage**.

    AppointmentImage is a variable that will be populated with an image either uploaded by the user, or as the result of taking a photograph. You'll initialize this variable in the **EditAppointment** screen later.

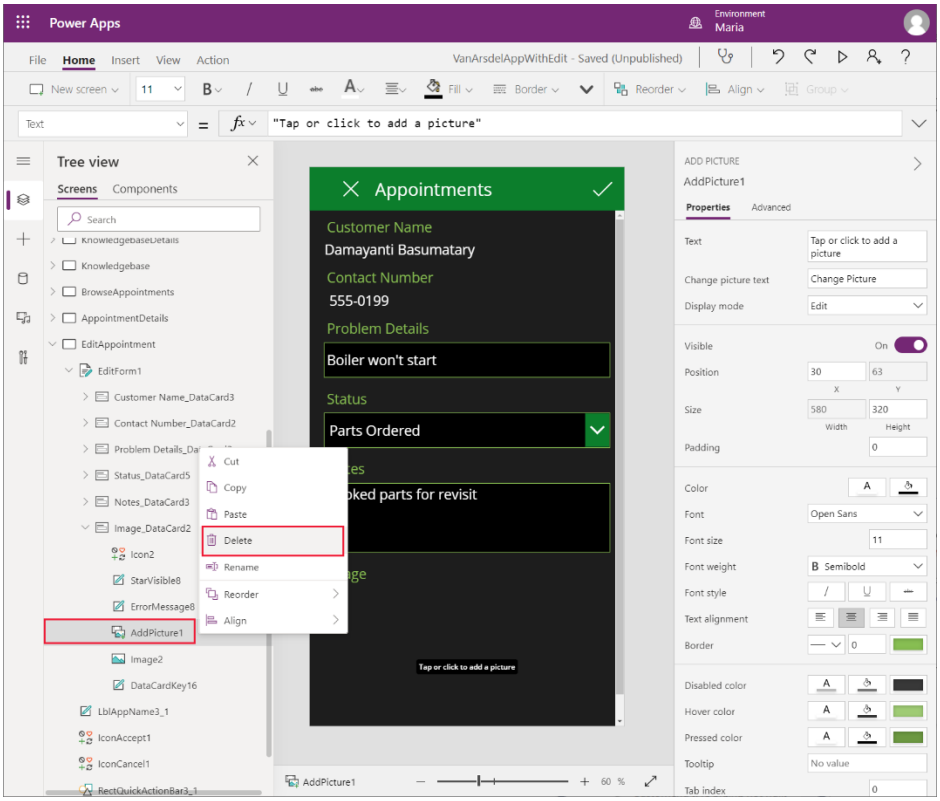
14. In the **Tree view** pane, select the **AddMediaButton*X*** control. Set the **UseMobileCamera** property of this control to **true**. Set the **OnChange** action property of the control to:

```
Set(AppointmentImage, AddMediaButton1.Media)
```

    This formula changes the **AppointmentImage** variable to reference the new image. The **UploadedImage*X*** control will display this image.

15. In the **Tree view** pane, select the **EditAppointment** screen.

16. Expand the **EditForm*X*** control. Under the **Image_DataCardX** control, remove the **AddPictureX** control:



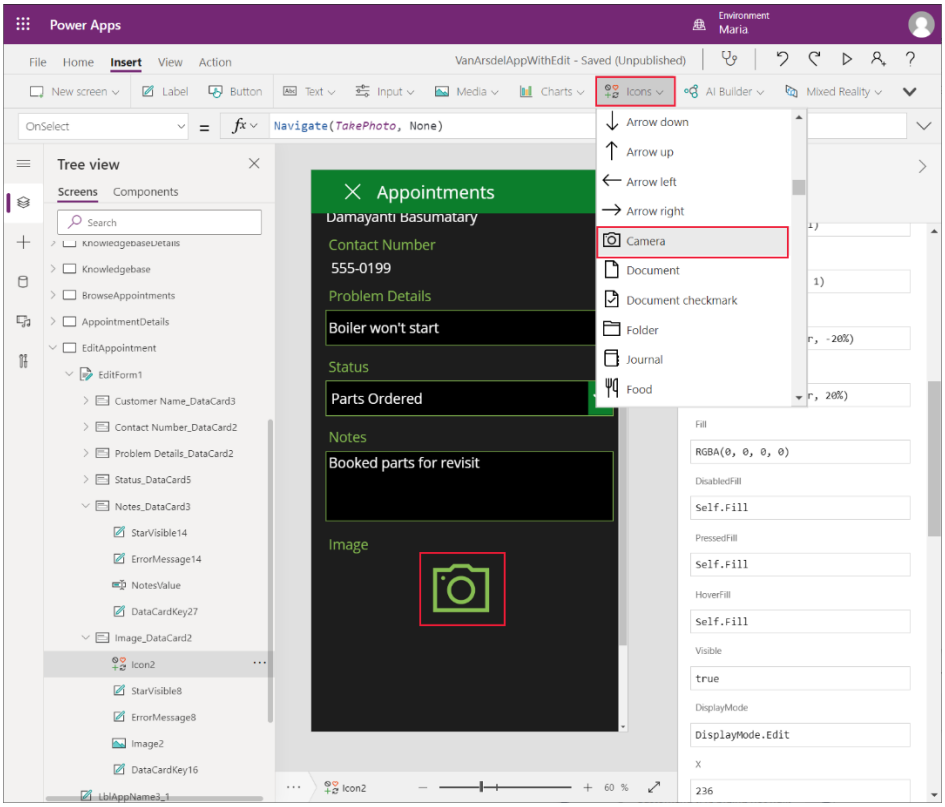
17. Select the **Image*X*** control. Change the following properties:

    - Image: **Parent.Default**
    - X: **30**
    - Y: **DataCardKey*X*.Y + DataCardKey*X*.Height + 150** (where **DataCardKey*X*** is the data card containing the **Image*X*** control)
    - Width: **Parent.Width – 60**
    - Height: **400**

> **Note**: The image control will drop down below the bottom of the screen, but a scroll bar will be added automatically to enable the image to be viewed.

18. Add a **Camera** icon to the data card then position it between the **Image** label and the **ImageX** control. Change the name of the control to **CameraIcon**:

> **Note**: Make sure you select the Camera Icon control, and **not** the Camera Media control.



19. Set the **OnSelect** action property of the **CameraIcon** control to:

```
Set(AppointmentImage, SampleImage);
Navigate(TakePhoto, ScreenTransition.None);
```

When the user clicks this icon, they will go to the **TakePhoto** screen, to enable them to take a photo or upload an image. The initial image displayed will be the default sample image.

To test the app:

1. In the **Tree view** pane, select the **Home** screen.
2. Press **F5** to preview the app.
3. On the **Home** screen, select **Appointments**.
4. In the browse screen, select any appointment.
5. On the details screen for the appointment, select the edit icon in the screen header.
6. On the edit screen, select the **Camera** icon for the image.
7. Verify that the **Take a photograph** screen appears.
8. Select **Change Picture** and upload a picture of your choice (or take a photograph if you're running on a mobile device).
9. Select **Save**. Verify that the image appears on the details page, and then select the tick icon to save the changes back to the database.
10. Close the preview window and return to Power Apps Studio.

DISPLAYING IMAGES OF PARTS

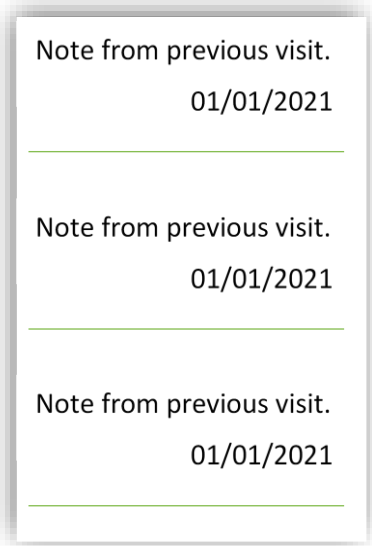
Having determined that Azure Blob Storage is an ideal location for saving pictures associated with appointments, Preeti and Kiana decide that they should use the same approach for storing the images of parts. A key advantage of this approach is that it doesn't require any modifications to the app. The app reuses the same storage account and the same connection. As a separate migration exercise, they can:

1. Create a new Blob container.
2. Upload the part images to this container.
3. Change the **ImageUrl** references in the **Parts** table in the **InventoryDB** database to the URL of each image.

The app will pick up the new URL for each part image automatically, and the **Image** control on the **PartDetails** screen will display the image.

TRACKING APPOINTMENT HISTORY FOR A CUSTOMER

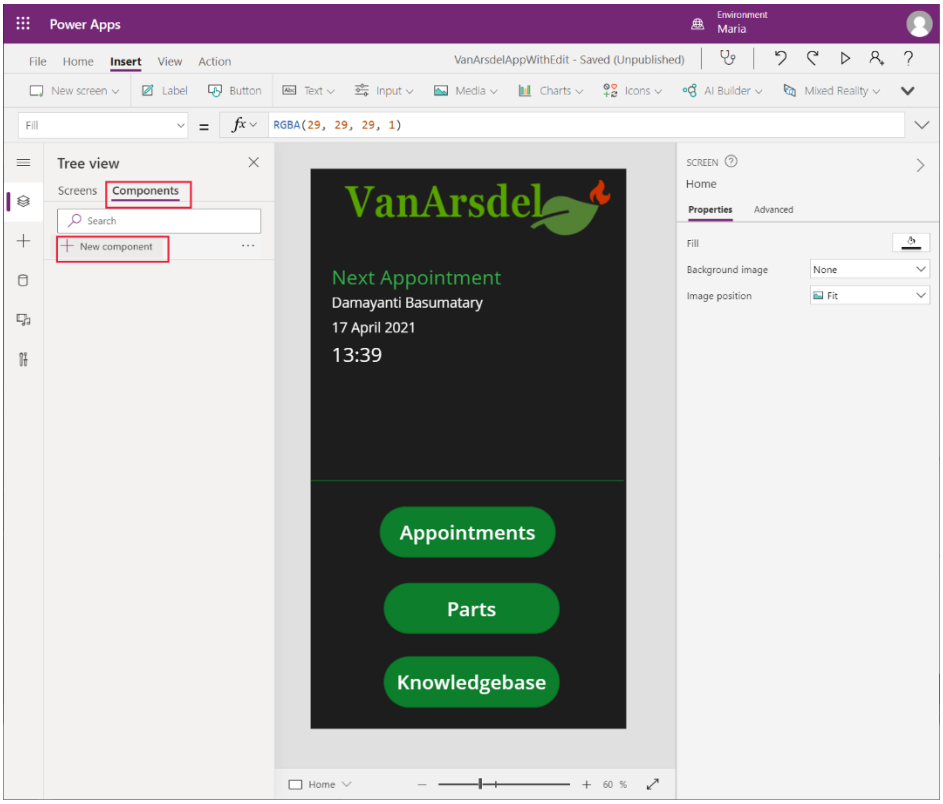
Maria thinks that being able to quickly view all the history from a customer's previous technician's visits could be added to the app by creating a custom component. Working with Caleb on what information they want to see, Maria sketches out a simple design comprising the notes and the date of each visit.



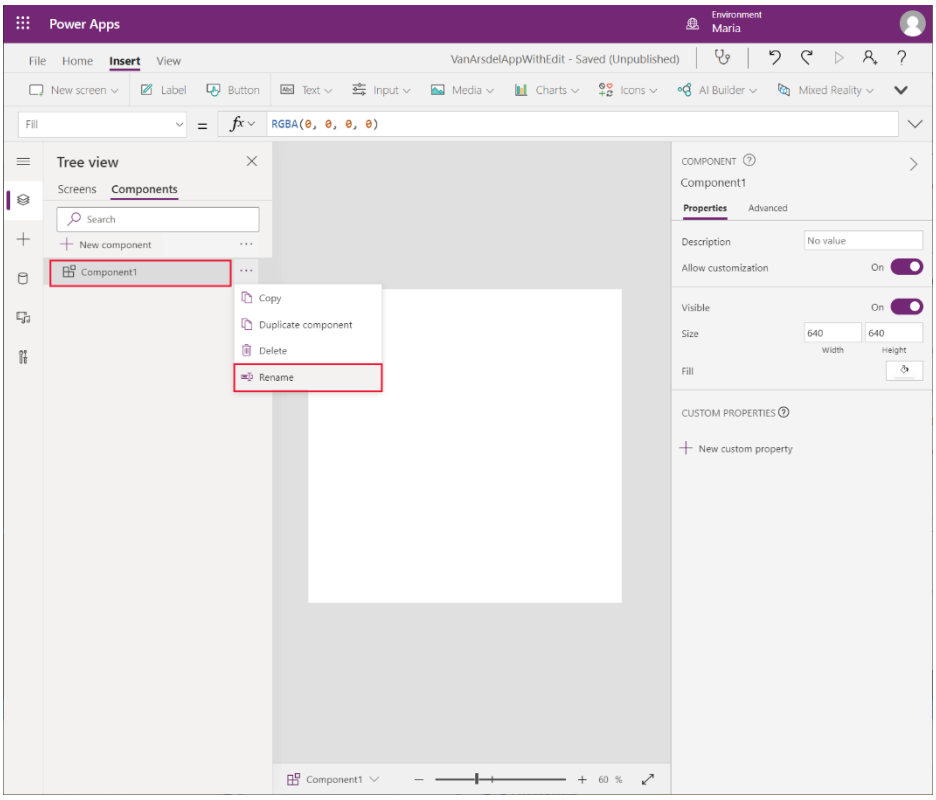
Looking at the data, Maria believes that a gallery control is the best way to display the table data on a screen.

Maria creates the custom component as follows:

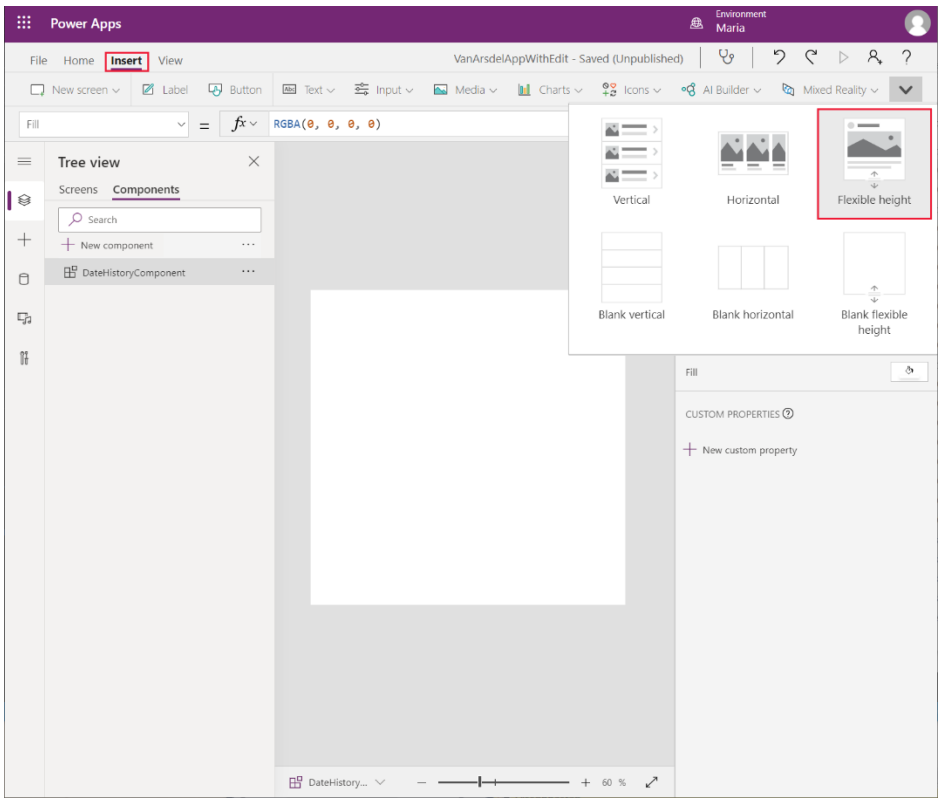
1. Using Power Apps Studio, in the **Tree view** pane, select **Components**, and then select **+ New component**:



A new blank component named **Component1** is created. Rename the component as **DateHistoryComponent**:

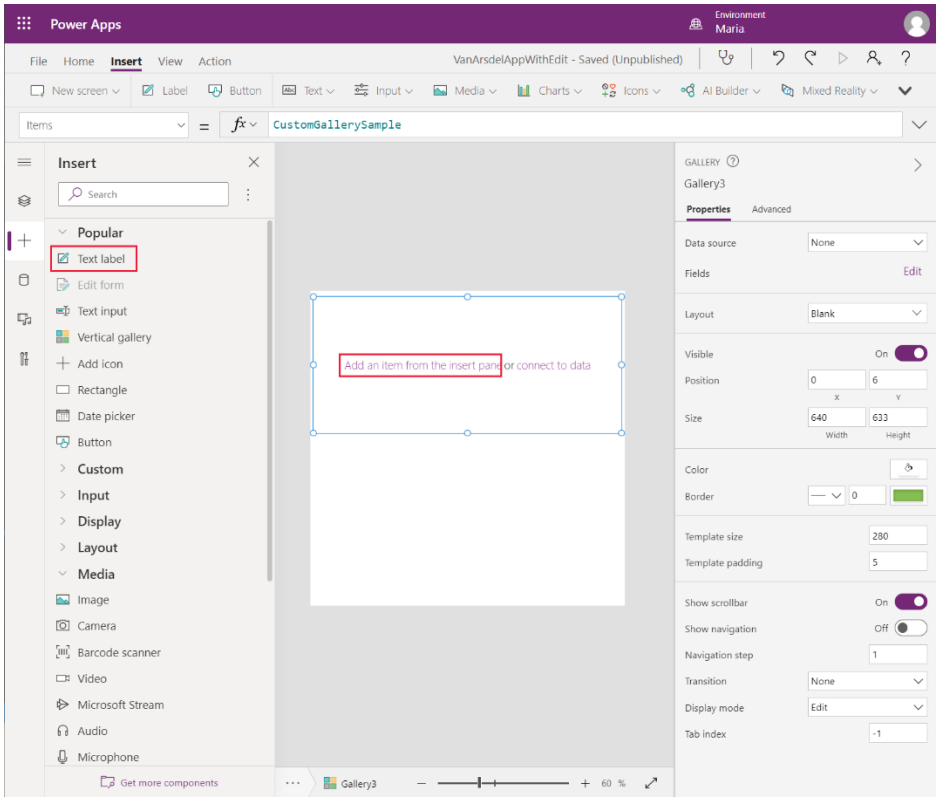


2. On the **Insert** menu, select **Gallery**, and then choose the **Blank flexible height** gallery template:



3. Move the gallery control and resize it to fill the custom component.

4. Select the **Add an item from the insert pane**, then select **Text label**.



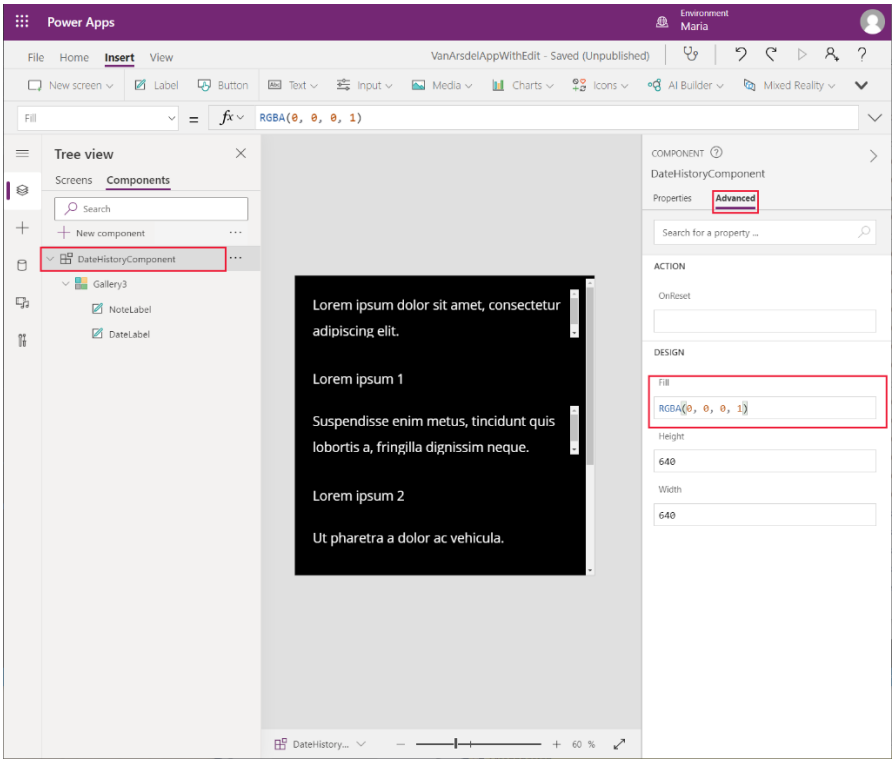
5. In the **Tree view** pane, rename the label control as **NotesLabel**. Set the **Overflow** property to **Overflow.Scroll**. This setting enables the control to display several lines of text and allow the user to scroll through it. Set the following properties so you can position and size the control:

   - LineHeight: **2**
   - X: **28**
   - Y: **18**
   - Width: **574**
   - Height: **140**

6. Add a second text label to the control. Rename this control as **DateLabel** and set the following properties:

   - LineHeight: **2**
   - X: **28**
   - Y: **174**
   - Width: **574**
   - Height: **70**

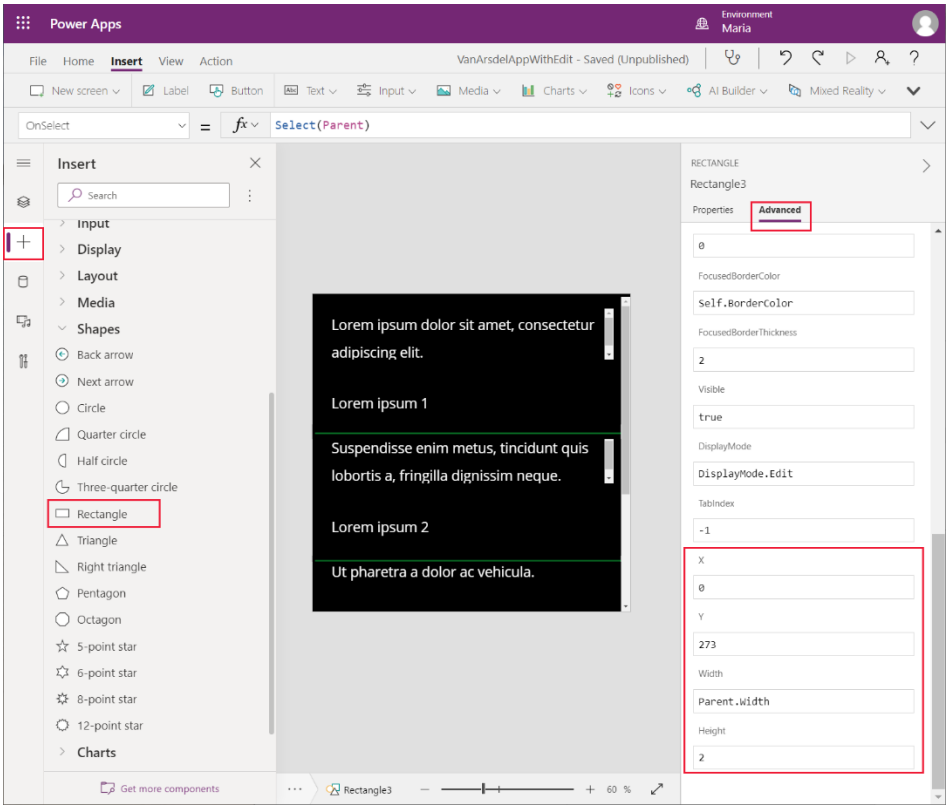
7. To see how the control will look when inserted into the app and using its theme, in the **Tree view** pane, select **DateHistoryComponent**. In the right pane, on the **Advanced** tab, select the **Fill** field and change the color to **RGBA(0, 0, 0, 1)**.



8. From the **Insert** pane, expand **Shapes**, and add a **Rectangle** control to the custom component. Set the following properties for this control:

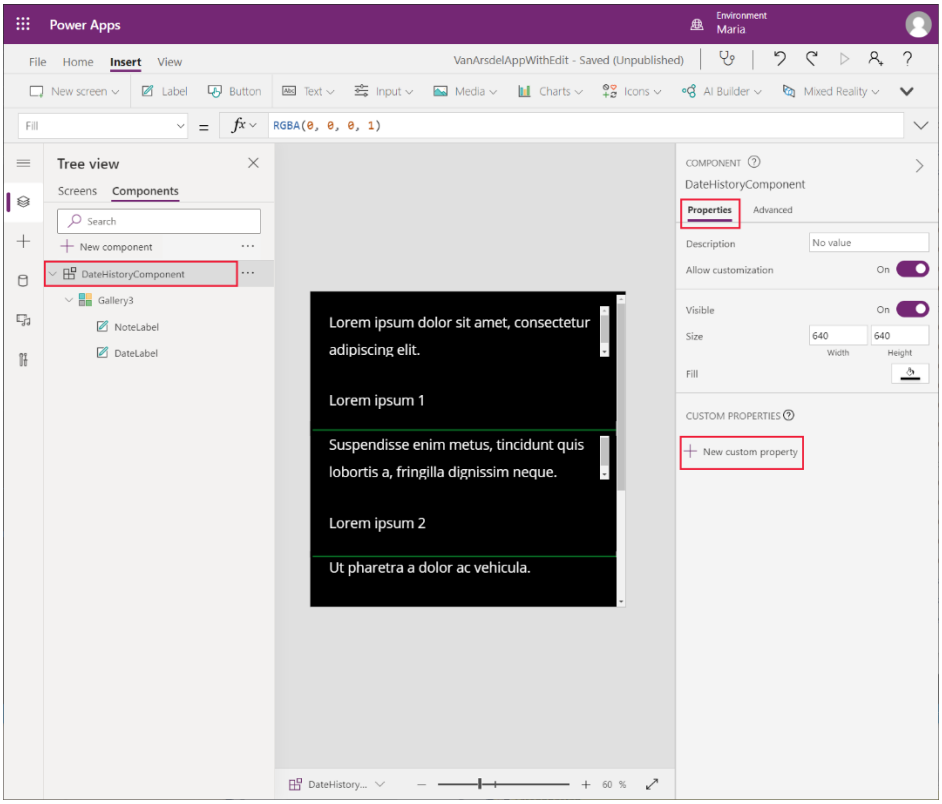
   - X: **0**
   - Y: **273**
   - Width: **Parent.Width**
   - Height: **2**

   This control acts as a separator between the records displayed in the gallery:



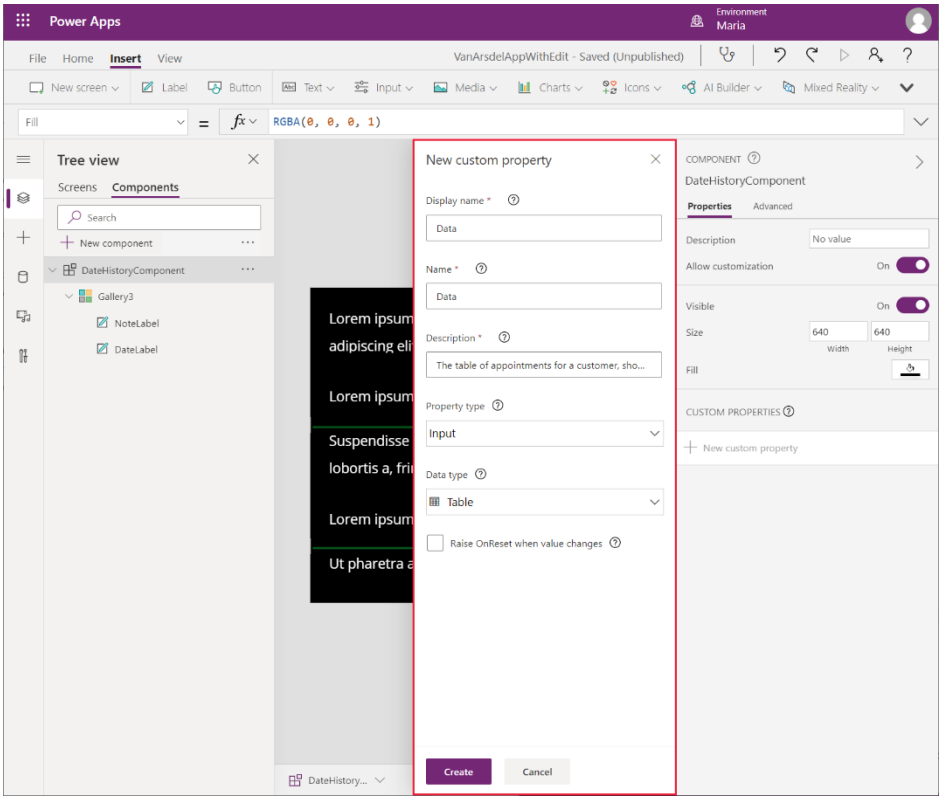
Maria is familiar with adding controls to screens and building Power Apps. However, reusable components don't work in quite the same way. Kiana described to Maria that to be able to use data in a custom component, she must add some additional custom input properties. Kiana also explained that Maria needs to provide example data for these properties, to allow her to reference the data fields in the controls in her component, as follows:

1. In the **Tree view** pane, select **DateHistoryComponent**. In the right pane, on the **Properties** tab, select **New custom property**:

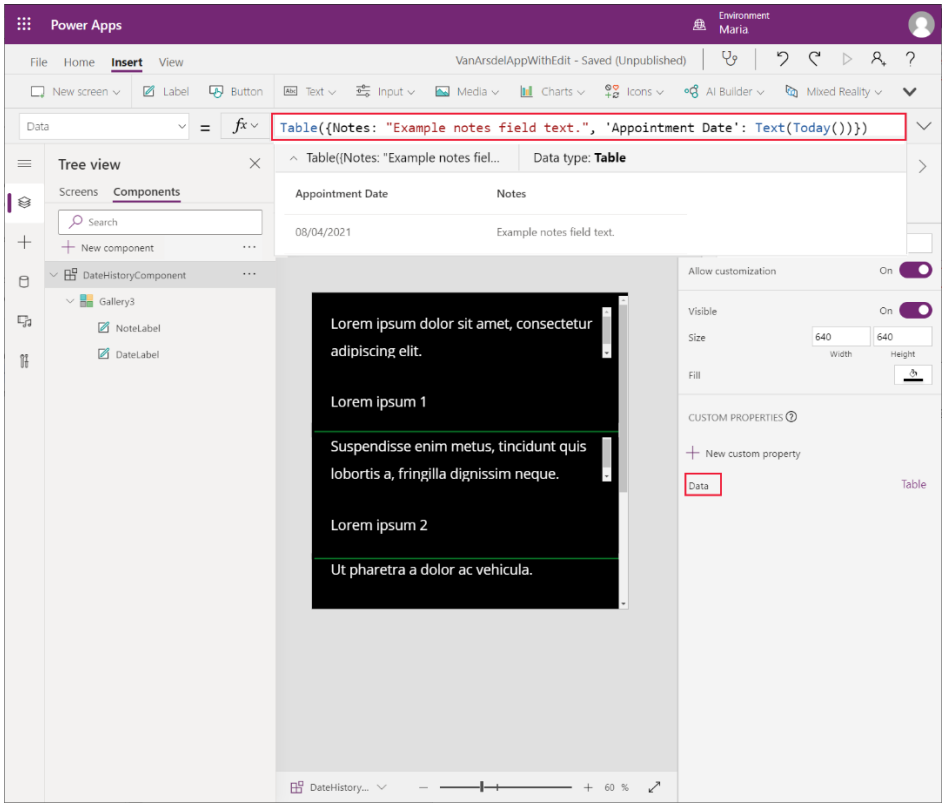


2. In the **New custom property** dialog box, specify the following values, and then select **Create**:

   - Display name: **Data**
   - Name: **Data**
   - Description: **The table of appointments for a customer, showing the notes and dates**
   - Property type: **Input**
   - Data type: **Table**
   - Raise OnReset when value changes: Leave blank

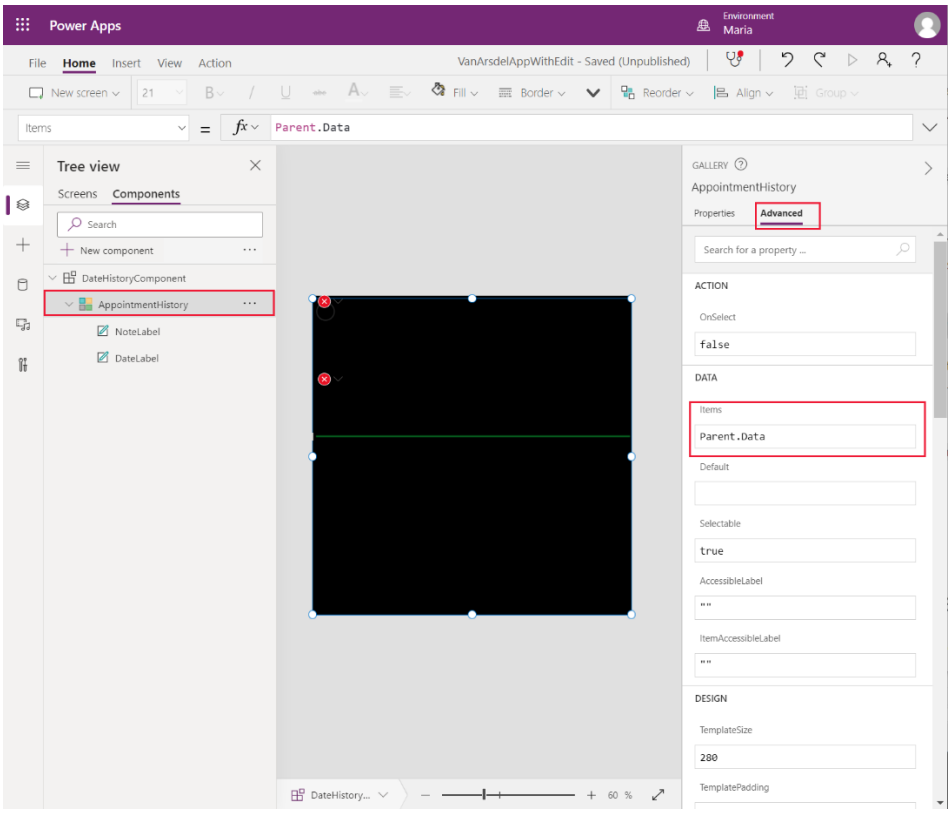


3. To change the sample data displayed by the control, select the new **Data** custom property. In the formula field, type **Table({Notes: "Example notes field text.", 'Appointment Date': Text(Today())})**.



4. In the **Tree view** pane, select the **Gallery*X*** control in **DateHistoryComponent**, and rename it as **AppointmentHistory**.

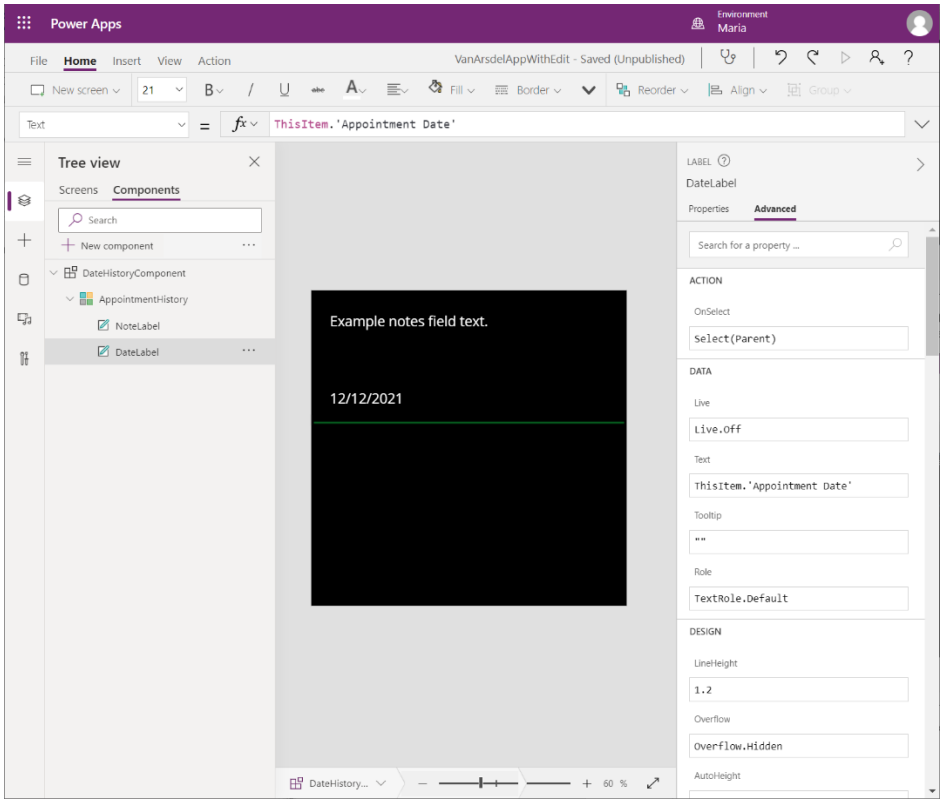
5. In the right pane, on the **Advanced** tab, set the **Items** property of the **AppointmentHistory** gallery control to **Parents.Data**.



6. Select the **NotesLabel** control. In the right pane on the **Advanced** tab, change the **Text** property to **ThisItem.Notes,** and change the **Size** property to **20**.

> **Note**: The **Size** property specifies the font size for the text displayed by the control.

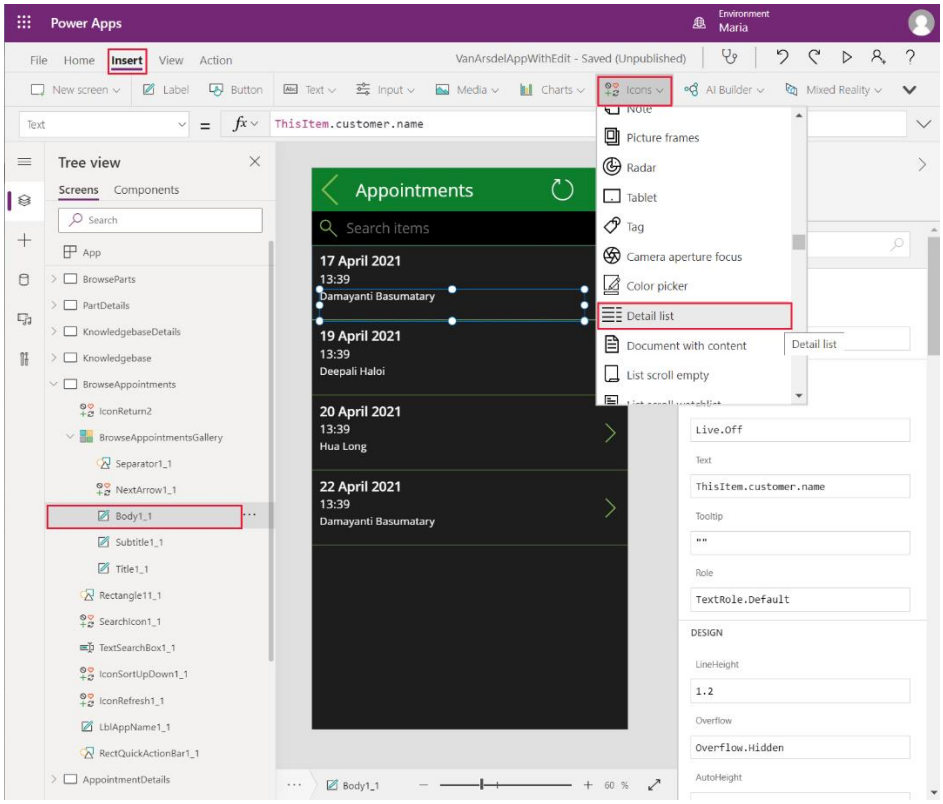
7. Select the **DateLabel** control to change the **Text** property to **ThisItem.'Appointment Date'** and change the **Size** property to **20**. The fields in the custom component should display the sample data:



The custom component is complete. Maria creates a new screen to display the appointment history for a customer using this component:

1. In the **Tree view** pane, select the **Screens** tab.

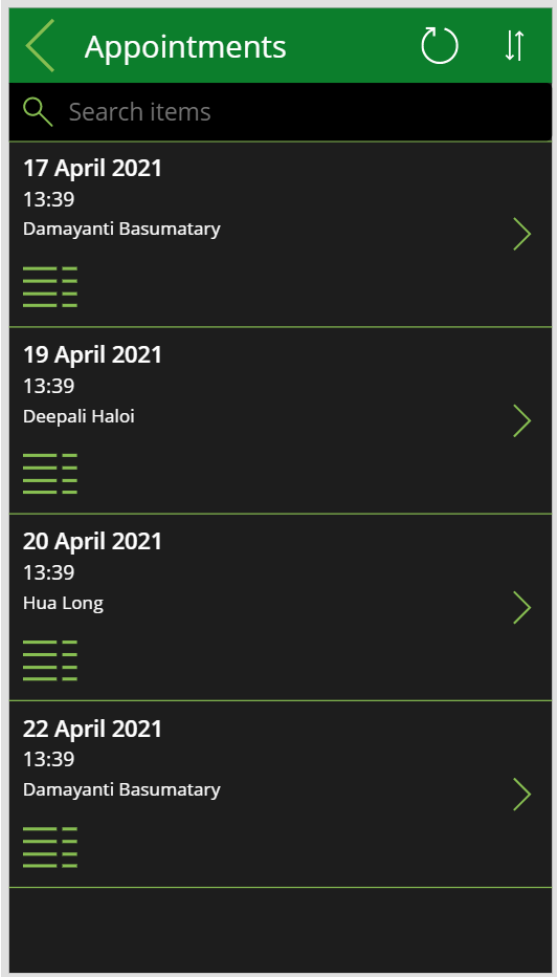
2. Expand the **BrowseAppointments** screen, expand the **BrowseAppointmentsGallery** control, and select the **Body1_1** control. On the **Insert** menu, select **Icons**, and then select the **Detail list** icon:



3. Change the name of the icon control to **ViewAppointments**.

4. In the **Tree view** menu, select the **BrowseAppointmentsGallery** control. In the right pane, on the **Advanced** tab, change the **TemplateSize** property to **220**. Increasing this property expands the space available in the gallery.

5. Move the **ViewAppointments** icon into the empty space below the customer name:

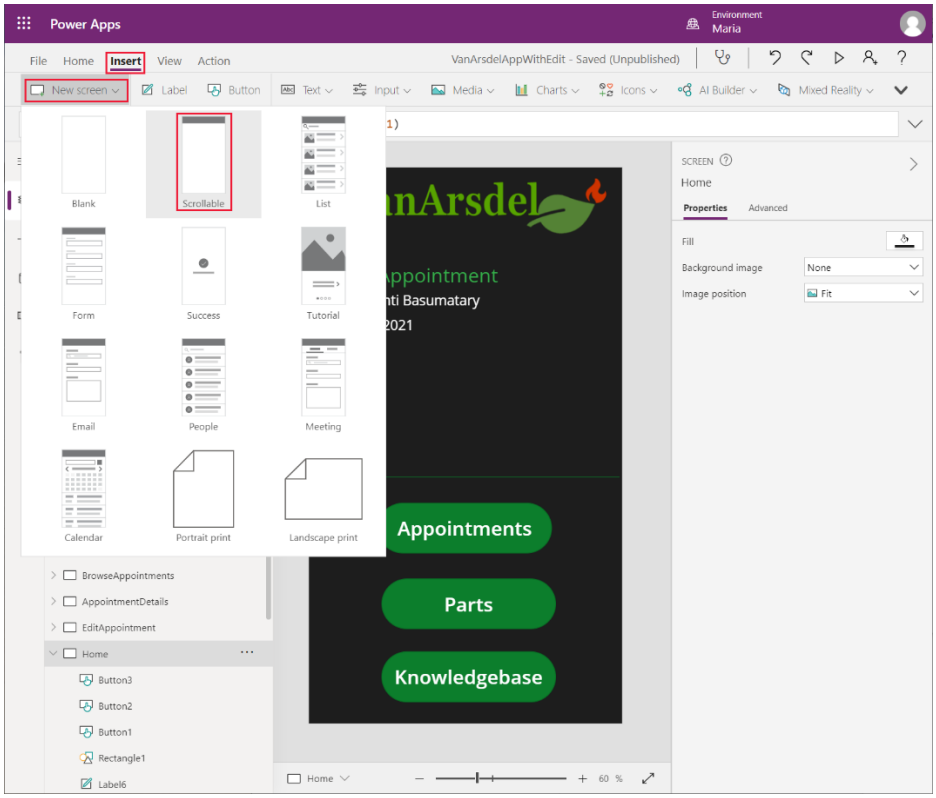


6. Select the **ViewAppointments** icon control. Set the **OnSelect** action property to the following formula:

```
ClearCollect(customerAppointmentsCollection,
FieldEngineerAPI.getapicustomeridappointments(ThisItem.customerId));

Navigate(AppointmentsHistoryScreen, ScreenTransition.Fade)
```

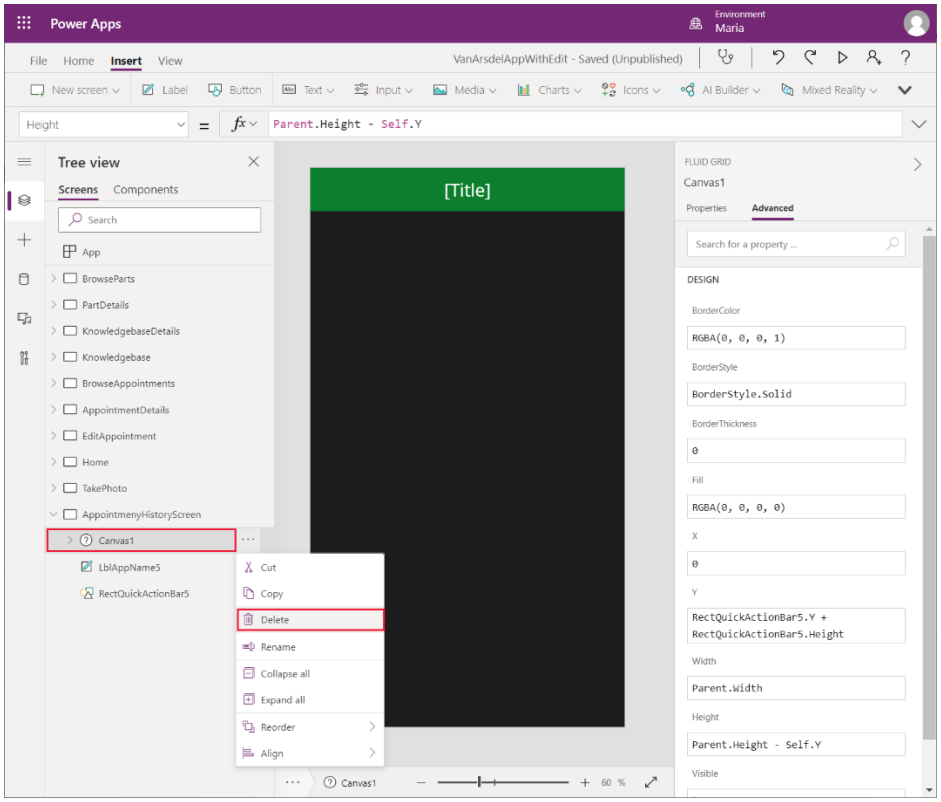
   This formula populates a collection named **customerAppointmentsCollection** with the appointments for the selected customer, and then moves to the **AppointmentHistoryScreen** to display them. You'll create this screen in the following steps.

7. On the **Insert** menu, select **New screen**, and then select the **Scrollable** template:



8. Change the name of the new screen to **AppointmentHistoryScreen**.

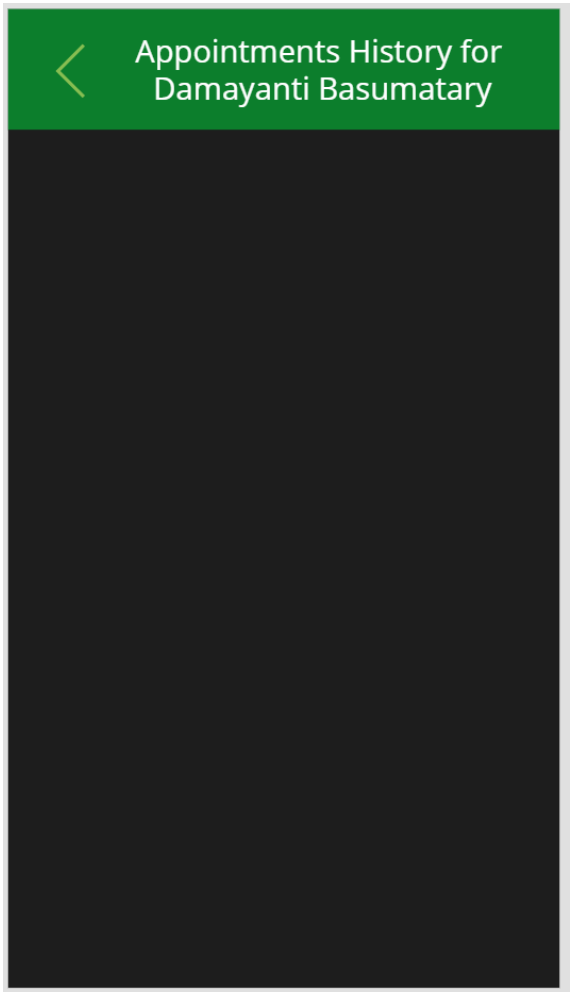
9. Delete the **Canvas*X*** control that was added to this screen:



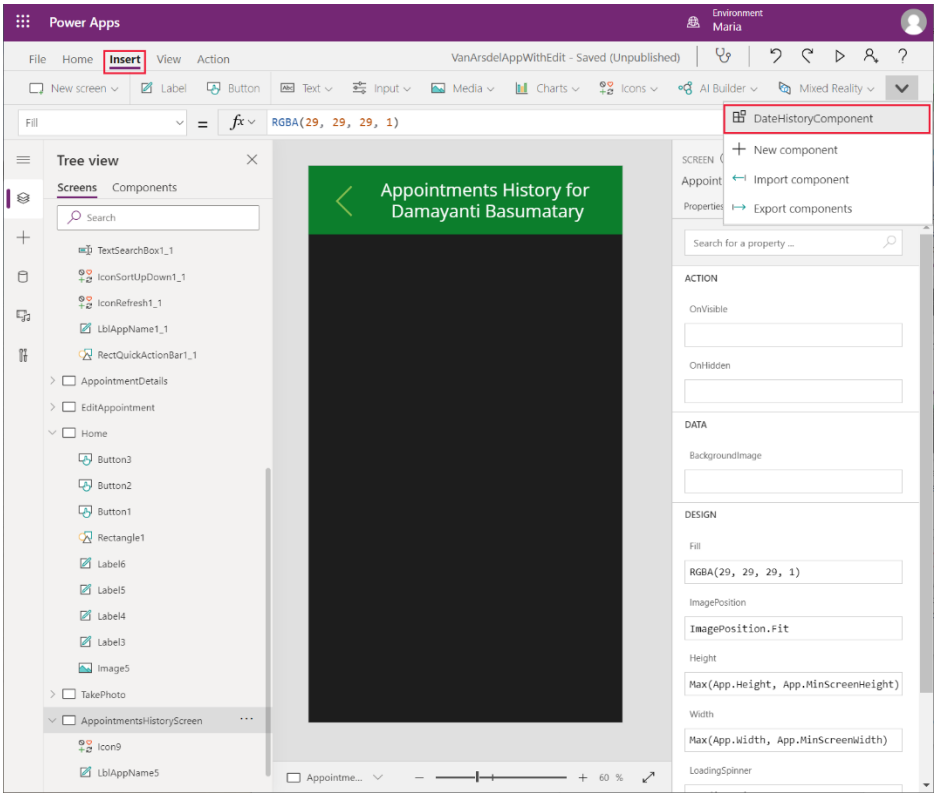
10. Select the **LblAppName*X*** control on this screen. In the right pane, on the **Advanced** tab, change the **Text** property to:

```
"Appointments History for " &
BrowseAppointmentsGallery.Selected.customer.name
```

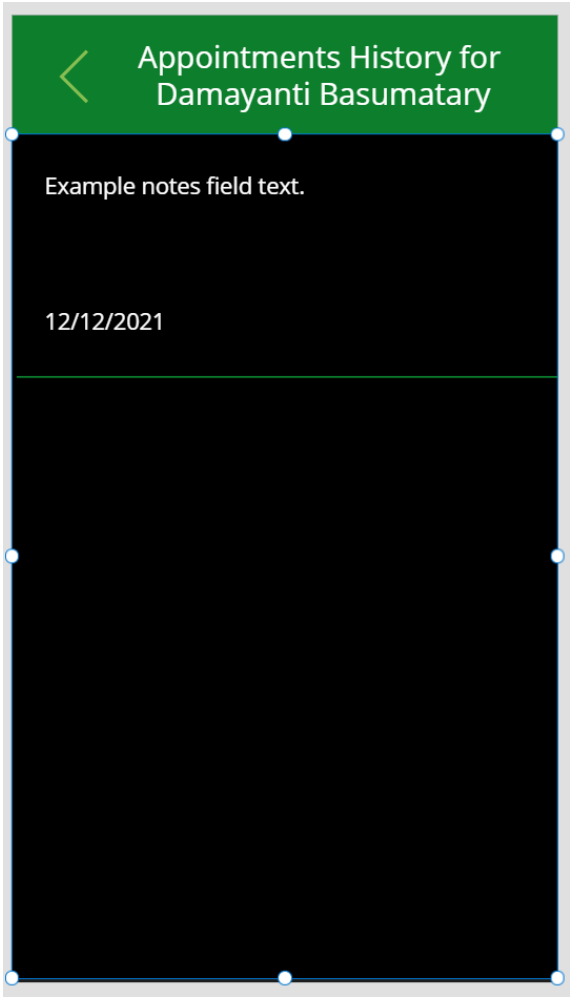
11. Set the following properties for the **LblAppName*X*** control to adjust the position and size:
    - X: **90**
    - Y: **0**
    - Width: **550**
    - Height: **140**
12. Select the **RectQuickActionBar*X*** control, and set the **Height** property to **140**.
13. Add a **Left icon** control to the screen header, to the left of the title. Set the **OnSelect** action property for this control to **Navigate(BrowseAppointments, Transition.None)**.



14. On the **Insert** menu, select **Custom**, and then select the **DateHistoryComponent**:



15. Move and resize the component so that it occupies the body of the screen, below the heading:



16. Set the following properties for this component:
    - Data: **customerAppointmentsCollection**
    - Appointment Date: **startDateTime**
    - Notes: **notes**
17. Save the app.

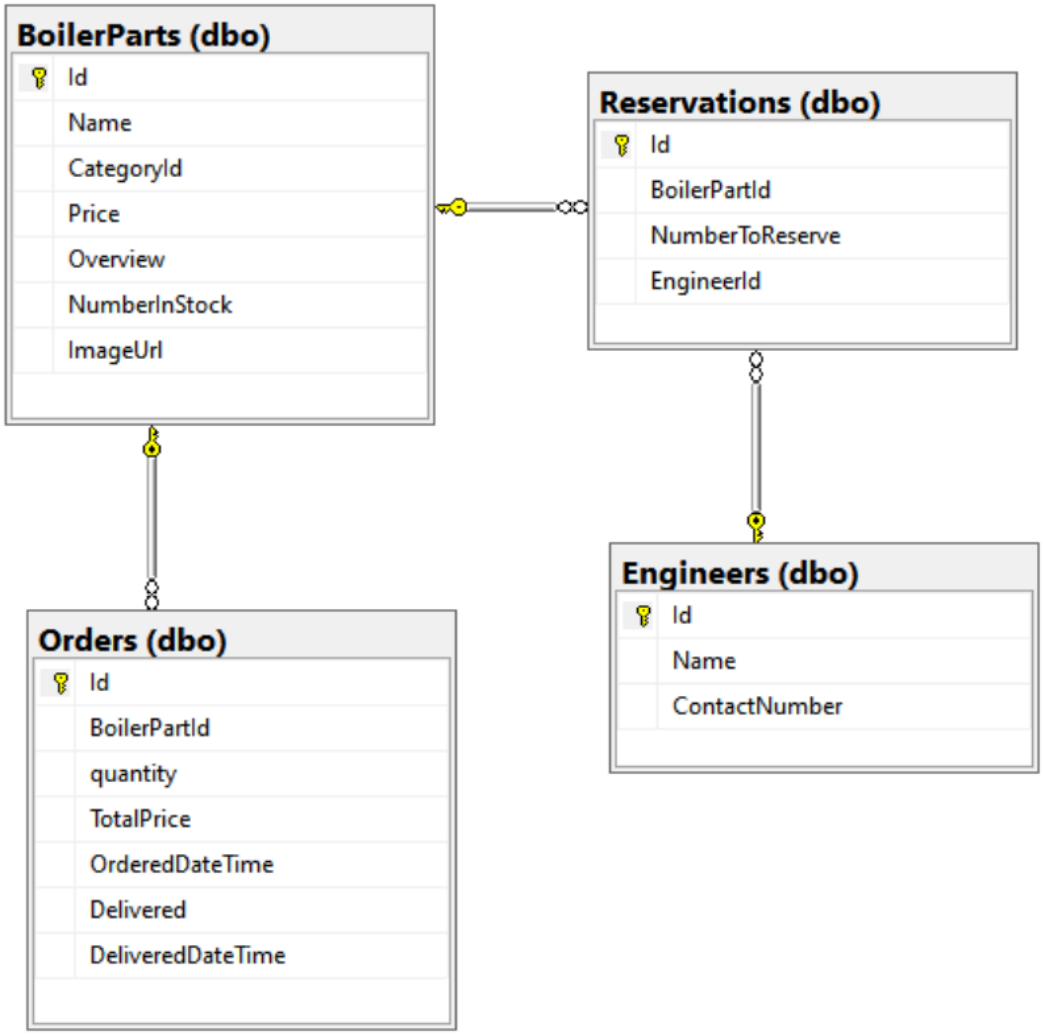
To test the Power Apps:

1. In the **Tree view** pane, select the **Home** screen.
2. Press **F5** to preview the app.
3. On the **Home** screen, select **Appointments**.
4. In the browse screen, select the **Detail list** icon for any appointment.
5. Verify that the **Appointments History** screen for the selected customer appears.
6. Close the preview window and return to Power Apps Studio.

ORDERING PARTS

A key requirement of the system is to enable a technician to order any parts required while visiting a customer. If the parts are in stock, it should be possible to schedule another visit to complete the repair at the next convenient date for the customer. If the parts are currently out of stock and have to be ordered, the technician can tell the customer. Malik can then arrange an appointment with the customer when Maria receives notice that the parts have arrived in the warehouse.

The reservations part of the app uses the tables in the **InventoryDB** database shown in the diagram below. The **Orders** table holds information about orders placed for parts. The **Reservations** table lists the reservation requests that technicians and engineers have made for parts. The **Engineers** table provides the name and contact number for the engineer who made the reservation, in case of any queries by Maria, the inventory manager.



To support this feature, Kiana has to update the Web API with a method that fetches the number of reserved items for a specified part:

1. Open the **FieldEngineerApi** Web API project in Visual Studio Code.

2. Add a file named **Order.cs** to the **Models** folder. Add the following code to this file. The **Orders** class tracks the details of orders placed for parts.

```
using System;
using System.ComponentModel.DataAnnotations;
```

```
using System.ComponentModel.DataAnnotations.Schema;

namespace FieldEngineerApi.Models
{
    public class Order
    {
        [Key]
        public long Id { get; set; }
        public long BoilerPartId { get; set; }
        public BoilerPart BoilerPart { get; set; }
        public long Quantity { get; set; }
        [Column(TypeName = "money")]
        public decimal TotalPrice { get; set; }
        [Display(Name = "OrderedDate")]
        [DataType(DataType.DateTime)]
        [DisplayFormat(DataFormatString = "{0:MM/dd/yyyy}")]
        public DateTime OrderedDateTime { get; set; }
        public bool Delivered { get; set; }
        [Display(Name = "DeliveredDate")]
        [DataType(DataType.DateTime)]
        [DisplayFormat(DataFormatString = "{0:MM/dd/yyyy}")]
        public DateTime? DeliveredDateTime { get; set; }
    }
}
```

3. Add another new file named **Reservation.cs** to the **Models** folder and add the code shown below to this file. The **Reservation** class contains information about the number of items for a given part that are currently reserved for other customers.

```
using System;
using System.ComponentModel.DataAnnotations;

namespace FieldEngineerApi.Models
{
    public class Reservation
    {
        [Key]
        public long Id { get; set; }
        public long BoilerPartId { get; set; }
        public BoilerPart BoilerPart { get; set; }
        public int NumberToReserve { get; set; }
        public string EngineerId { get; set; }
        public InventoryEngineer Engineer { get; set; }
    }
}
```

4. Add one more file, named **InventoryEngineer.cs,** to the **Models** folder, with the following code. The **InventoryEngineer** class records which engineers have made which reservations:

```
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;

namespace FieldEngineerApi.Models
{
    public class InventoryEngineer
    {
        [Key]
        public string Id { get; set; }
        [Required]
        public string Name { get; set; }
        public string ContactNumber { get; set; }
```

```
        public List<Reservation> Reservations { get; set; }
    }
}
```

5. Open the **InventoryContext.cs** file in the **Models** folder, and add the statements shown below in bold to the **InventoryContext** class:

```
public class InventoryContext : DbContext
{
    public InventoryContext(DbContextOptions<InventoryContext> options)
        : base(options)
    {

    }

    public DbSet<BoilerPart> BoilerParts { get; set; }
    public DbSet<InventoryEngineer> Engineers { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<Reservation> Reservations { get; set; }
}
```

6. In the Terminal window in Visual Studio Code, run the following commands to build controllers for handling orders and reservations:

```
dotnet aspnet-codegenerator controller ^
  -name OrdersController -async -api ^
  -m Order ^
  -dc InventoryContext -outDir Controllers

dotnet aspnet-codegenerator controller ^
  -name ReservationsController -async -api ^
  -m Reservation ^
  -dc InventoryContext -outDir Controllers
```

7. Open the **BoilerPartController.cs** file in the **Controllers** folder, and add the **GetTotalReservations** method, highlighted below in bold, to the **BoilerPartsController** class:

```
public class BoilerPartsController : ControllerBase
{
    private readonly InventoryContext _context;

    public BoilerPartsController(InventoryContext context)
    {
        _context = context;
    }

    ...

   // GET: api/BoilerParts/5/Reserved
   [HttpGet("{id}/Reserved")]
   public async Task<ActionResult<object>> GetTotalReservations(long id)
   {
     var reservations = await _context
        .Reservations
        .Where(r => r.BoilerPartId == id)
        .ToListAsync();

     int totalReservations = 0;

     foreach(Reservation reservation in reservations)
```

```
        {
            totalReservations += reservation.NumberToReserve;

        }

        return new {id, totalReservations};
    }

    ...
}
```

8. Edit the **OrdersController.cs** file, and modify the **PostOrder** method in the **OrdersController** class as highlighted below in bold:

```
[HttpPost]
public async Task<ActionResult<Order>> PostOrder(long boilerPartId, int
quantity)
{
    var part = await _context.BoilerParts.FindAsync(boilerPartId);
    Order order = new Order
    {
        BoilerPartId = boilerPartId,
        Quantity = quantity,
        OrderedDateTime = DateTime.Now,
        TotalPrice = quantity * part.Price
    };

    _context.Orders.Add(order);
    await _context.SaveChangesAsync();

    return CreatedAtAction("GetOrder", new { id = order.Id }, order);
}
```

9. Edit the **ReservationsController.cs** file. Modify the **PostReservation** method in the **ReservationsController** class as follows:

```
[HttpPost]
public async Task<ActionResult<Reservation>> PostReservation(long
boilerPartId, string engineerId, int quantityToReserve)
{
    Reservation reservation = new Reservation
    {
        BoilerPartId = boilerPartId,
        EngineerId = engineerId,
        NumberToReserve = quantityToReserve
    };

    _context.Reservations.Add(reservation);
    await _context.SaveChangesAsync();

    return CreatedAtAction("GetReservation", new { id =
reservation.Id }, reservation);
}
```

10. In the Terminal window, run the following commands to build and publish the Web API ready for deployment:

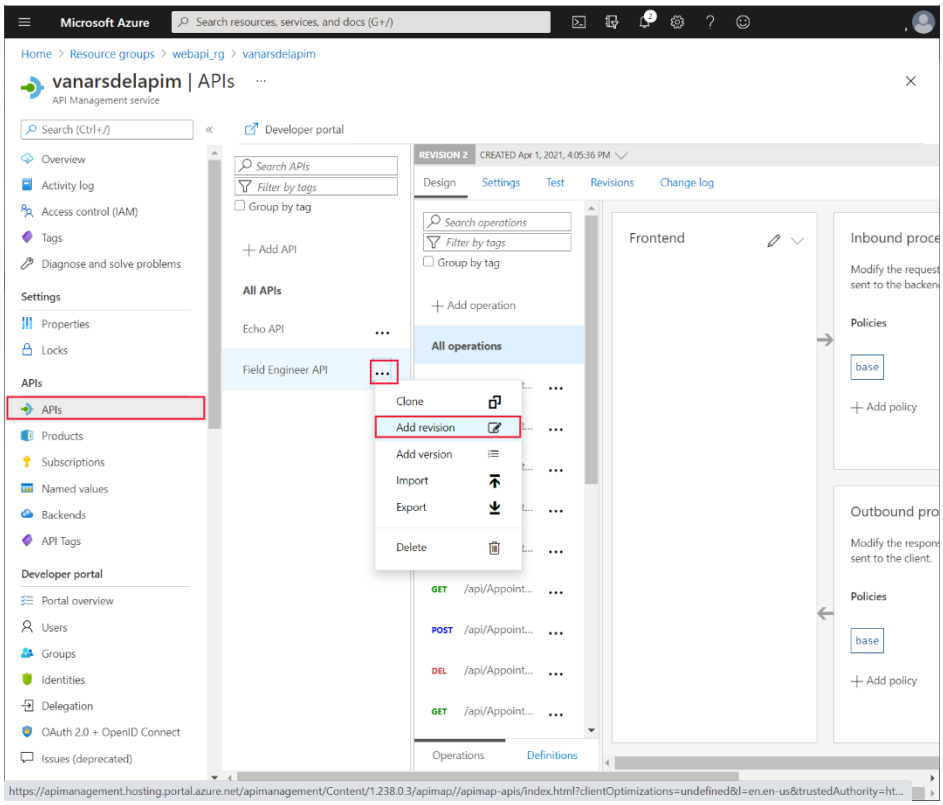
```
dotnet build
dotnet publish -c Release -o ./publish
```

11. In Visual Studio Code, right-click the **publish** folder, and then select **Deploy to Web App**:

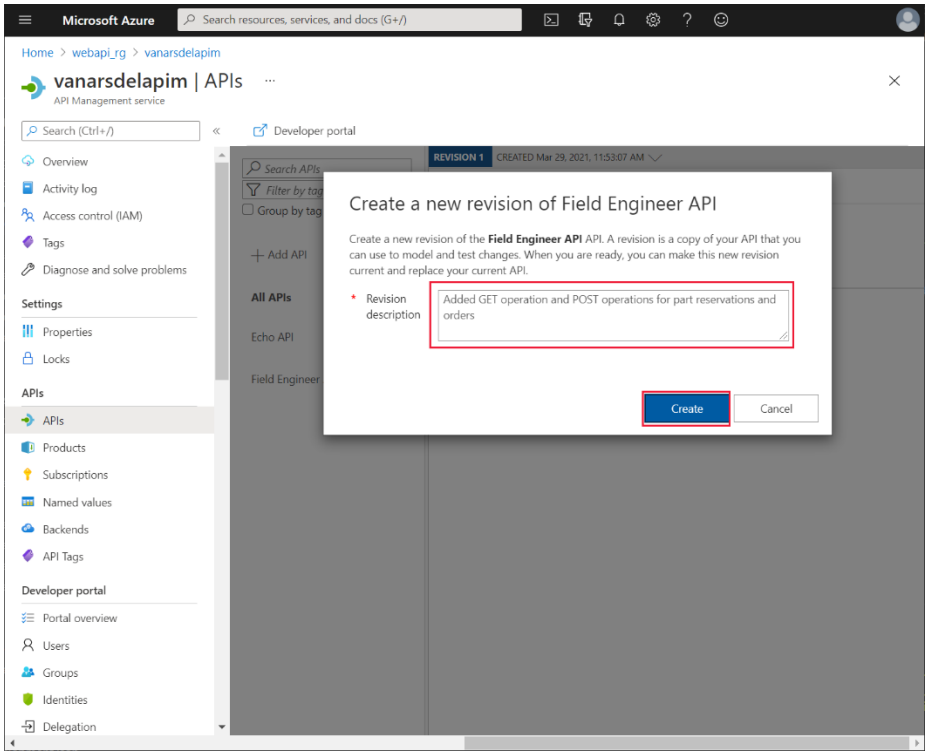
Preeti can now update the Azure API Management service used by the VanArsdel app to reflect the updated Web API. This is a non-breaking change; existing operations will continue to work, the differences being the new controllers and operations to make reservations and place orders. Preeti performs the following tasks:

> **Note**: Preeti could have chosen to delete the existing Field Engineer API and replace it with a new version, but this approach risks breaking any existing applications that may be currently using the API. It's better practice to leave the existing API in place and add the modifications as a revision.

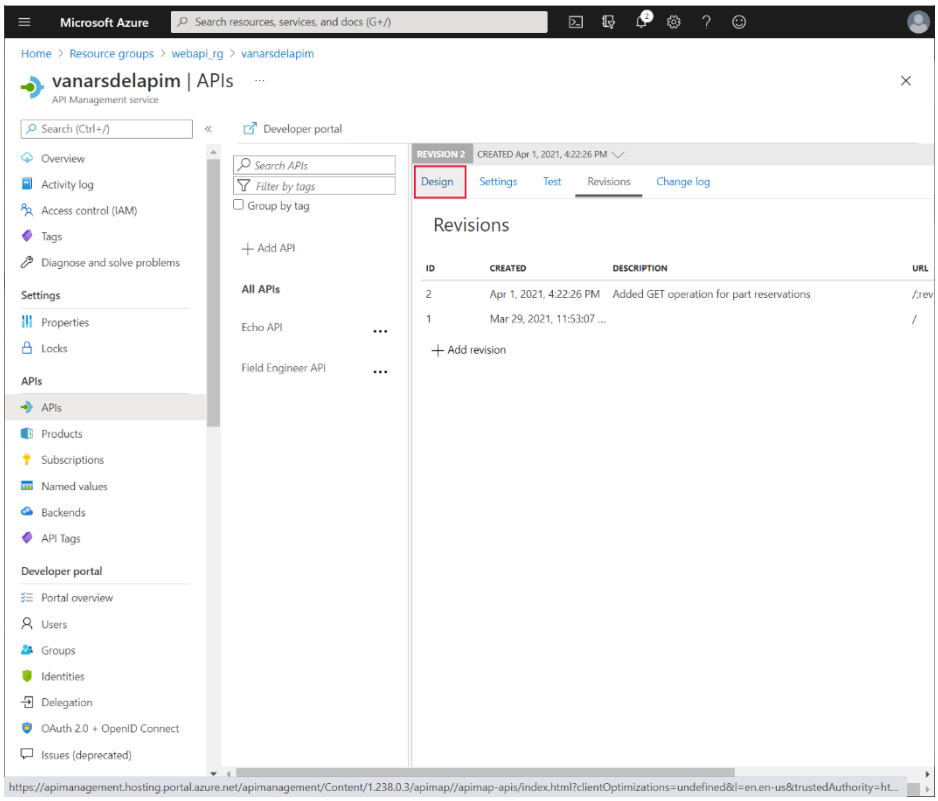
1. In the Azure portal, go to the APIM service.
2. On the **API Management service** page, in the left pane, under **APIs**, select **APIs**:
3. Select the **Field Engineer API**, select the ellipsis menu, and then select **Add revision**:



4. In the **Create a new revision of the Field Engineer API** dialog box, enter the description **Added GET operation and POST operations for part reservations and orders**, and then select **Create**:



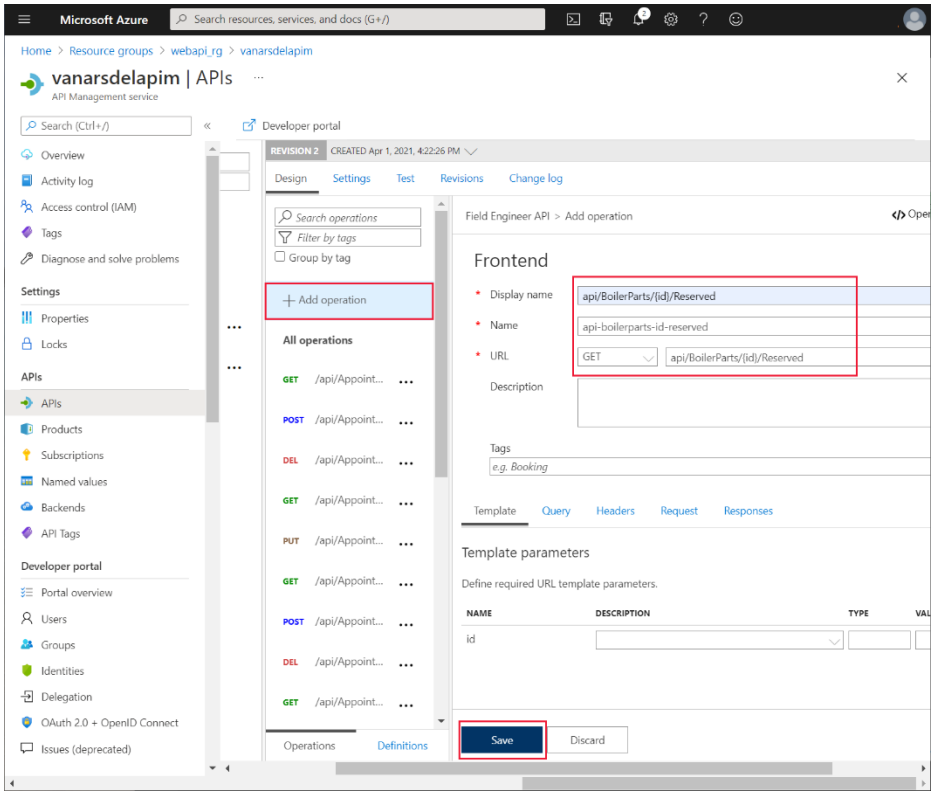
5. On the **REVISION 2** page, select **Design**:



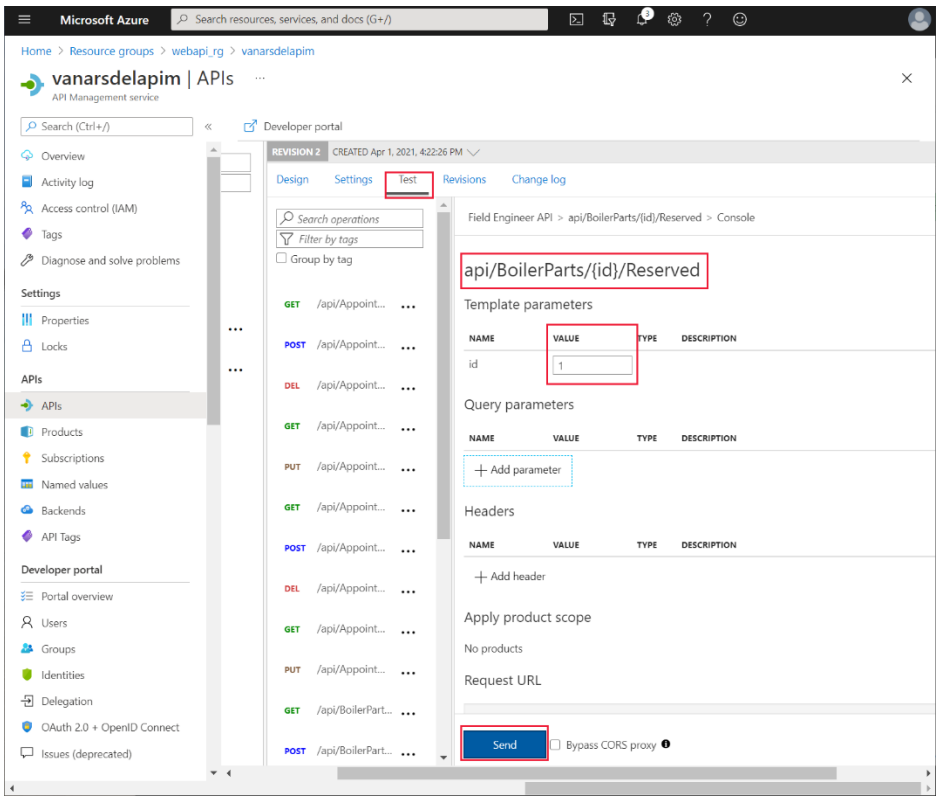
6. On the **Design** page, select **Add operation**. In the **FrontEnd** pane, set the following properties, and then select **Save**. This operation is used for retrieving the number of items reserved for a given boiler part:

   - Display name: **api/BoilerParts/{id}/Reserved**
   - Name: **api-boilerparts-id-reserved**

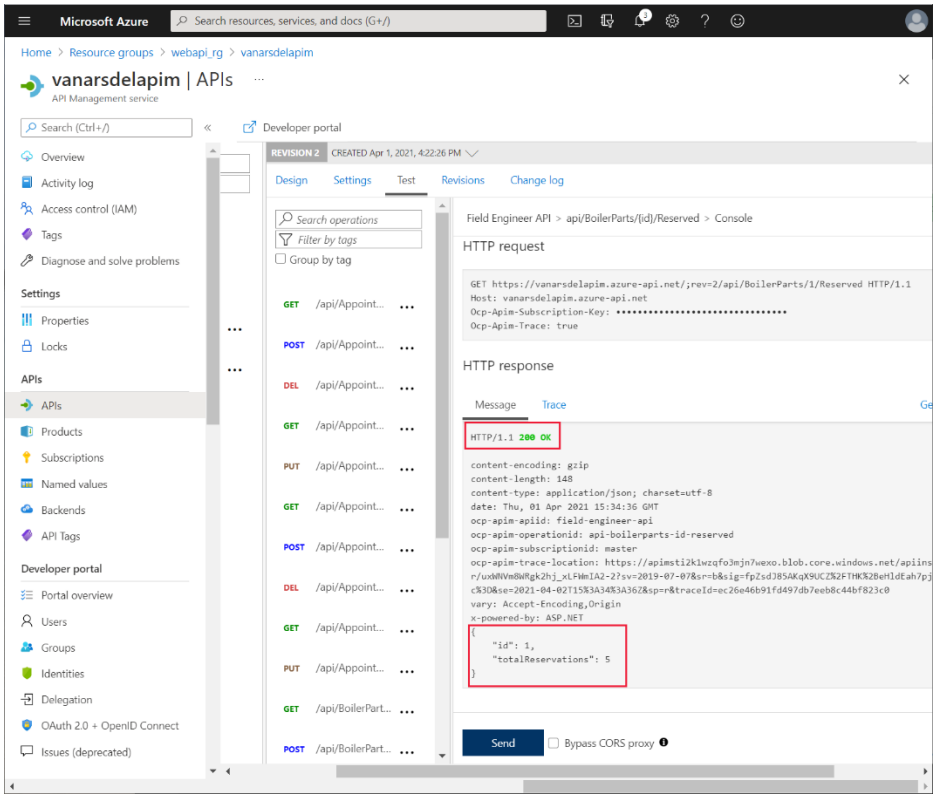
- URL: **GET api/BoilerParts/{id}/Reserved**



7. On the **Test** tab for the new operation, set the **id** parameter to a valid part number (the example in the image uses part 1), and then select **Send**:



8. Verify that the test is successful. The operation should complete with an HTTP 200 response, and a body that shows the number of reservations for the product:

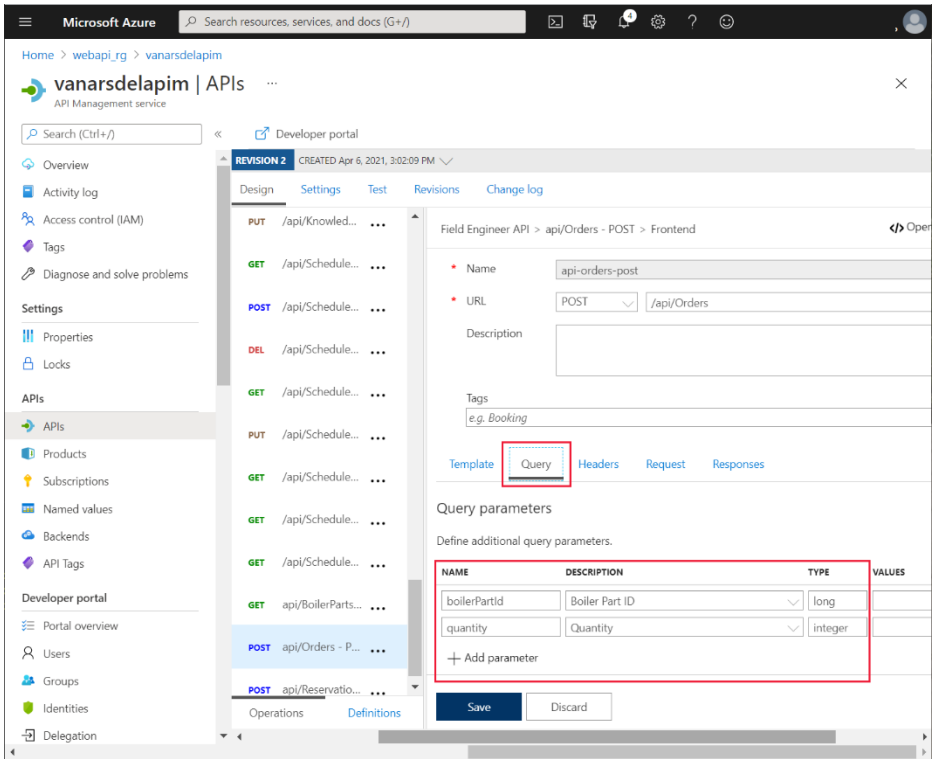


9. On the **Design** page, select **Add operation**. In the **FrontEnd** pane, set the following properties. This operation defines POST requests for creating new orders:

    - Display name: **api/Orders - POST**
    - Name: **api-orders-post**
    - URL: **POST api/Orders**

10. On the **Query** tab, select **+ Add parameter**, add the following parameters, and then select **Save**:

    - Name: **boilerPartId**, Descripion**: Boiler Part ID**, Type: **long**
    - Name: **quantity**, Descripion**: Quantity**, Type : **integer**



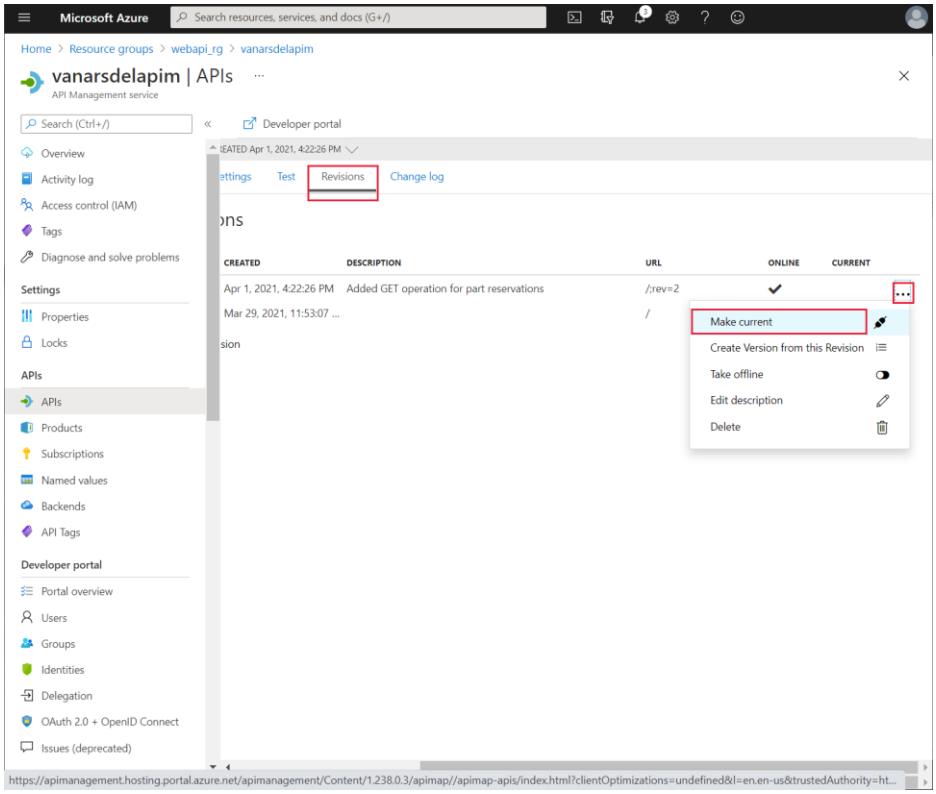
11. Select **Add operation** again In the **FrontEnd** pane, and set the following properties. This operation defines POST requests for creating new reservations:

    - Display name: **api/Reservations - POST**
    - Name: **api-reservations-post**
    - URL: **POST api/Reservations**

12. On the **Query** tab, add the following parameters, and then select **Save**:

    - Name: **boilerPartId**, Descripion: **Boiler Part ID**, Type: **long**
    - Name: **engineerId**, Description: **Engineer ID**, Type: **string**
    - Name: **quantityToReserve**, Descripion: **Quantity to reserve**, Type: **integer**

13. On the **Revisions** tab, select the new version. On the ellipsis menu for this version, select **Make current**:



14. In the **Make revision current** dialog box, select **Save**.

15. Open another page in your web browser and go to the URL **https://*<APIM name>*.azure-api.net/api/boilerparts/1/reserved** where **<APIM name>** is the name of your API service. Verify that you get a response similar to the following:

```
{"id":1,"totalReservations":5}
```

The updated Web API is now available. In theory, Kiana could create a new custom connector for the updated Web API and add it to the app. The app could then implement its own logic to determine how many items of the specified product are currently in stock, how many are reserved, compare the results to the number of items required, place an order for more stock if necessary, or reserve items from the existing stock. However, this kind of logic is better implemented in an Azure Logic App. The app can call the Logic App through a custom connector when a technician wishes to reserve or order a part.

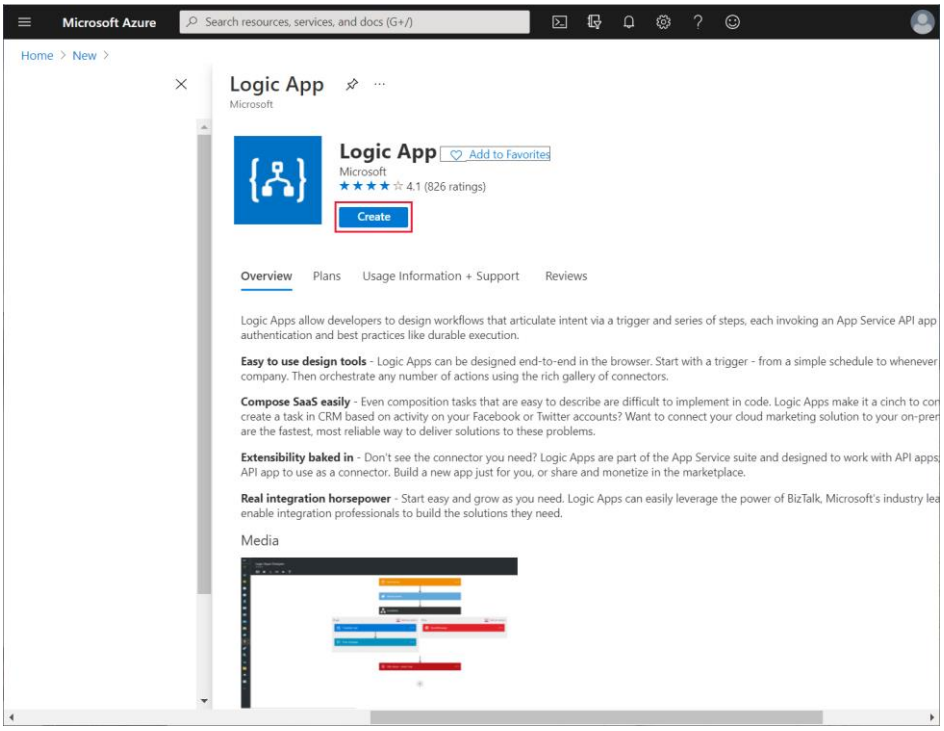
To create the Logic App, Kiana uses the following steps:

> **Note**: To keep things simple, the Logic App created in this example is non-transactional. It's possible that between checking the availability of a part and making a reservation, a concurrent user might make a conflicting reservation. You could implement transactional semantics by replacing some of the logic in this Logic App with a stored procedure in the **InventoryDB** database.

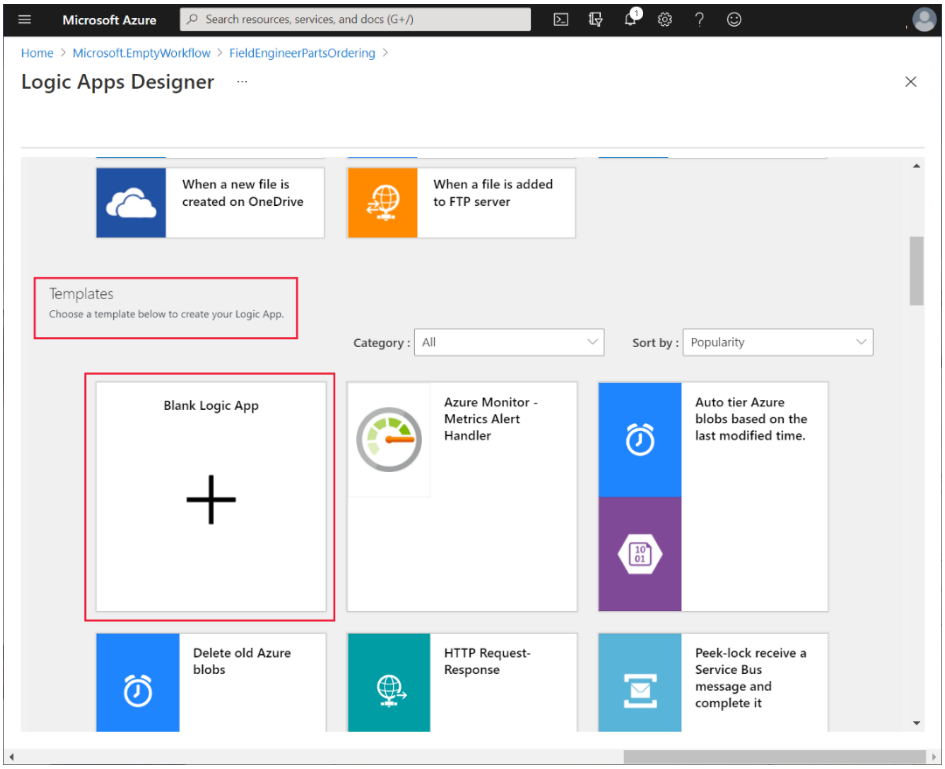
1. In the Azure portal, on the **Home** page, select **+ Create a resource**.

2. In the **Search the marketplace** box, type **Logic App**, and then press **Enter**.

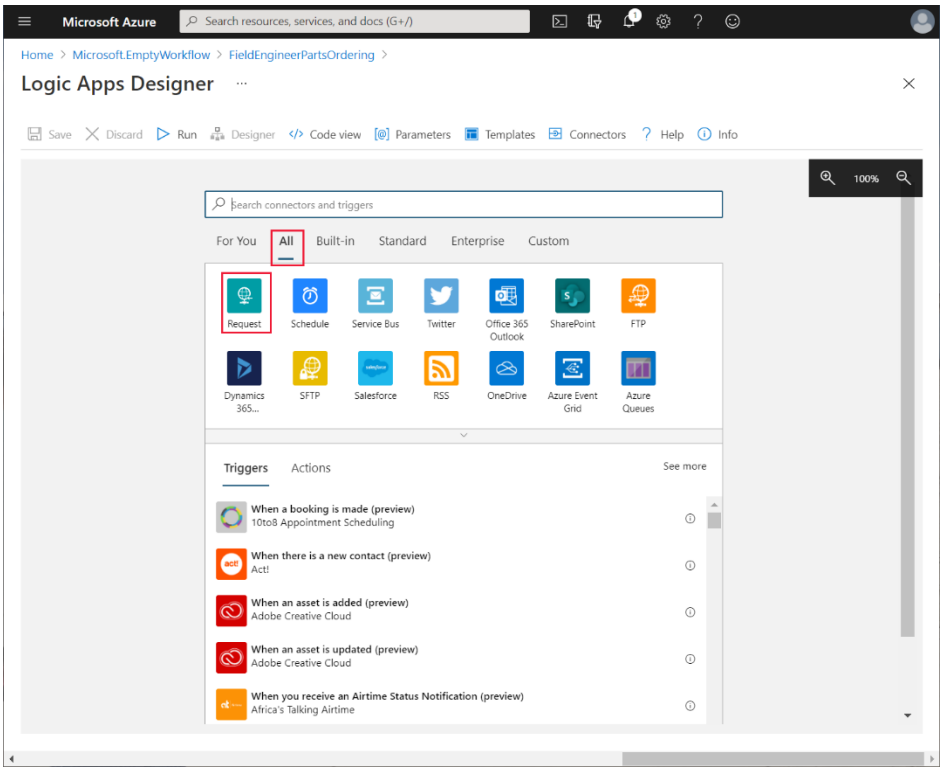
3. On the **Logic App** page, select **Create**.



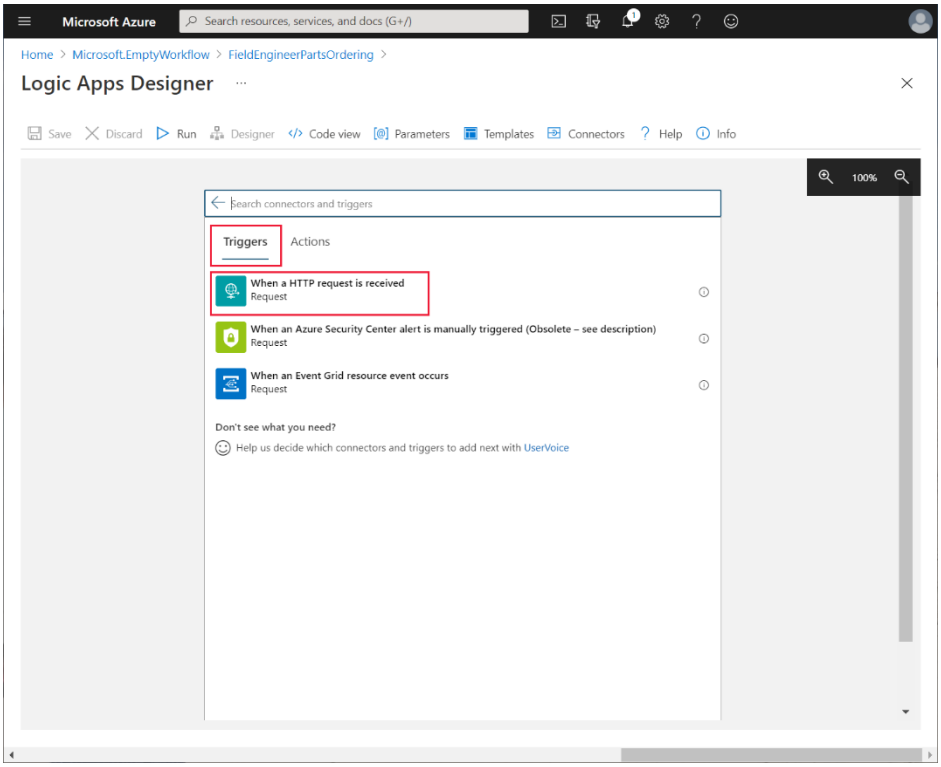
4. On the **Create a logic app** page, enter the following values, and then select **Review + create**.
    - Subscription: Select your Azure subscription
    - Resource group: **webapi_rg**
    - Logic App name: **FieldEngineerPartsOrdering**
    - Region: Select the same location you used for the Web API
    - Associate with integration service environment: Leave blank
    - Enable log analytics: Leave blank
5. On the verification page, select **Create**, and wait while the Logic App is deployed.
6. When the deployment is complete, select **Go to resource**.
7. On the **Logic Apps Designer** page, scroll down to the **Templates** section, and then select **Blank Logic App**:



8. On the **All** tab, in the **Search connectors and triggers** text box, select **Request**:

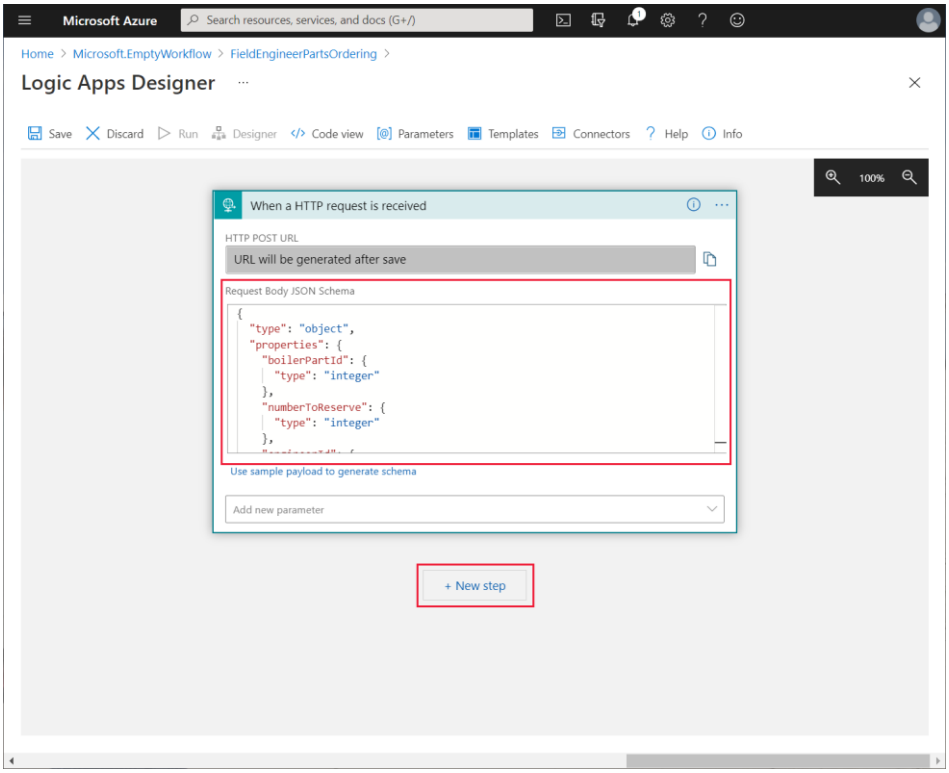


9. On the **Triggers** tab, select **When a HTTP request is received**:



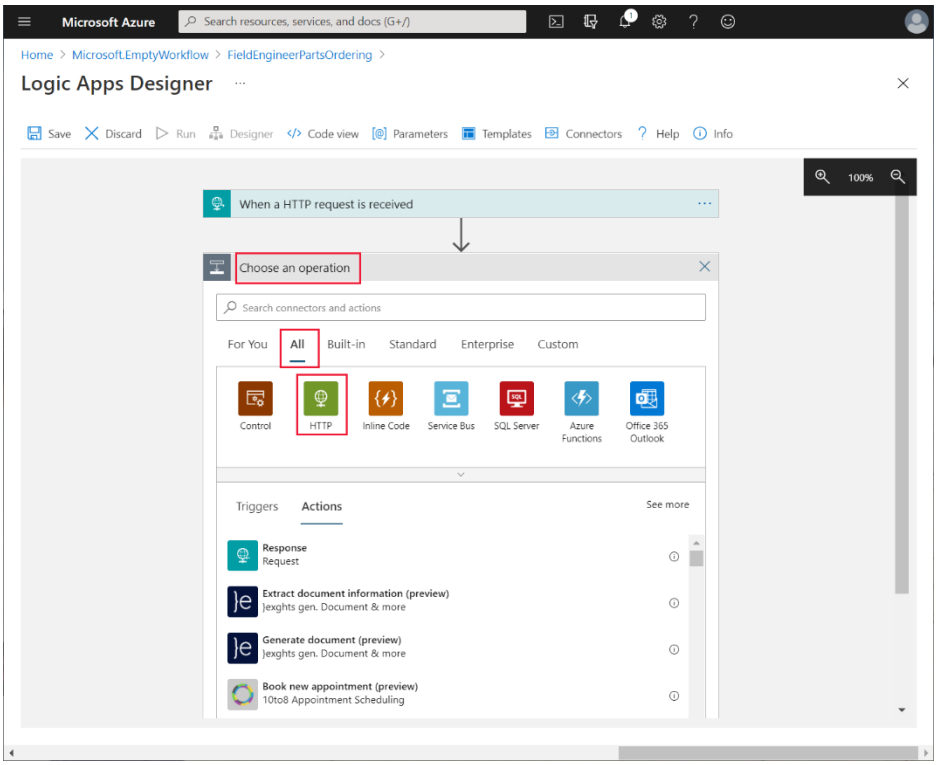
10. In the **Request Body JSON Schema** box, enter the following schema, and then select **+ New step**:

```
{
    "type": "object",
    "properties": {
        "boilerPartId": {
            "type": "integer"
        },
        "numberToReserve": {
            "type": "integer"
        },
        "engineerId": {
            "type": "string"
        }
    }
}
```



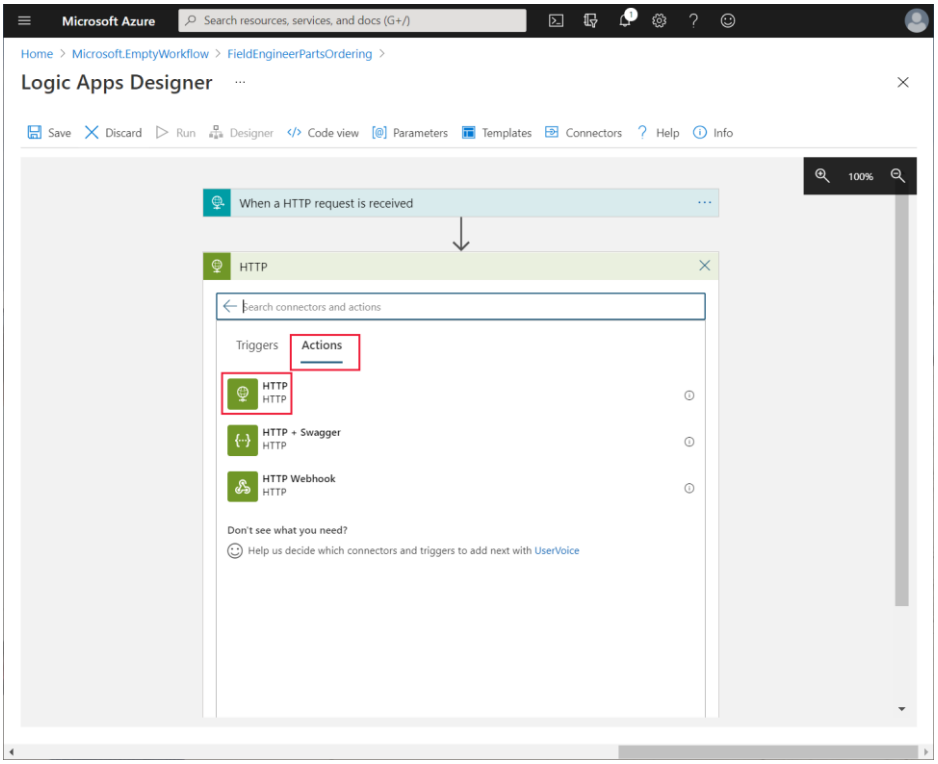
This schema defines the content of the HTTP request that the Logic App is expecting. The request body comprises the ID of a boiler part, the number of items to reserve, and the ID of the engineer making the request. The app will send this request when an engineer wants to reserve a part.

11. In the **Choose an operation** box, select **All**, and then select **HTTP**:

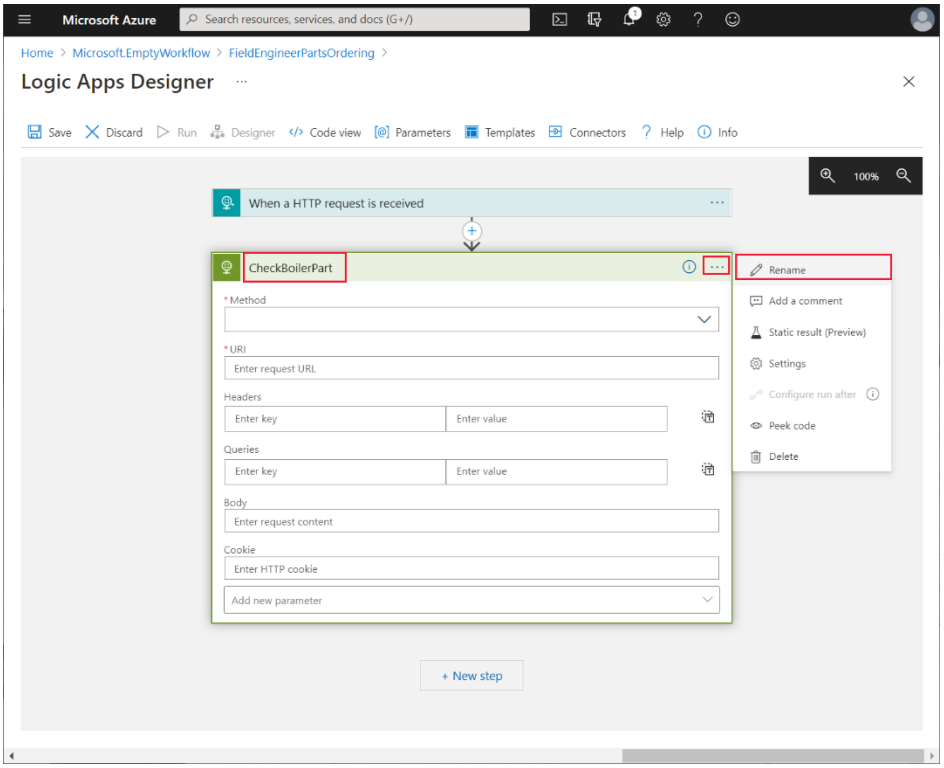


The Logic App will call the **BoilerParts{id}** operation of the Web API to retrieve information about the boiler part provided by the request from the app.

12. In the **Actions** pane, select the **HTTP** action:



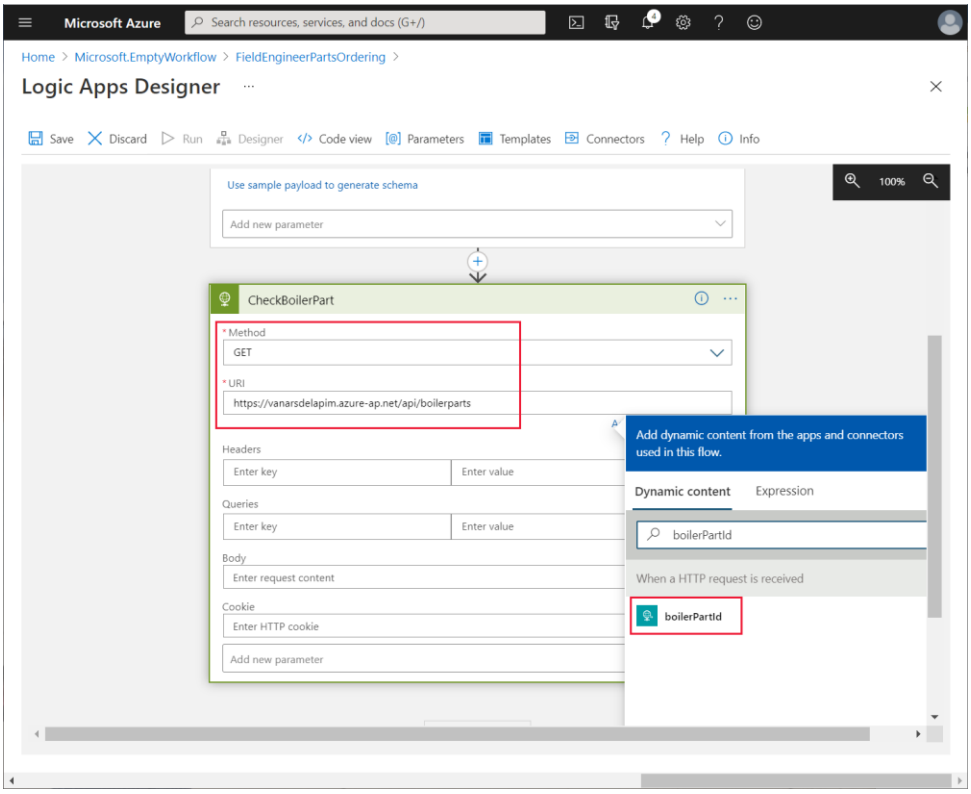
13. In the **HTTP** action box, on the ellipsis menu, select **Rename**, and change the name of the action to **CheckBoilerPart**:



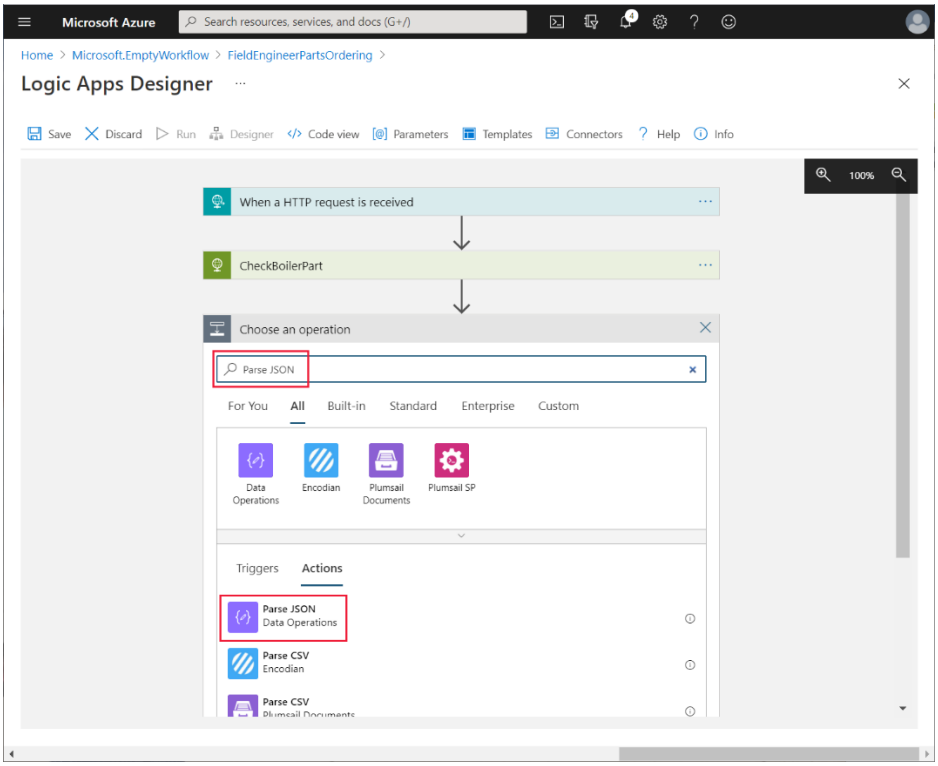
14. Set the properties of the HTTP action as follows, and then select **+ New Step**:

    - Method: **GET**

- URI: **https://*<APIM name>*.azure-api.net/api/boilerparts/**, where ***<APIM name>*** is the name of your APIM service. In the **Dynamic content** box for this URI, on the **Dynamic content** tab, select **boilerPartId**



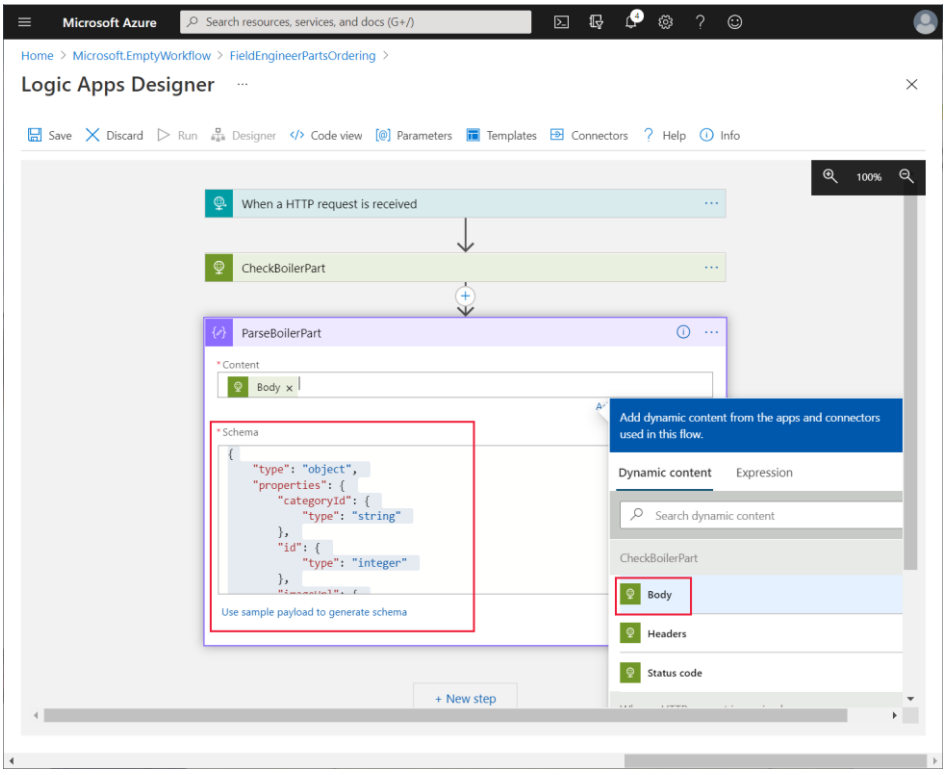
15. In the **Choose an operation** box, in the **Search connectors and actions** box, enter **Parse JSON**, and then select the **Parse JSON** action:



16. Using the ellipsis menu for the **Parse JSON** action, rename the action as **ParseBoilerPart**.

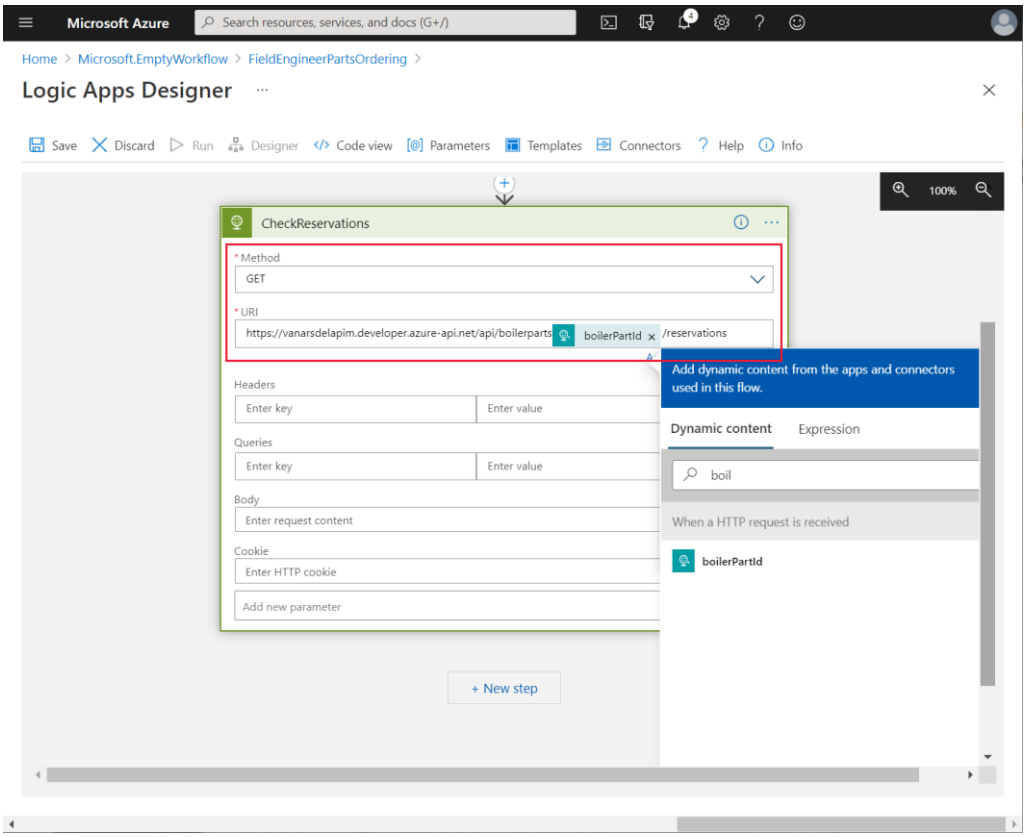
17. In the **Content** box for the **ParseBoilerPart** action, in the **Dynamic Content** box, select **Body**. In the **Schema** box, enter the following JSON schema, and then select **+ New step**:

```
{
    "type": "object",
    "properties": {
        "id": {
            "type": "integer"
        },
        "name": {
            "type": "string"
        },
        "categoryId": {
            "type": "string"
        },
        "price": {
            "type": "number"
        },
        "overview": {
            "type": "string"
        },

        "numberInStock": {
            "type": "integer"
        },
        "imageUrl": {
            "type": "string"
        },
    }
}
```



This action parses the response message returned by the **getBoilerParts/{id}** request. The response contains the details of the boiler part, including the number currently in stock.

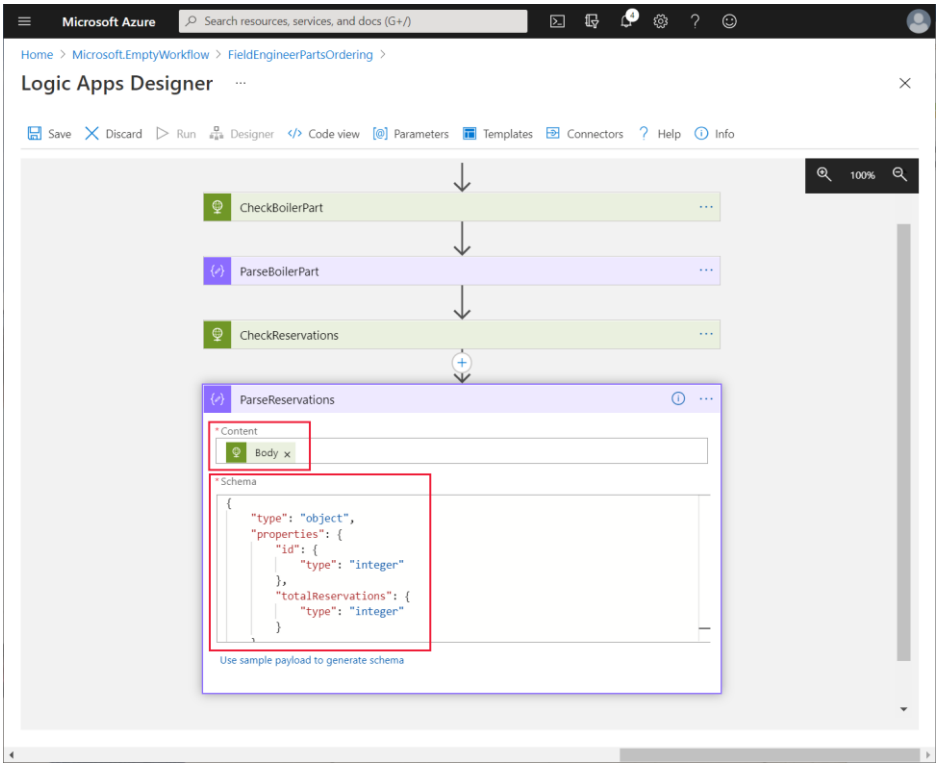
18. In the **Choose an operation** box for the new step, select the **HTTP** connector.
19. On the **Actions** tab, select the **HTTP** action.
20. Using the ellipsis menu for the operation, rename the operation as **CheckReservations**.
21. Set the following properties for this operation, and then select **+ New step**:
    - Method: **GET**
    - URI: **https://*<APIM name>*.azure-api.net/api/boilerparts/**. As before, in the **Dynamic content** box for this URI, on the **Dynamic content** tab, select **boilerPartId**. In the **URI** field, append the text **/reserved** after the **boilerPartId** placeholder



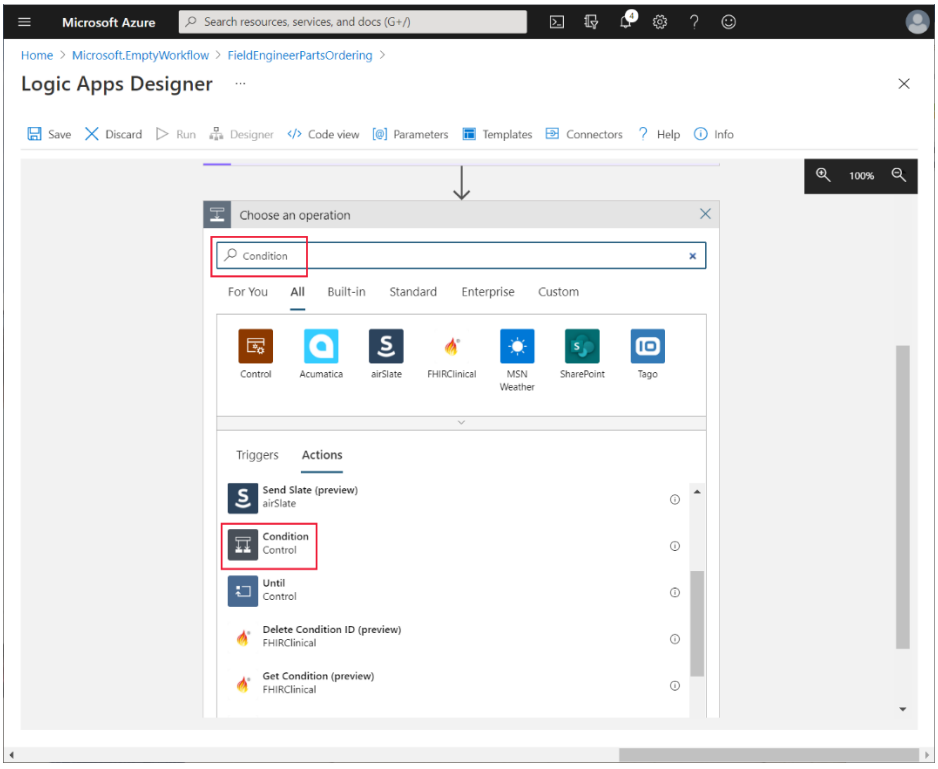
22. In the **Choose an operation** box for the new action, in the **Search connectors and actions** box, enter **Parse JSON**, and then select the **Parse JSON** action.
23. Rename the operation as **ParseReservations**.
24. Set the **Content** property to **Body**.
25. Enter the following schema, and then select **+ New step**:

```
{
    "type": "object",
    "properties": {
        "id": {
            "type": "integer"
        },
        "totalReservations": {
            "type": "integer"
        }
    }
```

```
}
```



26. In the **Choose an operation** box for the new action, in the **Search connectors and actions** box, enter **Condition**, and then select the **Condition Control** action:

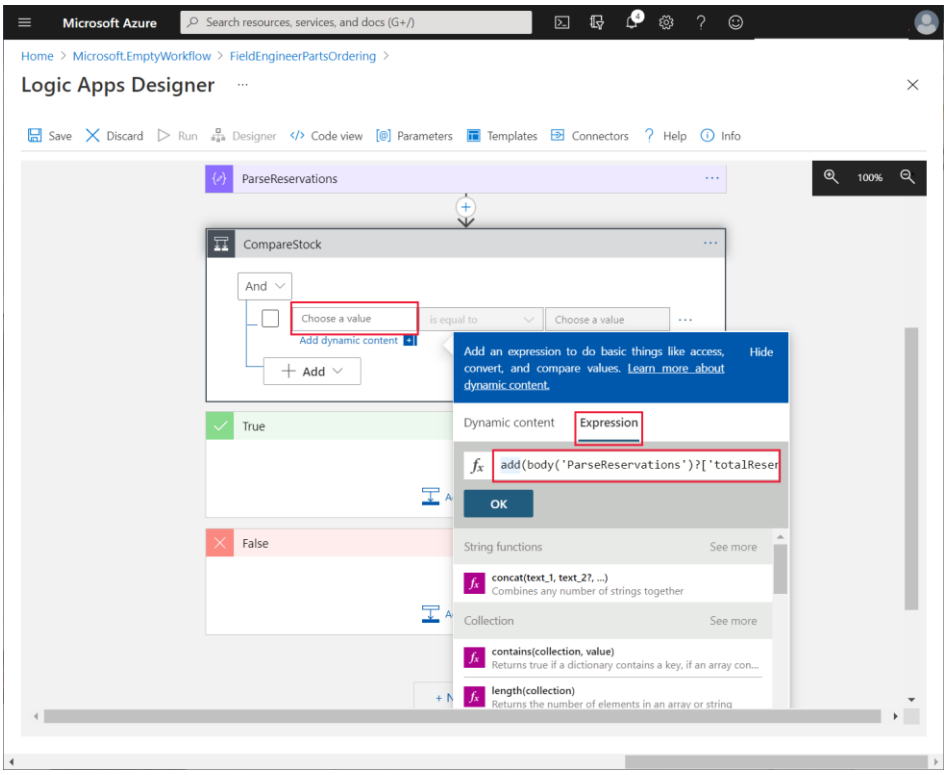


27. Rename the operation as **CompareStock**.

28. Select the **Choose a value** box. In the **Add dynamic content** box, on the **Expression** tab, enter the following expression, and then select **OK**:

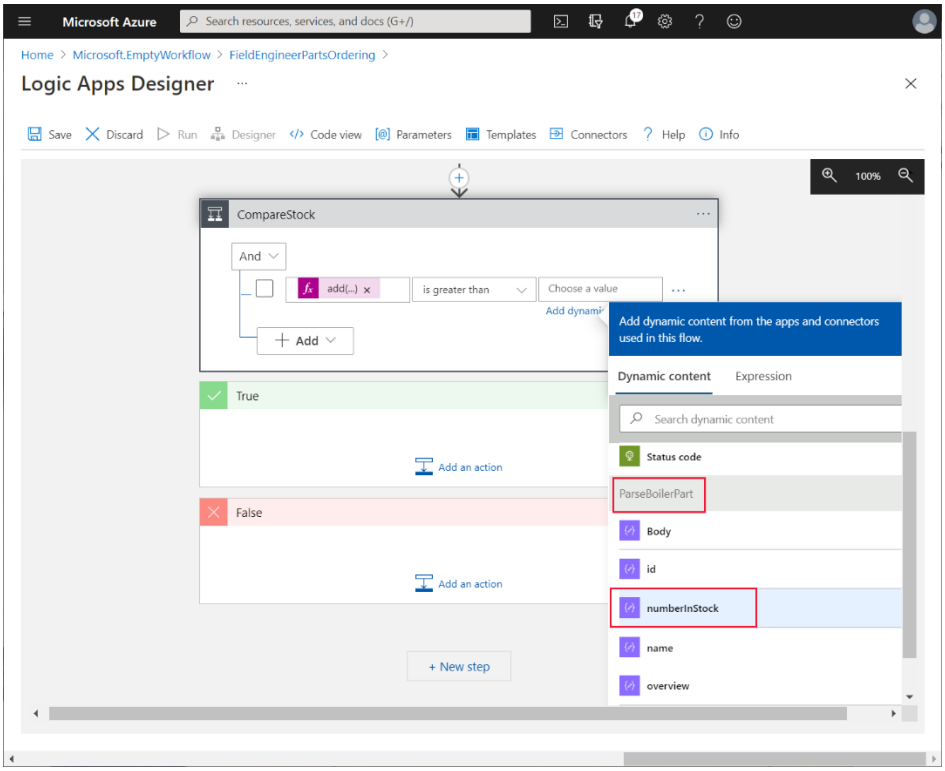
```
add(body('ParseReservations')?['totalReservations'],triggerBody()?['numberToReserve'])
```

This expression calculates the sum of the number of items of the specified boiler part that are currently reserved, and the number requested by the engineer.



29. In the condition drop-down list box, select **is greater than**.

30. In the remaining **Choose a value** box, in the **Dynamic content** box, on the **Dynamic content** tab, under **ParseBoilerPart**, select **numberInStock**:



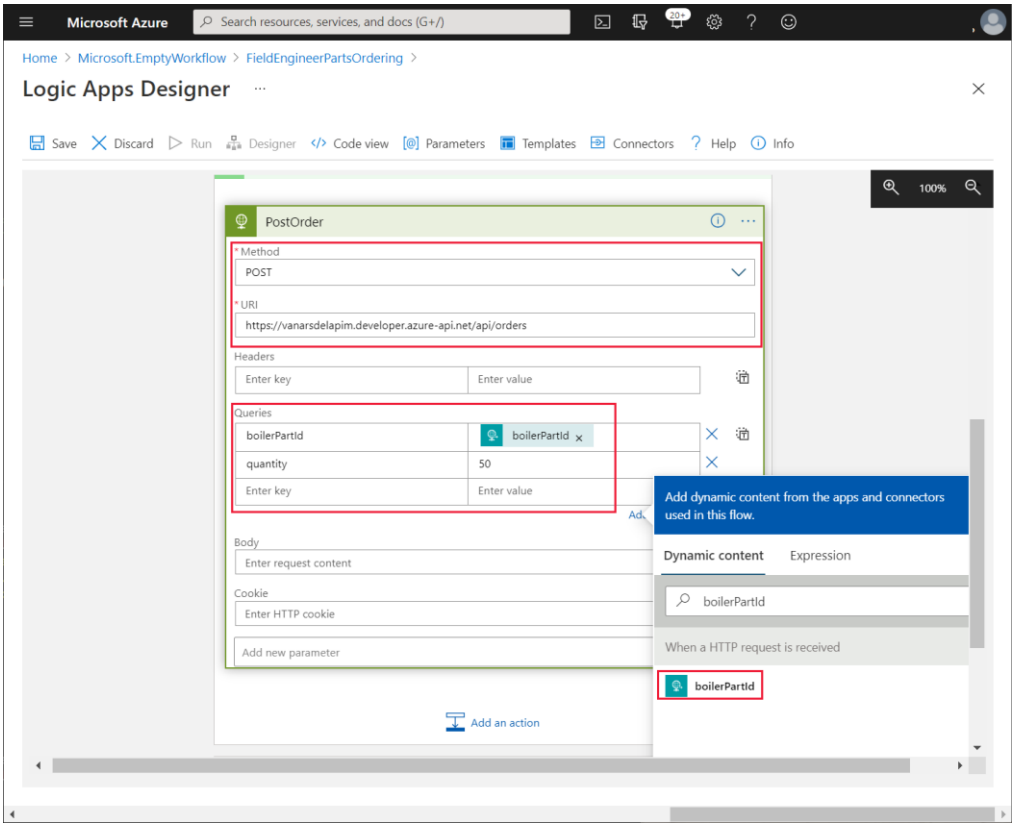
31. If the number of items required plus the number reserved is greater than the number in stock, then the app needs to place an order to replenish the inventory. In the **True** branch of the **CompareStock** action, select **Add an action**.

32. On the **All** tab for the new operation, select **HTTP**, and then select the **HTTP** action.

33. Rename the operation as **PostOrder**.

34. Set the following properties for the **PostOrder** operation:

    - Method: **POST**
    - URI: **https://*<APIM name>*.azure-api.net/api/orders**
    - In the **Queries** table, in the first row, enter the key **boilerPartId**. For the value in the **Add dynamic content** box, on the **Dynamic content** tab, select **boilerPartId**
    - In the second row of the **Queries** table, enter the key **quantity**. In the value field, enter **50**:

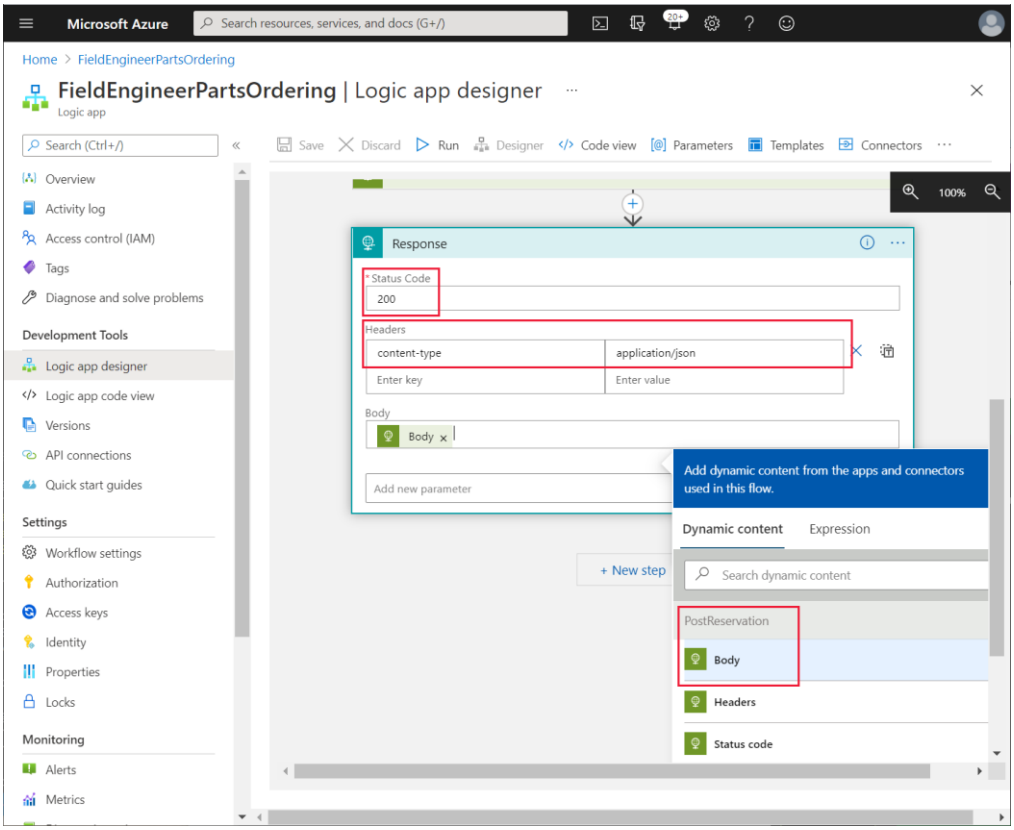


The Logic App will automatically order 50 items of the specified part when stock is running low.

> **Note**: The Logic App assumes that the engineer will nor actually attempt to reserve more than 50 items of a specified part in a single request!

35. Leave the **False** branch of the **CompareStock** action empty.
36. Below the **CompareStock** action, select **+ New step**.
37. On the **All** tab for the new operation, select **HTTP**, and then select the **HTTP** action.
38. Rename the operation as **PostReservation**.
39. Set the following properties for the **PostReservation** operation:
    - Method: **POST**
    - URI: **https://*<APIM name>*.azure-api.net/api/reservations**
    - In the **Queries** table, in the first row, enter the key **boilerPartId**. For the value in the **Add dynamic content** box, on the **Dynamic content** tab, select **boilerPartId**.
    - In the second row, enter the key **engineerId**. For the value in the **Add dynamic content** box, on the **Dynamic content** tab, select **engineerId**
    - In the third row, enter the key **quantityToReserve**. For the value in the **Add dynamic content** box, on the **Dynamic content** tab, select **numberToReserve**
40. Select **+ New Step**. In the **Choose an operation** box, search for and select the **Response** action.
41. Set the following properties for the **Response** action:
    - Status Code: **200**

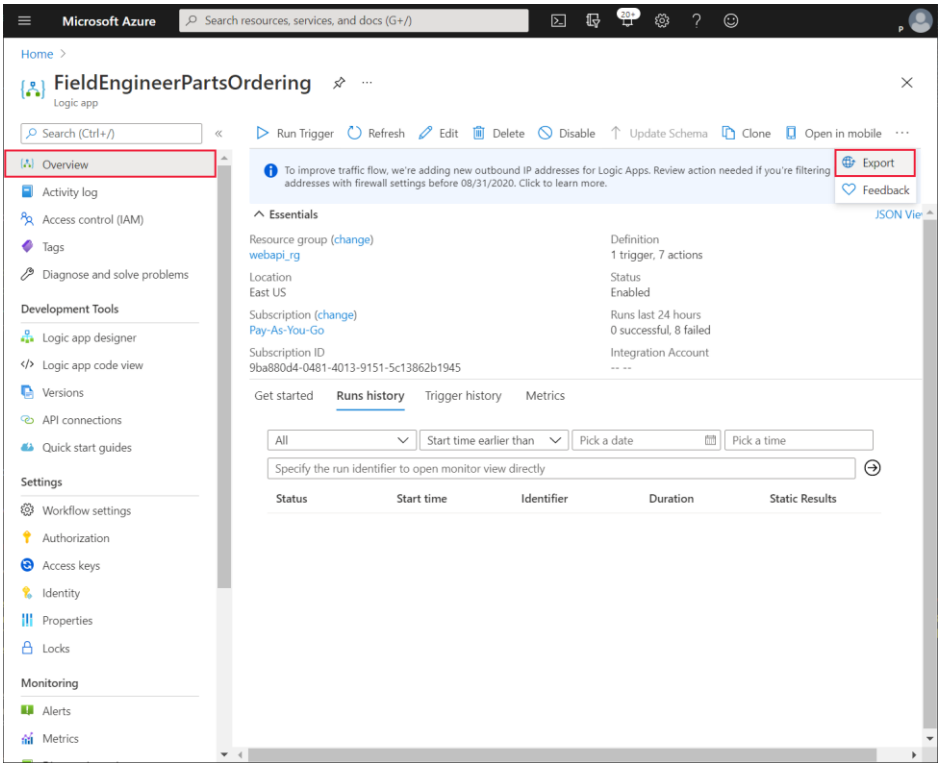
- Headers: Key – **content-type**, Value – **application/json**
- Body: In the **Dynamic content** box, select the **Body** element from the **PostReservation** request. This is the body returned when the reservation is made:



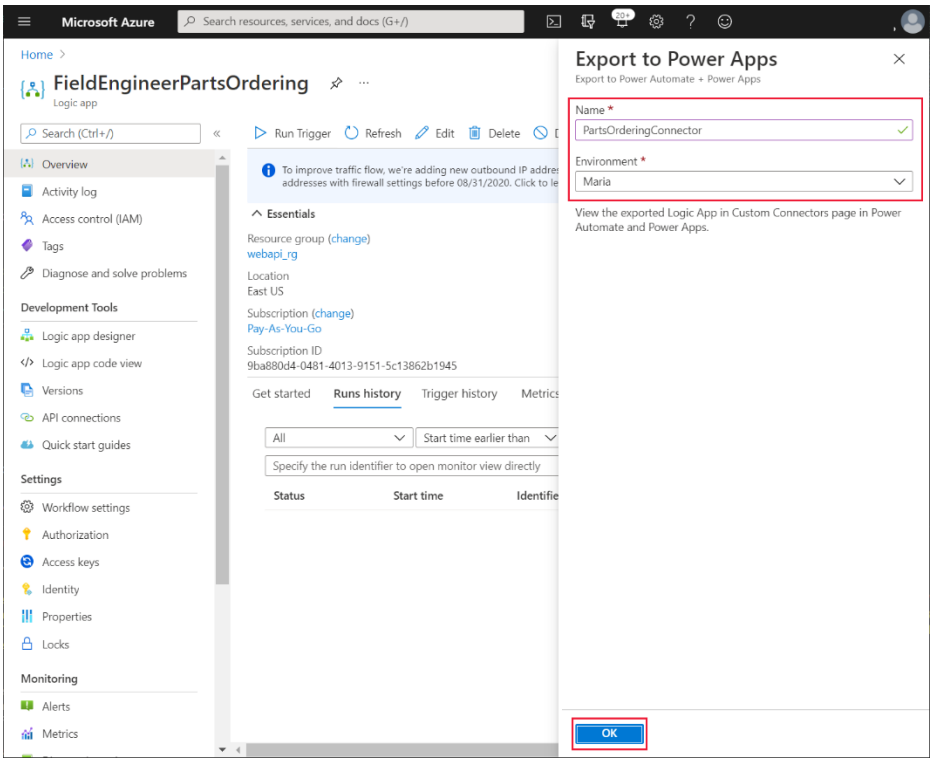
42. In the top left of the **Logic Apps Designer** page, select **Save**. Verify that the Logic App saves without any errors.

To create the custom connector that the app can use to trigger the Logic App, Kiana performs the following steps while still in the Azure portal:

1. On the **Overview** page for the Logic App, select **Export**.

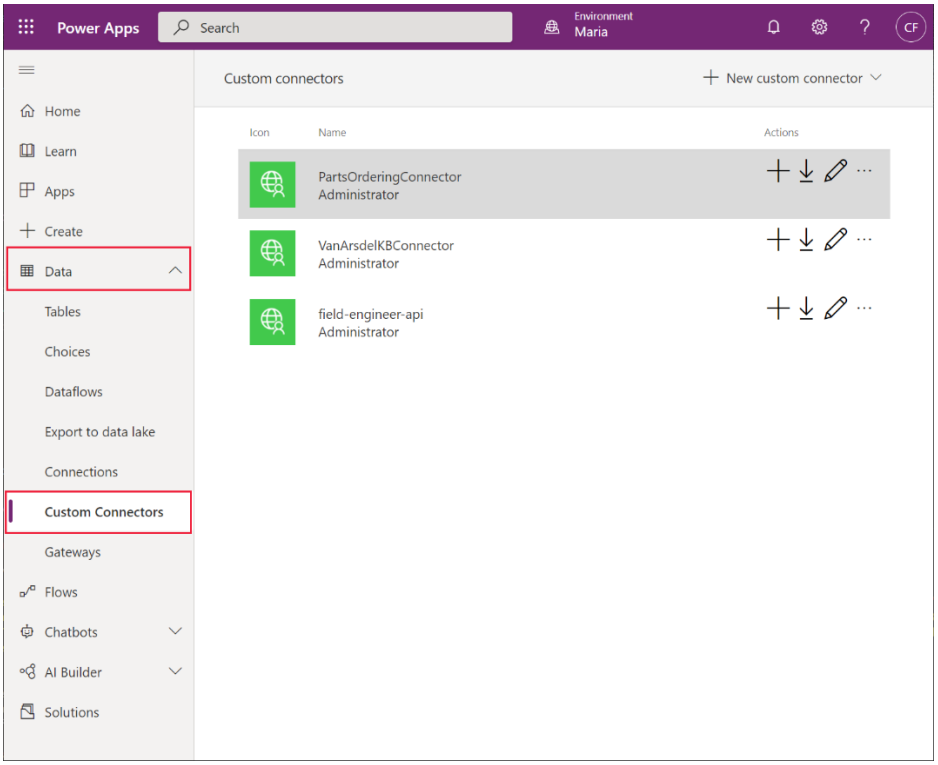


2. In the **Export to Power Apps** pane, name the connector **PartsOrderingConnector**, select your Power Apps environment**,** and then select **OK**.



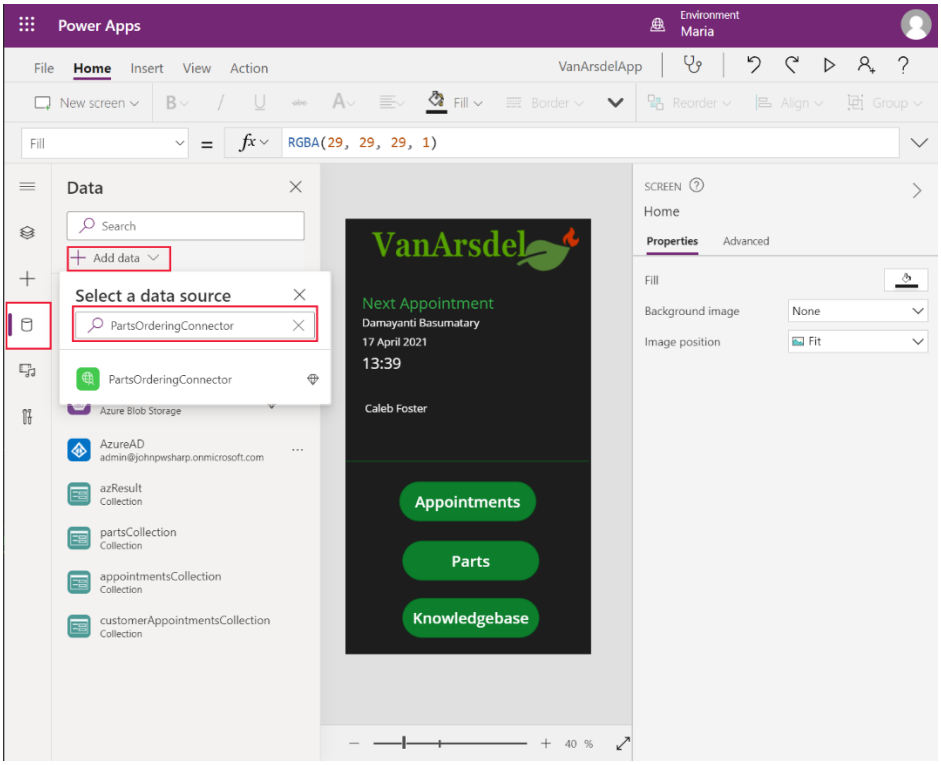
3. Sign in to Power Apps Studio at [http://make.powerapps.com.](http://make.powerapps.com)

4. In your environment, under **Data**, select **Custom Connectors** and verify that the **PartsOrderingConnector** is listed:



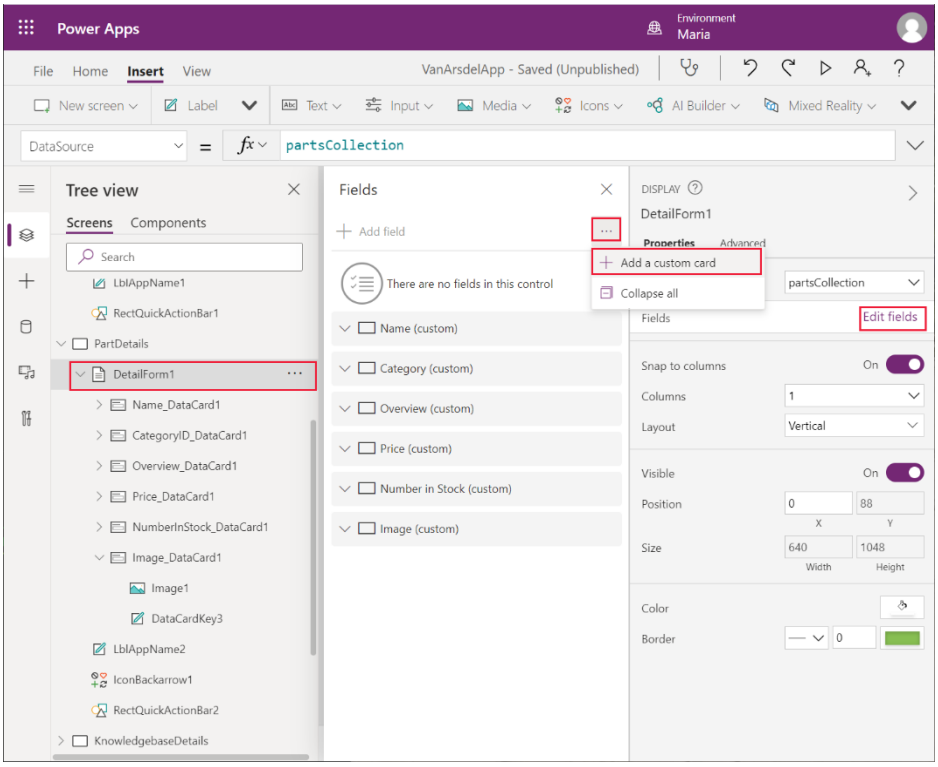
Maria can now modify the VanArsdel app to enable a technician to order parts while attending a customer site. She adds an **Order** button to the **PartDetails** screen.

1. Sign in to Power Apps Studio at http://make.powerapps.com (if not already signed in).
2. Under **Apps**, select the **VanArsdelApp** app. On the ellipsis menu for the app, select **Edit**.
3. In the **Data** pane, select **Add data**, search for the **PartsOrderingConnector** connector, and add a new connection using this connector:

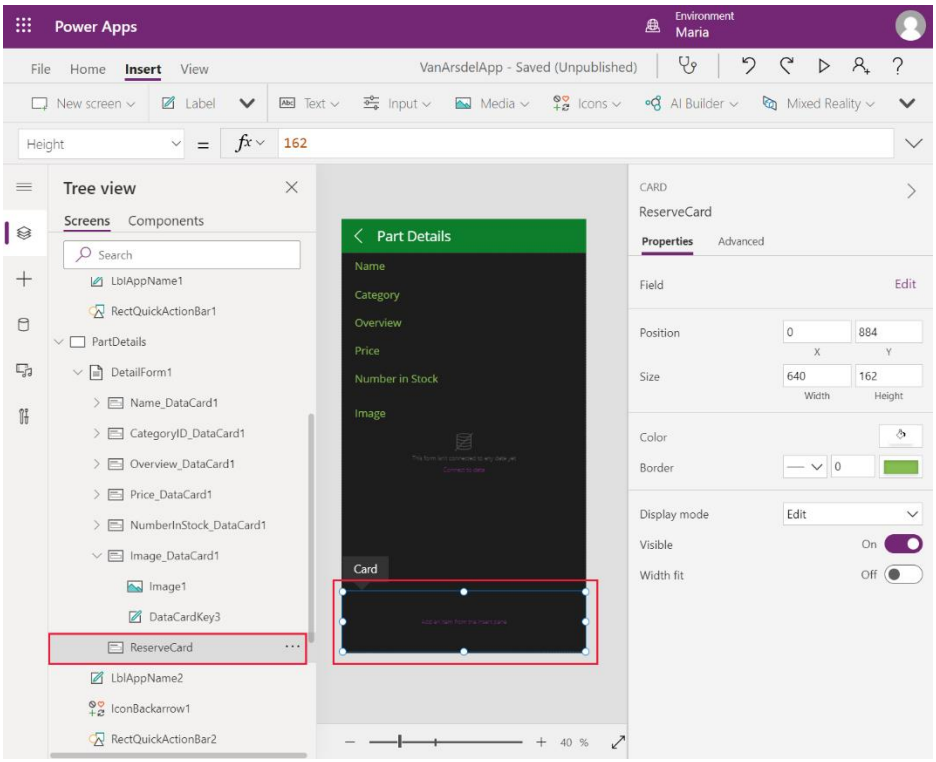


4. In the **Tree view** pane, expand the **PartDetails** screen, and then expand the **DetailForm1** form.

5. In the **Properties** pane on the right, select **Edit fields**. In the **Fields** pane, on the ellipsis menu, select **Add a custom card**:



6. In the **Tree view** pane, rename the new card from **DataCard1** to **ReserveCard**. In the **Design view** window, resize the card so that it occupies the lower part of the screen, below the **Image_DataCard1** control:

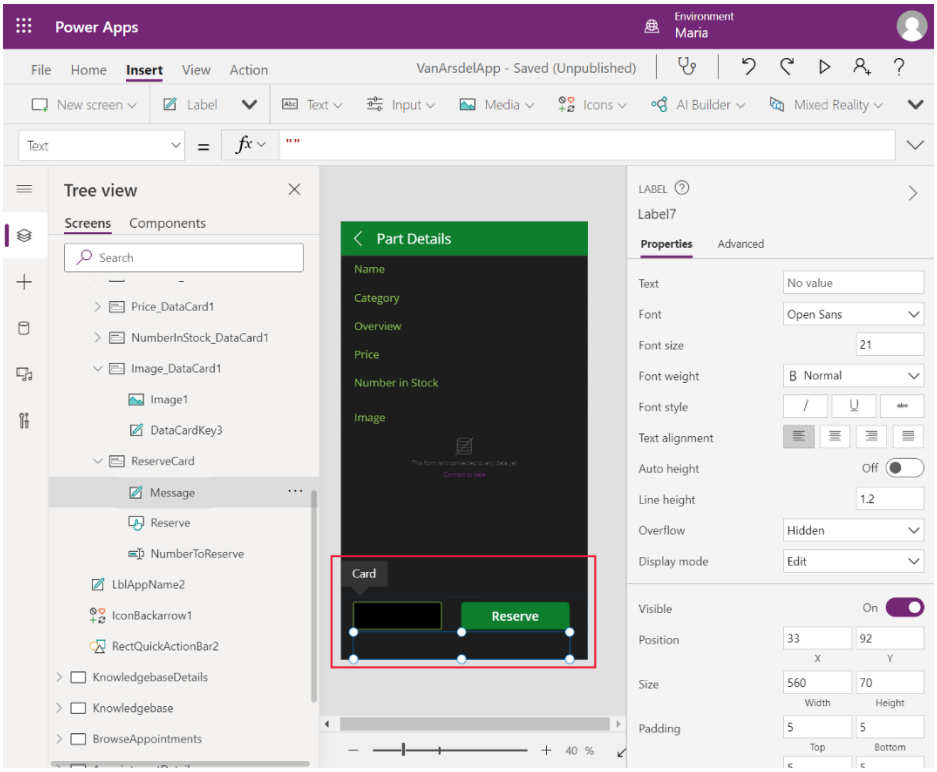


7. On the **Insert** menu, from the **Input** sub menu, add a **Text Input** control, a **Button** control, and a **Label** control to the **ReserveCard** control.

8. Resize and position the controls so that they're adjacent, with the **Button** control to the right of the **Text Input** control, and the **Label** underneath the **Button** control.

9. In the **Properties** pane for the **Text Input** control, clear the **Default** property.

10. In the **Properties** pane for the **Button** control, set the **Text** property to **Reserve**.



11. Rename the **Text Input** control as **NumberToReserve**, rename the **Button** control as **Reserve**, and rename the **Label** control as **Message**.

12. In the **Properties** pane for the **Message** control, set the **Text** property to **Parts Reserved**, and set the **Visible** property to **MessageIsVisible**.

> **Note**: **MessageIsVisible** is a variable that you will initialize to **false** when the screen is displayed, but is changed to **true** if the user hits the **Reserve** button.

13. Set the **OnSelect** property for the **Reserve** button control to the following formula:

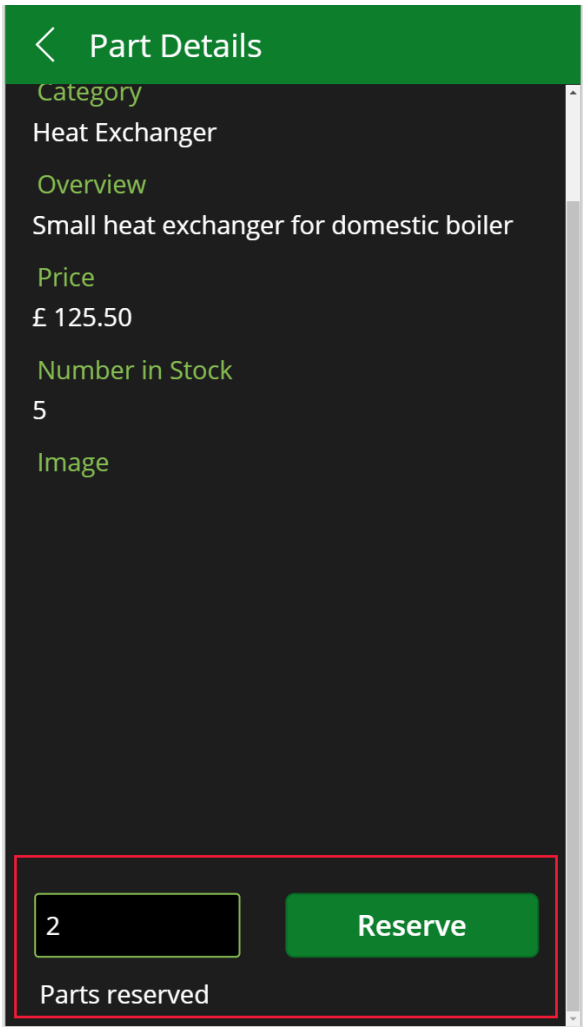
```
FieldEngineerPartsOrdering.manualinvoke({boilerPartId:ThisItem.id,
engineerId:"ab9f4790-05f2-4cc3-9f01-8dfa7d848179",
numberToReserve:NumberToReserve.Text});

Set(MessageIsVisible, true);
```

> **Note:** This formula uses a hard-coded engineer ID to represent the technician currently running the app. Chapter 8 describes how to retrieve the ID for the logged-on user.
>
> Additionally, the app performs no error checking; it assumes that the request to reserve parts always succeeds. For more information on error handling, read **Errors function in Power Apps** at https://docs.microsoft.com/powerapps/maker/canvas-apps/functions/function-errors.

14. Set the **OnVisible** property for the **PartDetails** screen to **Set(MessageIsVisible, false)**.

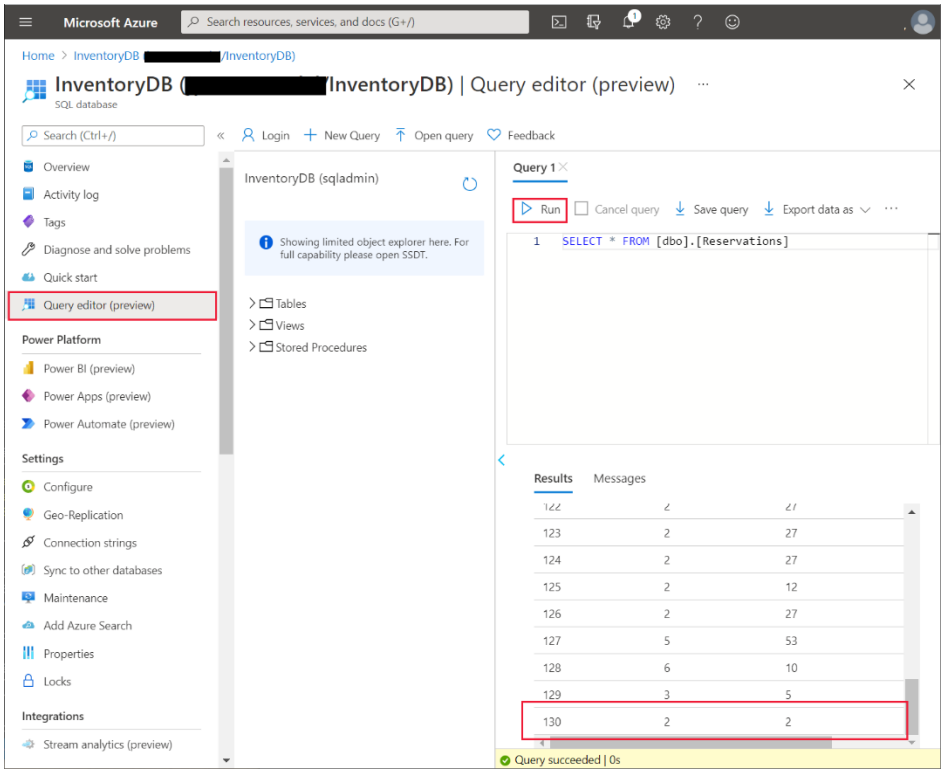
To test the app:

1. In the **Tree view** pane, select the **Home** screen.
2. Press **F5** to preview the app.
3. On the **Home** screen, select **Parts**.
4. In the browse screen, select any part.
5. On the **Part Details** screen, scroll down to the reservations section, enter a positive integer value, and then select **Reserve**. Verify that the **Parts reserved** message appears:



6. Close the preview window and return to Power Apps Studio.
7. In the Azure portal, go to the page for the **InventoryDB** Azure SQL Database.
8. Select the **Query editor**, and sign in as **sqladmin** with your password.
9. In the **Query 1** pane, enter the following query, and then select Run. Verify that the reservation you made in the VanArsdel app appears:

```
SELECT * FROM [dbo].[Reservations]
```



CHAPTER 8: PROTECTING AND DEPLOYING THE APP

The app is now functionally complete, but Preeti and Kiana want to ensure that the solution is safe to deploy, and that they have a mechanism for maintaining it as requirements change in the future.

PROTECTING THE APP AND RESOURCES

When you first sign in to Power Apps, you're required to authenticate yourself, typically by providing your email address and password. Office 365 utilizes its own Azure AD domain; each organization has its own domain. Your credentials are checked against your organization's domain for Office 365. Power Apps can only access the Office 365 resources to which you've been granted the appropriate authority. Authorization is managed by your Office 365 Administrator (Preeti in the VanArsdel scenario). For more information, read **Securing the app and data** at https://aka.ms/AAbvtkm.

The Azure resources that an app accesses are also subject to authorization. Services such as Azure Storage require an application to provide an access key. Additionally, many services can be protected through role-based access control (RBAC), which describes the operations that individual users and groups can perform. The IT Operations Manager (Preeti, again) can set the authorization policy that defines which accounts and machines can connect to services such as Azure SQL Database, Azure Blob Storage, Azure API Management, and Azure App Services. Some services also enable you to restrict the endpoints from which authenticated users can request access. For example, you can configure a firewall for Azure SQL Database to deny access to requests emanating from unexpected IP addresses.

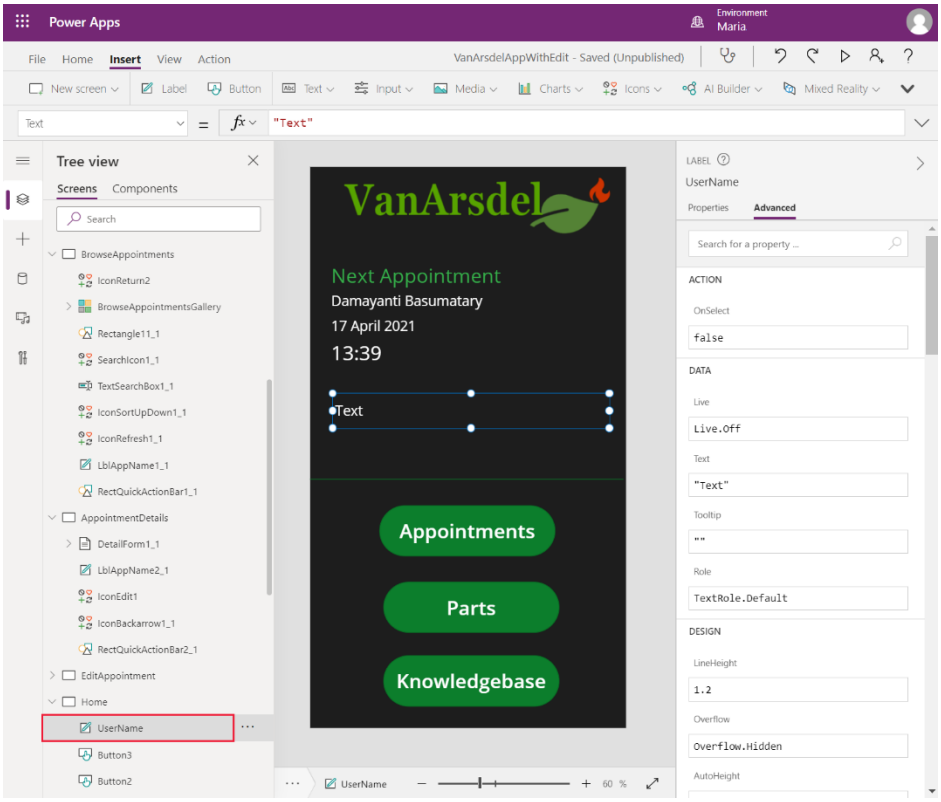
Azure helps to protect data in-flight by using transport layer security to encrypt it. This feature is vital for ensuring the integrity and privacy for any distributed system that transmits data over a network such as the public internet. In the case of VanArsdel, technicians will be running the app on mobile devices, utilizing roaming network connections that are outside the organization's control. Preeti is keen to ensure that unauthorized users cannot view or compromise sensitive data.

Data at rest, in storage accounts, databases, and other services in Azure, can also be encrypted. This provides an additional layer of privacy should the datacenter housing this information be breached. For a full list of the security features provided by Azure, read *Introduction to Azure security* at https://aka.ms/AAbvtkn.

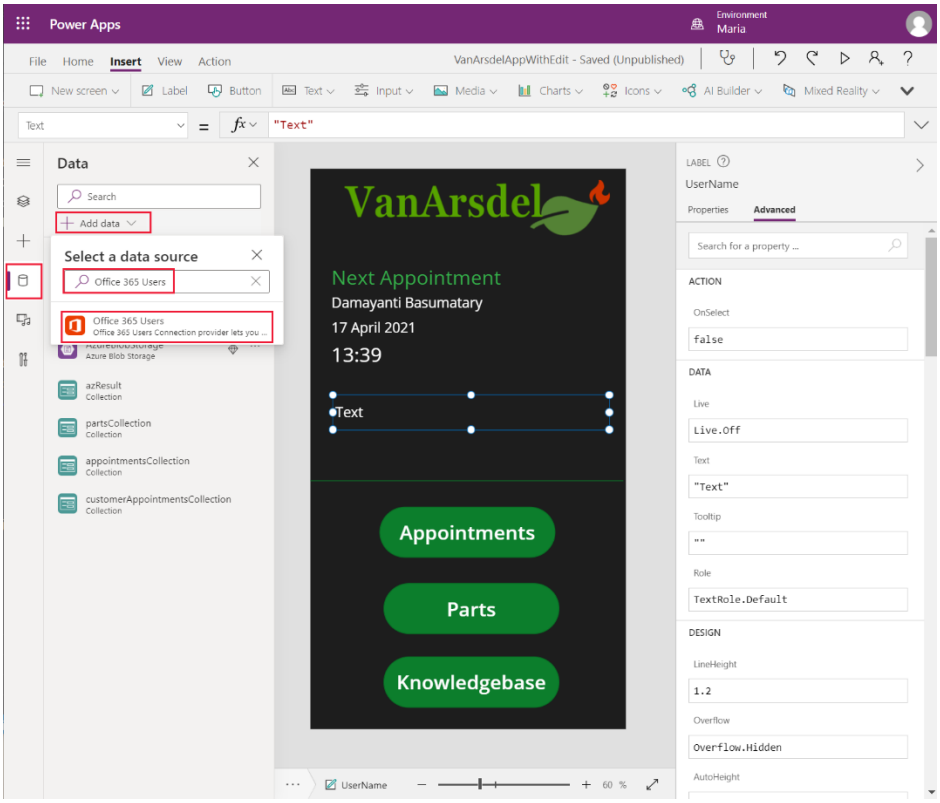
PERSONALIZING THE APP

When someone runs an app, it can retrieve information about the user from the Office365 environment. This information can be used to personalize the app. For example, currently the app that Maria and Kiana have developed doesn't distinguish between the different users; they all have access to the same data. Ideally, the app should be personalized to display the information most relevant to the engineer who uses it. Power Apps provides a function named **User** that enables the app to retrieve the email and full name of the current user. This app also requires the user ID (a unique Guid assigned to each user). The rationale behind this requirement is that usernames can be changed, but the ID cannot. The user ID is accessible using the features provided by the **Office365** connector. The steps below illustrate how to add this connector to the app:

1. Using Power Apps Studio, in the **Tree view** pane, select the **Home** screen.
2. On the **Insert** menu, from the **Text** drop-down list, add a **Label** control to the screen.
3. Rename the **Label** control as **UserName**.
4. Move the control so that it appears below the details showing the next appointment:



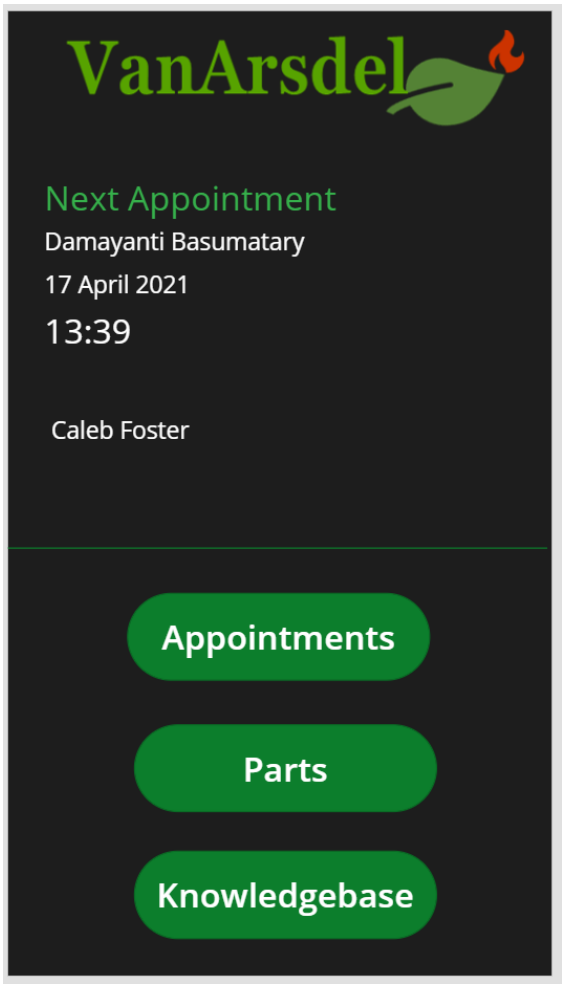
5. In the **Data** pane, select **Add data**. In the search box, enter **Office 365 Users**. Add the **Office 365 Users** connection to the app:



6. In the **Tree view** pane, select the **UserName** label, and set the **Text** property to the following formula:

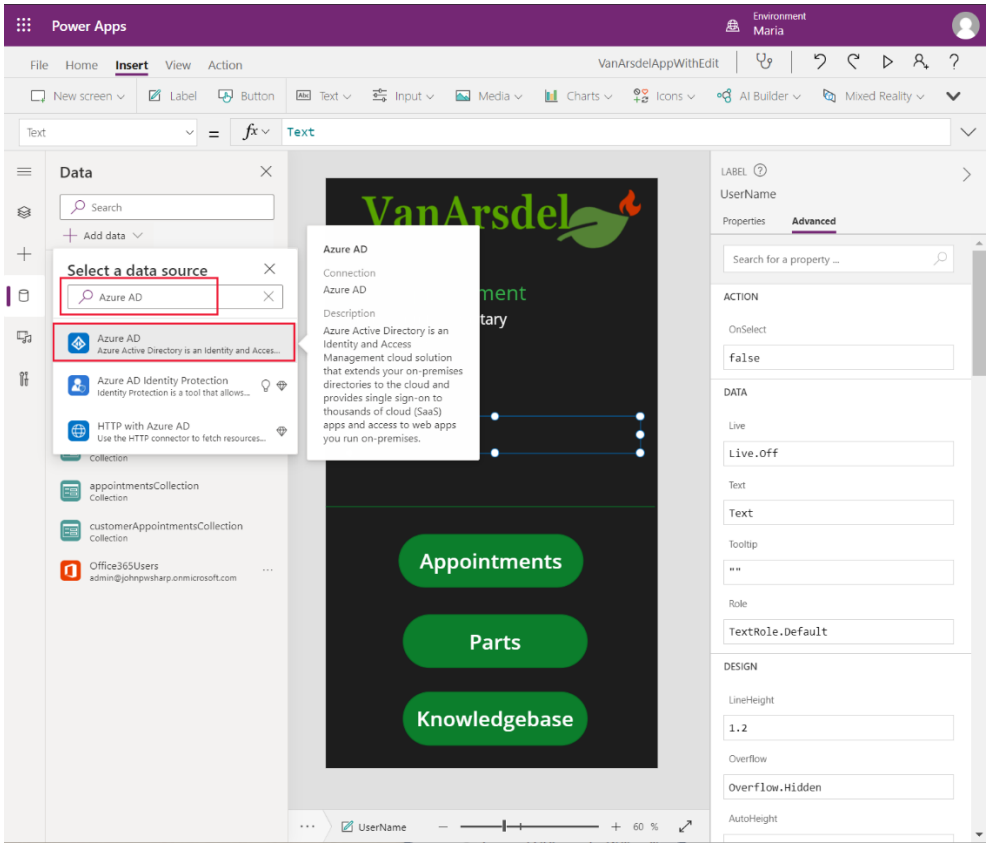
```
Office365Users.MyProfileV2().displayName
```

This formula uses the **Office365Users** connection to retrieve identity information about the current user. The **displayName** property of the **MyProfileV2** function contains the user's logged-on name.



Note: Feel free to style the **UserName** control to make it stand out more.

Office365 runs in an Azure AD domain, but you can also extend this security domain with your own Azure AD installation. If your organization authenticates users through your own Azure AD domain, you can obtain user information by using the **Azure AD** connector instead of **Office365Users**:



In this case, set the **Text** property of the **UserName** label to:

```
AzureAD.GetUser(User().Email).displayName
```

To personalize the appointments list requires calling a different Web API function in the **FieldEngineerAPI** connector. Currently, the **OnVisible** property of the **Home** screen contains the following formula:

```
ClearCollect(appointmentsCollection,
Sort(Filter(FieldEngineerAPI.getapiappointments(), DateDiff(Today(),
startDateTime) >= 0), startDateTime));
```

The Web API provides an alternative function that retrieves the appointments for a specified technician; you provide the ID of the technician as a parameter. You can use the Office365 user ID for this purpose. Update the **OnVisible** property as highlighted in bold below:

```
Set(id, Office365Users.MyProfileV2().id);

ClearCollect(appointmentsCollection,
Sort(Filter(FieldEngineerAPI.getapischeduleengineeridappointments(id),
DateDiff(Today(), startDateTime) >= 0), startDateTime);
```

If you're authenticating with Azure AD, use this formula instead:

```
Set(id, AzureAD.GetUser(User().Email).id);

ClearCollect(appointmentsCollection,
Filter(FieldEngineerAPI.getapischeduleengineeridappointments(id),
DateDiff(Today(), startDateTime) >= 0), startDateTime);
```

Note: This modification requires that the **EngineerId** column in the **Appointments** table is populated with the user's ID. This ID is a GUID, but is stored as a string in the database. The image below shows a few rows of sample data:

	Id	CustomerId	ProblemDetails	AppointmentS...	EngineerId	StartDateTime	Notes	ImageUrl
▶	1	1	Boiler won't start	2	aa76-f8bfe5060c32	2021-04-17 13:3...	Booked parts fo...	https://myapp...
	2	2	Can't change te...	3	f2f02537-f85a-4...	2021-04-19 13:3...	Needed a new ...	https://myapp...
	3	3	Radiators aren't...	2	f2f02537-f85a-4...	2021-04-20 13:3...	Bled radiators.	https://myapp...
	4	1	Boiler wont start	1	ab9f4790-05f2-...	2021-04-22 13:3...	Installed a seco...	https://myapp...
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

The **Engineers** table must also contain an Engineer with the corresponding ID:

	Id	Name	ContactNumber
	ab9f4790-05f2-...	Michelle Harris	554-1000
	cd3ed834-49fe-...	Quincy Watson	554-1002
▶	aa76-f8bfe5060c32	Caleb Foster	554-1003
	f97f7495-101d-...	Sara Perez	554-1001
*	NULL	NULL	NULL

The app is now ready to deploy and roll out.

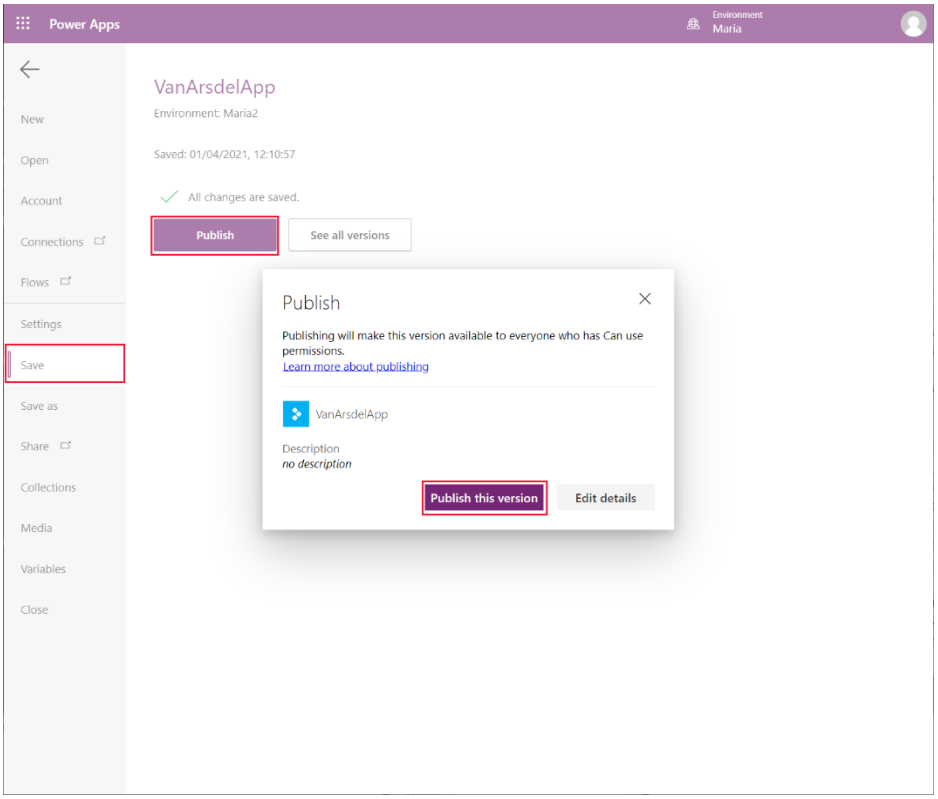
DEPLOYING THE APP

The simplest way to deploy an app is to publish it to your Office 365 domain. All users who have the **Can use** permission can run the app, either from Power Apps Studio, or by using Microsoft **Power Apps**, available in the Windows Store at https://aka.ms/AAbvtko. Power Apps can be run on mobile devices like tablets and phones as soon as they're published; users only need find the app in their devices' app store.

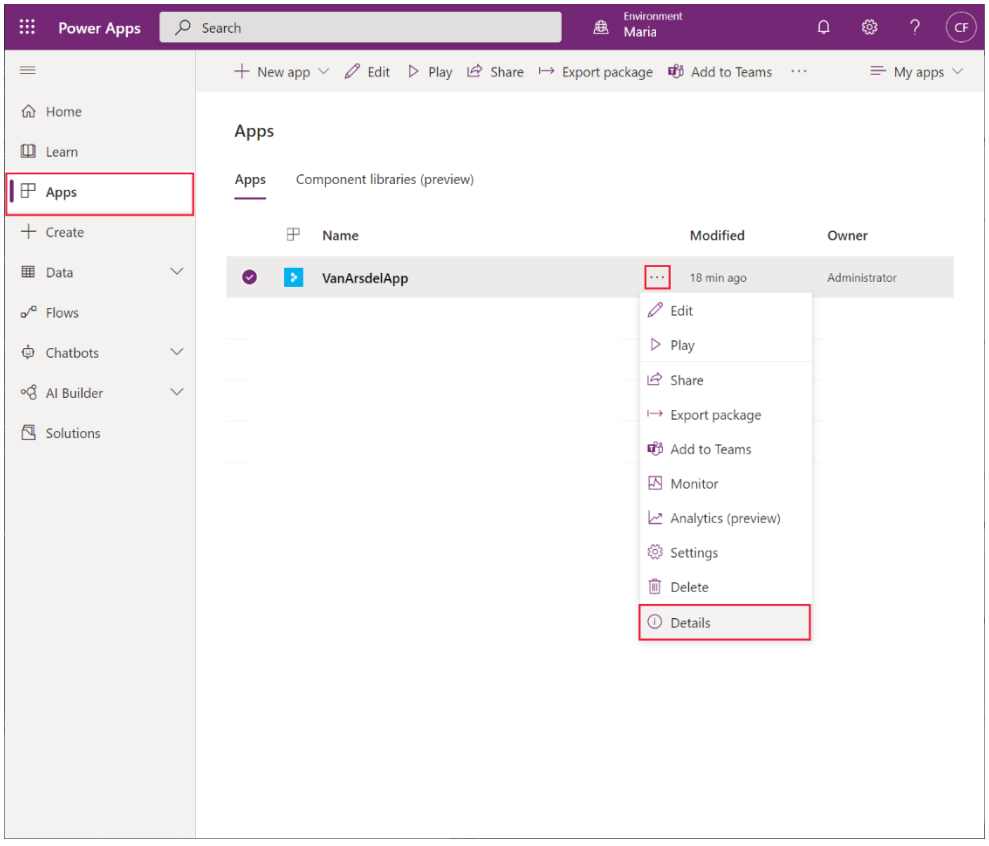
To publish an app:

1. In Power Apps Studio, on the **File** menu, select **Save**. Save the app if you've made any changes. When you save the app, the **Publish** button appears.

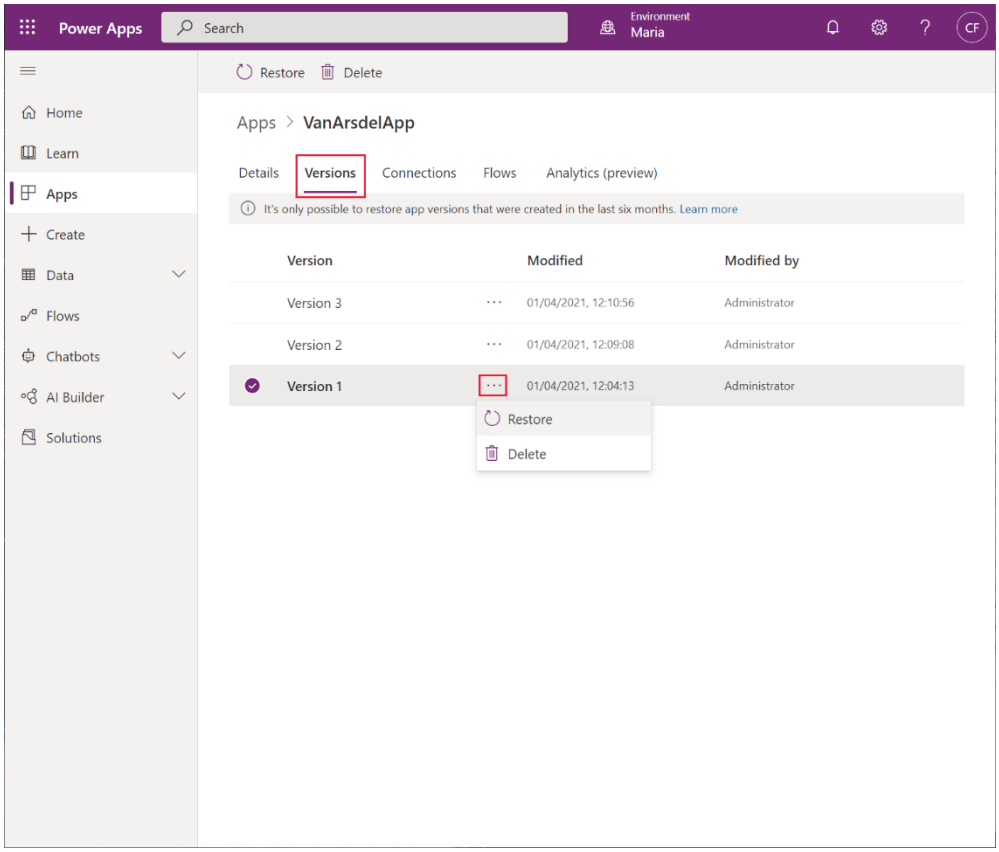
2. Select **Publish**. In the **Publish** dialog box, the **Edit details** option enables you to select settings, such as the name of the app, an icon for the app, and a description. You can also change the screen size and orientation used by the app. Select **Publish this version** to make the app available to other Power Apps users in your organization.



You can track the deployment history and app usage from the **Apps** tab on the **Administrators** page in Power Apps Studio, at https://make.powerapps.com. Select the app then, on the ellipses menu, select **Details**:



On the **Details** pane, the **Versions** tab shows the version history for the app. The options on the ellipses menu for an app enable you to restore a previous version if you need to roll back a recent publication.



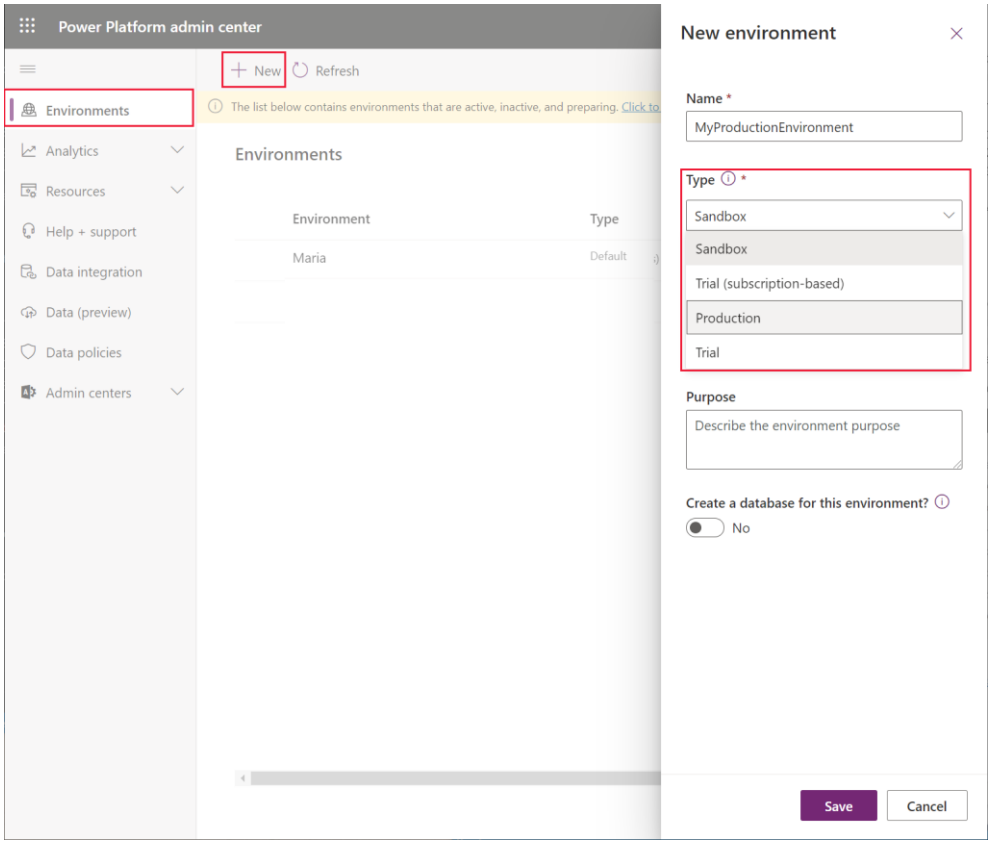
MAINTAINING THE APP

The Power Platform admin center allows you to manage the environments in which Power Apps reside. The suggested approach is to create and publish your apps through Dataverse environments. You use separate environments for development and production.

Dataverse provides four types of environments:

1. **Sandbox**: Ideal for your development.
2. **Production**: Where the app should be deployed for use.
3. **Developer**: Assets created in here can't be shared. As a single user environment, you could use it for learning and exploring the capabilities of Power Apps.
4. **Default**: An environment that's automatically created for each tenant. Microsoft doesn't recommend you use this for apps because everyone in your tenant could then access those apps.

You create environments using the Power Platform admin center at https://admin.powerplatform.microsoft.com/. On the **Environments** tab, select the **New** option in the menu bar. Specify the type of environment:



A good approach to application lifecycle management (ALM) is to start in a new sandbox environment, allowing you to safely develop and test your app in isolation from the production environment. Share and test your app as it's being developed. When your app is ready for real use, deploy it to a production environment and publish it from there. You can automate much of this process by using the Microsoft Power Platform Build Tools, described at https://aka.ms/AAbvfzw

For detailed information about ALM with Power Apps as it applies to VanArsdel, go to the **Scenario 1: Citizen Development** page at https://aka.ms/AAbvfzx

CHAPTER 9: CONCLUSIONS

Fusion development is not a strict methodology; rather, it's an approach and philosophy that encourages rapid software development.

Fusion development combines the technical and business skills of an organization's employees to design and build applications. This approach values the insights and abilities of the different members of the team. It utilizes their specific insights into the business requirements and technical challenges required to implement a solution. The synergy afforded by fusion development enables efficient communication between different team members and enables them to iterate quickly to produce a functional system.

In this guide, you've seen how the staff at VanArsdel followed a fusion development approach. They produced an app that meets the expectations of the users represented by Caleb, the technician, Maria, the inventory manager, and Malik, who schedules engineers. Preeti is also satisfied that the system is safe and maintainable. The project was completed in record time—from the initial discussions between Caleb and Maria, to the rollout to all technicians.

The VanArsdel team has now experienced how fusion development teams work and is excited to keep collaborating on future projects.