

Azure Data Factory by Example

Practical Implementation for
Data Engineers

Richard Swinbank

Apress®

Azure Data Factory by Example

Practical Implementation for
Data Engineers

Richard Swinbank

Apress®

Azure Data Factory by Example: Practical Implementation for Data Engineers

Richard Swinbank
Birmingham, UK

ISBN-13 (pbk): 978-1-4842-7028-8
<https://doi.org/10.1007/978-1-4842-7029-5>

ISBN-13 (electronic): 978-1-4842-7029-5

Copyright © 2021 by Richard Swinbank

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media LLC, 1 New York Plaza, Suite 4600, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484270288. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

To Catherine, for all the love and support.

Table of Contents

About the Author	xv
About the Technical Reviewer	xvii
Acknowledgments	xix
Introduction	xxi
Chapter 1: Creating an Azure Data Factory Instance.....	1
Get Started in Azure	2
Create a Free Azure Account	2
Explore the Azure Portal	2
Create a Resource Group	4
Create an Azure Data Factory	7
Explore the Azure Data Factory User Experience.....	9
Navigation Header Bar.....	10
Navigation Sidebar	11
Link to a Git Repository.....	12
Create a Git Repository in Azure Repos	13
Link the Data Factory to the Git Repository.....	15
The ADF UX as a Web-Based IDE	17
Chapter Review.....	19
Key Concepts.....	20
For SSIS Developers	22

TABLE OF CONTENTS

Chapter 2: Your First Pipeline	23
Work with Azure Storage	23
Create an Azure Storage Account	23
Explore Azure Storage	26
Upload Sample Data	27
Use the Copy Data Tool	28
Explore Your Pipeline	32
Linked Services	33
Datasets	34
Pipelines	35
Activities	36
Integration Runtimes	37
Factory Resources in Git	39
Debug Your Pipeline	40
Run the Pipeline in Debug Mode	41
Inspect Execution Results	42
Chapter Review	42
Key Concepts	42
For SSIS Developers	44
Chapter 3: The Copy Data Activity	45
Prepare an Azure SQL Database	45
Create the Database	46
Create Database Objects	49
Import Structured Data into Azure SQL DB	51
Create the Basic Pipeline	51
Process Multiple Files	59
Truncate Before Load	61
Map Source and Sink Schemas	62
Create a New Source Dataset	63
Create a New Pipeline	64
Configure Schema Mapping	65

TABLE OF CONTENTS

Import Semi-structured Data into Azure SQL DB	67
Create a JSON File Dataset	67
Create the Pipeline	68
Configure Schema Mapping	68
Set the Collection Reference	69
The Effect of Schema Drift	70
Understanding Type Conversion	72
Transform JSON Files into Parquet	73
Create a New JSON Dataset.....	74
Create a Parquet Dataset.....	74
Create and Run the Transformation Pipeline	75
Performance Settings	76
Data Integration Unit.....	76
Degree of Copy Parallelism	77
Chapter Review.....	77
Key Concepts.....	78
Azure Data Factory User Experience (ADF UX)	79
For SSIS Developers	81
Chapter 4: Expressions.....	83
Explore the Expression Builder	83
Use System Variables.....	86
Enable Storage of Audit Information.....	86
Create a New Pipeline	86
Add New Source Columns	86
Run the Pipeline	87
Access Activity Run Properties	88
Create Database Objects	89
Add Stored Procedure Activity	90
Run the Pipeline	93

TABLE OF CONTENTS

Use the Lookup Activity.....	94
Create Database Objects	94
Configure the Lookup Activity.....	96
Use Breakpoints	98
Use the Lookup Value	100
Update the Stored Procedure Activity.....	100
Run the Pipeline	101
User Variables	102
Create a Variable	102
Set a Variable.....	103
Use the Variable.....	104
Array Variables	105
Concatenate Strings.....	106
Infix Operators.....	107
String Interpolation.....	107
Escaping @.....	108
Chapter Review.....	108
Key Concepts.....	108
For SSIS Developers	110
Chapter 5: Parameters.....	113
Set Up an Azure Key Vault.....	113
Create a Key Vault	114
Create a Key Vault Secret	115
Grant Access to the Key Vault.....	116
Create a Key Vault ADF Linked Service.....	118
Create a New Storage Account Linked Service	119
Use Dataset Parameters	121
Create a Parameterized Dataset.....	123
Use the Parameterized Dataset.....	124
Reuse the Parameterized Dataset	126

TABLE OF CONTENTS

Use Linked Service Parameters	127
Create a Parameterized Linked Service	127
Increase Dataset Reusability	131
Use the New Dataset	132
Why Parameterize Linked Services?	133
Use Pipeline Parameters	133
Create a Parameterized Pipeline	133
Run the Parameterized Pipeline	135
Use the Execute Pipeline Activity	137
Parallel Execution	139
Global Parameters	139
Chapter Review	140
Key Concepts	141
For SSIS Developers	142
Chapter 6: Controlling Flow	145
Create a Per-File Pipeline	145
Use Activity Dependency Conditions	147
Explore Dependency Condition Interactions	149
Understand Pipeline Outcome	152
Raise Errors	156
Use Conditional Activities	157
Divert Error Rows	157
Load Error Rows	161
Understand the Switch Activity	165
Use Iteration Activities	167
Use the Get Metadata Activity	167
Use the ForEach Activity	169
Ensure Parallelizability	172
Understand the Until Activity	175

TABLE OF CONTENTS

Chapter Review.....	176
Key Concepts.....	177
For SSIS Developers	179
Chapter 7: Data Flows	181
Build a Data Flow	181
Enable Data Flow Debugging.....	182
Add a Data Flow Transformation	184
Use the Filter Transformation	188
Use the Lookup Transformation.....	191
Use the Derived Column Transformation	194
Use the Select Transformation.....	196
Use the Sink Transformation	197
Execute the Data Flow.....	198
Maintain a Product Dimension	202
Create a Dimension Table	203
Create Supporting Datasets.....	203
Build the Product Maintenance Data Flow	204
Execute the Dimension Data Flow	210
Chapter Review.....	212
Key Concepts.....	212
For SSIS Developers	214
Chapter 8: Integration Runtimes	217
Azure Integration Runtime	217
Inspect the AutoResolveIntegrationRuntime	218
Create a New Azure Integration Runtime	219
Use the New Azure Integration Runtime.....	221
Self-Hosted Integration Runtime.....	224
Create a Shared Data Factory.....	225
Create a Self-Hosted Integration Runtime.....	225
Link to a Self-Hosted Integration Runtime	226

TABLE OF CONTENTS

Use the Self-Hosted Integration Runtime	227
Azure-SSIS Integration Runtime	231
Create an Azure-SSIS Integration Runtime.....	231
Deploy SSIS Packages to the Azure-SSIS IR.....	234
Run an SSIS Package in ADF	236
Stop the Azure-SSIS IR	237
Chapter Review.....	238
Key Concepts.....	239
For SSIS Developers	240
Chapter 9: Power Query in ADF	241
Create a Power Query Mashup	241
Explore the Power Query Editor	243
Wrangle Data	245
Run the Power Query Activity.....	248
Chapter Review.....	250
Chapter 10: Publishing to ADF.....	253
Publish to Your Factory Instance.....	254
Trigger a Pipeline from the ADF UX.....	254
Publish Factory Resources	255
Inspect Published Pipeline Run Outcome	256
Publish to Another Data Factory	257
Prepare a Production Environment.....	257
Export ARM Template from Your Development Factory	259
Import ARM Template into Your Production Factory	260
Understand Deployment Parameters.....	262
Automate Publishing to Another Factory.....	263
Create a DevOps Service Connection	264
Create an Azure DevOps Pipeline	265
Trigger an Automatic Deployment	270

TABLE OF CONTENTS

Feature Branch Workflow.....	272
Azure Data Factory Utilities	274
Publish Resources as JSON.....	275
Chapter Review.....	278
Chapter 11: Triggers	281
Use a Schedule Trigger	281
Create a Schedule Trigger	281
Reuse a Trigger.....	283
Inspect Trigger Definitions.....	284
Publish the Trigger.....	285
Monitor Trigger Runs	286
Deactivate the Trigger.....	287
Advanced Recurrence Options	288
Use an Event-Based Trigger.....	289
Register the Event Grid Resource Provider.....	290
Create an Event-Based Trigger.....	291
Cause the Trigger to Run	293
Trigger-Scope System Variables.....	295
Use a Tumbling Window Trigger	296
Prepare Data.....	296
Create a Windowed Copy Pipeline	297
Create a Tumbling Window Trigger	299
Monitor Trigger Runs	299
Advanced Features	301
Publishing Triggers Automatically	302
Triggering Pipelines Programmatically	303
Chapter Review.....	303
Key Concepts.....	304
For SSIS Developers	305

TABLE OF CONTENTS

Chapter 12: Monitoring.....	307
Generate Factory Activity	307
Inspect Factory Logs	308
Inspect Trigger Runs.....	308
Inspect Pipeline Runs.....	309
Add Metadata to the Log	311
Inspect Factory Metrics	314
Export Logs and Metrics	316
Create a Log Analytics Workspace	316
Configure Diagnostic Settings	316
Inspect Logs in Blob Storage.....	318
Use the Log Analytics Workspace	319
Query Logs	319
Use a Log Analytics Workbook.....	321
Receive Alerts	323
Configure Metric-Based Alerts	323
Configure Log-Based Alerts.....	325
Deactivate ADF Triggers	327
Chapter Review.....	327
Key Concepts.....	328
For SSIS Developers	329
Index.....	331

About the Author



Richard Swinbank is a data engineer and Microsoft Data Platform MVP. He specializes in building and automating analytics platforms using Microsoft technologies from the SQL Server stack to the Azure cloud. He is a fervent advocate of DataOps, with a technical focus on bringing automation to both analytics development and operations. An active member of the data community and keen knowledge sharer, Richard is a volunteer, organizer, speaker, blogger, open source contributor, and author. He holds a PhD in Computer Science from the University of Birmingham, UK.

About the Technical Reviewer



Paul Andrew is a Microsoft Data Platform MVP with over 15 years of experience in the industry, working as a data engineer and solution architect. Day to day, Paul is accountable for delivering data insights to international organizations where he wields the complete stack of Azure Data Platform resources. Paul leads delivery teams around the globe implementing the latest design patterns, creating architectural innovations, and defining best practices to ensure technical excellence for customers across a wide variety of sectors. Paul is passionate about technology – this is demonstrated in the community. He speaks at events and shares his knowledge gained from real-world experiences through his blog. Paul maintains the view that his job is also his hobby and doesn't ever want to take his fingers off the developer's keyboard.

Acknowledgments

While this book is about one specific service – Azure Data Factory – it is the product of years of experience working as a data engineer. I am enormously grateful to the many colleagues, past and present, from whom I continue to learn every day. I’m indebted to the wider Microsoft data platform community, a group of engaged, generous people who are unstinting in their advice and support for others working in this space.

I want to thank my technical reviewer, Paul Andrew, for innumerable conversations which have made the book many times better than it could otherwise have been. Paul is a real expert in this technology, and I’m very fortunate to have benefited from his advice. I must thank Simon Swinbank and Liz Bell for their invaluable input in cleaning up and clarifying the text. Thanks also to the editorial team at Apress, Jonathan Gennick, Jill Balzano, and Laura Berendson, without whom this book would not have been possible.

Finally, to Catherine, who has been nothing but supportive and encouraging throughout the length of this project – I thank you from the bottom of my heart.

Introduction

Azure Data Factory (ADF) is Microsoft's cloud-based ETL service for scale-out serverless data movement, integration, and transformation. The earliest version of the service went into public preview in 2014 and was superseded by version 2 in 2018. ADF V2 contains so many improvements over V1 that it is all but a different product, and it is on ADF V2 that this book is exclusively focused.

From the outset, a major strength of ADF has been its ability to interface with many types of data source and to orchestrate data movement between them. Data transformation was at first delegated to external compute services such as HDInsight or Stream Analytics, but with the introduction of Mapping Data Flows in 2019 (now simply “Data Flows”), it became possible to implement advanced data transformation activities natively in ADF.

ADF can interact with 100 or more types of external service. The majority of these are data storage services – databases, file systems, and so on – but the list of supported compute environments has also grown over time and now includes Databricks, Azure Functions, and Azure Machine Learning, among others. The object of this book is not to give you the grand tour of all of these services, each of which has its own complexities and many of which you may never use. Instead, it focuses on the rich capabilities that ADF offers to integrate data from these many sources and to transform it natively.

Azure Data Factory is evolving and growing rapidly, with new features emerging with every month that passes. Inevitably, you will find places in which the ADF User Experience (ADF UX) differs from the screenshots and descriptions presented here, but the core concepts remain the same. The conceptual understanding that you gain from this book will enable you confidently to expand your knowledge of ADF in step with the development of the service.

Readers of the book will be aware of Azure Synapse pipelines, a serverless ETL service parallel to ADF inside Azure Synapse Analytics. Although this book makes no explicit reference to Synapse pipelines, many of the concepts and tools are immediately transferable. At the time of writing, Synapse pipelines are some distance from achieving feature parity with ADF – for the time being, readers using Synapse pipelines should be prepared for the absence of certain ADF features.

About You

The book is designed with the working data engineer in mind. It assumes no prior knowledge of Azure Data Factory so is suited to both new data engineers and seasoned professionals new to the ADF service. A basic working knowledge of T-SQL is expected.

If you have a background in SQL Server Integration Services (SSIS), you will find that ADF contains many familiar concepts. The “For SSIS developers” notes inserted at various points in the text are to help you leverage your existing knowledge or to indicate where you should be aware of differences from SSIS.

How to Use This Book

The book uses a series of tutorials to get you using ADF right away, introducing and reinforcing concepts naturally as you encounter them. To undertake exercises, you will need access to an Azure subscription and a web browser supported by the ADF UX – browsers supported currently are Microsoft Edge and Google Chrome. Choose a subscription in which you have sufficient permissions to create and manage the various Azure resources you will be using. Chapter 1 includes the creation of a free Azure trial subscription, ensuring that you have the necessary access. A Windows computer is necessary for certain parts of Chapter 8.

Work through the chapters in order, as later chapters rely on both knowledge and ADF resources developed in earlier chapters. When directed to give a resource a specific name, do so, because that name may later be used to refer back to the resource. References to labels in user interface components, for example field names or page titles, are given in italics. Input values, for example for text box input or radio button selection, are given in quotes – when you are asked to enter a value given in quotes, the quotes should not be included unless you are directed to do so.

CHAPTER 1

Creating an Azure Data Factory Instance

A major responsibility of the data engineer is the development and management of *extract, transform, and load* (ETL) and other data integration workloads. Real-time integration workloads process data as it is generated – for example, a transaction being recorded at a point-of-sale terminal or a sensor measuring the temperature in a data center. In contrast, *batch* integration workloads run at intervals, usually processing data produced since the previous batch run.

Azure Data Factory (ADF) is Microsoft’s cloud-native service for managing batch data integration workloads. ADF is an example of a *serverless* cloud service – you use it to create your own ETL applications, but you don’t have to worry about infrastructure like operating systems or servers or how to manage changes in demand. Access to the service is achieved by means of a data factory *instance* (often simply called “a data factory”). The majority of this book is concerned with the authoring and management of ADF *pipelines* – data integration workload units written and executed in an ADF instance.

In order to create pipelines, you need first to have access to an ADF instance. In this chapter, you will create a new ADF instance, ready to start building pipelines in Chapter 2. To get started, you will need nothing more than an Internet connection and either the Microsoft Edge or Google Chrome web browser.

Note You may be using variations on ETL like *extract, load, and transform* (ELT) or *extract, load, transform, and load* (ELTL). ADF can be used in any of these data integration scenarios, and I use the term ETL loosely to include any of them.

Get Started in Azure

To access cloud services in Microsoft Azure, you need an Azure *subscription*. My goal is to get you up and running at zero cost – in the following sections, I step through the creation of a free Azure trial subscription that you will be able to use throughout this book, then introduce the Azure portal to interact with it.

Create a Free Azure Account

Many of the exercises in the book require elevated access permissions in Azure. You may choose to skip this section if you already have an Azure subscription that you would prefer to use, but make sure that it grants you sufficient access to create and modify resources.

1. In your web browser, go to <https://azure.microsoft.com> and sign in. If you don't already have a Microsoft online account, you will need first to create one. The Azure Data Factory User Experience (introduced later in the chapter) is only supported in Microsoft Edge or Google Chrome, so you will need to use one of those two web browsers.
2. Click the *Free account* link in the top right, and on the following page, click *Start free*.
3. Follow the four-step process to set up your account. During the account setup, you will be required to provide billing information, but your credit card will not be charged unless you upgrade to a paying subscription.

After successful account creation, a *Go to the portal* button is displayed – click it. If you don't see the button, you can browse to the portal directly using its URL: <https://portal.azure.com>.

Explore the Azure Portal

The Azure portal is where you manage all of your Azure resources. You'll use the portal regularly, so it's a good idea to bookmark this page. The portal home page looks something like Figure 1-1. I say “something like” because you may see different tools,

recommendations, links, or other messages from time to time. Three or four features are always present:

1. If you are using a capped subscription, a notification about your remaining credit pops up briefly when you first open the portal. The remaining credit is displayed in your account's local currency. The free credit included with your Azure trial subscription is time-limited to 30 days.
2. On the home page, you will find a *Create a resource* button (plus icon). This option is also available from the portal menu, accessed using the button in the top left.
3. In the top right, the email address you used to sign in is displayed.
4. Immediately below your email address is your current directory. If you are using a free trial subscription, this will say *DEFAULT DIRECTORY*.

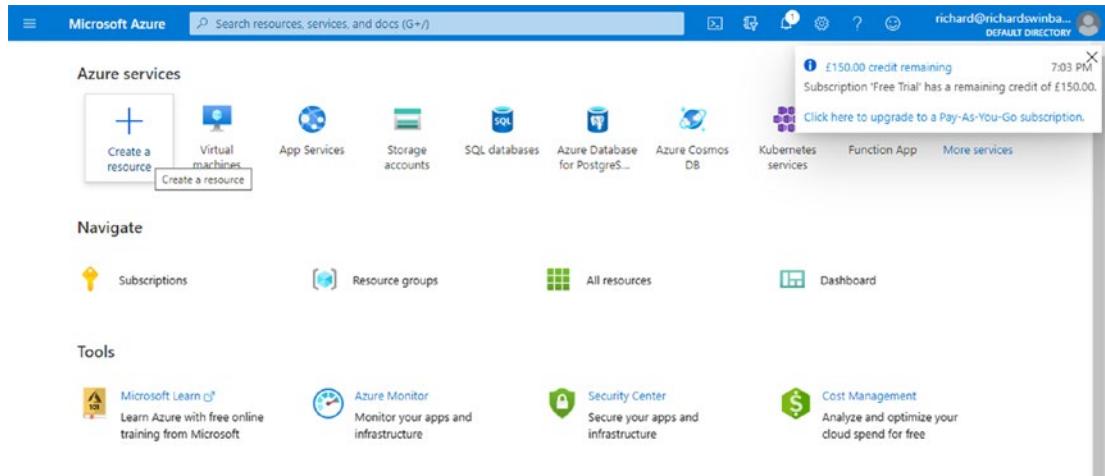


Figure 1-1. Azure portal home page

Your directory, commonly called a *tenant*, is an instance of Azure Active Directory (AAD). “Default Directory” is the default name of a new tenant. If you are already using Azure in your job, you will probably be using a tenant that represents your company or organization – often, all of an organization’s Azure resources and users are defined in the one same tenant.

A tenant contains one or more *subscriptions*. A subscription identifies a means of payment for Azure services – the cost of using any Azure resource is billed to the subscription with which it is associated. An Azure trial subscription includes an amount of time-limited free credit, and if you want to spend more, you can do so by upgrading to a paying subscription. Your organization might have multiple subscriptions, perhaps identifying separate budget holders responsible for paying for different resources.

Signing up for a trial Azure subscription creates a number of things, including

- An Azure tenant
- Your Azure user account, with administrator-level AAD permissions inside the tenant
- An Azure subscription in the tenant with some time-limited free credit for you to use

Create a Resource Group

Instances of Azure services are referred to generally as *resources*. An instance of Azure Data Factory is an example of a resource. Resources belonging to a subscription are organized further into *resource groups*. A resource group is a logical container used to collect together related resources – for example, all the resources that belong to a data warehousing or analytics platform.

Figure 1-2 illustrates the logical grouping of resources in Azure. In this section, you will create a resource group to contain an ADF instance and other resources that will be required in later chapters.

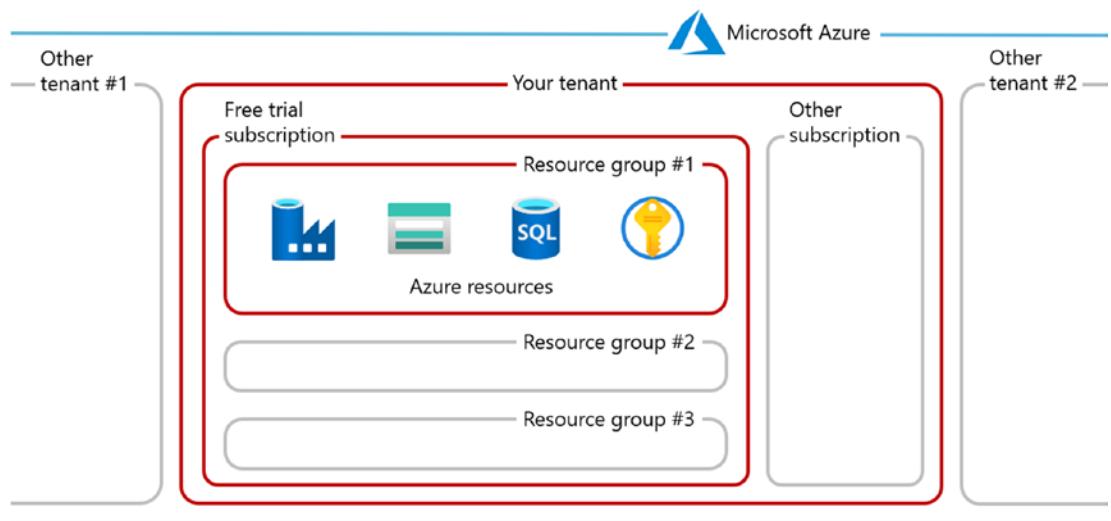


Figure 1-2. Logical resource grouping in Azure

1. Click *Create a resource*, using either the button on the portal home page or the menu button in the top left.
2. Pages in the Azure portal are referred to as *blades* – the new resource blade is shown in Figure 1-3. You can browse available services using the *Azure Marketplace* or *Popular* menus, or you can use the *Search the Marketplace* function. In the search box, start typing “resource group” (without the quotes). As you type, a filtered dropdown menu will appear. When you see the “Resource group” menu item, click it. This takes you to the resource group overview blade.

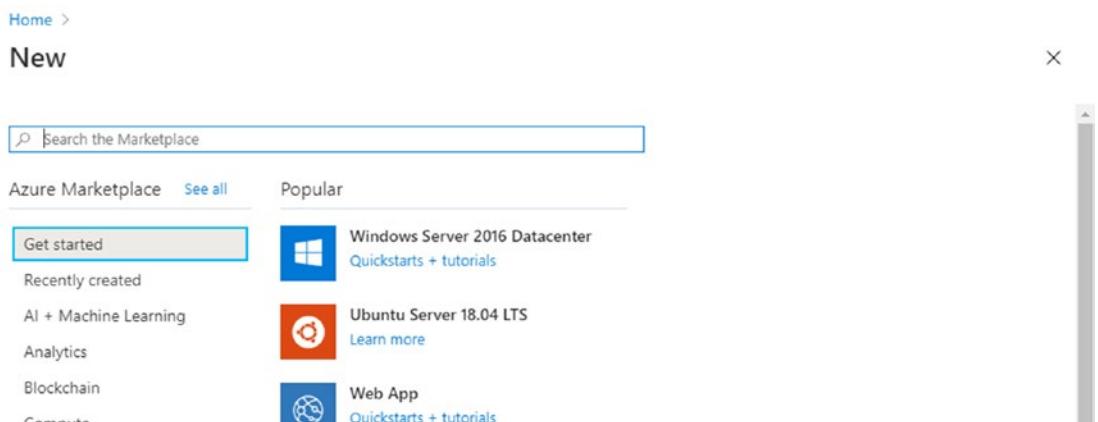


Figure 1-3. New resource blade

3. The resource group overview blade provides a description of resource groups and a *Create* button. Click the button to start creating a new resource group.
4. Complete the fields on the *Create a resource group* blade, shown in Figure 1-4. Ensure that your trial subscription is selected in the *Subscription* field, and provide a name for the new resource group. I use resource group names ending in “-rg” to make it easy to see what kind of Azure resource this is. Choose a *Region* geographically close to you – mine is “(Europe) UK South,” but yours may differ. When you are ready, click *Review + create*.

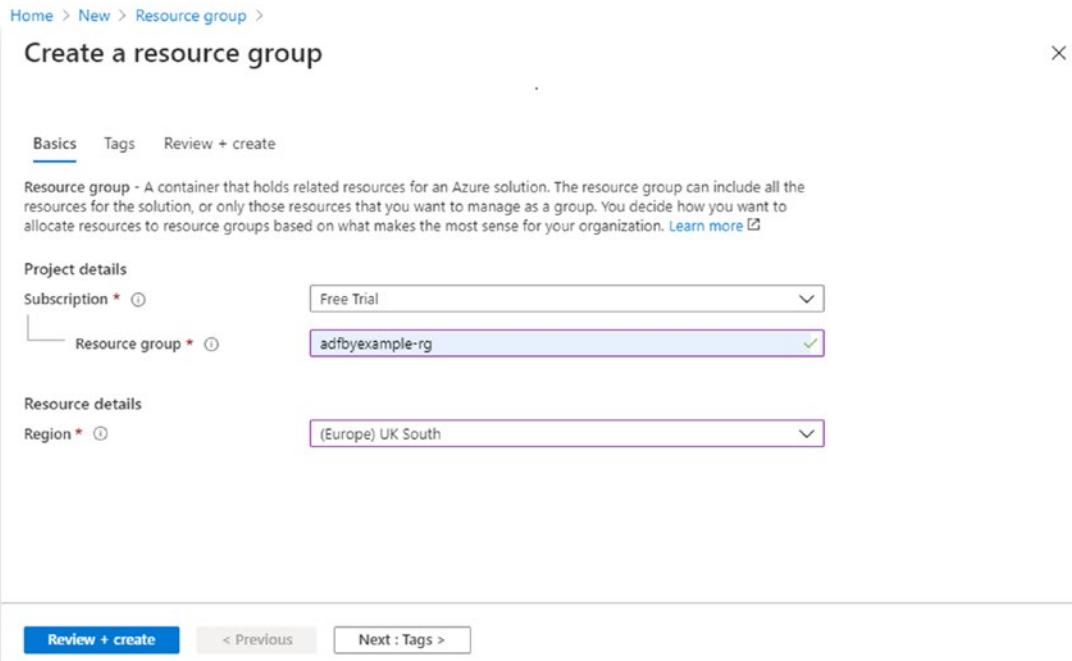


Figure 1-4. Create a resource group blade

5. On the *Review + create* tab which follows, check the details you have entered, then click *Create*.

Note You will notice that I have skipped the *Tags* tab. In an enterprise environment, tags are useful for labeling resources in different ways – for example, allocating resources to cost centers within a subscription or flagging development-only resources to enable them to be stopped automatically overnight and at weekends. I won't be using tags in this book, but your company may use a resource tagging policy to meet requirements like these.

Create an Azure Data Factory

The resource group you created in the previous section is a container for Azure resources of any kind. In this section, you will create the group's first new resource – an instance of Azure Data Factory.

1. Go back to the Azure portal home page and click *Create a resource*, in the same way you did when creating your resource group.
2. In the *Search the Marketplace* box on the new resource blade, enter “data factory”. When “Data Factory” appears as an item in the dropdown menu, select it, then on the data factory overview blade, click *Create*.
3. The *Basics* tab of the *Create Data Factory* blade is displayed, as shown in Figure 1-5. Select the *Subscription* and *Resource group* you created earlier, then choose the *Region* that is geographically closest to you.
4. Choose a *Name* for your ADF instance. Data factory names can only contain alphanumeric characters and hyphens and must be *globally* unique – your choice of name will not be available if someone else is already using it. I use data factory names ending in “-adf” to make it easy to see what kind of Azure resource this is.

CHAPTER 1 CREATING AN AZURE DATA FACTORY INSTANCE

The screenshot shows the 'Create Data Factory' blade in the Azure portal. At the top, there's a breadcrumb navigation: Home > New > Data Factory >. The main title is 'Create Data Factory'. On the right, there's a close button (X). Below the title, there are tabs: Basics (which is selected), Git configuration, Networking, Advanced, Tags, and Review + create.

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *: Free Trial

Resource group *: adfbbyexample-rg (highlighted in purple)

Instance details

Region *: UK South

Name *: adfbbyexample-adf (highlighted in purple with a green checkmark)

Version *: V2

At the bottom, there are three buttons: 'Review + create' (highlighted in blue), '< Previous', and 'Next : Git configuration >'.

Figure 1-5. Create Data Factory blade

5. Set *Version* to “V2.” (This book is concerned exclusively with Azure Data Factory V2 – ADF V1 remains available solely to support legacy implementations).
6. Click the *Next: Git configuration* button, then on the *Git configuration* tab, tick the *Configure Git later* checkbox.
7. Finally, click *Review + create*, check the factory settings you provided in steps 3 to 6, then click *Create* to start deployment. (I am purposely bypassing the three remaining tabs – *Networking*, *Advanced*, and *Tags* – and accepting their default values.)

When deployment starts, a new blade containing the message *Deployment is in progress* is displayed. The creation of a new ADF instance usually takes no more than 30 seconds, after which the message *Your deployment is complete* will be displayed. Click *Go to resource* to inspect your new data factory.

The portal blade displayed when you click *Go to resource* provides an overview of your data factory instance. It contains access controls and other standard Azure resource tools, along with monitoring information and basic details about the factory – for example, its subscription, resource group, and location. The portal does not provide tools for working inside ADF.

Beneath the factory's basic details, you will find two tiles: *Documentation* and *Author & Monitor*. Click the *Author & Monitor* tile to launch the *Azure Data Factory User Experience*. This is where you will spend most of your time when working with ADF.

Explore the Azure Data Factory User Experience

The Azure Data Factory User Experience (ADF UX) provides a code-free *integrated development environment* (IDE) for authoring ADF pipelines, publishing them, then scheduling and monitoring their execution. You'll use the ADF UX frequently, so it's a good idea to bookmark this page.

Figure 1-6 shows the ADF UX's overview page. Within the UX, you can return to this page by clicking the *Data Factory overview* button (home icon) in the navigation sidebar.

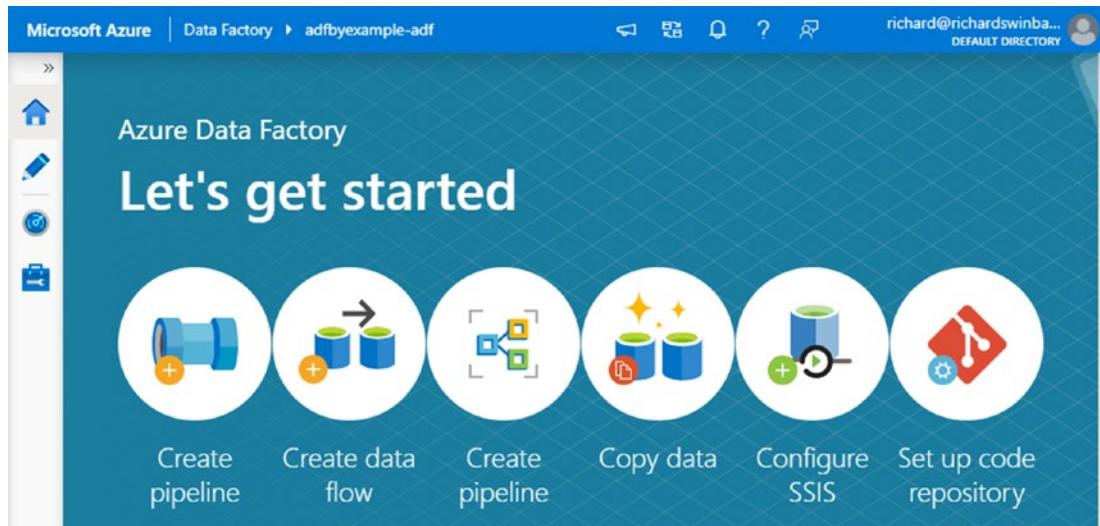


Figure 1-6. ADF UX Data Factory overview page

The overview page has three regions:

- A navigation header bar
- An expandable navigation sidebar
- A content pane, currently displaying the Data Factory overview.

The navigation header bar and sidebar are visible at all times, wherever you are in the ADF UX. The content pane displays different things, depending on which part of the UX you are using.

Navigation Header Bar

Figure 1-7 shows the ADF UX with the navigation sidebar expanded and the navigation header bar functions labeled. For clarity, the content pane has been removed from the screenshot.

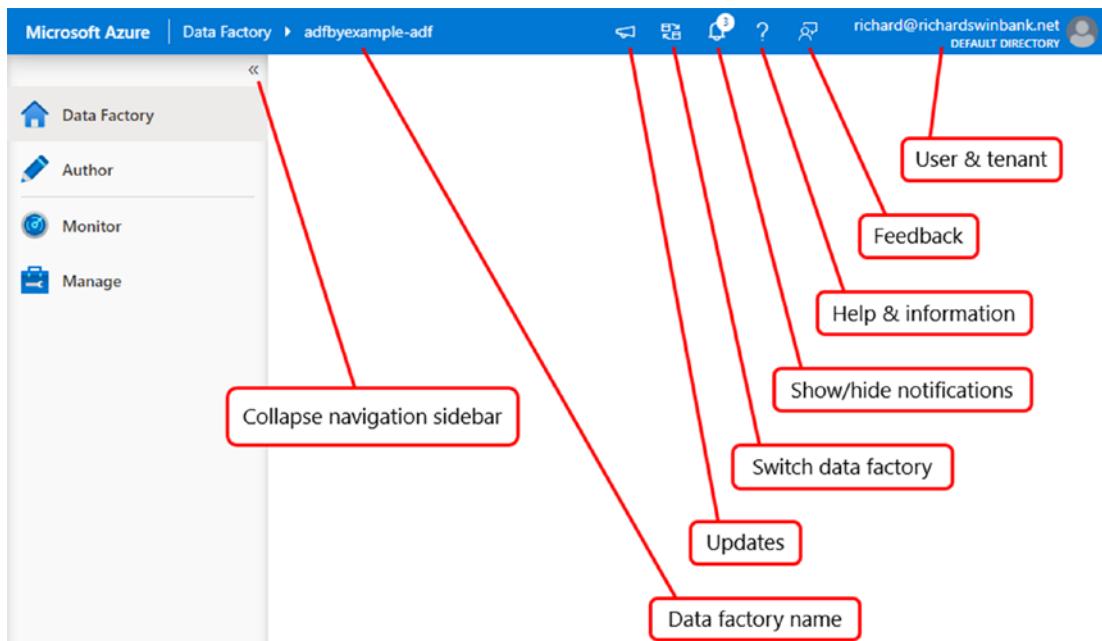


Figure 1-7. Labeled ADF UX navigation header bar

Toward its left-hand end, the navigation header bar indicates the name of the data factory instance to which the ADF UX is connected. At its other end, it identifies the current user and tenant, in the same way as in the Azure portal. Between the two is a row of five buttons:

- **Updates:** Displays recent updates to the Azure Data Factory service. ADF is in constant development and evolution – announcements about changes to the service are made here as they happen.
- **Switch Data Factory:** Enables you to disconnect from the current ADF instance and connect to a different one.

Note When you opened the ADF UX from the Azure portal data factory blade, it connected automatically to the new factory. In fact, the ADF UX is *always* connected to an ADF instance. If you access it directly (using the URL <https://adf.azure.com/>), you are required to select a data factory before the ADF UX opens.

- **Show notifications:** The ADF UX automatically notifies you of events that occur during your session – this button toggles display of those notifications. The circled “3” in the screenshot indicates that there are currently three unread notifications.
- **Help/information:** Provides links to additional ADF support and information.
- **Feedback:** If you wish to provide Microsoft with feedback about your experience of Azure Data Factory, you can do so here.

Navigation Sidebar

The navigation sidebar provides access to different parts of the ADF UX, changing what is displayed in the content pane. The chevron icon at the top of the sidebar toggles its state between collapsed and expanded – in Figure 1-6, the sidebar is collapsed, while Figure 1-7 shows it expanded.

- The *Data Factory overview* button (home icon) returns you to the overview page. This page contains quick links to a number of tools to support common ADF tasks, along with links to videos, tutorials, and other learning resources. You will use one of the tools here in Chapter 2.
- The *Author* button (pencil icon) loads the ADF authoring workspace. The authoring workspace provides a visual editor for building ADF pipelines. As this book is primarily about authoring pipelines, you will be spending a lot of time here.
- The *Monitor* button (gauge icon) provides access to visual monitoring tools. Here, you are able to see ADF pipeline runs executed in the factory instance and to drill down into execution details. Chapter 12 looks at the monitoring experience in more detail.
- The *Manage* button (toolbox icon) loads the ADF management hub. This includes a variety of features such as connections to external data storage and compute resources, along with the ADF instance's Git configuration, introduced in the next section. You will return to the management hub at various times throughout this book.

Link to a Git Repository

A data factory instance can be brought under source control by linking it to a cloud-based Git repository. While it is possible to undertake development work in ADF without linking your data factory to a Git repository, there are many disadvantages of doing so – without a linked repository, even saving work in progress is difficult. Before beginning work in your new ADF instance, you will link it to a Git repository.

Tip It is easier to configure a data factory's Git repository from the ADF UX than from the Azure portal – this is why you chose the *Configure Git later* option when you created your data factory.

Create a Git Repository in Azure Repos

Before linking a data factory to a Git repository, you need a Git repository to which it can be linked. Support for different Git service providers varies between different Azure services – currently, an ADF instance can be linked to a Git repository provided by either *Azure Repos* or GitHub. Azure Repos is one of a number of cloud-native developer tools provided by *Azure DevOps Services*. Git repositories (and other service instances) provided by Azure DevOps are grouped into *projects* – in this section, you will create a free Azure DevOps *organization* to host a project, then initialize a Git repository in the new project.

1. Browse to <https://microsoft.com/devops> and sign in, using the same account you used to create your Azure tenant. Click *Start free*.
2. The *Get started with Azure DevOps* page is displayed, as shown in Figure 1-8. Near the top of the dialog is displayed the email address you signed in with and a *Switch directory* link (indicated in the figure). This indicates the Azure directory (tenant) your new Azure DevOps organization will be connected to. Use the *Switch directory* link to verify that the selected tenant is the one containing your data factory, then click *Continue*.

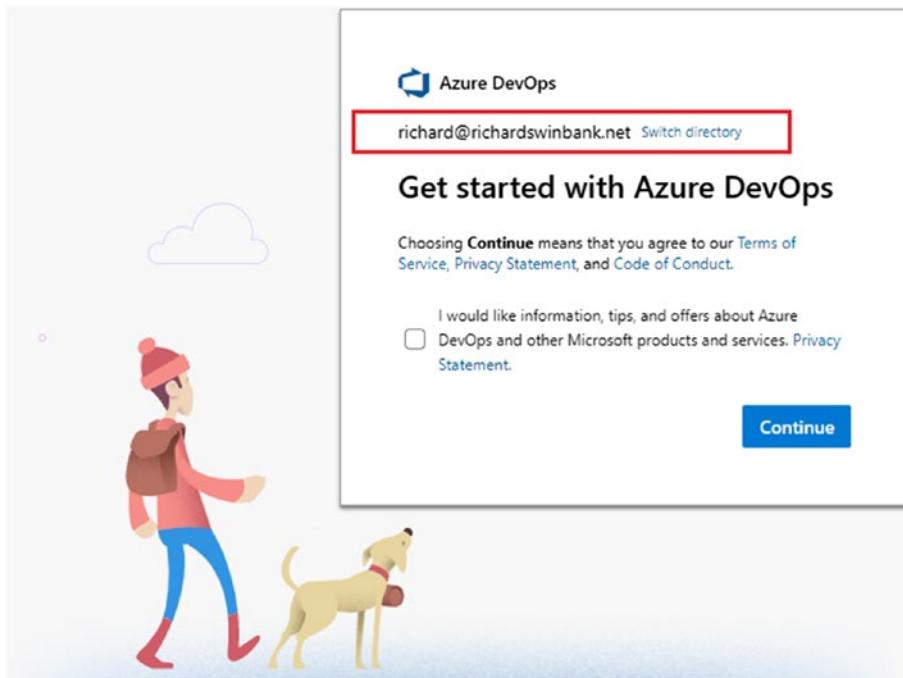


Figure 1-8. Get started dialog indicating the Azure tenant to be linked

Tip Creating your ADF instance and Git repository in the same tenant is not essential, but doing so simplifies integration between them.

3. Azure DevOps creates a new organization for you – if prompted, supply a name for it – and then displays the *Create a project to get started* pane. Choose a name for your project and enter it into the *Project name* field. Set the project's *Visibility* to “Private,” then click + *Create project*.
4. The new project’s welcome page is displayed, as shown in Figure 1-9. Choose to start with the *Azure Repos* service, either by clicking the welcome page’s *Repos* button or by selecting *Repos* (red button with branch icon) from the navigation sidebar.

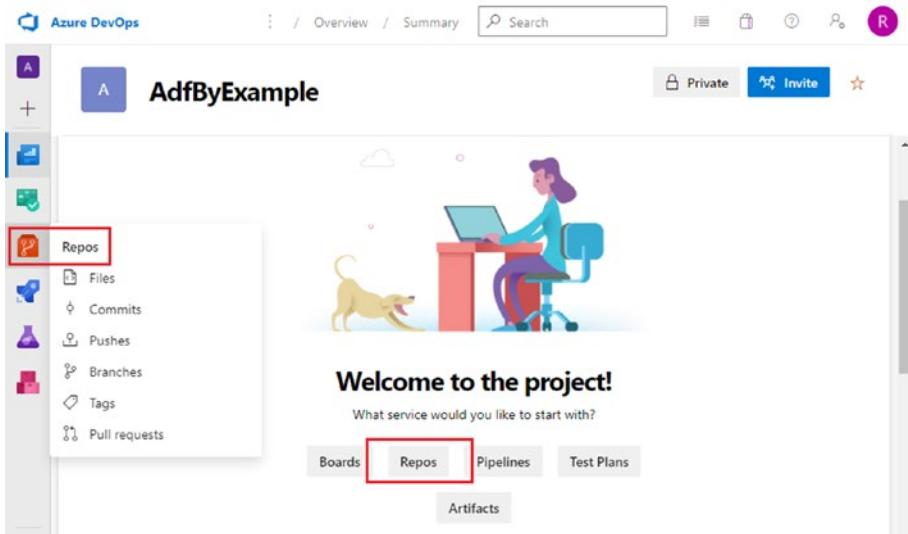


Figure 1-9. Azure DevOps project welcome page

5. Because no repositories exist yet, Azure DevOps prompts that your project is empty. Scroll down to the heading *Initialize main branch with a README or gitignore*, then click *Initialize* to create a new repository with the same name as your project.

You can choose to link a data factory to a Git repository provided either by Azure Repos or by GitHub. I have chosen an Azure Repos repository because doing so makes integration with other Microsoft services slightly simpler and because you will be using another service provided by Azure DevOps later in the book.

Link the Data Factory to the Git Repository

In this section, you will link your ADF instance to your new Git repository.

1. Return to the ADF UX and open the management hub by clicking *Manage* (toolbox icon) in the navigation sidebar.
2. In the *Source control* section of the management hub menu, click *Git configuration*.
3. The content pane indicates that no repository is configured, as shown in Figure 1-10. Click the central *Configure* button to connect the factory instance to your Git repository.

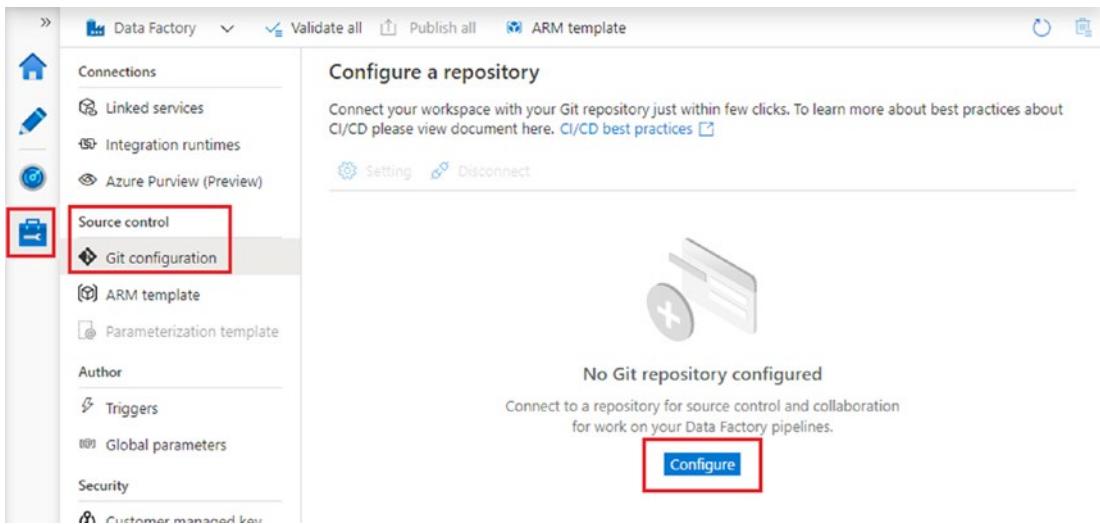


Figure 1-10. Configure a Git repository in the ADF UX management hub

4. The *Configure a repository* blade opens. Choose “Azure DevOps Git” from the *Repository type* dropdown. As you do so, more dropdown lists appear – select your Azure tenant from the *Azure Active Directory* list, then choose the Azure DevOps organization you created in the previous section from the *Azure DevOps Account* dropdown.
5. As more options appear, select the Azure DevOps project you created in the previous section from the *Project name* dropdown, then under *Repository name*, select “Use existing.” Choose your newly created repository from the dropdown list.
6. Set the factory’s *Collaboration branch* to “main” and accept the default value of “adf_publish” for *Publish branch*. Set the value of *Root folder* to “/data-factory-resources”. It is good practice to store your factory resources in a repository subfolder (rather than in the repository’s own root), because it enables you to segregate files managed by ADF from any other files stored in the same Git repository.
7. The correctly completed form, including default values for the remaining settings, is shown in Figure 1-11. Click *Apply* to link the data factory to the Git repository.

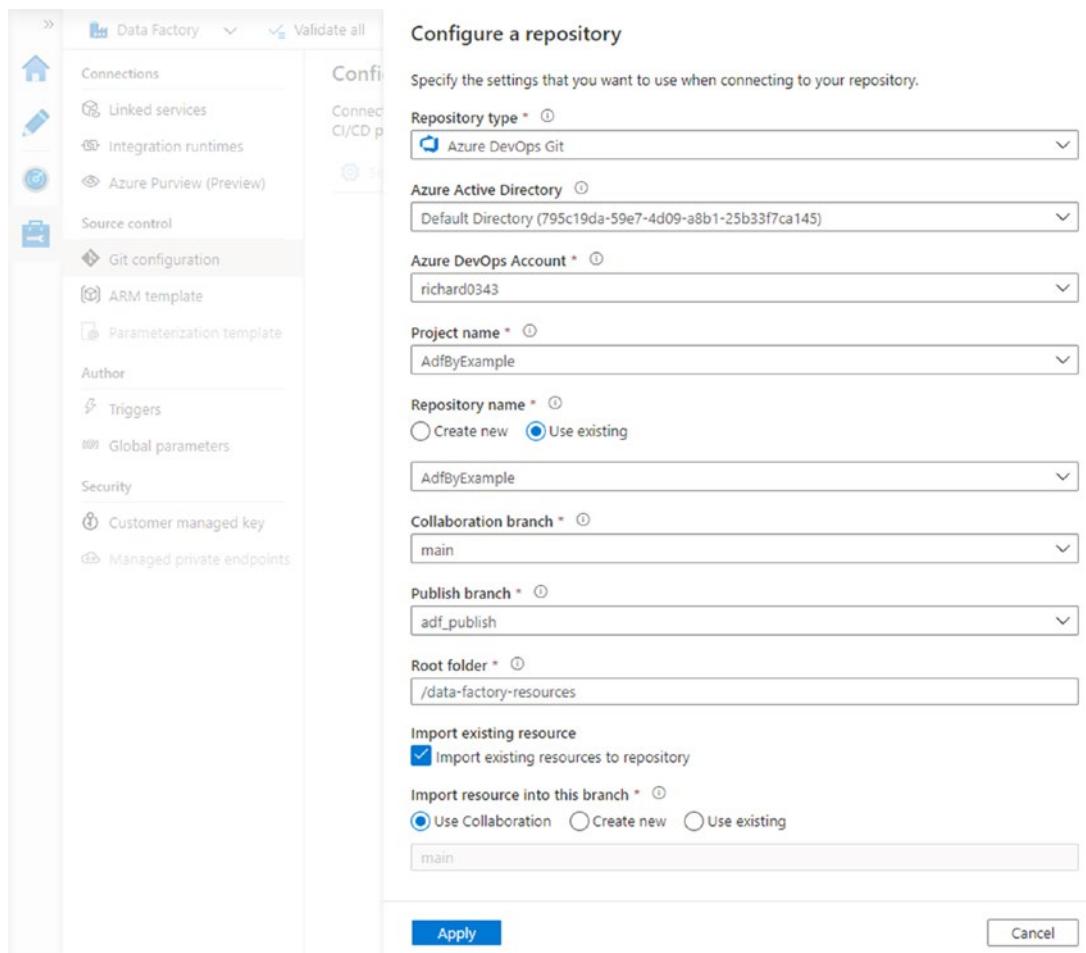


Figure 1-11. Linking an Azure DevOps Git repository to a data factory

When an ADF instance is linked to a Git repository, the “Data Factory” logo and label in the top left of the ADF UX (visible in Figure 1-11) are replaced by the logo of the selected Git repository service. Immediately to its right, the name of your working branch is displayed, defaulting to the repository’s collaboration branch.

The ADF UX as a Web-Based IDE

If you have experience with almost any other kind of development work, then the relationship between a data factory instance, Git, and the ADF UX may seem strange. In a “traditional” development model, you might use a locally installed tool like Visual

Studio to author developments on your own computer. Visual Studio enables you to debug your work using the local compute power of your own machine and stores Git repository settings locally to support source control.

In this hypothetical situation, when a piece of development work is complete, changes are deployed to target servers or services. Additional tools may be available to monitor the performance of the *published environment* – the Azure portal offers functionality like this for many Azure services. Figure 1-12 shows the high-level arrangement of components in this model. It shows two possible routes for publishing changes to the service – either directly from the development environment or, as is becoming more common, through automated deployments from the source control repository.

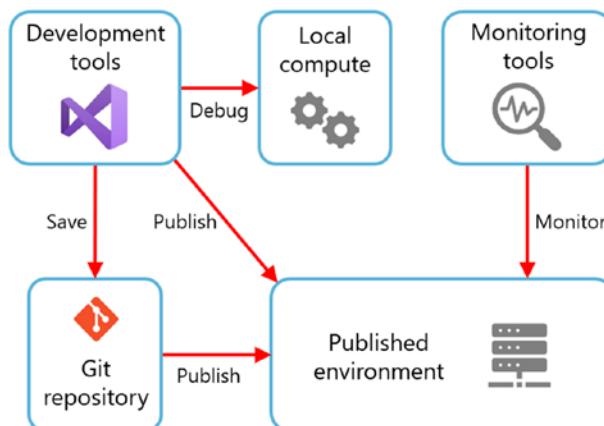


Figure 1-12. High-level components in a “traditional” development model

For SSIS developers This arrangement of components will be familiar to users of SQL Server Integration Services (SSIS). Typically, SSIS packages are authored in Visual Studio SSIS projects and committed to source control using an installed plugin. When ready, SSIS projects are published to an SSIS catalog, where reporting tools provided by SQL Server Management Studio (SSMS) enable you to monitor package behavior and performance.

When developing for Azure Data Factory, a significant difference is that its IDE – the ADF UX – is web-based. This means that the development environment has no local compute of its own. To be able to debug pipelines, the ADF UX must be attached to

cloud-based compute, which is why the ADF UX is always connected to a data factory instance. The ADF UX also has no storage of its own – all of its configuration information comes from the connected data factory – so the connected instance is the only place available to store Git repository settings. Although it is the data factory instance that is linked to a Git repository, the underlying objective is actually to link the ADF UX development environment to source control via the connected factory instance.

Figure 1-13 shows the equivalent arrangement of components for Azure Data Factory. The computing resources of the factory instance – described here informally as “factory compute” – are used in both the *debugging environment* (while developing in the ADF UX) and the published environment (after pipelines have been published). Development and management tools, frequently separated in the traditional development model, are unified for Azure Data Factory by the ADF UX.

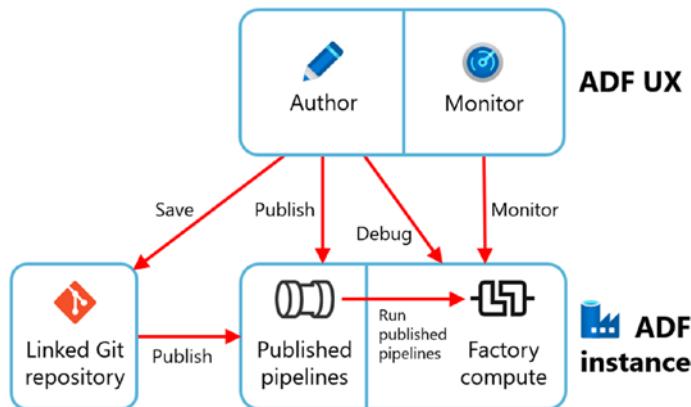


Figure 1-13. High-level components in the ADF development model

Chapter Review

In this chapter, you created an Azure subscription and a resource group inside it. You then created a Git-enabled instance of Azure Data Factory, using a repository hosted by Azure DevOps Repos. In the next chapter, you will start to use your new data factory by creating your first ADF pipeline.

Key Concepts

- **Azure tenant:** A tenant (or directory) is an instance of Azure Active Directory (AAD). It identifies Azure users and enables them to gain access to Azure services.
- **Resource:** An instance of an Azure service, enabling the service to be accessed by a tenant's users.
- **Subscription:** Specifies a means of payment for Azure services. A tenant may contain one or more subscriptions. Every Azure resource is associated with a subscription to which consumption charges can be billed.
- **Resource group:** A logical subgrouping of Azure resources inside a subscription.
- **Azure portal:** Browser-based console for creating, monitoring, and managing Azure resources.
- **Blade:** Pages in the web-based Azure portal are referred to as blades.
- **Software as a service (SaaS):** Delivery model in which end-user applications are delivered as cloud services. Microsoft Office 365 applications are examples of SaaS services.
- **Platform as a service (PaaS):** Delivery model in which application services and developer tools are delivered as cloud services.
Database management systems which do not require you to manage the underlying server infrastructure – for example, Azure SQL Database – are examples of PaaS services.
- **Serverless service:** Like PaaS, serverless cloud services provide applications typically not used directly except by IT specialists, but with additional automated management features – for example, the ability to auto-scale in response to changes in demand. Azure Data Factory is an example of a serverless service.
- **Infrastructure as a service (IaaS):** Cloud service delivery model in which computing infrastructure components – for example, virtual servers and networks – are provided without requiring you to manage

the physical hardware involved. Virtual machines give you the freedom to choose your own operating system and application software, but at the cost of managing installation and updates yourself.

- **Azure Data Factory (ADF):** Microsoft's cloud-native, serverless service for managing batch ETL and other data integration workloads.
- **Data factory instance:** An instance of the Azure Data Factory service, often referred to simply as "a data factory." An ADF instance is an example of an Azure resource.
- **ADF V2:** Version 2 of the ADF service is the latest and preferred choice for Azure Data Factory instances. ADF Version 1 only remains available to support legacy implementations.
- **Integrated development environment (IDE):** Application used for software development that includes supporting features such as editing, debugging, and breakpoint capabilities. Visual Studio is an example of an IDE that can be used with many Microsoft technologies.
- **Azure Data Factory User Experience (ADF UX):** Code-free IDE enabling visual authoring, debugging, and publishing of ADF pipelines. Additionally, the ADF UX includes tools for scheduling published pipelines and monitoring their execution.
- **Authoring workspace:** Visual editor in the ADF UX where you create, modify, and debug ADF pipelines. Use the *Author* button (pencil icon) in the navigation sidebar to access the workspace.
- **Monitoring experience:** ADF UX feature allowing you to inspect pipeline execution history. Use the *Monitor* button (gauge icon) in the navigation sidebar to access this tool.
- **Management hub:** Area of the ADF UX used to manage various factory features. In this chapter, you used the management hub to edit the ADF instance's Git configuration, and you will return to it in later chapters. Use the *Manage* button (toolbox icon) in the navigation sidebar to open the management hub.

- **Git-enabled:** An ADF instance linked to a Git repository is said to be “Git-enabled.” Linking a data factory to a Git repository allows you to commit development work from the ADF UX into source control.
- **Collaboration branch:** Git branch, usually the repository’s default branch, where changes intended for publishing to ADF are made.
- **Publish branch:** Reserved branch used by ADF to support pipeline publishing processes. Publishing to ADF is the subject of Chapter 10.
- **Root folder:** Path in a Git repository where a linked ADF instance saves its files. Choosing an ADF root folder below the level of the repository root is good practice, because it enables other data platform resources to be managed in the same Git repository.
- **Published environment:** Refers to the environment into which ADF pipelines are published after development; the environment in which real data integration work is done.
- **Debugging environment:** Refers to the use of factory resources by the ADF UX to run pipelines in development for debugging purposes.

For SSIS Developers

The ADF UX provides a collection of tools with purposes familiar to users of SQL Server Integration Services, but linked together in a new way:

- The authoring workspace provides a visual IDE for pipeline development, equivalent to using Visual Studio to develop an SSIS project.
- The monitoring experience enables you to inspect the outcome of published ADF pipelines, in a similar way to using SSIS catalog reports in SSMS to view package execution history.

Using the ADF UX is like editing an SSIS project paired permanently with a single Integration Services server. Connecting the ADF UX to a data factory is necessary because the web-based ADF UX has no compute power or storage of its own.

CHAPTER 2

Your First Pipeline

ETL workloads are implemented in Azure Data Factory in units called *pipelines*. Using the data factory instance you created in Chapter 1, in this chapter, you will create a pipeline using the *Copy Data tool* – a pipeline creation wizard that steps through creating the various components that make up a pipeline. Afterward, you’ll be able to examine the pipeline in detail to gain an understanding of how it is constructed.

The Copy Data tool guides you through building pipelines with the purpose of copying data from one place to another. Before you can do that, you need some data to copy. In the first section of this chapter, you will create an Azure Storage account and upload some sample data to work with.

Work with Azure Storage

Azure Storage is Microsoft’s managed cloud storage platform. Data stored using Azure Storage services is encrypted, replicated, and can be accessed securely from anywhere in the world. The scalability of the service’s capacity and speed makes it a good choice for many data storage and processing scenarios.

Create an Azure Storage Account

To use Azure Storage services, you must first create an *Azure Storage account*. Create your storage account as follows:

1. In the Azure portal, create a new resource of type *Storage account*. You will notice that the *Search the Marketplace* dropdown is limited to five entries – many Azure service names contain the word “storage,” so you may have to enter more text before you see the storage account option.

CHAPTER 2 YOUR FIRST PIPELINE

2. Complete the *Basics* tab of the *Create a storage account* form (Figure 2-1). Under *Project details*, select the *Subscription* and *Resource group* you used to create your data factory in Chapter 1.

Home > New > Storage account >

Create a storage account

X

Basics Advanced Networking Data protection Tags Review + create

Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below. [Learn more about Azure storage accounts](#)

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *

Resource group * [Create new](#)

Instance details

The default template offers choices that must be made at create time. You may choose to deploy using the original Resource Manager which offers all Azure feature, including legacy options. [Choose the original resource manager](#)

Storage account name *

Location *

Performance * Standard: GPV2 account recommended for most scenarios
 Premium: Recommended for scenarios that need minimal retrieval delays

Redundancy *

Review + create [< Previous](#) [Next : Advanced >](#)

Figure 2-1. Basics tab of the Create a storage account blade

Note I suggest you use the same resource group because it will help you keep track of resources you create while using this book. It definitely isn't a requirement for Azure Data Factory – your ADF instance can connect to resources pretty much anywhere! These might be resources in other resource groups, subscriptions, or Azure tenants; resources in competitor cloud platforms like Amazon Web Services (AWS) or Google Cloud Platform (GCP); or even your own on-premises systems.

3. Specify a globally unique *Storage account name*. I use names ending in “sa” (storage account names may only contain lowercase alphanumeric characters).
 4. Choose the *Location* closest to you geographically – the one where you created your data factory.
-

Tip Choosing a location close to you reduces data retrieval latency. Choosing the same location as your data factory reduces cost, because moving data out of one Azure region and into another incurs a *bandwidth* charge (sometimes referred to as an *egress* charge).

5. For *Performance*, select “Standard.” Performance tiers for storage are linked to the underlying hardware type. Premium storage uses solid-state disks and is more expensive.
6. Select the *Redundancy* option “Locally-redundant storage (LRS).” This is the cheapest of the available options because data is only replicated within the same data center. LRS protects you against hardware failure but not against data center outage or loss – this is sufficient for learning or development purposes, but in production environments, a higher level of resilience is likely to be required.
7. The final step on the *Basics* tab is to click *Review + create*, then after validation click *Create*. (I am purposely bypassing the four remaining tabs – *Advanced*, *Networking*, *Data protection*, and *Tags* – and accepting their default values.)

8. A notification message is displayed when deployment is complete, including a *Go to resource* button. Click it to open the portal's storage account blade (shown in Figure 2-2).

Tip You can navigate to any resource from the portal menu (top left) or home page. Use the *Resource groups* option to display all your resource groups, then select one from the list to explore the resources it contains.

Explore Azure Storage

A variety of tools are available for interaction with storage accounts. One easy-to-use option is *Azure Storage Explorer*, available as a downloadable app (available for Windows and other operating systems) or online, hosted inside the portal. You can launch both the online app and its locally installed equivalent directly from the portal – to launch the online app, scroll down and click *Storage Explorer (preview)* in the portal's storage account blade (Figure 2-2).

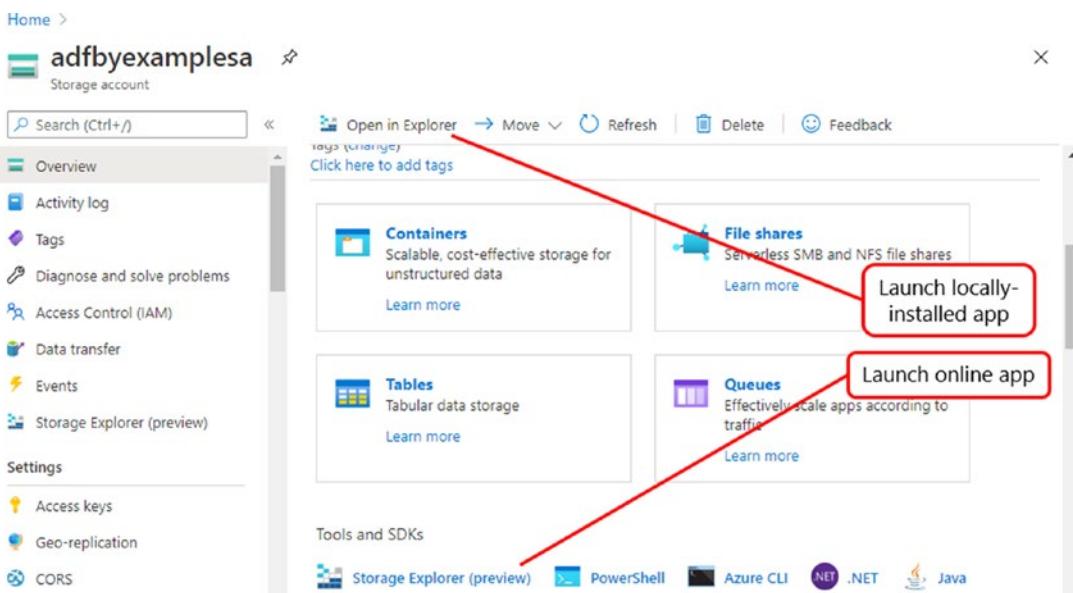


Figure 2-2. Azure portal storage account blade

Figure 2-3 shows the online Storage Explorer app with the portal navigation sidebar collapsed. The explorer sidebar shows the four types of storage supported by the storage account: *blob containers* (blob storage), *file shares*, *queues*, and *tables*. In the next section, you will add files to blob storage.

Note The term *blob* is used to refer to a file without considering its internal data structure. This doesn't imply that files described as blobs have no structure – it simply means that the structure isn't important for the task at hand. The name “blob storage” reflects the fact that the service provides a general-purpose file store, with no restrictions on the types of file it can contain.

Blob storage is divided into a single level of blob *containers* – containers cannot be nested. Right-click the *BLOB CONTAINERS* item and use the popup menu's *Create blob container* option to create two private blob containers called “landing” and “sampledata” as shown in Figure 2-3.

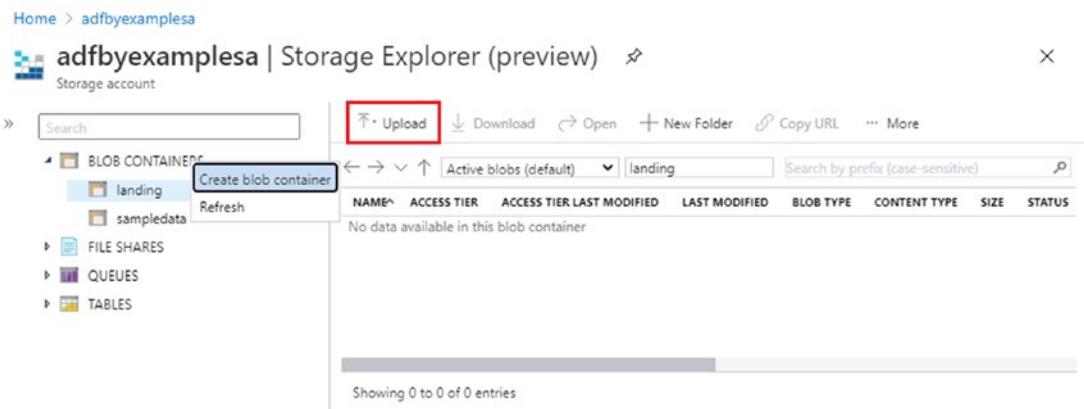


Figure 2-3. Azure Storage Explorer (online version)

Upload Sample Data

Sample data files used in this book are available from the book's GitHub repository, located at <https://github.com/Apress/azure-data-factory-by-example>.

1. Download the repository as a zip file, so that you can transfer sample data into your storage account. This option is available from the green *Code* menu button on the repository home page.
2. Select the “landing” container in the online Azure Storage Explorer and click *Upload* on the toolbar above the container contents list (indicated in Figure 2-3).
3. The *Upload blob* blade is displayed. In the *Files* field, click *Select a file*, then find and select *azure-data-factory-by-example-main.zip*, the zip file you downloaded.
4. Back in the *Upload blob* blade, click *Upload*, then close the blade.
5. An entry for the zip file now appears in the “landing” container contents list. (If you don’t see it, try selecting *Refresh* from the *More* menu items collection on the ribbon.)

The sample data files contain sales data for products made by a fictional multinational confectionery manufacturer, Acme Boxed Confectionery (ABC). The manufacturer does not sell directly to consumers, but to a variety of retailers which report monthly sales activity back to ABC. Sales reports are typically produced using retailers’ own data management systems and are supplied in a wide variety of file formats. Handling these formats will expose you to many of ADF’s data transformation features in the coming chapters.

Use the Copy Data Tool

Azure Data Factory’s Copy Data tool provides a wizard-style experience for creating a pipeline with a specific purpose: copying data from one place to another. In this section, you will use the Copy Data tool to create a pipeline that copies the zip file from the “landing” container in your Azure storage account, unzips it, then writes its contents into the “sampledata” container. This is purposely a very simple data movement task – implementing a simple task allows you to focus on the detail of the ADF pipeline setup.

The Copy Data tool is found on the ADF UX *Data Factory overview* page, accessed by clicking the home icon in the navigation sidebar. Under the heading *Let’s get started* are a number of bubbles – click the *Copy data* bubble to begin.

The tool launches a guided multistep process to create a pipeline – the page for the first step appears in Figure 2-4.

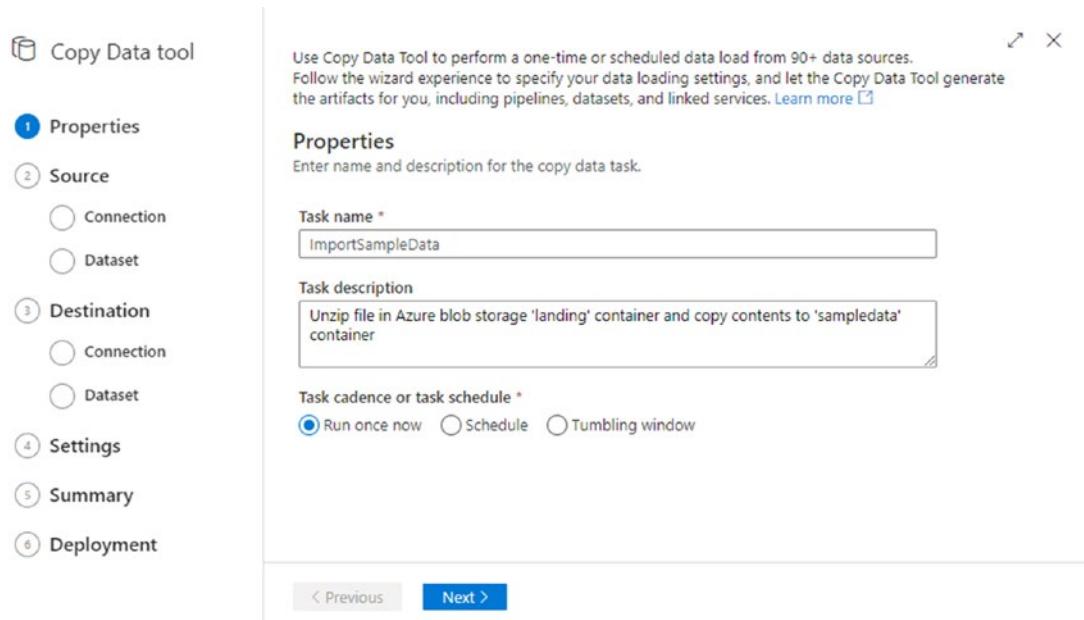


Figure 2-4. First step of the Copy Data tool

Tip If the process shown in Figure 2-4 fails to launch, click the pencil icon to access the authoring workspace, then find the search field below the *Factory Resources* explorer title. Click the plus symbol button to the right of the search box, then select *Copy Data tool* from the popup menu.

Complete the process as follows:

1. On the *Properties* page, set *Task name* to “ImportSampleData” – this will be the name of your pipeline – and provide a *Task description*. Descriptions are searchable, so adding one to every new pipeline makes it easier to manage a large data factory. Click *Next*.
2. On the *Source data store* page, click + *Create new connection*. Choose the linked service type *Azure Blob Storage* and click *Continue*.

3. Use the *New linked service (Azure Blob Storage)* blade (Figure 2-5) to create a connection to your Azure Storage account. Provide a name and description – I have reused the name of the underlying storage account – then under *Account selection method*, ensure that “From Azure subscription” is selected. Choose the relevant subscription and storage account you created earlier. At the bottom of the blade, click *Test connection* to check that it’s working, then click *Create*.

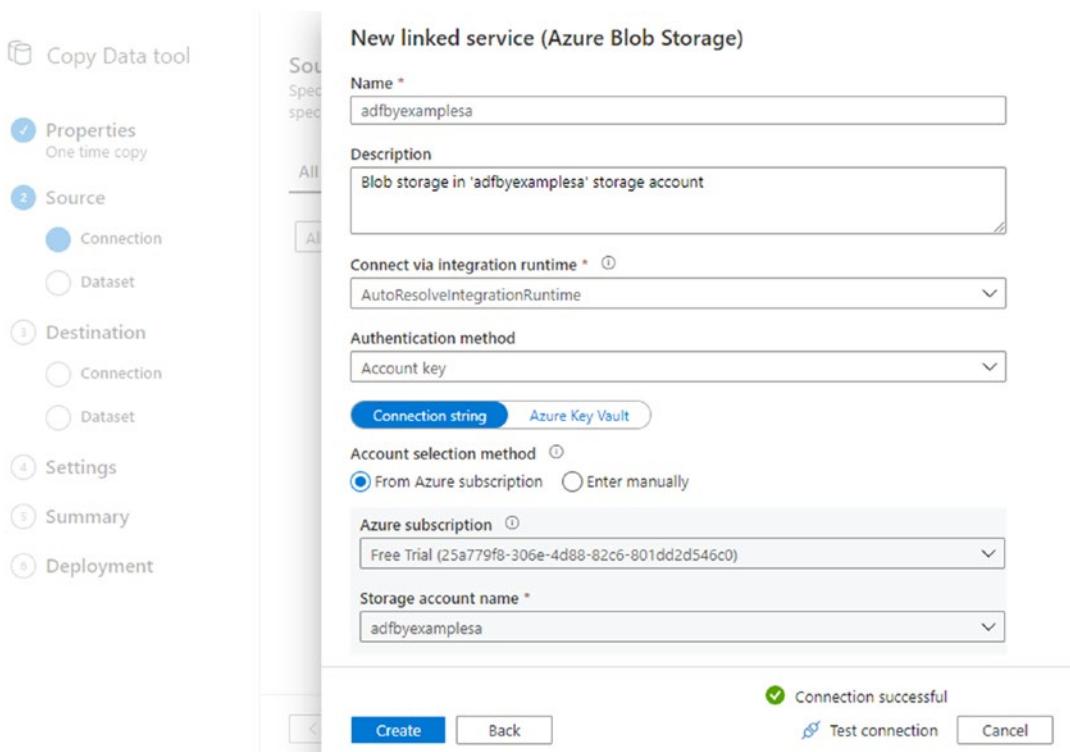


Figure 2-5. New linked service (Azure Blob Storage) dialog

Note The ADF UX uses a *storage key* (part of your storage account’s configuration) to authorize connection to your storage account. The reason you don’t need to specify a key explicitly is that the UX uses your logged-in identity to retrieve its value.

4. Back on the *Source data store* page, you will see a tile for your new linked service – ensure it is selected, then click *Next*.
5. On the *Choose the input file or folder* page, click the *Browse* icon to the right of the *File or folder* text box. Browse into the “landing” container, select the uploaded zip file, and click *Choose*.
6. Tick the *Binary copy* checkbox, select *Compression type* “ZipDeflate,” and uncheck *Preserve zip file name as folder*. Click *Next*.
7. You will find the linked server you created in step 3 displayed in a tile on the *Destination data store* page. Select it and click *Next*.
8. On the *Choose the output file or folder* page, click the *Browse* icon, select the “sampledata” container, and click *Choose*. Leave the other settings unchanged and click *Next*.
9. Click *Next* again on the *Settings* page, then inspect the details you have provided on the *Summary* page. When you are ready, click *Next* to start pipeline creation.

The last stage in the Copy Data tool process is the *Deployment* page. This will quickly run through resource creation before a *Finish* button appears, as in Figure 2-6, indicating that you have successfully created your pipeline. Click *Finish* to close the tool.

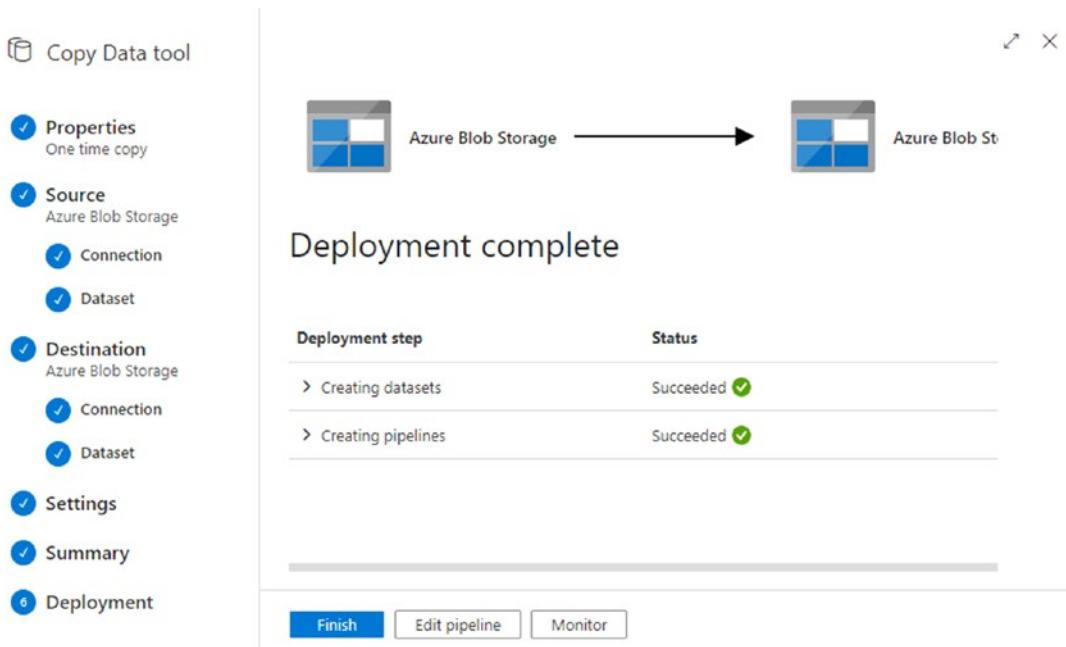


Figure 2-6. Copy Data tool's Deployment page

Explore Your Pipeline

In the previous section, you used the Copy Data tool to create your first ADF pipeline. To achieve this, the ADF UX took the following actions on your behalf:

- Created a linked service connection to your storage account
- Published the linked service connection
- Created datasets representing the inputs and outputs of the copy process
- Created an ADF pipeline to perform the copy process
- Committed and pushed the linked service, datasets, and pipeline to your Git repository

Note To prevent exposure, the storage key used to authorize ADF's connection to your storage account is not committed to Git. Instead, the key is saved directly to ADF by publishing the linked service connection immediately.

In this section, you will examine the various resources created by the Copy Data tool.

Linked Services

A common way to define resources for any data operation is in terms of *storage* and *compute*:

- **Storage** refers to the holding of data, without any additional processing or movement being performed (except, e.g., movement occurring within the storage system for storage replication).
- **Compute** describes computational power used to move stored data around, to transform it, and to analyze it.

The separation of storage and compute services in cloud platforms like Azure is very common. It adds flexibility by allowing the two to be scaled up and down independently of one another as demand rises and falls.

Azure Data Factory has no storage resources of its own, but factory instances can access and use external storage and compute resources via *linked services*. A linked service may provide a connection to a storage system – for example, an Azure storage account or a database – or may enable access to external compute resources like an Azure Function App or a Databricks cluster.

Using the Copy Data tool, you created an Azure Blob Storage linked service to connect to your Azure storage account. Linked services are defined in the ADF UX's management hub, accessed by clicking the toolbox icon in the navigation sidebar. Figure 2-7 shows the *Linked services* page of the management hub, containing the Azure Blob Storage connection created earlier.

Name	Type	Related	Annotations
adfbbyexamplesa	Azure Blob Storage	2	

Figure 2-7. Linked services in the ADF UX management hub

Datasets

A linked service storage connection provides information necessary to connect to an external data store, but nothing more. For a database system, this might consist of a database connection string, including server, database, and credential details, but with no information about database tables. Similarly, the linked service you created in the previous section contains metadata to locate the relevant Azure Blob Storage account, but has no information about files stored there. Azure Data Factory stores metadata to represent objects inside external storage systems as *datasets*.

Datasets are defined in the ADF UX's authoring workspace (accessed by clicking the pencil icon in the navigation sidebar). In here, you will find two datasets created by the Copy Data tool – one representing the source zip file of the copy activity, the other the destination container. The workspace has three regions, as shown in Figure 2-8:

- A factory header bar (beneath the main navigation header bar), containing various controls and indicating that the factory is linked to an Azure DevOps Git repository
- The *Factory Resources* explorer, listing pipelines, datasets, and other factory resources
- A tabbed *authoring canvas*, displaying details of selected factory resources

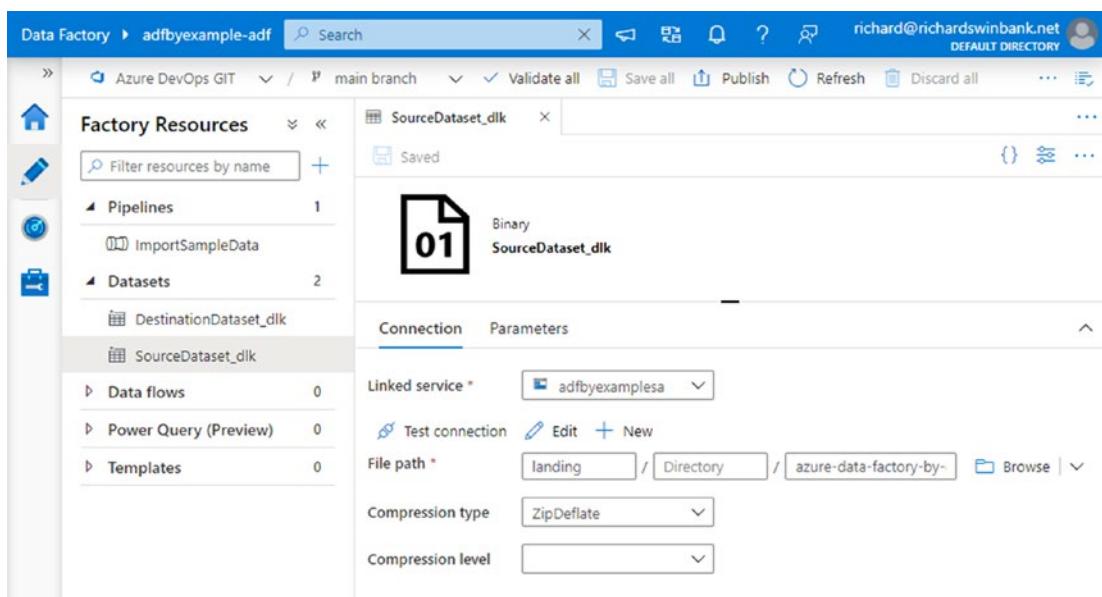


Figure 2-8. Dataset configuration in the ADF UX authoring workspace

Tip The term *resource* is used in the Azure portal to describe different instances of Azure services (e.g., a storage account or a data factory) and is used within ADF to describe various factory components such as pipelines and datasets. The reuse of terminology can be confusing, but it should be clear from the context whether I'm referring to Azure resources or ADF resources.

Notice also that, in the authoring workspace, the navigation header bar features a *Search* box. This allows you to search factory resource definitions, including text descriptions like the one you created for the “ImportSampleData” pipeline.

The authoring canvas in Figure 2-8 shows the source dataset “SourceDataset_dlk”. (The dataset name was generated automatically by the Copy Data tool). The *Connection* tab on the tabbed configuration pane in the lower half of the screen displays details of the selected file path: the “landing” container, file “azure-data-factory-by-example-main.zip,” and compression type “ZipDeflate.”

For SSIS developers Linked services behave in a similar way to project-scoped connection managers in SSIS, but unlike some connection managers (such as flat file), they do not contain schema metadata. Schema metadata can be defined separately in an ADF dataset, although it may not always be required (as in the case of the schemaless file copy you performed using the Copy Data tool).

Pipelines

Pipelines are at the very heart of Azure Data Factory. A *pipeline* is a collection of data movement and transformation *activities*, grouped together to achieve a higher-level data integration task. Figure 2-9 shows the ADF UX authoring workspace with the “ImportSampleData” pipeline displayed. When authoring pipelines, the workspace additionally contains an *Activities* toolbox, a menu of available pipeline activities. Activities can be dragged from the Activities toolbox and dropped onto the authoring canvas.

On the authoring canvas in Figure 2-9, you can see that the pipeline contains a single *Copy data* activity, named “Copy_dlk”. (The activity name was also generated automatically by the Copy Data tool.)

For SSIS developers A data factory pipeline is equivalent to an SSIS package, and the authoring canvas shown in Figure 2-9 provides functionality comparable to the SSIS control flow surface. This simple pipeline is like an SSIS package which contains a single File System Task to copy files from one location to another – but with the additional ability to unzip files on the fly.

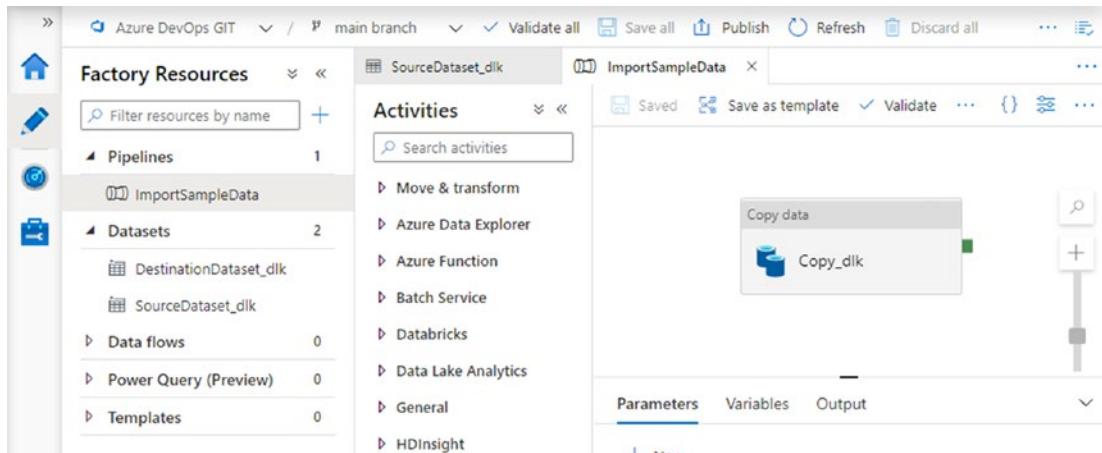


Figure 2-9. Pipeline configuration on the ADF UX authoring canvas

Activities

The Activities toolbox provides a variety of activity types available for use in an ADF pipeline. Activities are available for the native movement and transformation of data inside ADF, as well as to orchestrate work performed by external resources such as Azure Databricks and Machine Learning services.

This simple pipeline contains only one activity, but in Chapter 4 you will start to see how multiple activities can be linked together inside a pipeline to orchestrate progressively more complex ETL workflows. ADF defines a library of over 30 activity types, including the ability to write your own custom activities, making the scope of tasks that a pipeline can accomplish virtually limitless.

Integration Runtimes

Azure Data Factory accesses compute in one of two ways:

- Externally, by creating a linked service connection to a separate compute service like Azure Databricks or HDInsight
- Internally, using an ADF-managed compute service called an *integration runtime*

Native data transformations and movements – such as the Copy data activity – use compute resources provided by an integration runtime. This is what I described in Chapter 1 as “factory compute.”

Integration runtimes are listed in the ADF UX’s management hub, immediately below *Linked services* on the hub menu. Every ADF instance automatically includes one integration runtime, called the *AutoResolveIntegrationRuntime*. This provides access to Azure compute resources in a geographic location that is chosen automatically, depending on the task being performed. Under certain circumstances, you may wish to create integration runtimes of your own – Chapter 8 returns to this question in greater detail.

The compute required for your pipeline’s Copy data activity is provided by the *AutoResolveIntegrationRuntime*. This is not specified as part of the activity itself, but as part of the storage account linked service(s) it uses. If you review Figure 2-5, you will notice that the option *Connect via integration runtime* has the value “*AutoResolveIntegrationRuntime*.”

For SSIS developers The closest parallel to an integration runtime in SSIS is the Integration Services Windows service – it provides access to server compute resources for data movement and transformation. The concept is less visible in SSIS, simply because there is only one runtime, used by all tasks in all packages.

Figure 2-10 illustrates the relationship between linked services, datasets, activities, integration runtimes, and your pipeline. The arrows indicate the direction of data flow from the “landing” to the “sampledata” container.

In the figure, you can see that

- The dataset “SourceDataset_dlk” uses the “adfbyexamplesa” linked service to connect to the “landing” container in the storage account of the same name.
- The dataset “DestinationDataset_dlk” uses the “adfbyexamplesa” linked service to connect to the “sampleddata” container in the same storage account.
- The “ImportSampleData” pipeline contains a single Copy data activity, “Copy_dlk”, which uses the AutoResolveIntegrationRuntime to copy data from “SourceDataset_dlk” to “DestinationDataset_dlk”.

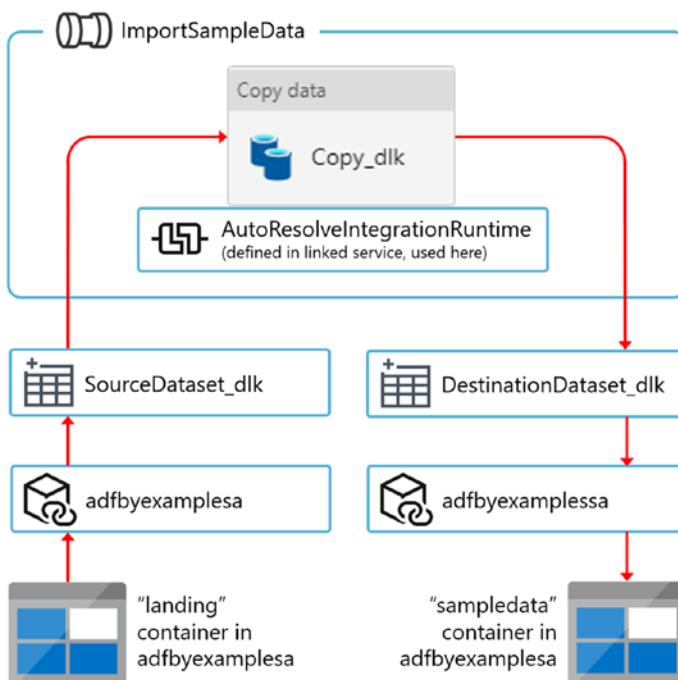


Figure 2-10. Relationship between Azure Data Factory resources

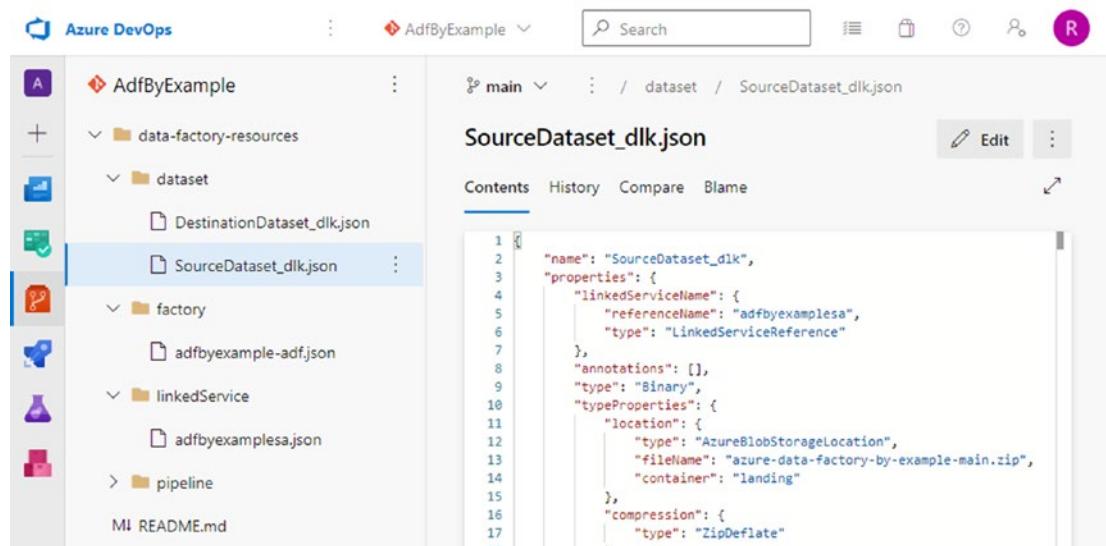
The features of pipeline implementation illustrated by Figure 2-10 are very common:

- Access to identified data in external storage is provided by a dataset via a linked service connection.
- Pipeline activities use an ADF integration runtime to move and transform data between datasets.

Factory Resources in Git

Linked services, datasets, and pipelines are all examples of factory resources. In addition to creating resource definitions in the ADF UX, the Copy Data tool also saved them, by committing and pushing them to the collaboration branch of your Git repository.

Look at your Azure DevOps repository again, and you will see a structure similar to Figure 2-11. The screenshot shows the folder structure of my “AdfByExample” repository, which now contains four folders – “dataset,” “factory,” “linkedService,” and “pipeline.” Each folder contains definitions for resources of the corresponding type, stored as JSON files – the content pane on the right shows the JSON definition of my “SourceDataset_dlk” dataset.



```

1  "name": "SourceDataset_dlk",
2  "properties": {
3      "linkedServiceName": {
4          "referenceName": "adfbbyexamplesa",
5          "type": "LinkedServiceReference"
6      },
7      "annotations": [],
8      "type": "Binary",
9      "typeProperties": {
10         "location": {
11             "type": "AzureBlobStorageLocation",
12             "fileName": "azure-data-factory-by-example-main.zip",
13             "container": "landing"
14         },
15         "compression": {
16             "type": "ZipDeflate"
17         }
18     }
}

```

Figure 2-11. Git repository contents after creating factory resources

Working with – and committing directly to – your collaboration branch is characteristic of a centralized Git workflow. More sophisticated development workflows using Git feature branches are also supported by the ADF UX. The branch dropdown, indicated in Figure 2-12, enables you to switch between Git branches, create new ones, and open pull requests. Development workflows are outside the scope of this book – during the following chapters, you should continue to commit changes directly to the factory’s collaboration branch, `main`.

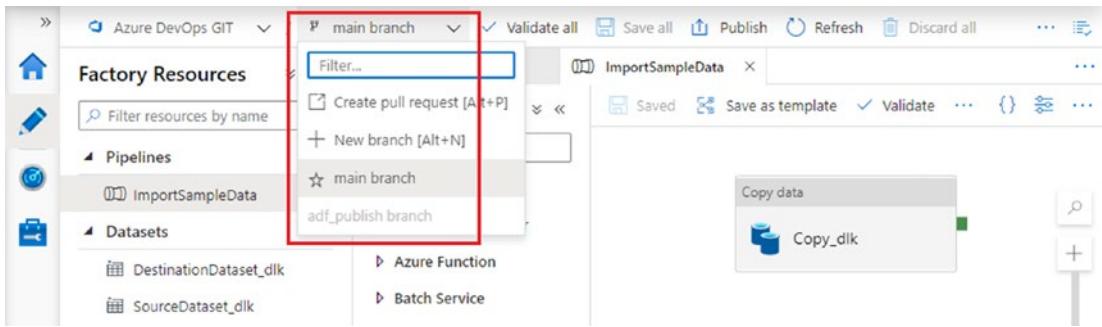


Figure 2-12. Git branch selection dropdown

Debug Your Pipeline

With the work you've done using the Copy Data tool, you now have resource definitions in a number of different places:

- Your Git repository contains saved linked service, dataset, and pipeline definitions.
- Those definitions are loaded into your ADF UX session, where you can edit them.
- The ADF published environment contains the linked service definition (because the Copy Data tool published it in order to save the storage account key securely).

To run the pipeline in the published environment, you would need first to publish *all* its related resources. Publishing factory resources is the subject of Chapter 10. Until then, you will be running pipelines interactively in the ADF UX, using its *Debug* mode – whenever I say “run the pipeline” from now on, I mean “Click *Debug* to run the pipeline.” (You may have found the *Trigger now* option on the *Add trigger* menu above the authoring canvas – this executes published pipelines and will also be examined in Chapter 10.)

Note “Debug” means simply “run the pipeline definition in my ADF UX session, without publishing it.” A pipeline debug run accesses and modifies external resources in exactly the same way as a published pipeline.

Run the Pipeline in Debug Mode

To run the “ImportSampleData” pipeline in debug mode, open the authoring canvas and select the pipeline from the *Factory Resources* explorer. The toolbar immediately above the authoring canvas contains a *Debug* button, as shown in Figure 2-13 – click it to run the pipeline.

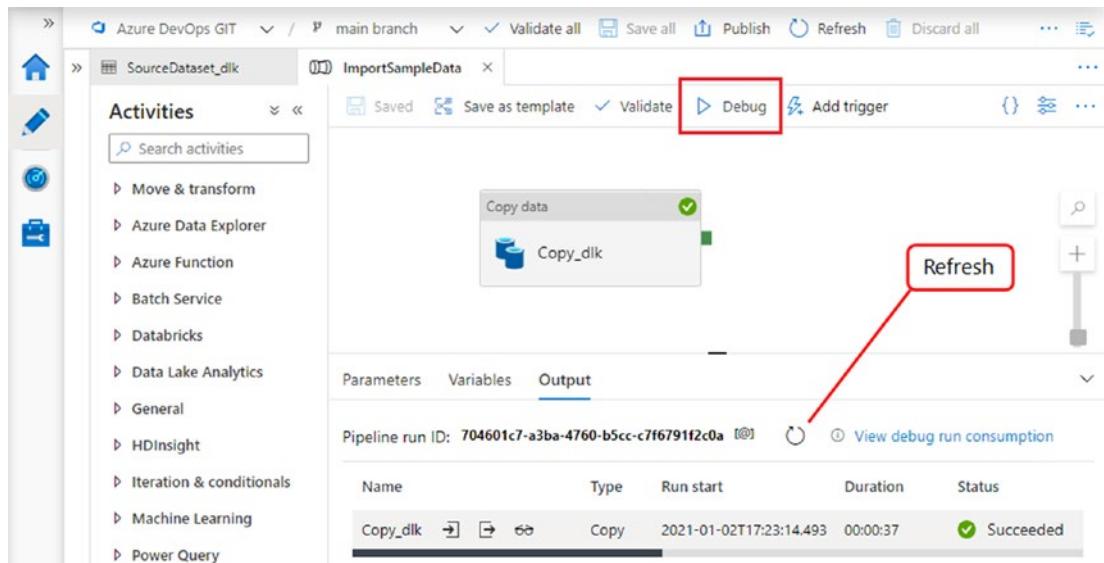


Figure 2-13. Debug controls on the authoring canvas

As soon as you click *Debug*, the tabbed configuration pane below the canvas automatically expands, with the *Output* tab selected. Figure 2-13 shows the tab after the debug run has completed successfully – while the pipeline is still running, you can click the *Refresh* button to update the tab with the latest status information.

At the bottom of the tab is a list of activity runs performed during the pipeline’s execution. In this case, there is only one – the Copy data activity. You can use the icons to the right of an activity’s name to view its inputs, outputs, and performance information. You can view consumption information for the pipeline run as a whole by clicking *View debug run consumption*.

Inspect Execution Results

An unpublished pipeline is still connected to all the same external resources as its published equivalent, and running the pipeline accesses and modifies those same resources. Running your pipeline in debug mode has performed a real data copy from the “landing” container to the “sampledata” container.

Return to Azure Storage Explorer to inspect the “sampledata” container and verify that it now contains a nested folder structure, extracted from the zip file in the “landing” container. You will make use of the data unzipped to this container in the following chapters.

Chapter Review

In this chapter, you created an Azure Storage account and used an ADF pipeline to unzip and copy files from one container into another.

The Copy data activity used by the pipeline is the workhorse of data movement in ADF. In your pipeline, the activity treats files as unstructured blobs, but in Chapter 3 you will explore its handling of structured and semi-structured text file formats, along with other structured datasets.

Key Concepts

This chapter introduced five core concepts for Azure Data Factory:

- **Pipeline:** A data integration workload unit in Azure Data Factory. A logical grouping of activities assembled to execute a particular data integration process.
- **Activity:** Performs a task inside a pipeline, for example, copying data from one place to another.
- **Dataset:** Contains metadata describing a specific set of data held in an external storage system. Pipeline activities use datasets to interact with external data.

- **Linked service:** Represents a connection to an external storage system or external compute resource.
- **Integration runtime:** Provides access to internal compute resource inside Azure Data Factory. ADF has no internal storage resources.

Figure 2-10 illustrates the interaction between these components. Other concepts encountered in this chapter include

- **Debug:** You can run a pipeline interactively from the ADF UX using “Debug” mode. This means that the pipeline definition from the ADF UX session is executed – it does not need to be published to the connected factory instance. During a debugging run, a pipeline treats external resources in exactly the same way as in published pipeline runs.
- **Copy Data tool:** A wizard-style experience in the ADF UX that creates a pipeline to copy data from one place to another. I have presented it in this chapter as a quick way to start exploring the pipeline structure, but in practice you are unlikely to use the tool very often.
- **Azure Storage:** Microsoft’s cloud-based managed storage platform.
- **Storage account:** A storage account is created in order to use Azure Storage services.
- **Storage key:** Storage keys are tokens used to authorize access to a storage account. You can manage an account’s keys in the Azure portal.
- **Blob storage:** General-purpose file (blob) storage, one of the types of storage offered by Azure Storage. Other supported storage types (not described here) include file shares, queues, and tables.
- **Container:** Files in blob storage are stored in containers, subdivisions of a storage account’s blob storage. Blob storage is divided into containers at the root level only – they cannot be nested.
- **Azure Storage Explorer:** An app used to manage Azure Storage accounts, available online and as a desktop application.
- **Bandwidth:** A term used by Microsoft to describe the movement of data into and out of Azure data centers. Outbound data movements incur a fee, sometimes referred to as an *egress charge*.

For SSIS Developers

Most core Azure Data Factory concepts have familiar parallels in SSIS.

ADF Concept	Equivalent in SSIS
Pipeline	Package
Activity	Task (on control flow surface)
Copy data activity	Used in this chapter like a File System Task. In Chapter 3, you will explore more advanced behavior where it acts like a basic Data Flow Task
Linked service	Project-scoped connection manager (with no schema metadata)
Dataset	Schema metadata, stored in various different places (such as a flat file connection manager or OLE DB data flow source)
Integration runtime	SSIS Windows service
Set of pipelines in ADF UX	SSIS project open in Visual Studio
Set of pipelines published to an ADF instance	SSIS project deployed to SSIS catalog

CHAPTER 3

The Copy Data Activity

Data integration tasks can be divided into two groups: those of *data movement* and those of *data transformation*. In Chapter 2, you created an Azure Data Factory pipeline that copied data from one blob storage container to another – a simple data movement using the *Copy data activity*. The Copy data activity is the core tool in Azure Data Factory for moving data from one place to another, and this chapter explores its application in greater detail.

The data movement performed in Chapter 2 was *unstructured*. By choosing the *Binary copy* option when you used the Copy data tool, you told the Copy data activity explicitly to disregard files' internal structure. You didn't define or try to infer any information about data structures within individual files, simply copying them from one place to another as blobs.

One of the Copy data activity's powerful features is its ability to infer and persist data structures – *schemas* – for files. In combination with multi-file handling capabilities, this provides simple but powerful support for loading file-based data into structured data stores like relational databases. In the next section, you will add an Azure SQL Database instance to your Azure resource group to enable you to explore this functionality.

Prepare an Azure SQL Database

An *Azure SQL Database* (or *Azure SQL DB*) is one of a number of Azure-based SQL Server services. It provides a platform as a service (PaaS) database engine, fully managed so that you don't have to worry about administration requirements like backups, patching, and upgrades. It is based on Microsoft's SQL Server database management system, so you can continue to interact with it using familiar client tools like SQL Server Management Studio or Azure Data Studio.

Create the Database

To create your Azure SQL DB:

1. In the Azure portal, create a new resource of type *Azure SQL*.
2. When prompted *How do you plan to use the service?*, choose “Single database” from the *Resource type* dropdown on the *SQL databases* tile. Click the *Create* button below the dropdown on the same tile.
3. Select the subscription and resource group that contains your ADF instance, then choose a name for your database. Below the *Server* dropdown box, click *Create new*.
4. The *New server* blade appears (Figure 3-1). You must provide a globally unique server name, a server admin username, and a password. Locate your server in the same region as your ADF instance, then click *OK*.

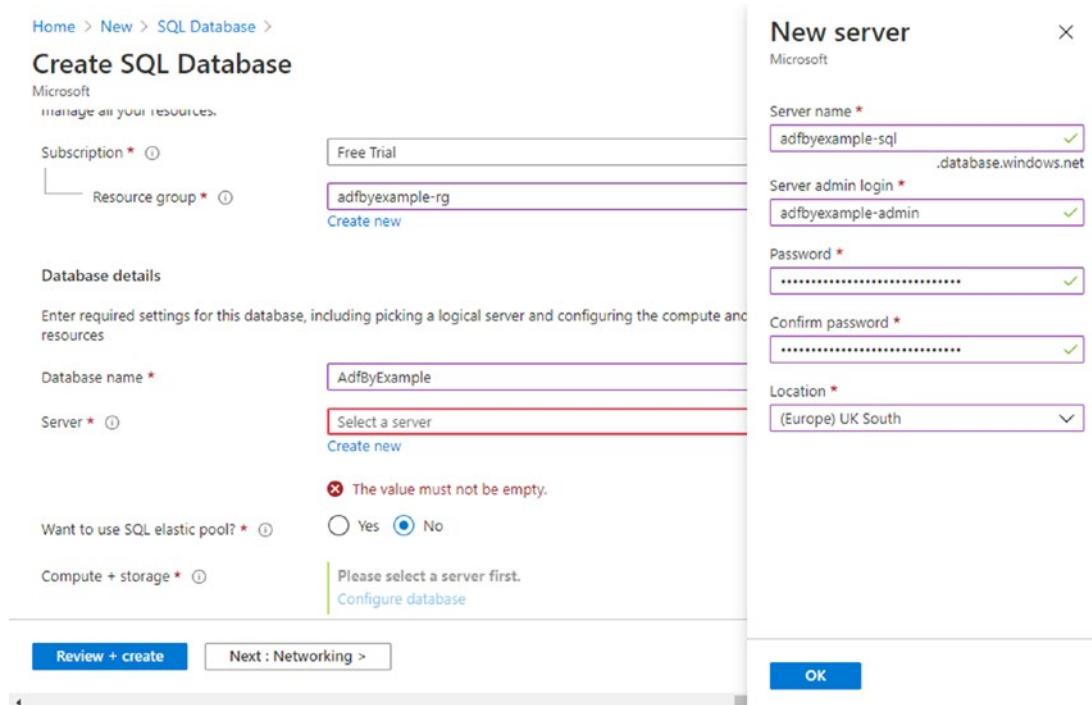


Figure 3-1. Create SQL Database and New server blades

The result of this process is not a traditional SQL Server instance, but a *logical* SQL Server. The logical server is a container for a group of one or more Azure SQL databases and provides services for all the databases in the group, including firewall settings and AAD user administration.

5. Ensure that *Want to use SQL elastic pool?* is set to “No,” then under *Compute + Storage*, click the *Configure database* link.
6. The *Configure* blade for your database controls how resources are allocated to your SQL Database and in what quantity. Under *Compute tier*, ensure that the *Serverless* tile is selected, then click *Apply*. The serverless computer tier does not reserve compute resource in advance and is billed on usage only – this is a low-cost choice for learning and development.

Note By default, databases on the serverless compute tier are paused after one hour of inactivity. This keeps running costs down, but means that, when you return to the database after an inactive period, you may need to wait a few minutes before you can connect successfully.

7. When you have specified server and database details, the *Create SQL Database* blade should look something like Figure 3-2. Click *Review + create*, then after validation click *Create*. (I am purposely bypassing the three remaining tabs – *Networking*, *Additional settings*, and *Tags* – and accepting their default values.)

CHAPTER 3 THE COPY DATA ACTIVITY

The screenshot shows the 'Create SQL Database' blade in the Microsoft Azure portal. At the top, there's a breadcrumb navigation: Home > New > SQL Database >. The title 'Create SQL Database' is displayed above a Microsoft logo. On the right side, there's a close button (X). Below the title, tabs for Basics, Networking, Additional settings, Tags, and Review + create are visible, with 'Basics' being the active tab. A note below the tabs says: 'Create a SQL database with your preferred configurations. Complete the Basics tab then go to Review + Create to provision with smart defaults, or visit each tab to customize.' A 'Learn more' link is also present. The main area is divided into sections: 'Project details' and 'Database details'. In 'Project details', fields for 'Subscription' (set to 'Visual Studio Enterprise Subscription') and 'Resource group' (set to 'adfbByExample-rg', with a 'Create new' option) are shown. In 'Database details', fields for 'Database name' (set to 'AdfbByExample'), 'Server' (set to '(new) adfbByExample-sql (UK South)', with a 'Create new' option), and 'Want to use SQL elastic pool?' (set to 'No') are shown. Under 'Compute + storage', the 'General Purpose' tier is selected, described as 'Serverless, Gen5, 1 vCore, 32 GB storage' with a 'Configure database' link. At the bottom, there are two buttons: 'Review + create' (highlighted in blue) and 'Next : Networking >'.

Figure 3-2. Completed Create SQL Database blade

8. The deployment process creates three new resources in your resource group: a logical SQL Server, a dedicated storage account for database logs, and your new Azure SQL Server database. A notification message is displayed when deployment is complete, including a *Go to resource* button. Click the button to open the portal's SQL database blade (Figure 3-3).

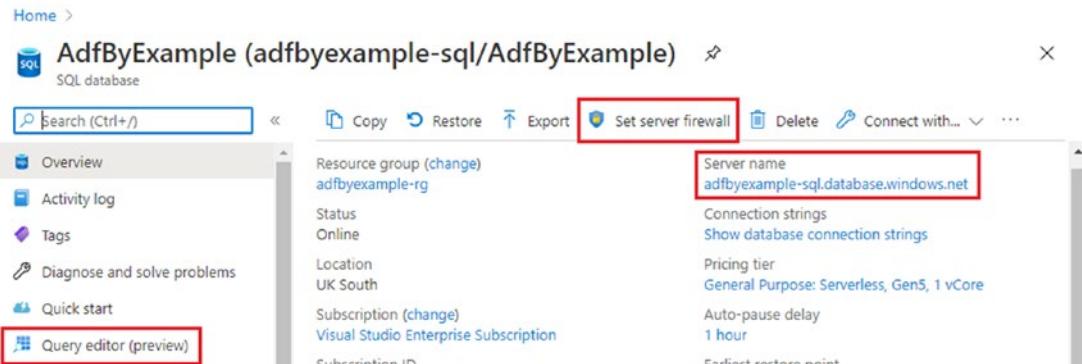


Figure 3-3. Azure portal SQL Database blade, indicating important features

9. Your database is now created, but you will be unable to access it until you have configured a server firewall rule to let you in. At the top of the SQL database blade, click the *Set server firewall* button (indicated in Figure 3-3).
10. At the top of the *Firewall settings* blade, click *+ Add client IP*. Further down the blade, set the *Allow Azure services and resources to access this server* toggle to “Yes” – this will allow your ADF instance to access the server. At the top of the blade, click *Save*.

Create Database Objects

Connection to your Azure SQL DB instance is possible using a number of client tools, for example:

- SQL Server Management Studio (SSMS) installed on your computer
- Azure Data Studio (ADS) installed on your computer
- The online SQL DB Query editor

To use SSMS or ADS, you will need to connect to the server, using its fully qualified domain name, with the username and password you configured in the previous section. The location of the server’s name on the SQL database blade is indicated in Figure 3-3 – in this example, it is `adfbByExample-sql.database.windows.net`. To use the online query editor, click *Query editor (preview)* (also indicated in Figure 3-3) and log in with the same credentials.

CHAPTER 3 THE COPY DATA ACTIVITY

Once you have connected successfully, you will be able to create database objects. Listing 3-1 provides SQL code to create a new table in your database – source code samples used in this book are available from the book’s GitHub repository, located at <https://github.com/Apress/azure-data-factory-by-example>. Copy and paste the SQL statement into the client tool of your choice and run the query.

Listing 3-1. Table creation script for dbo.Sales_LOAD

```
CREATE TABLE dbo.Sales_LOAD (
    RowId INT NOT NULL IDENTITY(1,1)
, Retailer NVARCHAR(255) NULL
, SalesMonth DATE NULL
, Product NVARCHAR(255) NULL
, ManufacturerProductCode NVARCHAR(50) NULL
, SalesValueUSD DECIMAL(18,2) NULL
, UnitsSold INT NULL
, CONSTRAINT PK__dbo_Sales_LOAD PRIMARY KEY (RowId)
);
```

Figure 3-4 shows the online query editor after running the query, with the object explorer sidebar’s *Tables* node expanded to show the result.

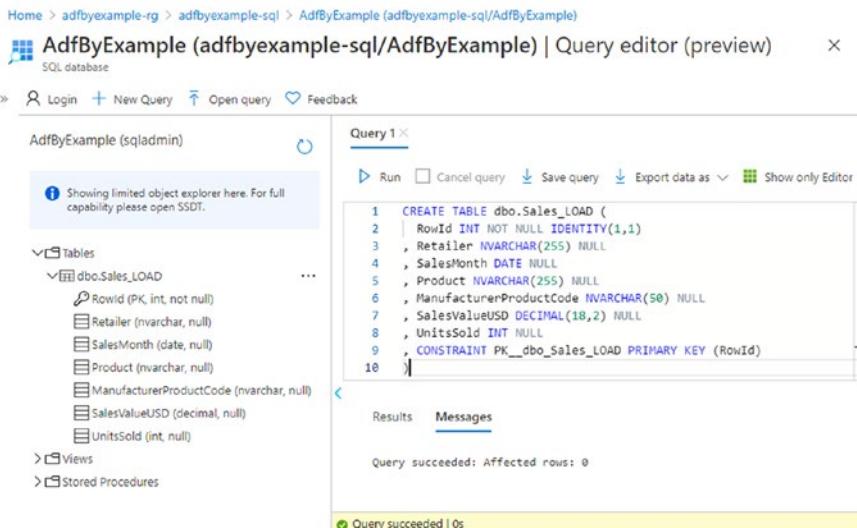


Figure 3-4. Azure SQL DB’s online query editor

Import Structured Data into Azure SQL DB

You have now created all the resources you need in order to start using ADF pipelines to move data between files in blob storage and SQL database tables. In this section, you will learn how to create pipelines that copy data from *structured* data files. Structured data formats are essentially *tabular* – a file contains a number of rows, each of which contains the same number of fields. Comma-separated values (CSV) files are a common structured data format.

Create the Basic Pipeline

You will now create a pipeline to copy CSV file data into your Azure SQL Database. Recall from Chapter 2 that this requires a dataset and linked service at either end of the data movement and a Copy data activity to connect the two.

Create the Database Linked Service and Dataset

Start by creating a linked service for your new Azure SQL DB, as follows:

1. Open the ADF UX and navigate to the management hub. Under *Connections* in the hub sidebar, select *Linked services*.
2. Click the *+ New* button at the top of the *Linked services* page, then select *Azure SQL Database*. Click *Continue*.
3. Complete the *New linked service (Azure SQL Database)* blade (Figure 3-5). Use the account selection method “From Azure subscription,” then select your subscription, server name, and database name. Choose the authentication type “SQL authentication” and supply your database server username and password.

CHAPTER 3 THE COPY DATA ACTIVITY

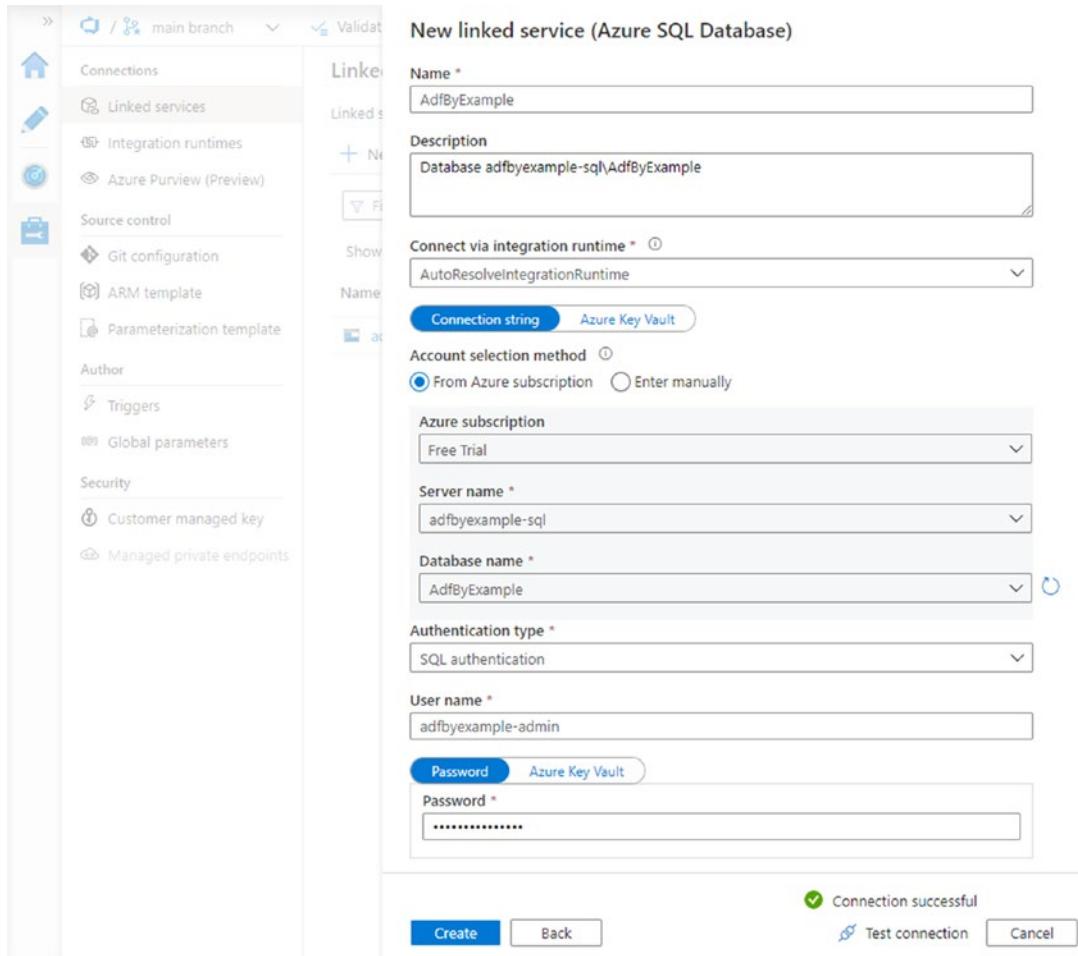


Figure 3-5. New linked service (Azure SQL Database) dialog

4. Use the *Test connection* button to verify your input values, then click *Create*. As when you created your storage account, the SQL DB linked service will be published immediately, to allow ADF to store the server password securely.

Now create a dataset to represent the table `dbo.Sales_LOAD` you created in the previous section:

1. Navigate to the authoring workspace. The *Factory Resources* explorer lists resources defined in your ADF UX session – these are the resource definitions loaded from your collaboration branch in Git. Each resource type section header includes a count of the

number of resources (e.g., 1 pipeline, 2 datasets). Hover over the datasets count to reveal an ellipsis *Actions* button (indicated in Figure 3-6).

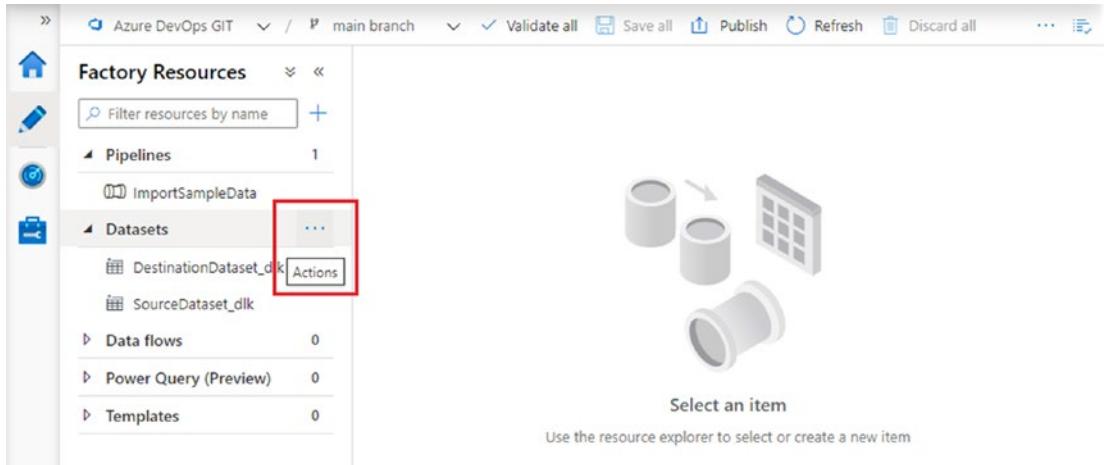


Figure 3-6. Datasets Actions button in the Factory Resources explorer

2. Click the *Actions* button to access the Actions menu. It contains two items: *New dataset* and *New folder*. Folders help organize resources inside your factory and can be nested. Create a new folder and name it “Chapter3.” (You may also like to create a “Chapter2” folder to contain the datasets created by the Copy data tool – if you do this, you can simply drag the existing datasets and drop them into that folder.)
3. The “Chapter3” folder also shows a resource count (currently zero) and Actions menu button. Choose *New dataset* from the Actions menu, then select *Azure SQL Database*. Click *Continue*.
4. Name the dataset “ASQL_dboSalesLoad” and select the new SQL DB linked service from the dropdown. A *Table name* dropdown appears – select the dbo.Sales_LOAD table, then click *OK*.

Tip You may wish to adopt a naming convention for factory resources. My preference is to avoid prefixes that indicate a resource's type, because resource types are usually clear from their contexts. I do however find prefixes indicating a dataset's storage type to be useful (e.g., "ASQL_" for Azure SQL DB datasets).

5. The new dataset opens in a new tab on the authoring canvas (Figure 3-7). On the right-hand side, you see the tabbed *Properties* blade, displaying the dataset's name and description. The blade's *Related* tab allows you to see which pipelines refer to a dataset, a feature also available for other factory resources. You can toggle the blade's visibility using the *Properties* button (slider icon) immediately above it – use the button to close the blade.

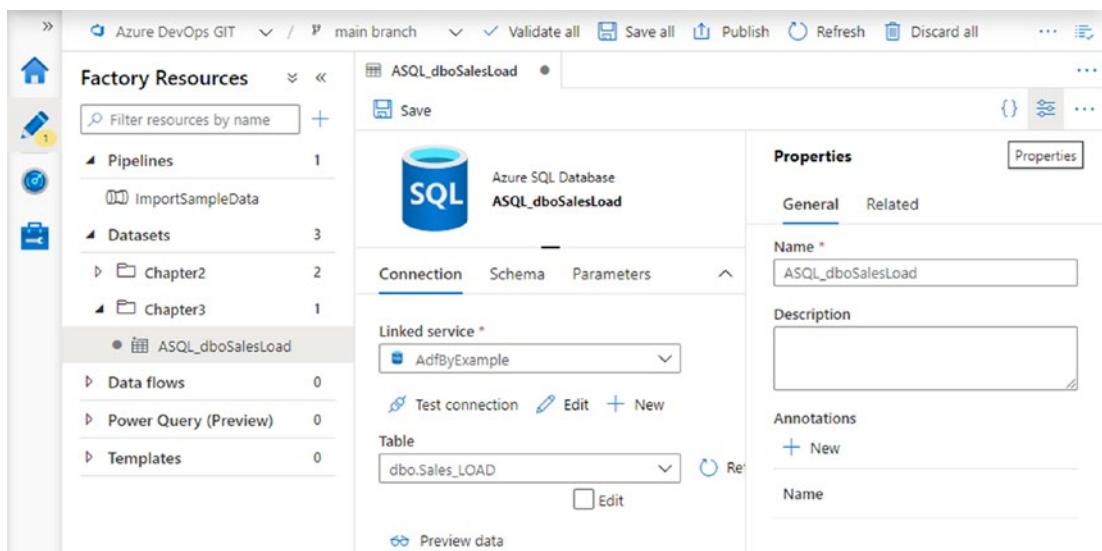


Figure 3-7. Editing a dataset in the authoring workspace

6. To the left of the Properties button, the *Code* button (braces or {} icon) allows you to inspect – and if you wish edit – a factory resource's definition as JSON. Click the button to view the code, then click *Cancel* to close the blade.

7. The tabbed configuration pane below the authoring canvas contains configuration information for the selected resource – in this case, the new dataset. The tabs present in the configuration pane vary between resource types. Select the *Schema* tab to inspect the schema of the `dbo.Sales_LOAD` table.
8. The ADF UX indicates unsaved changes in a number of ways. A gray dot appears to the left of your dataset's name in the Factory Resources explorer and to the right of its name on the authoring canvas tab. In the navigation sidebar, the number “1” in a yellow circle below the pencil icon indicates that your session contains a total of one unsaved change. Click *Save* on the dataset's tabbed pane – this will commit your dataset change and push it to your Git repository – or click *Save all* in the factory header bar to commit and push all changes. Make sure that you save your changes regularly.

Note The ADF UX is more tightly coupled to Git than is the case in other development environments. An ADF UX session has no permanent storage of its own, so changes are saved directly to your Git repository. Every *Save* you make in the ADF UX is a Git commit which is immediately pushed to your hosted repository.

Create a DelimitedText File Dataset

The pipeline you create will load sample CSV data from your blob storage account into an Azure SQL Database table. You created a linked service to connect to the storage account in Chapter 2 – you can reuse it to create a dataset representing a source CSV file.

1. Create a new dataset in the “Chapter3” folder by selecting *New dataset* from the folder's Action menu, then select *Azure Blob Storage*. Click *Continue*.
2. On the *Select format* blade, choose *DelimitedText* and click *Continue*.

3. Name the dataset “ABS_CSV_SweetTreats” – the file you are about to import is a CSV file in Azure Blob Storage (ABS) and contains sales data for a retailer called Sweet Treats. Select your existing blob storage linked service, then click the folder icon to the right of the *File path* fields.
4. Browse into the “sampledata” container until you find the “SampleData” folder. Navigate to the “SweetTreats/Apr-20” subfolder path and select the file “Sales.csv”. Click *OK* to select the file and dismiss the file chooser.
5. Tick the *First row as header* checkbox, then click *OK* to create the dataset.
6. The new dataset opens in another new tab on the authoring canvas – click *Save* to commit and push the dataset definition to Git.

Create and Run the Pipeline

You are now ready to create the pipeline. Before you start, you may wish to move the pipeline from Chapter 2 into a “Chapter2” pipelines folder.

1. In the Factory Resources explorer, create a “Chapter3” folder under *Pipelines*, then select *New pipeline* from the folder’s Actions menu.
2. A new pipeline tab opens with the *Properties* blade displayed. The new pipeline has a default name like “pipeline1” – change the name to “ImportSweetTreatsSales” and add a description, then close the blade.
3. If you need more space on the authoring canvas, collapse the Factory Resources explorer by clicking the left chevrons («) button to the right of the *Factory Resources* heading.
4. To the left of the authoring canvas is the activities toolbox, headed *Activities*. Expand the *Move & transform* group, then drag a *Copy data* activity from the toolbox and drop it onto the canvas. The configuration pane below the canvas automatically expands to allow you to configure the activity.

5. On the *General* tab of the configuration pane, change the name of the Copy data activity to “Import sales data.”

Note Activity names may contain alphanumeric characters, hyphens, underscores, and spaces, but names cannot start or end with a space. Activity names are validated by the ADF UX as you type, so when you add a space character, you will receive an immediate validation error – this disappears when you enter the next valid nonspace character.

6. On the *Source* tab, select the “ABS_CCSV_SweetTreats” dataset, and on the *Sink* tab, choose the dataset “ASQL_dboSalesLoad”. Azure Data Factory refers to pipeline data destinations as *sinks*.
7. In the canvas toolbar (above the authoring canvas), click *Save* to commit and push your changes, then *Debug* to run the pipeline.

Pipeline run information is displayed on the *Output* tab of the pipeline’s configuration pane (shown in Figure 3-8). Hovering over the name of the Copy data activity on the *Output* tab reveals three icons for accessing information on the activity’s *Input*, *Output*, and *Details*.

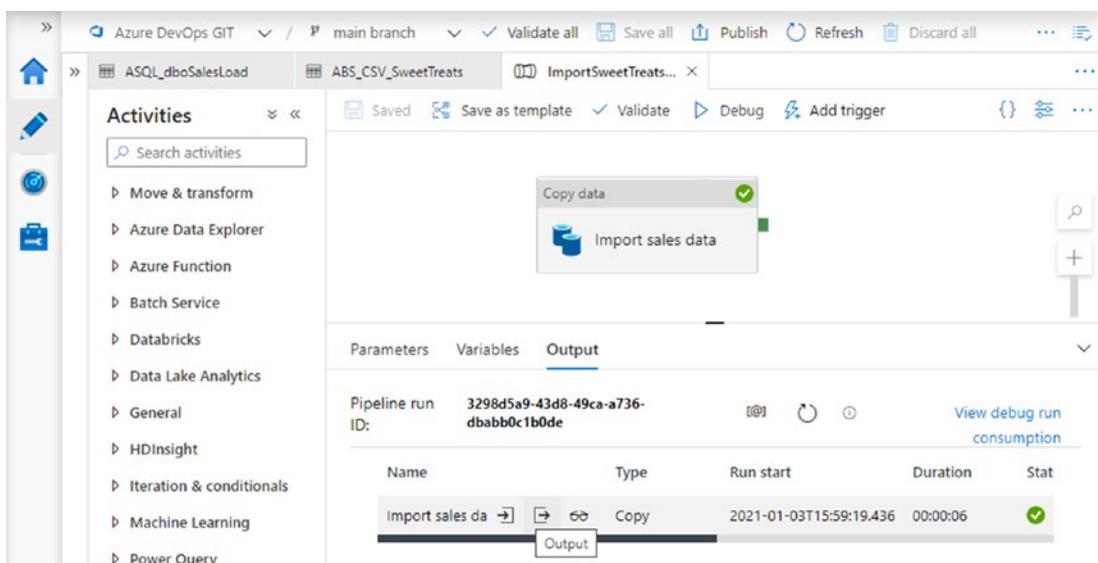
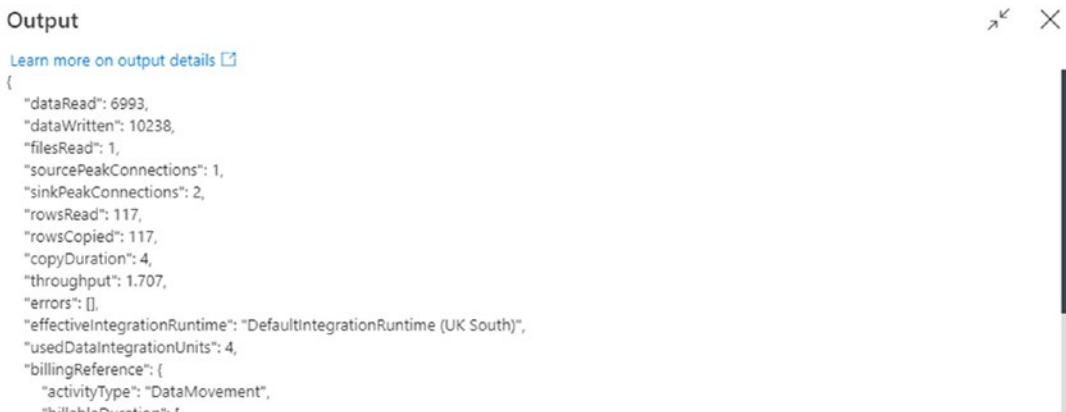


Figure 3-8. Authoring canvas with pipeline configuration pane’s Output tab

CHAPTER 3 THE COPY DATA ACTIVITY

The Copy data activity's output is a JSON object containing data about the activity run – the first part of an example output is shown in Figure 3-9. Notice particularly the attributes:

- **filesRead:** The number of files read by the Copy data activity run (in this example, one file)
- **rowsRead:** The total number of source rows read by the activity run
- **rowsCopied:** The number of sink rows written by the activity run



The screenshot shows a JSON object representing the output of a Copy data activity run. The object contains various performance metrics and context information. Key values include:

- "dataRead": 6993
- "dataWritten": 10238
- "filesRead": 1
- "sourcePeakConnections": 1
- "sinkPeakConnections": 2
- "rowsRead": 117
- "rowsCopied": 117
- "copyDuration": 4
- "throughput": 1.707
- "errors": []
- "effectiveIntegrationRuntime": "DefaultIntegrationRuntime (UK South)"
- "usedDataIntegrationUnits": 4
- "billingReference": { "activityType": "DataMovement", "billableDuration": 1 }

Figure 3-9. First part of a Copy data activity run's output JSON object

Verify the Results

Return to the SQL client of your choice and verify that the `dbo.Sales_LOAD` table now contains the number of rows reported by the Copy data activity run.

You may wish also to inspect the contents of the April 2020 “Sales.csv” file to verify that its contents have been correctly uploaded. The first few rows of the file are shown in Listing 3-2. In particular, notice the field names in the first row of the file – they match column names in the database table exactly. During pipeline creation, you did not provide any mapping from CSV fields to table columns, but the Copy data activity inferred the mapping automatically using matching column names.

Listing 3-2. The first few rows of the Sales.csv file

```
SalesMonth,Retailer,Product,SalesValueUSD,UnitsSold
"01-Apr-2020","Sweet Treats","Schnoogles 8.81oz",3922.31,409
"01-Apr-2020","Sweet Treats","Creamies 10.57oz",3057.18,502
"01-Apr-2020","Sweet Treats","Caramax 6.59oz",1147.37,443
```

For SSIS developers In Chapter 2, I compared the Copy data activity to an SSIS File System Task, but here it provides functionality like a simple Data Flow Task containing two components: a flat file data source and a SQL Server destination.

Process Multiple Files

The sample dataset includes six months of sales data for Sweet Treats. Each month's file has the same name – “Sales.csv” – and like April 2020’s file is found in a folder named for the month to which it relates. The dataset you created in the previous section identifies April’s data file specifically, but the Copy data activity supports wildcard behavior that enables you to specify multiple files for processing.

1. On the ADF UX authoring canvas, select your sales data import pipeline’s Copy data activity. The configuration pane for the activity automatically expands below the canvas.

Tip When selected, an activity on the authoring canvas displays additional activity-specific functions including *Delete*, *Clone*, and *View source code*. The *Add output* function is introduced in Chapter 6.

2. Select the *Source* tab, then change the *File path type* to “Wildcard file path.” This means that the original file path specified in the dataset will no longer be used. Fields for Wildcard paths are displayed (Figure 3-10).

CHAPTER 3 THE COPY DATA ACTIVITY

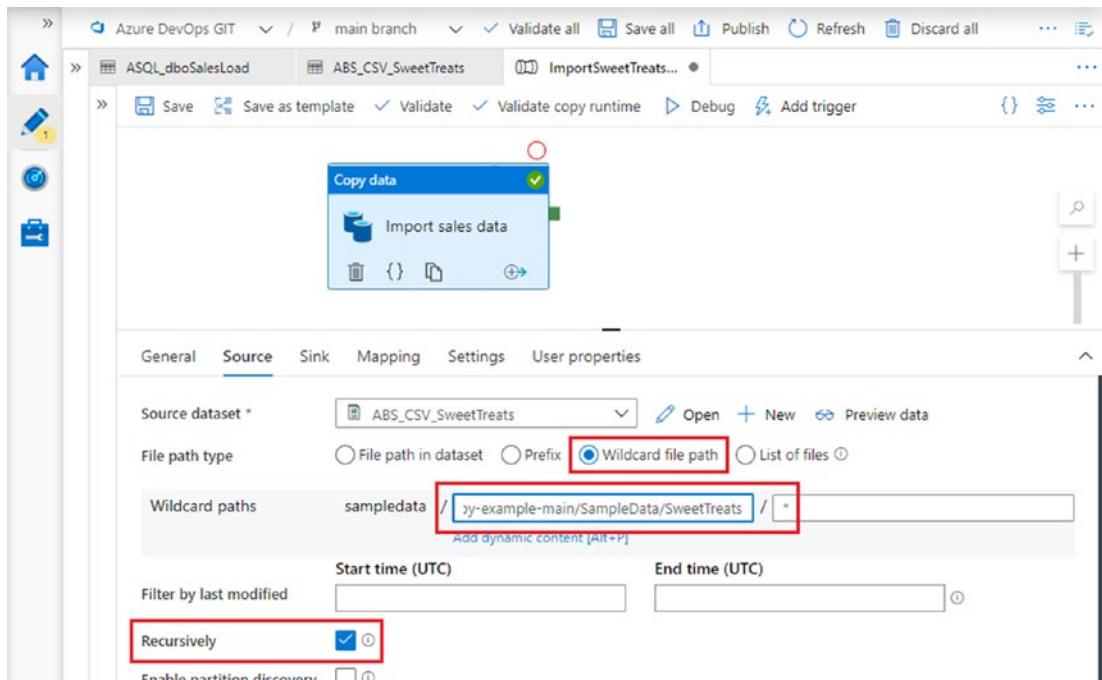


Figure 3-10. Wildcard file path in the Copy data activity source

3. The container portion of the blob storage path is populated automatically from the “ABS_CCSV_SweetTreats” dataset. In the *Wildcard folder path* field, enter “azure-data-factory-by-example-main/SampleData/SweetTreats” (the path to the “SweetTreats” folder). The path ends with “SweetTreats” and has no leading or trailing slash character.

Tip Unlike Windows environments, Azure Blob Storage file paths are case-sensitive, and paths are delimited using the *forward-slash* character (“/”).

4. In *Wildcard file name*, enter an asterisk (“*”) – this indicates that all files found in the “SweetTreats” folder are to be imported.
5. Scroll down and ensure that the *Recursively* checkbox is ticked. This indicates that files matching the wildcard path in any nested subfolder of “SweetTreats” are to be included.

Save your changes, then click *Debug* to run the pipeline. When complete, look at the Copy data activity's Output JSON. This time, you will notice that the activity read a total of 6 files and 684 rows – all of the “Sales.csv” files for the six months between April and September 2020.

Truncate Before Load

Return to your SQL client of choice and run the query in Listing 3-3 to count the number of records loaded for each month.

Listing 3-3. Count records loaded per month

```
SELECT
    Retailer
    , SalesMonth
    , COUNT(*) AS [Rows]
FROM [dbo].[Sales_LOAD]
GROUP BY
    Retailer
    , SalesMonth;
```

As expected, there are now rows present for each of the six months, but notice that the row count for April is roughly twice that of the other months. This is because you have loaded April’s data twice – once when you ran your pipeline for the first time, then a second time while loading all six months’ data. To use the dbo.Sales_LOAD table correctly, it must be truncated before each load activity.

This requirement can be supported using Copy data activity functionality provided for SQL database sinks. On the activity’s *Sink* tab in the ADF UX, add this SQL command to the *Pre-copy script* field (Figure 3-11):

```
TRUNCATE TABLE dbo.Sales_LOAD
```

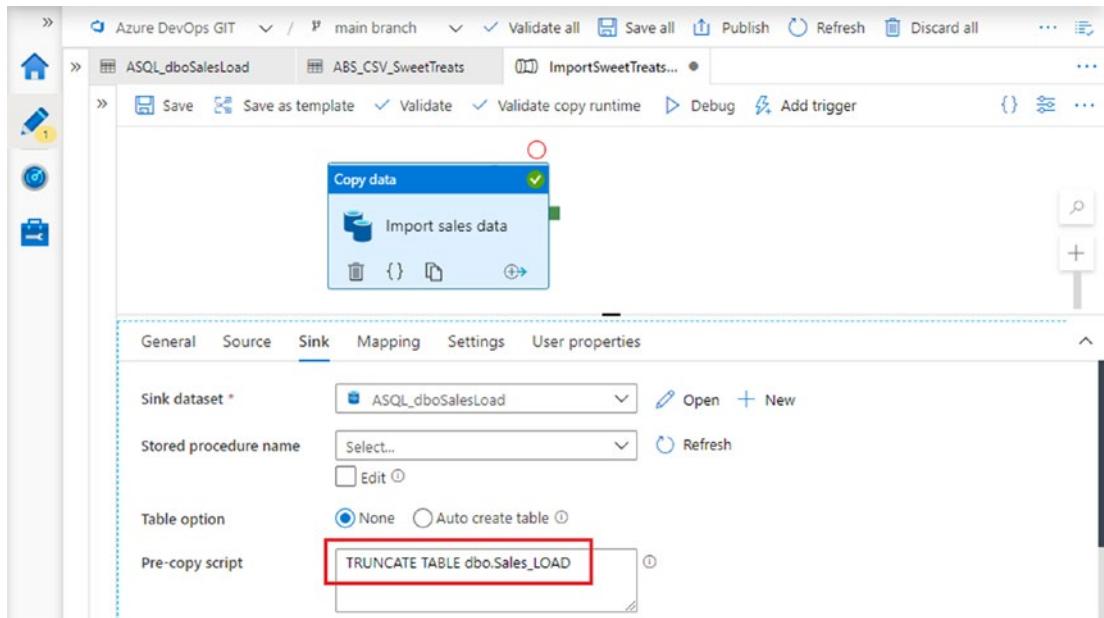


Figure 3-11. Pre-copy script in the Copy data activity sink

This SQL command will be executed at the database containing the sink dataset every time the Copy data activity runs, before any data is copied.

Run the pipeline again by clicking *Debug* – notice that debug mode allows you to run your revised pipeline without even having to save your changes – then check the table's row counts using the script in Listing 3-3. This time, you will find that a similar number of rows is reported for each of the six months.

Map Source and Sink Schemas

The pipeline you created in the previous section relies on the Copy data activity's ability to infer column mappings by matching names in the source files and sink table. In this section, you will discover what happens when source and sink column names do not match and how to handle that situation.

Create a New Source Dataset

The pipeline you're about to create will load data for a different retailer called Candy Shack. Its source data is also supplied in CSV files, so you can base its source dataset on "ABS_CSV_SweetTreats" by *cloning* it and editing the clone.

1. Figure 3-12 shows the ADF UX with the *Factory Resources* explorer expanded. Hovering over the right-hand end of the "ABS_CSV_SweetTreats" dataset (below the "Chapter3" folder pipeline count) reveals the ellipsis *Actions* button – click it to access the dataset Actions menu, then select *Clone*.

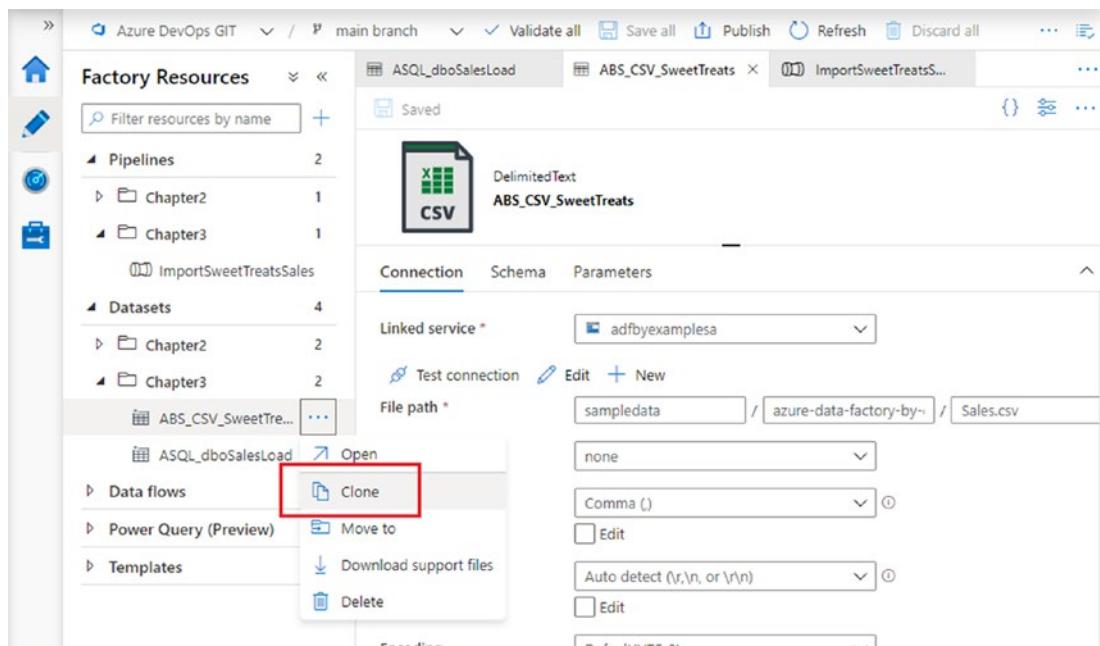


Figure 3-12. Dataset Actions menu

2. *Clone* creates a copy of the original dataset and opens it in the ADF UX. The new dataset is automatically given a unique name based on the original – when it opens, the *Properties* blade appears so that you can rename it. Change its name to "ABS_CSV_CandyShack" and close the blade.

3. Edit the dataset's *File path* by browsing to the "CandyShack" folder nested below the "sampledata" container and choosing file "2020-04.csv".
4. Click *Save all* to save the new dataset and any unsaved changes.

Create a New Pipeline

The pipeline itself is very similar to the one you created to load Sweet Treats data. Clone the "ImportSweetTreatsSales" pipeline in the same way that you cloned the dataset, and amend it as follows:

1. Name the new pipeline "ImportCandyShackSales."
2. Change the Copy data activity's *Source dataset*; using the dropdown list, select "ABS_CSV_CandyShack".
3. Still on the *Source* tab, edit *Wildcard paths*, replacing the reference to the folder "SweetTreats" with "CandyShack" (making sure that you leave the rest of the path intact).
4. Click *Debug* to run the pipeline. When pipeline execution stops, you will discover that the Copy activity run has failed.
5. As you discovered in Chapter 2, pipeline run information appears on the *Output* tab of the pipeline's configuration pane. Because the Copy data activity run has failed, an additional *Error* button (speech bubble icon) now appears when hovering over its name. Click *Error* to view information about why the activity failed.

The error popup window describing the activity failure is shown in Figure 3-13. Notice particularly the text "The column SkuCode is not found in target side." This indicates that a field called "SkuCode" was found in the source data, but that the sink table contains no corresponding field of the same name.

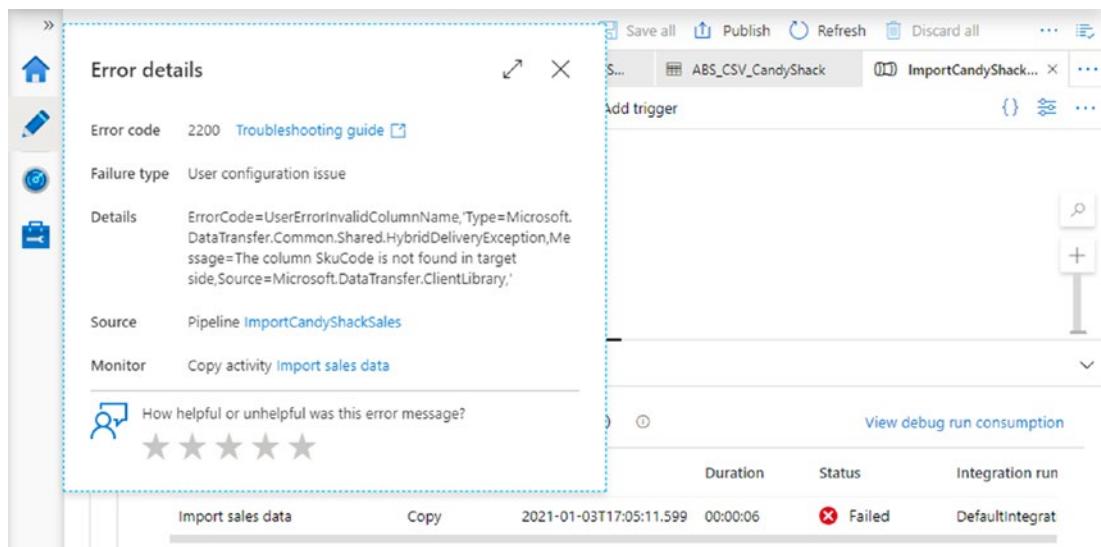


Figure 3-13. Copy data activity error details

Configure Schema Mapping

For this Copy data activity to succeed, you must first map source columns to sink columns explicitly, as follows:

1. Select the *Mapping* tab on the Copy data activity's configuration pane and click the *Import schemas* button. Wait a few moments while the ADF UX retrieves schema information from a Candy Shack source file and the database sink table.
2. The ADF UX attempts to map imported source and sink columns automatically, displaying error or warning messages where necessary. Figure 3-14 shows the initial results of importing the two schemas.

CHAPTER 3 THE COPY DATA ACTIVITY

The screenshot shows the 'Copy Data' activity configuration window. At the top, there are buttons for 'Import schemas', 'Preview source', 'New mapping', 'Clear', 'Reset', and 'Delete'. Below this is a table with four columns: Source, Type, Destination, and Type. The Source and Destination columns contain dropdown menus for selecting field names. The Type columns show the data types of the fields. A red box highlights the 'Select or edit column' dropdown in the first row. A red error message 'Mapping source is empty.' is displayed next to it. A yellow warning message 'Copying from column SalesMonth to column SalesMonth may have data truncation.' is also present. The table rows represent the following mappings:

Source	Type	Destination	Type
Select or edit column		RowId	int
Retailer	String	Retailer	nvarchar
SalesMonth	String	SalesMonth	date
Product	String	Product	nvarchar
Select or edit column		ManufacturerProduct...	nvarchar
Select or edit column		SalesValueUSD	decimal
UnitsSold	String	UnitsSold	int

At the bottom left, there is a link 'Add dynamic content [Alt+P]'

Figure 3-14. Imported Candy Shack source and database sink schemas

3. When you hover over a column mapping, additional controls appear to the right, enabling you to delete it or to add a new mapping. In this example, the destination [RowId] database column is an automatically generated integer identity, so the mapping can be deleted – use the trash can icon to the right of the mapping to do so. The source file contains no field corresponding to [ManufacturerProductCode], so remove this mapping too.
4. The dropdown lists for each field mapping allow you to select participating source and destination field name mappings. The source field corresponding to the destination [SalesValueUSD] column is called simply “SalesValue” – select it from the list.
5. Click *Debug* to run the pipeline again, this time successfully. Use Listing 3-3 in your SQL client to verify that six months’ sales data for Candy Shack has been loaded into dbo.Sales_LOAD. Notice that the previously loaded Sweet Treats data has been removed by the activity’s Pre-copy script.

Import Semi-structured Data into Azure SQL DB

Structured, tabular data files – such as the CSV files described in the previous section – resemble database tables in structure. In contrast, nontabular or *semi-structured* data formats use embedded metadata to identify data elements and frequently contain nested components. Common semi-structured data formats include JSON and XML; in this section, you will author a pipeline using the Copy data activity’s features for handling JSON data.

The sample data for loading is a collection of monthly sales data reports for a confectionery retailer called Sugar Cube. The Sugar Cube sales report format has the following features, as shown in Listing 3-4:

- The JSON document has a Month field identifying the month to which the report relates, a Company field identifying Sugar Cube, and a Sales field containing a list of individual product sales summaries.
- Each product sales summary object identifies the product by name and the manufacturer’s code, reporting the quantity sold (Units) and the total sales revenue (Value).

Listing 3-4. Start of a Sugar Cube sales report file

```
{
  "Month": "01-apr-2020",
  "Company": "Sugar Cube",
  "Sales": [
    {
      "Product": "Schnoogles 8.81oz",
      "ManufacturerProductCode": "CS-20147-0250",
      "Units": 745,
      "Value": 6995.55
    },
  ]
```

Create a JSON File Dataset

The new pipeline will load sample JSON data from your blob storage account into Azure SQL DB. In the previous section, you created the “ABS_CSV_CandyShack” dataset by cloning “ABS_CSV_SweetTreats”, but in this case you must create a new dataset from

scratch. A dataset's file type can only be specified when the dataset is created – to load JSON data, you must first create a JSON blob storage dataset.

1. On the “Chapter3” folder’s *Actions* menu in the *Factory Resources* explorer, click *New dataset*.
2. Select the *Azure Blob Storage* data store and click *Continue*.
3. On the *Select format* blade, choose *Json*. Click *Continue*.
4. Name the dataset “ABS_JSON_SugarCube” and select your existing storage account linked service. When the *File path* option appears, browse to the “SugarCube” subfolder in the “sampleddata” container and find the file “April.json”. Click *OK* to select the file.
5. Click *OK* to create the new dataset, then save it.

Create the Pipeline

Create a new pipeline named “ImportSugarCubeSales,” either by cloning one of your existing “Chapter3” pipelines or from scratch. Add a Copy data activity to the pipeline and configure the activity’s source and sink as follows:

- **Source:** Choose the “ABS_JSON_SugarCube” dataset and select *File path type* “Wildcard file path.” Under *Wildcard paths*, set the *Wildcard folder path* to “azure-data-factory-by-example-main/SampleData/SugarCube” (omitting the year and quarter subfolders) and specify “*.json” as *Wildcard file name*. Ensure that the *Recursively* checkbox is ticked.
- **Sink:** Choose your Azure SQL DB dataset and ensure that *Pre-copy script* is configured to truncate the destination database table.

Configure Schema Mapping

Select the Copy data activity’s *Mapping* configuration tab and click *Import schemas*.

Figure 3-15 shows the initial mapping, generated automatically by the ADF UX. The imported source schema includes information about nested JSON objects. “Sales” is correctly identified as being an array of objects having four fields: “Product,” “ManufacturerProductCode,” “Units,” and “Value.”

The screenshot shows the Data Flow Designer interface with the following configuration:

Name	Type	Collection reference	Column name	Include
Month	abc string		Edit column	<input checked="" type="checkbox"/>
Company	abc string		Edit column	<input checked="" type="checkbox"/>
Sales	[] array	<input type="checkbox"/>		
Product	abc string		Product	<input checked="" type="checkbox"/>
ManufacturerProductCode	abc string		ManufacturerProduct...	<input checked="" type="checkbox"/>
Units	123 integer		Edit column	<input checked="" type="checkbox"/>
Value	123 number		Edit column	<input checked="" type="checkbox"/>

Figure 3-15. Imported Sugar Cube source and database sink schemas

The “Product” and “ManufacturerProductCode” source fields have already been mapped to matching destination column names. Map the remaining JSON field names to their corresponding database table columns:

- Month → SalesMonth
- Company → Retailer
- Units → UnitsSold
- Value → SalesValueUSD

Now run the pipeline. When you inspect the results in your SQL client, you will discover that you have loaded exactly six rows.

Set the Collection Reference

The nested structure of JSON objects allows them to represent normalized data structures. The Sugar Cube sales report format contains the following relationships:

- Each report file contains exactly one monthly sales report JSON document.
- Each monthly sales report contains many product sales summaries.

The *Collection reference* property on the Copy data activity's *Mapping* configuration tab controls what an output row of data represents: in this case, either an individual product sales summary or the entire monthly report document.

By default, *Collection reference* is the JSON document root. The six rows loaded when you ran the pipeline correspond to the six Sugar Cube report files, but you will notice that some lower-level data (product, sales details) was also loaded into the six rows. To understand this, enable the *Advanced editor* on the *Mapping* tab (Figure 3-16) – this shows the underlying JSON path expressions used to extract output field values.

The screenshot shows the 'Advanced editor' mode of the mapping configuration. At the top, there are buttons for 'Import schemas', 'New mapping', 'Clear', 'Reset', 'Delete', and a toggle switch for 'Advanced editor' which is turned on. Below these are fields for 'Collection reference' (a dropdown menu) and 'Map complex values to string' (a checked checkbox). The main area is a table mapping source JSON paths to destination columns:

Source	Type	Destination	Type
<code>\$['Month']</code>		<code>SalesMonth</code>	date
<code>\$['Company']</code>		<code>Retailer</code>	nvarchar
<code>\$['Sales'][0]['Product']</code>		<code>Product</code>	String
<code>\$['Sales'][0]['Manufact...</code>		<code>ManufacturerProduct...</code>	String
<code>\$['Sales'][0]['Units']</code>		<code>UnitsSold</code>	int
<code>\$['Sales'][0]['Value']</code>		<code>SalesValueUSD</code>	decimal

Figure 3-16. Advanced mapping editor showing JSON path expressions

The expression for the “UnitsSold” field is `$['Sales'][0]['Units']`, which means “use the Units field from the *first* product sales summary element of the Sales array.”

To specify the correct collection reference, disable the *Advanced editor* and tick the checkbox in the “Sales” row (or select “Sales” from the *Collection reference* dropdown above the list of column mappings). Run the pipeline again, then use your SQL client to verify that 100–120 rows have been loaded for each month.

The Effect of Schema Drift

Use your SQL client to look at the Sugar Cube data loaded for September 2020 (using a query such as the one shown in Listing 3-5).

Listing 3-5. Select September's Sugar Cube data

```
SELECT *
FROM [dbo].[Sales_LOAD]
WHERE Retailer = 'Sugar Cube'
AND SalesMonth = '2020-09-01';
```

A close inspection of the query results reveals that all of September's data is lacking a name value in the database table's [Product] column. This has been caused by a small change to the sales report's JSON structure, introduced in that month. Listing 3-6 shows the start of the September file, in which you can see that the product name field, formerly called "Product," is now called "Item."

Listing 3-6. Start of the September 2020 Sugar Cube sales report file

```
{
  "Month": "01-sep-2020",
  "Company": "Sugar Cube",
  "Sales": [
    {
      "Item": "Schnoogles 8.81oz",
      "ManufacturerProductCode": "CS-20147-0250",
      "Units": 643,
      "Value": 6745.07
    },
  ]
```

The result of renaming the "Product" field is that it appears simply to have been omitted, which ADF has accepted without error. JSON handling by the Copy data activity is necessarily tolerant of missing fields, because optional elements may not always be present in a JSON object.

There is no simple fix here, but it illustrates the point that you cannot rely on JSON schema mappings to detect schema drift. Unlike when loading a structured data file, schema drift may not cause failure. In Chapter 6, you will add a basic data quality check on loaded data to improve pipeline robustness.

Understanding Type Conversion

When moving or transforming data, integration tasks must mediate differences in type systems used by source and sink data stores. The source field types shown in Figure 3-15 are JSON data types, while the corresponding sink field types, visible in Figure 3-16, are native to SQL Server. Data conversion between these two type systems takes place via a third: Azure Data Factory *interim data types* (IDTs).

The Copy data activity performs a three-step type conversion to translate data from a source type system to the sink type system:

1. Convert the native source type into an ADF interim type.
2. Convert the interim type to a possibly different interim type compatible with the sink type.
3. Convert the sink-compatible interim type into a native sink type.

This knowledge isn't essential to use the Copy data activity, but it may help you to understand the behavior of schema mappings between different formats. For example, CSV files have no consistent type system, so ADF treats all incoming fields as strings and automatically converts them to the String interim data type (as can be seen in the source types shown in Figure 3-14).

IDTs allow Azure Data Factory to support an ever-growing range of supported data stores in a scalable way. Without IDTs, adding support for a new kind of data store would require Microsoft to specify type conversions between the new store and every other existing kind of data store. This requirement would grow in size with every addition to the list of supported data stores. Using IDTs means that, to extend support to new kinds of data store, ADF needs only to be able to convert the new store's data types to and from its own interim data type system.

For SSIS developers ADF's approach to mediating type conversions via an intermediate type system will be familiar from SSIS Data Flow Tasks. Integration Services data types (e.g., DT_STR, DT_WSTR, and DT_I4) perform the same role in SSIS, providing extensible support for arbitrary pairings of source and destination type systems.

Transform JSON Files into Parquet

So far, in this chapter, you have used ADF's ability to read data from CSV and JSON files and its ability to write data to Azure SQL DB tables. The fact that the Copy data activity's source and sink are both described using ADF *datasets* means that you can just as easily extract data from a SQL database and write it out to CSV or JSON files. Similarly, you could use the Copy data activity to read data from CSV files and write it out as JSON – or in fact to read data from *any* supported dataset type and write it to any other.

Apache *Parquet* is a compressed, column-oriented structured data storage format. Analytics applications that process large numbers of rows benefit from a column-oriented format, because data in columns outside the scope of analysis need not be read. Individual columns often contain a few distinct values, allowing column data to be heavily compressed for efficient storage and retrieval. For these reasons, Parquet is frequently the format of choice for data lake storage of tabular data.

The confectionery retailer Handy Candy reports sales data in the form of individual sales transaction JSON messages. Listing 3-7 contains one such message object. In this section, you will create a pipeline to ingest these messages and output them as Parquet. This ingestion pattern is frequently used to integrate new data into a data lake, but in this case you will simply emit the Parquet file to Azure blob storage.

Listing 3-7. Handy Candy sales transaction message

```
{
  "TransactionId": "0C90CC54-392B-4322-BB23-B4B34CE403D9",
  "TransactionDate": "2020-06-08",
  "StoreId": "114",
  "Items": [
    {
      "Product": "Boho 2.82oz",
      "Price": 2.89
    }
  ]
}
```

Create a New JSON Dataset

In order to be able to import the schema of the Handy Candy message file, you need a JSON dataset that refers to a message file in the “HandyCandy” subfolder of the SampleData folder.

1. Either create a new JSON dataset using your blob storage account or clone “ABS_JSON_SugarCube”. Name the new dataset “ABS_JSON_HandyCandy”.
2. Set the dataset’s *File path* by selecting one of the message files in the “sampledata” container’s “HandyCandy” subfolder.

Create a Parquet Dataset

Create a new dataset in the “Chapter3” datasets folder with the following properties:

- **Data store:** Azure Blob Storage.
- **File format:** Parquet.
- **Name:** Enter “ABS_PAR_HandyCandy”.
- **Linked service:** Choose your existing blob storage linked service.
- **File path:** Specify *Container* “output,” *Directory* “datalake,” and *File* “HandyCandy.parquet”.

Before you click *OK* on the *Set properties* blade, ensure that *Import schema* is set to “None.”

Tip If no “output” container exists when the pipeline is executed, the pipeline will create one automatically.

Create and Run the Transformation Pipeline

Create a new pipeline named “IngestHandyCandyMessages” in the “Chapter3” pipelines folder, then drag a Copy data activity onto the authoring canvas. Configure the activity as follows:

- **Source:** Choose the “ABS_JSON_HandyCandy” dataset and select *File path type* “Wildcard file path.” Under *Wildcard paths*, set *Wildcard folder path* to “azure-data-factory-by-example-main/SampleData/HandyCandy” and *Wildcard file name* to “*.json”.
- **Sink:** Choose the “ABS_PAR_HandyCandy” dataset and set *Copy behavior* to “Merge files.” This instructs ADF to merge incoming JSON message files into a single Parquet output file.
- **Mapping:** On the Mapping tab, click *Import schemas*. Verify that the imported schema matches the message structure shown in Listing 3-7. Select the “Items” field as *Collection reference*, and notice that the ADF UX has automatically provided field names for the new file – no further changes are necessary unless you want to refine sink field names or types. Figure 3-17 shows an updated mapping in which automatic field names have been preserved but additional type information has been provided.

The screenshot shows the Mapping tab in the Azure Data Factory authoring interface. At the top, there are buttons for Import schemas, New mapping, Clear, Delete, and Advanced editor. Below these, the Collection reference is set to \${'Items'} and there is a checkbox for Map complex values to string. The main area displays a table mapping JSON fields to Parquet columns:

Name	Type	Collection reference	Column name	Type	Include
TransactionId	abc string		TransactionId	abc Guid	<input checked="" type="checkbox"/>
TransactionDate	abc string		TransactionDate	abc DateTime	<input checked="" type="checkbox"/>
StoreId	abc string		StoreId	12s Int16	<input checked="" type="checkbox"/>
Items	[] array	<input checked="" type="checkbox"/>			
Product	abc string		Product	abc String	<input checked="" type="checkbox"/>
Price	123 number		Price	1.2f Single	<input checked="" type="checkbox"/>

Figure 3-17. Prepared mapping for the JSON to Parquet transformation

Run the pipeline. When execution is complete, inspect your storage account's output container to verify that the Parquet file is present. Compression of columnar data means that the output file is about one third of the total size of the input JSON files.

If you wish to verify the contents of the Parquet file, create a pipeline containing a Copy data activity with the "ABS_PAR_HandyCandy" dataset as the source. You can inspect source data by copying it into a database table or CSV file sink or by using the *Preview data* option on the Copy data activity's *Source* configuration tab.

Performance Settings

In addition to the tabs you have already explored, the Copy data activity's configuration pane features a *Settings* tab. Two of these settings relate to activity performance – others are covered later in the book.

Data Integration Unit

The main focus of this chapter has been how to copy data using the Copy data activity, without paying a great deal of attention to performance. This has not been a problem because the sample datasets are small, but when working with larger, real-world datasets, you may be able to improve copy performance by adjusting default performance characteristics.

The Copy data activity power is measured in *data integration units* (DIUs), a single measure that combines CPU, memory, and network usage. You can increase the power allocated to a Copy data activity by increasing the number of DIUs available to it, specified using the *Data integration unit* option on the *Settings* tab.

The default setting for the Data integration unit is "Auto," which means that the number of DIUs allocated is determined automatically, based on information about your source and sink data stores. If you wish, you can increase the number of allocated DIUs above the default, but bear in mind that doing so will increase the financial cost of executing the activity. Conversely, the default DIU value for many scenarios is four, so you may wish to reduce your execution cost – particularly in learning and development scenarios – by limiting the number of DIUs to its minimum value of two.

You can see the number of DIUs used by a Copy data activity execution in the `usedDataIntegrationUnits` field of the execution's JSON output (as shown in Figure 3-9). More information about DIUs and Copy data activity performance is available at <https://docs.microsoft.com/en-us/azure/data-factory/copy-activity-performance-features>.

Degree of Copy Parallelism

The *Degree of copy parallelism* option allows you to override the Copy data activity's default degree of parallelism. The degree of parallelism actually used at runtime appears in the `usedParallelCopies` field of the execution's JSON output. You can see how this varies even within your sample data pipelines – the Handy Candy ingestion pipeline has to process many more files and exhibits a higher degree of parallelism. This kind of automatic scaling out behavior is a key advantage of serverless services such as Azure Data Factory.

The default degree of parallelism is also determined automatically and varies based on information about your source and sink data stores. While it might seem tempting to override the default, Microsoft's advice is that the best data throughput is usually achieved with the default behavior. As for DIUs, a more common use case might be to throttle parallelism, to avoid overloading a source data store.

Chapter Review

At the beginning of this chapter, I described the Copy data activity as being the core tool for data *movement* in Azure Data Factory. While this is true, you will now have gained an appreciation of some of the data *transformation* capabilities also provided by the activity, enabling you to convert datasets between storage formats.

Conversion between data formats has three requirements:

- The ability to read data from a source format
- The ability to write data to a sink format
- Support for mapping elements from source data into the sink format

Azure Data Factory provides read/write functionality for a large number of storage formats and services by means of its rich library of linked services and dataset types. The abstract concept of a dataset allows the Copy data activity to transform data between any source/sink dataset pairing, using ADF's interim data type system to manage type conversions between the two.

Key Concepts

The key concept for this chapter is the *Copy data activity*. This powerful activity supports data movement and transformation between a wide variety of storage formats and services. Related concepts include

- **Unstructured file:** A file treated as having no internal data structure – a blob. The Copy data activity treats files as unstructured when a binary copy is specified.
- **Structured file:** A file with a tabular data structure such as CSV or Parquet.
- **Parquet file:** A column-oriented, compressed structured file format supporting efficient storage and querying of large volumes of data.
- **Semi-structured file:** A file with a nontabular, frequently nested data structure, such as XML or JSON.
- **Collection reference:** Nested data structures can represent multiple collections of data simultaneously. In a Copy data activity schema mapping, the collection reference indicates which of the collections is being transformed.
- **Sink:** Azure Data Factory refers to data pipeline destinations as sinks.
- **Interim data type:** The Copy data activity converts incoming data values from their source types to interim ADF data types, then converts them to corresponding sink system type. This makes extensibility of ADF to support new datasets easier and faster.
- **Data integration unit (DIU):** A DIU is a measure of computing power incorporating CPU, memory, and network usage. Power is allocated to Copy data activity executions as a number of DIUs; the cost of an execution is determined by the duration for which it was allocated those DIUs.
- **Degree of parallelism (DoP):** A Copy data activity can be performed in parallel using multiple threads to read different files simultaneously. The maximum number of threads used during an activity's execution is its degree of parallelism; the number can be set manually for the activity, but this is not advised.

- **Azure SQL DB:** Azure-based, PaaS SQL Server service.
- **Logical SQL Server:** Logical grouping of Azure SQL Databases for collective management.
- **Online query editor:** Web-based query editor available for use with Azure SQL DB (and other Azure database platforms).

Azure Data Factory User Experience (ADF UX)

Many of the ADF UX's features will be familiar to you now as a result of working through this chapter. Figure 3-18 shows the ADF UX's authoring workspace subdivided into its various regions:

- Region 1 is the *Factory Resources* explorer. From here, you used the various Actions menus to create and clone pipelines and datasets and to organize them into folders.
- Region 2 is the *Activities toolbox*. This contains the Copy data activity, used extensively throughout this chapter, among others.
- Region 3 contains tabs for open resources such as pipeline or dataset definitions.
- Region 4 is the *canvas toolbar*. Its controls include the *Debug* button to run pipelines and the *Properties* pane toggle (slider icon). The code editor (braces icon) allows you to edit factory resources' JSON definitions directly.
- Region 5 is the *authoring canvas*, a visual editor used to interact with pipeline activities. Display tools for search, zoom, and auto-alignment are on the far right-hand side.
- Region 6 is the authoring canvas's *configuration pane*.

CHAPTER 3 THE COPY DATA ACTIVITY

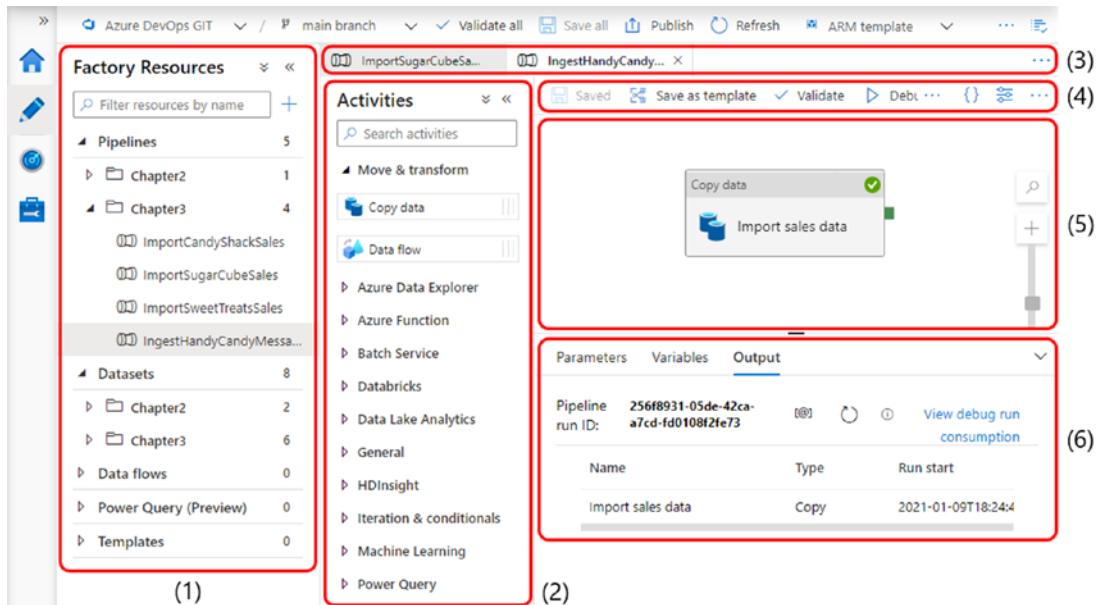


Figure 3-18. Regions in the ADF UX authoring workspace

The configuration pane contains settings tabs specific to the currently selected object, for example, a pipeline, dataset, or activity. These vary widely depending on the selected object. In this chapter, you saw configuration options for

- Azure SQL DB and CSV file dataset connections (Figures 3-7 and 3-12)
- CSV datasets as sources (Figure 3-10)
- SQL DB datasets as sinks (Figure 3-11)
- Structured and semi-structured schema mapping (Figures 3-14 to 3-17)
- Pipeline debug execution output (Figure 3-8)

The wide range of available connectors means that the list of dataset configuration options is very large. My intention here is not to provide an exhaustive introduction to all connector types, but to introduce a few common datasets and to provide you with the tools to find your own way around.

For SSIS Developers

The Copy data activity was introduced in Chapter 2, in a role similar to that of an SSIS File System Task. This chapter has extended its use to something like a rudimentary Data Flow Task. By abstracting sources and sinks as datasets, the activity is able to act like every kind of source and destination component, but it does not support intermediate per-row transformations familiar from the SSIS data flow surface. True Data Flow Task functionality will be introduced in Chapters 7 and 9.

Like SSIS, ADF achieves flexible transformation between source and destination type systems by using an intermediate type system of its own. As with Integration Services data types, this means that new connectors need only be concerned with type conversions into and out of ADF interim data types.

CHAPTER 4

Expressions

The pipelines you authored in Chapter 3 all have at least one thing in common: the values of all their properties are *static* – that is to say that they are determined at development time. In very many places, Azure Data Factory supports the use of *dynamic* property values – determined at runtime – through the use of *expressions*.

The main focus of this chapter is to introduce you to expressions, but you will also start to encounter more pipeline activities. The Copy data activity is sufficiently broad and important to have deserved a chapter to itself, but from now on you will start to make use of a wider variety of activities found in the activities toolbox.

Explore the Expression Builder

Expressions can be used in the ADF UX wherever you see the text *Add dynamic content* *[Alt+Shift+D]* – you may already have noticed this message appearing immediately below text or other input fields as you edit them. Figure 4-1 shows the *Wildcard folder path* field (on the Copy data activity’s *Source* tab) with focus so that the message is visible.

CHAPTER 4 EXPRESSIONS

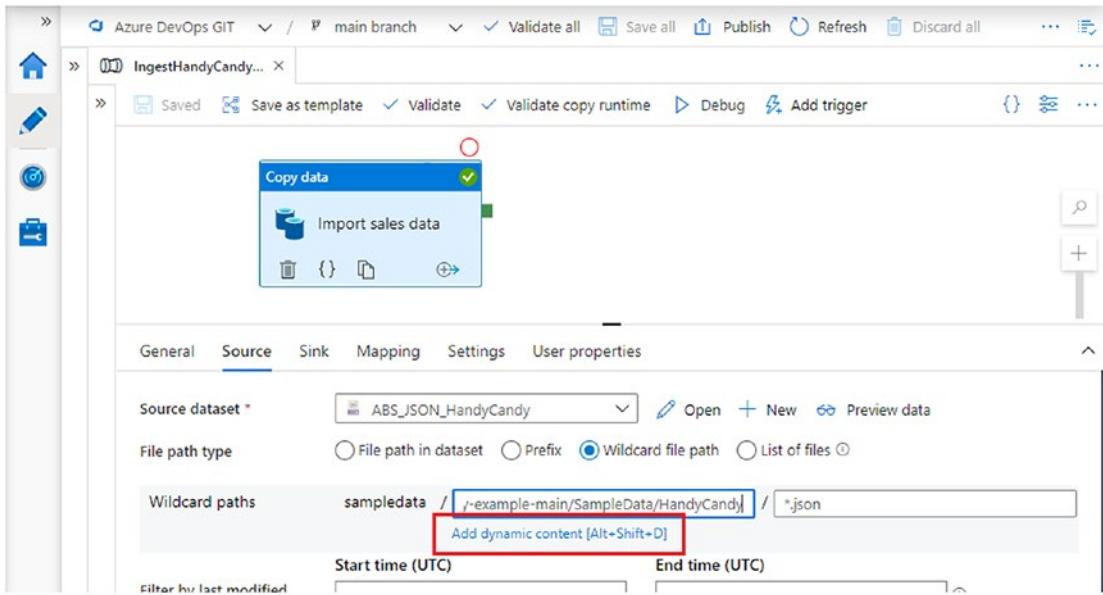


Figure 4-1. Add dynamic content [Alt+Shift+D] appears when editing a field

The message itself is a link to launch the ADF UX *expression builder*. As the message suggests, you can also launch the expression builder for a field by pressing *Alt+Shift+D* when the field has focus. Figure 4-2 shows the opened expression builder.

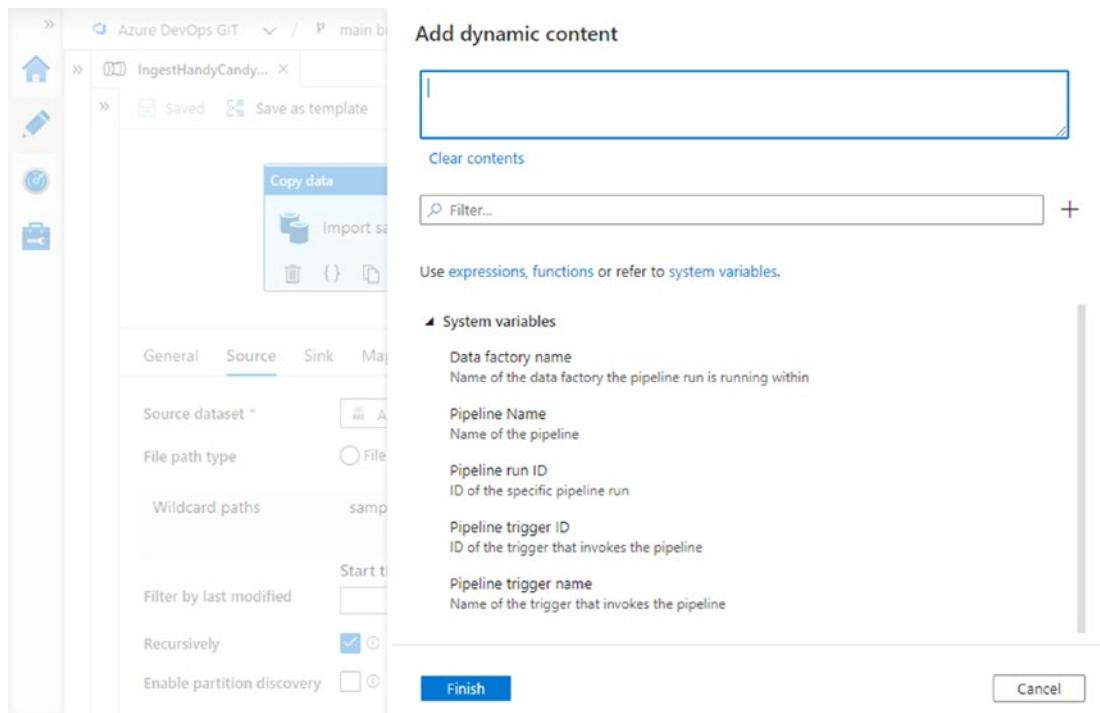


Figure 4-2. ADF expression builder

Expressions can make use of literal values, properties of factory resources, and a library of expression functions. The lower half of the expression builder is a scrollable list of available system variables, functions, user variables, and resource properties, arranged in collapsible sections. The *Filter...* search box above the list enables you to locate required elements quickly.

A later part of this chapter is concerned with user variables. Before that, you will start to use system variables and functions to add to pipeline functionality.

Note ADF expressions are *case-sensitive* – when entering expressions by hand, make sure that you use the correct case when referring to a function, variable, or property.

Use System Variables

Among its source dataset configuration options, the Copy data activity allows you to specify additional columns that can be copied to the activity sink. You can use this feature to duplicate an existing column, to include an additional static value, or to add a value determined at runtime. A common use case for dynamic values is to add audit information to incoming source data. In this section, you will extend your handling of Sweet Treats data by adding audit values in this way.

Enable Storage of Audit Information

Before proceeding, extend the [dbo].[Sales_LOAD] table to accommodate additional audit information by using your SQL client to run the script given in Listing 4-1 against your Azure SQL Database.

Listing 4-1. Add columns to [dbo].[Sales_LOAD]

```
ALTER TABLE [dbo].[Sales_LOAD]
ADD PipelineRunId UNIQUEIDENTIFIER NULL
, SourceFileName NVARCHAR(1024) NULL;
```

Create a New Pipeline

Create a new pipeline by cloning your “ImportSweetTreatsSales” pipeline from Chapter 3. Name the cloned pipeline “ImportSweetTreatsSales_Audit”, then create a “Chapter4” folder for pipelines and drag the new pipeline into it.

Add New Source Columns

Select the new pipeline’s Copy data activity. Add new source columns like this:

1. Navigate to the *Source* tab and scroll down until you find the *Additional columns* section.
2. Add a new column by clicking the *+ New* button. Give it the name “SourceFileName” and leave its value set to the default `$$FILEPATH`. This special reserved variable indicates to ADF that the runtime source file path is to be provided as the field’s value – it is not a true ADF expression and can only be used here.

3. Add a second new column called “PipelineRunId.” Expand the new field’s *VALUE* field dropdown, then select “Add dynamic content.”
4. In the expression builder, click *Pipeline run ID* in the *System variables* list. The text `@pipeline().RunId` appears in the expression pane at the top of the builder. Click *Finish*.

Tip Make sure that the new column names are exactly as specified here. The Sweet Treats pipeline relies on ADF’s inferred schema mapping – the source and sink dataset column names must match for the mapping to succeed.

Run the Pipeline

Run the pipeline and wait for it to finish execution, then use your SQL client to inspect the contents of the [dbo].[Sales_LOAD] table – Figure 4-3 shows the two new database columns populated with a pipeline run ID value and a source file path. (The file path is not the full, absolute path to the file in blob storage but is relative to the source dataset’s folder path – this is the behavior of `$$FILEPATH`.)

The pipeline run ID inserted into the new column is a *globally unique identifier (GUID)*. GUIDs are used to identify pipeline runs in the same way they are used to identify almost everything in Azure. The value of the run ID – “91d506d0-b5a4-4b53-8124-d8d22ea9f46f” in Figure 4-3 – is the value returned by the expression `@pipeline().RunId`.

The screenshot shows the Azure Data Factory Query editor interface. At the top, it displays the URL "Home > AdfByExample (adfbyexample-sql/AdfByExample)" and the title "AdfByExample (adfbyexample-sql/AdfByExample) | Query editor (preview)". Below the title, there are navigation links for "Login", "New Query", "Open query", and "Feedback". The main area is titled "Query 2" and contains a query editor with the following content:

```
1  SELECT TOP (1000) * FROM [dbo].[Sales_LOAD]
```

Below the query editor, there are two tabs: "Results" (which is selected) and "Messages". The "Results" tab displays a table with the following data:

Product	M...	SalesValueUSD	UnitsSold	PipelineRunId	SourceFileName
Schnoogles 8.81oz	271.32	28		91d506d0-b5a4-4b53-8124-d8d22ea9f46f	Aug-20/Sales.csv
Schnoogles 8.81oz	6450.93	687		91d506d0-b5a4-4b53-8124-d8d22ea9f46f	Jun-20/Sales.csv
Twisters 10.57oz	1625.26	494		91d506d0-b5a4-4b53-8124-d8d22ea9f46f	Sep-20/Sales.csv

The last three columns of the table (PipelineRunId, SourceFileName) are highlighted with a red border.

Figure 4-3. Query results showing populated audit columns

The purpose of the initial @ symbol is to indicate to ADF that what follows is to be evaluated as an expression. If you omit the @ symbol, the expression becomes a fixed value: the string “pipeline().RunId”. Such an omission would cause this pipeline to fail, because the string is not a valid value for the sink column’s UNIQUEIDENTIFIER SQL type.

Tip You must use the expression editor to create expressions – if you enter them directly into a field, the ADF UX stores them as string values. Fields that contain values are displayed with a white background; those containing expressions are light blue. This effect can be seen in Figure 4-5, later in this chapter.

Access Activity Run Properties

In this section, you will extend your pipeline to write some audit information about the pipeline’s execution into your Azure SQL Database. You will do this using ADF’s *Stored procedure activity*.

Create Database Objects

After the Copy data activity is complete, your pipeline will call a stored procedure to record some information about the pipeline's execution in a database log table. Use Listings 4-2 and 4-3 to create the log table and stored procedure in your database.

Listing 4-2. Create a log table

```
CREATE TABLE dbo.PipelineExecution (
    RunSeqNo INT IDENTITY PRIMARY KEY
    , PipelineRunId UNIQUEIDENTIFIER UNIQUE NOT NULL
    , RunStartTime DATETIME NULL
    , RunEndTime DATETIME NULL
    , RunStatus NVARCHAR(20) NULL
    , FilesRead INT NULL
    , RowsRead INT NULL
    , RowsCopied INT NULL
    , Comments NVARCHAR(1024) NULL
);
```

Listing 4-3. Create a logging stored procedure

```
CREATE PROCEDURE dbo.LogPipelineExecution (
    @PipelineRunId UNIQUEIDENTIFIER
    , @RunEndTime DATETIME
    , @FilesRead INT
    , @RowsRead INT
    , @RowsCopied INT
) AS

INSERT INTO dbo.PipelineExecution (
    PipelineRunId
    , RunEndTime
    , FilesRead
    , RowsRead
    , RowsCopied
) VALUES (
    @PipelineRunId
```

```
, @RunEndTime  
, @FilesRead  
, @RowsRead  
, @RowsCopied  
);
```

Add Stored Procedure Activity

The Stored procedure activity is found in the *General* group of the activities toolbox.

1. Open the “ImportSweetTreatsSales_Audit” pipeline in the ADF UX and drag a Stored procedure activity from the toolbox onto the authoring canvas. The configuration pane below the canvas expands automatically – on the activity’s *General* tab, name it “Log pipeline outcome.”
2. The new activity is to be executed *after* the Copy data activity has completed successfully. To enforce this, hold the left mouse button down over the green “handle” on the Copy data activity’s right-hand edge, then drag the mouse pointer over the Stored procedure activity. Releasing the left mouse button here creates an *activity dependency*. Figure 4-4 shows the pipeline configured with the correct dependency.

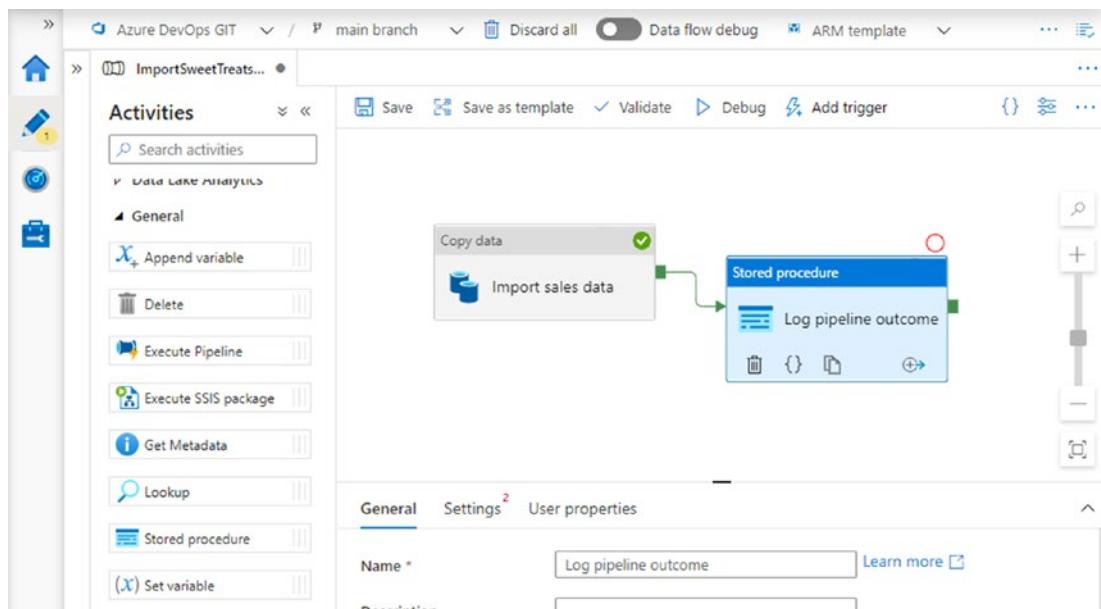


Figure 4-4. Stored procedure activity dependent on the Copy data activity

Tip To delete an activity dependency, right-click the dependency arrow and select *Delete* from the popup menu, or left-click to select it and hit your *Delete* key. These options are also available for pipeline activities, in addition to the *Delete* button (trash can icon) visible when an activity is selected.

3. Select the Stored procedure activity's *Settings* tab. It contains two mandatory input fields – choose your Azure SQL Database *Linked service*. After doing so, you will be able to select the newly created stored procedure from the *Stored procedure name* dropdown.
4. Below the *Stored procedure parameters* section heading is an *Import* button – click it to import the stored procedure's parameters from its definition in the database.
5. Use the expression builder to set the value of the parameter “PipelineRunId” as before, using the *Pipeline run ID* system variable.

6. Set the “RunEndDateTime” parameter to use the current UTC date and time. This is available using the *utcnow* function, found in the *Date Functions* section of the expression builder’s function list.
-

Note Expression date functions return *strings*, because the expression language has no date or time types. The language has six types: string, integer, float, boolean, arrays, and *dictionaries*. A dictionary is a collection of named elements.

7. The values of “FilesRead,” “RowsRead,” and “RowsCopied” will be taken from the Copy data activity’s output JSON object – recall that you encountered the activity output in Chapter 3. Open the expression builder for the “FilesRead” parameter and scroll to the bottom of the function list.
8. The *Activity outputs* section includes a list of activities with available outputs, in this case, the pipeline’s Copy data activity (displayed by its name, “Import sales data”). Click the activity and observe that the text `@activity('Import sales data').output` appears in the expression pane.
9. `activity('Import sales data')` refers to the activity named “Import sales data,” and the `.output` suffix specifies the activity’s output object. To identify a field within the object, you must add another dot, followed by its JSON path expression. The `filesRead` field is in the root of the output object, so its path is simply the field’s name. Add `.filesRead` to the expression pane so that the expression reads `@activity('Import sales data').output.filesRead`. Click *Finish*.
10. Set the “RowsRead” and “RowsCopied” parameter values in the same way. The completed parameter configuration is shown in Figure 4-5.

The screenshot shows the 'Settings' tab of a pipeline configuration in Azure Data Factory. It includes fields for 'Linked service' (AdfByExample), 'Stored procedure name' ([dbo].[LogPipelineExecution]), and a table for 'Stored procedure parameters'. The table lists six parameters with their types and expressions:

NAME	TYPE	VALUE
FilesRead	Int32	@activity('Import sales data').output.filesRead
PipelineRunId	Guid	@pipeline().RunId
RowsCopied	Int32	@activity('Import sales data').output.rowsCopied
RowsRead	Int32	@activity('Import sales data').output.rowsRead
RunEndDateTime	DateTime	@utcnow()

Figure 4-5. Parameter expressions for [dbo].[LogPipelineExecution]

For SSIS developers The Stored procedure activity is conceptually similar to the SSIS Execute SQL Task, but it does not support the execution of arbitrary SQL statements, nor can it return a result set. Activity dependencies have a clear parallel to SSIS precedence constraints and are discussed in greater detail in Chapter 6.

Run the Pipeline

Run the pipeline, then use your SQL client to inspect the contents of the two tables [dbo].[PipelineExecution] and [dbo].[Sales_LOAD].

- [dbo].[PipelineExecution] contains a single row containing pipeline execution information. You can verify the values in columns [PipelineRunId], [FilesRead], [RowsRead], and [RowsCopied] by comparing them to the Copy data activity output (on the *Output*

tab of the pipeline's configuration pane). Notice that the *Output* configuration tab now lists two activity executions – the Copy data activity and the Stored procedure activity.

- Rows in [dbo].[Sales_LOAD] have a [PipelineRunId] value matching that in [dbo].[PipelineExecution].

Use the Lookup Activity

For published pipelines, the information copied into [dbo].[PipelineExecution] is available in ADF's own pipeline execution history, but log tables can be more convenient, particularly for long-term pipeline monitoring and analysis. In this scenario, describing pipeline executions using GUIDs can be unwieldy or inconvenient. Using an integer identifier for *lineage tracking* – in place of the [PipelineRunId] in table [dbo].[Sales_LOAD], for example – may have the desirable effect of reducing database row size.

ADF's *Lookup activity* is used to obtain small amounts of information from an ADF dataset for use in a pipeline's execution. In this section, you will use the Lookup activity to retrieve a log row's integer identity value (already allocated in [dbo].[PipelineExecution]) and use it in place of the pipeline's execution GUID.

Create Database Objects

In the previous section, you inserted a log table row *after* the Copy data activity had completed. Now you will need to insert a row *before* copying starts, in order to generate and return its identity value. You will update the same row afterward to add pipeline execution information. Using your SQL client, run the code given in Listings 4-4 and 4-5 to create two new stored procedures.

Listing 4-4. Create [dbo].[LogPipelineStart]

```
CREATE PROCEDURE dbo.LogPipelineStart (
    @PipelineRunId UNIQUEIDENTIFIER
    , @RunStartTime DATETIME
    , @Comments NVARCHAR(1024) = NULL
) AS
```

```

INSERT INTO dbo.PipelineExecution (
    PipelineRunId
    , RunStartTime
    , Comments
) VALUES (
    @PipelineRunId
    , @RunStartTime
    , @Comments
);
SELECT SCOPE_IDENTITY() AS RunSeqNo;

```

Listing 4-5. Create [dbo].[LogPipelineEnd]

```

CREATE PROCEDURE dbo.LogPipelineEnd (
    @RunSeqNo INT
    , @RunEndDateTime DATETIME
    , @RunStatus VARCHAR(20)
    , @FilesRead INT
    , @RowsRead INT
    , @RowsCopied INT
) AS

UPDATE dbo.PipelineExecution
SET RunEndDateTime = @RunEndDateTime
    , RunStatus = @RunStatus
    , FilesRead = @FilesRead
    , RowsRead = @RowsRead
    , RowsCopied = @RowsCopied
WHERE RunSeqNo = @RunSeqNo;

```

The stored procedure shown in Listing 4-4 returns a result set consisting of one row and one column, containing the unique INT value inserted into the log table. To use this value in place of ADF's pipeline run ID, some additional modification to table [dbo].[Sales_LOAD] is necessary. Run the code given in Listing 4-6 to replace the table's 16-byte UNIQUEIDENTIFIER [PipelineRunId] column with a smaller 4-byte INT field.

Listing 4-6. Alter logging table

```
ALTER TABLE dbo.Sales_LOAD  
DROP COLUMN PipelineRunId;
```

```
ALTER TABLE dbo.Sales_LOAD  
ADD RunSeqNo INT;
```

Configure the Lookup Activity

You're now ready to revise your "ImportSweetTreatsSales_Audit" pipeline in the ADF UX.

1. Drag a Lookup activity from the *General* section of the activities toolbox onto the authoring canvas and name it "Lookup RunSeqNo." Create a dependency to ensure that the Copy data activity executes after the Lookup activity. Figure 4-6 shows the correctly configured dependency.
2. Select the Lookup activity so that its configuration pane is displayed, then open its *Settings* tab. Recall that the Lookup activity obtains data from a dataset, so the tab offers you a dataset dropdown – choose the Azure SQL DB dataset you have been using to copy data into [dbo].[Sales_LOAD].
3. Previously, you were able to override the file path configured in a blob storage dataset by specifying a wildcard path in the Copy data activity – Copy data activities using an Azure SQL DB connection can override their dataset's configuration in the same way. The dataset was created to represent the [dbo].[Sales_LOAD] table, but you can override this setting to represent other result sets, such as other tables (or views), SQL queries, or stored procedures that return a result set. Set *Use query* to "Stored procedure" to indicate that you will be using a stored procedure.
4. When you set *Use query* to "Stored procedure," the ADF UX offers you a dropdown *Name* list. Choose your new [dbo].[LogPipelineStart] procedure, then click the *Import parameter* button to acquire its parameter list.

5. Set the values of “PipelineRunId” and “RunStartTime” as before, using expressions `@pipeline().RunId` and `@utcnow()`. The new stored procedure also has a “Comments” parameter – leave it NULL for now, by ticking the *Treat as null* checkbox.
6. Finally, scroll down to verify that the *First row only* checkbox is ticked.

Figure 4-6 shows the correctly completed *Settings* pane for the Lookup activity.

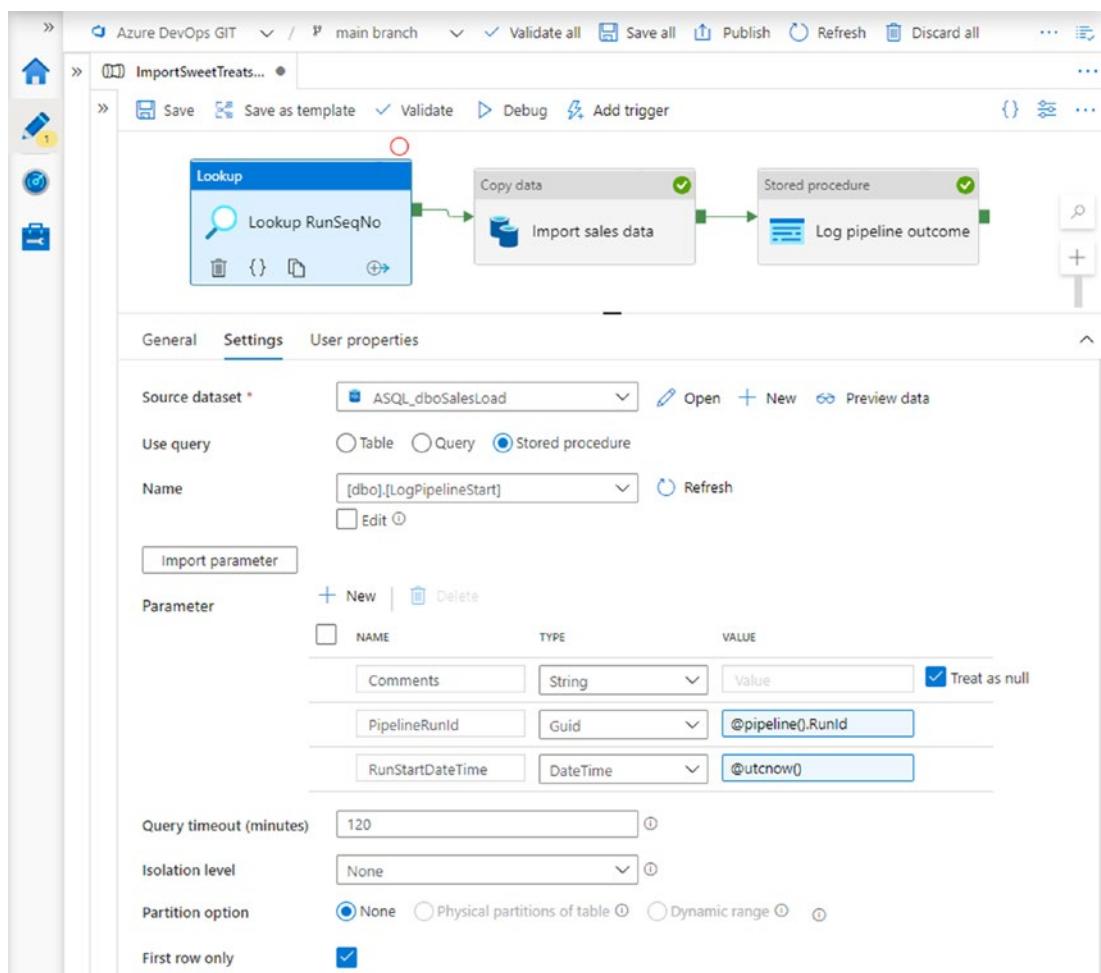


Figure 4-6. *Lookup activity with correct dependency and source settings*

For SSIS developers Do not confuse ADF's Lookup activity with the Lookup transformation in SSIS – the authoring canvas is equivalent to the SSIS control flow surface, so there is no stream of rows being processed here. A closer comparison is the use of an SSIS task to populate a package variable – for example, using an Execute SQL Task to retrieve a result set of one or more rows. The result set functionality absent from ADF's Stored procedure activity is present instead in the Lookup activity.

Use Breakpoints

To make use of the Lookup activity's output, you must read the value(s) you need from its execution output object. The exact JSON path required depends on the structure of the value returned: what field names it contains and whether it contains only one row or more. A simple way to find the JSON path in the activity execution's output object is to run the activity.

It is not possible to run ADF activities in isolation, but the ADF UX enables you to stop execution at a certain point using a *breakpoint*. When an activity is selected on the authoring canvas – such as the Lookup activity in Figure 4-6 – a red circle is displayed above its top-right corner. To set a breakpoint on the activity, click the circle. When the breakpoint is set, the circle is filled red – to remove the breakpoint, click the filled circle.

Figure 4-7 shows the pipeline with a breakpoint set on the Lookup activity. The pipeline's other activities are grayed out because setting the breakpoint disables them – the activity where the breakpoint is set will still be executed. Unlike other IDEs you may be familiar with, the ADF UX does not permit you to resume execution after the breakpoint has been hit – a pipeline executes up to and including the first activity with a breakpoint set, but no further.

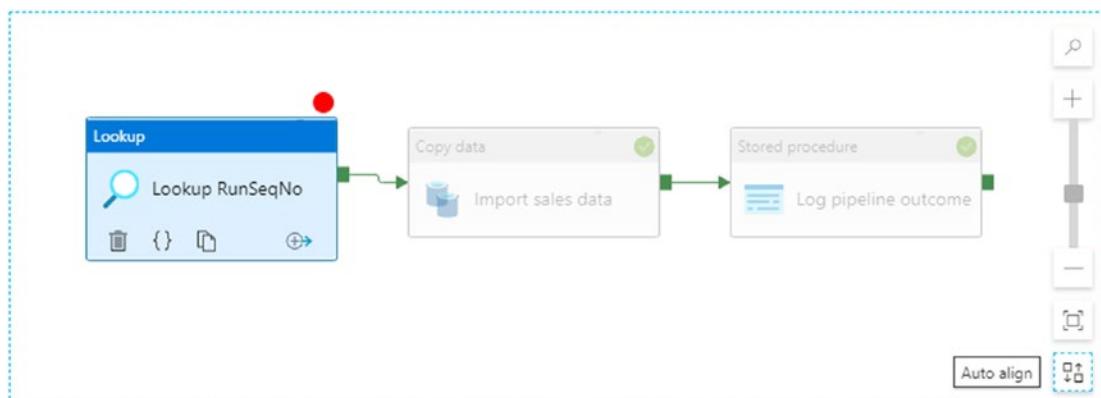


Figure 4-7. Breakpoint set on *Lookup RunSeqNo*

Tip To arrange activities neatly on the authoring canvas, use the *Auto align* button, found at the bottom of the column of display tools on the right-hand side of the canvas (shown in Figure 4-7).

Set a breakpoint on the *Lookup* activity as shown in Figure 4-7, then run the pipeline. When execution is complete, inspect the activity's output JSON in the pipeline's *Output* configuration tab. Figure 4-8 shows the output pane, highlighting the area of interest – the JSON path to the value returned by the stored procedure is `firstRow.RunSeqNo`.

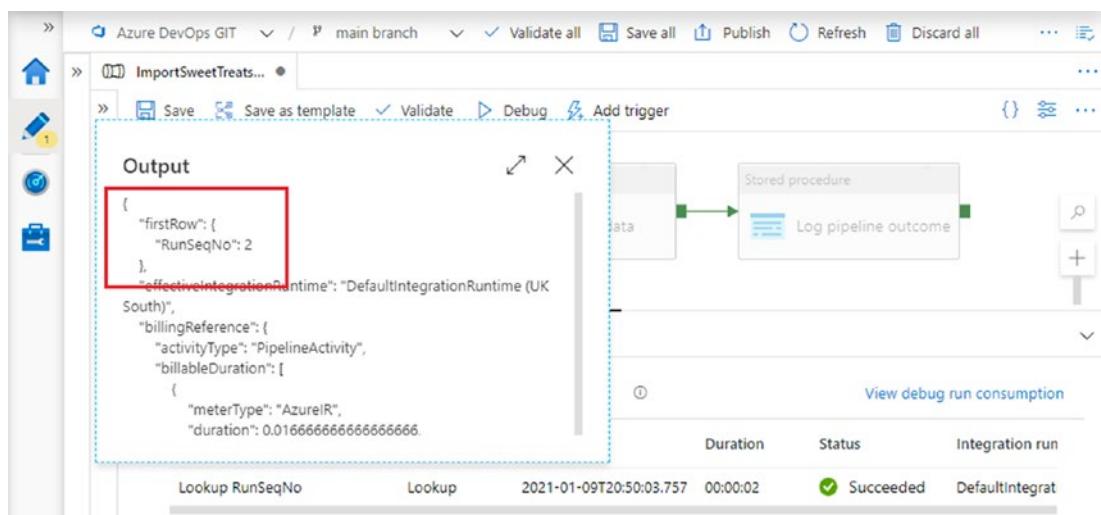


Figure 4-8. The *Lookup* activity's *Output* object

Use the Lookup Value

With the lookup value's JSON path, you can now use the value in the Copy data activity.

1. Select the pipeline's Copy data activity and open its *Source* configuration tab. Scroll down until you find the *Additional columns* section.
2. Change the name of the "PipelineRunId" column to "RunSeqNo." Recall that the activity is using an inferred mapping, so the column name must match that of the new database table column exactly.
3. Open the expression builder for the "RunSeqNo" column's *VALUE* field. Remove the expression – there is a *Clear contents* link below the expression pane – then scroll down to the *Activity outputs* section to find the Lookup activity output.
4. Click the Lookup activity output to add it to the expression pane, then add a dot and the JSON path identified earlier. Verify that the expression reads `@activity('Lookup RunSeqNo').output.firstRow.RunSeqNo`, then click *Finish*.

You may notice that the expression builder detects invalid syntax automatically, but it cannot detect an invalid JSON path – this is because the path is not determined until runtime.

Update the Stored Procedure Activity

You must now update your pipeline's Stored procedure activity to use the new stored procedure specified in Listing 4-5.

1. Select the Stored procedure activity and open its *Settings* configuration tab. Select "[dbo].[LogPipelineEnd]" from the *Stored procedure name* dropdown list.

Tip If you cannot see the stored procedure in the dropdown list, click the *Refresh* button to its right – this will reload the list from the database. If the field itself appears to be a text box instead of a dropdown list, verify that the *Edit* checkbox is not ticked.

2. Under *Stored procedure parameters*, click the *Import* button to import the new procedure's parameters. Set expressions for the values of parameters “FilesRead,” “RowsCopied,” “RowsRead,” and “RunEndDateTime” as before.
3. Set the value of “RunSeqNo” to use the same expression as in the previous section (based on the output of the Lookup activity). Set “RunStatus” to “Done” – this is a string literal, so you can enter it directly into the field without launching the expression builder.

An activity's output properties are available for use by any activity that is dependent on it, either directly or indirectly.

Run the Pipeline

Ensure that you have unset the breakpoint from the previous section, then run the pipeline. When execution has successfully completed, use your SQL client to inspect table contents. You should find that

- The latest row in [dbo].[PipelineExecution] – the row with the greatest [RunSeqNo] value – contains values for both [RunStartEndTime] and [RunEndDateTime].
- The value of [RunSeqNo] in [dbo].[Sales_LOAD] matches the value in the latest [dbo].[PipelineExecution] row.

The value of [RunSeqNo] in [dbo].[PipelineExecution] is the value that was created and retrieved by the Lookup activity, then subsequently used in the pipeline's Copy data and Stored procedure activities.

Note You will also notice an incomplete row in [dbo].[PipelineExecution] where an execution started but did not finish – this is the record created when you ran the pipeline up to the Lookup activity's breakpoint.

User Variables

In the previous section, you used the expression:

```
@activity('Lookup RunSeqNo').output.firstRow.RunSeqNo
```

in two places – once in the Copy data activity, then again in the Stored procedure activity. Having to use the same expression twice is not ideal, because if something changes (e.g., the activity name or the name of the returned field), you will need to update the expression wherever it is used. You can eliminate this redundant code by using a *user variable*.

Create a Variable

User variables are created at the level of the pipeline.

1. Click somewhere in blank space on the authoring canvas to open the configuration pane containing pipeline-level settings.
 2. Select the *Variables* tab and click the *+ New* button to create a new variable.
 3. A variable requires a name, a type, and optionally a default value.
Name the variable “RunSeqNo” and set its type to “String.”
-

Note The RunSeqNo value returned by the Lookup activity is an integer, but variables can be one of only three types: String, Boolean, or Array.

4. Specify a default value of “-1”. Providing a default value for “RunSeqNo” doesn’t make much sense from the perspective of this pipeline’s purpose, but setting one makes it easier to detect bugs in cases where the variable has not been assigned properly.

Set a Variable

Variable assignment is implemented by two ADF activities, both found in the *General* section of the activities toolbox:

- The *Set variable* activity sets the value of a variable, overwriting its default or any previous value.
- The *Append variable* activity appends an element to the end of an Array-type variable.

Set the “RunSeqNo” variable for use in your pipeline as follows:

1. Drag a Set variable activity onto the authoring canvas and name it “Set RunSeqNo.” The variable will be set to the value obtained by the Lookup activity, so create a dependency between the two activities.
2. On the Set variable activity’s configuration pane, select the *Variables* tab and choose the “RunSeqNo” variable from the *Name* dropdown. A *Value* field appears below the dropdown.
3. Click the *Value* field to reveal the expression builder link. Open the expression builder.
4. In the Activity outputs section, click *Lookup RunSeqNo* – if the activity is missing, verify that the Set variable activity is dependent on the Lookup activity. Complete the expression by adding `.firstRow.RunSeqNo`.
5. Although limited in number, variable types are *strict*. Without further modification, this variable assignment will fail, because the value obtained by the Lookup activity is a number, not a string. To convert the value to a string, wrap it in the *string* function (found in the expression builder’s *Conversion functions* section).

Figure 4-9 shows the Set variable activity, configured correctly with the expression:

```
@string(activity('Lookup RunSeqNo').output.firstRow.RunSeqNo)
```

Use the Variable

Variables are scoped at the pipeline level, so they can be used by any of the pipeline's activities. You must create activity dependencies as necessary to ensure that a variable is not used until it has been set.

1. Create a dependency between the Set variable activity and the Copy data activity. Figure 4-9 shows this dependency.

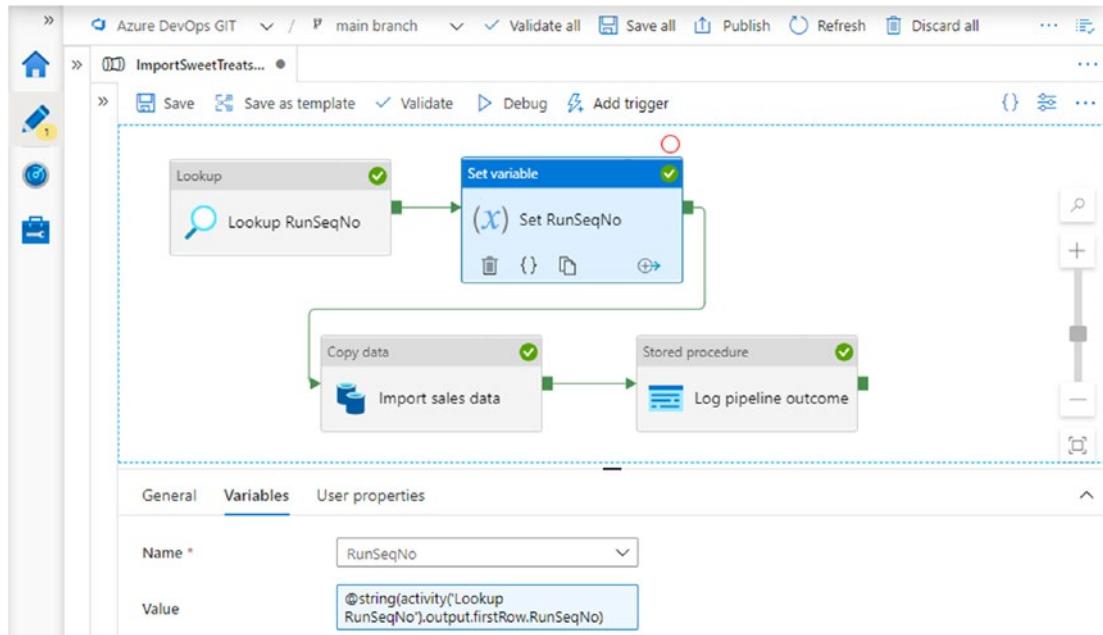


Figure 4-9. Set variable activity with value expression and dependencies

2. The chain of dependencies means that the direct dependency between the Lookup and Copy data activities is redundant – delete it.
3. Open the Copy data activity's *Source* configuration tab and open the expression builder for the “RunSeqNo” column (in the *Additional columns* section).

4. Remove the existing expression using the *Clear contents* link, then scroll down to the *Variables* section which now contains the “RunSeqNo” variable. Click it to add it to the expression pane, then click *Finish*.
5. Unlike system variables which use the property-like syntax `pipeline().property`, a user variable is referred to using the collection-like syntax `variables('variableName')`. The expression pane should now contain the expression `@variables('RunSeqNo')`.
6. Open the Stored procedure activity’s *Settings* configuration tab and update the expression for the “RunSeqNo” parameter so that it too uses the new variable.
7. Finally, run the pipeline and verify that it continues to function correctly.

Tip As the number of activities in your pipelines grows, it may become impossible to see them all on screen at the same time. You can move the canvas viewport around by clicking and dragging in canvas whitespace. Alternatively, change the display zoom level using the display tools on the right-hand side of the canvas.

Array Variables

The decision to use a variable of type “String” in the previous section was appropriate because the Lookup activity returned a single row. Ticking the *First row only* checkbox enforces this behavior (no matter how many rows are returned by the source dataset).

With the checkbox left unticked, the Lookup activity returns a JSON array called `value`. The array contains one element for each row returned by the dataset, up to a maximum of 5000 rows or 2MB of data. To store this value for reuse in the pipeline, assign it to a variable of type “Array.”

A frequent requirement for array values is the ability to iterate over them, processing each element individually but in the same way. ADF activities for doing this are introduced in Chapter 6.

Concatenate Strings

While using the expression builder, you will have noticed that there are six groups of expression functions:

- *String* functions, which operate on strings (e.g., *concat*, *substring*)
- *Math* functions, which perform mathematical operations (*add*, *max*)
- *Logical* functions, which return true or false (*equals*, *if*)
- *Date* functions, which return strings representing dates (*utcnow*, *addhours*) – recall that the expression language has no date or time types
- *Collection* functions, which act on arrays and dictionaries (*length*, *contains*)
- *Conversion* functions, which convert values into different data types (e.g., *string*) or representations (e.g., *base64*).

In this section, you will use the *concat* function to build a message by concatenating several strings together. The message will be logged using the [dbo].[LogPipelineStart] stored procedure's “Comments” parameter.

1. On the authoring canvas, open the Lookup activity's *Settings* configuration tab.
2. Untick the *Treat as null* checkbox on the “Comments” parameter and open the expression builder to set its value.
3. Click the *concat* function – found in the *String functions* section – to add it to the expression pane.
4. The function takes a list of string arguments separated by commas. Literal string values are surrounded by single quotes. Enter the first argument 'Pipeline ' (including a final space after the word “Pipeline”). The expression should now read @ concat('Pipeline ').
5. Add a second argument by inserting a comma after the first, then selecting *Pipeline Name* from the list of *System variables*.

6. Add a third string literal argument ' executed in ' and a fourth by selecting the system variable *Data factory name*. The entire expression should now read @concat('Pipeline ', pipeline().Pipeline, ' executed in ', pipeline().DataFactory). Click *Finish* and run the pipeline.

When pipeline execution is complete, inspect the latest record in table [dbo].[PipelineExecution] – its [Comments] field should contain something like “Pipeline ImportSweetTreatsSales_Audit executed in MyDataFactory” (depending on the name of your ADF instance).

Infix Operators

The ADF expression language contains no *infix* operators – all operations are implemented as functions. In the same way that you use `concat('a', 'b')` (instead of something like `'a' + 'b'`), all mathematical operations are invoked using functions found in the expression builder’s *Math functions* section, for example:

- To calculate $1 + 2$, you must use the *add* function: `add(1, 2)`
- Subtraction, multiplication, and division are implemented by the functions *sub*, *mul*, and *div*, respectively.

String Interpolation

Another approach to building string expressions is to use *string interpolation*. An interpolated string is a string value that contains *placeholder expressions* which are evaluated at runtime. A placeholder expression is contained in braces and preceded by the @ symbol like this: `@{placeholderExpression}`.

Interpolated strings are more readable than those built using the *concat* function – the equivalent of the function call you built in the previous section is

`Pipeline @{pipeline().Pipeline} executed in @{pipeline().DataFactory}`

Interpolated strings cannot be nested inside other expressions. If an interpolated string appears as a function argument, its placeholder expressions are not evaluated but are treated as string literals.

Escaping @

The initial @ symbol is necessary to indicate to ADF that what follows it should be evaluated as an expression. If you need to specify a string literal that begins with the @ character, you must *escape* it by using it twice:

- @add(1,2) is an expression that evaluates to 3.
- @@add(1,2) is the string literal “@add(1,2)”.

Chapter Review

Expressions allow you to specify many factory resource properties dynamically – values for such properties are evaluated at runtime using the configured expression. You can use an expression wherever the link *Add dynamic content [Alt+Shift+D]* appears, allowing you to launch the expression builder.

An expression can be a string literal (possibly containing placeholder expressions), a variable, or a function. Nesting of functions allows you to build complex, powerful expressions.

User variables are created at the pipeline level and accessible to all activities for the duration of a pipeline run. Use the Set variable activity to change a variable’s default or previously set value. The Append variable activity allows you to add elements to the end of an Array variable’s value.

Key Concepts

- **Expression:** An expression is evaluated at pipeline execution time to determine a property value. The data type of an expression is string, integer, float, boolean, array, or dictionary.
- **Array:** A collection of multiple values referred to as *elements*. Elements are addressed by an integer index between zero and one less than the array’s length.
- **Dictionary:** A collection whose elements are referred to by name.
- **Expression builder:** An expression editor built into the ADF UX.

- **System variable:** System variables provide access to the runtime values of various system properties.
- **User variable:** Created to store String, Boolean, or Array values during a pipeline's execution.
- **Expression function:** One of a library of functions available for use in expressions. Function types include String, Math, Logical, Date, Collection, and Type conversions.
- **Interpolated string:** A string literal containing placeholder expressions.
- **Placeholder expression:** An expression embedded in an interpolated string, evaluated at runtime to return a string.
- **Escape:** String literals beginning with the @ character must be escaped to prevent their interpretation as expressions. @ is escaped by following it with a second @ character.
- **Stored procedure activity:** ADF activity that enables the execution of a database stored procedure, specifying values for the stored procedure's parameters as required.
- **Lookup activity:** ADF activity that returns one or more rows from a dataset for use during a pipeline's execution. In the case of SQL Server datasets, rows can be returned from tables, views, inline queries, or stored procedures.
- **Set variable activity:** ADF activity used to update the value of a user variable.
- **Append variable activity:** ADF activity used to add a new element to the end of an existing Array variable's value.
- **Activity dependency:** Constraint used to control the order in which a pipeline's activities are executed.
- **Activity output object:** JSON object produced by the execution of an ADF activity. An output object and its properties are available to any activity dependent on the source activity, either directly or indirectly.

- **Breakpoint:** An ADF UX breakpoint allows you to run a pipeline, in debug mode, up to and including the activity on which the breakpoint is set (breakpoints do not exist in published pipelines). Unlike in other IDEs, it is not possible to resume execution after hitting a breakpoint.
- **\$\$FILEPATH:** A reserved system variable that enables the Copy data activity to label incoming file data with its source file. \$\$FILEPATH is available solely to populate additional columns in the Copy data activity and cannot be used in expressions.
- **\$\$COLUMN:** A reserved system variable that enables the Copy data activity to duplicate a specified column in incoming data. \$\$COLUMN is available solely to populate additional columns in the Copy data activity and cannot be used in expressions.
- **Additional columns:** A Copy data activity source can be augmented with additional columns, the values of which are specified by an expression, \$\$FILEPATH, \$\$COLUMN, or a hard-coded static value.
- **Lineage tracking:** The practice of labeling data as it is processed, to enable later identification of information related to its source and/or processing.

For SSIS Developers

This chapter introduced two new activities which have similar functions to SSIS activities. The correspondence is not exact:

- The Stored procedure activity allows you to execute a stored procedure. Unlike SSIS's Execute SQL Task, you cannot use it to execute arbitrary SQL code or to return a result set.
- The Lookup activity enables you to load small datasets. With a SQL connection, this is similar to the use of an Execute SQL Task to execute SQL code to return a result set, but the activity can also be used to load datasets of different kinds.

The order of activities' execution is controlled using activity dependencies. These are conceptually similar to SSIS precedence constraints but have some important differences that are discussed in Chapter 6.

Compared to SSIS in Visual Studio, the ADF UX's support for breakpoints is limited. All expression functions use the syntax `functionName(arguments)` – infix operators like “+” used in SSIS expressions are not supported.

CHAPTER 5

Parameters

Chapter 4 introduced expressions as a way of setting property values in factory resources at runtime. The examples presented used expressions to determine values for a variety of properties, all of which were under the internal control of the pipeline. But sometimes it is convenient to be able to inject external values into factory resources at runtime, either to share data or to create generic resources which can be reused in multiple scenarios. Injection of runtime values is achieved by using *parameters*.

Runtime parameters are supported not only by pipelines but also by datasets, linked services, and data flows. Data flows are introduced in Chapter 7 – this chapter looks at the use of parameters with the other factory resource types. Parameters can also be defined at the level of a data factory instance. Unlike other parameter types, these *global parameters* do not enable runtime value substitution, but rather represent *constants*, shared by all pipelines.

You will be using an instance of Azure Key Vault to enable you to parameterize linked services securely, so in the first part of the chapter, you will create and configure your key vault.

Set Up an Azure Key Vault

Azure Key Vault provides a secure, cloud-based repository for cryptographic keys and other secrets such as database or storage account connection strings. In this section, you will add a key vault to your resource group, store a connection string inside it, and configure an Azure Data Factory linked service using the securely stored connection string.

Create a Key Vault

Azure Key Vault instances are created and managed in the Azure portal.

1. Open the portal and create a new resource of type *Key Vault*.
2. Select the subscription and resource group that contains your ADF instance, then choose a globally unique *Key vault name*.
3. Choose the *Region* closest to you geographically.
4. Ensure that the *Pricing tier* is set to “Standard,” then accept the rest of the default values by clicking *Review + create*. (I am purposely bypassing the *Access policy*, *Networking*, and *Tags* tabs.) Figure 5-1 shows the completed *Basics* tab of the *Create key vault* blade.

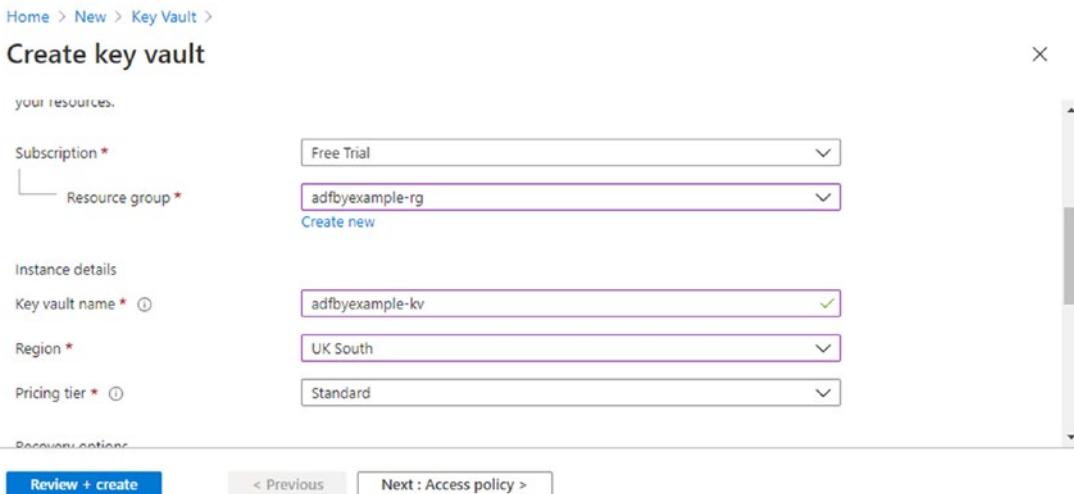


Figure 5-1. Create key vault Basics tab

5. After validation, click *Create*. A notification message is displayed when deployment is complete.

Create a Key Vault Secret

Your storage account's connection string can be found in the Azure portal. In this section, you will store its value securely in the new key vault.

1. Open the portal blade for your storage account (you can find it in the list of resources contained in your resource group or using the *Search resources, services and docs (G + /)* box in the portal toolbar).
2. Select *Access keys* from the sidebar's *Settings* group. In order to copy key values, you must first click *Show keys*. Figure 5-2 shows the blade after doing so – the *Show keys* button has been replaced by *Hide keys*. Secret values have been truncated in the screenshot.
3. Copy the *Connection string* value for *key1* to the clipboard using the copy button at the right-hand end of the field (indicated in Figure 5-2).

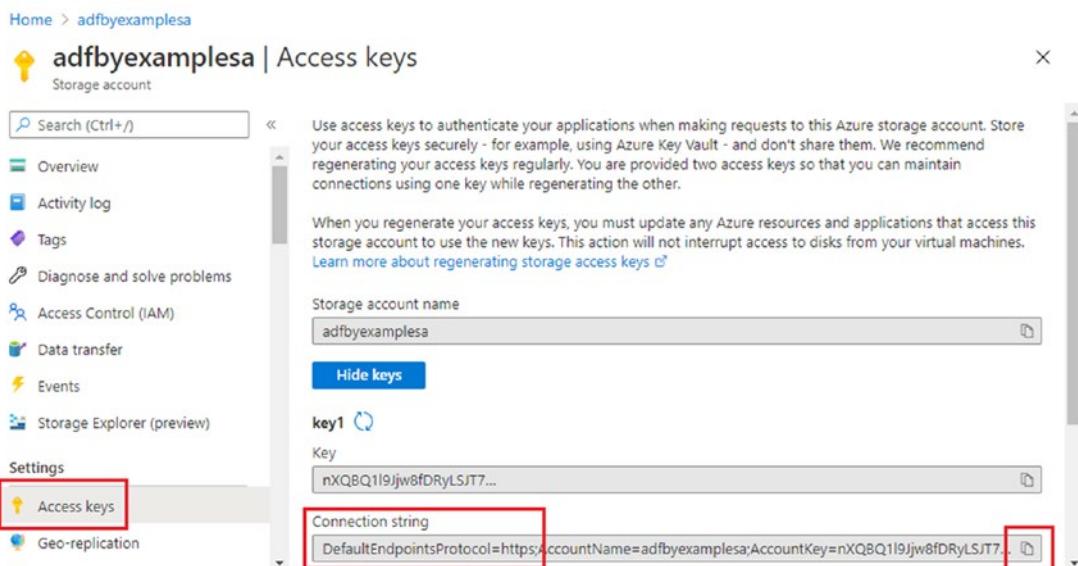


Figure 5-2. Storage account access keys

4. Open the portal blade for your key vault and select *Secrets* from the sidebar's *Settings* group. At the top of the *Secrets* blade, click the *+ Generate/Import* button to add a new secret.
5. The *Create a secret* blade is displayed. Paste the contents of the clipboard into the *Value* field, then enter a *Name* for the secret. Figure 5-3 shows the completed form for my "StorageAccountConnectionString" secret. Click *Create*.

The screenshot shows the 'Create a secret' blade in the Azure portal. At the top left is the breadcrumb navigation: Home > adfbyexample-kv >. The main title is 'Create a secret'. Below it is a section titled 'Upload options' with a dropdown menu set to 'Manual'. The 'Name' field is required and contains 'StorageAccountConnectionString'. The 'Value' field is required and contains a redacted string. The 'Content type (optional)' field is empty. There are two sections for 'Set activation date?' and 'Set expiration date?' both with checkboxes that are unchecked. At the bottom is a blue 'Create' button.

Figure 5-3. Creating a secret in the key vault

Grant Access to the Key Vault

Your data factory cannot use the secrets stored in your key vault until you grant it permission to do so. The data factory instance has an associated *managed identity* – a managed application registered in Azure Active Directory – which was created automatically when you created the data factory. You must grant access to this identity.

1. Open the portal blade for your key vault and select *Access policies* from the sidebar's *Settings* group. On the *Access policies* blade, locate and click the *+ Add Access Policy* button.

2. On the *Add access policy* blade, select *Get* from the *Secret permissions* multiselect dropdown, then under *Select principal*, click *None selected* – this opens the security principal selection blade.
3. At the top of the blade is a search input field. An ADF managed identity service principal has the same name as the ADF instance it represents – enter the name of your data factory to search for the service principal. The search will return one matching item, as shown in Figure 5-4. Click the item to choose it, then click the *Select* button at the bottom of the blade.

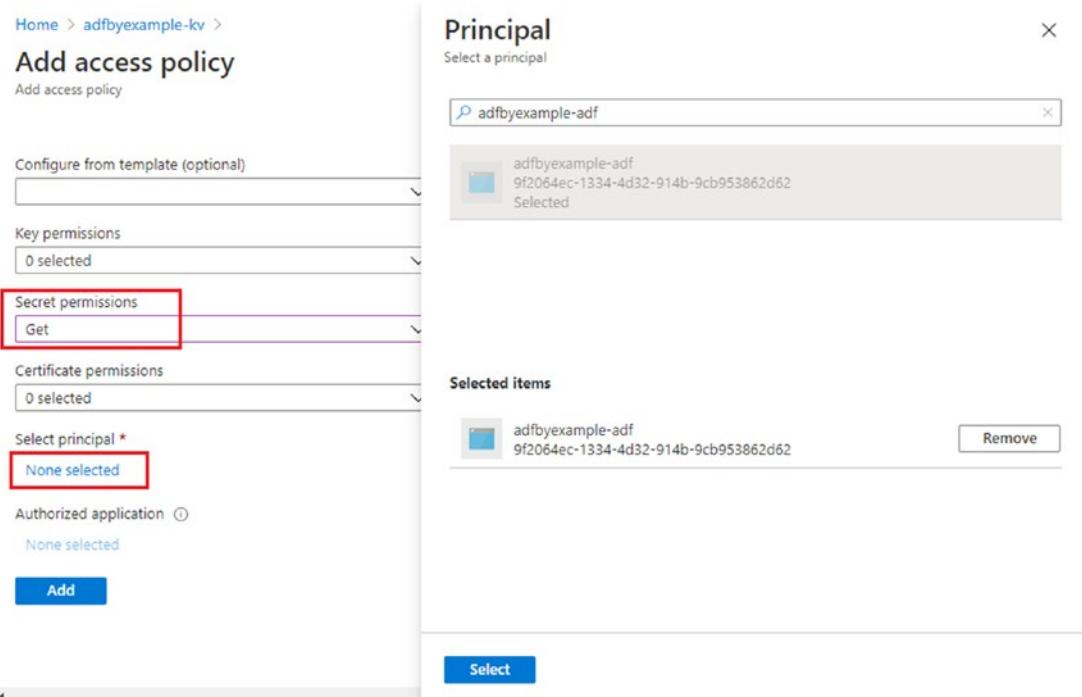


Figure 5-4. Select ADF managed identity principal for key vault access policy

4. On the *Add access policy* blade, click *Add*, then at the top of the *Access policies* blade, click *Save*.

Note Access policies can also be configured during key vault creation – the *Next: Access policy* button visible in Figure 5-1 leads to this step after completing the vault's basic details.

Create a Key Vault ADF Linked Service

Azure Data Factory accesses a key vault in exactly the same way it does other types of external resource: using a linked service. To refer to a key vault from within your data factory, you must create a linked service to represent it.

1. Open the ADF UX management hub and select *Linked services* in the *Connections* section of its sidebar.
2. Click the *+ New* button to add a new linked service, then search for and select the *Azure Key Vault* data store. Click *Continue*.
3. On the *New linked service (Azure Key Vault)* blade, provide a *Name* for the key vault linked service, then select your key vault from the *Azure key vault name* dropdown.
4. Figure 5-5 shows the completed blade. Use the *Test connection* button to check the linked service configuration, and when successful, click *Create*.

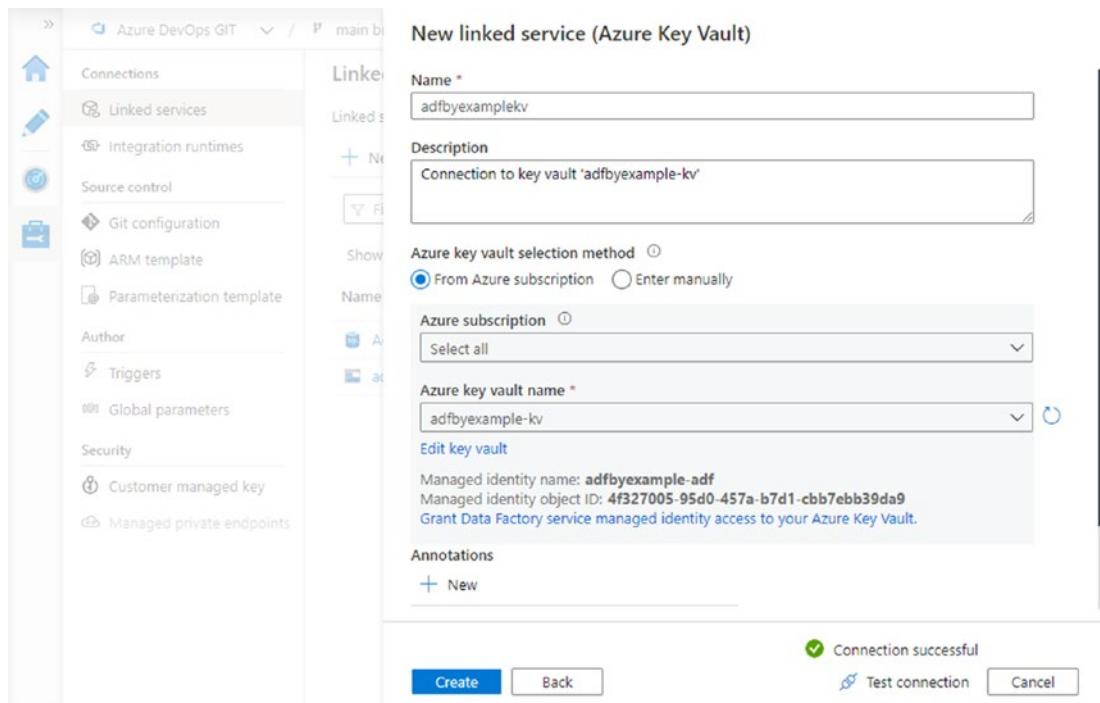


Figure 5-5. Create linked service for Azure Key Vault

Create a New Storage Account Linked Service

The data factory linked services that you created in Chapters 2 and 3 were published to ADF immediately, to allow the associated connection credentials to be stored securely. Using key vault secrets is a much better practice – in this section, you will create a new storage account linked service that obtains the value of its connection string at runtime, by referencing your key vault secret.

1. On the same *Linked services* page in the management hub, create another new linked service, this time using the *Azure Blob Storage* data store.
2. Ensure that *Authentication method* is set to “Account key,” then use the toggle below that field to change the connection type from “Connection string” to “Azure Key Vault.”

3. Select your key vault linked service from the *AKV linked service* dropdown, then enter the name of your storage account connection string secret.
4. Figure 5-6 shows the completed blade. Use the *Test connection* button to check the linked service configuration, and when successful, click *Save*.

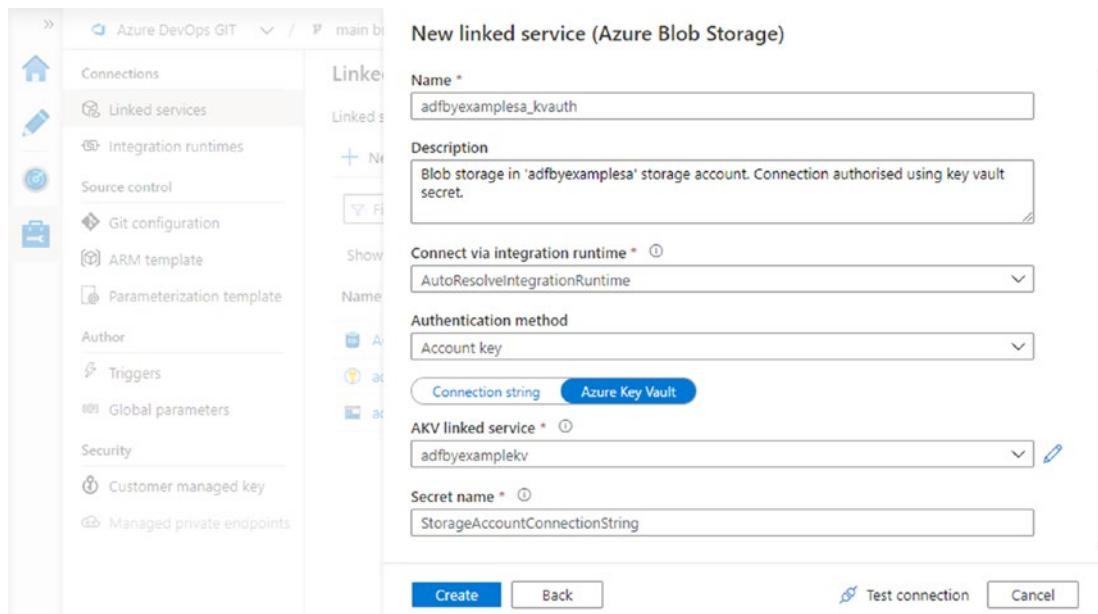


Figure 5-6. Blob storage linked service using a key vault secret

The new linked service obtains credentials from the key vault at runtime, by obtaining the value of your named secret, authorized using the ADF instance's managed identity. Unlike your previous storage account linked service, the new linked service has not been published to ADF, because there is no longer any part of its definition that requires secure storage.

Use Dataset Parameters

In Chapter 3, you created a number of ADF datasets, as and when you needed them, to represent different source files. The dataset “ABS_CSV_CandyShack” (shown in Figure 5-7) has the following features:

- It is of type *DelimitedText*, indicated in the canvas space at the top of the figure. Some properties shown on the *Connection* tab are specific to this file type – for example, *Column delimiter* and *Row delimiter*.
- It has a specified *Linked service*, a blob storage account.
- It has a *File path* consisting of a *container*, a *directory*, and a *file*. Each time you used a dataset with a file path, you overrode its directory and file settings using wildcard settings on the Copy data activity’s *Source* configuration tab.
- It has an associated file schema, visible on the *Schema* tab. When you used the “ABS_CSV_CandyShack” dataset, you overrode its schema with settings in the Copy data activity’s *Mapping* configuration tab. If that isn’t clear, look at the Candy Shack dataset’s *Schema* tab – it contains the schema for Sweet Treats data (from when you cloned the dataset). It does not match the schema stored in the Copy data activity mapping.

CHAPTER 5 PARAMETERS

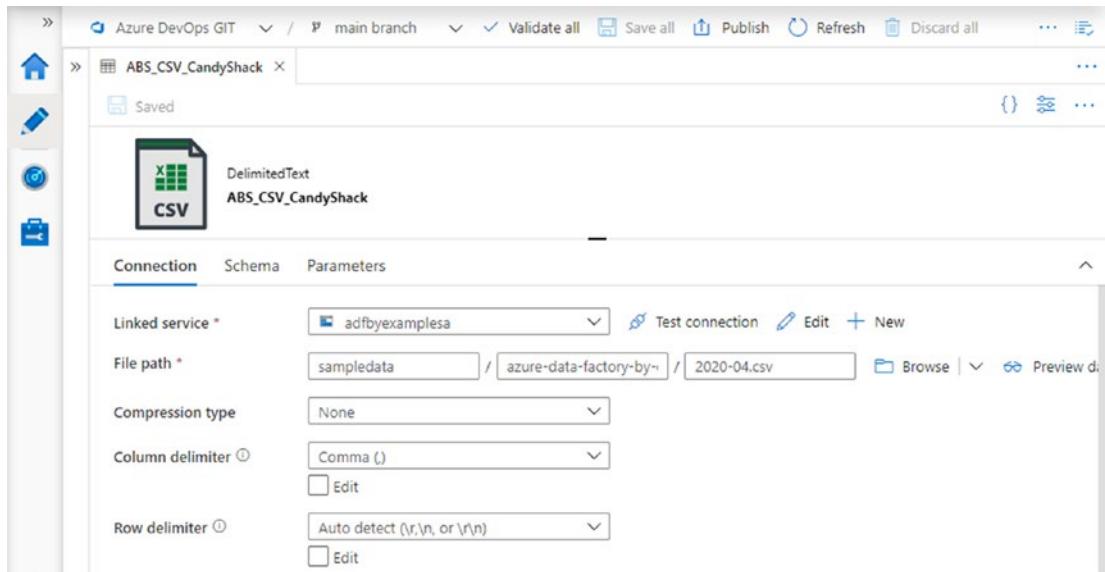


Figure 5-7. “ABS_CSV_CandyShack” dataset configuration

In fact, the “ABV_CSV_CandyShack” and “ABS_CSV_SweetTreats” datasets are equivalent and interchangeable – they are of the same type, use the same linked service, and specify the same container. The values where they differ – file path and schema – are provided at runtime by the Copy data activity. The same is true of the two JSON file datasets.

Interchangeable datasets are *redundant* – they can be replaced by a single, generic, reusable dataset. There are several advantages to generic datasets:

- You avoid having to create multiple equivalent datasets that differ only in file location and schema.
- A single generic dataset (replacing interchangeable datasets) can be clearly labeled as such. A source-specific dataset suggests a tighter coupling to its source location than is really the case.
- Your collection of datasets is less cluttered, making it smaller and easier to manage.

You could create a dataset like this with the knowledge you already have, but it would still require a hard-coded container name. In the following section, you will create a file-independent dataset by using *dataset parameters* to specify file path components at runtime.

Create a Parameterized Dataset

In the ADF UX, create a new “Chapter5” datasets folder, then create a new dataset inside it as follows:

1. On the *New dataset* blade, choose *Azure Blob Storage*, then *Continue*.
2. Select *JSON* format, then *Continue*.
3. Name your dataset “ABS_JSON_<newlinkedservice>”, where <newlinkedservice> is the name of the linked service you created in the previous section. Select that linked service from the *Linked service* dropdown.
4. Leave every other option with its default values and click *OK* to create the dataset. The dataset and its *Properties* blade open automatically – close the *Properties* blade.

The new dataset is not yet usable because it contains no file path information. To inject file path information at runtime, create and use dataset parameters as follows:

1. Select the new dataset’s *Parameters* configuration tab and click the *+ New* button.
2. Name the new parameter “Container” and ensure its type is “String.” Leave its default value blank.
3. Create two more parameters called “Directory” and “File” in the same way, giving each one a default value of “.” (period).
4. Return to the *Connection* tab, and click into *File path*’s *Container* field. The *Add dynamic content [Alt+Shift+D]* message appears – launch the expression builder.
5. You will notice that the options available for selection in the expression builder are different to those you encountered in Chapter 4. (For example, pipeline-related system variables are not available.) At the bottom of the expression builder is a *Parameters* section – click the “Container” parameter.

6. The expression `@dataset().Container` appears in the expression pane. Click *Finish*.
7. Repeat steps 3–5 for the *File path*'s *Directory* and *File* fields, using the corresponding dataset parameters. Save the dataset.

Figure 5-8 shows the dataset's *Connection* configuration tab using its three parameters.

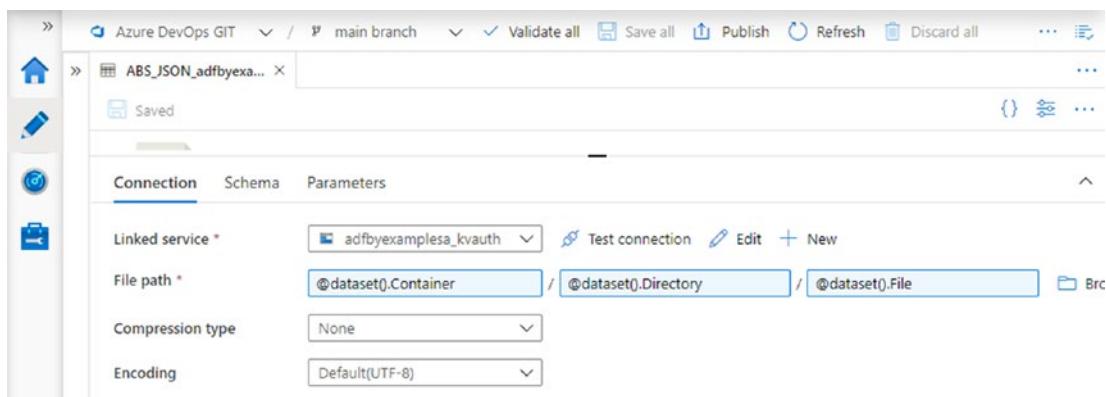


Figure 5-8. JSON dataset with parameterized container name

Note The syntax `dataset().ParameterName` identifies a parameter called “ParameterName” within the context of the dataset where the parameter is defined. The `dataset()` prefix can only be used inside a dataset definition and only to refer to that dataset’s own parameters.

Use the Parameterized Dataset

Create a new “Chapter5” pipelines folder, then create a new pipeline by cloning Chapter 3’s “ImportSugarCubeSales” pipeline. Drag the cloned pipeline into the “Chapter5” pipelines folder.

The cloned Sugar Cube pipeline contains a single activity – the Copy data activity used to copy data from JSON files into your Azure SQL DB.

1. Select the Copy data activity, then open its *Source* configuration tab.
2. Change its *Source dataset* from “ABS_JSON_SugarCube” to your new, parameterized dataset. When you select a parameterized dataset, a *Dataset properties* region appears below the *Source dataset* dropdown – this is where you specify values or expressions for the dataset’s parameter(s).
3. Enter the value “sampledata” for the “Container” parameter.

Figure 5-9 shows the Copy data activity’s *Source* configuration tab specifying the container name. Run the pipeline and use your SQL client to verify that Sugar Cube data has been read using the parameterized dataset and loaded correctly.

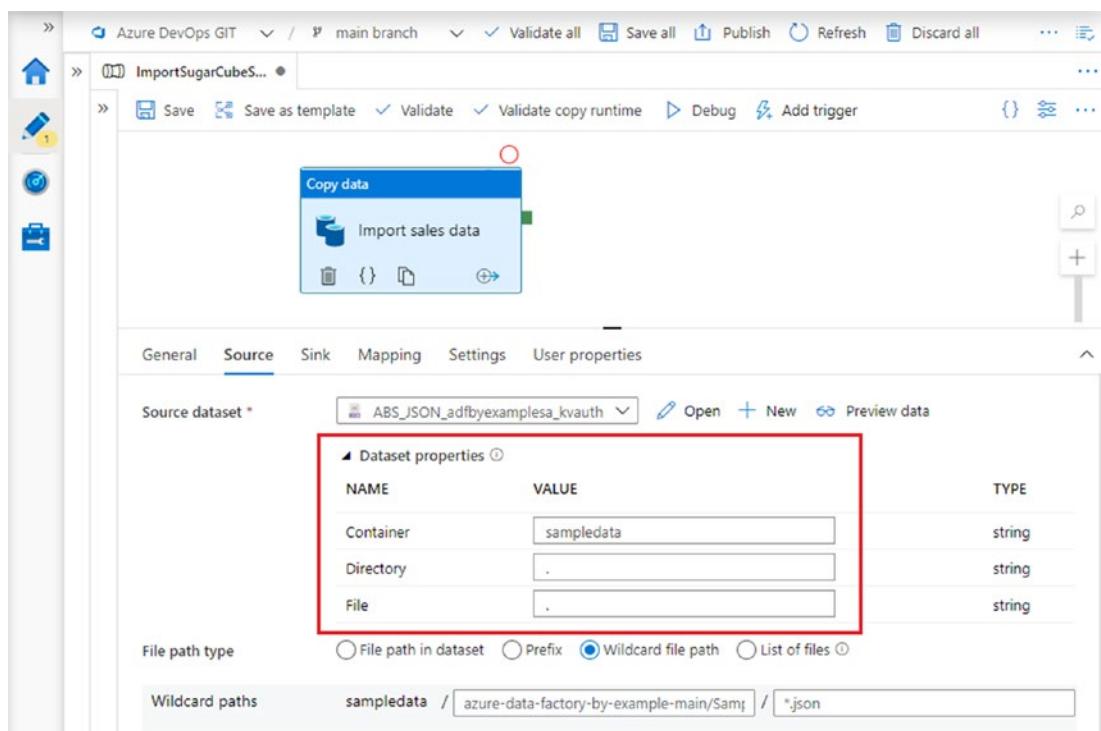


Figure 5-9. Copy data activity configured to pass dataset parameters

Tip You can make a parameter *optional* by giving it a default value – if no value is provided at runtime, the default value is used instead. In this example, the default values were overridden because the Copy data activity specified a *File type path* of “Wildcard file path” – but without them, you would still have to supply parameter values for the pipeline to be valid.

Reuse the Parameterized Dataset

Repeat the steps from the previous section, this time cloning Chapter 3’s “IngestHandyCandyMessages” pipeline into the “Chapter5” folder. Run the resulting pipeline and check the results. (Recall that this pipeline transforms source JSON messages into a Parquet file in blob storage, so use Azure Storage Explorer to see its effect.)

The major result here is that both pipelines are now using the same source dataset – in fact, any pipeline reading JSON data from your blob storage account could use this dataset. This is a significant improvement:

- You need fewer source datasets – you could choose to specify a single parameterized dataset for each blob storage linked service/file type combination.
- The parameterized source dataset is generic by design. Anyone looking at the dataset in future is in no danger of believing that the dataset is coupled to any specific file location.

Given that all the source data is in the “sampledata” container, it would be reasonable to ask why parameterize the dataset container at all. How your source data is organized may be a consideration here:

- If your source data is in multiple containers, a parameterized dataset provides useful reusability.
- If your source data is guaranteed always to occupy a single container, then a dataset with a hard-coded container name relieves you of providing a parameter value every time you use it.

If your source data is in multiple storage accounts, you might want to go a step further and implement a generic dataset per file type. This would require linked service connection details to be injected at runtime, which is the subject of the next section.

Use Linked Service Parameters

By parameterizing the dataset in the previous section, you increased its reusability, because it can now be used to refer to any JSON file in the underlying storage account. A dataset that could refer to any JSON file in *any* storage account would be even more reusable, but would require the storage account also to be specified at runtime.

The storage account used by a dataset is specified in the linked service it uses and can be provided at runtime using a *linked service parameter*. Passing a storage account connection string as a parameter value would be possible, but insecure. Configuring a linked service to use a key vault secret means that the same effect can be achieved securely, by allowing the secret's name to be passed in as a parameter.

Create a Parameterized Linked Service

In the first part of this chapter, you created a key vault and added a secret – your storage account's connection string. You created a new storage account linked service, using the corresponding key vault secret name to access the connection string value at runtime. In this section, you will create a parameterized linked service for blob storage accounts, enabling you to pass different secret names in at runtime – this will enable the linked service to be used to connect to different storage accounts.

You may already have noticed that the linked service editor lacks some of the features you have been using for dynamic properties. Input fields to define parameters are not always present, and when editing properties you may not be offered a link to launch the expression builder. This does not mean that linked services lack support for parameters or expressions, just that they are not always supported by the ADF UX.

Recall that all ADF resource definitions are stored as JSON files – you can see these files in your Git repository. Linked service parameters and expressions can be configured by specifying them directly in a linked service's JSON definition.

1. Open the ADF UX management hub and select *Linked services* in the *Connections* section of its sidebar.

2. Click the *+ New* button to add a new linked service, then select the *Azure Blob Storage* data store and click *Continue*.
3. Call the new linked service “AnyBlobStorage” – it could be used to connect to any Azure storage account.
4. Using the button below the *Authentication method* dropdown, change the connection type to *Azure Key Vault*.
5. Scroll down to the bottom of the *New linked service* blade and expand the *Advanced* section. Tick the *Specify dynamic contents in JSON format* checkbox.
6. Beneath the *Specify dynamic contents in JSON format* checkbox is a text input area. Replace its “{}” default content with the JSON object shown in Listing 5-1. You will need to update the code with the name of your own key vault linked service, replacing `adfbyexamplekv` in the listing. (The location of the value is indicated in Figure 5-10.)
7. Click *Test connection* at the bottom of the blade. The ADF UX now prompts you to supply a value for the parameter “ConnectionSecret” – enter the name of the key vault secret that contains your storage account connection string, then click *OK*.
8. Figure 5-10 shows the completed blade after a successful connection test. When the test has successfully completed, click *Create* to save the new linked service.

The screenshot shows the Azure DevOps interface for creating a new linked service. The left sidebar lists various options like Connections, Linked services, and Integration runtimes. The main area is titled "New linked service (Azure Blob Storage)". It has fields for "Name" (AnyblobStorage), "Description" (Connect to any blob storage account with a connection string in the key vault), "Connect via integration runtime" (AutoResolveIntegrationRuntime), and "Test connection" (selected to "To linked service"). A checkbox "Specify dynamic contents in JSON format" is checked, and the JSON code editor displays the following configuration:

```
1 "properties": {  
2   "parameters": {  
3     "ConnectionSecret": {  
4       "type": "String"  
5     }  
6   },  
7   "annotations": [],  
8   "type": "AzureBlobStorage",  
9   "typeProperties": {  
10     "connectionString": {  
11       "type": "AzureKeyVaultSecret",  
12       "store": {  
13         "referenceName": "adfbbyexamplekv",  
14       }  
15     }  
16   }  
17 }
```

The line "referenceName: "adfbbyexamplekv"" is highlighted with a red box. At the bottom, there are "Create", "Back", "Test connection" (with a green checkmark indicating success), and "Cancel" buttons.

Figure 5-10. Linked service JSON configuration indicating the key vault linked service name

Listing 5-1. Dynamic linked service definition in JSON

```
{
  "properties": {
    "parameters": {
      "ConnectionSecret": {
        "type": "String"
      }
    },
    "annotations": [],
    "type": "AzureBlobStorage",
    "typeProperties": {
      "connectionString": {
        "type": "AzureKeyVaultSecret",
        "store": {
          "referenceName": "adfbyexamplekv",
          "type": "LinkedServiceReference"
        },
        "secretName": "@linkedService().ConnectionSecret"
      }
    }
  }
}
```

Take a closer look at Listing 5-1 to understand its components. It consists of a JSON object with a single `properties` field that contains

- A `parameters` field, defining a single string parameter called `ConnectionSecret`
- A `type` field, indicating that this linked service is a blob storage connection
- A `typeProperties` field specifying the blob storage connection string

The blob storage connection string also has a `type`, indicating that the connection string's value is stored externally in a key vault secret. `store` identifies the key vault

(using its linked service connection name), and `secretName` specifies the secret in the vault. The value of `secretName` is referenced in an expression:

```
@linkedService().ConnectionSecret
```

which refers to the linked service's `ConnectionSecret` parameter, defined in the service's parameters field described earlier. The definition of the parameter and its use in the expression is what allows you to inject the key vault secret name at runtime, from outside the linked service definition.

Note As in the case of datasets, the `linkedService()` syntax can only be used inside a linked service definition and only to refer to the linked service's own parameters.

Increase Dataset Reusability

Use your parameterized linked service to implement a dataset with greater reusability as follows:

1. Return to the authoring workspace and create a new dataset by cloning the parameterized dataset from earlier in the chapter. Name it "ABS_JSON" – it will be able to represent any JSON file(s) in any Azure blob storage account.
2. On the cloned dataset's *Parameters* configuration tab, add a fourth String parameter called "LinkedServiceConnectionSecret."
3. On the dataset's *Connection* configuration tab, choose the "AnyBlobStorage" linked service from the *Linked service* dropdown.
4. Selecting a parameterized linked service displays a *Linked service properties* region below the *Linked service* dropdown. Populate the linked service's "ConnectionSecret" parameter with an expression that returns the value of the dataset's "LinkedServiceConnectionSecret" parameter.

5. Test your linked service parameter value by clicking *Test connection* (on the right of the *Linked service* dropdown). The ADF UX will prompt you for a value for “*LinkedServiceConnectionSecret*” – provide the name of the key vault secret containing your storage account connection string. When the test has successfully completed, save your changes.

Figure 5-11 shows the configuration of the new parameterized dataset using the parameterized linked service. This dataset can be used to refer to any JSON file in any Azure blob storage account for which a connection string is stored in your key vault.

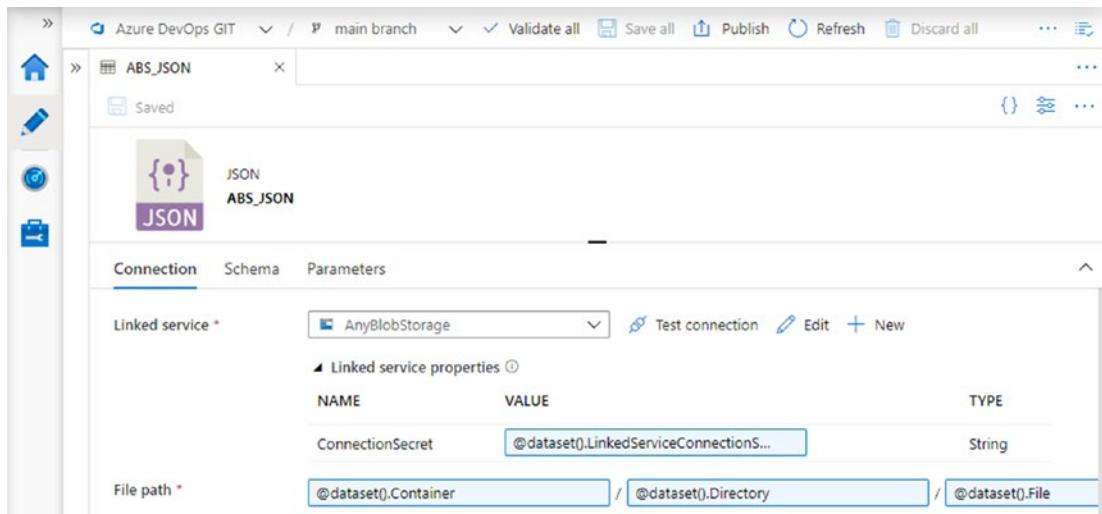


Figure 5-11. Parameterized dataset using a parameterized linked service

Use the New Dataset

To test the new dataset, clone the Sugar Cube pipeline you created earlier in the chapter. Change the Copy data activity’s source dataset to “ABS_JSON” and supply the two mandatory parameter values:

- Set “ContainerName” to the value “sampledata.”
- Set “LinkedServiceConnectionSecret” to the name of the key vault secret containing your storage account connection string.

Verify the results using your SQL client and/or the *Output* tab in the pipeline's configuration pane.

Why Parameterize Linked Services?

Parameterizing ADF datasets is a powerful way to avoid creating file-specific datasets, allowing a single dataset to be reused for files of the same type in different folder locations. Similarly, a parameterized linked service can be reused for different data stores of the same storage type.

One reason for doing this might be to reduce the number of linked services required in order to connect to multiple stores of the same type – for example, multiple blob storage accounts. Another important reason is that your Azure tenant is likely to contain multiple Azure Data Factory instances for development, testing, and production. Parameterizing linked services using a key vault allows you to centralize configuration information outside your data factory instance, reducing the complexity of copying factory resources between environments. The publication of ADF resources and management of multiple environments is discussed in greater depth in Chapter 10.

Use Pipeline Parameters

The sample data files you uploaded to your blob storage account in Chapter 2 include sales data for another customer of fictional confectionery manufacturer ABC – an online retailer named “Desserts4All.” Desserts4All reports sales of ABC’s products in CSV files, using exactly the same format as Sweet Treats.

You could reuse your Sweet Treats pipeline to load data for Desserts4All, but file location information for the pipeline – the *Wildcard folder path* and *Wildcard file name* – are hard-coded in the Copy data activity. In this section, you will use *pipeline parameters* to specify their values at runtime, creating a single pipeline that can process sales data from either vendor.

Create a Parameterized Pipeline

Start by cloning your Chapter 4 pipeline – “ImportSweetTreatsSales_Audit” – into the “Chapter5” folder. Name the new pipeline “ImportSTFormatFolder.”

CHAPTER 5 PARAMETERS

1. Below the authoring canvas, in the pipeline's configuration pane, you will find a *Parameters* tab – select it.
2. Use the *+ New* button to create a new parameter of type “String.” Name it “WildcardFolderPath.” Repeat the process to create a “String” parameter called “WildcardFileName,” giving it a default value of “*.csv”. Figure 5-12 shows the completed *Parameters* tab.

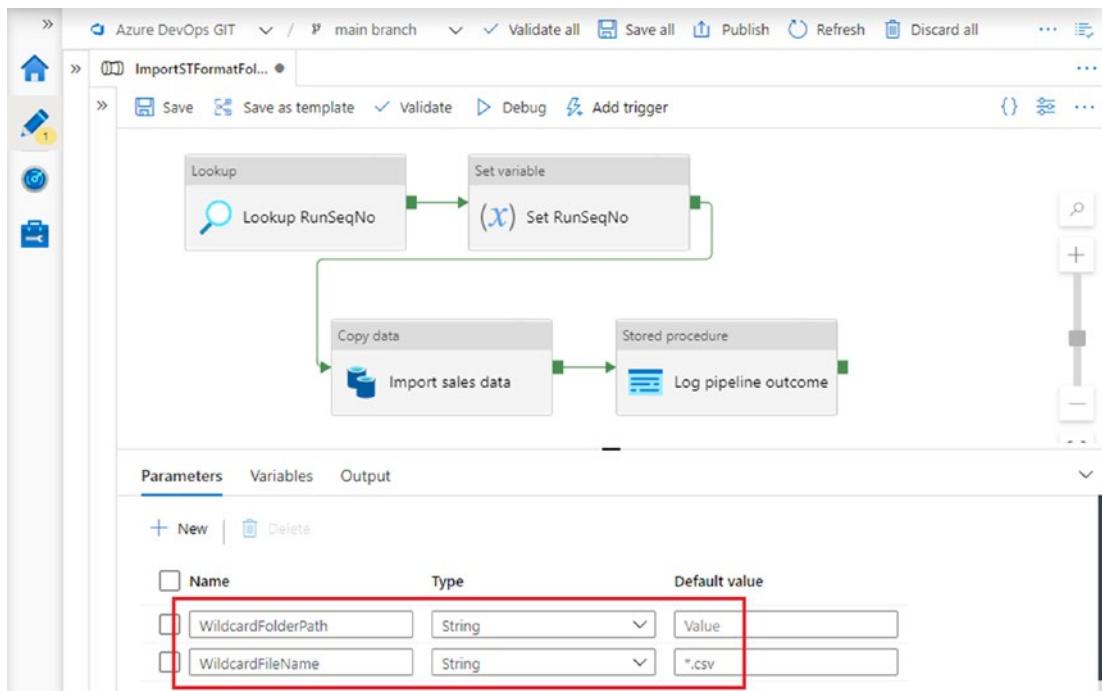


Figure 5-12. Pipeline parameter definitions

3. Select the pipeline's Copy data activity and open its *Source* configuration tab. Click the *Wildcard folder path* component of *Wildcard paths* and launch the expression editor.
4. Scroll down the editor to find the Parameters section – here you will find listed the pipeline parameters you created in step 2. Select the “WildcardFolderPath” parameter.

5. The expression `@pipeline().parameters.WildcardFolderPath` appears in the expression pane – pipeline parameters are referred to using the syntax `pipeline().parameters.ParameterName`. Click *Finish*.
6. Repeat steps 4 and 5 for the *Wildcard file name* component of *Wildcard paths*, this time choosing the “`WildcardFileName`” parameter.

Note The `pipeline()` syntax can be used in any of a pipeline’s activities and is used to refer to system variables (e.g., `pipeline().RunId`) as well as parameters. This is why pipeline parameters must be addressed using the syntax `pipeline().parameters.ParameterName`.

Run the Parameterized Pipeline

When you run the parameterized pipeline, the ADF UX prompts you to supply values for the pipeline’s parameters.

1. Provide parameter values required for Sweet Treats data (Figure 5-13 shows the relevant inputs) and click *OK* to run the pipeline. Notice that “`WildcardFileName`” is prepopulated with the default value you specified for the parameter.

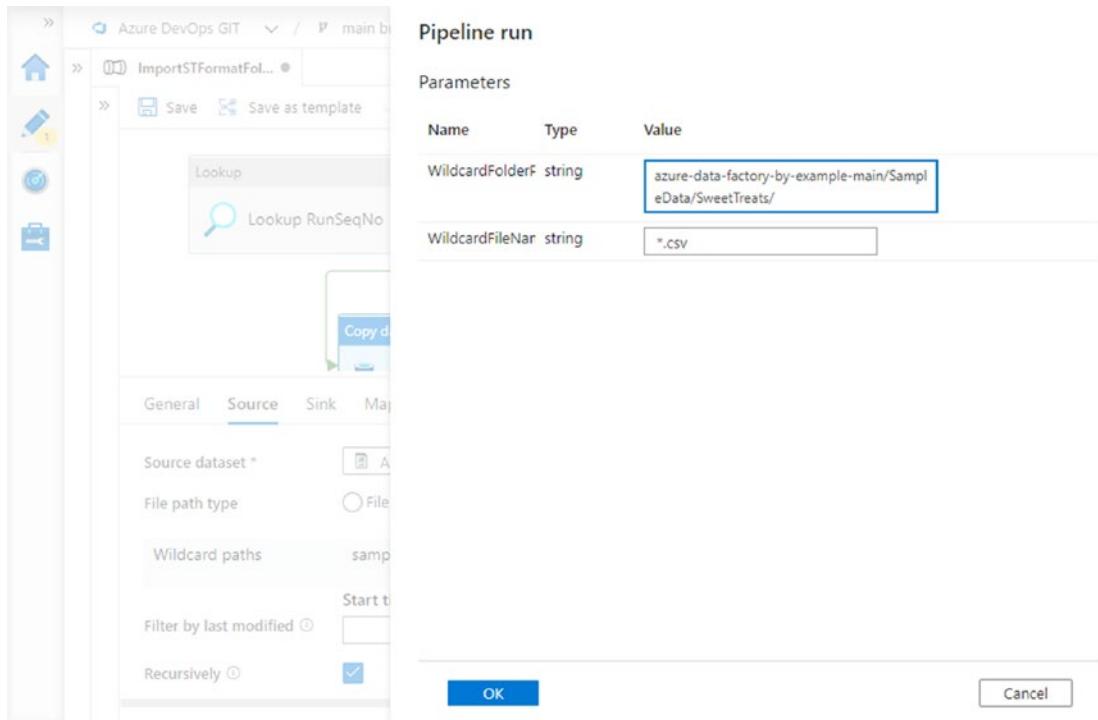


Figure 5-13. Parameter inputs required for the pipeline run

2. Inspect the contents of database tables [dbo].[PipelineExecution] and [dbo].[Sales_LOAD] to verify that Sweet Treats data has been loaded successfully.
3. Run the pipeline again, this time providing parameter values for Desserts4All data. For “WildcardFolderPath,” edit the value you used for Sweet Treats data, replacing the final “SweetTreats” folder name with “Desserts4All.” Use the same “WildcardFileName” value (“*.csv”).
4. Inspect the contents of database tables [dbo].[PipelineExecution] and [dbo].[Sales_LOAD] again, verifying that this time the data loaded relates to Desserts4All.

As you have just demonstrated, the new pipeline can now load data stored in the Sweet Treats file format from whichever folder you choose. Note that the Copy data activity in the new pipeline is still using the “ABS_CSV_SweetTreats” dataset. The pipeline runs successfully because the activity overrides the dataset’s directory and path

at runtime (and all your source data is in the same blob storage container), but this gives the pipeline's definition an inconsistent appearance.

A cleaner approach would be to create a reusable “ABS_CSV” dataset, similar to the “ABS_JSON” dataset you created earlier. Create a parameterized “ABS_CSV” dataset and test it now – you will need it again in Chapter 6.

Tip Don't forget to specify *First row as header* in the new “ABS_CSV” dataset.

Use the Execute Pipeline Activity

Inputting parameter values manually is acceptable in the context of an ADF UX session, but most pipeline executions do not take place interactively. In unsupervised pipeline runs, parameter values must be supplied by whichever process invokes the pipeline. In many cases, this process will be another ADF pipeline, using the *Execute Pipeline activity*.

In this section, you will create a new pipeline that uses the Execute Pipeline activity to run the “ImportSTFormatFolder” pipeline twice – once for Sweet Treats data and once for Desserts4All.

1. Create a new pipeline in the “Chapter5” folder and name it appropriately.
2. Drag an Execute Pipeline activity onto the authoring canvas. This activity is found in the *General* section of the activities toolbox.
3. On the activity's *Settings* configuration tab, select “ImportSTFormatFolder” from the *Invoked pipeline* dropdown. When you do so, input fields for the pipeline's parameters appear immediately below.
4. Ensure that the *Wait on completion* checkbox is ticked, then supply parameter values for Sweet Treats data, as before.
5. Repeat steps 2–4, this time supplying parameter values for Desserts4All data.
6. Create an activity dependency between the two activities (to prevent them from running simultaneously). Figure 5-14 shows the configured pipeline and the Execute Pipeline activity's *Settings* configuration tab.

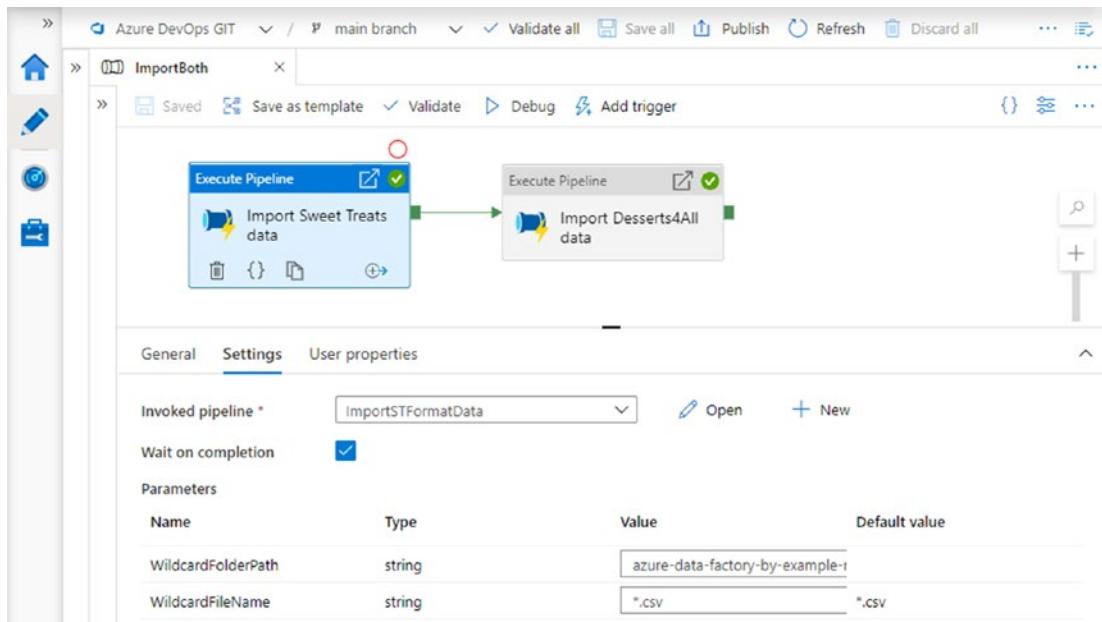


Figure 5-14. Sequential Execute Pipeline activities

Tip If you leave the WildcardFileName parameter blank, the Execute Pipeline activity passes an empty string to the pipeline “ImportSTFormatFolder” – the pipeline’s default parameter values are not substituted in this situation.

Run the pipeline and verify that two new log records have been created in database table [dbo].[PipelineExecution]. Table [dbo].[Sales_LOAD] will only contain data loaded by the second Execute Pipeline activity, because the loading pipeline’s Copy data activity truncates the sink table before copying in new data. The pipelines you have developed so far do not persist loaded data, but in Chapter 6 you will extend the loading pipeline to transfer data into a table for permanent storage.

The *Output* tab on the pipeline’s configuration pane contains two rows, one for each Execute Pipeline activity. The output from each activity includes the pipeline run ID of the pipeline execution it invoked, displayed as a link – follow the link to view activity run information for the invoked pipeline.

For SSIS developers When choosing a pipeline for the Execute Pipeline activity, you are offered a fixed dropdown list – there is no option to launch the expression editor. Unlike SSIS’s Execute Package Task, you cannot specify a pipeline name using a parameter or variable. (Even in SSIS, this requires the value of the task’s DelayValidation property to be “true” – the concept of delayed validation is not supported in ADF.)

Parallel Execution

The activity dependency between the two Execute Pipeline activities forces them to run sequentially – without it, the two activities will run simultaneously, in parallel. Even with the dependency in place, execution is only sequential because the first activity waits for its pipeline execution to complete – this is the effect of the *Wait for completion* checkbox.

In this case, the two activities must run sequentially because the invoked pipeline is poorly designed for parallel execution. Both executions truncate the sink table independently, so running them in parallel might allow one execution to truncate data while the other is in the process of loading it. In Chapter 6, you will revise the loading pipeline’s design to make it safe for parallelization.

Global Parameters

Unlike the other parameter types covered earlier in the chapter, *global parameters* are not a means of injecting different values at runtime. The role of global parameters is instead to provide *constant* values, shared by all pipelines. For example, if your Azure tenant contains separate Azure Data Factory instances for development, testing, and production, a global parameter is a convenient way to store instance-specific information.

Global parameters are created and edited on the *Global parameters* page of the ADF UX management hub. Figure 5-15 shows parameters being used to identify the factory’s resource group and to indicate that this is the development environment.

Name	Type	Value
EnvironmentName	string	Development
ResourceGroupName	string	adfbyexample-rg

Figure 5-15. Global parameter editor in the ADF UX management hub

Global parameters can be referenced in expressions within the scope of pipeline activities and are available for selection in the ADF expression builder. The syntax used to refer to a global parameter is

```
pipeline().globalParameters.ParameterName
```

For example, the “EnvironmentName” global parameter shown in Figure 5-15 is referred to in an ADF expression using `pipeline().globalParameters.EnvironmentName`.

Chapter Review

In this chapter, you defined parameters for pipelines, datasets, and linked services. Parameters allow a factory resource to require that certain data values be provided at runtime, exploiting those values by using them in ADF expressions. You also saw that resource JSON definitions can be edited directly – the ADF UX is usually a more user-friendly way to manage resources, but the ability to edit JSON directly is sometimes useful.

A common use case for parameters is to allow factory resources to be defined generically. This allows reuse, reducing clutter in your data factory and preventing unnecessary reimplementation of frequently used components. Reusable linked services may require sensitive connection information to be passed in at runtime – Azure Key Vault secrets provide a secure way to do this.

The degree to which you choose to parameterize factory resources depends heavily on your own use cases and usage patterns. As you continue to use ADF, you will find other places in which design decisions can only be made in the context of your own specific technical requirements.

Key Concepts

- **Runtime parameter:** A placeholder in a factory resource definition for a value substituted at runtime. Pipeline, dataset, and linked service parameters are runtime parameters; global parameters are not.
- **Optional parameter:** A runtime parameter can be made optional by defining a default value for it. If a value is not supplied at runtime, the default value is substituted for the parameter instead.
- **Reusability:** Runtime parameters enable factory resources to be defined in a reusable way, using different parameter values to vary resource behavior.
- **Global parameter:** A constant value, shared by all pipelines, not expected to change frequently. Global parameters are referred to in ADF expressions, within the scope of pipeline activities, using the syntax `pipeline().globalParameters.ParameterName`.
- **Pipeline parameter:** Runtime parameter for an ADF pipeline. Pipeline parameters are referred to in ADF expressions, within the scope of pipeline activities, using the syntax `pipeline().parameters.ParameterName`.
- **Dataset parameter:** Runtime parameter for an ADF dataset. Dataset parameters are referred to in ADF expressions, within the scope of the dataset, using the syntax `dataset().ParameterName`.
- **Linked service parameter:** Runtime parameter for an ADF linked service. Dataset parameters are referred to in ADF expressions, within the scope of the linked service, using the syntax `linkedService().ParameterName`. The ADF UX does not always support parameter management for linked services, but parameters can be defined and used by editing a linked service's JSON definition.

- **Execute Pipeline activity:** ADF pipeline activity used to execute another pipeline within the same data factory instance.
- **Azure Key Vault:** A secure repository for secrets and cryptographic keys.
- **Secret:** A name/value pair stored in an Azure Key Vault. The value usually contains sensitive information such as service authentication credentials. A secure way to handle this information is to refer to the secret by name – a service that requires the secret's value may retrieve it from the vault by name if permitted to do so.
- **Service principal:** An identity created for use with an application or service – such as Azure Data Factory – enabling the service to be identified when external resources require authentication and authorization.
- **Managed identity:** A managed identity associates a service principal with an instance of Azure Data Factory (or other Azure resources). A *system-assigned managed identity* is created automatically for new ADF instances created in the Azure portal and is automatically removed if and when its factory is deleted.
- **Access policy:** A key vault's access policy defines which service or user principals are permitted to access data stored in the vault.

For SSIS Developers

ADF pipeline parameters are conceptually similar to SSIS package parameters. As in SSIS packages, pipeline parameters are specified as part of the pipeline definition and can be referred to anywhere within the pipeline's activities.

Data factory linked services most closely resemble SSIS's project-level connection managers, but behave somewhat differently. SSIS connection managers do not support external injection of parameters – package-level connection managers are parameterized using SSIS expressions that reference package parameters or variables.

The concept of a variable or parameter scoped to the exact level of the connection manager is not found in SSIS. Project-level connection managers can make use of project parameters, but the scope of these is project-wide (much like the factory-wide scope of ADF's global parameters).

ADF's Execute Pipeline activity serves the same purpose as SSIS's Execute Package Task, but invoked pipelines can only be specified at design time. SSIS allows you to override this constraint by setting the Execute Package Task's DelayValidation property to true, but ADF activities have no comparable setting.

SSIS packages are frequently executed using SQL Server Agent jobs, requiring the agent's service account (or proxy) to be granted access to external resources. An ADF instance's managed identity serves the same purpose as a domain account used to run the SQL Server agent service – it enables processes executed by the factory to be authenticated and authorized against external data stores and services.

CHAPTER 6

Controlling Flow

In Chapter 4, you began building ADF pipelines containing more than one activity, controlling their order of execution using activity dependencies configured between them. Activity dependencies are among a range of tools available in ADF for controlling a pipeline's flow of execution.

In this chapter, you will develop more sophisticated control flows using activity dependencies with different *dependency conditions*. You will also encounter other ADF activities with the specific purpose of controlling flow. To begin with, you will create a new pipeline to use with some of these activities.

Create a Per-File Pipeline

The pipelines you built in previous chapters exploited the Copy data activity's ability to describe multi-file sources using directory and filename wildcards. This chapter's pipelines will introduce finer-grained file load controls, making use of a pipeline that loads a single specified file.

1. Create a clone of the “ImportSTFormatFolder” pipeline from Chapter 5 and move it into a new “Chapter6” pipelines folder. Name the new pipeline “ImportSTFormatFile.”
2. Open the new pipeline's *Parameters* configuration tab. Rename the two existing parameters to “Directory” and “File” and remove any default value settings.
3. Select the “Import sales data” Copy data activity, dismiss any prompt for parameter values, and open its *Source* tab. You will immediately see error messages relating to the original parameter names (which no longer exist) – remove the two invalid expressions. Change the source's *File path type* to “File path in dataset.”

4. Change the activity's *Source dataset*, selecting your "ABS_CSV" parameterized dataset from Chapter 5. In the *Dataset properties* region, ensure that the value of the "Container" parameter is set to "sampleddata", then update the "Directory" and "File" dataset parameter values to use the pipeline parameters of the same names. Figure 6-1 shows the configuration of the Copy data activity's source.

5. On the *Settings* tab of the pipeline's first activity – the Lookup activity named "Lookup RunSeqNo" – extend the "Comments" parameter expression to include the file being loaded, for example, Pipeline @{pipeline().Pipeline} executed in @{pipeline().DataFactory} - loading file "@{pipeline().parameters.Directory}/@{pipeline().parameters.File}".

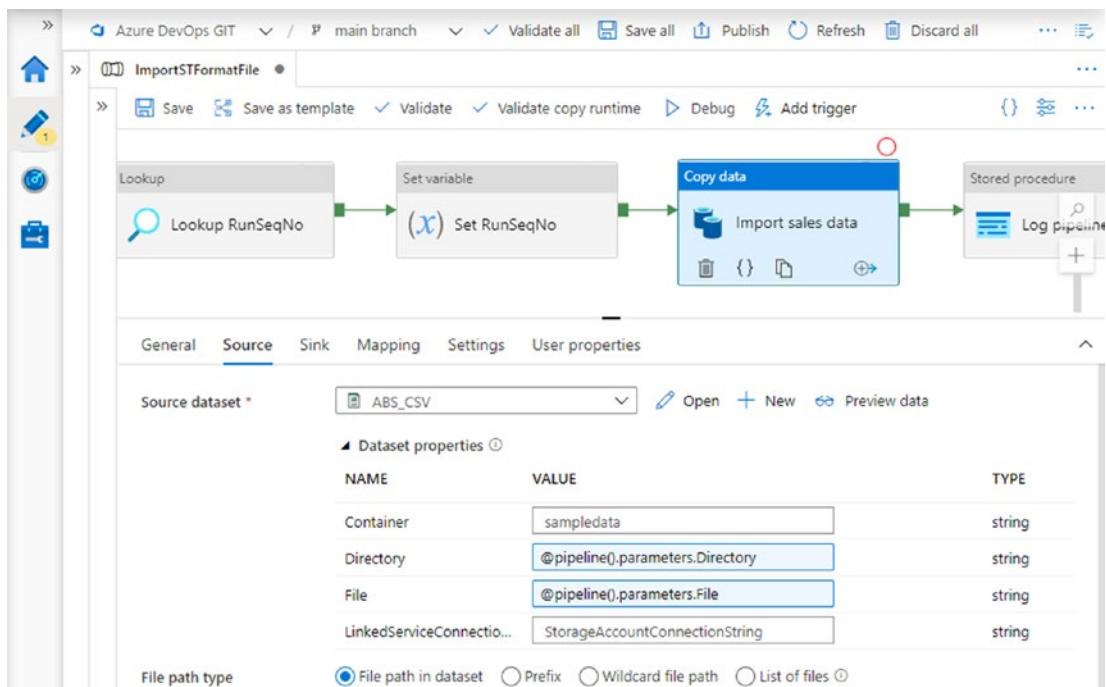


Figure 6-1. Copy data activity configured to load a single file

Save and run the pipeline. You will be prompted to supply values for the pipeline's "Directory" and "File" parameters – specify the full path and filename of a Sweet Treats or Desserts4All sales data file. When execution is complete, verify the results in the [dbo].[PipelineExecution] and [dbo].[Sales_LOAD] database tables.

Use Activity Dependency Conditions

A third retailer – “Naughty but Nice” – supplies sales data in the Sweet Treats format. The “NaughtyButNice” folder (in the same place as the “SweetTreats” and “Desserts4All” folders) contains corresponding data files. Run the “ImportSTFormatFile” pipeline again, this time with these parameters:

- For “Directory,” enter “azure-data-factory-by-example-main/SampleData/NaughtyButNice”.
- Set “File” to “NBN-202006.csv”.

Tip You will be rerunning the pipeline using these values a number of times – you may wish to set them as the parameters’ default values.

When you run the pipeline, it will fail, reporting a type conversion error. This is due to some badly formatted data – a comma is missing between the first two fields of a few records in the middle of the file.

The execution record in [dbo].[PipelineExecution] is incomplete for the pipeline’s execution, because its final Stored procedure activity was not reached. Using the log table alone, it is not possible to determine whether the pipeline has failed or is simply still running. You can improve this situation by logging the pipeline’s failure.

1. Create a copy of the activity “Log pipeline outcome,” the pipeline’s final Stored procedure activity. To do this, select the activity to reveal its authoring controls, then click the *Clone* button. Rename the new activity “Log pipeline failure.”
2. The new activity is to be executed only if the Copy data activity fails. To configure this, select the Copy data activity. From its authoring controls, click the *Add output* button and choose the *Failure* dependency condition from the popup menu (shown in Figure 6-2).
3. A red handle appears on the Copy data activity’s right-hand edge. Click and drag it onto the “Log pipeline failure” activity.

- The “Log pipeline failure” activity will now execute only if the Copy data activity fails. Open the activity’s *Settings* configuration tab and scroll down to the *Stored procedure parameters* section. Change the value of the “RunStatus” parameter to “Failed.”

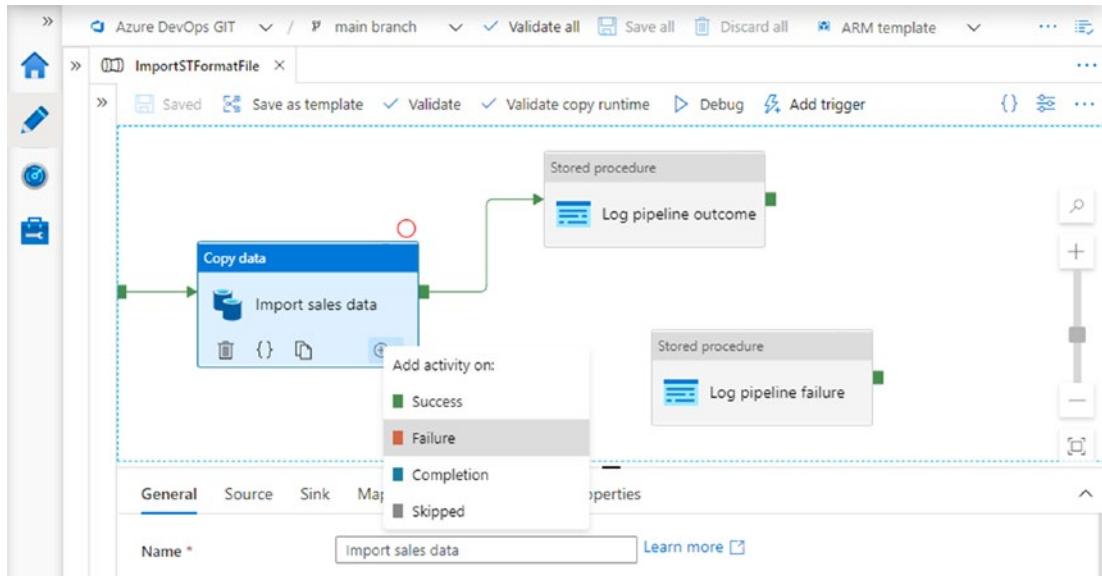


Figure 6-2. Adding a failure dependency condition to the Copy data activity

Save and run the pipeline, using the same parameter values as before. In the pipeline configuration pane’s *Output* tab, you will notice that the Copy activity data fails but that the failure-logging task is then executed. The log record in the database table [dbo].[PipelineExecution] shows the pipeline’s execution with a [RunStatus] value of “Failed.”

Tip As you drag the canvas viewport around to see different activities in a large pipeline, it is surprisingly easy to lose your place and end up looking at an empty screen! To recover this, right-click somewhere in canvas whitespace and select *Zoom to fit*. After the canvas has recentered, choose *Reset zoom level* from the right-click popup menu to restore activities to their usual sizes.

Explore Dependency Condition Interactions

The popup menu shown in Figure 6-2 contains the four possible values for an activity dependency condition: *Success*, *Failure*, *Completion*, and *Skipped*. By combining the four condition values in different ways, you can achieve sophisticated flow control in your pipelines. In this section, you will create a simple testing pipeline to allow you to explore how different combinations of dependency condition interact with one another.

1. Create a new pipeline and define a pipeline variable of type String. Name the variable.
2. Drag three Set variable activities onto the authoring canvas and connect them into a chain with *Success* activity dependencies.
3. On the first activity's *Variables* tab, set the value of your variable to the expression `@string(int('not an int'))`. This expression attempts to cast the string “not an int” to an integer, which will fail. (The enclosing string conversion is necessary to match the variable's type.)
4. Configure the second and third activities to update the same variable, this time with valid string values of your choice.

Figure 6-3 shows the chain of Set variable activities along with the first activity's *Variables* configuration tab.

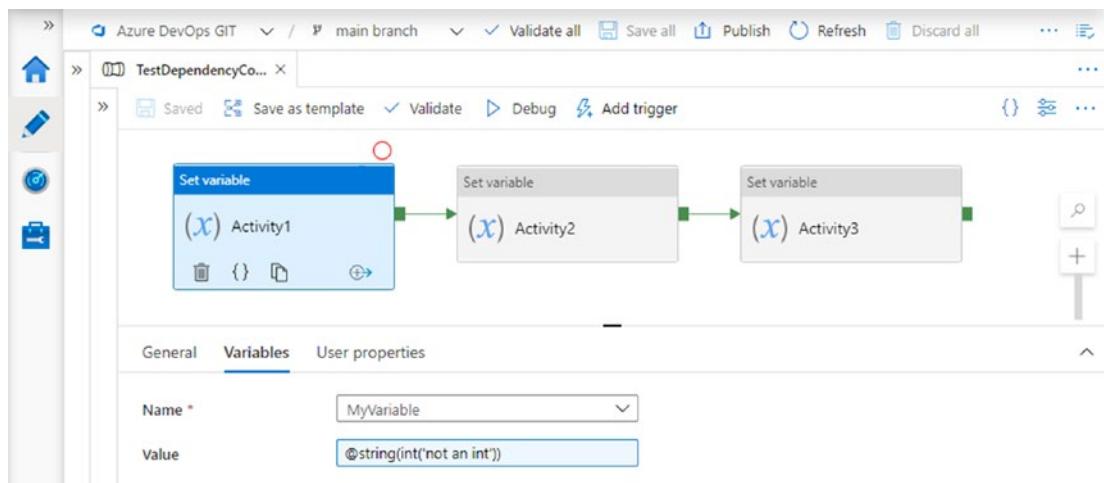


Figure 6-3. Set variable activity configured to produce failure

Run the pipeline. As you will expect, the first activity fails, and the next two do not run.

Understand the Skipped Condition

Right-click over the dependency between the second and third activities and change the dependency condition to *Skipped*. Run the pipeline again. This time

- Activity2 is not executed because its dependency condition requires Activity1 to succeed.
- Activity3 is executed because its dependency condition requires Activity2 to have been skipped.

Understand the Failed Condition

Right-click over the dependency between the first and second activities and change the dependency condition to *Failure*. Run the pipeline, noticing that

- Activity2 is executed because its dependency condition requires Activity1 to fail.
- Activity3 is no longer executed because Activity2 has not been skipped.

Combine Conditions

Drag the green handle from the second activity and drop it onto the third, creating a *Success* dependency condition, in addition to the existing *Skipped* condition. Figure 6-4 shows the two conditions between the activities. Run the pipeline again to discover that all three activities are now executed.



Figure 6-4. Two dependency conditions between Activity2 and Activity3

Change the condition between the first two activities back to *Success* and run the pipeline again. This time, Activity2 is not executed – Activity3 still is, because the *Skipped* condition is met.

If multiple dependency conditions are set between a pair of activities, the second activity will be executed if *any* of the conditions are met. In this example, Activity3 will run if Activity2 succeeded OR Activity2 was skipped.

Create Dependencies on Multiple Activities

Add a fourth Set variable task to your pipeline and configure it to assign a valid string value to your variable. Create a *Success* dependency condition by dragging the new activity's dependency handle to Activity2, as shown in Figure 6-5. Run the pipeline. On this run

- Activity2 is not executed because Activity1 fails.
- Activity3 is executed because Activity2 is skipped.

When an activity like Activity2 is dependent on more than one prior activity run, it is only executed if dependency conditions are met from *all* of its predecessors. In this example, Activity2 will run only if Activity1 succeeded AND Activity4 succeeded.

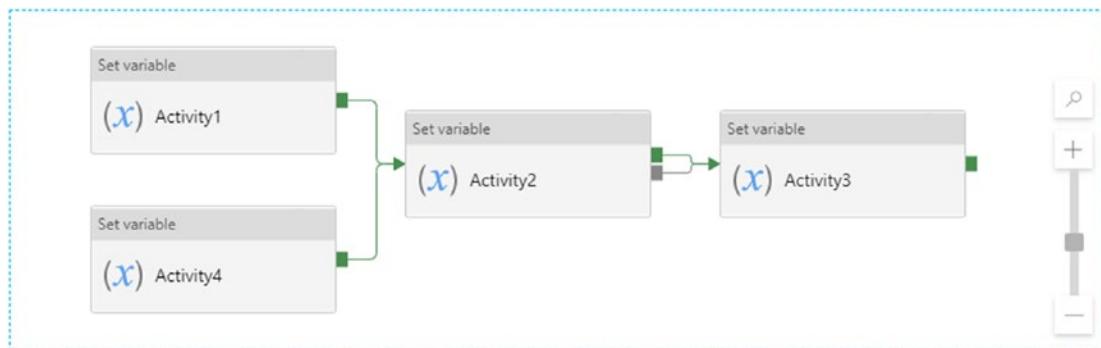


Figure 6-5. Activity dependent on multiple predecessors

Understand the Completion Condition

Change the dependency condition between the failing activity and the second activity to *Completion* and run the pipeline. This time, all four activities are executed.

Completion means “either Success or Failure” – creating the *Success* and *Failure* conditions separately has exactly the same effect. Activity2 is executed in this case because the following combined condition is true:

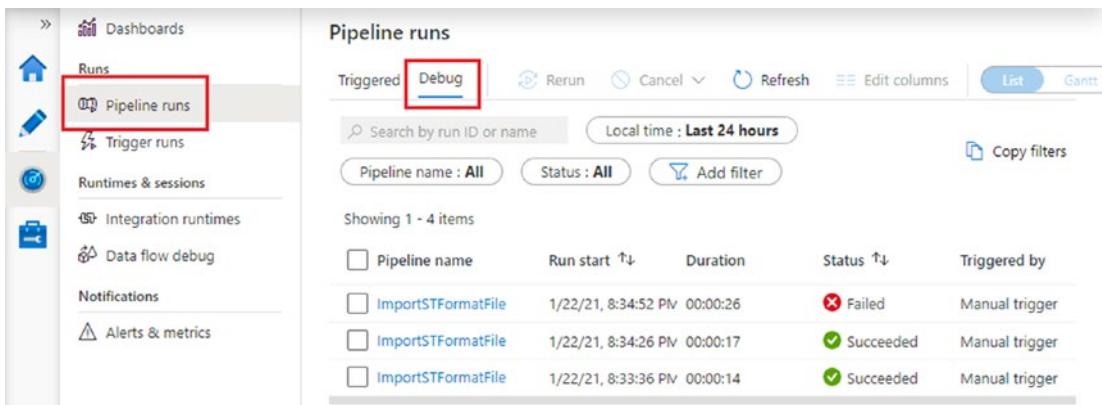
(Activity1 succeeded OR Activity1 failed) AND Activity4 succeeded

For SSIS developers ADF activity dependency conditions are comparable to SSIS precedence constraints, with two main differences. First, for an activity to be executed, a dependency condition must be met for every activity on which it depends – ADF activity dependencies have no equivalent to SSIS precedence constraints’ *LogicalAnd* property. Second, SSIS supports the use of expressions in precedence constraints, while ADF does not.

Understand Pipeline Outcome

You have been using the *Output* tab on the pipeline’s configuration pane to inspect the outcomes of pipeline activities. In addition to individual activity outcomes, a pipeline execution has an overall outcome of its own, reported in the ADF UX monitoring experience. To open the monitoring experience, click the *Monitor* button (gauge icon) in the navigation sidebar.

The monitoring experience is discussed in depth in Chapter 12. For the time being, select the *Pipeline runs* page, then ensure that the *Debug* tab is selected – the relevant controls are indicated in Figure 6-6.



The screenshot shows the ADF UX monitoring interface. On the left, a sidebar menu includes options like Dashboards, Runs (selected), Runtimes & sessions, Integration runtimes, Data flow debug, Notifications, and Alerts & metrics. Under the Runs section, 'Pipeline runs' is highlighted with a red box. The main area is titled 'Pipeline runs' and shows a table of recent runs. The 'Triggered' tab is selected, and the 'Debug' tab is also highlighted with a red box. The table displays four rows of data:

Pipeline name	Run start	Duration	Status	Triggered by
ImportSTFormatFile	1/22/21, 8:34:52 PM	00:00:26	Failed	Manual trigger
ImportSTFormatFile	1/22/21, 8:34:26 PM	00:00:17	Succeeded	Manual trigger
ImportSTFormatFile	1/22/21, 8:33:36 PM	00:00:14	Succeeded	Manual trigger

Figure 6-6. Pipeline debug runs in the ADF UX monitoring experience

Figure 6-6 shows the result of running the “ImportSTFormatFile” pipeline three times, the third time loading the Naughty but Nice file “NBN-202006.csv”, as you did earlier in the chapter. This pipeline run is reported to have failed, but not simply because the Copy data activity failed. It is useful to be able to understand the combination of activity outcomes that produces this result.

A pipeline’s execution outcome is determined as follows:

- The outcome of every “leaf” activity is inspected. A leaf activity is an activity that has no successor activities in the pipeline. Importantly, if a leaf activity is skipped, its predecessor is considered to be a leaf activity for the purpose of evaluating the pipeline’s outcome.
- If any leaf activity has *Failed*, the pipeline’s outcome is also *Failed*.

Figures 6-7 and 6-8 illustrate the difference between these scenarios. In Figure 6-7, if Activity1 fails, then Activity2 is executed and Activity3 is skipped. Because Activity3 is a skipped leaf, Activity1 is considered to be a leaf – and because it has failed, the pipeline itself fails.

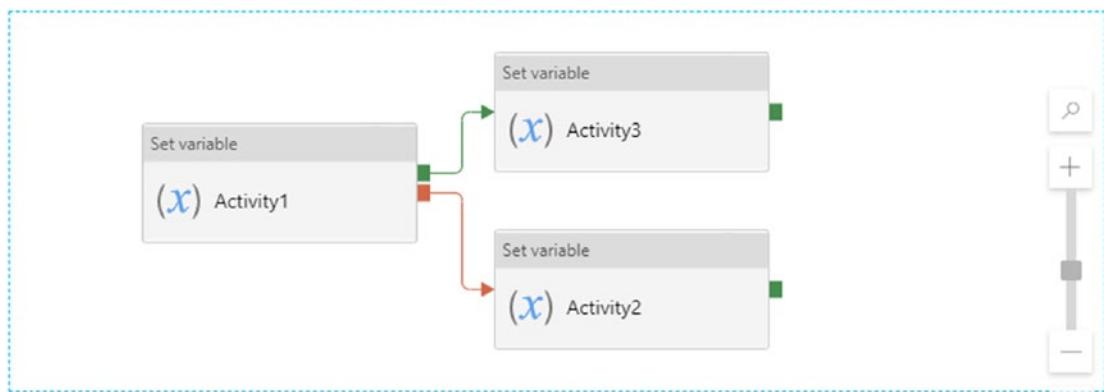


Figure 6-7. When Activity1 fails, the pipeline fails

Figure 6-8 shows the same pipeline with Activity3 removed. In this case, if Activity1 fails, then Activity2 is executed as before, but no activity is skipped. Activity3 is the only leaf activity, so, assuming it succeeds, the pipeline also succeeds.

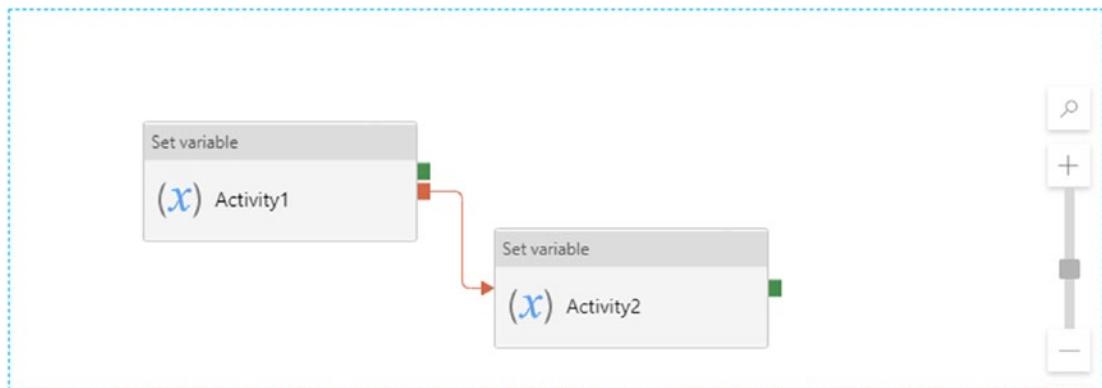


Figure 6-8. When *Activity1* fails, the pipeline succeeds

In the case of the “ImportSTFormatFile” pipeline, one leaf activity is always skipped – if the Copy data activity succeeds, the “Log pipeline failure” activity is skipped, but if the copy fails, “Log pipeline outcome” is skipped instead. This means that the pipeline’s Copy data activity is always considered to be a leaf activity (as the predecessor of whichever Stored procedure activity has been skipped). This has the effect – in this example – that the pipeline’s overall outcome is determined by the success or failure of the Copy data activity.

Figure 6-9 shows an alternative version of “ImportSTFormatFile,” modified to use a single Stored procedure activity with a *Completion* dependency condition. An ADF expression is used to calculate the value of the stored procedure’s “RunStatus” parameter (indicated in the figure). This version of the pipeline has only one leaf activity – “Log pipeline outcome” – which is executed whether the Copy data activity succeeds or not. Assuming that the logging activity succeeds, the pipeline is reported to succeed even if the copy fails, because all its leaf activities have succeeded.

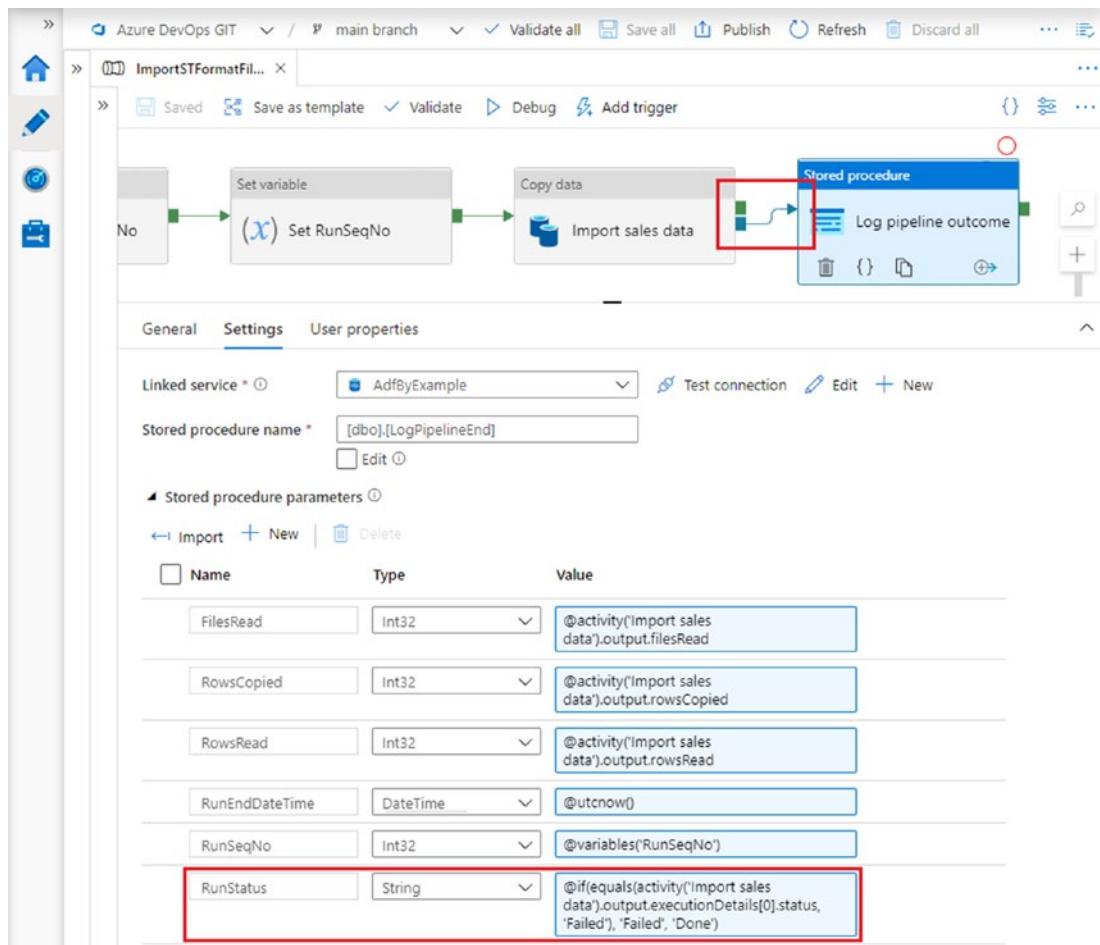


Figure 6-9. A single Stored procedure activity reporting either success or failure

For SSIS developers By default, any error that occurs during SSIS package execution causes the package to fail, but you can modify this behavior by preventing errors from propagating to the calling package or process. The way to meet this requirement in ADF is to understand the behavior of activity dependency conditions and to organize them to achieve the desired effect.

Raise Errors

Understanding why a pipeline is reported to fail allows you to control propagation of errors raised during a pipeline's execution. In contrast, sometimes you may wish to raise errors of your own, for example, if some invalid system state is detected.

At the time of writing, no Azure Data Factory activity enables you to raise errors directly – but a common approach is to call another activity in such a way that an error occurs. The forced failure of the Set variable activity in the previous example (by attempting to cast a nonnumeric string to an integer) is an example of this pattern. Another example is the use of T-SQL's RAISERROR or THROW statement to cause a Stored procedure or Lookup activity to fail.

In Chapter 3, you encountered an issue with schema drift, where a renamed field in the Sugar Cube JSON schema caused product names to disappear from loaded data. The effect of this was to populate the [Product] field in [dbo].[Sales_LOAD] with NULLs in rows relating to September 2020 – this is an example of invalid state that you might choose to raise as an error.

[Listing 6-1](#) provides a revised version of the [dbo].[LogPipelineEnd] stored procedure. The procedure inspects the database table's [Product] field, using RAISERROR to cause the loading pipeline to fail if NULLs are encountered. This provides a basic data quality check, converting undesirable data features into detectable pipeline failures. If you wish, modify your existing stored procedure using the script provided – to test it, you will need to create a new pipeline, combining the Sugar Cube data load from Chapter 3 with the logging functionality you developed after that.

Listing 6-1. [dbo].[LogPipelineEnd] raises an error if [Product] is NULL

```
ALTER PROCEDURE [dbo].[LogPipelineEnd] (
    @RunSeqNo INT
    , @RunEndDate DateTime
    , @RunStatus VARCHAR(20)
    , @FilesRead INT
    , @RowsRead INT
    , @RowsCopied INT
) AS

UPDATE dbo.PipelineExecution
SET RunEndDateTime = @RunEndDate
```

```

, RunStatus = @RunStatus
, FilesRead = @FilesRead
, RowsRead = @RowsRead
, RowsCopied = @RowsCopied
WHERE RunSeqNo = @RunSeqNo;

IF EXISTS (
    SELECT * FROM dbo.Sales_LOAD
    WHERE [Product] IS NULL
)
RAISERROR('Unexpected NULL in dbo.Sales_LOAD.[Product]', 11, 1);

```

Use Conditional Activities

There are a wide variety of different ways to handle data errors, depending on factors ranging from business data requirements to technical process control. So far, in this chapter, you have implemented pipelines that report loading failures, using activity dependencies to control whether pipeline execution succeeds or fails under those conditions.

Whether the loading pipeline fails or not, format errors in the Naughty but Nice sales file “NBN-202006.csv” cause loading to be abandoned – when an error is encountered, no rows are loaded. An alternative approach is to load whatever valid data you can, diverting other rows elsewhere to be inspected and handled later. In this section, you will use a Copy data activity feature to divert errors, then use one of ADF’s *conditional activities* to load records into a separate database table for inspection.

Divert Error Rows

The Copy data activity includes a feature that allows you to redirect failed rows to a log file in blob storage. Configure that option as follows:

1. Clone the “ImportSTFormatFile” pipeline from earlier in the chapter. Name the clone “ImportSTFormatFile_Divert”, then open the Copy data activity’s *Settings* configuration tab.
2. Scroll down to the *Fault tolerance* multiselect dropdown and select “Skip incompatible rows.” Ensure that the *Enable logging*

checkbox is ticked, causing additional *Logging settings* to be displayed.

3. In *Logging settings*, choose your Azure blob storage linked service from the *Storage connection name* dropdown.
4. You will be prompted for a *Folder path* and – if you have chosen a parameterized linked service – required *Linked service properties*. Accept the default values provided for *Logging level* and *Logging Mode*.
5. Click the *Browse* button to the right of the *Folder path* field, choose the blob storage “output” container, then click *OK*. Edit the *Folder path* value, appending “/errors”.

Figure 6-10 shows the configured fault tolerance settings for the Copy data activity.

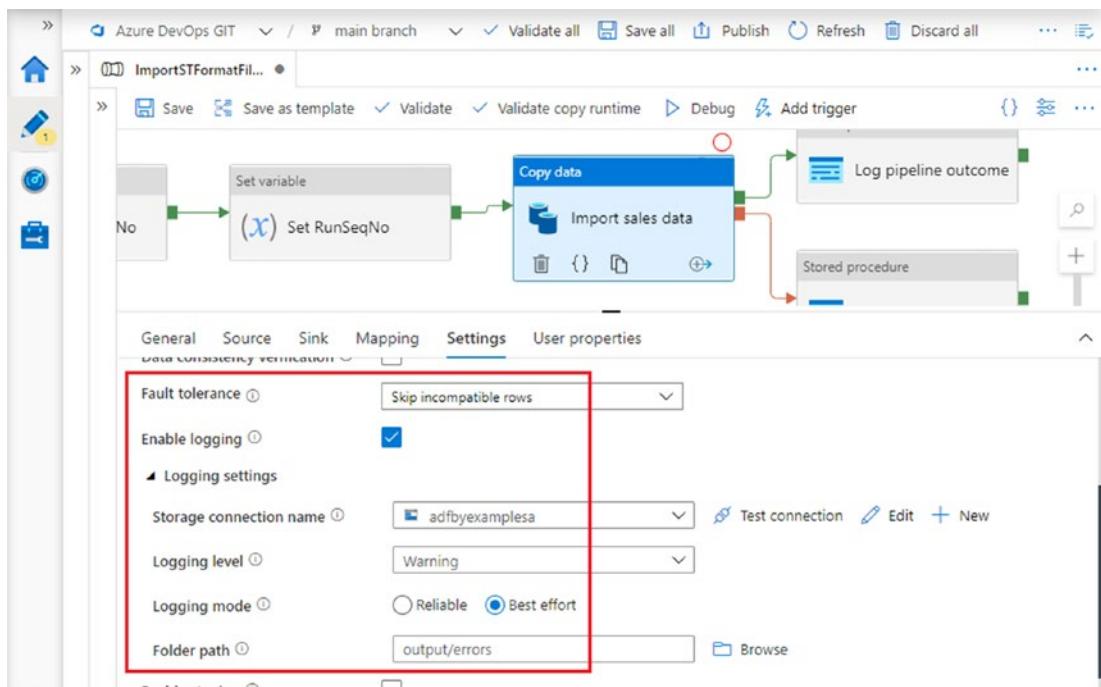


Figure 6-10. Copy data activity fault tolerance settings

Tip Copy data activity logging can be enabled independent of fault tolerance settings. Setting *Logging level* to “Info” causes information about successfully copied files to be logged, in addition to skipped files and rows.

Run the modified pipeline, again loading the badly formatted Naughty but Nice sales file “NBN-202006.csv”. This time, the Copy data activity’s new fault tolerance settings will enable it to succeed. When the pipeline has successfully completed, inspect the Copy data activity’s output JSON object (via the link in the *Output* tab on the pipeline’s configuration pane). An example is shown in Figure 6-11, including the following information:

- `rowsRead`: The number of rows read from the source file.
- `rowsCopied`: The number of rows copied to your Azure SQL DB. This is fewer than the number of rows read, but is greater than zero because correctly formatted rows have been loaded successfully into the database table.
- `rowsSkipped`: The number of rows not copied. These are the badly formatted rows in the source file.
- `logFilePath`: Skipped rows have been written into a log file specific to this activity run. The log file path ends in a GUID folder name – this is the activity run GUID which you can find on the right-hand side of the *Output* tab on the pipeline’s configuration pane.

Output

```

Learn more on output details ▾
{
  "dataRead": 7434,
  "dataWritten": 12612,
  "filesRead": 1,
  "sourcePeakConnections": 1,
  "sinkPeakConnections": 2,
  "rowsRead": 115,
  "rowsCopied": 99,
  "rowsSkipped": 16,
  "copyDuration": 6,
  "throughput": 1.21,
  "redirectRowPath": "",
  "logFilePath": "output/errors/copyactivity-logs/Import sales data/316e4ceb-16df-48fc-96d1-1deb93556c46/",
  "errors": [],
  "effectiveIntegrationRuntime": "DefaultIntegrationRuntime (UK South)"
}

```

Figure 6-11. Fault tolerance information in the Copy data activity's output JSON

Using Azure Storage Explorer, browse to the log file's location. Download the ".txt" file you find there and open it in a text editor. Figure 6-12 shows a similar log file, also produced by loading "NBN-202006.csv". The log is a CSV file containing five columns, the fourth of which – "OperationItem" – contains the entire diverted row.

As this is a quote-enclosed CSV file, every double quote character in the original record has been escaped by preceding it with another. Even so, you can easily see that the problem records are missing a comma between the date and retailer fields (indicated in Figure 6-12).

Timestamp	Level	OperationName	OperationItem	Message
2021-01-23 10:28:16.1724749	Warning	TabularRowSkip	"01-jun-2020""Naughty but Nice""",""Obelisk 35.24oz"",""37477.86"",""2534"",""NBN-202006.csv"",""35""	"Exception occurred when converting value '01-jun-2020'"Naughty but Nice"" for column name 'SalesMonth' from type 'String' (precision: , scale:) to type 'DateTime' (precision:255, scale:255). Additional info: String was not recognized as a valid DateTime."
2021-01-23 10:28:16.1774725	Warning	TabularRowSkip	"01-jun-2020""Naughty but Nice""",""Boho 2.82oz"",""4191.66"",""1314"",""NBN-202006.csv"",""35""	"Exception occurred when converting value '01-jun-2020'"Naughty but Nice"" for column name 'SalesMonth' from type 'String' (precision: , scale:) to type 'DateTime' (precision:255, scale:255). Additional info: String was not recognized as a valid DateTime."
2021-01-23 10:28:16.1774725	Warning	TabularRowSkip	"01-jun-2020""Naughty but Nice""",""Minotaur 13.39oz"",""3423.07"",""553"",""NBN-202006.csv"",""35""	"Exception occurred when converting value '01-jun-2020'"Naughty but Nice"" for column name 'SalesMonth' from type 'String' (precision: , scale:) to type 'DateTime' (precision:255, scale:255). Additional info: String was not recognized as a valid DateTime."
2021-01-23 10:28:16.1774725	Warning	TabularRowSkip	"01-jun-2020""Naughty but Nice""",""Ritterhutter	

Figure 6-12. Badly formatted source records in the Copy data activity fault log

Load Error Rows

A convenient way to inspect tabular log files is to collect their contents in a database table. In this section, you will add another Copy data activity to load the log file into an error table. Create an error log table in your Azure SQL Database using the code in Listing 6-2.

Listing 6-2. Create a loading error table

```
CREATE TABLE dbo>LoadingError (
    [Timestamp] DATETIME2 NULL
    , [Level] NVARCHAR(50) NULL
    , [OperationName] NVARCHAR(50) NULL
    , [OperationItem] NVARCHAR(4000) NULL
    , [Message] NVARCHAR(4000) NULL
);
```

Create a New Sink Dataset

Your existing Azure SQL Database dataset contains a hard-coded table name in its definition – [dbo].[Sales_LOAD]. Create a new dataset (and a “Chapter6” folder) to enable you to load log data into [dbo].[LoadingError], preferably by parameterizing the table’s schema and name in the dataset definition. You will get further use out of a parameterized table dataset in Chapter 7.

Revise the Source Dataset

Open the “ABS_CSV” dataset you created in Chapter 5 and inspect the properties on its *Connection* configuration tab. So far, you have only been making changes to *File path* settings, but the dataset’s properties can be configured to match many variations in file format. Notice in particular that the *Escape character* default is “Backslash (\)”. As you saw in Figure 6-12, the log file’s escape character is a double quote – change the selected value to “Double quote(“)”, then save the revised dataset.

When modifying shared resources such as datasets, take care not to introduce faults in pipelines or other resources that use them. The change is safe in this situation, because none of the source CSV files you have encountered so far contains escape characters. An alternative approach is to parameterize the escape character, giving the dataset parameter a default value of “\” to ensure backward compatibility.

Tip The *Properties* blade, available for datasets and other factory resources, includes a *Related* tab, indicating related factory resources. When modifying an existing resource, use the *Related* tab to identify resources that may be affected by changes.

Use the If Condition Activity

The new Copy data activity will use the log file path reported in the output of the original file copy. A log file is found in this location only when one or more records have been skipped – if no rows are skipped, then no file is created. Running the log file Copy data activity when the log file does not exist would itself cause an error, so the activity can only be permitted to run if rows have been skipped.

You can control this behavior using an *If Condition* activity, found in the *Iteration & conditionals* section of the activities toolbox.

1. Return to the “ImportSTFormatFile_Divert” pipeline, then drag an *If Condition* activity onto the authoring canvas. Rearrange activity dependencies to place it between the “Import sales data” Copy data and “Log pipeline outcome” (log success) Stored procedure activities, as shown in Figure 6-13. Name the activity “If rows skipped.”
2. Select the *If Condition*’s *Activities* configuration tab. It contains an *Expression* field and two *cases*: *True* and *False*. The two cases are containers for other activities – the activities contained in the *True* case are executed when the expression in the *Expression* field evaluates to true; those in the *False* case when it does not.
3. Click the *Expression* field to launch the expression builder. Enter an expression which evaluates to true if any rows were skipped by the prior Copy data activity. A possible choice is shown in Figure 6-13.
4. Below the *Expression* field, click the pencil icon on the right of the *True* case – this opens a new canvas showing the case’s activities, currently none. (Alternatively, use the pencil icon displayed in

the activity graphic on the authoring canvas.) The breadcrumb trail in the top left of the canvas contains the pipeline name, If Condition activity name, and *True activities*, indicating the part of the pipeline you are currently editing.

5. Drag a Copy data activity onto the authoring canvas and open its *Source* configuration tab. Select the “ABS_CSV” dataset and enter the value “output” for its “Container” parameter. Set the *File path type* option to “Wildcard file path,” then in *Wildcard folder path*, enter an expression to return the log file path from the earlier Copy data activity. Set *Wildcard file name* to “*”.

Note The log file path reported by the Copy data activity includes the blob storage container – your expression must remove this prefix. One expression that achieves this is @join(skip(split(activity('Import sales data')).output.logFilePath, '/'), 1), '/')�

6. Configure the Copy data activity’s *Sink* tab to specify the [dbo].[LoadingError] table using your new Azure SQL Database dataset.

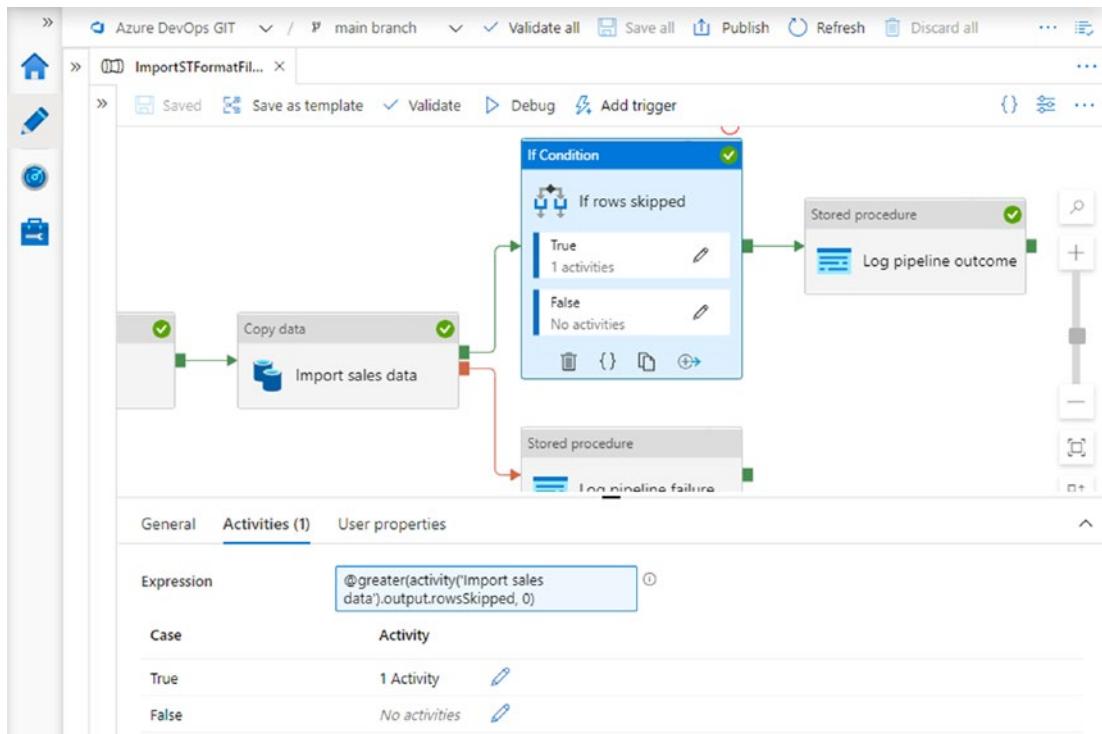


Figure 6-13. If Condition activity expression configuration

Note When you select an activity on the authoring canvas inside the If Condition, no breakpoint circle is displayed. The ADF UX does not permit you to set a breakpoint on any nested activity.

Run the Pipeline

Run the pipeline again to load the Naughty but Nice sales file “NBN-202006.csv”. Watch the pipeline configuration pane’s *Output* tab as the pipeline runs, refreshing it regularly to monitor progress. Notice that the pipeline’s six activities appear in a list, but that the If Condition activity’s status remains *In progress* until the Copy data activity it contains is queued, executes, and completes.

When the pipeline has completed successfully, verify that the database table [dbo].[LoadingError] now contains 16 rows copied from the log file. The database log table [dbo].[PipelineExecution] reports that 16 fewer rows were copied than were read.

Finally, run the pipeline again, loading a different Naughty but Nice sales file, for example, “NBN-202004.csv”. This file contains no formatting errors – the pipeline configuration pane’s *Output* tab shows that the If Condition activity is executed (to allow its expression to be evaluated), but that the log file copy is not.

Understand the Switch Activity

The If Condition is made up of

- Two *cases* – *True* and *False* – each of which contains zero or more activities.
- An *expression* which evaluates either to true or to false. When this expression is evaluated at execution time, the activities in the corresponding case are then executed.

The *Switch activity* is ADF’s other conditional activity. Switch is a generalization of the If Condition activity to more than two cases. It consists of

- Up to 25 cases – identified by different string values – each of which contains one or more activities.
- An expression which evaluates to a string value. When this expression is evaluated at execution time, the activities in the corresponding case are executed.

Additionally, the Switch activity always contains a *Default* case. If the evaluated expression matches none of the identified cases, activities in the Default case – of which there may be none – are executed.

Tip ADF does not permit conditional activities to be used within other conditional activities. If you required nested-if logic, the Switch activity may help you to implement it as a series of cases. An alternative workaround is to create your “inner” If Condition or Switch in one pipeline and your “outer” conditional activity in another – you can then use the Execute Pipeline activity to call the inner pipeline from the outer conditional activity.

CHAPTER 6 CONTROLLING FLOW

Figure 6-14 shows a configured Switch activity with four cases: “A,” “B,” “C,” and *Default*. When the expression `@variables('MyVariable')` evaluates to “A,” case A’s two activities are executed. The three activities in cases B and C are executed when the expression evaluates to “B” or “C,” respectively. If the expression evaluates to any value except “A,” “B,” or “C,” the *Default* case’s single activity is executed.

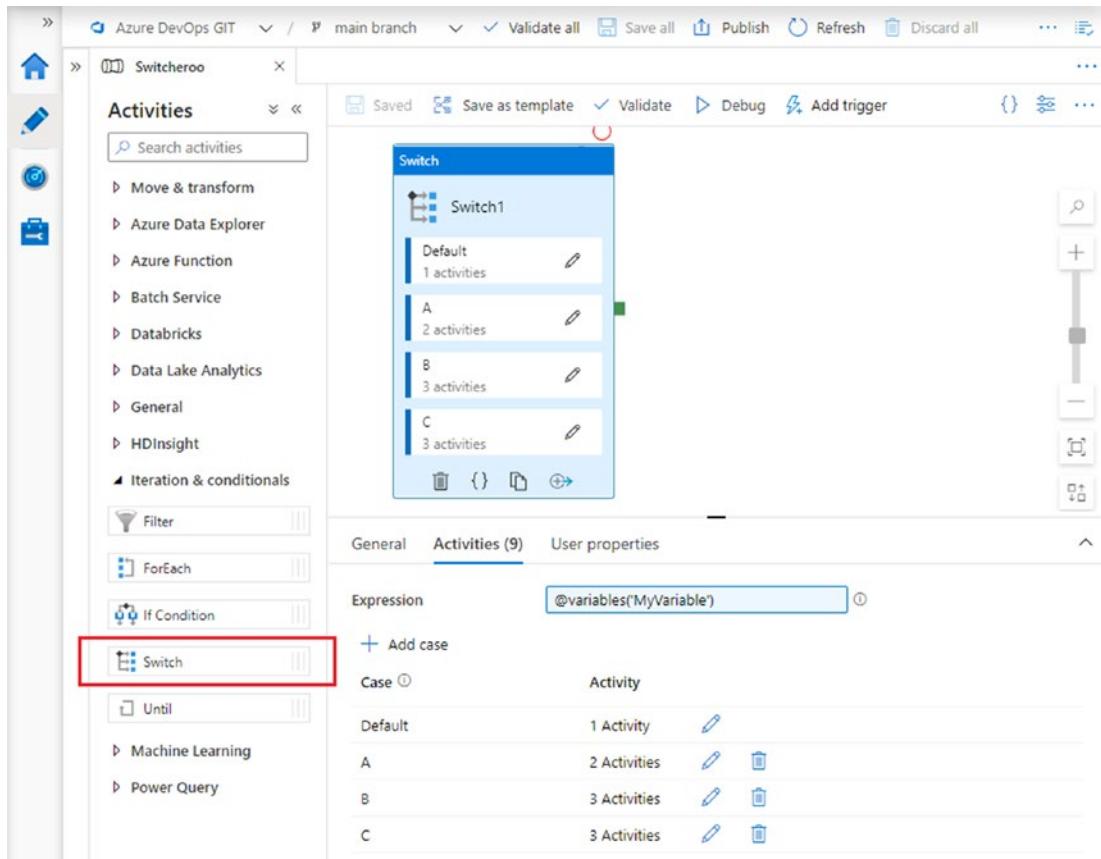


Figure 6-14. Switch activity configuration

Use the If Condition activity when your control flow requires only two cases – if you need more cases, use the Switch activity.

For SSIS developers SSIS contains no control flow task directly comparable to the If Condition or Switch activities, but provides similar functionality using expressions in precedence constraints. In SSIS, it is occasionally necessary to create a dummy predecessor task, to provide a precedence constraint to implement if/switch-like behavior – this is not required in Azure Data Factory because expressions and activity dependencies are decoupled.

Use Iteration Activities

The pipeline you developed in the previous section provides more sophisticated process control and error handling than those from earlier chapters, but – by design – loads only a single, specified file. To load all Naughty but Nice sales files, the pipeline must be executed once for each file. In this section, you will use one of Azure Data Factory's *iteration activities* to do this automatically, iterating over a list of files to be loaded.

Use the Get Metadata Activity

Before you can iterate over a list of files, you must first obtain one – you can do this in ADF using the *Get Metadata* activity.

1. Create a new pipeline in the “Chapter6” folder and name it “ImportSTFormatFiles.” Create a pipeline parameter called “FolderPath,” giving it the default value of the Naughty but Nice sales data folder (“azure-data-factory-by-example-main/SampleData/NaughtyButNice”).
2. Drag a Get Metadata activity – found in the *General* section of the activities toolbox – onto the authoring canvas. Name it “Get file list.” The Get Metadata activity returns metadata relating to an ADF dataset – on its *Dataset* configuration tab, choose the “ABS_CSV” dataset.

3. The files to be loaded are those in the “NaughtyButNice” blob storage folder – you will use the Get Metadata activity to list the folder’s files. Set the dataset’s “Container” property to “sampledata.” Populate the “Directory” property with an expression that returns the value of the pipeline’s “FolderPath” parameter.
4. The “ABS_CSV” dataset requires a file parameter. If you do not specify one, ADF will use the default value (“.”) which will fail to match the “NaughtyButNice” folder. To work around this, replace the period in the dataset’s “File” property field with a space character, as indicated in Figure 6-15.
5. Scroll down the *Dataset* configuration tab to the *Field list* section (also indicated in the figure). This is where metadata items to be returned by the activity are specified. Add three arguments: “Exists,” “Item type,” and “Item name.”
6. Run the pipeline. If you have configured it correctly, the activity’s output JSON will include three fields: `exists = true`, `itemType = “Folder”`, and `itemName = “NaughtyButNice”`. If `exists` is false, check your configured file path for errors.
7. Add another argument to the *Field list* on the activity’s *Dataset* configuration tab: “Child Items.” Run the pipeline again and verify that the activity’s output now includes a `childItems` array field containing the list of files in the “NaughtyButNice” folder.

Tip If the Get Metadata activity’s target file or folder does not exist, and the “Exists” argument is not specified, the activity will fail. Similarly, if the “Child Items” argument is specified for a path that identifies a file instead of a folder, the Get Metadata activity will fail. Delaying using the “Child Items” argument until step 7 allows you to ensure that your configured path refers correctly to a folder.

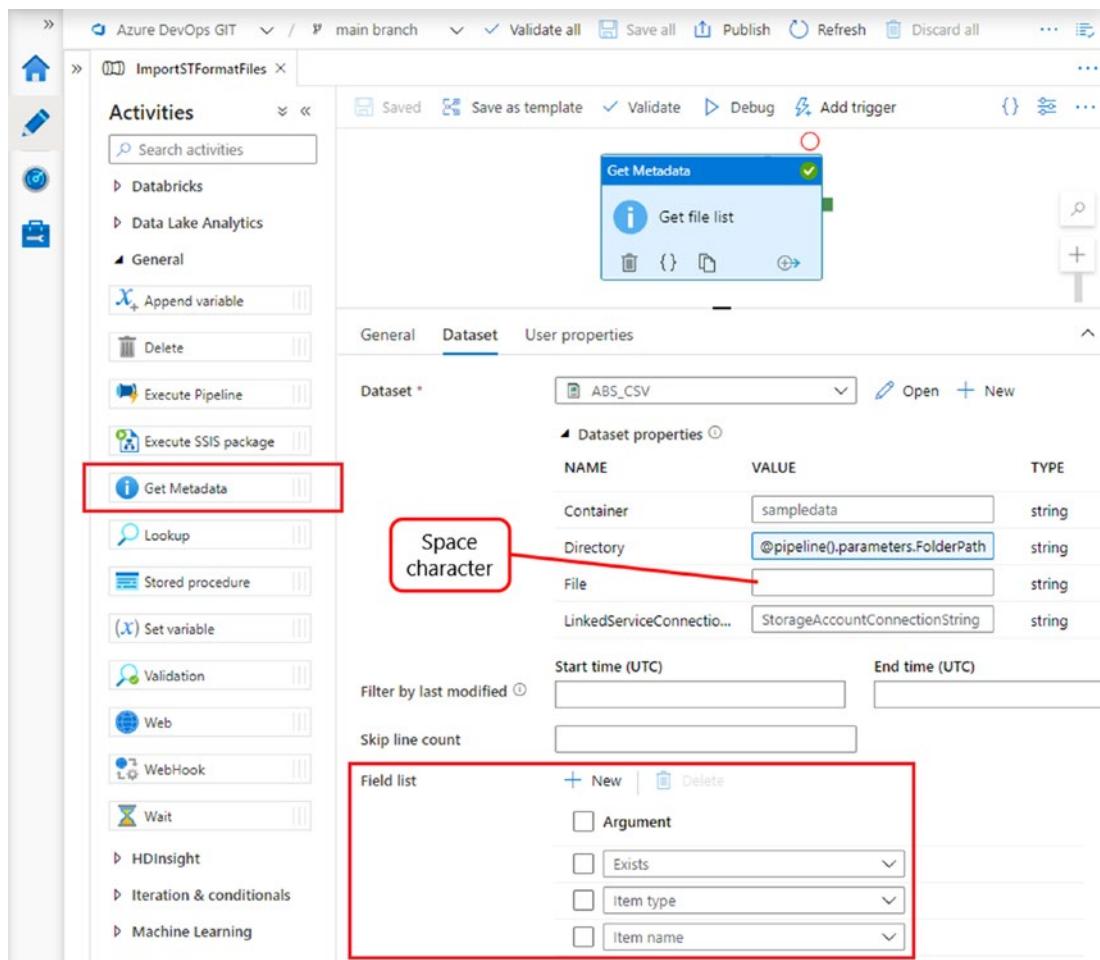


Figure 6-15. Configuring the Get Metadata activity

Use the ForEach Activity

The *ForEach activity* is one of Azure Data Factory's two iteration activities. It allows you to specify a set of activities to be performed repeatedly, once for every item in a given JSON array, for example, the array of child items returned by the Get Metadata activity.

1. Drag a ForEach activity onto the authoring canvas, making it dependent on the Get Metadata activity.
2. On the activity's *Settings* configuration tab, open the expression builder in the *Items* field.

3. Choose the output of your Get Metadata activity from the expression editor's *Activity outputs* list, then add ".childItems" to the expression in the expression pane. The final expression should be @activity('Get file list').output.childItems.
4. To edit the set of activities to be executed on each iteration, click the pencil icon on the *Activities* configuration tab. (Alternatively, use the pencil icon displayed in the activity graphic on the authoring canvas.)
5. Drag an Execute Pipeline activity onto the empty ForEach canvas. Open its *Settings* configuration tab and set its *Invoked pipeline* to "ImportSTFormatFile_Divert". Configure the invoked pipeline's "Directory" parameter to use the value of this pipeline's "FolderPath" parameter.
6. The invoked pipeline's "File" parameter value should be the name of the file for the current iteration of the ForEach activity. The current item in a ForEach activity's array is specified using the expression `item()`. In this example, `item()` returns an object element of the `childItems` array in the Get Metadata activity's output. The name of the file is located in the object's `name` field, so the full file expression is `@item().name`. The configured Execute Pipeline activity is shown in Figure 6-16.
7. Run the pipeline.

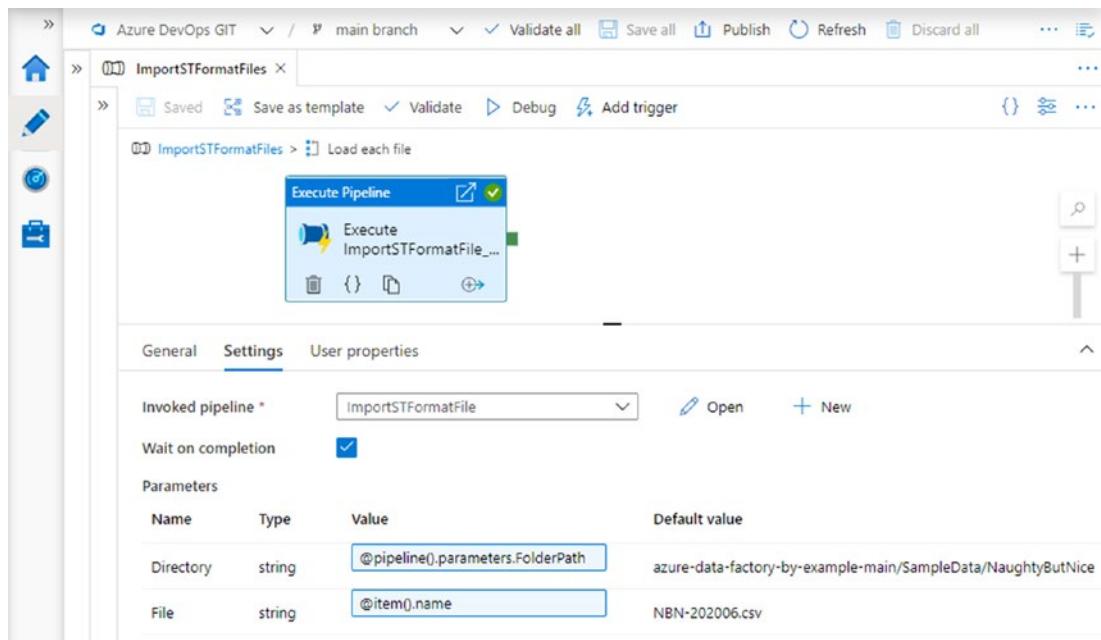


Figure 6-16. Execute Pipeline activity using the containing ForEach activity's item() expression

When you run the pipeline in Debug mode, the ADF UX warns you that “All the activities inside the foreach loop will be executed sequentially and each execute pipeline activity will wait on completion for debugging purposes.” This is in contrast to published pipelines, where the default behavior of the ForEach activity is to execute up to 20 iterations simultaneously, in parallel. One of the key advantages of Azure Data Factory over on-premises services is the ability to scale out in this way, briefly leveraging more resource to reduce end-to-end processing time.

When you configured the ForEach activity’s *Items* array (on its *Settings* configuration tab), you will have noticed two additional options: a *Sequential* checkbox and a *Batch count* field. These options allow you to constrain the maximum parallelism exhibited by the activity. Ticking the *Sequential* checkbox eliminates parallel execution, requiring iterations to run sequentially. When the checkbox is unticked, *Batch count* specifies the maximum number of iterations permitted to run in parallel (up to 50).

Tip Because ForEach activity iterations are often run sequentially in debug mode, it is not sufficient to test pipelines in the ADF UX alone. Issues with parallelization may not become apparent until pipelines are tested in a published ADF environment. Publishing pipelines is discussed in Chapter 10.

Use your SQL client to inspect the latest records in the log table [dbo].[PipelineExecution]. The table shows six new pipeline runs, one for each month's Naughty but Nice sales data. The [Comments] column indicates which file was loaded by each pipeline run – the run for the file “NBN-202006.csv” again indicates that more rows were read than were copied, while all records read from each of the other five files were copied successfully.

Ensure Parallelizability

You first encountered issues with parallelizability at the end of Chapter 5, when it became necessary to chain two Execute Pipeline activities together to prevent them from running simultaneously. The pipelines you have built so far are not parallel-safe, because in each case the Copy data activity’s pre-copy script truncates the [dbo].[Sales_LOAD] table. Multiple simultaneous pipeline runs using this pattern incur the risk that any one of them could truncate the table while others are mid-copy.

The ForEach activity cannot benefit from ADF’s scale-out ability unless iterations can be isolated from one another. In this section, you will modify your per-file pipeline to move loaded data into a separate table, allowing the pipeline to take responsibility for removing only its own records from [dbo].[Sales_LOAD]. This will replace the previous approach of housekeeping the table by regularly truncating it.

1. Listings 6-3 and 6-4 provide code to create a table called [dbo].[Sales] and a stored procedure to update the table from [dbo].[Sales_LOAD]. Run these two scripts in your SQL client to create the two database objects.
2. Edit your “ImportSTFormatFile_Divert” pipeline, adding a Stored procedure activity between the “Import sales data” Copy data activity and the “If rows skipped” If Condition activity.

3. Configure the Stored procedure activity to call procedure [dbo].[PersistLoadedSales] from Listing 6-4, using the pipeline's "RunSeqNo" variable to provide the stored procedure's parameter value.
4. Edit the Copy data activity, removing the *Pre-copy script* from the activity's *Sink* configuration tab.

Listing 6-3. Table [dbo].[Sales]

```
CREATE TABLE dbo.Sales (
    RowId INT NOT NULL IDENTITY(1,1)
    , Retailer NVARCHAR(255) NOT NULL
    , SalesMonth DATE NOT NULL
    , Product NVARCHAR(255) NOT NULL
    , SalesValueUSD DECIMAL(19,2) NOT NULL
    , UnitsSold INT NOT NULL
    , RunSeqNo INT NOT NULL
    , CONSTRAINT PK_dbo_Sales PRIMARY KEY (RowId)
);
```

Listing 6-4. Procedure [dbo].[PersistLoadedSales]

```
CREATE PROCEDURE dbo.PersistLoadedSales (
    @runSeqNo INT
) AS
DELETE tgt
FROM dbo.Sales tgt
INNER JOIN dbo.Sales_LOAD src
    ON src.Retailer = tgt.Retailer
    AND src.SalesMonth = tgt.SalesMonth
    AND src.Product = tgt.Product
WHERE src.RunSeqNo = @runSeqNo;
DELETE
FROM [dbo].[Sales_LOAD]
OUTPUT
```

```
    deleted.[Retailer]
    , deleted.[SalesMonth]
    , deleted.[Product]
    , deleted.[SalesValueUSD]
    , deleted.[UnitsSold]
    , deleted.[RunSeqNo]
INTO dbo.Sales (
    [Retailer]
    , [SalesMonth]
    , [Product]
    , [SalesValueUSD]
    , [UnitsSold]
    , [RunSeqNo]
)
WHERE RunSeqNo = @runSeqNo;
```

The [dbo].[Sales_LOAD] table will now be empty most of the time, except for the intervals between a pipeline loading data from a file and copying it into [dbo].[Sales]. To complete this setup, use your SQL client to truncate the [dbo].[Sales_LOAD] table, consistent with its new resting state.

Finally, rerun the pipeline “ImportSTFormatFiles” to load all six Naughty but Nice sales files. When execution is complete, verify that sales data for all six months between April and September 2020 is present in the table [dbo].[Sales] and that the table [dbo].[Sales_LOAD] is empty.

Although the ADF UX forces the pipeline’s ForEach activity to load the six files sequentially during debugging, the pipeline can now safely load files in parallel when published. This is because each pipeline run removes from [dbo].[Sales_LOAD] only the data it loaded, so will not interfere with other activities (in this or other pipelines) running at the same time.

Tip Remember that pipeline variables are scoped at the level of the pipeline, so ADF's Set variable and Append variable activities are never parallel-safe. ForEach iterations that modify variables will produce unpredictable results and must be avoided. A common workaround is to encapsulate an iteration's activities in a nested Execute Pipeline activity, just as you have done here.

Understand the Until Activity

Like the ForEach activity, ADF's *Until activity* allows you to specify a set of activities to be performed repeatedly. Instead of iterating over a fixed array of elements, the Until activity repeats indefinitely until its *terminating condition* is met (or the activity times out). Figure 6-17 shows an example of an Until activity's terminating condition expression – the activity will repeat until the value of a variable called "FilesRemaining" falls to zero, or the default timeout of seven days is reached.

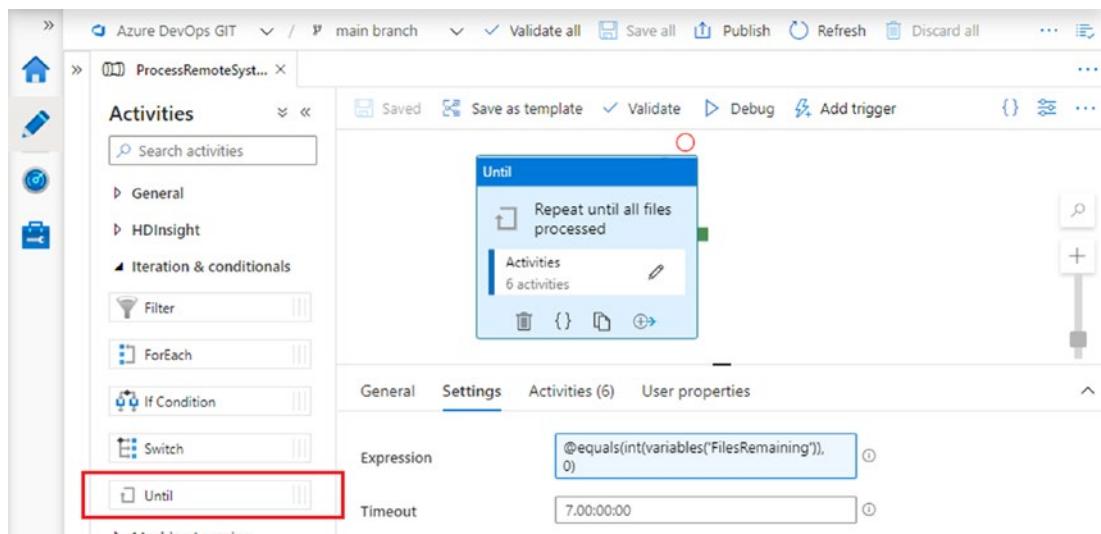


Figure 6-17. Expression providing an Until activity's terminating condition

Tip ADF does not permit nesting of iteration activities, nor does it allow you to use them inside conditional activities. As before, a workaround is to use different pipelines to implement the inner and outer loops of a nested iteration.

Typical use cases for the Until activity are situations involving external dependencies – for example, to delay extracting data from a data source until the source system has finished some internal housekeeping. One or more activities inside the Until activity usually have the task of reevaluating the situation in each iteration – in the example of Figure 6-17, this might consist of recalculating the number of files remaining, then using the Set variable activity to assign the calculated value to the “FilesRemaining” variable.

Because the Until activity’s iterations do not correspond to array elements, the item() syntax is not supported in this context. For the same reason, parallel iterations are not possible – execution of Until activity iterations is always sequential. The Until activity’s terminating condition expression is reevaluated at the end of every iteration – this means that the activities inside an Until activity are always executed at least once.

For SSIS developers ADF’s ForEach activity is directly comparable to SSIS’s Foreach Loop Container, although SSIS additionally permits nested loops within a single package. The indefinite iteration pattern of ADF’s Until activity is not directly supported in SSIS. A For Loop Container specifying no AssignExpression achieves a similar effect, but has the small difference that EvalExpression is evaluated at the start of every iteration (so an execution with zero iterations is possible).

Chapter Review

In this chapter, you used three varieties of tool to control pipeline execution flow:

- **Activity dependency conditions:** To determine what happens after an activity’s execution is complete
- **Conditional activities:** To execute separate subsets of a pipeline’s activities under different conditions

- **Iteration activities:** To execute subsets of a pipeline's activities repeatedly

Used collectively – including by nesting conditional and iteration activities in different pipelines – this forms a comprehensive arsenal of tools for controlling flow.

Key Concepts

- **Dependency condition:** Characterizes an activity dependency. An activity dependent on another is only executed if the associated dependency condition is met – that is, depending on whether the prior activity succeeds, fails, or is skipped.
- **Multiple dependencies:** An activity dependent on multiple activities is only executed if each prior activity satisfies a dependency condition. If an activity specifies multiple dependency conditions on the same prior activity, only one needs to be met.
- **Leaf activity:** A leaf activity is a pipeline activity with no successors.
- **Conditional activities:** ADF has two conditional activities – the If Condition activity and the Switch activity.
- **If Condition activity:** Specifies an expression that evaluates to true or false and two corresponding contained sets of activities. When the activity runs, its expression is evaluated, and the corresponding activity set is executed.
- **Switch activity:** Specifies an expression that evaluates to a string value and up to 25 corresponding contained sets of activities. When the activity runs, the expression is evaluated, and the corresponding activity set, if any, is executed. If no matching case is found, the default activity set is executed instead.
- **Iteration activities:** ADF has two iteration activities – the ForEach activity and the Until activity.
- **ForEach activity:** Specifies a JSON array and a set of activities to be executed once for each element of the array. The current element of the array is addressed in each iteration using the `item()` expression.

- **Parallelism:** By default, ForEach activity executions take place in parallel, requiring care to ensure that activities from simultaneous iterations do not interfere with one another. Variable modification must be avoided, but the Execute Pipeline activity provides an easy way to isolate iterations. The ADF UX executes ForEach iterations sequentially in Debug mode, which can make parallelism faults hard to detect at development time.
- **Until activity:** Specifies a terminating condition and a set of activities to be executed repeatedly until the terminating condition is met. Activities within an Until activity are executed at least once and never in parallel.
- **Nesting:** Iteration activities may not be nested in other iteration activities. Conditional activities may not be nested in other conditional activities, although nesting in iteration activities is permitted. A common workaround is to implement inner and outer activities in separate pipelines, calling the inner activity from the outer via the Execute Pipeline activity.
- **Breakpoints:** The ADF UX does not support breakpoints inside iteration or conditional activities.
- **Get Metadata activity:** Returns metadata that describes attributes of an ADF dataset. Not all metadata attributes are supported by all datasets – nonexistent dataset targets will cause errors unless the “Exists” argument is specified; specifying the “Child Items” argument on a nonfolder target will cause the activity to fail.
- **Fault tolerance:** The Copy data activity supports enhanced error handling through its Fault tolerance settings, enabling individual error rows to be diverted into an external log file without completely abandoning a data load.
- **Raising errors:** At the time of writing, ADF has no “raise error” activity. Approaches available to manufacture errors include defining illegal type casts in Set variable activities or exploiting error raising functionality in external services, for example, by using SQL Server’s RAISERROR or THROW statements.

For SSIS Developers

Azure Data Factory activity dependencies and conditions are similar to SSIS precedence constraints, but activity dependencies do not support expressions. Conditional execution is controlled separately using the If Condition or Switch activities. An activity will only be executed if a dependency condition is satisfied by every activity on which it depends – unlike SSIS, there is no alternative “Logical OR” behavior configurable for multiple dependencies.

Unlike in SSIS, error propagation cannot simply be switched on or off in ADF. Pipeline outcome is determined by the outcome of a pipeline’s leaf activities – error propagation is controlled by structuring activity dependencies to produce the required behavior.

SSIS’s Foreach Loop Container is similar in operation to the ADF ForEach activity, albeit without the automatic scale-out capabilities of ADF. The SSIS For Loop Container could be simulated by the ADF ForEach activity given an appropriate JSON array of numbers; the Until activity provides a more direct implementation of indefinite iteration than is available in SSIS.

CHAPTER 7

Data Flows

The Copy data activity is a powerful tool for moving data between data storage systems, but it has limited support for data transformation. Columns can be added to the activity's source configuration, or removed by excluding them from the source to sink mapping, but the activity does not support manipulation of individual rows or allow data sources to be combined or separated.

The Azure platform supports a number of tools capable of large-scale data transformation, one of which is *Azure Databricks*. Databricks is a cloud-based analytics platform based on Apache Spark, an open source framework that automatically distributes data processing workloads across a cluster of servers (referred to as *nodes*) to enable highly parallelized execution. Transformation processes are implemented using code written in one of several supported languages, such as Scala, R, or Python.

You can execute Azure Databricks processes from ADF using one of the Databricks activities found in the activities toolbox, but this requires you to implement Databricks data transformations in the code language of your choice and to create and manage a separate Databricks workspace in Azure. Azure Data Factory *data flows* provide an alternative, low-code route to the power of Azure Databricks. Data flows are implemented using a visual editor and converted automatically by ADF into Scala code for execution in a managed Databricks cluster. This chapter introduces you to authoring ADF data flows.

Build a Data Flow

Azure Data Factory data flows are independent, reusable ADF resources, implemented in the ADF UX using a visual *data flow canvas*. The execution of a data flow occurs in an ADF pipeline using the *Data flow activity* – in this section, you will create a data flow, create a pipeline to execute it, and run that pipeline.

1. Open the ADF UX authoring workspace. In the *Factory Resources* explorer, you will find the *Data flows* resource type (below the familiar *Pipelines* and *Datasets* resource types). Create a new “Chapter7” folder using the Data flows *Actions* menu.
2. On the “Chapter7” folder’s *Actions* menu, click *New data flow*.
3. The data flow canvas opens. If this is your factory’s first data flow, the callout message *Start by adding source to data flow* is displayed – dismiss it using its close button. In the *Properties* blade, change the name of the data flow to “TransformJollyGoodSales,” then close the blade in the usual way.

This data flow will be used to load ABC sales data for a UK-based confectionery vendor named “Jolly Good Ltd.”

Enable Data Flow Debugging

Although triggered from a pipeline activity, the execution of ADF Azure Data Factory data flows takes place on a Databricks cluster. A cluster must be provisioned before a data flow can be executed, whether in a published environment or when debugging in the ADF UX. Provisioning a cluster takes several minutes, so your first task when developing a data flow is to spin up a cluster for convenient debugging.

1. Set the *Data flow debug* toggle to “On.” When in the “Off” position, as in Figure 7-1, the toggle’s background color is gray.

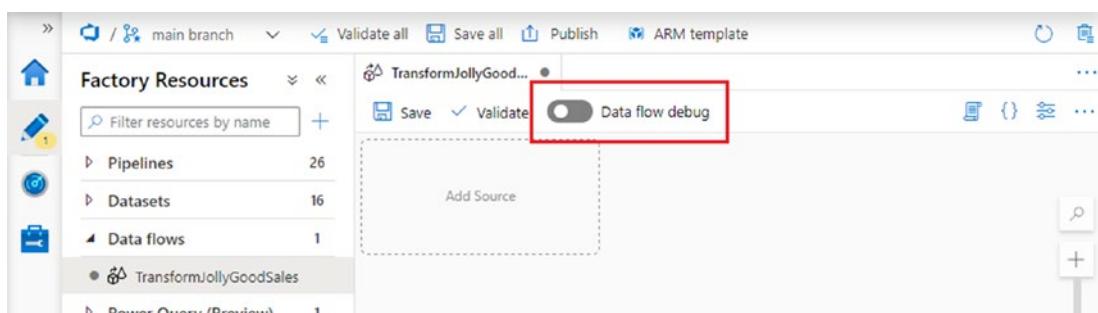


Figure 7-1. Data flow debug disabled

2. The *Turn on data flow debug* dialog is displayed, prompting you to select an *Integration runtime* and *Time to live*. Leave the default values selected and click *OK*. Integration runtimes are discussed in more detail in Chapter 8.

The *Data flow debug* toggle's background color changes to blue, and to its right a spinner indicates that the cluster is starting. After a few minutes, the spinner is replaced by a tick mark in a green circle, as shown in Figure 7-2, indicating that the cluster is ready for use. When the cluster is ready, you can use it to run data flows in debug mode from the ADF UX.

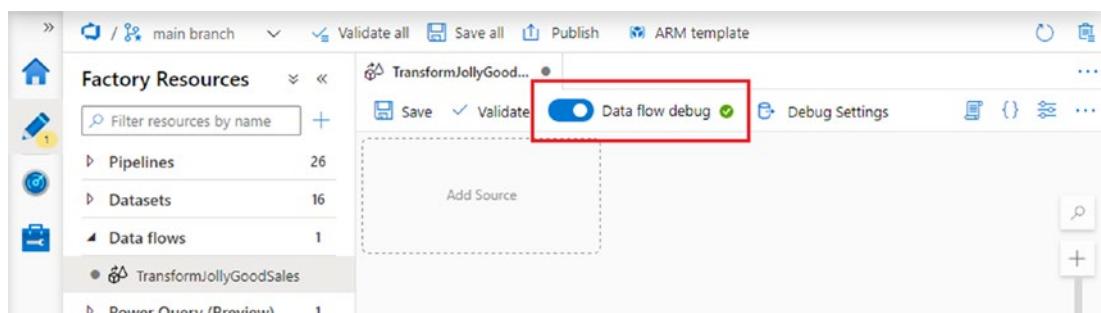


Figure 7-2. Data flow debug enabled

The debug cluster's default time to live (TTL) is one hour – after that time, if the cluster is not being used, it is automatically shut down. When cluster timeout is approaching, the ADF UX issues a warning, and the green tick icon is replaced by a filled orange circle. If you wish to continue working with the cluster – and don't want to have to wait for a new one to be provisioned – you must run the data flow or take another action that uses the cluster.

When you have finished developing and debugging, disable the debug cluster – although it will shut down automatically when its TTL is reached, you can save yourself the cost of running the cluster for the remainder of that time.

Add a Data Flow Transformation

You are now ready to start data flow development. A data flow is made up of a sequence of connected *transformations*. Conceptually, each transformation in the sequence

- Receives a stream of data rows from the previous transformation
- Modifies rows in the stream as they pass
- Emits modified rows to the next transformation

For SSIS developers In Chapter 3, I described the Copy data activity as providing functionality like a basic SSIS Data Flow Task. ADF Data Flows are much more similar – the behavior of the data flow canvas and transformations is closely comparable to that of SSIS’s data flow surface and components.

A special *Source transformation* reads data from an external source and emits it as a stream of rows to be transformed by the data flow. Every data flow contains at least one Source transformation.

1. The empty data flow canvas for your new data flow contains an *Add Source* tile with a dashed outline. Click the tile to add a source to your data flow. You must add a source before you can do anything else.
2. If this is your first data flow, a three-step callout is displayed, providing tips for how to interact with the data flow canvas. Step through these, then on step 3 click *Finish*.
3. Like ADF activities on the pipeline authoring canvas, data flow transformations are configured using a tabbed configuration pane below the data flow canvas. Every transformation has a unique name within the data flow called its *output stream name* – set this value appropriately on the *Source settings* tab (shown in Figure 7-3).

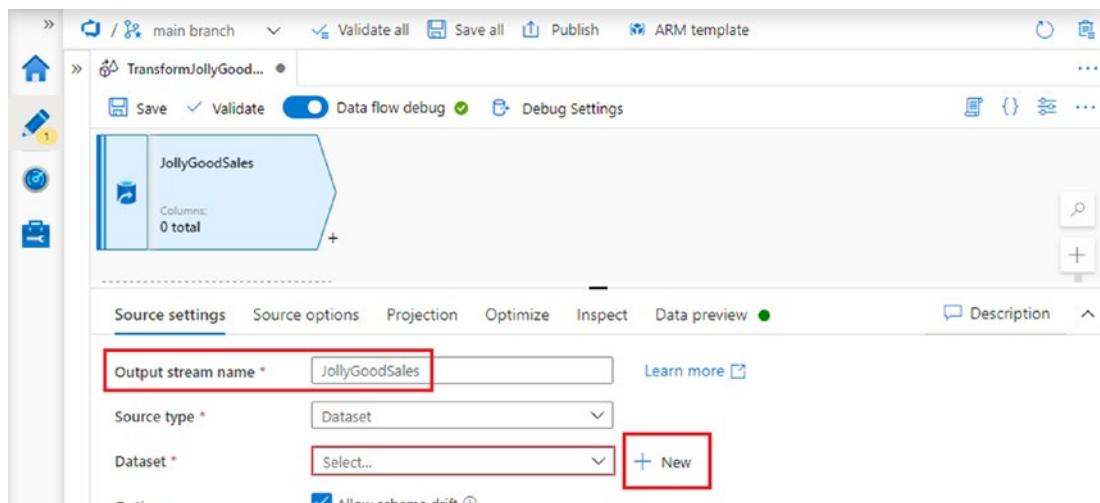


Figure 7-3. Data flow source transformation's Source settings tab

4. Ensure that the transformation's *Source type* is set to "Dataset," then to the right of the *Dataset* dropdown list, click the *+ New* button.
5. The *New dataset* blade familiar from earlier chapters opens. Select the *Azure Blob Storage* data store, click *Continue*, then choose the *Excel* file format (Jolly Good sales data is supplied in Excel spreadsheet files). Click *Continue*.
6. Name the dataset, then choose your original blob storage linked service (a linked service which defines no parameters). Browse to locate and select the file "Sales Apr-Sep 2020.xlsx" from the "sampledata" container's "JollyGood" folder. Select *Sheet name* "SALES" and ensure that *First row as header* is ticked. The completed blade is shown in Figure 7-4 – click *OK* to create the dataset.

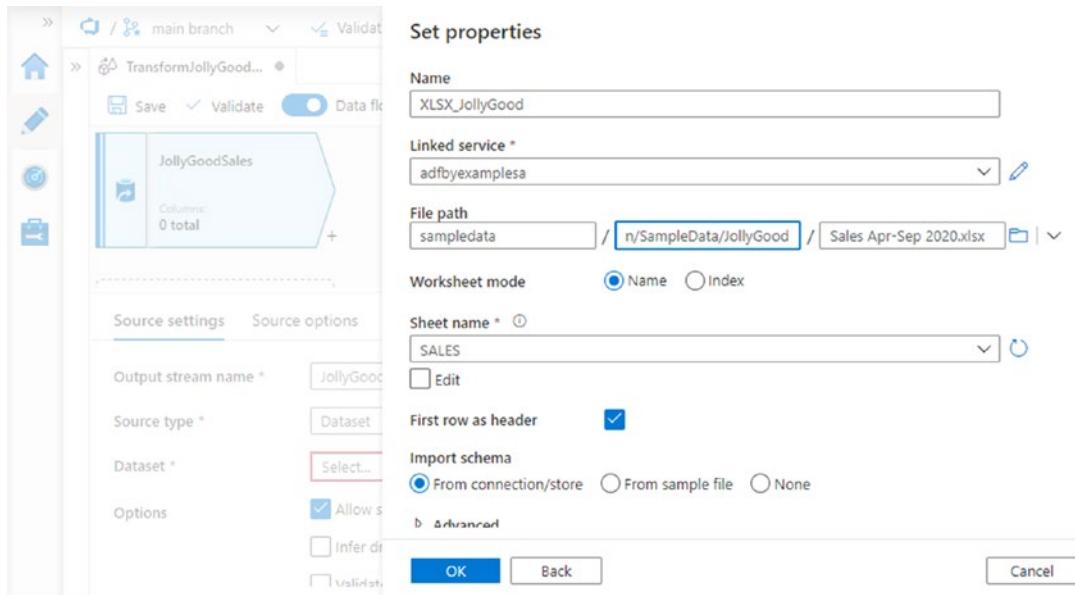


Figure 7-4. Properties for Jolly Good Excel dataset

7. Select the Source transformation's *Inspect* tab. This tab – which also appears for every other type of transformation – provides details of a transformation's input and/or output schema. (A Source transformation has no input schema). The schema shown here is the one imported into the new dataset.
8. Select the Source transformation's *Data preview* tab. This tab, also shared by every type of transformation, enables you to preview data emitted by a transformation. Click *Refresh* to load the preview.

Tip Data preview is only available with debug mode enabled – that is, when a debug cluster is running. If the ADF UX warns you that your debug cluster session is about to time out, previewing a transformation's output is a convenient way to extend the cluster's TTL.

- From the preview (shown in Figure 7-5), notice features of Jolly Good sales data that would be difficult to handle using a Copy data activity. The value of product sales is reported as a combination of units sold and unit price rather than as a single sales total. Unit price is provided in a mixture of currencies. The file includes a total row for each month which must be excluded. You can transform all of these using a data flow.

The screenshot shows the Azure Data Flow (ADF) UX tool interface. At the top, there's a toolbar with various icons like Save, Validate, Publish, and ARM template. Below the toolbar, the main workspace shows a dataset named "JollyGoodSales" with a blue arrow icon. A tooltip indicates it has 5 columns. The "Data preview" tab is selected, showing a table with the following data:

Period	Product	Unit Price	Currency	Units Sold
2020-04-01	Month Total	NULL	NULL	245920
2020-04-01	Flair 500g	11.29	EUR	2552
2020-04-01	Minotaur 95g	1.19	GBP	56
2020-04-01	LARGE ITEM	2.20	GBP	2103

Below the table, there are buttons for Refresh, Typecast, Modify, Map drifted, Statistics, and Remove. The bottom of the screen shows a status bar with metrics: Number of rows (100), INSERT (100), UPDATE (0), DELETE (0), UPSERT (0), LOOKUP (0), and TOTAL (740).

Figure 7-5. Preview Jolly Good sales data

- Select the Source transformation's *Projection* tab. This tab is specific to the Source transformation and displays the five columns present in the Excel file's "SALES" sheet. The column types are all reported as "string" – use each column's *Type* dropdown to refine types. As the data values previewed in Figure 7-5 suggest, the column "Period" is of type "date," "Unit Price" is of type "double," and "Units Sold" is an "integer" column.
- Click *Save all* in the ADF UX toolbar to save your work so far (including the new Excel dataset).

Like other factory resources, data flows are saved to your Git repository as JSON files. The sequence of data flow transformations is stored as *Data Flow Script*, embedded in the data flow JSON file's `properties.typeProperties.script` attribute. You can use the *Code* button (braces icon) to the top right of the data flow canvas to view data flow JSON and can inspect the formatted Data Flow Script directly using the *Script* button (next to the *Code* button).

Use the Filter Transformation

The Jolly Good sales data format includes monthly total rows interleaved with product-specific sales data – the first of these is visible in the data preview shown in Figure 7-5. Exclude these rows using the data flow *Filter transformation*.

Tip You may have noticed an activity called *Filter* in the pipeline activities toolbox. The pipeline activity serves a different purpose, allowing you to select a subset of elements from an input array.

1. To add a transformation to the data flow, click the small “+” button at the bottom right of the Source transformation on the data flow canvas. This button appears in the same position for every transformation except the Sink.
2. A popup menu of available transformations is displayed, as shown in Figure 7-6. Find and select the Filter transformation (toward the bottom of the list). When the transformation has been added to the data flow canvas, set its *Output stream name* on the *Filter settings* configuration tab.

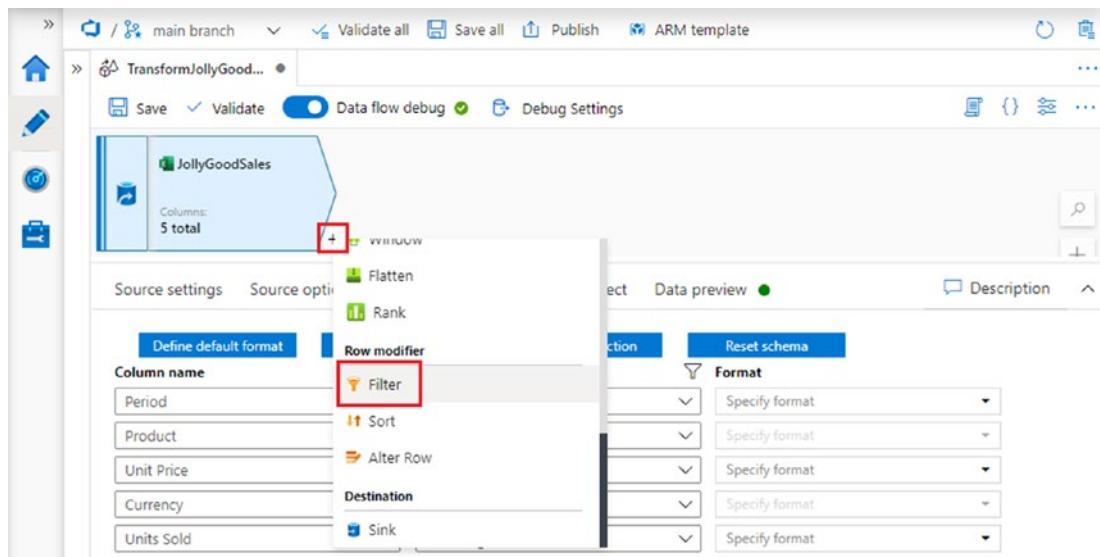


Figure 7-6. Connect Filter transformation to Source transformation

3. Also found on the *Filter settings* tab is a *Filter on* text area. This stores a *data flow expression* used to select rows for the transformation's output. Click the text area to open the data flow *Visual expression builder*.
4. Figure 7-7 shows the Visual expression builder for the Filter transformation, including an *Expression* pane above an operator toolbar and a menu of *Expression elements*. The list of *Expression values* corresponds to the selected element type. These features – along with the *Data preview* pane at the bottom of the builder window – are always available in the expression builder.

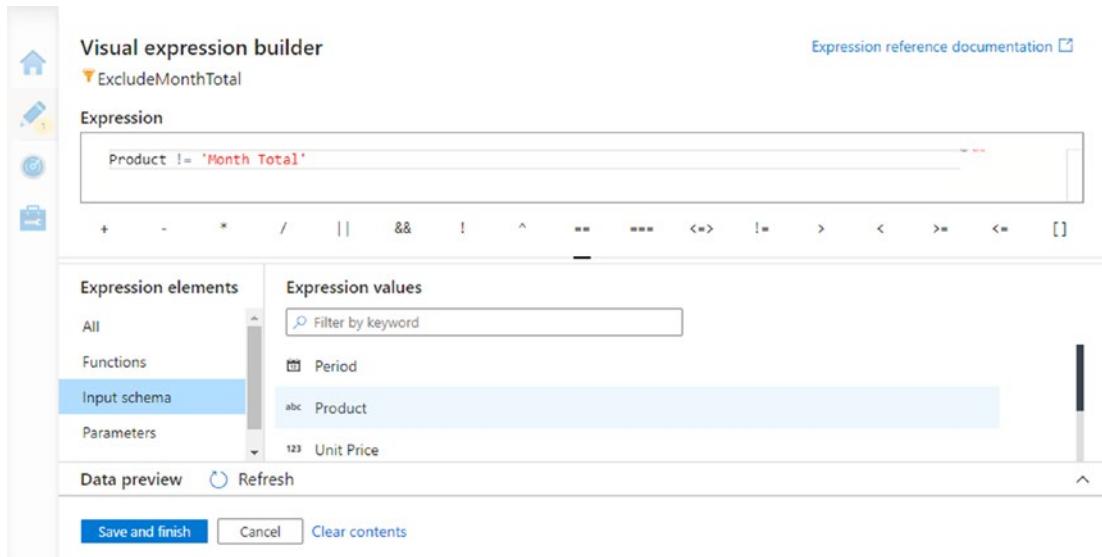


Figure 7-7. Visual expression builder for the Filter transformation

To build the expression shown in Figure 7-7, select the “Product” column from the list of expression values in the *Input schema*, use the operator toolbar to add the *Not equal* (!=) operator, then add ‘Month Total’ to the *Expression* pane by hand. Click *Save and finish*.

Note The data flow expression language is different to that used in ADF pipeline expressions and is much richer. All text comparisons are case-sensitive unless you use an explicitly case-insensitive option (like the `<=` operator or its equivalent function `equalsIgnoreCase`). Unlike pipeline expressions, the data flow expression language includes a variety of infix operators, shown in the operator toolbar in Figure 7-7.

5. Select the Filter transformation’s *Data preview* tab and click *Refresh* to load the preview. Notice that the initial Month Total row no longer appears.
6. Save your updated data flow.

Use the Lookup Transformation

The *Lookup transformation* enables you to use values from a data stream to look up matching rows in another data stream. (This is different from ADF's Lookup activity, which loads a dataset into a pipeline, allowing you to refer to elements within it.) In this section, you will use the Lookup transformation to obtain exchange rate information for currencies in the Jolly Good sales data file.

Add a Lookup Data Stream

A second data stream containing exchange rate information is required for the Lookup activity.

1. Add another data source to your data flow by clicking the *Add Source* tile displayed below your Jolly Good sales data Source transformation. Name it “ExchangeRates.”
2. In addition to ADF datasets, data flow Source transformations support a number of *inline datasets*. The set of formats supported by inline datasets is not the same as that supported by ADF dataset objects, although there is some overlap. Set the transformation’s *Source type* option to “Excel.”
3. A *Linked service* dropdown appears below *Source type* – choose the linked service for your blob storage account.
4. Select the *Source options* tab – it will now contain options specific to blob storage datasets. Use the *Browse* button to the right of the *File path* fields to select the file “ExchangeRates.xlsx” from the “sampledata” container’s “SampleData” folder. Select *Sheet name* “Sheet1” and ensure that *First row as header* is ticked.
5. Select the *Projection* tab. It is empty because no schema information has been imported yet – the inline dataset has no preexisting dataset object to refer to. Click *Import schema*. On the displayed *Import schema* blade, click *Import* to accept the default options and proceed. Four appropriately typed columns appear, as shown in Figure 7-8.

CHAPTER 7 DATA FLOWS

Column name	Type	Format
FromCurrency	abc string	Specify format
ToCurrency	abc string	Specify format
Date	date	Specify format
ExchangeRate	12 double	Specify format

Figure 7-8. Imported exchange rate schema on the Projection configuration tab

6. Select the *Data preview* tab and click *Refresh* to inspect the data in the file. The data includes conversion rates between three currencies – USD, GBP, and EUR – on the first of each month in the period April to September 2020.
7. The Lookup transformation behaves in a similar way to a SQL join. To ensure that the lookup joins to the correct row, filter the exchange rate source to exclude conversions to currencies other than USD. Figure 7-9 shows a Filter transformation configured to achieve this.

Import data from XLSX_JollyGood	+	Filtering rows using expressions on columns Product	+
+			
ExchangeRates	Add source dataset	RatesToUSD	Column: 4 total
+		+	

Filter settings

Output stream name * RatesToUSD

Incoming stream * ExchangeRates

Filter on * ToCurrency == 'USD'

Figure 7-9. Filter transformation excluding non-USD currency conversions

Add the Lookup Transformation

The exchange rate stream you have prepared is now ready for use as a rate lookup.

1. Use the “+” button on the Jolly Good data stream’s Filter transformation to connect a Lookup transformation. On the *Lookup settings* tab, set the *Output stream name* appropriately.
2. The *Primary stream* field is prepopulated with the name of the upstream transformation where you connected the Lookup. Set the value of *Lookup stream* to the output stream name of the exchange rate Filter transformation – take care to select the correct transformation in the exchange rate stream.
3. *Match multiple rows* and *Match on* control join behavior. Ticking the *Match multiple rows* checkbox creates an effect like a SQL join, where every matching pair of records is emitted in the output stream – without this option selected, *Match on* determines which matching pair is emitted. In this example, you require the matching pair in which the “ToCurrency” value is “USD” – this requirement is more sophisticated than *Match on* can support and is why you filtered the exchange rate stream in advance.
4. Specify two *Lookup conditions*. First, match the “Currency” field on the *Left* to the “FromCurrency” field on the *Right* of the lookup. By default, Lookup conditions contain only one criterion – add a second by clicking the “+” button to the right of the first condition. Match the “Period” field on the left to the “Date” field on the right. Both conditions should use the == equality operator for matching. Figure 7-10 shows the correctly configured lookup.

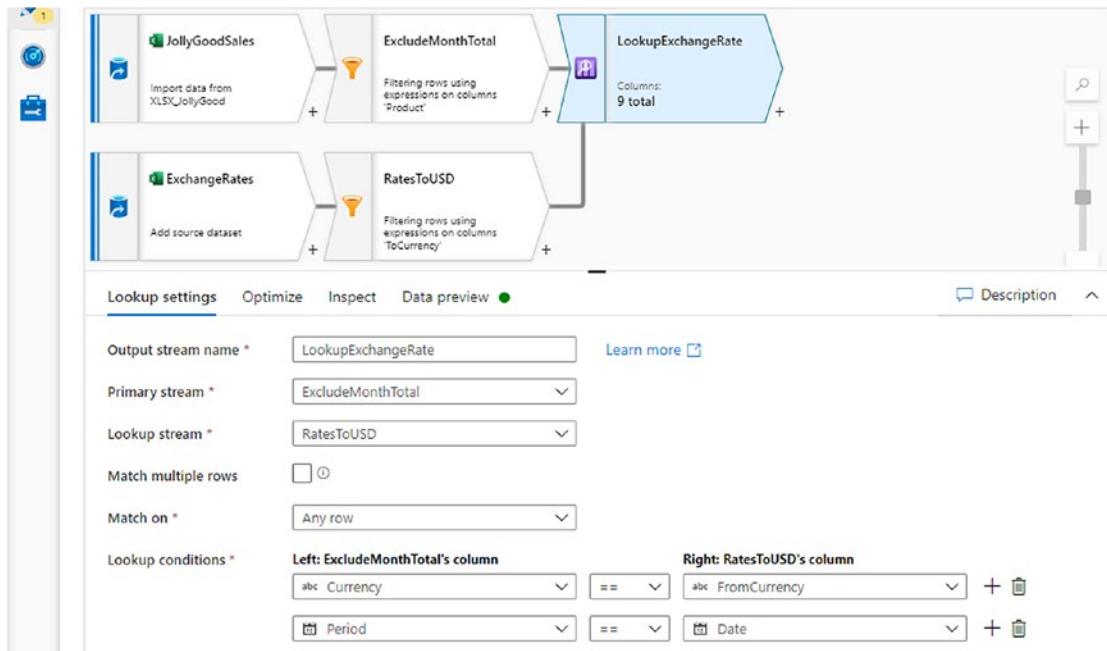


Figure 7-10. Configured exchange rate Lookup transformation

5. Select the *Data preview* tab and click *Refresh* to inspect the output of the transformation. Save your changes.

Use the Derived Column Transformation

The *Derived Column transformation* allows you to add new columns to the data flow. In this section, you will use the transformation to add columns necessary for the [dbo].[Sales_LOAD] table: total sales value (in USD), retailer, and the run sequence number used for pipeline execution logging.

1. Connect a Derived Column transformation to the Lookup transformation on the data flow canvas.
2. You can add columns on the *Derived column's settings* tab either directly or using the expression builder. For now, click *Open expression builder*.
3. The Visual expression builder for the Derived Column transformation contains transformation-specific features

(in addition to those you saw for the Filter transformation) including a *Column name* field and a *Derived Columns* sidebar enumerating columns derived in this transformation. Set the column name to “SalesValueUSD.”

4. The total USD sales value is the product of the unit price, the number of units sold, and the exchange rate. Use *Input schema* expression elements and the * operator to build this expression. The completed expression appears in Figure 7-11.
5. In the *Derived Columns* sidebar, click the + *Create new* menu button and select *Column* to add another column. Name the new column “Retailer” and give it the literal string value ‘Jolly Good Ltd’.
6. Add a third column in the same way, this time called “RunSeqNo” – this column will contain the pipeline run sequence number. Its value will be obtained in the same way as before, using the ADF pipeline’s Lookup activity, and will be passed to the data flow as a parameter.
7. You can create the “RunSeqNo” parameter right here in the expression builder by selecting *Parameters* from the *Expression elements* list. When you click + *Create new* under *Expression values*, the *Create new parameter* blade slides over the expression builder. In the top left, name the parameter “RunSeqNo” and set its type to “integer.” In the *Default value* pane, provide a value of -1. Click *Create* to create the parameter.
8. The expression builder resumes with the new parameter displayed in the *Expression values* list. Click the parameter name to select it as the derived column expression. Click *Save and finish*.

Figure 7-11 shows the configured Derived Column transformation, including the column expressions constructed in the expression builder. If you wish to edit any one expression, you can click the expression field to reveal an *Open expression builder* link (as shown in the figure for the “Retailer” column), or you can simply edit the expression in situ.

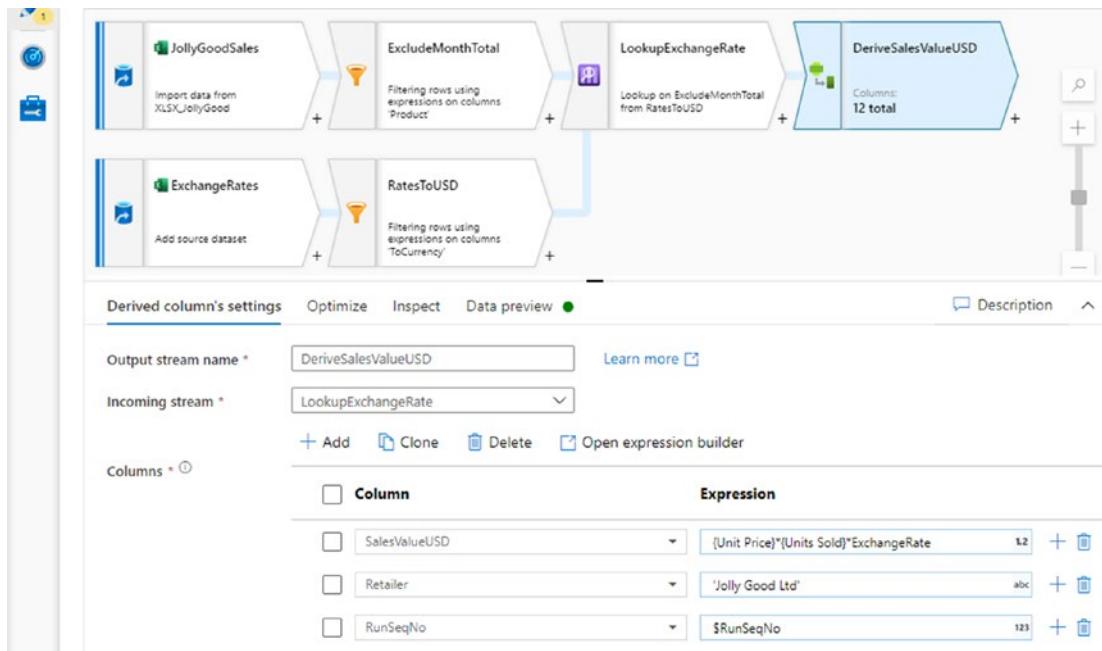


Figure 7-11. Configured Derived Column transformation

The “RunSeqNo” parameter you created using the expression builder is an input parameter for the data flow itself. Click some blank space on the data flow canvas to view configuration options at the data flow level, then select the flow’s *Parameters* configuration tab to view the definition of the parameter you created. If it is not immediately visible, close the data flow and reopen it to refresh the user interface.

Use the Select Transformation

Data flows’ *Select transformation* enables columns in the data stream to be renamed or removed. You will use it here to align the output schema with that of the sink table.

1. Connect a Select transformation to the Derived Column transformation configured in the previous section.
2. The *Select settings* tab contains a list of column mappings between the transformation’s input and output schemas. To the right of each mapping is a trash can *Remove mapping* button. Use it to remove the mappings for “Unit Price,” “Currency,” “FromCurrency,” “ToCurrency,” “Date,” and “ExchangeRate.” This removes those columns from the transformation’s output stream.

3. Use the *Name as* field (on the right of each mapping) to rename “Period” to “SalesMonth” and to remove the space character from the middle of “Units Sold.”
4. Use the *Data preview* tab to verify the effect of the transformation. The value in the “RunSeqNo” column will be -1, the default value for your “RunSeqNo” parameter. This will be replaced by a real run sequence number at runtime.

Use the Sink Transformation

A *Sink transformation* is used to persist data flow outputs in external data storage. A valid data flow requires at least one Sink transformation – you can save incomplete flows as you work, but you will be unable to run them. In this section, you will add a Sink transformation to write transformed Jolly Good sales data into your [dbo].[Sales_LOAD] database table.

1. Connect a Sink transformation to the Select transformation you added earlier.
2. On the *Sink* tab, name the transformation, and ensure that *Sink type* is set to “Dataset.” Choose the dataset “ASQL_dboSalesLoad” from the *Dataset* dropdown.
3. On the *Mapping* tab, disable auto-mapping using the *Auto mapping* toggle. This has the effect of displaying the existing set of automatically created mappings.
4. You may find that the [RunSeqNo] and [SourceFileName] output columns are missing – this is because they were absent from the [dbo].[Sales_LOAD] table when the corresponding ADF dataset was created. To remedy this, reimport the dataset’s schema by opening the dataset directly – it should be in your “Chapter3” datasets folder – then using the *Import schema* button on its *Schema* configuration tab. Return to the Sink transformation’s *Mapping* tab on the data flow canvas and click *Reset* to synchronize the transformation with the updated dataset.

5. Verify that all six input columns are now mapped correctly. The output columns [RowId], [ManufacturerProductCode], and [SourceFileName] have no corresponding inputs and can be left unmapped.
6. Inspect the transformation output using the *Data preview* tab, then save your changes. Data preview in the Sink transformation indicates the data that would be written to the sink at runtime, but no data is actually written.

Execute the Data Flow

With the addition of the final Sink transformation, the data flow is ready for execution. The complete flow is shown in Figure 7-12.

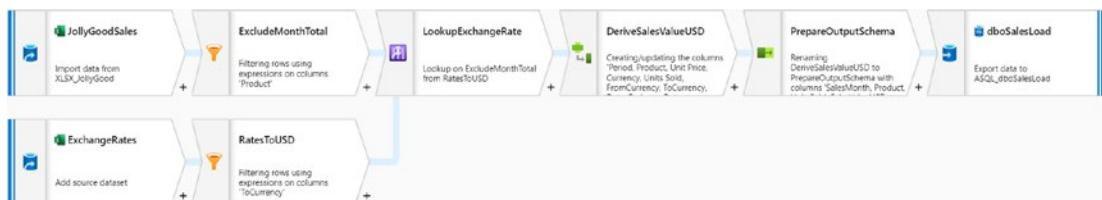


Figure 7-12. Complete data flow for loading Jolly Good Ltd sales data

Create a Pipeline to Execute the Data Flow

The data flow will be executed using an ADF pipeline similar to the pipelines you created earlier. In this case, instead of using the Copy data activity, the Data flow activity will be used to move data by executing the data flow.

1. In the ADF UX *Factory Resources* pane, create a “Chapter7” pipelines folder, then create a new pipeline inside it. Name the new pipeline “LoadJollyGoodSales.” You may also wish to move your Excel dataset into a folder for this chapter.
2. Add a Lookup activity to the pipeline. On its *Settings* configuration tab, select *Source dataset* “ASQL_dboSalesLoad”. Set *Use query* to “Stored procedure,” then select the “[dbo].[LogPipelineStart]” procedure from the *Name* dropdown. Click *Import parameter*,

then provide appropriate expressions for the three stored procedure parameters.

3. Drag a Data flow activity from the activities toolbox's *Move & transform* group onto the authoring canvas. Connect the new activity as a dependent of the Lookup activity.
4. On the activity's Settings tab, select the "TransformJollyGoodSales" data flow from the *Data flow* dropdown. On the *Parameters* tab, you will find the "RunSeqNo" parameter you created for the data flow, with its default value of -1.
5. Data flow parameters can be specified using either the pipeline expression language or the data flow expression language. Click the *VALUE* field to edit the parameter value, then select *Pipeline expression* from the popup that appears.
6. In the pipeline expression builder, enter an expression to return the `firstRow.RunSeqNo` property from the Lookup activity's output. Be careful to wrap the expression in the `int` conversion function, to avoid type conflicts when the activity attempts to start the data flow.
7. Execute the pipeline in the usual way by clicking *Debug*. Pipelines containing Data flow activities can be executed either using a data flow debug session – the default behavior when you click *Debug* – or using a *just-in-time (JIT)* Databricks cluster, provisioned automatically when pipeline execution starts. A dropdown to the right of the *Debug* button allows you to choose between these approaches.

A JIT cluster takes around five minutes to be provisioned. Even when using an active debug session, the pipeline will take a few moments longer to start than you have been used to – there is a short delay while compute resource is acquired from the debug cluster.

Inspect Execution Output

As the pipeline executes, activity execution information is displayed in its *Output* tab as shown in Figure 7-13.

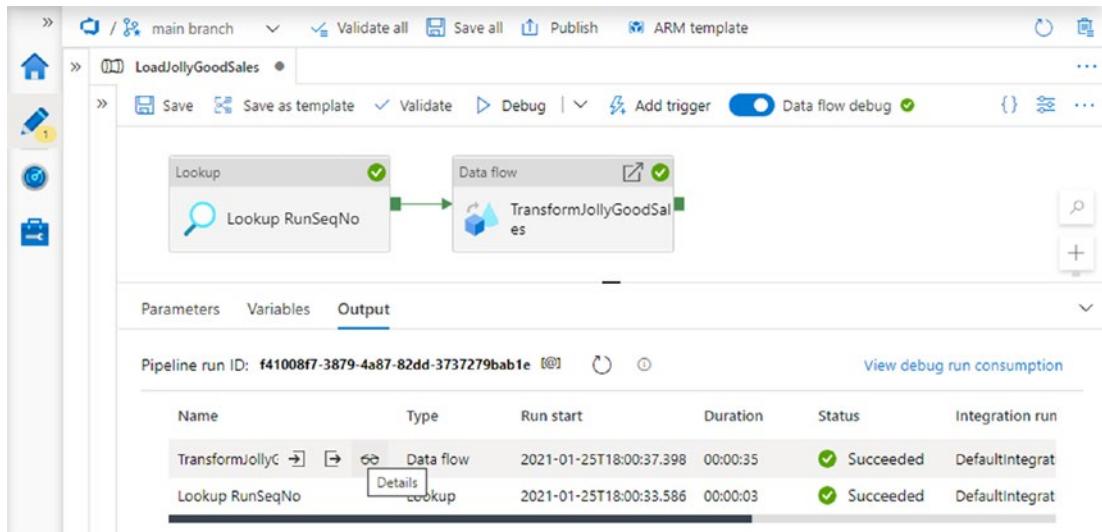


Figure 7-13. Activity output for the pipeline using the Data flow activity

The execution record for the Data flow activity, like that of the Copy data activity, features an eyeglasses *Details* button, visible in Figure 7-13 on the right of the activity's *Output* button. Click the *Details* icon to open a graphical monitoring view of the data flow.

The graphical monitoring view (shown in Figure 7-14) provides detailed information about data flow performance. The lower half of the figure breaks down the flow's execution time into groups of transformations that were executed together. The upper half contains a simplified view of the data flow structure, similar to the layout on the data flow canvas. Clicking a node on the canvas displays execution information for that transformation in a blade on the right – Figure 7-14 shows the result of selecting the “LookupExchangeRate” transformation.

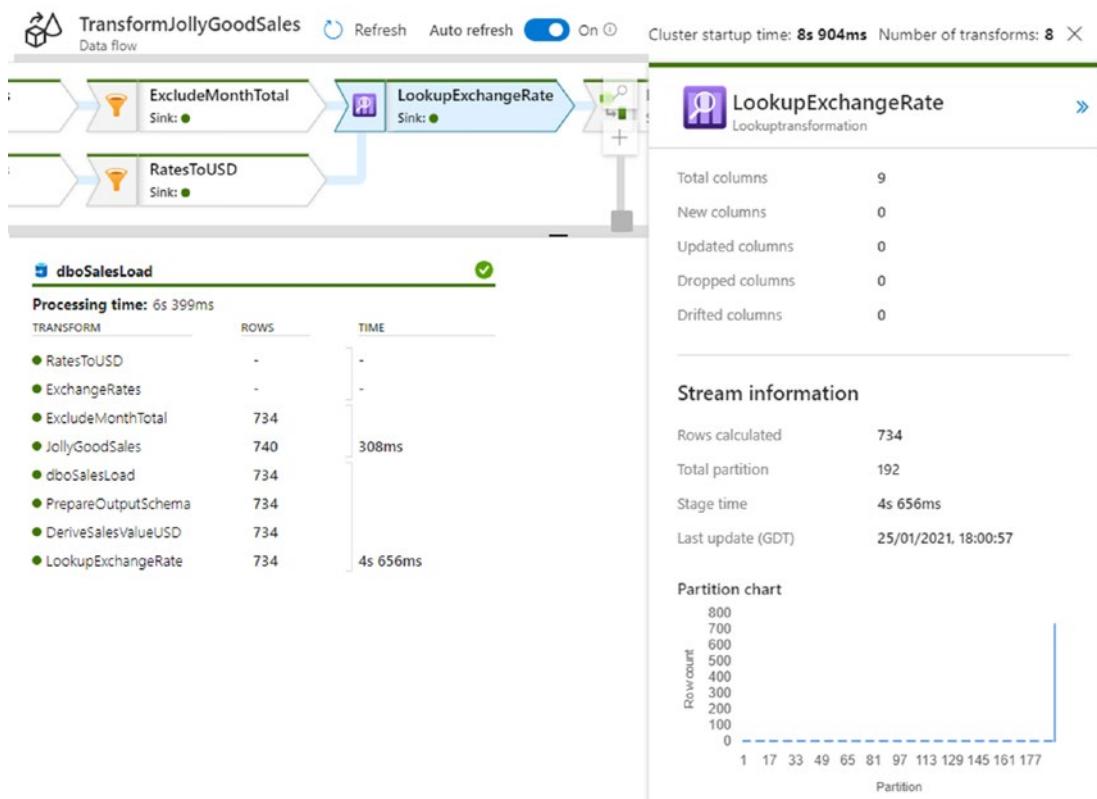


Figure 7-14. Data flow graphical monitoring view

The information presented in the blade is specific to the selected transformation, but often includes information about how data is partitioned during execution. Every transformation features an *Optimize* configuration tab on the data flow canvas – you can use this to make changes to how data is partitioned, but in most situations the default partitioning behavior is recommended.

Persist Loaded Data and Log Completion

To persist the data loaded by the Data flow activity, call the database stored procedure [dbo].[PersistLoadedSales].

1. Add a Stored procedure activity to your pipeline, configured to execute after the Data flow activity has successfully completed. On its *Settings* configuration tab, set its *Linked service* to your linked Azure SQL database.

2. Choose “[dbo].[PersistLoadedSales]” from the *Stored procedure name* dropdown, then beneath *Stored procedure parameters*, click the *Import* button. Provide an appropriate expression for the stored procedure’s “RunSeqNo” parameter (using the output of the pipeline’s initial Lookup activity).
3. Add a second stored procedure activity, also following the Data flow activity. Configure this one to call the stored procedure “[dbo].[LogPipelineEnd]”, importing its parameters in the same way as before. Configure expressions for “RunEndDate”,”RunSeqNo,” and “RunStatus.”
4. The Copy data activity’s output property paths you used previously for the parameters “FilesRead,” “RowsRead,” and “RowsCopied” are not valid for the Data flow activity. You can either treat these fields as null or find alternative paths in the Data flow activity’s output. Values corresponding to “RowsRead” and “RowsCopied” can be found, but you may have no option other than to treat “FilesRead” as null.
5. Rerun the pipeline, verifying that Jolly Good sales data has now been successfully persisted in database table [dbo].[Sales].

Maintain a Product Dimension

The data stored in table [dbo].[Sales] is not always easy to analyze, because the [Product] column combines two attributes. Each value includes the product’s weight, so grouping different formats of the same product is difficult. Calculating the weight of the product sold is hard, because product weights are not stored numerically and are a mixture of ounces and grams.

Allowing individual records to be grouped and aggregated using shared characteristics is a classic application for a dimension table. In this section, you will build a data flow to maintain a product dimension to support analysis of product sales.

Note By now, you should have accumulated sales data for several retailers in [dbo].[Sales]. If not, use the “ImportSTFormatFiles” pipeline you developed in Chapter 6 to reload data for Desserts4All and Naughty but Nice sales.

Create a Dimension Table

Listing 7-1 provides SQL code to create a dimension table called [dbo].[Product]. It has no integer key, but would be suitable for use in a SQL Server Analysis Services (SSAS) tabular model or for use in SQL-based analysis.

Use your SQL client to run the script and create the table in your Azure SQL Database.

Listing 7-1. Create a [dbo].[Product] table

```
CREATE TABLE dbo.Product (
    Product NVARCHAR(255) PRIMARY KEY
    , ProductName NVARCHAR(255) NOT NULL
    , WeightInOunces DECIMAL(19,2) NOT NULL
    , WeightInGrams DECIMAL(19,2) NOT NULL
);
```

Create Supporting Datasets

If you created a parameterized Azure SQL Database dataset as suggested in Chapter 6, you can use it here in the data flow. Alternatively, create ADF datasets for each of the two tables you’ll be using in this section:

1. Create a dataset to represent the existing [dbo].[Sales] table.
2. Create a second dataset to represent the new [dbo].[Product] table.

Build the Product Maintenance Data Flow

In this section, you will use a data flow to read product details out of the [dbo].[Sales] table, extract product names and weights, then append that information to [dbo].[Product].

1. Create a new data flow in your “Chapter7” data flows folder and name it “UpdateProduct.”
2. Add a Source transformation using the dataset for the [dbo].[Sales] table.

Tip When using parameterized datasets, runtime parameter values are supplied by the executing pipeline’s Data flow activity. At development time, specify values in the *Debug Settings* blade canvas (Figure 7-15), accessed using the button above the data flow. The button is only visible when the debug cluster is running.

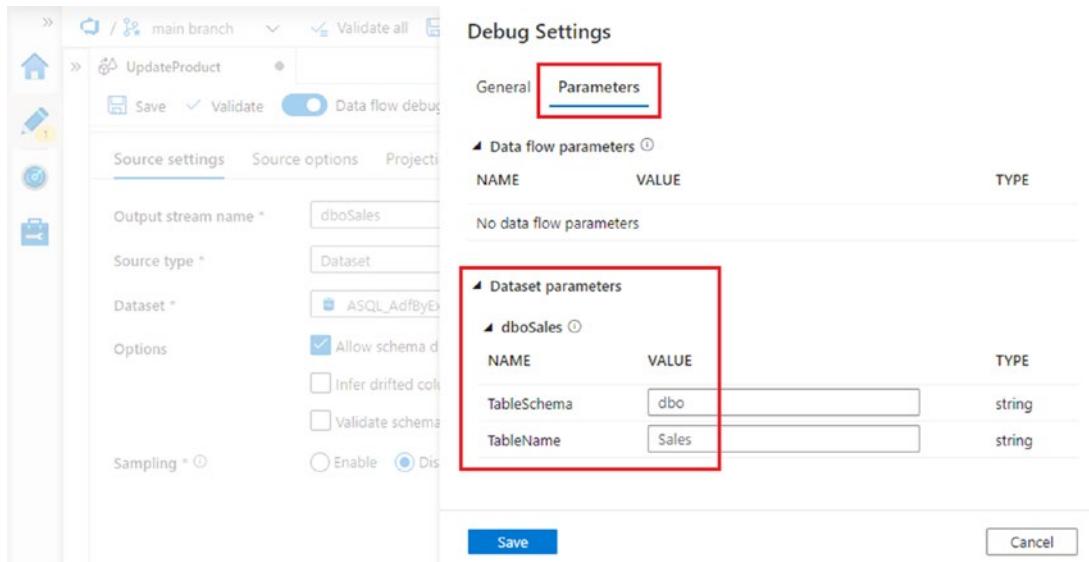


Figure 7-15. Use Debug Settings to set development dataset parameter values

3. If you are using a parameterized dataset, import the table schema using the *Import projection* button on the Source transformation’s *Projection* tab.

4. Connect a Derived Column transformation to the output of the Source transformation, then open the expression builder.

Use Locals

In the last section, you saw how to add new columns to a data flow using the Derived Column transformation. Sometimes, it is convenient to be able to store intermediate expressions for reuse in derived column definitions, without adding their results to the data flow stream. You can achieve this using *Locals*.

The value derived by a local expression can be used by other expressions within the same Derived Column transformation, but it is not included in the transformation's output. The product descriptions stored in [dbo].[Sales] contain a product name and weight, for example, "Chocolatey Nougat 3.52oz". Removing the weight string "3.52oz" from the product description yields the product name, and the same string can be reused as a simpler source of numeric weights.

1. In the *Expression elements* list of the *Visual expression builder*, select the *Locals* item. Under *Expression values*, click + *Create new* to open the *Create new local* blade.
2. Name the new local "WeightString" and create an expression to return only that portion of the product description. The weight string is the part of the description following the final space character – bear in mind that some descriptions contain multiple spaces.
3. Figure 7-16 shows the *Create new local* blade containing a suitable expression. The expression uses three functions: right, locate, and reverse. If you hover over a function in the *Expression values* list (as shown in the figure), or over its name in the *Expression* pane, a tooltip describing the function is displayed. Click *Create* to save the new local.

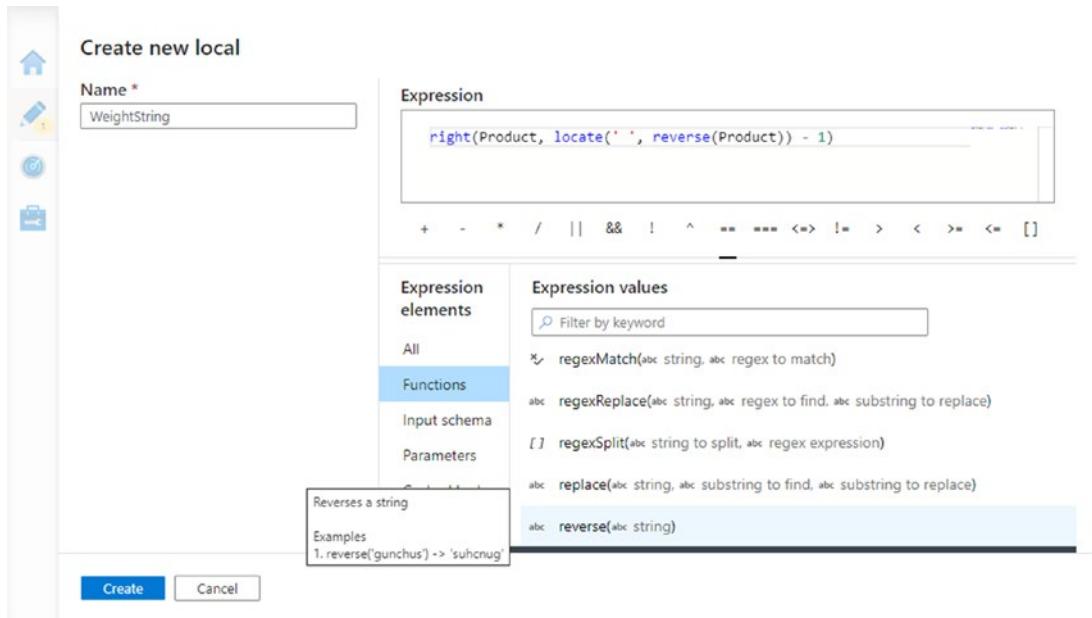


Figure 7-16. Creating a local expression in the Derived Column transformation

4. The new local appears in the list of *Expression values* when you select the *Locals* expression element – as shown in Figure 7-17 – and can be selected for use in an expression just like any other expression value. Create another local, this one called “WeightUnit,” using the expression `iif(endsWith(Product, 'oz'), 'oz', 'g')`. The expression returns the unit of weight being used in the product description.
5. Back in the expression builder for derived columns, name the new column “ProductName.” Use the “WeightString” local in an expression that returns only the name of the product. Figure 7-17 provides one such expression – the `:WeightString` element is a reference to the local.

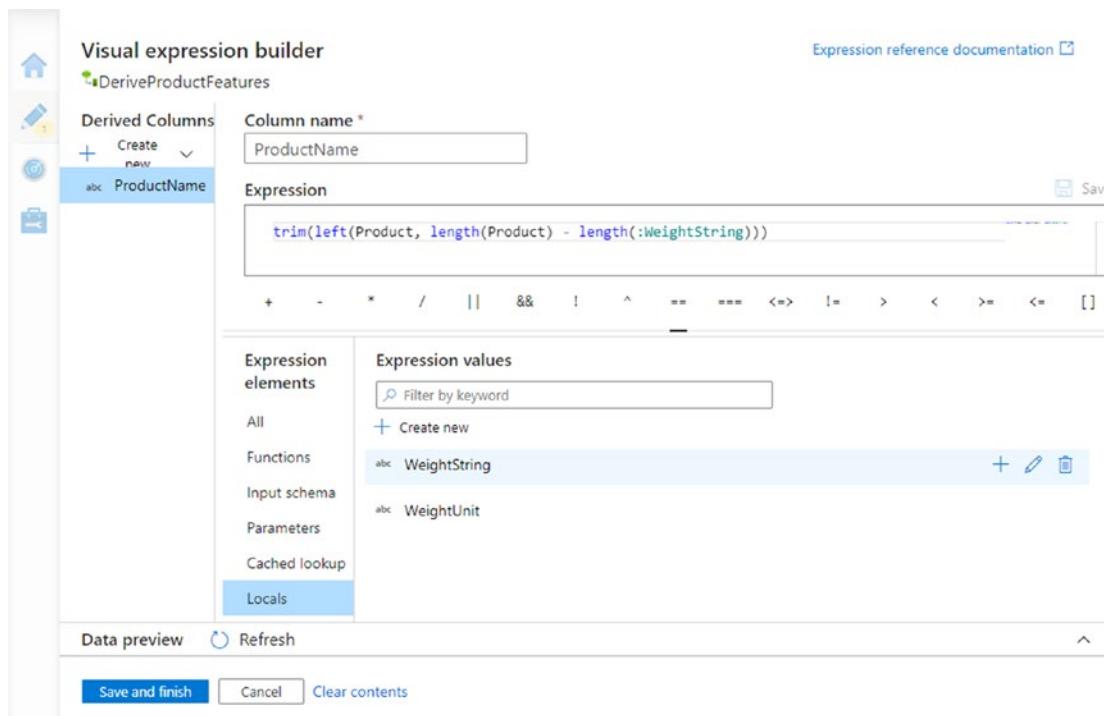


Figure 7-17. Using locals in a derived column expression

6. Add a second derived column called “WeightInOunces” – this should extract the numeric portion of the weight string and convert it from grams if necessary. A possible expression to achieve this is `toFloat(replace(:WeightString, :WeightUnit, '')) / iif(:WeightUnit=='g', 28.3495, 1.0)`.
7. Add a third derived column called “WeightInGrams.” Create an expression to perform a similar function to “WeightInOunces,” this one converting each product’s given weight to grams.
8. Use the *Data preview* tab to check the results of the transformation. Verify that your derived columns appear as expected, and notice that your local expressions do not.

Note The numbers resulting from product weight conversions will not round tidily to a few decimal places. This is because the input values used in product descriptions have already been rounded off.

Use the Aggregate Transformation

The Derived Column transformation's output stream contains one row for each time a product appears in the [dbo].[Sales] table. To make the "Product" field unique - as required by the primary key of [dbo].[Product] - you must remove duplicates. Do this using the *Aggregate transformation*.

1. Connect an Aggregate transformation to the output of the Derived Column transformation and name it appropriately.
2. On the *Aggregate settings* tab, ensure that the toggle below *Incoming stream* is set to *Group by*, then choose "Product" from the dropdown under *Columns*.
3. Switch the toggle to *Aggregates*. Under *Column*, select "ProductName" from the dropdown. The aggregate function required in the *Expression* is `first` - you can open the expression builder to construct this or simply enter `first(ProductName)` directly.
4. In many places, the data flow UX supports the specification of multiple columns using a *Column pattern*. Click the *+ Add* button above the Aggregates column list and select *Add column pattern*.
5. A column pattern uses an expression to identify a set of columns. In *Each column that matches*, enter the expression `startsWith(name, 'Weight')`. You can do this directly or using the expression builder. The expression identifies the two columns whose names begin with "Weight."
6. Below the column pattern expression are fields for the aggregate column name and expression. When using a column pattern, the string `$$` is a placeholder for each of the matched columns - set the column name to `$$` and the expression to `first($$)` as shown in Figure 7-18.

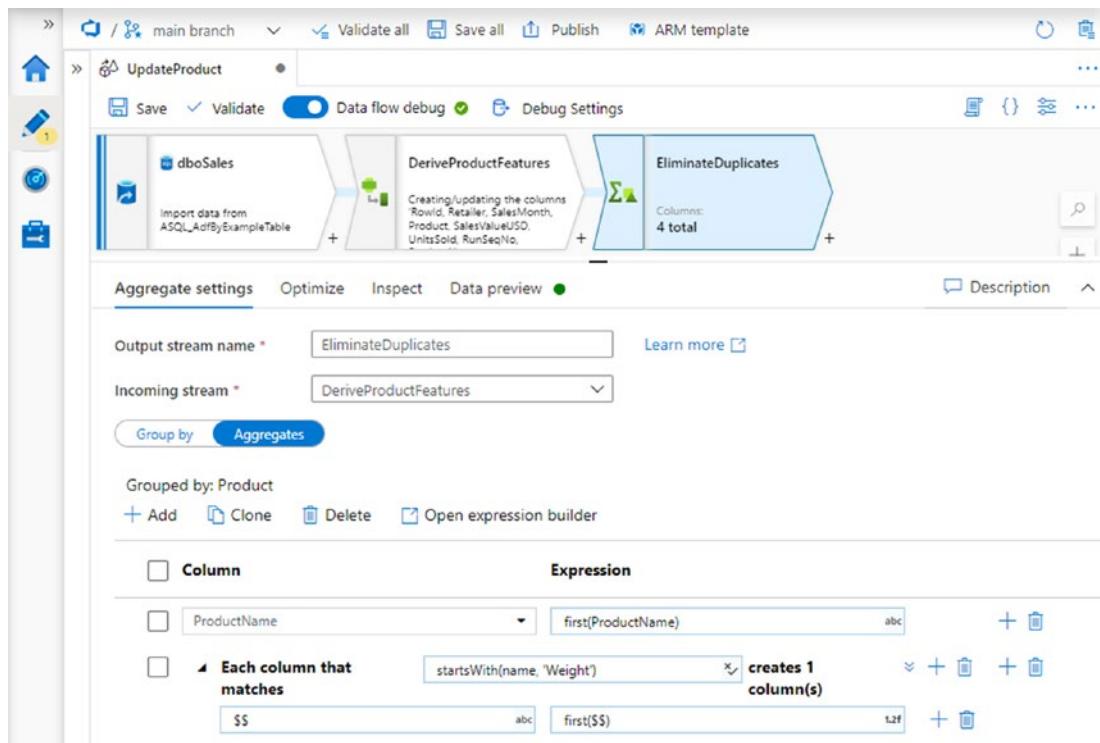


Figure 7-18. Using a column pattern to specify multiple aggregate columns

7. Use the *Data preview* tab to check the results of the transformation.

Use the Exists Transformation

At this moment, the [dbo].[Product] table is empty, but repeated runs of the data flow must add new rows only – attempting to re-add existing rows will cause a primary key violation in the database table, and the data flow will fail.

The *Exists transformation* enables you to filter values in a data stream based on whether they exist – or not – in another data stream. This data flow should emit rows only if they are not already present in the [dbo].[Product] table.

1. Add a second Source transformation to the data flow, using a dataset for the [dbo].[Product] table. If you are using a parameterized dataset, remember to import the table's schema on the transformation's *Projection* tab.

2. Connect an Exists transformation to the Aggregate transformation.
Set its *Right stream* to the new [dbo].[Product] Source transformation and its *Exist type* to “Doesn’t exist.”
3. The expressions used to define existence are specified under *Exists conditions*. Select the “Product” column on both the left and right sides on the condition.

Use the *Data preview* tab to check the results of the transformation if you wish. As table [dbo].[Product] is currently empty, the results will look the same as those returned by the Aggregate transformation preview.

Output the transformed data with a Sink transformation that uses the [dbo].[Product] table dataset, checking that the output column auto-mapping is correct. Figure 7-19 shows the completed data flow.



Figure 7-19. [dbo].[Product] maintenance data flow

Execute the Dimension Data Flow

The dimension maintenance data flow is now ready for execution.

1. Create a new pipeline named “UpdateProductDimension” in your “Chapter7” pipelines folder.
2. Add a Data flow activity to the canvas, selecting your new data flow. If you are using a parameterized dataset, the activity’s *Settings* configuration tab will be populated with input fields for parameter values – populate fields with the correct values to be used when the pipeline runs.
3. Run the pipeline. When execution is complete, verify that the [dbo].[Product] table has been correctly populated.

4. The data flow's Exists transformation ensures that the Sink transformation only adds new records to [dbo].[Product]. Run the pipeline again to verify that this is the case.
5. As a final test, delete some of the records in [dbo].[Product] (but not all of them). Run the pipeline again to demonstrate that the removed records are restored.

The [dbo].[Product] table populated here is a simple dimension-style table intended to support more flexible, ad hoc analysis of data in table [dbo].[Sales]. For example, the query in Listing 7-2 returns the number, total value, and total kilogram weight of each product sold between April and September 2020 – this calculation is possible because you have used the data flow to convert the unstructured product description into structured product attributes.

Listing 7-2. Sales analysis using [dbo].[Product]

```

SELECT
    p.ProductName
    , SUM(s.UnitsSold) AS UnitsSold
    , SUM(s.SalesValueUSD) AS SalesValueUSD
    , SUM(p.WeightInGrams * s.UnitsSold)/1000 AS KgSold
FROM dbo.Sales s
    INNER JOIN dbo.Product p ON p.Product = s.Product
GROUP BY p.ProductName
ORDER BY p.ProductName;

```

Tip The data flow you have created here implements a basic Slowly Changing Dimension (SCD) maintenance pattern, in this case, for a type 0 SCD. ADF UX *templates* provide reusable pipeline and data flow patterns and are available in ADF's *Template gallery*. To access the gallery, click the *Create pipeline from template* bubble on the ADF UX *Data Factory overview* page (home icon in the navigation sidebar).

Chapter Review

Chapters 3 to 6 used the Copy data activity to move data between source and sink datasets without much ability to modify data in flight. Introduced in this chapter, ADF's data flows close that gap, presenting powerful Databricks capabilities in an easy-to-use graphical user interface.

The requirement for a Databricks cluster may seem onerous given the time required to start one up on demand – a JIT cluster takes around five minutes to be provisioned, extending the execution time of ADF pipelines that make use of Data flow activities. In the published environment, Databricks clusters are always created just in time, when a Data flow activity execution begins.

The payoff is that Azure Data Factory is capable of transforming even extremely large datasets efficiently, saving time and scaling well by distributing dataset processing across cluster nodes. Furthermore, automatic cluster provision and teardown reduces cluster operating costs, by allowing you to pay only for what you use.

Key Concepts

Concepts encountered in this chapter include

- **Apache Spark:** Open source data processing engine that automatically distributes processing workloads across a cluster of servers (referred to as *nodes*) to enable highly parallelized execution.
- **Databricks:** Data processing and analytics platform built on Apache Spark and adding a variety of enterprise features.
- **Data flows:** ADF's visual data transformation tool, built on Azure Databricks.
- **Data flow debug:** Data flow debug mode provisions a Databricks cluster on which you can execute data flows from the ADF UX.
- **Time to live (TTL):** The data flow debug cluster has a default TTL of one hour, after which – if it is not being used – it automatically shuts down.
- **Data flow activity:** ADF pipeline activity used to execute a data flow.

- **Parameters:** Data flow parameters are specified in Debug Settings during development and substituted for values supplied by the calling Data flow activity at runtime.
- **Data flow canvas:** Visual development environment for data flows.
- **Transformation:** A data flow is made up of a sequence of connected transformations, each of which modifies a data stream in some way.
- **Output stream name:** Name that uniquely identifies each transformation in a data flow.
- **Inspect tab:** Use a transformation's Inspect tab to view input and output schema information.
- **Data preview tab:** Use a transformation's Data preview tab to preview data emitted by the transformation. Using data preview requires data flow debug to be enabled and can be used to delay an approaching cluster timeout.
- **Optimize tab:** Use a transformation's Optimize tab to influence data partitioning in Spark when the transformation is executed.
- **Source transformation:** Reads input data from an external source. Every data flow starts with one or more Source transformations.
- **Sink transformation:** Write transformed data to an external source. Every data flow ends with one or more Sink transformations.
- **Data flow expression language:** Data flow expressions have their own language and expression builder, different from those of ADF pipeline expressions.
- **Data Flow Script:** Language in which data flow transformations are stored, embedded in a data flow's JSON file.
- **Column patterns:** Where supported, use column patterns to specify multiple columns which are to be handled in the same way. Columns are specified using data flow expressions to match column metadata.
- **Filter transformation:** Selects rows from its input data stream to be included in its output stream, on the basis of criteria specified as a data flow expression. Other rows are discarded.

- **Lookup transformation:** Conceptually similar to a SQL join between two data streams. Supports a variety of join styles and criteria.
- **Derived Column transformation:** Uses data flow expressions to derive new columns for inclusion in a data flow.
- **Locals:** Named intermediate derivations in a Derived Column transformation. Used to simplify expressions and eliminate redundancy.
- **Select transformation:** Used to rename columns in a data flow or to remove them.
- **Aggregate transformation:** Aggregates one or more columns in a data flow, optionally grouping by other specified columns.
- **Exists transformation:** Selects rows from its input data stream to be included in its output stream, on the basis of the existence (or not) of matching rows in a second data stream. Other rows are discarded.
- **Templates:** Reusable implementations of common pipeline and data flow patterns.
- **Template gallery:** Source of provided templates, accessed using the *Create pipeline from template* bubble on the *Data Factory overview* page.

For SSIS Developers

Azure Data Factory data flows are very similar to the implementation of an SSIS Data Flow Task on the data flow surface. ADF's Data flow activity performs the same function as a Data Flow Task – representing the data flow on the pipeline's control surface – but unlike in SSIS, the data flow implementation is a completely separate factory resource that can be referenced independently and reused.

Familiar notions from the SSIS data flow surface are found within a data flow. SSIS's wide variety of source and destination components is unnecessary, partially because connection details are already abstracted away using ADF datasets and linked services, and partially because the single Source and Sink transformations encapsulate the detail of multiple external systems.

Most of the ADF data flow transformations encountered in this chapter have an equivalent component in SSIS, although detailed behavior may differ:

- SSIS's Conditional Split component has a direct ADF equivalent, also called Conditional Split. The *Filter* transformation's behavior is a simplified version of the Conditional Split in which rows are either retained or discarded.
- The *Lookup* ADF transformation and SSIS components provide similar functionality, although the more sophisticated join behavior of the ADF transformation makes it closer to a Merge Join component. (The Lookup transformation's functionality overlaps that of ADF's own Join transformation.)
- Both ADF and SSIS feature a *Derived Column* transformation/component, although SSIS lacks the additional flexibility of Locals.
- ADF's *Select* transformation to rename or remove columns has no equivalent in SSIS.
- Both ADF and SSIS feature an *Aggregate* transformation/component, although in general SSIS lacks the flexibility of column pattern support used in the preceding example.
- Integration Services provide no direct *Exists* transformation – existence testing in SSIS must be performed using the Lookup component and the Lookup Match/No Match outputs.

CHAPTER 8

Integration Runtimes

ADF linked services represent connections to storage or compute resource – decoupled from one another in the way described in Chapter 2 – that are *external* to Azure Data Factory. The linked services you have been using in previous chapters represent connections to external storage, and access to external compute (such as HDInsight or Azure Databricks) is managed in the same way.

This chapter is concerned with *internal* resource, specifically compute, as ADF has no storage resource of its own. *Internal pipeline activities* such as the Copy data, Lookup, Get Metadata, and Data flow activities are not executed using external compute, but are executed instead using a compute resource internal to ADF, called an *integration runtime* (IR).

In contrast, an *external pipeline activity* uses external compute resource (specified as a linked service) to do its work, although its execution is still managed in ADF by an integration runtime. For example, an ADF IR *dispatches* the Stored procedure activity, but actual T-SQL execution takes place in an external database service. The detail of most external activities is outside the scope of this book, because their function is to execute processes implemented in technologies other than ADF.

Azure Integration Runtime

An *Azure Integration Runtime* can be used to perform Copy data and Data flow activities in ADF and to manage the execution of a variety of other activities. Every data factory is equipped at creation with a default Azure Integration Runtime, called *AutoResolveIntegrationRuntime* – this is the integration runtime you have been using to execute pipeline activities in earlier chapters. With the exception of the Data flow activity (which specifies its IR directly), the IR used to execute an internal pipeline activity is the one specified in the activity's linked service storage connection.

Inspect the AutoResolveIntegrationRuntime

Integration runtimes are defined and managed in the ADF UX's management hub. Use the management hub to locate and inspect the definition of the AutoResolveIntegrationRuntime.

1. In the ADF UX, open the management hub and select *Integration runtimes* from the menu. The factory's integration runtimes are displayed – at this stage, only one is defined, the default AutoResolveIntegrationRuntime (shown in Figure 8-1).

Name	Type	Sub-type	Status	Related	Region
AutoResolveIntegrationRuntime	Azure	Public	Running	0	Auto Res

Figure 8-1. Integration runtimes visible in the management hub

2. Click the name of the runtime to open the *Integration runtime* blade. You cannot edit any of the default runtime's settings, but notice particularly that the runtime's *Region* is set to "Auto Resolve."
3. Click *Close* to close the blade. If prompted, click *Discard changes* – you are unable to make or save changes to the AutoResolveIntegrationRuntime.

While in the management hub, you may also wish to inspect the definitions of your blob storage and Azure SQL Database linked services. You will notice that in each case the linked service's *Connect via integration runtime* option is set to "AutoResolveIntegrationRuntime" – both of these services are using the default Azure IR.

Create a New Azure Integration Runtime

Reasons you might want to create your own Azure Integration Runtime (instead of using the default AutoResolveIntegrationRuntime) include

- You want to increase the default time to live (TTL) for a just-in-time (JIT) Databricks cluster.
- You need to guarantee that data movement takes place within a given geographical region.

These reasons are explained in greater detail later. First, create a new Azure Integration Runtime as follows:

1. On the management hub's *Integration runtimes* page, click the *+ New* button.
2. The *Integration runtime setup* blade appears. Select the *Azure, Self-Hosted* tile and click *Continue*.
3. Under *Network environment*, select the *Azure* tile and click *Continue*.
4. You will create the integration runtime in a region geographically distant from that of your data factory and other Azure resources. Specify a *Name* for the IR that reflects this.
5. Use the *Region* dropdown to select a region different from your data factory instance. For the purposes of this exercise, choose a region far away from that of your data factory instance.
6. Expand *Data flow run time* options and set *Time to live* to “10 minutes.”
7. The correctly completed blade is shown in Figure 8-2 - my data factory is in the “UK South” Azure region, so I am creating an IR in “Australia Southeast.” Click *Create* to create the IR.

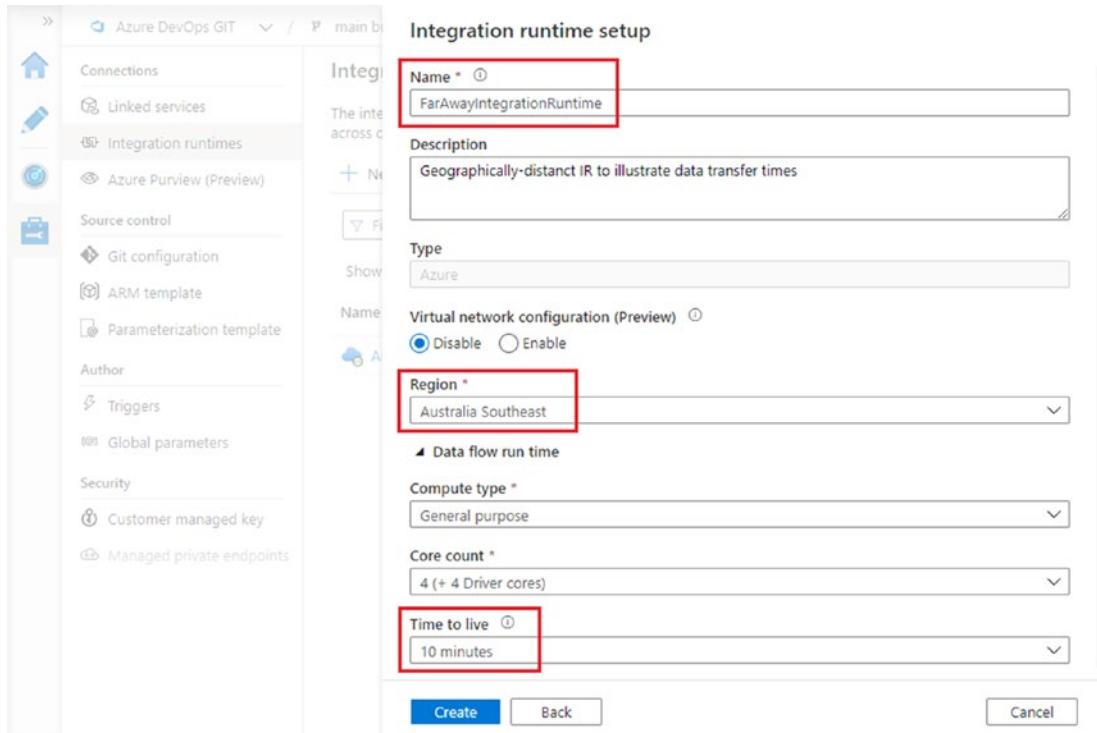


Figure 8-2. Creating a new Azure Integration Runtime

Databricks Cluster TTL

In Chapter 7, in order to run data flows in debug mode, you enabled *Data flow debug*. Enabling Data flow debug provisions a Databricks cluster where your data flows can be executed in pipeline debug runs. Provisioning a cluster takes several minutes, but the debug cluster's TTL of one hour enables you to continue working in the ADF UX without interruption.

When a Data flow activity is executed by a published pipeline, the Databricks cluster is provisioned just in time, when the activity's execution begins. The cluster's TTL is configured in the integration runtime chosen for the activity (selected using the *Run on (Azure IR)* option on the Data flow activity's *Settings* configuration tab). The AutoResolveIntegrationRuntime specifies a TTL of zero minutes, so a JIT cluster provisioned for a Data flow activity using this runtime is torn down as soon as the activity's execution is complete.

The effect of this in a pipeline containing multiple Data flow activities is that a new Databricks cluster must be provisioned before every such activity, adding a few

minutes' delay to the execution of each one. To reduce the overall pipeline execution duration, create an integration runtime with a longer TTL, as before, so that the cluster provisioned for the first Data flow activity is still running when subsequent activities are queued.

Controlling the Geography of Data Movement

The AutoResolveIntegrationRuntime IR takes its name from the “Auto Resolve” region specified in its definition. In the case of the Copy data activity, “Auto Resolve” means that Azure Data Factory will attempt to use an IR located in the same region as the activity’s sink data store, but this is not guaranteed. Under some circumstances it might be possible for data to be copied by an IR in a region different from both source and sink data stores.

For certain data processing applications, you may have a legal obligation to ensure that data does not leave a given jurisdiction. In this situation, you need to be able to guarantee the location of the Copy data activity’s integration runtime as well as the location(s) of the source and sink data stores. Using an integration runtime in a specified region, as before, enables you to achieve this.

Use the New Azure Integration Runtime

Your new IR is now ready for use in new or existing linked services. In this section, you will see the effect of doing so.

Identify the Copy Data Effective Integration Runtime

To see the effect of changing a linked service’s IR, first run one of your existing pipelines to obtain a point of reference.

1. In the ADF UX, open and run your “ImportSampleData” pipeline from Chapter 2.
2. When complete, inspect the Copy data activity’s execution in the pipeline configuration’s *Output* tab. The *Integration runtime* column identifies the IR used by the activity.

As illustrated in Figure 8-3, the IR used is “DefaultIntegrationRuntime,” in the region selected by Auto Resolve.

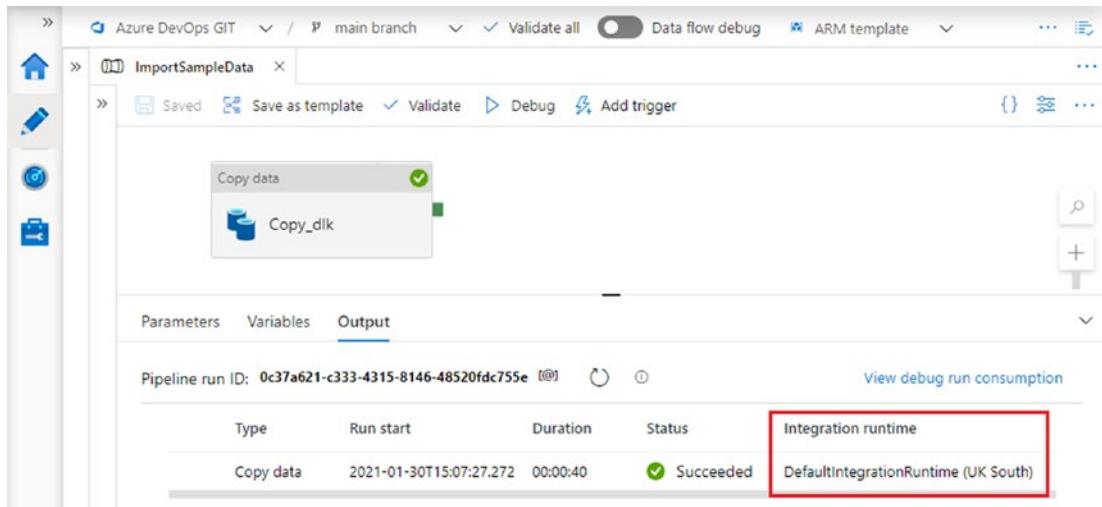


Figure 8-3. DefaultIntegrationRuntime used for the Copy data activity

Revise the Sink Integration Runtime

To use the new integration runtime in one of your existing pipelines, update the linked service used by the Copy data activity's sink data store. You can do this directly from the activity definition on the authoring canvas.

1. Select the Copy data activity on the authoring canvas, then open its *Sink* tab. To the right of the *Sink dataset* dropdown, click the *Open* button.
2. The activity's destination dataset opens in a new authoring workspace tab. Click the *Edit* button on the right of the *Linked service* dropdown.
3. On the *Edit linked service (Azure Blob Storage)* blade, select the new IR from the *Connect via integration runtime* dropdown, then click *Apply* to save your changes.
4. Return to the “ImportSampleData” pipeline and run it again. You are likely to experience a considerably longer runtime.
5. While the pipeline is executing, hover over the activity name in the pipeline configuration’s *Output* tab to reveal the *Details* button (eyeglasses icon). Click the button to open the *Details* popup.

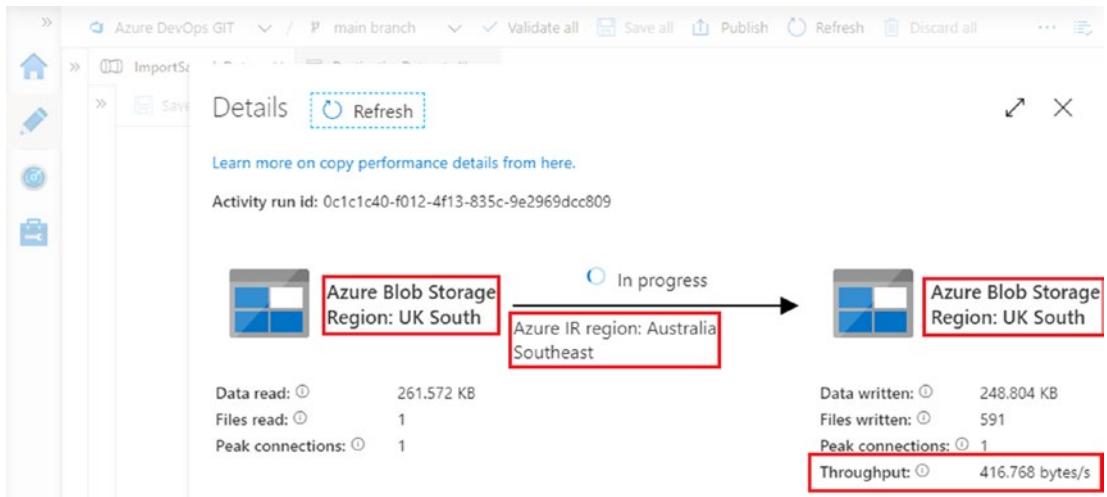


Figure 8-4. Data copy by an IR distant from storage exhibits a low throughput

Figure 8-4 shows the *Details* popup for the in-progress Copy data activity. The “ImportSampleData” pipeline copies data from one container to another, in the same blob storage account. In this example, the blob storage account is in the UK South Azure region, but the copy is being performed by compute in Australia Southeast – loosely speaking, a computer in Southeast Australia is reading data from a UK blob storage location, then writing it back to a different UK blob storage location. The result is that the data throughput is very low – when I ran the pipeline using the revised IR, it took over 14 minutes to complete compared to 41 seconds when using the AutoResolveIntegrationRuntime.

By creating a new IR in a different region and using it in a linked service, you have relocated the Copy data activity’s processing away from a data center in your data factory’s region and into another data center of your choice, somewhere else in the world. This has an impact on activity execution times and incurs additional processing costs in the form of a bandwidth charge (sometimes called an egress charge). In contrast, using an IR located explicitly in the same region as your source and sink data stores guarantees that data does not leave that region during processing.

Before continuing, reset the linked service’s integration runtime to “AutoResolveIntegrationRuntime.”

Self-Hosted Integration Runtime

The purpose of a *self-hosted integration runtime* is to provide secure access to resources in private networks, for example, on-premises source systems, or resources located in an Azure virtual network. Installed on a Windows machine inside a private network, the Integration Runtime Windows service provides a gateway into that network for Azure Data Factory. A self-hosted IR supports connections to a variety of private resource types, including file systems, SQL Server instances, and ODBC-compliant database services.

A self-hosted IR service inside a private network is linked to exactly one Azure Data Factory instance – this is insufficiently flexible in environments requiring multiple data factories. A common solution to the problem is to implement a separate data factory instance, used to encapsulate and share self-hosted IR connections. Other data factories are able to link to self-hosted IRs using the shared data factory, without requiring a direct relationship to the underlying gateway implementation.

Figure 8-5 illustrates this implementation pattern. Although the diagram indicates a boundary between a private network and Microsoft Azure, it applies equally to private networks implemented inside Azure – in such cases, the boundary is between the private Azure virtual network and the rest of the Azure cloud.

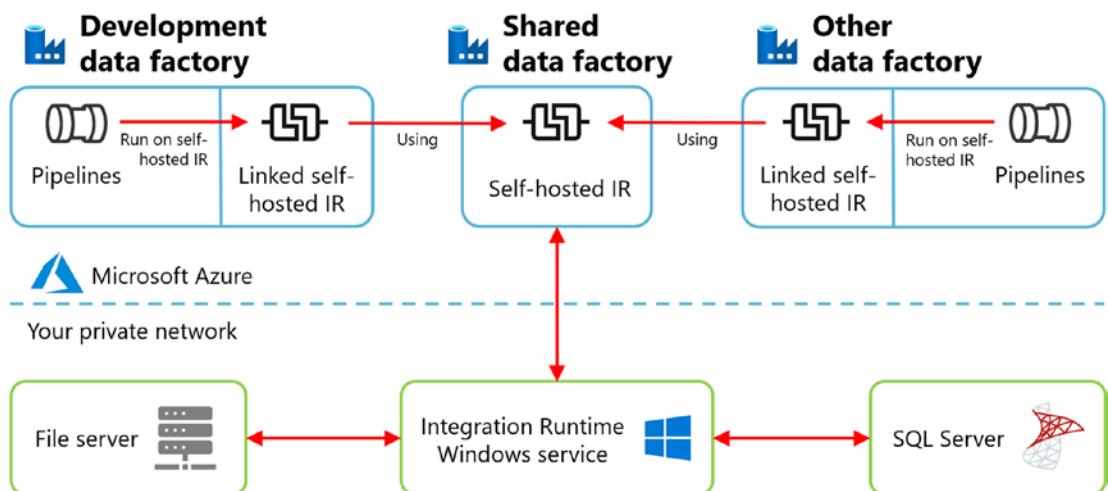


Figure 8-5. Multiple data factories sharing a self-hosted integration runtime

In this section, you will create a shared data factory, install a self-hosted integration runtime, and link to it from your existing development factory. If your computer is already supporting a self-hosted IR, you cannot install another one – the “*Link to a Self-Hosted Integration Runtime*” section in this chapter describes how you can link your development factory to an existing self-hosted IR.

Create a Shared Data Factory

To begin with, create a new data factory instance to contain your self-hosted IR and to share access to it.

1. In the Azure portal, use the *Create a resource* button to start the creation of a new data factory.
2. Complete the *Basics* tab, providing a unique name for the new factory and specifying other details to match your development factory.
3. On the *Git configuration* tab, tick the *Configure Git later* checkbox. From here, you can skip straight to *Review + create*, then click *Create* to create the new factory instance.

It is good practice also to link your shared data factory to a Git repository, but for the purposes of this exercise, it is not necessary to do so.

Create a Self-Hosted Integration Runtime

You can visualize a self-hosted IR as a gateway between Azure Data Factory and your private network. The private network end of that connection consists of a Windows service installed on a nominated machine inside the private network. (At the time of writing, self-hosted IRs are only supported on Windows systems.) In this section, you will create both ends of the connection.

1. Open the ADF UX in your new data factory instance.

Tip When connected to your development data factory, use the *Switch Data Factory* button in the ADF UX navigation header bar to connect to your new shared factory.

2. Use the management hub's *Integration runtimes* page to create a new IR of the *Azure, Self-Hosted* kind. This time, choose the *Self-Hosted* network environment option.
3. Name the IR appropriately and click *Create*. This creates the Azure end of the connection into your network.
4. The *Integration runtime setup* blade displays configuration tabs for the self-hosted IR. The *Settings* tab offers two options for setting up the self-hosted IR machine (the private network end of the connection). *Option 2: Manual setup* requires you to install the integration runtime, then to register it using one of the displayed authentication keys, while *Option 1: Express Setup* downloads a customized installer that manages both installation and registration. Click the option 1 link *Click here to launch the express setup for this computer*.
5. The customized installer ".exe" file is downloaded to your computer – run it to install the self-hosted IR service. The installer steps through four stages before displaying a confirmation message. Click *Close* and return to the ADF UX.
6. Close the *Integration runtime setup* blade. You can see the new self-hosted integration runtime displayed in the list alongside your Azure IRs. If its status is not shown as "Running," click the *Refresh* button to update the list.

The status of the self-hosted IR is dependent on the corresponding Windows service running on your machine. If your machine is offline or the Windows service is stopped, the IR will become unavailable.

Link to a Self-Hosted Integration Runtime

To link to the self-hosted runtime from your development data factory, you must first share the runtime. Having done so as follows, you will proceed to create the linked self-hosted IR.

1. In the ADF UX management hub of your shared data factory, open the *Edit integration runtime* blade for the self-hosted IR from the *Integration runtimes* page.
2. The *Edit integration runtime* blade's *Sharing* tab displays the self-hosted IR's fully qualified Azure resource ID – click the *Copy* button to the right of the *Resource ID* field to copy its value to the clipboard.
3. Still on the *Sharing* tab, click *+ Grant permission to another Data Factory*. You can search for your development factory using its name or managed identity, but you will probably be able to see it immediately below the search box. Tick the checkbox to select the factory, then click *Add*. Click *Apply* to save your changes and close the blade.
4. Use the *Switch Data Factory* function to reconnect the ADF UX to your development data factory, then open its management hub.
5. Use the management hub's *Integration runtimes* page to create a new IR, again of the *Azure, Self-Hosted* kind. Under *External Resources*, select the *Linked Self-Hosted* tile and click *Continue*.
6. Name the IR appropriately, then in the *Resource ID* field, paste the self-hosted IR's fully qualified resource ID, copied to the clipboard in step 2. Click *Create*.

You have now created both sides of the IR link between your two data factories – in the shared factory, you have granted the development factory access to the self-hosted IR, and in the development factory, you have created a link to use it.

Use the Self-Hosted Integration Runtime

A major purpose of a self-hosted IR is to enable the secure transfer of data between Azure and a private network, for example, to upload data from on-premises file stores or database systems. In this section, you will use your development data factory to copy a file from your own computer into blob storage, using the linked self-hosted IR.

Create a Linked Service Using the Shared Self-Hosted IR

To connect to your computer's file system from Azure Data Factory, a linked service is required.

1. In the development data factory's ADF UX management hub, create a new linked service. Select the *File system* data store and click *Continue*.
2. On the *New linked service (File system)* blade, name the linked service and choose the linked self-hosted IR from the *Connect via integration runtime* dropdown.
3. The *Host* field indicates the location inside your network from which you will be copying data – this could be a server UNC path if copying data from a separate file store or a location on the self-hosted IR machine itself. For now, you will copy data from your own computer – enter “C:\” (or an alternative drive letter of your choice).
4. Enter a *User name* and *Password* for ADF to use when connecting to your computer.

Tip To avoid publishing your account credentials to ADF, create a key vault secret to store your account password.

5. Click *Test connection* to verify that the linked service has been correctly configured and is able to access the integration runtime. When your test connection has succeeded, click *Create* to create the new linked service.

Figure 8-6 shows the configured linked service blade, using a key vault secret to provide the local account login password.

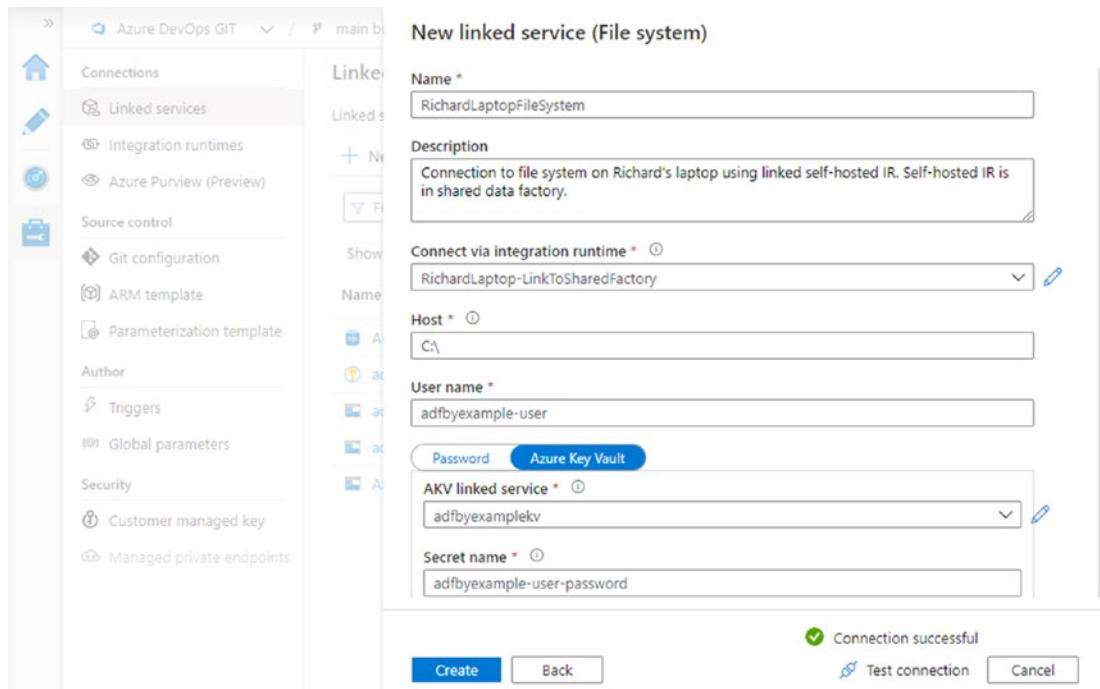


Figure 8-6. File system linked service using a self-hosted IR

Create a File System Dataset

The new linked service represents a connection from Azure Data Factory to your computer's file system. To represent specific files, a dataset is required.

1. On the ADF UX authoring canvas, create a new dataset and a new “Chapter8” folder to contain it. Select the *File system* data store again, then choose the *Binary* file format. (The object of this exercise is simply to copy some files into Azure, so their internal structure is not important right now.)
2. Choose the file system linked service you created in the previous section, noticing that the first *File path* field is prepopulated with the *Host* value from the linked service.
3. Use the browse button (to the right of the three *File path* fields) to locate a file to load from your computer. The completed blade in Figure 8-7 shows a file of sample data chosen from my own computer.

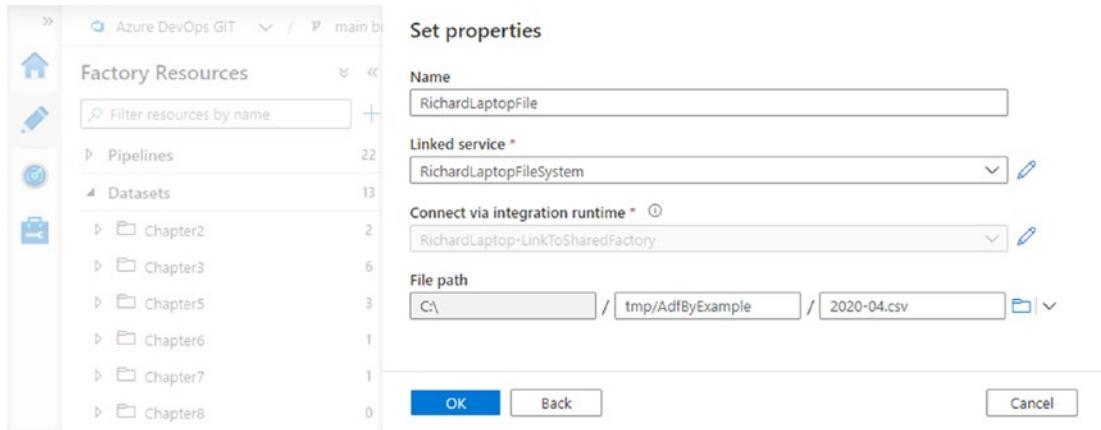


Figure 8-7. Dataset configured to represent a file on a local computer

4. Click *OK* to create the dataset, then save your changes.

Copy Data Using the File System Dataset

The dataset you created in the previous section can be used like any other. In this section, you will use it as a Copy data source to copy the file from your computer and into Azure blob storage.

1. Create a new pipeline and drag a Copy data activity onto the authoring canvas. Set the Copy data activity's source dataset to the new file system dataset.
2. Create a sink dataset for the Copy data activity using the *+ New* button on the activity's *Sink* configuration tab. The new dataset should use the *Azure Blob Storage* data store and *Binary* file format. Use your existing Azure Blob Storage linked service and specify a file path in your “landing” container.
3. Run the pipeline. When it has completed successfully, use the Azure portal's Storage Explorer to verify that the file is present in your “landing” container.

In addition to copying data between private network and cloud data storage, the self-hosted IR can dispatch certain activities to on-premises compute services – for example, you can use the Stored procedure activity to execute a stored procedure on an on-premises SQL Server. You cannot use a self-hosted IR to perform every ADF

activity – to run data flows using on-premises data, you must first copy data into Azure storage (using a self-hosted IR), after which it can be used by a Data flow activity running on an Azure IR.

In production environments, care must be taken to ensure that a self-hosted IR does not become a bottleneck or single point of failure. A self-hosted IR can be scaled out over up to four machines – referred to as *nodes* – running the Integration Runtime Windows service, allowing greater throughput and increasing availability in the event that one of them fails. Each node is registered to the same self-hosted IR in Azure Data Factory, so the abstraction of a single logical gateway into your network is maintained.

Figure 8-5 showed an Integration Runtime service installed on a machine separate from the resources being exposed to ADF. While not essential, this can assist throughput by avoiding contention with other workloads. If CPU and memory usage is low but throughput remains limited, you may consider increasing the number of concurrent jobs permitted to run on the node, via the *Nodes* tab on the *Edit integration runtime* blade.

Azure-SSIS Integration Runtime

As its name suggests, the purpose of the *Azure-SSIS Integration Runtime* is to permit the execution of SQL Server Integration Services packages in Azure Data Factory. SSIS packages running on an Azure-SSIS IR are unable to take full advantage of ADF's capabilities, but this can be a useful intermediate stage for organizations seeking to migrate legacy data processing quickly into Azure. This section assumes some familiarity with SSIS; a wider introduction is outside the scope of this book.

An Azure-SSIS IR uses an SSIS catalog database in exactly the same way as an on-premises SSIS installation, so you are required to manage some of the service infrastructure yourself. While you can generally consider ADF to be a serverless service, this makes the Azure-SSIS IR closer to an IaaS offering.

Create an Azure-SSIS Integration Runtime

In this section, you will create an Azure-SSIS Integration Runtime.

1. Use the management hub's *Integration runtimes* page to create an IR. Choose the *Azure-SSIS* tile to open the *General settings* page of the *Integration runtime setup* blade, then provide a name for the new IR. Set its location to match that of your data factory.

2. An Azure-SSIS IR is a cluster of Azure virtual machines, managed for you by ADF. *Node size* and *Node number* settings describe the physical makeup of the cluster – how powerful each node in the cluster needs to be and how many nodes it contains. You can tune these settings to the needs of your workload – large, compute-heavy packages benefit from more powerful nodes, while a greater number of nodes may permit more packages to be processed in parallel.

The running cost of the Azure-SSIS IR is in proportion to the size and power of its cluster, your choice of SQL Server Standard or Enterprise edition on cluster VMs, and whether or not you already own an existing SQL Server license. Choose a variety of different settings to observe the effect on the estimated running cost, displayed at the bottom of the blade as shown in Figure 8-8.

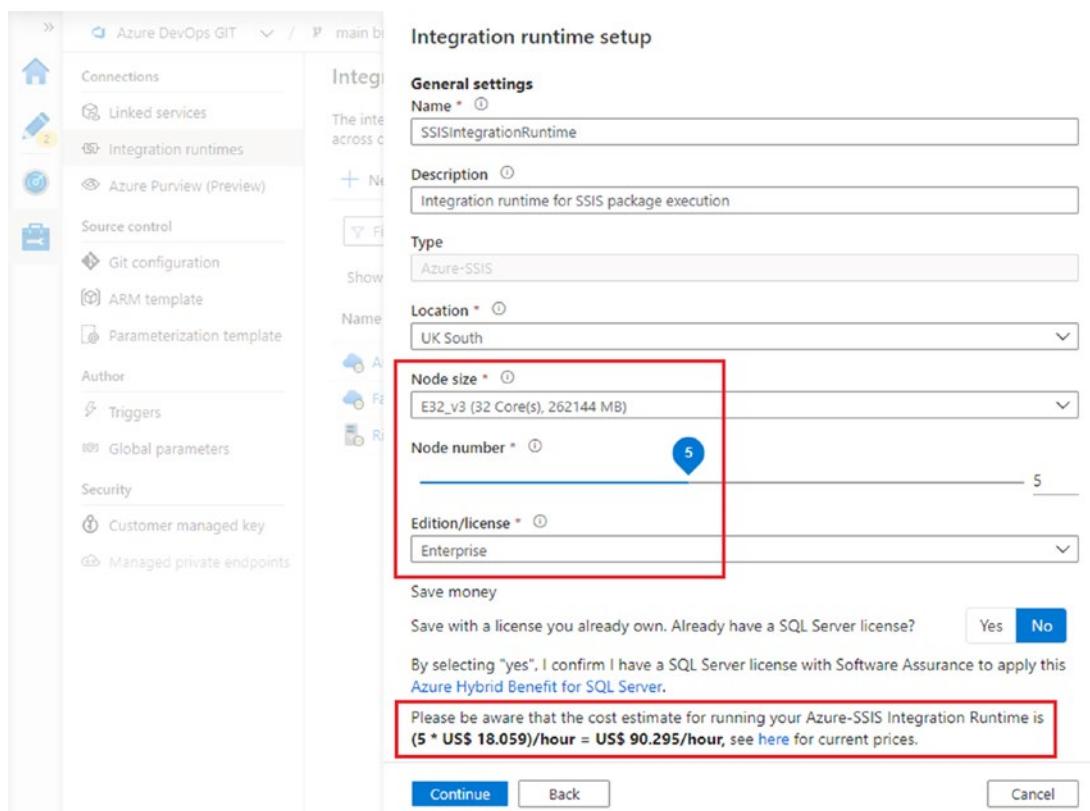


Figure 8-8. Azure-SSIS IR running costs are linked to cluster power and size

3. For the purposes of this exercise, choose the smallest available node size and number, and set *Edition/license* to “Standard.” Click *Continue*.
4. The *Deployment settings* page relates to the SSIS catalog database. Ensure that the *Create SSIS catalog...* checkbox is ticked at the top of the blade, then choose your existing Azure SQL Database server from the *Catalog database server endpoint* dropdown – this is where ADF will create the catalog database.
5. Provide your server’s admin username and password, then use the *Test connection* button at the bottom of the blade to verify that you have entered them correctly. Finally, set the *Catalog database service tier* to “GP_S_Gen5_1”, then click *Continue*.

Note “GP_S_Gen5_1” is a serverless service tier, the same one used by the Azure SQL Database you created in Chapter 3. This is a low-cost choice for learning purposes, but a provisioned service tier may be more appropriate in a production environment. The cost of running the SSIS catalog database is additional to that incurred by operating the Azure-SSIS Integration Runtime.

6. Accept the default values on the *Advanced settings* page by clicking *Continue*. Review the settings you have provided and the associated cost estimate on the *Summary* page, then click *Create*.
7. The new Azure-SSIS IR appears in the list of integration runtimes with a Status of “Starting.” A few minutes is required to provision and start the IR cluster – use the *Refresh* button above the list to monitor the IR’s status (indicated in Figure 8-9).

The screenshot shows the Azure DevOps interface for managing integration runtimes. On the left, there's a sidebar with various project management and development tools like Connections, Linked services, and Source control. The main area is titled "Integration runtimes" and displays a table of existing runtimes. The table includes columns for Name, Type, Sub-type, Status, Related, and Region. The "SSISIntegrationRuntime" entry is highlighted with a red box and has a status of "Starting".

Name	Type	Sub-type	Status	Related	Region
AutoResolveIntegrationRuntime	Azure	Public	Running	0	Auto F...
FarAwayIntegrationRuntime	Azure	Public	Running	0	Austral...
RichardLaptop-LinkToSharedFactory	Self-Hosted	Linked	Running	1	...
SSISIntegrationRuntime	Azure-SSIS	---	Starting	0	UK So...

Figure 8-9. Azure-SSIS IR starting

8. Use the Azure portal to inspect the list of databases present in your Azure SQL Database server. Notice that a new SSISDB database has been created – this is the SQL database for the new IR's SSIS catalog.

Deploy SSIS Packages to the Azure-SSIS IR

The GitHub repository that accompanies this book can be found at <https://github.com/Apress/azure-data-factory-by-example> and includes a Visual Studio 2019 SSIS project. To build and deploy the project, you will need a Windows machine, running Visual Studio 2019, with the SQL Server Integration Services Projects extension installed.

In this section, you will deploy the SSIS project to the Azure-SSIS IR. The deployment of SSIS packages to ADF requires a running Azure-SSIS IR to enable packages to be installed in the IR's catalog database.

1. Open the solution “Azure-SSIS-IntegrationRuntimeTest.sln” (found in the repository’s “SSIS\Azure-SSIS-IntegrationRuntimeTest” folder) in Visual Studio. The solution contains a single package called “TestPackage.dtsx” – open it.

2. The package contains two Execute SQL Tasks that call the database stored procedures [dbo].[LogPipelineStart] and [dbo].[LogPipelineEnd]. (The package has no real function other than to demonstrate that it can be executed within Azure Data Factory.) Both tasks use the “AzureSqlDatabase” connection manager defined in the package. Open the connection manager and update the fields *Server name*, *User name*, *Password*, and *Select or enter a database name* to specify your Azure SQL Database.
3. Click *Test Connection* to verify that you have configured the connection manager correctly, then click *OK* to save your changes. Build the project.
4. Right-click the project in Visual Studio’s solution explorer and click *Deploy* to launch the deployment wizard. Click *Next* to skip the introduction page (if displayed), then select the *SSIS in Azure Data Factory* deployment target. Click *Next*.
5. On the *Select Destination* page, check that *Authentication* is set to “SQL Server Authentication,” then set *Server name*, *Login*, and *Password* to specify your Azure SQL Database server. (This information is used to connect to the SSIS catalog database – the details only match those in step 2 because both databases happen to be hosted in the same server.) Click *Connect* to enable the *Path* field, then click *Browse*.
6. In the *Browse for Folder or Project* dialog, click *New folder...* to create a catalog folder with a name of your choice. Click *OK* to create the folder, ensure that it is selected, then click *OK* to close the dialog.
7. Click *Next*, then review your deployment configuration. When you are ready, click *Deploy* to deploy the project to the Azure-SSIS IR. When the deployment is complete, close the deployment wizard dialog.

Run an SSIS Package in ADF

SSIS packages are run in ADF using the *Execute SSIS package* pipeline activity. Create a new pipeline to run the package as follows:

1. Open the authoring workspace in the ADF UX and create a new pipeline. The Execute SSIS package activity is found in the activities toolbox's *General* group – drag one onto the authoring canvas.
2. On the activity's *Settings* configuration tab, choose your integration runtime from the *Azure-SSIS IR* dropdown. Click *Refresh* (to the right of the *Folder* dropdown) to load metadata from the catalog database. Select the *Folder* you created at SSIS project deployment, then choose *Project* “Azure-SSIS-IntegrationRuntimeTest” and *Package* “TestPackage.dtsx”. Figure 8-10 shows the correctly configured *Settings* tab.

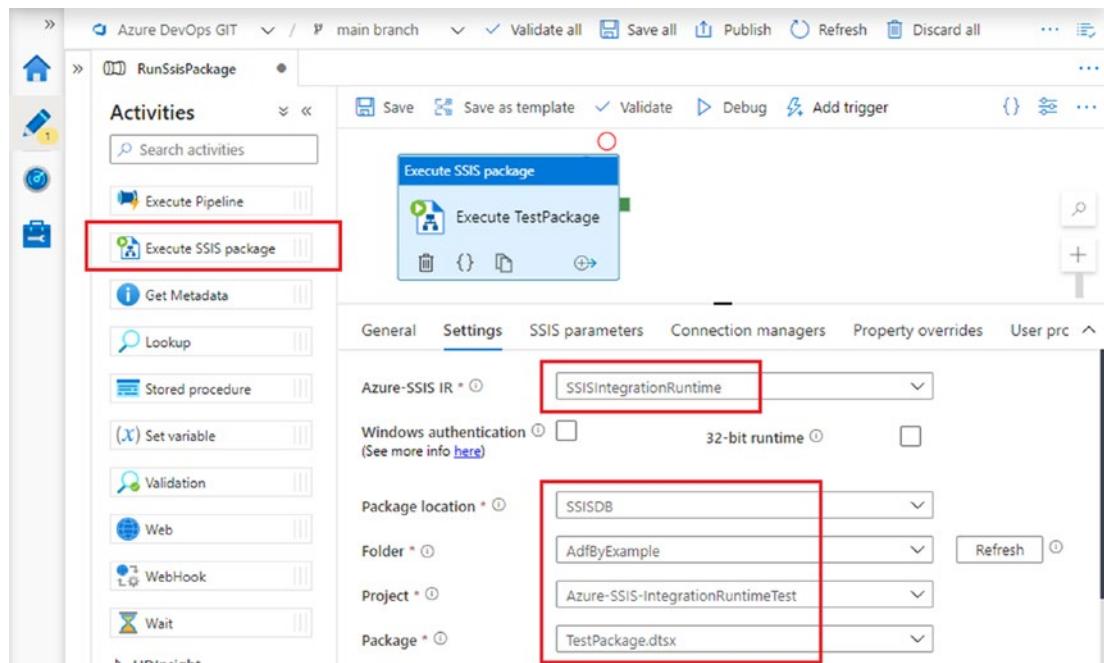


Figure 8-10. Execute SSIS package activity settings

3. The SSIS package specifies a “PipelineRunId” parameter value to pass into its call to [dbo].[LogPipelineStart]. The parameter is loaded automatically from package metadata and can be found on the *SSIS parameters* configuration tab. Replace its default value with a pipeline expression specifying the ADF pipeline run ID.
4. Save and run the pipeline. When execution has successfully completed, inspect the [dbo].[PipelineExecution] table for the log record created by the package – it contains the comment “Test execution from SSIS package.”

While SSIS packages can be executed in ADF, doing so is less convenient and less powerful than using native ADF capabilities. Limited SSIS support for Azure-specific resources (like blob or data lake storage) is available, but requires the additional installation of the Azure Feature Pack for Integration Services in Visual Studio. By default, the Azure-SSIS IR uses TLS 1.0 communication encryption, whereas Azure Storage uses the more secure TLS 1.2 protocol – this requires the additional custom setup of the IR or the explicit downgrading of a storage account’s TLS version. Using an Azure-SSIS IR to run existing SSIS packages in ADF can be a useful stepping stone into Azure, but undertaking new development directly in ADF allows you to benefit from cloud-first design features that are simply unavailable to SSIS.

Stop the Azure-SSIS IR

Because an Azure-SSIS IR consists of a real cluster of virtual machines, it incurs costs for any time it spends running, even when not executing SSIS packages. For this reason, you should start the IR only when SSIS package execution is required and stop it afterward. Stop the IR as follows:

1. Open the management hub’s *Integration runtimes* page and hover over your Azure-SSIS IR in the list of integration runtimes.
2. A set of controls appears to the right of the runtime’s name. The “pause” symbol (shown in Figure 8-11) in the set of controls is the IR’s stop button – click it to stop the IR.

The screenshot shows the 'Integration runtimes' page in the Azure DevOps interface. On the left, there's a sidebar with various navigation options like 'Connections', 'Linked services', 'Integration runtimes' (which is selected and highlighted in blue), 'Azure Purview (Preview)', 'Source control', 'Git configuration', 'ARM template', 'Parameterization template', 'Author', 'Triggers', 'Global parameters', 'Security', and 'Customer managed keys'. The main area has a heading 'Integration runtimes' and a sub-instruction: 'The integration runtime (IR) is the compute infrastructure to provide the following data integration capabilities across different network environment. Learn more'. Below this is a 'New' button and a 'Refresh' button. A 'Filter by name' input field is present. The table lists four items:

Name	Type	Sub-type	Status	Related
AutoResolveIntegrationRuntime	Azure	Public	Running	0
FarAwayIntegrationRuntime	Azure	Public	Running	0
RichardDesktop_LinkToSharedFactory	Self Hosted	Linked	Running	1
SSISIntegrationRuntime	Azure-SSIS	---	Running	0

The last row, 'SSISIntegrationRuntime', is highlighted with a red box. To its right is a 'Stop' button.

Figure 8-11. Stop control for the Azure-SSIS Integration Runtime

3. Click *Stop* in the confirmation dialog, then use the *Refresh* button above the list of runtimes to monitor the Azure-SSIS IR until its status is *Stopped*.

Tip You can start and stop an Azure-SSIS IR programmatically from an ADF pipeline by using the *Web activity* to call the integration runtime’s REST API endpoint – use this pattern to ensure that the IR runs only when required.

Chapter Review

This chapter introduced the concept of an integration runtime (IR). Every ADF instance contains at least one Azure IR, called “AutoResolveIntegrationRuntime,” which you have been using throughout the book – the chapter described other IR options and types available to support specific processing requirements.

Key Concepts

Key concepts encountered in the chapter include

- **External pipeline activity:** An ADF pipeline activity executed using compute resource provided by a service outside Azure Data Factory, for example, Stored procedure, Databricks, or HDInsight activities.
- **Internal pipeline activity:** An ADF pipeline activity executed using compute resource provided internally by Azure Data Factory.
- **Integration runtime:** Internal compute resource managed by Azure Data Factory.
- **Dispatching:** Management of ADF activity execution, particularly external pipeline activities.
- **Azure IR:** A fully managed, serverless integration runtime that executes data movements and transformations defined by the Copy data and Data flow activities. Azure IRs also manage dispatching of external activities to storage and compute environments like Azure blob storage, Azure SQL Database, and others.
- **AutoResolveIntegrationRuntime:** An Azure IR present in every data factory instance. The location of IR compute is determined automatically at runtime, and Databricks clusters created for Data flow activities using this IR have a TTL of zero – you can modify these characteristics by creating and using your own Azure IR.
- **Self-hosted integration runtime:** An IR installed on one or more servers provided by you in a private network. A self-hosted IR permits you to expose private resources to ADF, for example, source systems which are hosted on-premises or for which no native ADF connector is available.
- **Linked self-hosted IR:** A self-hosted IR is connected to exactly one Azure Data Factory instance. A common pattern used to enable access in other data factories is to share it, enabling other data factories to create linked self-hosted IRs that refer to the shared self-hosted IR.

- **Azure-SSIS IR:** A fully managed integration runtime that supports SSIS package execution in ADF. An Azure-SSIS IR consists of a VM cluster of a specified size and power – although the cluster is managed for you, its infrastructure is more visible than in serverless Azure IRs.
- **Web activity:** ADF pipeline activity supporting calls to REST API endpoints.

For SSIS Developers

Prior to the introduction of the Azure-SSIS IR, the usual approach to running SSIS packages in Azure was to host an instance of SQL Server Integration Services on an Azure VM. Replacing this IaaS approach with the Azure-SSIS IR in ADF removes the burden of infrastructure maintenance and provides more flexible scale-up and scale-out capabilities, by allowing you to control the size and power of the underlying VM cluster.

The existence of the Azure-SSIS IR makes it possible to expedite migration of existing ETL workloads into Azure with PaaS support, but SSIS packages are unable to exploit many of the features available to ADF. SSIS package development and maintenance for Azure requires additional tooling and cannot benefit from ADF's rich library of external system connectors or automatic on-demand resource scaling.

CHAPTER 9

Power Query in ADF

ADF data flows, covered in Chapter 7, model a data integration process as a stream of data rows that undergoes a succession of transformations to produce an output dataset. This approach to conceptualizing ETL operations is long established and may be familiar from other tools, including SQL Server Integration Services. While powerful, this view of a process can be inconvenient – when a new, unknown source dataset is being evaluated and understood, for example, or for users new to data engineering.

Power Query is a data preparation tool which takes a *wrangling* approach to data exploration, allowing you to interact directly with your data as you design and build the transformations you need. Power Query transformations, or *mashups*, are defined using M formula language expressions, built automatically by the visual Power Query Editor. The tool is available in a number of Microsoft products – it may be familiar to you from Microsoft Excel, Power Platform dataflows, or Power BI. Azure Data Factory includes built-in support for Power Query mashups using the *Power Query activity*.

Power Query mashups implemented in ADF benefit from at-scale distributed execution on a managed Azure Databricks cluster. A mashup's M script is translated at runtime into Data Flow Script, enabling execution in the same way as the data flows you implemented directly in Chapter 7. At the time of writing, the Power Query activity is in public preview and does not support the full range of M functions or ADF linked service connectors. A richer feature set is expected when the activity reaches general availability – the purpose of this chapter is to prepare you for that by demonstrating the approach and its potential.

Create a Power Query Mashup

In this section, you will create a Power Query mashup to wrangle the Handy Candy data you saved in Parquet format in Chapter 3.

1. In the ADF UX, create a “Chapter9” folder in the *Power Query* section of the *Factory Resources* sidebar. On the new folder’s *Actions* menu, click *New power query*.
2. The *New power query* blade opens, as shown in Figure 9-1. Enter a name for the new mashup. A *Source dataset* entry is already present but requires configuration – expand the *Dataset* dropdown list and click *+ New* to create a new one.
3. On the *New dataset* blade, select the *Azure Blob Storage* data store and click *Continue*. The *Select format* page follows – choose the *Parquet* file format, then click *Continue*.
4. Name the new dataset, then select your original blob storage *Linked service*. Note that, at the time of writing, the Power Query activity is unable to use linked services that access key vault secrets.
5. When you have selected your linked service, the usual *File path* fields are shown. Use the browse button to their right to find and select the “HandyCandy.parquet” folder (located in the “datalake” folder of the storage account’s “output” container). Click *OK* to close the dataset properties blade.
6. Click *OK* to create the mashup and launch the Power Query Editor.

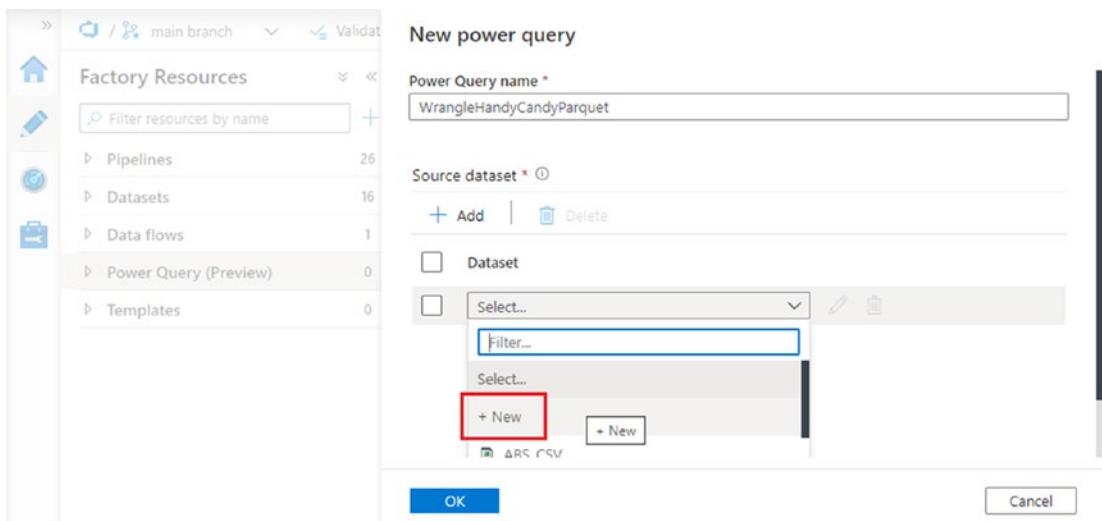


Figure 9-1. The New power query blade

Explore the Power Query Editor

The Power Query Editor opens inside the ADF UX – the editor may be familiar to you if you have previously used Power Query in other Microsoft products and services. Above the Power Query Editor is the standard authoring canvas toolbar containing *Save*, *Code*, and *Properties* buttons, along with a *Data flow debug* toggle. A cluster is not required during mashup development (unlike in the case of data flows), but you will need one later in order to run the Power Query activity in an ADF pipeline.

The newly opened editor is shown in Figure 9-2. At the top of the editor is a warning that “not all Power Query M functions are supported for data wrangling.” During public preview, unsupported functions can still be selected from the Power Query Editor and will even display correct results in the editor’s data preview pane, but they are not valid in ADF because the required M expression translation is not yet available. When you select an unsupported function, a “UserQuery : Expression.Error” warning appears, followed by a description of the error.

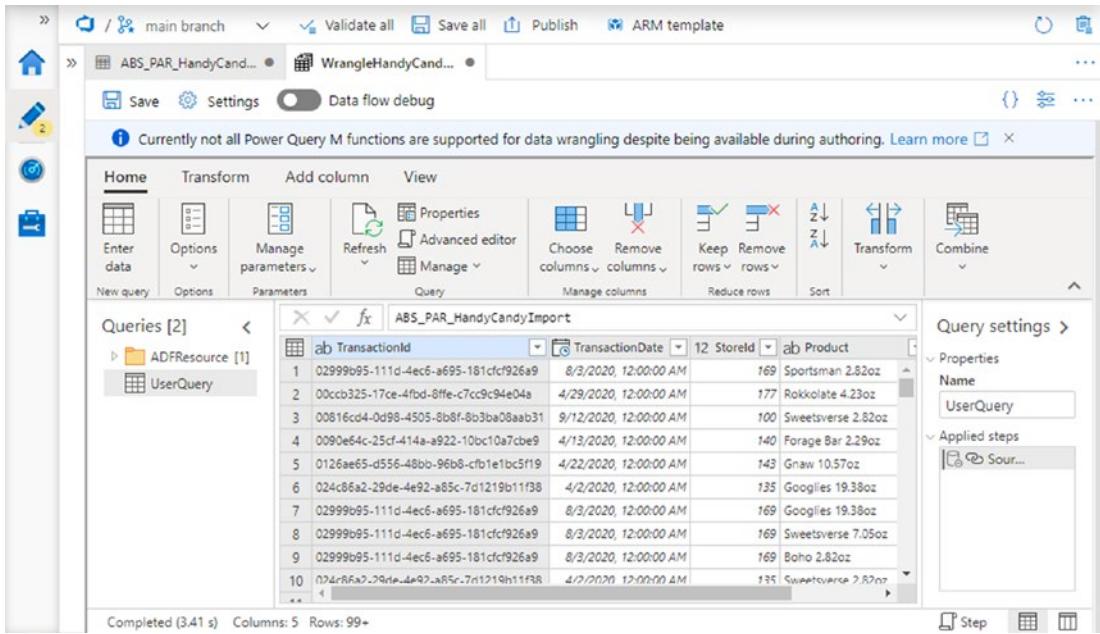


Figure 9-2. Power Query Editor opened for data wrangling

Immediately below the warning message (which you can close if you wish) is the command ribbon with four tabs.

- Dataset editing features on the *Home* tab include column and row removal.
- The *Transform* tab provides operations to make modifications to the dataset as is.
- The *Add column* tab provides operations to define additional columns.

Many column definition features appear on both the *Transform* and *Add column* tabs, differing only in their being used to modify existing columns or to define new ones.

Below the command ribbon, the data preview pane is displayed. To its right, the *Query settings* pane displays a history of modifications applied to the dataset. (The detail of the modification selected in the *Query settings* pane is displayed in the step script bar at the top of the data preview pane.) To the left of the data preview, the *Queries* pane contains a query called “UserQuery” and a folder containing queries representing the mashup’s source dataset(s). You are unable to interact with queries usefully here, so may prefer to collapse this pane. Before doing so, make sure that “UserQuery” is selected – it

should be highlighted in gray, causing the name of the underlying dataset to appear in the step script bar, as shown in Figure 9-2.

Wrangle Data

The original Handy Candy data consists of a set of sales transaction JSON messages. Each message contains a transaction ID and the details of one or more items included in the transaction. The Handy Candy Parquet file provides a denormalized view of the same data: each row represents a single item, accompanied by the ID of the transaction to which it belongs.

Unlike Handy Candy, the majority of confectionery retailer data is supplied to ABC in a pre-aggregated format. In this section, you will wrangle the Handy Candy data into this structure.

1. Begin by removing columns not required for the output. Right-click the “TransactionId” column header in the data preview pane and select *Remove columns* from the popup menu. Repeat this step for the “StoreId” column.
2. Other retailers’ data identifies sales month using the date of the first day of each month. Select the “TransactionDate” column by clicking its header, then open the *Add column* ribbon tab. The data type of the column is Date, so controls that do not apply to values of that type are disabled in the ribbon.
3. In the *Date and time column* group, open the *Date* dropdown and select *Start of month* from the *Month* group, as shown in Figure 9-3. A new column called “Start of month” appears in the data preview pane, containing the first day of the month in which the corresponding transaction date appears. However, Power Query’s *Date.StartOfMonth* function is not yet supported in ADF, causing an error message to be displayed as described earlier – this function cannot be used.

Note A list of supported wrangling functions is available at <https://docs.microsoft.com/en-us/azure/data-factory/wrangling-functions>.

The screenshot shows the Microsoft Power Query Editor interface. The ribbon at the top has 'Home', 'Transform', 'Add column' (which is currently selected), and 'View'. Below the ribbon is a toolbar with various icons for data manipulation. On the left, there's a sidebar titled 'Queries [2]' showing two items: 'WrangleHandyCand...' and 'WrangleHandyCand...'. The main area displays a table with columns: TransactionDate, Product, and Price. A context menu is open over the 'TransactionDate' column, listing options like 'Month', 'Start of month' (which is highlighted with a red box), 'End of month', 'Days in month', 'Name of month', and others. The 'Date' button in the ribbon is also highlighted with a red box. The status bar at the bottom indicates 'Columns: 3 Rows: 99+'.

Figure 9-3. Adding start of month based on the transaction date

4. Notice that the *Query settings* pane now includes three *Applied steps*. Remove the last of these – “Inserted start of month” – by selecting the step in the list, then clicking the cross button to its right. You can remove any step except “Source” from the list in this way, but removing a step from the middle of the list may invalidate later steps.
5. On the *Add column* tab’s ribbon, choose *Custom column*. In the custom column dialog, name the new column “EndOfPreviousMonth” and set *Custom column formula* to `Date.AddDays([TransactionDate], -Date.Day([TransactionDate]))`. The correctly configured formula is shown in Figure 9-4. Click *OK* to create the custom column.

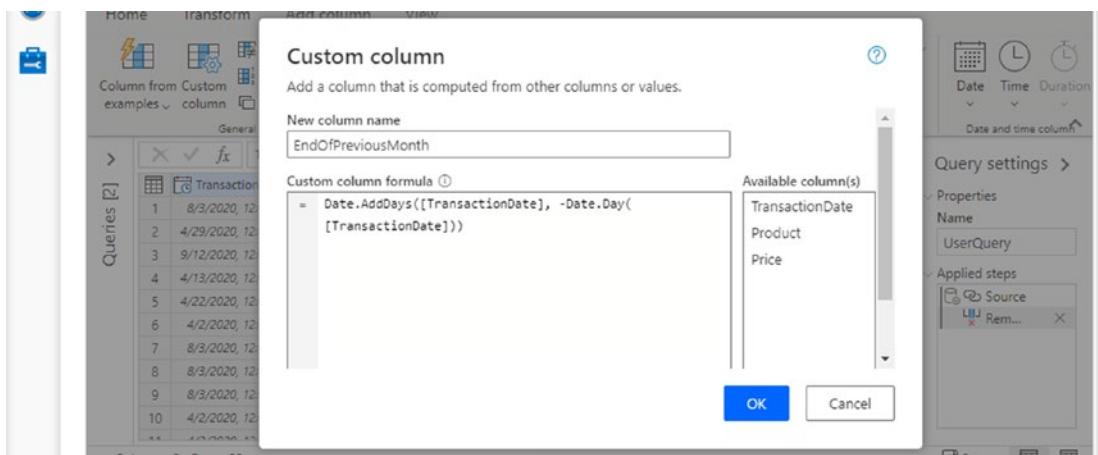


Figure 9-4. Add an EndOfPreviousMonth column

6. The date value required here is the start of the transaction month, not the end of the previous one, but at the time of writing, the formula `Date.AddDays([TransactionDate], 1 -Date.Day([TransactionDate]))` is also unsupported. Instead, use the formula `Date.AddDays([EndOfPreviousMonth], 1)` to add one day to the “EndOfPreviousMonth” value, in another custom column called “SalesMonth.”
7. Remove the “TransactionDate” and “EndOfPreviousMonth” columns as they are no longer required. Now create a third custom column called “ItemCount,” setting its value simply to the numeric value 1. Until row counts are supported for the Power Query activity, summing this column will enable you to count items.
8. You’re now ready to aggregate transaction items into monthly sales data. Select the *Transform* tab, then click *Group by* in the ribbon. The “Basic” setting allows you to specify one grouping field and one aggregate function – select “Advanced” to be able to add more than one of each. Group by the “SalesMonth” and “Product” columns, then sum the “Price” and “ItemCount” columns to produce new columns called “SalesValueUSD” and “UnitsSold,” respectively. The correctly completed dialog is shown in Figure 9-5.

- Finally, add another custom column called “Retailer,” with the constant value "Handy Candy" – include the quotation marks around the retailer name to indicate that this is a string literal.

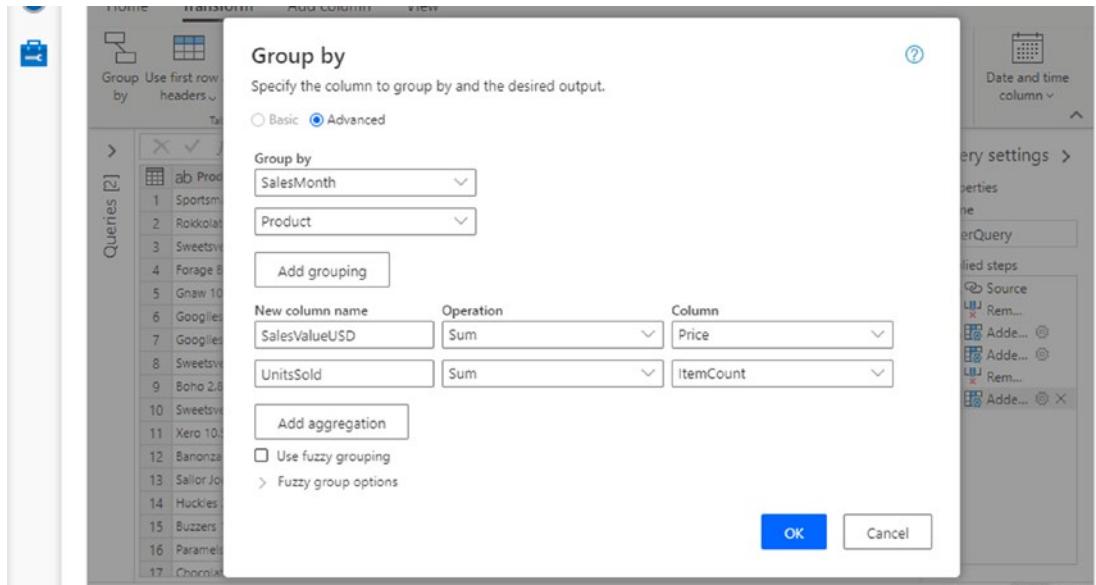


Figure 9-5. Aggregation of Price and ItemCount

Save your Power Query mashup and source dataset before proceeding. If you wish to inspect the M expression that the Power Query Editor has built for you, click the *Advanced editor* button (found in the *Query* group on the *Home* ribbon tab).

Run the Power Query Activity

Like data flows, Power Query mashups are executed inside an ADF pipeline by a dedicated pipeline activity – in this section, you will create a pipeline using the Power Query activity.

- Create a new pipeline (in a new “Chapter9” pipelines folder) and name it appropriately.
- Drag a *Power Query* activity from the activities toolbox’s *Power Query* group and onto the pipeline authoring canvas. Name the

activity, then use its *Settings* configuration tab to choose the Power Query you created in the previous section.

3. The Power Query mashup specifies a source of data, but no sink for transformation output. The sink must be specified in the Power Query activity's *Sink* tab – choose the Azure SQL DB dataset representing the database table [dbo].[Sales_LOAD].
4. Click *Debug* to run the pipeline from the ADF UX in the usual way. If a debug cluster is not already running, you will be warned that one is required – you can either turn on debugging (provisioning a debug cluster with a TTL of one hour) or choose *Use activity runtime* from the *Debug* dropdown menu.
5. After pipeline execution is complete, verify that rows have been loaded into the database table [dbo].[Sales_LOAD]. In addition to [RowId], the columns [Retailer], [SalesMonth], [Product], [SalesValueUSD], and [UnitsSold] should have been populated. If this is not the case, check that the names of the columns added to your mashup exactly match those in the database table – the Power Query activity uses column names to map its output to sink columns automatically.

This example pipeline omits activities to log pipeline start and end or to manage pipeline run IDs – the reason for this is that Power Query activities do not yet support the injection of external values. The *Parameters* group on the Power Query Editor *Home* tab refers to Power Query parameters, which are not supported in the ADF public preview.

As in the case of ADF's Data flow activity, the *Details* button (eyeglasses icon) in the Power Query activity run (displayed on the *Output* tab of the pipeline's configuration pane) opens a graphical monitoring view of the activity's execution on a Databricks cluster. In this case, each of the transformations corresponds to one of the steps applied in the Power Query wrangling process. The name of each transformation is allocated automatically, as can be seen in Figure 9-6.

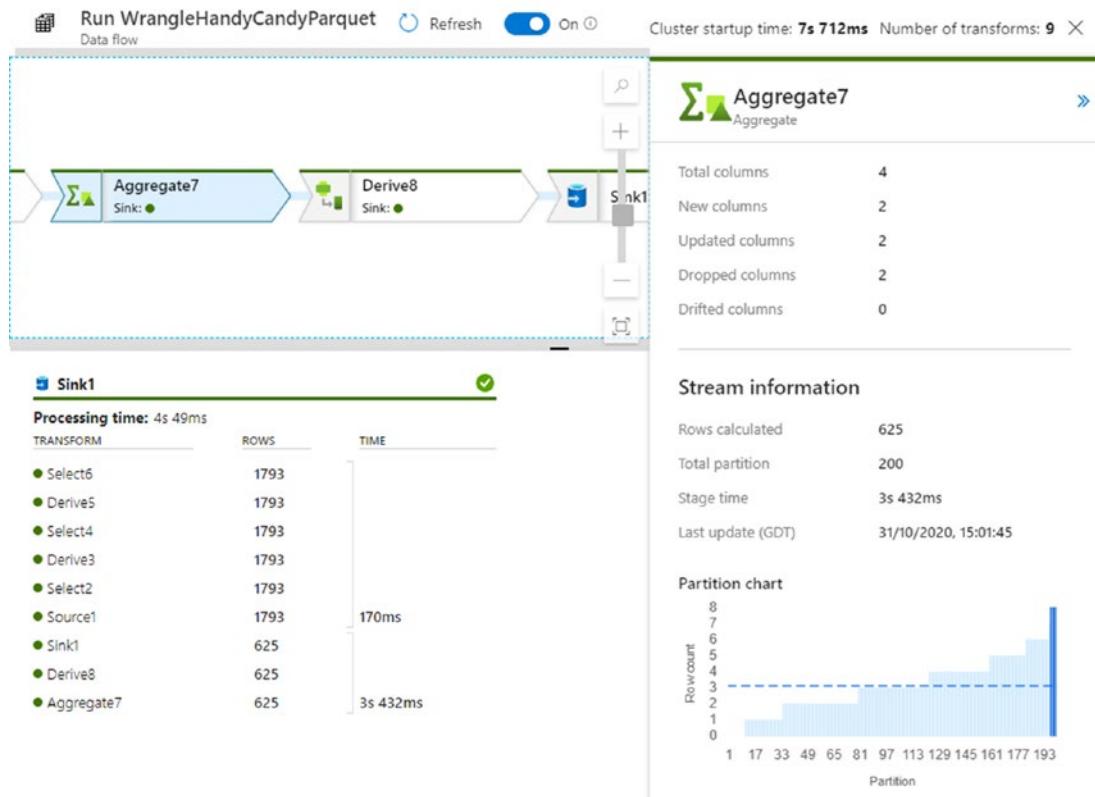


Figure 9-6. Power Query activity execution graphical monitoring view

Chapter Review

When generally available, Power Query mashups in ADF will make wrangling an attractive choice for exploratory transformation of new datasets. The approach aims to offer an intuitive, data-first alternative to implementing data preparation and transformation. This allows users from outside the field of data engineering – sometimes referred to as *citizen integrators* – to benefit simultaneously from the direct data interaction offered by the Power Query Editor and the at-scale operationalization of data pipelines running in Databricks.

At the time of writing, the Power Query activity remains in public preview and lacks support for important ADF features like parameters, key vault secret-based authentication, and the full breadth of available linked service connectors. M formula language support is also incomplete, but the accessibility of the mashup editor will make Power Query an increasingly popular tool as feature support increases.

Key concepts introduced in this chapter include

- **Power Query:** Graphical data preparation tool available in a number of Microsoft products, including ADF Power Query activities, Excel, Power Platform dataflows, or Power BI.
- **Data wrangling:** Interactive exploration and preparation of datasets.
- **Mashup:** Data wrangling transformation implemented in Power Query.
- **M formula language:** Language used to express transformations built in Power Query. M expressions built using the graphical Power Query Editor are translated into Data Flow Script at runtime by Azure Data Factory, for execution in the same way as an ADF data flow. M language support remains incomplete while the Power Query activity is in public preview – a list of supported functions is available at <https://docs.microsoft.com/en-us/azure/data-factory/wrangling-functions>.
- **Power Query activity:** ADF pipeline activity used to execute Power Query mashups implemented in the ADF UX.

CHAPTER 10

Publishing to ADF

The relationship between the ADF UX, Git, and your development data factory, first introduced in Chapter 1, is shown in Figure 10-1. In Chapters 2 to 9, you have been authoring factory resources in the ADF UX, saving them to the Git repository linked to your development factory, and running them in Debug mode using the development factory's compute (integration runtimes). Those interactions are shown in Figure 10-1 as dashed arrows.

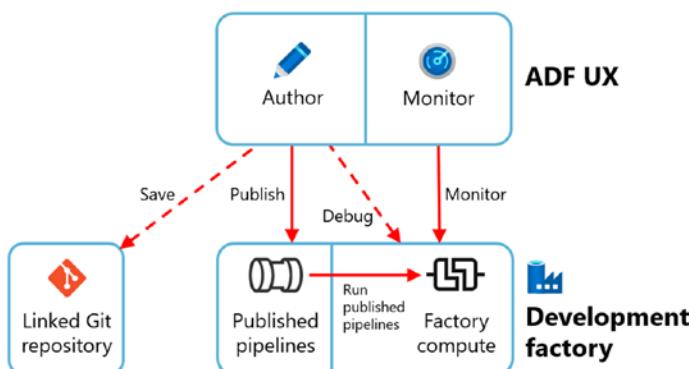


Figure 10-1. Azure Data Factory instance, the ADF UX and Git

The ultimate goal of your development work is to run pipelines automatically in a production environment. The interactions indicated in Figure 10-1 by solid arrows are described here and in the remaining two chapters:

- This chapter describes *publishing* to ADF, both from the ADF UX and from Git.
- Chapter 11 introduces *triggers*, used to run published pipelines automatically.
- Chapter 12 addresses the topic of *monitoring*.

The first step toward execution in production is to deploy – or *publish* – factory resources. This chapter introduces a variety of approaches to doing so.

Publish to Your Factory Instance

In this section, you will publish factory resources to your development ADF instance. Published pipelines can be executed independently, outside the context of an ADF UX session, although for the time being you will continue to execute them manually.

Trigger a Pipeline from the ADF UX

Published pipelines are typically executed automatically, either on a schedule or in response to an event like the creation of new input files in blob storage. Patterns of automatic execution are defined and configured in ADF using *triggers*, the subject of Chapter 11. By extension, starting a pipeline execution is often referred to as “triggering” a pipeline – the ADF UX supports the manual triggering of pipelines that have been published to its connected factory instance.

As Figure 10-1 suggests, published pipeline definitions are separate both from the versions stored in Git and any unsaved changes on the ADF UX authoring canvas and may differ. To illustrate this point, try to trigger one of your development pipelines that has not yet been published. (If you have already been experimenting with publishing and have no unpublished resources, create and save a new pipeline containing a single *Wait* activity, and use that one instead.)

1. Open the pipeline of your choice in the ADF UX authoring canvas.
2. Above the authoring canvas, to the right of the *Debug* button, find the *Add trigger* dropdown (shown in Figure 10-2). If you can’t see it, look on the overflow dropdown accessed by clicking the ellipsis (...) button.
3. From the *Add trigger* dropdown, select *Trigger now*. Unlike *Debug* (which runs the pipeline in your ADF UX session’s debugging environment), *Trigger now* executes the pipeline in the factory’s published environment.

The *Pipeline run* blade appears, but the *OK* button at the bottom is grayed out, and the message “Pipeline is not found, please publish first” is displayed – you are unable to trigger a pipeline run because the pipeline does not exist in the published environment. The pipelines in your ADF UX session are those loaded from your Git repository working branch and may not match the pipelines (or versions of pipelines) already published.

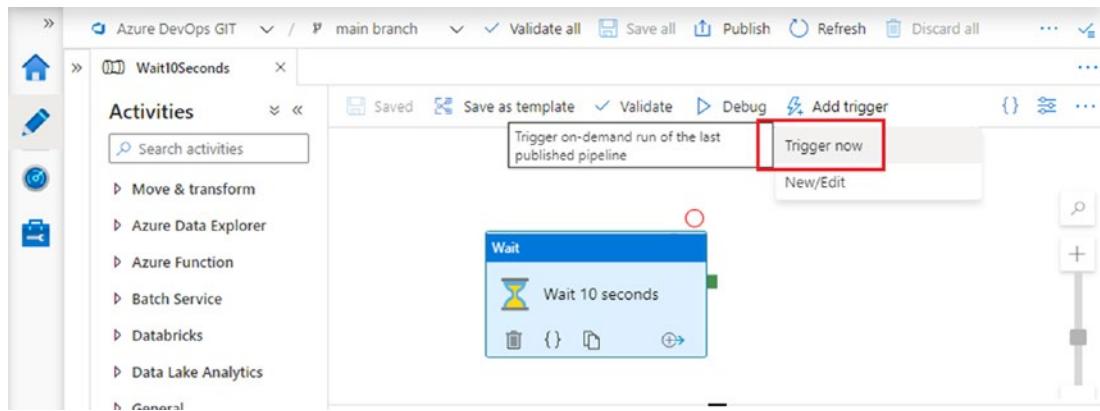


Figure 10-2. Trigger now button

Publish Factory Resources

Deploy resources to the factory’s published environment as follows:

1. Click the *Publish* button (visible in Figure 10-2, above the authoring canvas’s *Debug* button). If you have unsaved resources, the ADF UX prompts for confirmation, warning you that publishing will save all pending changes.

Note Publishing changes is only permitted from a factory’s collaboration branch. The *Publish* button is disabled when working in other branches.

2. The message “Gathering your new changes from collaboration branch” is displayed, before the *Pending changes* blade opens.

The blade lists changes that will be published into the data factory and identifies the factory’s *publish branch*. This is a nominated Git branch, by default `adf_publish`, specified in the ADF instance’s Git configuration. Click *OK* to publish the identified changes.

A series of messages is displayed as publishing proceeds, first indicating that resources have been published to the factory, then confirming that ARM templates have been saved into the Git repository’s publish branch. The purpose of these ARM templates is described later in the chapter.

3. Use the *Trigger now* button to run your chosen pipeline again.

Now that the pipeline has been published, the *OK* button on the *Pipeline run* blade is available. Click it to run the pipeline.

As the pipeline starts, a “Running” message is displayed in the ADF UX, replaced by a “Succeeded” message when pipeline execution has successfully completed.

Note The previous section demonstrated that factory resources visible in the ADF UX may not match those in the published environment. To inspect published resources in a Git-enabled factory, expand the Git dropdown in the top left of the ADF UX, then select “Switch to live mode.”

Inspect Published Pipeline Run Outcome

Unlike when running pipelines using *Debug*, activity execution information is not displayed on the pipeline’s *Output* configuration tab. Output from published pipelines is accessed instead using the ADF UX’s monitoring experience (Figure 10-3).

Chapter 12 examines the monitoring experience in greater detail. For now, open the monitoring experience by clicking the *Monitor* button (gauge icon) in the navigation sidebar. Published pipeline runs are shown on the *Triggered* tab of the *Pipeline runs* page.

Pipeline name	Run start	Run end	Status
Wait10Seconds	2/6/21, 8:59:07 PM	2/6/21, 8:59:19 PM	✓ Succeeded

Figure 10-3. ADF UX monitoring experience

Publish to Another Data Factory

A recommended model of ADF development and deployment is to build pipelines and other factory resources using a dedicated, Git-enabled development factory instance (as you have been doing). After development, resources are published into a separate production data factory, perhaps via one or more testing or other preproduction environments.

A number of approaches to deploying into other environments are available. In this section, you will use the Azure portal to publish resources manually into another factory.

Prepare a Production Environment

In a real multi-environment scenario, each environment typically has one or more data factories of its own, along with corresponding external resources such as key vaults, storage accounts, SQL databases, and others. For the purpose of this exercise, your “production environment” will consist simply of a data factory instance. As would be the case in a real multi-environment situation, the production factory will gain access to shared self-hosted integration runtimes using your shared data factory.

At the end of this section, you will have three ADF instances:

- *Development* refers to the factory you have been working in throughout.
- *Production* is the data factory instance to which you will be publishing resources from development.

- *Shared* is the shared data factory you created in Chapter 8. It will be used by the two other factories, as described in that chapter.

Create the Production Factory

Start by creating the production data factory, into which you will deploy the ADF resources you have already developed. You will create the factory in the same way that you did in Chapter 1, but with one important difference: by design, it will not be Git-enabled.

1. In the Azure portal, use the *Create a resource* button to start the creation of a new data factory.
2. Complete the *Basics* tab, providing a unique name for the new factory and specifying other details to match your development factory. (This is just for convenience here – it is common practice to segregate resources belonging to different environments into separate resource groups or subscriptions.)
3. On the *Git configuration* tab, tick the *Configure Git later* checkbox. From here, skip straight to *Review + create*, then click *Create* to create the new factory instance.

The Git repository attached to your development instance is the authoritative store of factory resource definitions. These definitions are copied to other ADF instances (including production) by the deployment process and should be treated as read-only in those data factories – under normal circumstances, you should not modify them directly.

Grant Access to the Self-Hosted Integration Runtime

In Chapter 8, you created a self-hosted integration runtime in a shared data factory to enable it to be reused by multiple other factories. Another important reason for doing so is that – because the self-hosted IR service is connected to exactly one data factory – you cannot publish self-hosted integration runtimes from one environment into another.

By creating a shared data factory, you have avoided this common deployment problem. To ensure that the development factory's link to the self-hosted IR can be published successfully, the production factory must also be granted access to the shared self-hosted IR.

1. Connect the ADF UX to your shared data factory, then open the management hub. Select the *Integration runtimes* page.
2. Open the *Edit integration runtime* blade by clicking the name of the self-hosted IR.
3. Select the *Sharing* tab, then click + *Grant permission to another Data Factory*. Your production factory should be visible immediately below the search field – if not, use search to locate it.
4. Tick the checkbox next to the production factory, then click *Add*. Click *Apply* to save your changes and close the blade.

Export ARM Template from Your Development Factory

Resources for publishing are described in *Azure Resource Manager (ARM) templates*. An ARM template is a set of one or more JSON files that defines Azure resources, in this case, your data factory. An Azure Data Factory ARM template describes all the resources defined in a factory – everything from global parameters to pipelines. You can obtain an ADF ARM template by exporting one using the ADF UX.

1. Open the ADF UX management hub in your development factory, then open the *Global parameters* page. Ensure that the *Include in ARM template* checkbox is ticked.
2. Open the *ARM template* page, then select the *Export ARM template* tile to export resource definitions from your ADF UX session. A zip archive containing template files is downloaded to your computer.
3. Unzip the downloaded archive file.

The file `arm_template.json` in the root of the zip archive describes every resource inside the ADF instance, as defined in your ADF UX session – it reflects the state of resources in your session, saved or not. The file specifies a number of deployment parameters, default values for which are stored in the file `arm_template_parameters.json`. The files in subfolders represent various subsets of resource information, useful in other situations. In this example, you will simply use the main `arm_template.json` file.

Import ARM Template into Your Production Factory

You can copy your exported factory resource definitions into another data factory instance by importing the ARM template.

1. Connect the ADF UX to your production data factory, then open its management hub.
2. Open the *ARM template* page, then select the *Import ARM template* tile to import the ARM template you downloaded from your development data factory. This launches the Azure portal's *Custom deployment* blade in a separate browser tab.
3. On the *Custom deployment* blade's *Select a template* tab, click *Build your own template in the editor*. The *Edit template* blade appears.
4. Click the *Load file* button (indicated in Figure 10-4), then browse to your unzipped ARM template archive. Select the file `arm_template.json` to open it in the template editor. Click *Save* – the portal returns you to the *Custom deployment* blade's *Basics* tab.
5. The *Basics* tab includes fields that identify the deployment target (resource group, region, factory name) and fields for each deployment parameter. Select the resource group containing your production factory and specify its name.
6. Nonsensitive deployment parameters will already be prepopulated with default values from your development factory. Review these, noticing that parameterized values include your key vault URL, connection strings, and any global parameters you have created. Global parameters used to identify environment-specific features (e.g., an environment identifier) should be updated here.
7. You will need to provide values for sensitive parameters yourself – these include your storage account and SQL database connection strings (found in their respective Azure portal resource blades) and the password for the SSISDB login. Click *Review + create*. Review your deployment settings, then click *Create* to initiate deployment.

8. When the deployment has successfully completed, return to the ADF UX in your production factory. Click *Refresh* in the factory header bar to load the published resource definitions – when using a factory that is not Git-enabled, the ADF UX is loaded with resource definitions from the published environment.

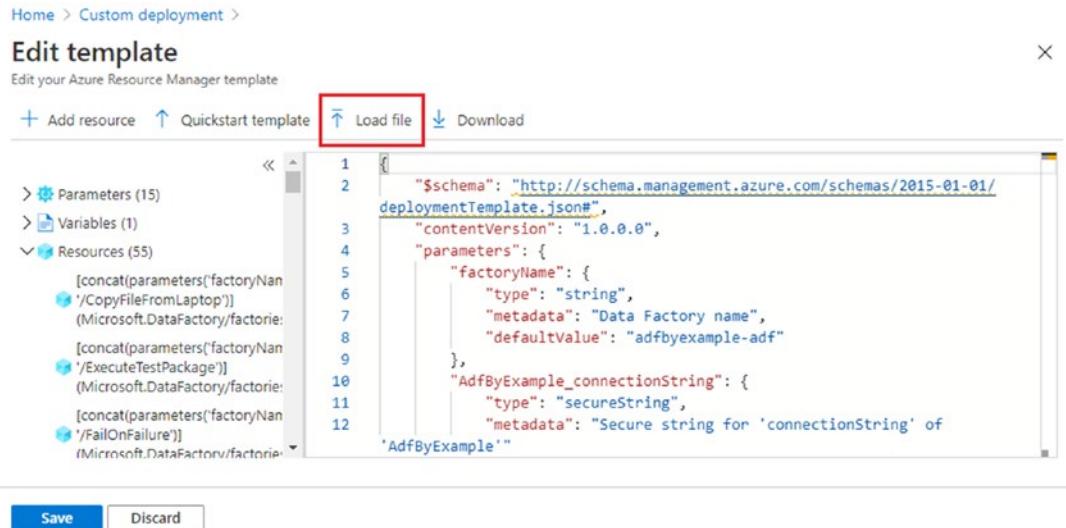


Figure 10-4. ARM template editor during custom deployment from the Azure portal

You can make and publish changes to your production environment directly from a connected ADF UX session, but this is not advisable. Changes made in this way are not saved to your Git repository – they will not be included in future deployments from your development environment and may be overwritten. You should consider restricting write access to nondevelopment factories, to prevent direct modification under all but exceptional circumstances (e.g., making a manual hotfix which is later also added to Git).

Tip No standard Azure Data Factory security role provides read-only factory access – to achieve this, create an *Azure custom role* containing read permissions to data factory resources.

Understand Deployment Parameters

While importing an ADF ARM template into a data factory instance, the Azure portal prompts you to supply values for a number of deployment parameters. The purpose of these is to enable you to supply environment-specific values for particular factory resources when deploying into the corresponding environment. In the deployment example of the previous section, you reused parameter values from your development environment to specify connection information for your development key vault, storage account, and SQL database linked services. In a real multi-environment scenario, where each environment contains its own corresponding external resources, the values of such parameters must be updated to match the properties of the target environment.

The selection of factory properties to be parameterized in an ADF ARM template is determined by the factory's *parameterization template*, which can be edited via the *ARM template* page of the ADF UX management hub. A simple pattern for managing parameters in a multi-environment scenario is as follows:

- Create a separate key vault for each environment.
- Store environment-specific secrets like connection strings and access keys in the key vault.
- Implement nonsecret environment-specific variables as global parameters.

Listing 10-1 shows a simplified parameterization template, suitable for use in this pattern, which specifies that only global parameters and key vault URLs are to be parameterized. Note that for global parameters to be included in an ARM template, the *Include in ARM template* checkbox must be ticked on the ADF UX management hub's *Global parameters* page.

Listing 10-1. Simplified parameterization template

```
{
  "Microsoft.DataFactory/factories": {
    "properties": {
      "globalParameters": {
        "*": {
          "value": "="
        }
      }
    }
  }
}
```

```
        }
    },
},
"Microsoft.DataFactory/factories/linkedServices": {
    "AzureKeyVault": {
        "properties": {
            "typeProperties": {
                "baseUrl": "="
            }
        }
    }
}
}
```

Automate Publishing to Another Factory

Publishing to your development factory instance from the ADF UX has two effects, as you saw earlier in the chapter:

- ADF pipelines are present and can be triggered in the published environment.
- Factory ARM templates are saved to your Git repository.

The ARM templates written to Git are named differently from those exported directly from the ADF UX (and include additional template decompositions) but can be used in exactly the same way. Furthermore, when coupled with Git and *Azure Pipelines*, you can arrange for an ARM template to be published to a different environment whenever it is updated.

Note The term “pipeline” is somewhat overloaded when discussing Azure Data Factory and Azure DevOps together. The remainder of this chapter deals almost exclusively with Azure Pipelines, which for clarity I refer to as Azure DevOps pipelines.

Azure Pipelines is the name of Microsoft’s cloud-based CI/CD pipeline service. CI/CD – continuous integration and continuous delivery – is a development practice in which software changes are integrated into the main code base and deployed into production continuously, as they are developed. An Azure DevOps pipeline specifies a series of tasks to be performed automatically in response to events in your Git repository – for example, when a change is made in a particular location or branch. In the following sections, you will create an Azure DevOps pipeline to deploy Azure Data Factory ARM templates automatically when they are updated in Git.

Create a DevOps Service Connection

In order to be permitted to modify Azure resources, an Azure DevOps pipeline uses a *service connection* – a nominated AAD principal with the necessary permissions. If you do not already have a pipeline service connection in your Azure DevOps project, create one as follows:

1. Open Azure DevOps and navigate to your AdfByExample project.
Select *Project settings*, found in the bottom left of the browser window.
2. In the *Pipelines* section of the *Project Settings* menu, select *Service connections*. Click the *Create service connection* button that appears in the main window.
3. In the *New service connection* dialog, select “Azure Resource Manager.” Click *Next*.
4. Choose the recommended authentication method “Service principal (automatic),” then click *Next*.
5. Ensure that *Scope level* is set to “Subscription” and that the correct subscription is selected in the *Subscription* dropdown. Provide a *Service connection name*, then click *Save*.

Saving the service connection automatically creates an associated AAD service principal with the necessary resource management permissions in your Azure subscription. You will now be able to use this service connection to authorize deployment pipeline actions.

Create an Azure DevOps Pipeline

Azure Data Factory's publish process writes ARM templates into a nominated publish branch in Git, by default `adf_publish` (although you can change this if you wish in the factory's Git configuration). A data factory's publish branch is not used like a conventional feature branch – it is never merged back into the collaboration branch, nor does it receive updates from the collaboration branch. Instead, it provides a stand-alone location to store published ARM templates – the branch only ever receives updates when changes are published in the development environment via the ADF UX. Azure Data Factory's use of the `adf_publish` branch specifically to contain published ARM templates means that updates to the branch can be used to trigger automated deployments.

Azure DevOps pipelines can be implemented in two different styles, either using a graphical user interface (“classic”) or with a definition file written in *YAML*, a *data serialization language*. The YAML style of pipeline is preferable because, as code, a pipeline’s definition can be stored under version control in your Git repository and subjected to the same review processes elsewhere in your development workflow.

Create a YAML Pipeline File

When an Azure DevOps pipeline is executed, it uses the YAML definition file found in the Git feature branch that triggered the execution. In the case of Azure Data Factory, this means that the YAML file must be created directly within the repository’s publish branch. In this section, you will create a simple pipeline YAML file.

1. In Azure DevOps, browse to the *Files* page of your Git repository. Use the branch dropdown to select the repository’s `adf_publish` branch. The branch contains a folder with the name of your development ADF instance – this contains the ARM template files created by the publish process.
2. Using the vertical ellipsis button to the right of the repository name (indicated in Figure 10-5), choose *+ New* and then *File*.

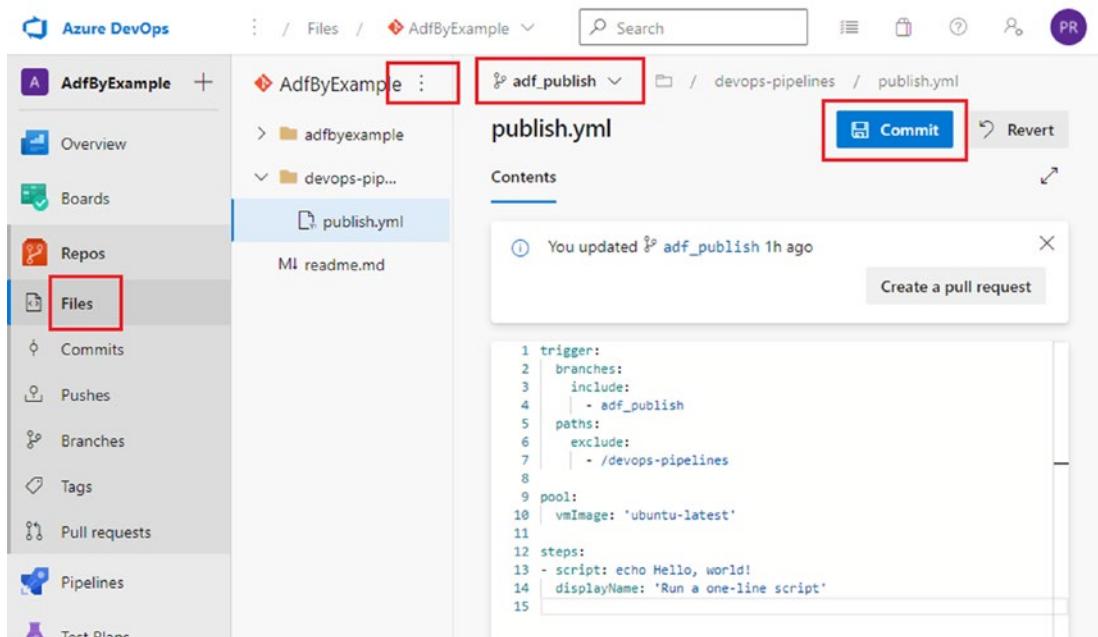


Figure 10-5. Creating publish.yml directly in adf_publish branch

3. Name the new file “devops-pipelines/publish.yml”, then click *Create*. An empty file editor pane opens on the right-hand side.
4. Add the text given in Listing 10-2 to the editor pane. YAML uses text indentation to group elements together, so make sure that you indent the text exactly as in the listing. A copy of the “publish.yml” file is available in the book’s GitHub repository.
5. Click *Commit* to save the new file directly into the adf_publish branch. (You will be prompted for a commit comment – you can accept the default by clicking *Commit* again in the comment dialog.)

Listing 10-2. Basic YAML pipeline definition

```

trigger:
branches:
  include:
    - adf_publish
paths:

```

```

exclude:
- /devops-pipelines

pool:
vmImage: 'ubuntu-latest'

steps:
- script: echo Hello, world!
  displayName: 'Run a one-line script'

```

Create an Azure DevOps Pipeline Using the YAML File

The YAML file you have created contains a pipeline definition, but on its own is not an Azure DevOps pipeline. In this section, you will create an Azure DevOps pipeline that uses the definition stored in your YAML pipeline file.

1. In Azure DevOps, select the *Pipelines* item in the sidebar.
2. If this is your first Azure DevOps pipeline, a *Create Pipeline* button is displayed in the center of the screen – click it. (If you don't see it, use the *New pipeline* button in the top right of the screen.) A pipeline creation wizard launches.
3. On the *Connect* tab (headed *Where is your code?*), select *Azure Repos Git*. On the *Select* tab which follows, choose your factory Git repository.
4. On the *Configure* tab, select *Existing Azure Pipelines YAML file*. In the dialog box that appears, select your `adf_publish` branch, then choose “/devops-pipelines/publish.yml” from the *Path* dropdown.
5. Click *Continue*, then on the *Review your pipeline YAML* page, click *Run* to create and run the pipeline.

The Azure DevOps pipeline you have created does nothing more than print “Hello, world!” to the console, but it will now do so automatically every time that the `adf_publish` branch is updated – whenever you publish changes from the ADF UX in your development environment.

Add the Factory Deployment Task

Azure DevOps pipelines are made up of *tasks*, configurable processes that perform activities required for deployment. The `script` task in your YAML pipeline is one example of a task. In this section, you will add an ARM template deployment task – the new task will make changes to resources in Azure, authorized using the service connection you created earlier.

1. Select the *Pipelines* item from the Azure DevOps sidebar again, then select your new pipeline from the list of recently run pipelines. You can edit the pipeline's YAML from here – click *Edit* to open the YAML pipeline editor.
2. Verify that the branch shown at the top of the editor pane is `adf_publish`, then add the ARM template deployment task given in Listing 10-3 to the bottom of the YAML file. (You may remove the “Hello, world!” script task from earlier if you wish.)
3. Configure the YAML task, replacing the placeholders in angle brackets with appropriate values. “<your-service-connection-name>” is the name of the service connection you created earlier in the chapter, and “<your-production-factory-location>” is the Azure region where your production data factory is deployed. Figure 10-6 shows the task configured for my development and production factory instances.
4. Save the pipeline, providing a commit message when prompted, and run it again. (If you receive an error message indicating that the YAML file could not be found, make sure that you have selected the `adf_publish` branch.)

Listing 10-3. ARM template deployment task YAML for ADF

```
- task: AzureResourceManagerTemplateDeployment@3
  inputs:
    deploymentScope: 'Resource Group'
    azureResourceManagerConnection: '<your-service-connection-name>'
    subscriptionId: '<your-azure-subscription-id>'
    action: 'Create Or Update Resource Group'
```

```

resourceGroupName: '<your-resource-group-name>'
location: '<your-production-factory-location>'
templateLocation: 'Linked artifact'
csmFile: '<your-development-factory-name>/ARMTemplateForFactory.json'
csmParametersFile: '<your-development-factory-name>/
ARMTemplateParametersForFactory.json'
overrideParameters: |
  -factoryName "<your-production-factory-name>"
deploymentMode: 'Incremental'

```

Note The Incremental deployment mode specified in Listing 10-3 is necessary because deployment is scoped at the resource group level – deploying a factory ARM template in Complete mode would cause other resources outside the data factory to be deleted.

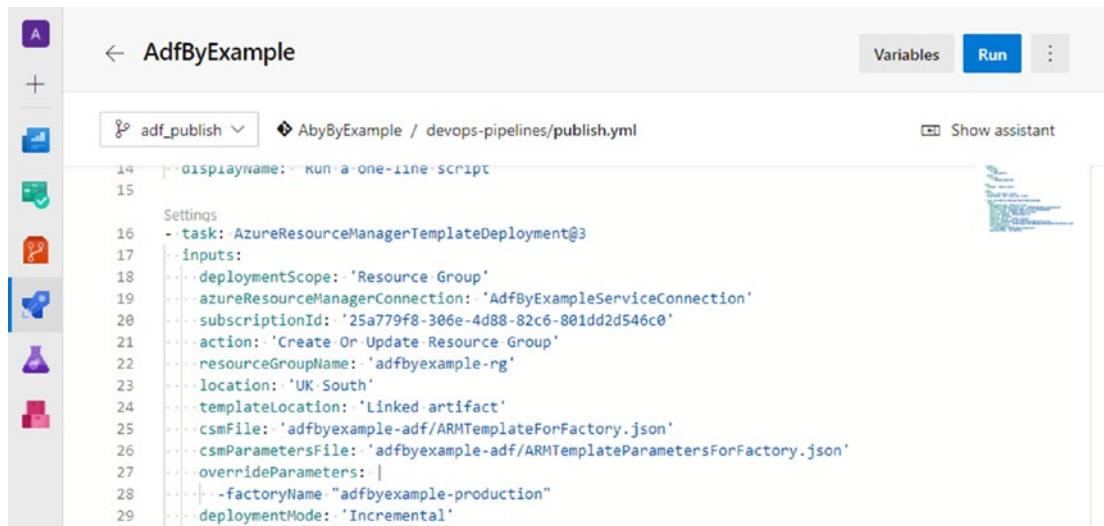


Figure 10-6. Configured ARM template deployment task in YAML

The deployment task's `overrideParameters` argument allows you to specify values for use when the task is executed. Listing 10-3 specifies a single override parameter – the name of the deployment's target factory – but you can specify other parameters in the same way. These might include the target environment's key vault URL or values for factory global parameters.

Avoid specifying secret values directly in the `overrideParameters` argument, as this will cause them to be stored in your Git repository in plain text. Instead, create pipeline variables using the *Variables* button visible in the pipeline editor. When creating a new variable, tick the *Keep this value secret* checkbox to prevent its value from being displayed anywhere, then use the variable in the YAML task, for example:

```
overrideParameters: |
  -factoryName "adfbyexample-production"
  -myKeyVaultUrl "https://adfbyexample-kv.vault.azure.net/"
  -mySecretParam "$(SecretVariableName)"
```

Trigger an Automatic Deployment

The Azure DevOps pipeline you have created deploys an ARM template automatically from the Git repository's `adf_publish` branch into your production factory, whenever you publish changes in your development factory. Test this by making a change to your development factory and publishing it.

1. Open the ADF UX authoring workspace in your development factory, then create a “Chapter10” pipelines folder.
2. Create a new Azure Data Factory pipeline by cloning an existing one from an earlier chapter. Rename the new ADF pipeline and move it to the “Chapter10” folder.
3. Click *Save all* to save your changes to your Git collaboration branch in the usual way.
4. Click *Publish* to include your changes in the ARM templates stored in the factory's Git `adf_publish` branch. This causes your Azure DevOps pipeline to be triggered.

5. Return to Azure DevOps, then navigate to your pipeline definition in the *Pipelines* list. Click your pipeline to view the list of recent runs. Each run indicates how it was triggered – “manually triggered” runs are those that you started directly while developing your Azure DevOps pipeline, while those triggered automatically when publishing from the ADF UX are described as “Individual CI.” Figure 10-7 shows a history containing both manually triggered and individual CI pipeline runs.
6. When execution of the Azure DevOps pipeline is complete, open the ADF UX authoring workspace in your production factory and verify that the factory now also contains the ADF pipeline you created in step 2.

Description	Stages	
#20201122.6 ARM template and parameters deployed on 11-22-2...	✓	15m ago 27s
#20201122.5 Update publish.yml for Azure Pipelines	✓	2h ago 26s
#20201122.4 Update publish.yml for Azure Pipelines	✓	2h ago 26s
#20201122.3 Update publish.yml for Azure Pipelines		7h ago

Figure 10-7. Azure DevOps pipeline run history

The Azure DevOps pipeline you have developed here is almost completely automatic – when development work is ready for deployment, clicking *Publish* in the ADF UX is the only additional action required to promote your work into the production environment. In this example, I referred to the target environment as “production,” but DevOps pipelines are frequently used to deploy development work through a sequence of environments, enabling development work to be tested in a variety of ways before being delivered into production.

Figure 10-8 illustrates the relationship between the ADF UX, Git, and your development and production ADF instances. In this scenario, the role of the ADF UX depends on the data factory to which it is connected – in development it is predominantly an authoring tool, while in production its purpose is almost exclusively factory monitoring.

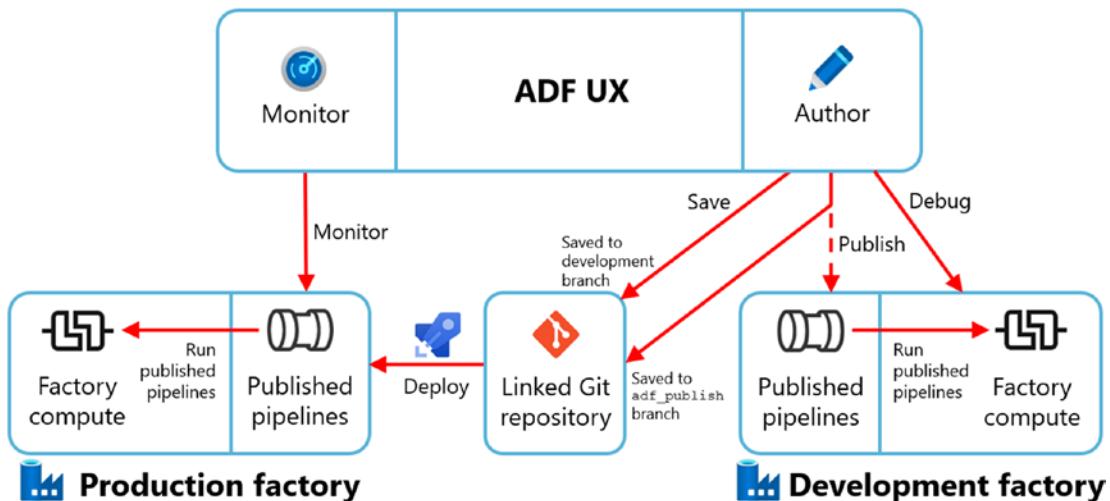


Figure 10-8. The role of the ADF UX in different factory instances

Clicking *Publish* in the ADF UX publishes resources to the development factory instance (indicated by a dashed arrow in the diagram), but this is rarely of interest – publishing to the development factory at all is a relic of single-factory working practices which predate ADF’s integration with Git. The primary purpose of *Publish* is to save ARM templates to the Git repository’s publish branch, triggering the automatic deployment pipeline.

Feature Branch Workflow

Throughout this book, you have been working with a Git-enabled development factory, saving changes directly into the factory’s collaboration branch, `main`. One of the reasons for Git’s popularity as a version control system is its support for feature branches, allowing different developers to work on separate areas in isolation. Typically, a *feature branch workflow* involves creating a new branch to contain work on a specific new feature – for example, in the case of ADF, creating or modifying a data factory pipeline. The ADF UX supports feature branch workflows via the Git dropdown (shown in Figure 10-9).

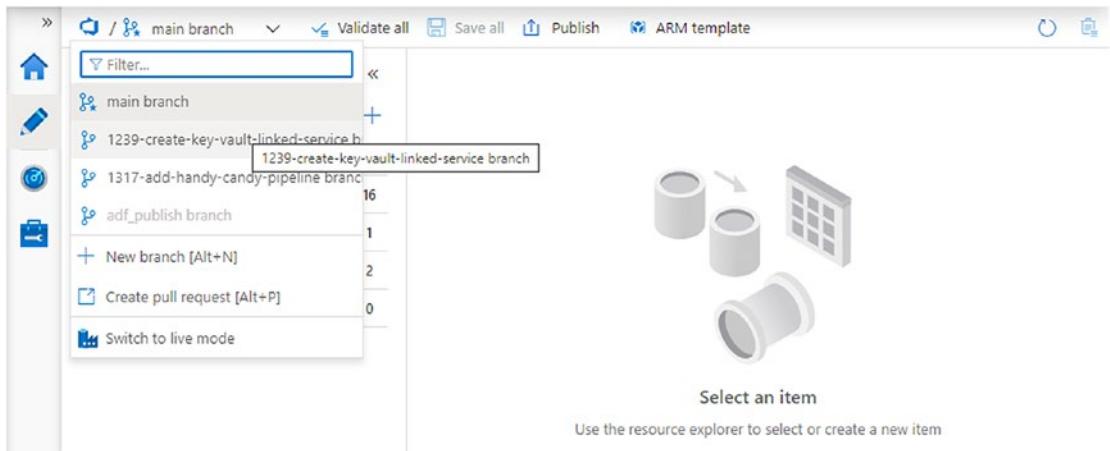


Figure 10-9. Feature branch selection in the ADF UX

A frequent choice when using a feature branch workflow is to prevent any direct modification of the collaboration branch. Instead, when development work in a feature branch is complete, that branch is merged into `main` via a *pull request*. A pull request enables other developers to review and approve the work in the feature branch, as part of wider quality assurance measures for new development. The ADF UX itself contributes to enforcing this workflow, by only permitting changes to be published from the collaboration branch.

In a feature branch workflow, merging changes to `main` on approval of a pull request is a natural trigger for automated deployment. In contrast, triggering a deployment pipeline built on the `adf_publish` branch requires additional manual intervention – to update ARM templates in the publish branch, you must publish resources interactively using the ADF UX.

The following section reviews approaches to aligning ADF deployment and Git feature branch workflows. Recall that although Azure Data Factory resources are represented in the `adf_publish` branch as one or more ARM template files, they are stored in `main` and other repository branches in per-resource JSON files. Using these JSON files as the basis for deployment enables resources to be deployed automatically, using an Azure DevOps pipeline triggered from the `main` branch.

Azure Data Factory Utilities

A recent innovation provided by Microsoft is the *Azure Data Factory Utilities* npm package, available from www.npmjs.com/package/@microsoft/azure-data-factory-utilities. The package provides two functions previously available only within the ADF UX: validation of factory resources and generation of ARM templates. The package operates on factory resource JSON files stored on the same computer – you can clone your data factory's Git repository and use the package locally, or you can download and install the package in your CI/CD pipeline.

A possible deployment workflow in this scenario is as follows:

1. A data engineer completes work on a feature branch, opening a pull request in the factory's Git repository.
2. After approval, the pull request is merged to the factory's collaboration branch, usually `main`.
3. Merging to `main` triggers an Azure DevOps pipeline.
4. The Azure DevOps pipeline downloads and installs Node.js and the Azure Data Factory Utilities npm package.
5. The Azure DevOps pipeline uses the npm package to validate factory resources and to generate an ARM template for deployment.
6. The Azure DevOps pipeline deploys the generated ARM template to the target environment, in the same way as your pipeline earlier in the chapter.

This approach eliminates the manual *Publish* step required earlier, triggering deployment from an action taken as part of the feature branch workflow instead. The removal of this step also avoids publishing resources unnecessarily to the development factory.

At the time of writing, the construction of a CI/CD pipeline as described here is more complicated than is desirable. It is to be expected that, as adoption of the npm package increases, further development of the tool will make it easier and simpler to use.

Publish Resources as JSON

Using ARM templates to automate Azure Data Factory CI/CD workflows is Microsoft's recommended approach, but has the limitation that the only unit of deployment is the factory itself – you are unable to deploy subsets of factory resources. If you wish to deploy factory resources individually, you can do so by publishing resources directly from their JSON definitions. This approach integrates naturally into a feature branch workflow, because no additional deployment format – such as an ARM template – is required.

Deploy ADF Pipelines Using PowerShell

Azure Data Factory resources can be created and deployed in a number of different ways. This book focuses on using the interactive ADF UX, but in addition to supporting ARM templates, ADF provides a REST API and libraries for .NET, Python, and PowerShell. PowerShell is well supported in Azure DevOps pipelines, making it a convenient alternative tool for automated factory deployment.

In this section, you will build a DevOps pipeline that deploys data factory pipelines to production when changes are made to the main branch, using PowerShell cmdlets from the `Az.DataFactory` module. Factory resources deployed using PowerShell, or other libraries or APIs, appear directly in the target factory's published environment – the debugging environment is a feature of the ADF UX alone.

A new, empty pipeline can be created directly using the Azure DevOps pipeline creation wizard.

1. In Azure DevOps, select the *Pipelines* item in the sidebar. Use the *New pipeline* button in the top right of the screen to create a new Azure DevOps pipeline.
2. As before, under *Where is your code?*, select *Azure Repos Git*, then choose your factory Git repository on the *Select* tab which follows.
3. On the *Configure* tab, select *Starter pipeline*. A new skeleton Azure DevOps pipeline file opens in the pipeline editor.
4. The Azure DevOps pipeline file path is displayed and can be edited immediately below the *Review your pipeline YAML* header. Change the default filename to “`devops-pipelines/publish-json.yml`”.

5. Replace the entire contents of the pipeline editor window with the YAML given in Listing 10-4 (available in the book's GitHub repository).
6. The code in the listing uses the DevOps AzurePowerShell@4 task to iterate over ADF pipeline JSON files, deploying each one using the Az.DataFactory PowerShell cmdlet Set-AzDataFactoryV2Pipeline. Configure the task, replacing the placeholders in angle brackets with appropriate values as before.
7. Save the pipeline, providing an appropriate Git commit message and choosing the option *Commit directly to the main branch*.
8. Use the ADF UX to create or modify an ADF pipeline in your development factory, then verify that saving changes causes the Azure DevOps pipeline to run. Verify that your changes subsequently appear in your production data factory.

Listing 10-4. Deploy ADF pipelines using PowerShell in a DevOps pipeline

```
trigger:  
  branches:  
    include:  
      - main  
  paths:  
    exclude:  
      - /devops-pipelines  
  
pool:  
  vmImage: 'ubuntu-latest'  
  
steps:  
- task: AzurePowerShell@4  
  inputs:  
    azureSubscription: '<your-service-connection-name>'  
    azurePowerShellVersion: latestVersion  
    scriptType: inlineScript  
    inline: |  
      foreach($file in Get-ChildItem "data-factory-resources/pipeline") {
```

```

Write-Host "Deploying $($file.Basename)"
Set-AzDataFactoryV2Pipeline -Force ` 
    -ResourceGroupName "<your-resource-group-name>" ` 
    -DataFactoryName "<your-production-factory-name>" ` 
    -Name "$($file.Basename)" ` 
    -File "$($file.FullName)"
}

```

Note The purpose of this example is to illustrate that a change to `main` can trigger a deployment pipeline that requires no ARM templates to be generated. Deployment from `main` is convenient in a feature branch workflow, but would be most undesirable in a workflow that allows data engineers to commit directly to the collaboration branch (in the way you have done here).

Resource Dependencies

The prior example illustrates the deployment of ADF pipelines using PowerShell, but a full implementation needs also to manage the deployment of other resource types. A corresponding PowerShell `Set-AzDataFactoryV2...` cmdlet is available for each type of resource:

- `Set-AzDataFactoryV2IntegrationRuntime`: Deploy an integration runtime
- `Set-AzDataFactoryV2LinkedService`: Deploy a linked service
- `Set-AzDataFactoryV2Dataset`: Deploy a dataset
- `Set-AzDataFactoryV2DataFlow`: Deploy a data flow
- `Set-AzDataFactoryV2`: Deploy or update a data factory, including global parameters

Using cmdlets allows you to deploy all or a subset of factory resources as you prefer, but in either case you must also account for dependencies between resources. For instance, attempting to deploy a dataset will fail if the linked service it uses does

not already exist. The example Azure DevOps pipeline in the previous section runs successfully, simply because the same factory has already been deployed, so all the resources required by each pipeline are already present.

You can avoid many dependency problems by deploying resource types in an order that respects common dependencies: linked services before datasets, for example, or datasets before pipelines. Resolving dependencies between resources of the same type is more complicated, but can be achieved in a variety of different ways. A simple approach is to catch individual cmdlet failures and to retry them after other resources have been successfully deployed; a more sophisticated approach is to use the JSON `referenceName` attribute in resource definitions to identify other required resources, allowing you to deploy them in advance.

Tip Neither the ARM template nor the JSON-based deployment approach supports deletion of removed resources. To identify resources for deletion, a comparison must be made between the set of JSON files in source control and the full list of factory resources in the published environment. A separate postdeployment PowerShell script, for use with ARM template deployments, is available at <https://docs.microsoft.com/en-us/azure/data-factory/continuous-integration-deployment>.

Chapter Review

This chapter introduced a variety of techniques for deploying developed factory resources into published factory environments, whether into the same (development) factory or into other factory instances.

- ADF resources are deployed from the development factory's collaboration branch into its published environment by clicking *Publish* in the ADF UX.
- Publishing to the development factory also generates corresponding ARM templates, storing them in the `adf_publish` branch of the factory's Git repository. These can be automatically deployed to other factories using an Azure DevOps pipeline triggered by changes to the publish branch.

- You can export ARM templates directly from your ADF UX session, which allows you to create deployments from other Git branches or even from unsaved changes. ARM templates can be imported into a data factory directly, also using the ADF UX.
- The Azure Data Factory Utilities npm package can be used to create ARM templates from a set of JSON factory resource definition files. This allows ARM template deployments to be triggered from updates to the factory's collaboration branch.
- PowerShell cmdlets in the Az.DataFactory module enable per-resource deployment strategies to be adopted. The approach also allows deployment from `main` but is more complicated to manage. A range of information and open source projects supporting JSON-based deployments can be found on the Internet.

Key concepts introduced in this chapter include

- **Azure Resource Manager (ARM) template:** An ARM template is a JSON file that defines components of an Azure solution, for example, the contents of an Azure Data Factory instance.
- **Publish:** To run a pipeline independently of the ADF UX, it must be deployed into a data factory's published environment. Published pipelines are executed using triggers (Chapter 11); published pipeline runs can be observed in the ADF UX monitoring experience (Chapter 12).
- **Publish branch:** A nominated branch in a factory's Git repository, by default `adf_publish`. The publish branch contains ARM templates produced when publishing factory resources in the ADF UX.
- **Azure custom role:** A custom security role, built by assembling a required set of permissions, to provide security profiles not supported in the standard Azure role set.
- **Deployment parameters:** Specified in an ARM template, deployment parameters enable different values to be substituted at deployment time. In the case of ADF, this permits a single template to be used for deployments to multiple different data factories.

- **Parameterization template:** A development data factory's parameterization template specifies which factory resource properties should be made parameterizable using deployment parameters.
- **CI/CD:** Continuous integration and continuous delivery (CI/CD) is a development practice in which software changes are integrated into the main code base and deployed into production continuously.
- **Azure Pipelines:** Microsoft's cloud-based CI/CD pipeline service.
- **Data serialization language:** Human-readable language used to represent data structures in text for storage or transmission. XML, JSON, and YAML are examples of data serialization languages.
- **YAML:** Data serialization language with an indentation-based layout, used in Azure DevOps to define Azure DevOps pipelines. YAML pipeline files are stored under version control like any other code file.
- **Task:** Configurable process used in an Azure DevOps pipeline, for example, `script`, `AzureResourceManagerTemplateDeployment@3`, or `AzurePowerShell@4`.
- **Pipeline variable:** Variable defined for use in an Azure DevOps pipeline. Secret variables allow secret values to be specified in pipeline YAML without storing them in version control.
- **Service connection:** Represents a nominated AAD principal with the permissions required by an Azure DevOps pipeline.
- **Feature branch workflow:** A common Git workflow in which development work takes place in isolated feature branches.
- **Pull request:** A request to merge a feature branch back into the collaboration branch when feature development is complete.
- **Az.DataFactory:** PowerShell module providing cmdlets for interacting with Azure Data Factory.

CHAPTER 11

Triggers

In Chapter 10, you explored how to deploy Azure Data Factory resources into published factory environments. You tested running one or more published pipelines by executing them manually from the ADF UX – in this chapter, you will explore how pipelines can be executed automatically using *triggers*.

A trigger defines conditions of different types for pipeline execution:

- A *schedule trigger* runs at configured dates and times.
- An *event-based trigger* runs when a file is created or deleted in Azure blob storage.
- A *tumbling window trigger* also runs at configured dates and times, but unlike a schedule trigger, it makes the interval (window) between successive executions explicit. A tumbling window trigger can pass interval start and end time values in pipeline parameters, allowing data processing to be time sliced.

A trigger is associated with zero or more data factory pipelines. When the conditions for a trigger's execution are met, it runs. When the trigger runs, it starts an execution of each of its associated pipelines.

Use a Schedule Trigger

In this section, you will create and use a schedule trigger to run pipelines at configured dates and times.

Create a Schedule Trigger

Triggers can be created and linked to pipelines in the ADF UX authoring workspace. Create a trigger for your Chapter 4 pipeline “ImportSweetTreats_RunSeqNo” as follows:

1. Open the pipeline in the ADF UX authoring workspace.
2. Expand the *Add trigger* dropdown and select *New/Edit*. The *Add triggers* blade appears.
3. Open the *Choose trigger...* dropdown and click *+ New* to open the *New trigger* blade.
4. Name the new trigger “RunEvery2Minutes” and ensure its *Type* is set to “Schedule.” Set *Time zone* to your local time zone. *Start date* is prepopulated with the current UTC time – the default value does not change as you adjust your time zone, so amend it if necessary.

Tip UTC times are used throughout the Azure platform, but a schedule in your local time zone is easier to read. A local time zone schedule permits ADF to adjust trigger start times automatically to align with daylight savings, where applicable.

5. Set *Recurrence* to “Every 2 Minute(s)” and ensure *Activated* is set to “Yes” – Figure 11-1 shows the completed blade. Click *OK* to save the trigger.
6. The *New trigger* blade now offers you the opportunity to set pipeline parameter values (if applicable) and reminds you that triggers do not take effect until they are published. Click *Save* – this commits your changes to Git, but does not publish them.

A schedule trigger runs for the first time at *Start date*, then again every time the interval specified in *Recurrence* has elapsed. If the trigger’s start date is in the past when the trigger is published, the first execution will take place at the next scheduled recurrence. This means that start date serves two purposes:

- It defines a date and time before which the trigger is not executed. Choosing *Specify an end date* allows you similarly to specify a date and time after which the trigger will no longer run.
- It provides a base date and time which, in combination with the recurrence interval, determines the date and time of subsequent executions.

For example, to schedule a trigger to execute nightly at 11 p.m., set the time component of its start date to 11 p.m. and set its recurrence interval to 1 day.

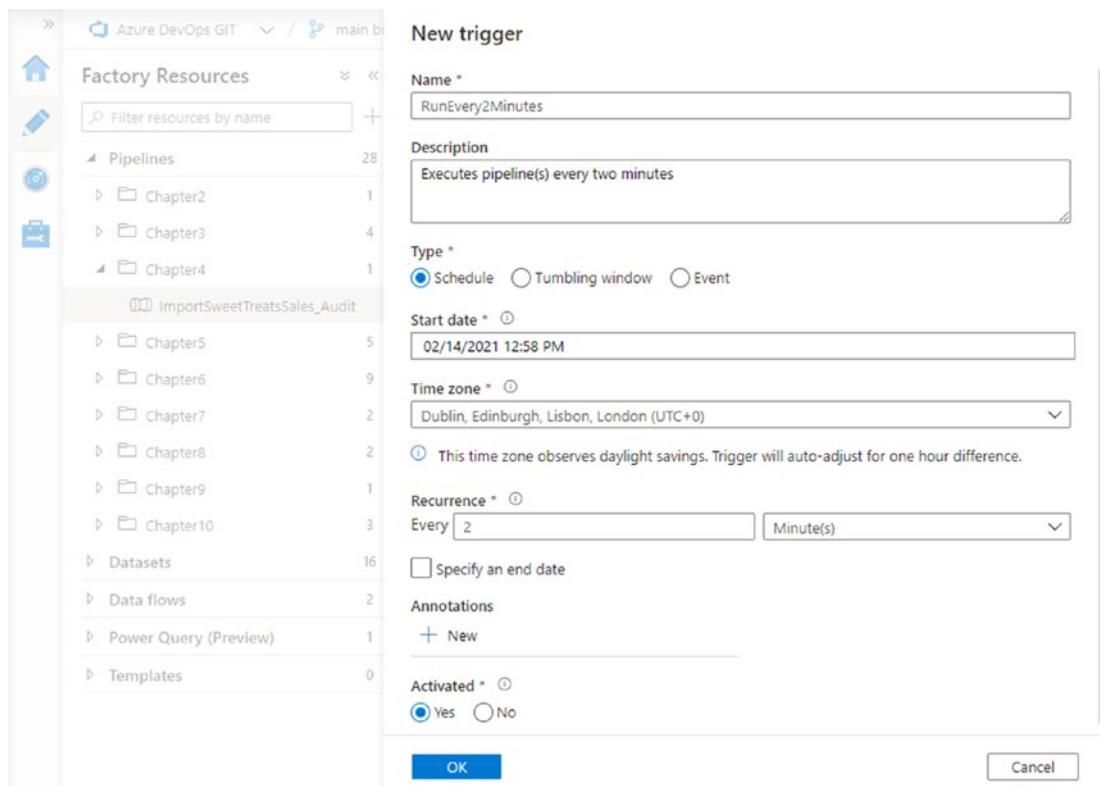


Figure 11-1. New schedule trigger configuration

Reuse a Trigger

Schedule triggers can be reused by multiple pipelines. In this section, you will configure another pipeline to make use of the “RunEvery2Minutes” trigger.

1. In the ADF UX authoring workspace, open the “ImportSTFormatFolder” pipeline you created in Chapter 5.
2. Expand the *Add trigger* dropdown and click *New/Edit*. The *Add triggers* blade appears.

3. Open the *Choose trigger...* dropdown and select “RunEvery2Minutes” from the list. The *Edit trigger* blade is displayed.
4. Click *OK* to accept the existing trigger definition. This pipeline requires two parameters: “WildcardFolderPath” and “WildcardFileName.” Provide the values “azure-data-factory-by-example-main/SampleData/Desserts4All” and “*.csv”, then click *Save*.

Whenever it runs, the “RunEvery2Minutes” trigger will now start two pipeline executions, one run of each linked pipeline.

Inspect Trigger Definitions

You can inspect all a factory’s trigger definitions in one place using the ADF UX management hub.

1. Open the management hub and select the *Triggers* page. The page contains a list of defined triggers, including the “RunEvery2Minutes” trigger you defined in the previous section, as shown in Figure 11-2. The number “2” in the list’s *Related* column indicates that the trigger is linked to two pipelines.
2. Hover over the trigger name to reveal additional controls. Changes to trigger definitions made using any of the three buttons are committed to Git immediately (without you explicitly choosing to save changes). Click the central *Code* button (braces icon) to inspect the trigger’s JSON definition.
3. The trigger definition includes a *pipelines* property: this is a JSON array which identifies the two pipelines linked to the trigger, including parameter values to be used at runtime. The *runtimeState* property value is “Started” – this corresponds to your selection of “Yes” to activate the trigger when you created it. Click *OK* to close the code window.

Name ↑↓	Type ↑↓	Status ↑↓	Related ↑↓
RunEvery2Minutes	Schedule	Started	2

Figure 11-2. Inspect triggers in the ADF UX management hub

As with other resources in a Git-enabled data factory, information displayed here is consistent with the contents of your working branch, not the published factory environment. In particular, the value displayed in the *Status* column does not indicate that the trigger is running in the published environment – “Started” is simply the value of the `runtimeState` property saved in Git.

You can create new triggers directly in the management hub, using the *+ New* button at the top of the page to open the *New trigger* blade. This approach creates a trigger in isolation, with no linked pipelines – to link a pipeline, you must add the trigger to the pipeline via the ADF UX authoring canvas.

Publish the Trigger

To bring the “RunEvery2Minutes” trigger into effect, it must be deployed into a published factory environment. For convenience, you will publish the trigger to your development factory. If the ARM template deployment Azure DevOps pipeline you built in Chapter 10 is still active, the trigger will also be deployed to your production data factory – this will not cause a problem in this case, because triggers created by ARM template deployments are not started automatically. More information about trigger deployment is provided at the end of this chapter.

Publish the trigger in the same way that you published other factory resources, by clicking *Publish* in the factory header bar.

Monitor Trigger Runs

After publishing the trigger to your data factory, executions start to occur automatically. Trigger runs, like published pipeline runs, can be monitored using the ADF UX monitoring experience.

- The *Trigger runs* page contains one record per trigger execution (irrespective of the number of pipeline executions started by the trigger).
- The *Pipeline runs* page contains records of individual pipeline runs and indicates the way in which each was triggered.

Figures 11-3 and 11-4 show the *Trigger runs* and *Pipeline runs* pages a few minutes after deploying the “RunEvery2Minutes” trigger.

Trigger name	Trigger type	Trigger time	Status
RunEvery2Minutes	ScheduleTrigger	2/14/21, 1:36:00 PM	✓ Succeeded
RunEvery2Minutes	ScheduleTrigger	2/14/21, 1:34:00 PM	✓ Succeeded

Figure 11-3. Runs of the RunEvery2Minutes trigger

The trigger’s start time and recurrence interval (as shown in Figure 11-1) define executions at 12:58, 13:00, 13:02, 13:04, and so on – its first execution at 13:34 indicates that the trigger was not published until after 13:32. Figure 11-4 shows that two pipeline runs were started by each trigger run, one for each of the trigger’s two linked pipelines. The pipeline runs identify “RunEvery2Minutes” as the trigger that started each one.

The screenshot shows the ADF UX Pipeline runs page. On the left, a sidebar menu includes options like Dashboards, Runs, Pipeline runs (which is selected), Trigger runs, Runtimes & sessions, Integration runtimes, Data flow debug, Notifications, and Alerts & metrics. The main area is titled 'Pipeline runs' and has tabs for Triggered (selected), Debug, Rerun, Cancel, Refresh, Edit columns, List, and Gantt. It features search and filter controls for Pipeline name (All), Status (All), and Local time (Last 24 hours). Below these are buttons for 'Add filter' and 'Copy filters'. A table lists five pipeline runs, each with a checkbox, Pipeline name, Run start, Triggered by, and Status. All runs are labeled 'ImportSTFormatFolder' and 'ImportSweetTreatsSales_Audit', triggered by 'RunEvery2Minutes', and have a status of 'Succeeded'.

<input type="checkbox"/> Pipeline name	Run start ↑↓	Triggered by	Status
<input type="checkbox"/> ImportSTFormatFolder	2/14/21, 1:36:00 PM	RunEvery2Minutes	✓ Succeeded
<input type="checkbox"/> ImportSweetTreatsSales_Audit	2/14/21, 1:36:00 PM	RunEvery2Minutes	✓ Succeeded
<input type="checkbox"/> ImportSTFormatFolder	2/14/21, 1:34:00 PM	RunEvery2Minutes	✓ Succeeded
<input type="checkbox"/> ImportSweetTreatsSales_Audit	2/14/21, 1:34:00 PM	RunEvery2Minutes	✓ Succeeded

Figure 11-4. Pipeline runs started by the RunEvery2Minutes trigger

Note When starting a pipeline run, a schedule trigger takes no account of whether or not the scheduled pipeline is currently running – it simply starts a new execution, overlapping any pipeline run still in progress.

Deactivate the Trigger

As the “RunEvery2Minutes” trigger has no configured end date, it will continue to run indefinitely, every two minutes. Deactivate the trigger to avoid incurring unnecessary pipeline execution costs.

1. Open the *Triggers* page in the ADF UX management hub.
2. Hover over the “RunEvery2Minutes” trigger name, then click the *Deactivate* (pause icon) button which appears. This changes the trigger’s configured runtime status and saves the revised definition to Git.
3. Publish your changes to deactivate the published trigger.

You may wish to monitor the *Trigger runs* and *Pipeline runs* pages in the ADF UX monitoring experience, to assure yourself that the trigger has been successfully deactivated.

Remember that in a nondevelopment environment (not Git-enabled), resource definitions shown in the ADF UX are always those present in the published environment. Changes to triggers must still be published to take effect, but with no linked Git repository, the status reported in the management hub for a published trigger is its true runtime status in the published environment.

Advanced Recurrence Options

Schedule triggers offer more advanced scheduling options when the frequency of recurrence is given in days, weeks, or months. These enable you to specify execution times explicitly, instead of implying them using a recurrence pattern. Using a frequency specified in weeks or months additionally permits you to nominate specific days of the week or month, respectively. Figure 11-5 shows a schedule trigger configured to run at 9:30 a.m. and 5:30 p.m. every Monday, Wednesday, and Friday. The *Hours* and *Minutes* fields are used to specify the execution times of day, which the ADF UX then presents as an easy-to-read list of resulting *Schedule execution times*.

In the same way that a single schedule trigger can be used by multiple pipelines, so a single pipeline can use multiple schedule triggers. Composing multiple triggers enables you to implement execution schedules that cannot be expressed in a single trigger. For example, no combination of hours and minutes in Figure 11-5 would cause executions at 9:15 a.m. and 5:45 p.m., but the effect can be achieved using two triggers, one for each execution time.

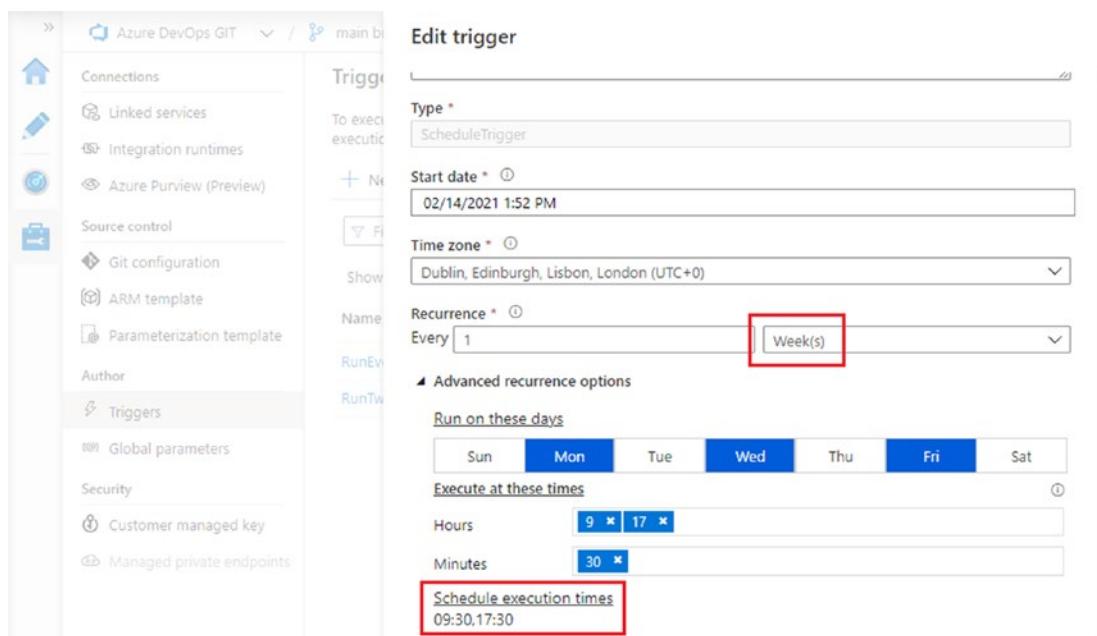


Figure 11-5. Schedule trigger using weekly recurrence pattern

For SSIS developers Time-based scheduling of SSIS package execution is commonly achieved using the SQL Server Agent. The advanced recurrence options offered by Azure Data Factory schedule triggers offer a similar degree of scheduling flexibility.

Use an Event-Based Trigger

In this section, you will create and use an event-based trigger to run a pipeline whenever a file is created in a given blob storage location.

Register the Event Grid Resource Provider

To work with any Azure resource, the corresponding *resource provider* must be registered to your Azure subscription. The blob storage file creation and deletion events that cause an event-based trigger to run are produced by another Azure service called *Azure Event Grid*. Before you can use an ADF event-based trigger, ensure that the Microsoft. EventGrid resource provider is registered to your Azure subscription.

1. Select “Subscriptions” from the list of Azure services displayed at the top of the Azure portal home page. (If you can’t see the “Subscriptions” tile, use the search box at the top of the page to find it.) On the *Subscriptions* blade, choose the subscription containing your data factory and other resources.
2. On the subscription blade, select *Resource providers* from the *Settings* section of the sidebar menu (shown in Figure 11-6).
3. Use the filter function to locate the “Microsoft.EventGrid” resource provider, then click its entry in the list to select it.
4. If the resource provider’s status is “NotRegistered,” click *Register* to add it to your subscription. You may need to refresh the portal screen to see that the registration has taken effect.

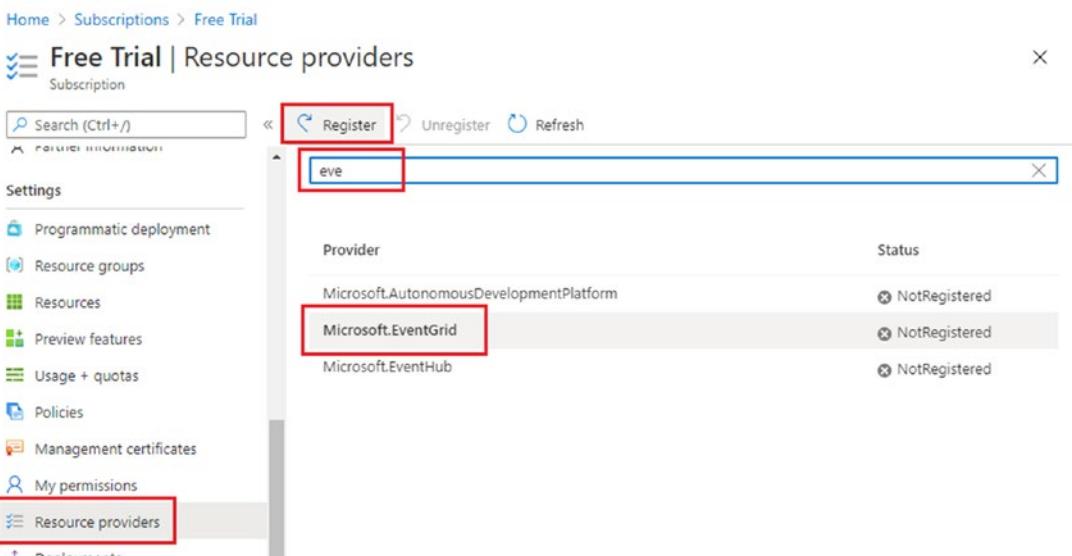


Figure 11-6. Locate and register the Microsoft.EventGrid resource provider

Required resource providers are often registered automatically, for example, when creating a resource of the corresponding type in the Azure portal. This is why you have not encountered the need to register a resource provider in earlier chapters.

Create an Event-Based Trigger

A typical use case for an event-based trigger is to run a pipeline to process files as soon as they arrive in blob storage. Create a trigger for the “ImportSTFormatFile” pipeline you created in Chapter 6, as follows:

1. Open the pipeline in the ADF UX authoring workspace and use the *Add trigger* dropdown to access the *Add triggers* blade. Select option + *New* from the *Choose trigger...* dropdown.
2. Name the new trigger “RunSTFormatImport” and set its *Type* to “Event.” The set of fields in the blade changes automatically to allow you to specify storage account details.
3. Use *Account selection method* “From Azure subscription,” then select your blob storage account from the *Storage account name* dropdown. Use the *Container* name dropdown to choose your “sampledata” container.
4. In the *Blob path begins with* field, enter “azure-data-factory-by-example-main/SampleData/JustOneMore/SalesData/”, then set *Blob path ends with* to “.txt”.

Note File/folder path wildcards are not supported here – entering “*.txt” will match no files.

This sample data folder contains sales data relating to another confectionery retailer named “Just One More.” This retailer also uses the Sweet Treats sales data format.

5. Under *Event*, tick the “Blob created” checkbox, then click *Continue*. (Figure 11-7 shows the correctly completed blade.)

6. The *Data preview* blade opens, allowing you to check your blob path pattern against any existing files that match. If you have entered the path pattern correctly, a single file called “Apr-2020.txt” will be matched. Click *Continue*.
7. The *New trigger* blade now offers you the opportunity to set pipeline parameter values. This pipeline uses two parameters – the file and folder name of the Sweet Treats format file to be loaded. The values to be used here identify the file whose creation causes the trigger to run. Enter the value “@triggerBody().fileName” in the *File* parameter value field.

Note @triggerBody() is an ADF expression, but pipeline parameter fields on the *New trigger* blade are not currently supported by the ADF expression builder. You can use other functions from the expression language here, but you are unable to access the clickable functions list or benefit from expression validation.

8. The pipeline’s *Directory* parameter represents a folder path inside a blob storage container, but the trigger body’s folder path property includes the blob storage container’s name. Set the parameter value to “@replace(triggerBody().FolderPath, ‘sampledata/’, ”)”, removing the container name component. Click *Save* to create the new trigger definition and commit it to Git. Deploy the new trigger to the factory’s published environment by clicking *Publish* in the factory header bar.

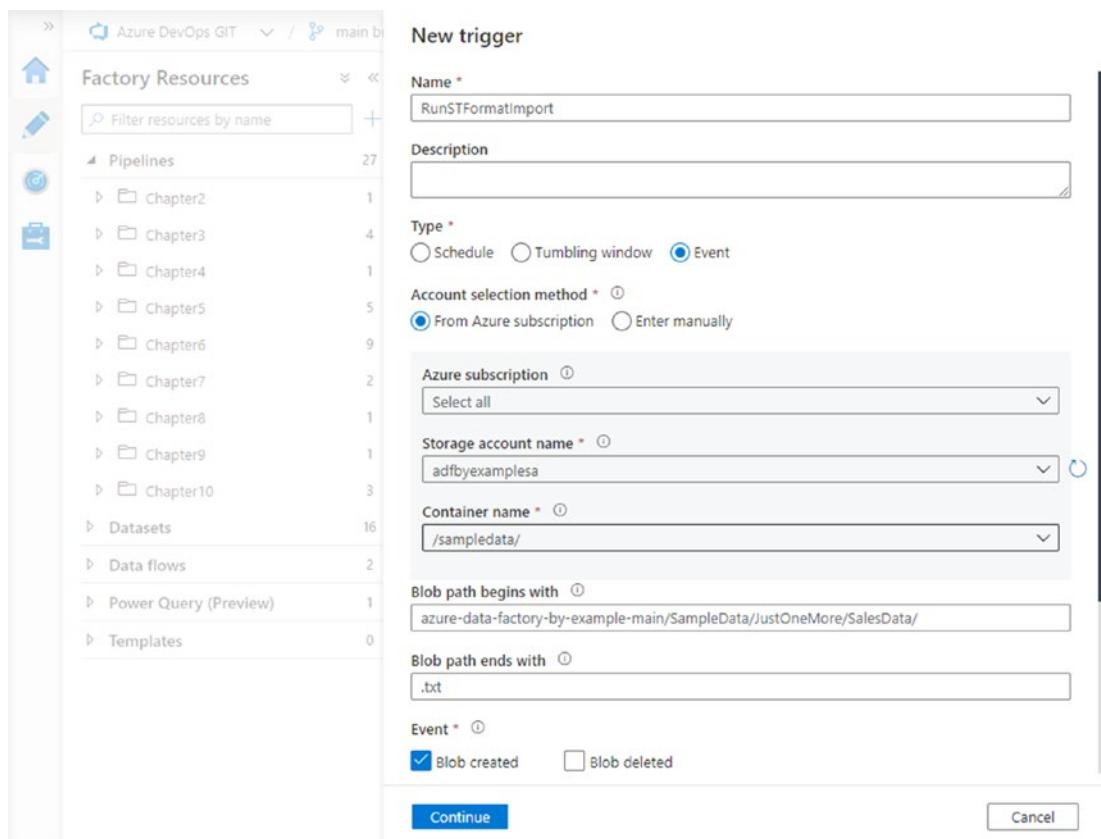


Figure 11-7. New event-based trigger configuration

If you receive a “trigger activation error” message when publishing, ensure that you have successfully registered the Microsoft.EventGrid resource provider, as in the previous section.

Cause the Trigger to Run

The event-based trigger you have created will run when a file matching the configured path pattern is created. In this section, you will run the trigger by creating files in blob storage.

1. Open the online Storage Explorer app, used in Chapter 2, and browse to your “sampledata” container. Navigate to the “JustOneMore” subdirectory of the “SampleData” folder.

CHAPTER 11 TRIGGERS

2. The “JustOneMore” folder contains two subdirectories: “TestData” and “SalesData.” Open the “TestData” folder, then use the *Download* button to copy a few of the files to your computer.
3. Navigate to the “SalesData” folder. It already contains one file – use the *Upload* button to upload one of the files you downloaded in step 2.
4. Return to the ADF UX and open the monitoring experience’s *Trigger runs* page. Verify that the “RunSTFormatImport” trigger has just executed – notice that its *Trigger type* is “BlobEventsTrigger.”
5. Open the *Pipeline runs* page, verifying that it reports a run of the pipeline “ImportSTFormatFile” triggered by “RunSTFormatImport.” The *Parameters* column contains an @ symbol in square brackets – click it to inspect the parameter values passed to the pipeline by the trigger (shown in Figure 11-8).
6. Repeat steps 3–5 for other files if you wish.

The screenshot shows the Azure Data Factory Pipeline runs page. The left sidebar has a 'Runs' section with 'Pipeline runs' selected. The main area is titled 'Pipeline runs' and shows a table of runs. One run is highlighted: 'ImportSTFormatFile' triggered by 'RunSTFormatImport' with a status of 'Succeeded'. A red box highlights the '@' symbol in the 'Parameters' column for this run. A modal window titled 'Parameters' is open over the table, showing two entries: 'Directory' with value 'azure-data-fac...' and 'File' with value 'Jun-2020.txt'. Red boxes highlight the 'Edit columns' buttons for both rows in the modal.

Pipeline name	Triggered by	Status	Parameters
ImportSTFormatFile	RunSTFormatImport	Succeeded	[@]
ImportSTFormatFile			
ImportSTFormatFile			
ImportSTFormatFolder			
ImportSweetTreatsSales_Audit			
ImportSTFormatFolder			
ImportSweetTreatsSales_Audit			
ImportSTFormatFolder			
ImportSweetTreatsSales_Audit			
ImportSweetTreatsSales_Audit			
ImportSTFormatFolder			

Name	Value	Edit columns
Directory	azure-data-fac...	+ Add column
File	Jun-2020.txt	+ Add column

OK Cancel

Figure 11-8. Parameter values passed by the trigger “RunSTFormatImport”

Like a schedule trigger, an event-based trigger can be reused across multiple pipelines, and a pipeline can use a combination of multiple event-based and schedule triggers. In practice, the reuse of event-based triggers is less common, simply because files in a particular blob storage location usually require the specific processing activity implemented in an associated pipeline.

In blob storage, the Event Grid service raises an event each time a file is created or deleted, so an upload of multiple files will cause the same number of file creation events. This in turn will cause an event-based trigger to run several times, once for each file created. This behavior is exactly what is required to process each and every new file, but you must ensure that your pipelines will run safely when multiple runs are triggered simultaneously.

Trigger-Spaced System Variables

The `@triggerBody()` function introduced in the previous section is shorthand for `@trigger().outputs.body`. In the same way that system variables that begin with `@pipeline()` can only be used in the scope of an ADF pipeline, so `@trigger()` system variables are available only within trigger definitions. Different trigger types support different system variables:

- Schedule triggers support `@trigger().scheduledTime` (the time at which a trigger was scheduled to run) and `@trigger().startTime` (the time at which it actually started). Dates and times returned by system variables are in UTC, irrespective of the time zone used to schedule the trigger.
- Event-based triggers also support `@trigger().startTime`, in addition to `@triggerBody().folderName` and `@triggerBody().fileName`.
- Tumbling window triggers support the same variables as schedule triggers, along with `@trigger().outputs.windowStartTime` and `@trigger().outputs.windowEndTime`.

Tumbling window triggers are described in the next section.

Tip The function `@triggerOutputs()` is shorthand for `@trigger().outputs` and can also be used to refer to tumbling window system variables. Trigger-scoped system variable expressions originate from the *Workflow Definition Language of Azure Logic Apps*, the technology used to implement ADF triggers.

Use a Tumbling Window Trigger

Tumbling window triggers are based on the same simple recurrence pattern used by schedule triggers – a start date and time paired with a recurrence interval. The result is a sequence of execution times formed by adding one or more recurrence intervals to the start date and time. Using the recurrence pattern from Figure 11-1, a tumbling window series begins like this:

- 12:58 p.m. + **1** * 2 minutes = 1:00 p.m.
- 12:58 p.m. + **2** * 2 minutes = 1:02 p.m.
- 12:58 p.m. + **3** * 2 minutes = 1:04 p.m. and so on

The effect of this is to create an infinite sequence of time slices or “windows” – periods between one execution time and the next – “tumbling” over one another as each window ends and the next begins.

As described earlier, the scope of a tumbling window trigger defines two additional system variables: `@trigger().outputs.windowStartTime` and `@trigger().outputs.windowEndTime`, which can be passed to a pipeline run as parameter values. Unlike a schedule trigger, a tumbling window trigger runs for every window end time defined by its recurrence pattern, even those in the past. Combining this behavior with window start and end time variables supports time-sliced processing requirements, including the ability to rerun selected slices on demand or to break large legacy data loads into smaller, time-bound pieces.

Prepare Data

To enable you to examine tumbling window behavior, in this section, you will create some new data files alongside existing sample data.

1. Open the online Storage Explorer app and browse to your “sampledata” container. Navigate to the “HandyCandy” subdirectory of the “SampleData” folder, used in Chapter 3.
2. The “HandyCandy” folder contains a subdirectory called “Subset1” – it contains a small subset of the Handy Candy message files and another subdirectory, called “Subset2.” Open the “Subset2” folder.
3. “Subset2” also contains a few Handy Candy message files. Copy some of these into the (parent) “Subset1” folder. To copy a file in the online Storage Explorer, right-click it and select *Copy*, then navigate to the parent folder and select *Paste* (from the *More* menu).

When you have finished copying files, the folder “Subset1” will contain two groups of files:

- Files created when you loaded the sample data into your storage account, with a corresponding last modified date and time
- Files created when you copied them from the “Subset2” folder, with more recent modification times

Create a Windowed Copy Pipeline

Tumbling window triggers define a sequence of processing windows. In this section, you will create a pipeline able to use and respect window boundaries.

1. In the ADF UX, create a clone of the pipeline “IngestHandyCandyMessages” from Chapter 3. Rename it to “IngestHandyCandyWindow” and move it into a new “Chapter11” pipelines folder.
2. Define two pipeline parameters for the new pipeline, of type String, called “WindowStartUTC” and “WindowEndUTC.”
3. Select the pipeline’s Copy data activity and open its *Source* configuration tab. Edit the *Wildcard paths* folder path field, appending the “Subset1” path segment.

4. To the right of *Filter by last modified*, update the *Start time (UTC)* and *End time (UTC)* fields to use the values of pipeline parameters “WindowStartUTC” and “WindowEndUTC.” To avoid loading the “Subset2” files twice (the originals and the copies), untick the *Recursively* checkbox. Figure 11-9 shows the correctly configured *Source* tab.
5. Run the pipeline in Debug mode. The ADF UX will prompt you to supply values for the two pipeline parameters – set *WindowStartUTC* to “2021-01-01”. Choose a value for *WindowEndUTC* that falls after the modification time of the oldest files in the “Subset1” folder, but before the modification time of the newest.
6. Click *OK* to run the pipeline. When the pipeline has finished executing, use the Copy data activity output to verify that the expected number of files has been copied – it should match the number of files in the folder “Subset1” having modification timestamps between the specified *WindowStartUTC* and *WindowEndUTC* values.

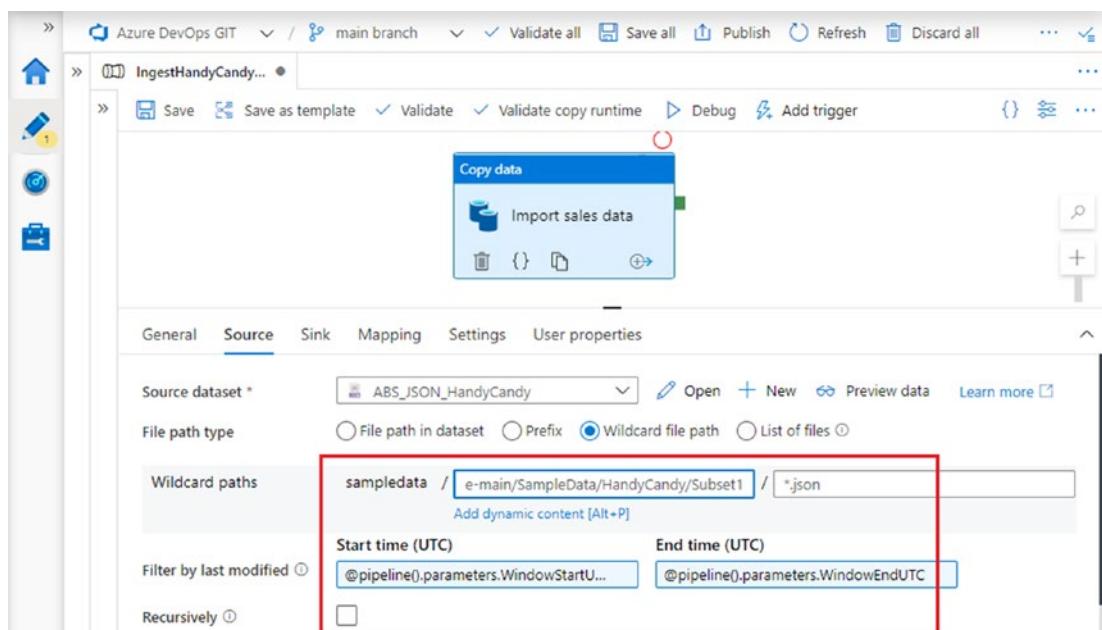


Figure 11-9. Source tab filtering input files by last modified time

The pipeline you have created can now be used to load arbitrary subsets of files, based on their last modified date and time.

Create a Tumbling Window Trigger

The pipeline you created in the previous section is suitable for use with a tumbling window trigger. In this section, you will create and configure that trigger.

1. Use the *Add trigger* dropdown to access the *Add triggers* blade, then select the option *+ New* from the *Choose trigger...* dropdown.
2. Name the trigger “IngestHandyCandyWindow” and set its *Type* to “Tumbling window.” Choose a *Start Date (UTC)* value that falls before the earliest file modification time in the “Subset1” folder.
3. Specify a *Recurrence* interval. For this example, try to find a value that defines several windows, but not so many that very few contain modified files. Click *OK* to continue.
4. The *New trigger* blade now prompts for pipeline parameter values. Set *WindowStartUTC* to “@trigger().outputs.windowStartTime” and *WindowEndUTC* to “@trigger().outputs.windowEndTime”.
5. Click *Save* to save your changes and close the blade, then click *Publish* to deploy both the new pipeline and the trigger to the published factory environment.

Tumbling window triggers are associated with a single pipeline and cannot be reused. If you try to add an existing trigger to a different pipeline, you will notice that “IngestHandyCandyWindow” is not available for selection from the list of triggers.

Monitor Trigger Runs

After a tumbling window trigger is published, trigger and pipeline runs begin as follows:

- Runs for windows whose end time has already passed begin immediately. Up to 50 trigger runs can take place concurrently – remaining runs are queued until others have finished executing.

- Runs for future windows start automatically when their window end time is reached.

Verify that trigger runs have taken place as you expected – use the *Tumbling window* tab of the monitoring experience's *Trigger runs* page to access a convenient view showing window start and end times. Figure 11-10 shows five runs of a trigger with the following features:

- The trigger's *Start date* was set to 3 p.m., with a recurrence interval of one hour.
- The trigger was deployed to the published environment shortly after 6:15 p.m. on the trigger's start date.

Three trigger runs took place immediately after it was published, at the same *Trigger time*, for the finished windows 3–4 p.m., 4–5 p.m., and 5–6 p.m. A fourth run took place at 7 p.m., as scheduled, at the end of the 6–7 p.m. window.

Trigger name	Window start time	Window end time	Trigger time	Status	Run
IngestHandyCandyWindow	2/16/21, 4:00:00 PM	2/16/21, 5:00:00 PM	2/16/21, 7:07:37 PM	✓ Succeeded	Rerun (Latest)
IngestHandyCandyWindow	2/16/21, 4:00:00 PM	2/16/21, 5:00:00 PM	2/16/21, 6:16:58 PM	✓ Succeeded	Original
IngestHandyCandyWindow	2/16/21, 6:00:00 PM	2/16/21, 7:00:00 PM	2/16/21, 7:00:00 PM	✓ Succeeded	Original
IngestHandyCandyWindow	2/16/21, 5:00:00 PM	2/16/21, 6:00:00 PM	2/16/21, 6:16:58 PM	✓ Succeeded	Original
IngestHandyCandyWindow	2/16/21, 3:00:00 PM	2/16/21, 4:00:00 PM	2/16/21, 6:16:58 PM	✓ Succeeded	Original

Figure 11-10. Tumbling trigger runs

The latest trigger run is a rerun of the 4–5 p.m. window, invoked by clicking the *Rerun* button (indicated in the screenshot for the 5–6 p.m. run). The original run for the 4–5 p.m. window is nested beneath the rerun. This illustrates a key use case for tumbling window triggers – the ability to rerun pipelines for a specific time slice, for example, after failure, or to incorporate late-arriving data.

Inspect the pipeline runs invoked by your tumbling window trigger, and use the Copy data activity output to verify that the expected files were loaded in each run. The original “Subset1” files are loaded by the pipeline run corresponding to the window

during which they were created. The copied “Subset2” files fall into a later window and are loaded by the corresponding run. Any additional pipeline runs should have loaded no files. Tumbling window triggers’ simple recurrence pattern means that they continue to run indefinitely until deactivated – remember to deactivate the trigger and to publish that change.

Advanced Features

Tumbling window triggers additionally support advanced features such as start time offsets, retry policies, and trigger dependencies. Advanced features are configured in the *Advanced* section of the *New trigger* or *Edit trigger* blades:

- By default, trigger runs for future windows take place at each window’s end time – a nonzero *Delay* value allows you to postpone this. For example, to process a 24-hour, midnight-based window at 3 a.m., use a delay of 3 hours.
- *Max concurrency* controls the maximum number of trigger runs allowed to occur in parallel, 50 by default. You cannot increase the number above 50, but you can lower it if necessary, for example, to control the load on a source system from which multiple windows of data are being extracted.
- *Retry policy* attributes allow you to specify trigger behavior when the executed pipeline fails with a “retryable” error – for example, an intermittent error related to service conditions outside your control. You can specify the number of times a failing pipeline should be retried before abandoning it and how long ADF should wait before retrying a failed execution.
- *Trigger dependencies* permit you to link tumbling windows together, from the same or different tumbling window triggers. A dependency identifies a trigger, an offset, and a size – specifying that trigger T2 is dependent on T1 with an offset of zero means that (for example) T2’s 1-2 p.m. window trigger run will not start until the pipelines executed by T1’s 1-2 p.m. window have completed.

A tumbling window trigger *self-dependency* can be used to ensure that successive windows are processed in order or that pipeline runs cannot begin while prior runs are still in progress. If ensuring that scheduled runs do not overlap is important to you, tumbling window trigger self-dependencies may be of use.

Publishing Triggers Automatically

Like other Azure Data Factory resources, trigger definitions are stored as JSON files in your Git repository (in the “triggers” folder of your ADF root folder) and included automatically in ARM templates for publishing. However, if the deployment process attempts to update an active trigger, it may fail. To avoid this contingency during automated deployments, include tasks in your Azure DevOps pipeline to stop existing triggers prior to deployment and to restart them afterward.

This can be achieved using the Azure PowerShell task (`AzurePowerShell@4`) with three trigger-related cmdlets:

- Use `Get-AzDataFactoryV2Trigger` to list and identify active triggers in the deployment target factory.
- Use `Stop-AzDataFactoryV2Trigger` to stop active triggers prior to deployment.
- Use `Start-AzDataFactoryV2Trigger` to restart active triggers when deployment is complete.

This approach is equally applicable to deployments made using ARM templates or individual JSON resource definitions. To create or update a published trigger from its JSON definition, use the `Set-AzDataFactoryV2Trigger` cmdlet.

When an ARM template deployment creates a new trigger, its runtime state is always “Stopped,” even if its JSON definition in Git indicates that it is to be started. New triggers must be started explicitly – a postdeployment step using `Start-AzDataFactoryV2Trigger` is useful in this scenario. In contrast, a deployed trigger’s runtime state after a direct JSON-based deployment is always as specified in the deployed JSON definition.

Triggering Pipelines Programmatically

You saw in Chapter 10 that it is possible to run a published pipeline without using an ADF trigger, by triggering a pipeline manually from the ADF UX. You can trigger pipelines in the published environment programmatically, from outside the Azure Data Factory service, using a variety of different technologies, for example:

- The `Invoke-AzDataFactoryV2Pipeline` cmdlet can be used to start a pipeline from PowerShell.
- The .NET API is a convenient way to start pipeline runs from a .NET application (e.g., an Azure Function).
- ADF's REST API permits you to start a pipeline run by calling an HTTP endpoint.

ADF's Execute Pipeline activity is limited to starting a named pipeline (hard-coded) from within the same factory – using the Web activity to call the REST API enables you to parameterize pipeline names and to call pipelines in other factory instances.

Chapter Review

This chapter introduced the three types of ADF trigger resource: schedule, event-based, and tumbling window triggers.

- Schedule triggers execute pipelines on a wall clock schedule.
- Event-based triggers execute pipelines when a file is created or deleted in Azure blob storage.
- Tumbling window triggers add advanced behaviors, extending the idea of recurring schedule to define a sequence of processing windows.

In addition to using an ADF trigger resource, pipelines can be triggered directly from the ADF UX or using a variety of programmatic routes into ADF.

Key Concepts

Key concepts introduced in this chapter include

- **Trigger:** A unit of processing that runs one or more ADF pipelines when certain execution conditions are met. A pipeline can be associated with – and run by – more than one trigger.
- **Trigger run:** A single execution of a trigger. If the trigger is associated with multiple pipelines, one trigger run starts multiple pipeline runs. The ADF UX monitoring experience reports trigger and pipeline runs separately.
- **Trigger start date:** The date and time from which a trigger is active.
- **Trigger end date:** The date and time after which a trigger is no longer active.
- **Recurrence pattern:** A simple time-based scheduling model, defined by a repeated interval after a given start date and time.
- **Schedule trigger:** A trigger whose execution condition is defined by a recurrence pattern based on the trigger's start date or using a wall clock schedule.
- **Event-based trigger:** A trigger whose execution condition is the creation or deletion of a file from Azure blob storage.
- **Resource provider:** Azure uses resource providers to create and manage resources. In order to use a resource in an Azure subscription, the corresponding resource provider must be registered to that subscription.
- **Azure Event Grid:** Cloud service providing infrastructure for event-driven architectures. Azure blob storage uses Event Grid to publish file creation and other events; ADF subscribes to Event Grid to consume events and run event-based triggers.

- **Tumbling window trigger:** A trigger that uses a recurrence pattern based on the trigger's start date to define a sequence of processing windows between successive executions. Tumbling window triggers also support more advanced scheduling behaviors like dependencies, concurrency limits, and retries.
- **Pipeline run overlap:** Pipeline runs may overlap if a trigger starts a new pipeline run before a previous one has finished. Use a tumbling window self-dependency with a concurrency limit of one to prevent this.
- **Reusable triggers:** A single schedule or event-based trigger can be used by multiple pipelines. A tumbling window trigger can be used by a single pipeline.
- **Trigger-scoped system variables:** ADF system variables available for use in trigger definitions. Some trigger-scoped variables are specific to the type of ADF trigger in use.
- **Azure Logic Apps:** Cloud service for general-purpose task scheduling, orchestration, and automation. Internally, ADF triggers are implemented using Azure Logic Apps.
- **Trigger publishing:** Triggers do not operate in the ADF UX debugging environment and must be published to a factory instance to have any effect.

For SSIS Developers

Users of SQL Server Integration Services will be used to scheduling SSIS packages for execution using the SQL Server Agent. The advanced scheduling options supported by ADF's schedule trigger provide a similar experience.

When the time for a new job execution is reached, the SQL Server Agent only starts the job if it is not already running. This is not the case when using an ADF schedule trigger. To achieve the effect in Azure Data Factory, use a self-dependent tumbling window trigger with a maximum concurrency of one.

CHAPTER 12

Monitoring

The previous two chapters have been principally concerned with what happens to Azure Data Factory resources after you have finished developing them – how to get them into a production environment and how to run them automatically. This final chapter completes a trio of requirements for operating a production ADF instance: monitoring the behavior of deployed factory resources to ensure that individual resources and the factory as a whole continue to operate correctly.

Generate Factory Activity

To assist your exploration of factory monitoring tools, begin by generating a factory workload as described in the following:

1. Navigate to the *Triggers* page in the ADF UX management hub and activate the “RunEvery2Minutes” trigger from Chapter 11. When published, this will cause two pipelines to be executed every two minutes, until the trigger is deactivated.
2. In the pipeline authoring workspace, open the pipeline “ImportSTFormatFile” from Chapter 6. Add a new schedule trigger to run the pipeline automatically.
3. Name the new trigger “RunEvery8Minutes” and set its *Recurrence* to every 8 minutes. Click *OK* to set the pipeline parameter values.
4. Set the value of the *Folder* parameter to “azure-data-factory-by-example-main/SampleData/NaughtyButNice” and the *File* parameter value to “NBN-202006.csv”. Recall that this file contains a format error, so every triggered run of the pipeline will fail – the intention here is to simulate a mixed workload that contains occasional failures.

5. Save the new trigger, then click *Publish* to deploy your changes into the factory's published environment.

Remember to deactivate and republish the two triggers when you have finished exploring monitoring capabilities at the end of this chapter.

Inspect Factory Logs

The three Azure Data Factory components that “run” – triggers, pipelines, and activities – are reported in execution logs that can be inspected in the ADF UX monitoring experience.

Inspect Trigger Runs

As you discovered in Chapter 11, trigger runs are reported separately from pipeline runs – open the ADF UX monitoring experience and select the *Trigger runs* page.

1. The triggers you published in the previous section will not run until the first time determined by their recurrence patterns after publishing, so the list of trigger runs may be empty. Use the *Refresh* button at the top of the page to update the list until you see new trigger runs start to appear.
2. By default, trigger runs are reported from the last 24 hours. Each trigger run includes details such as the trigger name, type, actual trigger execution time, and status. Notice that runs of both triggers are reported to succeed – this indicates that each trigger was able to start the associated pipeline runs, even if those runs subsequently failed.
3. Use the *Trigger time* column header to sort the list alternately into ascending and descending order of execution time. The pair of up/down arrows to the right of the column name indicates both the fact that the list can be sorted using that column and the column's current sort order.

4. The *Properties* column provides links to JSON objects containing information specific to the type of trigger. For example, in the case of a schedule trigger, this includes the run's schedule time, while for a tumbling window trigger, you can verify the associated window start and end times. You can promote these properties into list columns by selecting them using the *Edit columns* button (next to the *Refresh* button).

The remaining controls at the top of the page enable you to filter the list in various different ways – for example, by trigger type, name, or status.

Inspect Pipeline Runs

The monitoring experience's *Pipeline runs* page reports runs from both the published and debugging environments. Selecting the *Debug* tab in the top left allows you to see a history of debug runs, enabling you to look at multiple prior debugging runs for a given pipeline. The *Triggered* tab lists pipeline runs that have taken place in the published environment and is the focus of this section.

1. Select the *Pipeline runs* page in the monitoring experience and ensure that the *Triggered* tab is selected. The page includes a similar set of controls as those for trigger runs, allowing you to choose which rows or columns are displayed.
2. Hover over a pipeline's name to reveal additional controls. For pipelines still in progress – as shown in Figure 12-1 – buttons to cancel a pipeline's execution are displayed. *Cancel* stops a running pipeline, while *Cancel recursive* also stops any executions initiated by the pipeline using the Execute Pipeline activity.
3. Hover over a completed pipeline's name to display *Rerun* and *Consumption* buttons. *Rerun* allows you to run a pipeline using the same parameter values supplied for its original run, whatever the outcome of its previous execution. (The *Run* column indicates whether a given pipeline execution was a rerun or not, with any prior runs displayed beneath the latest rerun). The *Consumption* button displays a popup summarizing resources used in the pipeline's execution.

Pipeline name	Run start	Triggered by	Status	Run	Parameters	Error
ImportSTFormatFolder	12/19/20, 1:06:00 PM	RunEvery2Minutes	In progress	Original	@1	
ImportSweetTreat...	12/19/20, 1:06:00 PM	RunEvery2Minutes	In progress	Original		
ImportSTFormatFile	12/19/20, 1:04:00 PM	RunEvery8Minutes	Failed	Original	@1	Details
ImportSTFormatFolder	12/19/20, 1:04:00 PM	RunEvery2Minutes	Succeeded	Original	@1	
ImportSweetTreats_RunSeqNo	12/19/20, 1:04:00 PM	RunEvery2Minutes	Succeeded	Original		
ImportSTFormatFolder	12/19/20, 1:03:10 PM	Manual trigger	Succeeded	Rerun (Latest)	@1	
ImportSTFormatFolder	12/19/20, 12:58:46 PM	Manual trigger	Succeeded	Rerun	@1	
ImportSTFormatFolder	12/19/20, 12:46:01 PM	RunEvery2Minutes	Succeeded	Original	@1	
ImportSTFormatFolder	12/19/20, 1:03:11 PM	RunEvery2Minutes	Succeeded	Original	@1	

Figure 12-1. Pipeline runs in the ADF UX monitoring hub

4. Use the checkboxes to the left of the *Pipeline name* column to select multiple runs. This enables you to cancel multiple running pipelines at once or to rerun several pipelines, using the *Rerun* and *Cancel* buttons at the top of the page. Use the *List/Gantt* toggle at the top of the page to switch back and forth between the list of pipeline runs and a Gantt chart view.
5. Click a pipeline name in the list view's *Pipeline name* column to access the detail of each pipeline run's activity executions, in a form familiar from debugging in the authoring workspace. You are unable to access nested activities using the pipeline diagram, but you can use it to select a specific activity from which to rerun a pipeline, as shown in Figure 12-2. The full list of activity executions appears below the pipeline diagram. Use the *List/Gantt* toggle to display a Gantt chart view of activity runs, then click *All pipeline runs* in the top left to return to the list of pipeline runs.

Tip Unlike in a debug run output, details of activity failures are not displayed next to the activity name. Scroll the activity run list further to the right to locate error details.

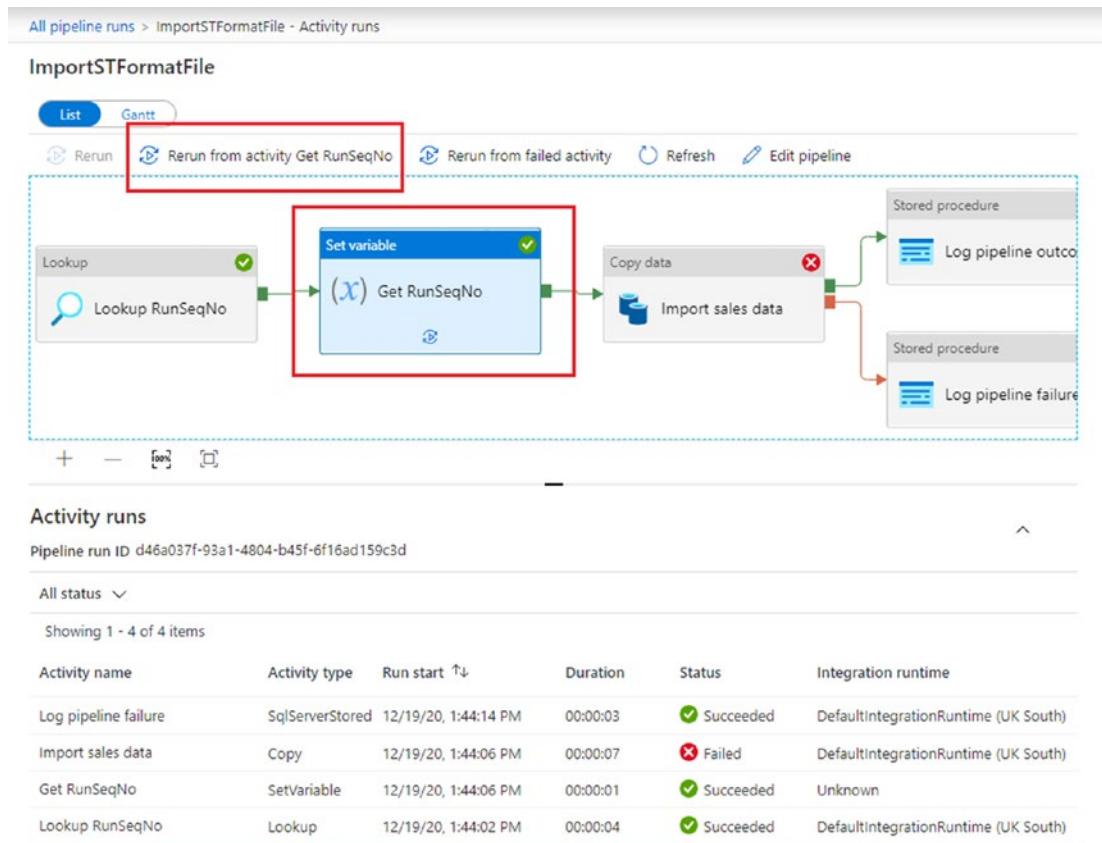


Figure 12-2. Pipeline run detail

Add Metadata to the Log

The trigger, pipeline, and activity attributes reported so far are all standard elements of Azure Data Factory's logging functionality. Sometimes, however, it is useful to be able to add additional metadata of your own – for example, to make it easier to identify particular subsets of factory activity in an otherwise noisy log. The following sections describe the use of pipeline *annotations* and activity *user properties* to achieve this.

Add a Pipeline Annotation

A pipeline annotation is a label added optionally to a pipeline's definition. Annotate your "ImportSTFormatFolder" pipeline as follows:

1. In the ADF UX authoring workspace, open Chapter 5 pipeline "ImportSTFormatFolder," one of the two pipelines triggered by "RunEvery2Minutes."
2. Open the *Properties* blade for the pipeline using the button (slider icon) at the top right of the authoring canvas. In the *Annotations* section, click the *+ New* button to add a new annotation.
3. A *Name* field is displayed – enter an annotation of your choice, then save the pipeline.

Add an Activity User Property

Unlike pipeline annotations, activity user properties are name-value pairs, enabling you to specify different values for a common property across multiple activities.

1. On the authoring canvas, select the "ImportSTFormatFolder" pipeline's Copy data activity. Select the *User properties* tab in the configuration pane.
2. Click the *+ New* button to add a new user property. Enter a property *Name* and *Value* of your choice.
3. Click the *Auto generate* button. This option is specific to the Copy data activity and adds two copy-specific properties named "Source" and "Destination." Their values are populated with expressions that will be translated at runtime into the activity's data source and sink, then written into the pipeline run history.
4. Save the pipeline and publish your changes.

Inspect Pipeline Annotations in the Log

Annotations attached to a pipeline at the time of its execution are recorded in the log, allowing them to be used to filter and group pipeline runs. You will be able to see

annotations in the log as soon as executions of the revised, annotated pipeline have taken place.

1. Open the *Pipeline runs* page in the ADF UX monitoring experience. When the annotated pipeline next runs, the Annotations column will contain a luggage label icon, indicating that the run has one or more annotations. Click the icon to display the run's annotation.
2. The primary purpose of pipeline annotations is to enable you to apply custom filters to the pipeline run list. Click *Add filter*, then select “Annotations” from the dropdown list.
3. Choose your annotation value from the list, then click some empty space in the *Pipeline runs* page to apply the filter and exclude non-annotated runs from the list.
4. Remove the annotations filter, then use the *List/Gantt* toggle to switch to the Gantt chart view. By default, the Gantt chart is grouped by pipeline name – tick the *Group by annotations* checkbox to enable an alternative presentation. Notice that only the most recent pipeline runs appear with your annotation value – runs which took place before the annotation was created appear in the “No annotations” group.

The annotation functionality described here for pipelines and pipeline runs is also available for triggers and trigger runs. The behavior of trigger annotations is very similar and is not described separately here.

Inspect User Properties in the Log

Select one of your annotated pipeline runs to examine its activity runs in more detail. You can do this directly from the Gantt chart view by clicking the bar representing the pipeline run, then clicking the pipeline name in the displayed popup box.

1. Locate the *User properties* column in the activity runs pane below the pipeline diagram (you may need to toggle the view back to *List* to do so). For activity runs where one or more user properties were defined, the column contains a bookmark icon – click it.

2. A *Parameters* dialog is displayed. This allows you to inspect configured user properties and their values and to promote them as columns in the activity runs list if you wish.
3. Add or remove columns using the button in the column headed *Edit columns*, then click *OK*.
4. Verify that the columns you selected for inclusion are now visible in the list of activity runs.

Inspect Factory Metrics

The structure of Azure Data Factory logs is naturally tightly coupled to the nature of factory resources, allowing concepts like triggers, pipelines, and activities to be represented in detail. In contrast, a *metric* is a simple count of a given system property over a period of time, emitted and logged automatically. *Azure Monitor* is a resource monitoring service used to collect data – including metrics – from all Azure resources.

Metrics emitted by ADF include factory activity levels, for example, the number of failed or successful pipeline runs. Other indicators, related to factory size and integration runtimes, contribute to an overall picture of the health of a factory instance. Inspect metrics emitted by your factory as follows:

1. Open the Azure portal, then browse to the resource blade for your data factory.
2. In the left sidebar menu, scroll down to the *Monitoring* section, then select *Metrics*.
3. The *Metrics* page displays a metric selection tool above an empty chart. (If the metric selection tool is not visible, click *Add metric* in the chart header bar.) Choose “Succeeded activity runs metrics” from the selection tool’s *Metric* dropdown.
4. Click *Add metric* in the chart header bar to add another metric – this time, choose “Failed activity runs metrics.”

Figure 12-3 shows these two metrics under a test workload like the one you created at the beginning of the chapter. The pattern is unusually regular due to the nature of the test workload.

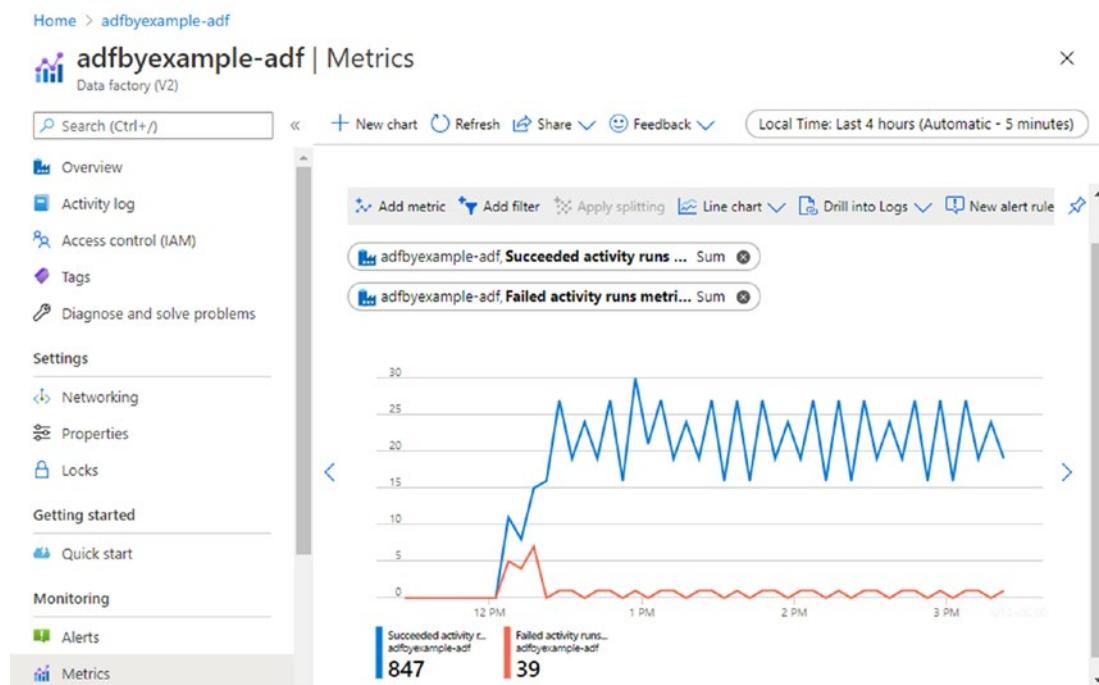


Figure 12-3. Azure Data Factory activity runs metrics

For SSIS developers Metrics emitted by Azure Data Factory serve the same purpose as SSIS performance counters. They do not provide information about specific system activities, but can be used in a general way to understand system health. Just as Integration Services is not the only Windows service to maintain performance counters, so many Azure resources emit resource-specific metrics that can be accessed in the same way.

Export Logs and Metrics

The log analysis capabilities offered by the ADF monitoring experience are somewhat limited, as are those for metrics in the data factory resource blade. To perform more sophisticated analysis and monitoring of the factory activity, you must export log and metric data to a service better suited to doing so. Approaches to doing this, examined in the following sections, offer the additional benefit of being able to keep diagnostic data for longer – logs generated by Azure Data Factory are purged automatically after 45 days, and metrics emitted to Azure Monitor are stored for no more than 93 days before being deleted.

Create a Log Analytics Workspace

Log Analytics is a component of Azure Monitor that supports sophisticated analysis of both system logs and metrics. In the next section, you will configure Azure Data Factory to send logs and metrics to Log Analytics – before doing so, you must create a Log Analytics *workspace* in which to store them.

1. In the Azure portal, create a new resource of type *Log Analytics Workspace*.
2. Select the subscription and resource group that contains your ADF instance, then under *Instance details*, enter a globally unique *Name*.
3. Ensure that *Region* matches the Azure region containing your data factory, then select *Review + Create*.
4. Review the details you have provided, then click *Create* to begin deployment.

Configure Diagnostic Settings

Although not configured by default, every Azure resource has the ability to interact with Azure Monitor, sending logs and metrics to Log Analytics and other destinations by means of *diagnostic settings*. A diagnostic setting specifies a set of resource logs and metrics to be sent to up to three destinations – one Azure storage account, one Log Analytics workspace, and one Azure Event Hub.

In this section, you will configure a diagnostic setting for your ADF instance, sending factory logs and metrics to blob storage and to Log Analytics.

1. Return to the resource blade for your data factory in the Azure portal, then in the *Monitoring* section of the sidebar, select *Diagnostic settings*.
2. The *Diagnostic settings* page lists settings created for the factory resource (currently none). Click *+ Add diagnostic setting* to create a new one.
3. Provide a new *Diagnostic setting name*, then under *Category details*, tick checkboxes to select “ActivityRuns,” “PipelineRuns,” and “TriggerRuns” from the *log* section and “AllMetrics” from the *metric* section.
4. Under *Destination details*, tick the “Send to Log Analytics workspace” checkbox to enable selection of your new workspace. Select your subscription and workspace from the respective dropdowns, and ensure that the *Destination table* toggle is set to “Resource specific.”
5. Tick the “Archive to a storage account” checkbox to configure a storage account destination. This means that log and metric information will be streamed from your ADF instance to two separate destinations: your Log Analytics workspace and the storage account you specify here. Select your subscription and the storage account you have been using throughout from the respective dropdowns.

Note Your existing storage account is sufficient for the purpose of this exercise, but Microsoft recommends that production logs are sent to a dedicated storage account.

6. Choosing the “Archive to a storage account” option causes the portal to offer you a choice of retention period for each of the log and metric categories. Leave each value at its default of zero and save your changes.

A nonzero retention period will cause log data to be deleted from blob storage automatically after it has been stored for the specified period, while a value of zero means “retain forever.” Retention periods are attached to log entries as they are written and cannot be changed – if you modify a diagnostic setting’s retention period later, retention settings for log data already in storage will not change.

Inspect Logs in Blob Storage

Blob storage is a convenient and cost-effective form of long-term storage for data of all kinds, including platform logs and metrics. It is not however particularly convenient for log analysis or querying, as you will see here.

1. Open the online Storage Explorer app for your storage account.
As logs start to be written to your storage account – this may take a few minutes – new blob storage containers are created automatically to contain them. A separate container is created for metrics and another for logs from each of trigger runs, pipeline runs, and activity runs. Figure 12-4 shows the four storage account containers created by my ADF instance’s diagnostic settings.
2. Explore the activity runs container – it holds a single root folder called “resourceId=”. The folder subtree corresponds to the path segments of the fully qualified Azure resource ID of the data factory, broken down further into year, month, day, hour, and minute. Keep drilling down into folders until you reach the minute folder, named something like “m=00”.
3. Logs are written by appending JSON objects to text files – the log file in Figure 12-4 is named “PT1H.json”. Use the *Download* button to copy a log file to your computer.
4. Open the downloaded log file in a text editor to view its contents – you will be able to see pipeline and activity names familiar from the pipelines being triggered, along with a variety of other log information.

The complete log file is not valid JSON – it consists of JSON object definitions appended one after the other rather than being (for example) a true JSON array. The reason for sending log data to blob storage here was to illustrate both the possibility

and some of its shortcomings – while highly cost-effective, further analysis of log data stored like this requires it to be loaded into a query engine of some kind. In the following section, you will examine how your Azure Log Analytics workspace supports this requirement directly.

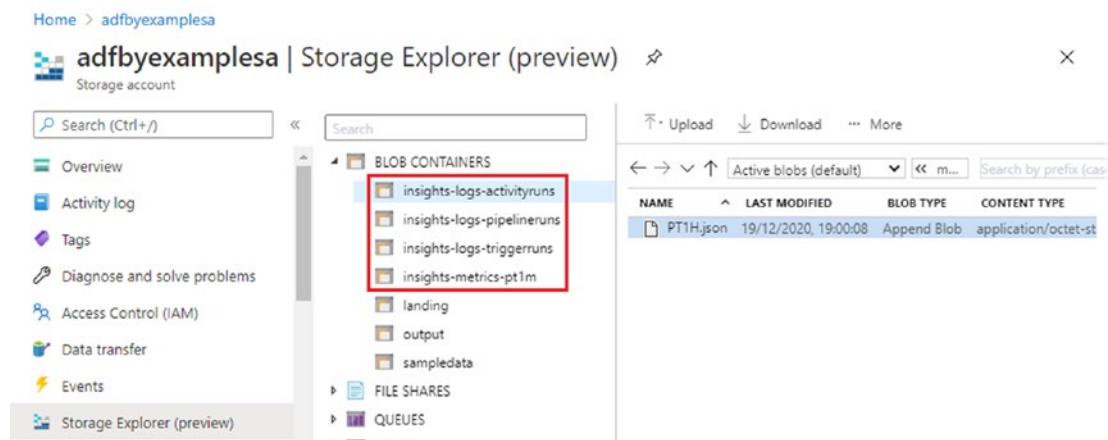


Figure 12-4. Diagnostic settings containers in online Storage Explorer

Use the Log Analytics Workspace

As you discovered in the previous section, using a diagnostic setting to copy logs to Azure blob storage solves only the problem of retention – further work would be required to be able to query and analyze the collected log data. As the name suggests, an Azure Monitor Log Analytics workspace can support both requirements. By default, data sent to a Log Analytics workspace is retained for 31 days, but you can increase this to up to 2 years. In the following sections, you will explore two approaches for interacting with log data sent to a Log Analytics workspace.

Query Logs

A Log Analytics workspace provides the sort of query engine functionality not available when logs are persisted simply as files in Azure blob storage. In this section, you will examine the tabular presentation of log data provided by Log Analytics and write queries to explore it.

1. In the Azure portal, browse to the Log Analytics workspace you created earlier in the chapter. Select *Logs* from the *General* section of the sidebar to open a new query tab.
2. If the *Queries* dialog is displayed, close it. The new query tab contains a tabbed sidebar, a query pane, and a history of recent queries.
3. Make sure that *Tables* is selected in the tabbed sidebar. The *LogManagement* section lists log tables being maintained in this workspace – as log data begins to arrive, a total of four tables appear, corresponding to the category details you selected when creating your ADF diagnostic settings. A fifth table called “Usage” reports usage data for the Log Analytics service itself.
4. Log Analytics queries are written using the *Kusto* query language. Kusto queries contain at least one *tabular expression statement* – a statement with table-like output. A tabular expression statement always begins with a data source, followed optionally by a sequence of transformations separated by the pipe (“|”) character. The simplest query identifies a data source – one of the tables listed in the *LogManagement* section. Enter the query `ADFPipelineRun;` in the query pane and click *Run*.
5. The query returns pipeline run details sent to Log Analytics by your ADF diagnostic setting – the data factory’s own logs are not interrogated directly. Refine the query to return pipeline runs that have failed in the last 10 minutes – the query shown in Figure 12-5 is one solution.

Run a few more queries on “ADFPipelineRun” and other tables to explore logs and metrics sent to Log Analytics by your ADF diagnostic setting. The Kusto query pane provides IntelliSense support to help you write queries more quickly.

The screenshot shows the Azure Log Analytics workspace interface. At the top, there's a navigation bar with 'Home > adfbylexample-law'. Below it is the workspace title 'adfbylexample-law | Logs' and a 'Log Analytics workspace' status indicator. The main area has tabs for 'Tables' (which is selected), 'Queries', and 'Filter'. A search bar and filter dropdown are also present. On the right, there are buttons for 'Run', 'Time range: Last 24 hours', 'Save', 'Copy link', and 'New alert rule'. A code editor window contains a Kusto query:

```

1 ADFPipelineRun
2 | where End > datetime_add("Minute", -10, now())
3 | where Status == "Failed"
4 | order by End desc
5 | project PipelineName, OperationName, Level;
6

```

The results table below shows the query results:

PipelineName	OperationName	Level
ImportSTFormatFile	ImportSTFormatFile - Failed	Error

At the bottom, there are pagination controls ('Page 1 of 1'), a '50 items per page' dropdown, and a 'Display time (UTC+00:00)' dropdown.

Figure 12-5. Querying logs in a Log Analytics workspace

Use a Log Analytics Workbook

Log Analytics *workbooks* provide a modern, notebook-like interface for presenting Log Analytics data. A workbook enables you to embed Kusto queries that extract and visualize data, interleaving result cells with your own headings and narrative text to support and give structure to the presentation. While you can author your own Log Analytics workbooks, a convenient place to start is by installing an operations management solution available from the Azure marketplace.

1. Browse to <https://azuremarketplace.microsoft.com> and search for “data factory analytics.”
2. Select the *Azure Data Factory Analytics* tile from the results. (At the time of writing, this app is still marked as being in preview.) On the app overview page, click the *GET IT NOW* button. Click *Continue* to accept the terms and conditions.

3. The *Create new Solution* blade opens in the Azure portal. Choose your Log Analytics workspace as the location where the solution is to be created, then click *Create*. When deployment is complete, a notification appears in the portal including a *Go to resource* button – click it.
4. In the solution blade, select *Workbooks* from the *General* section of the sidebar. Locate the *AzureDataFactoryAnalytics* tile and click it to open the workbook.
5. The workbook contains a variety of data visualizations for trigger, pipeline, and activity runs, including breakdowns by factory, frequency, and type. Figure 12-6 shows the workbook cell reporting pipeline run breakdown by factory.
6. Click *Edit* in the header bar (indicated in Figure 12-6) to inspect the workbook definition. In edit mode, an *Edit* button is displayed to the right of every workbook cell – click the button next to a cell to edit its definition.

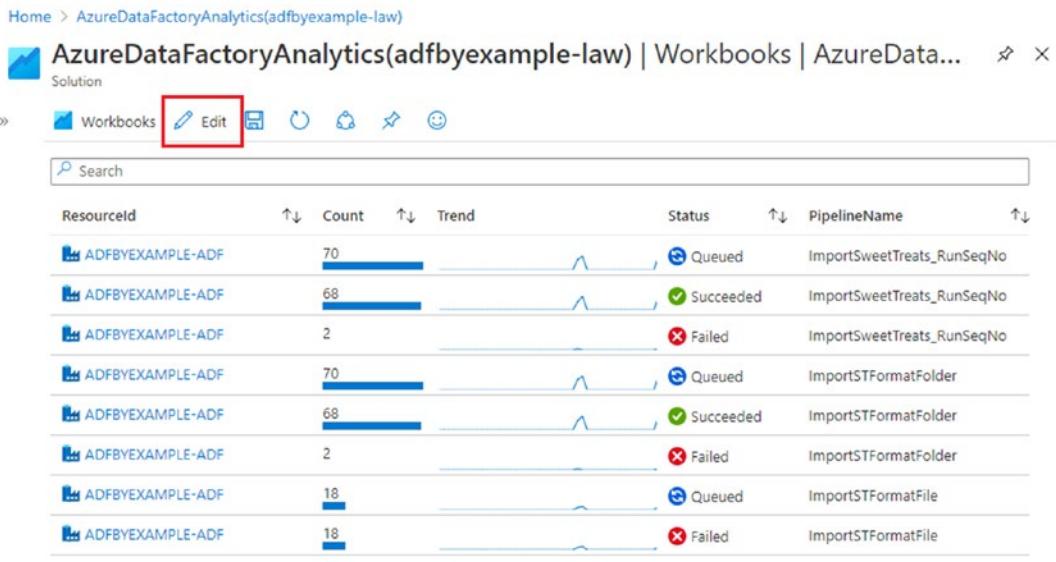


Figure 12-6. Workbook in the AzureDataFactoryAnalytics solution

In small- to medium-sized environments, Log Analytics provides a convenient central point to gather service logs and metrics from across your Azure estate and to analyze them together. At a larger scale, *Azure Data Explorer* offers the ability to query high volumes of raw log data in near real time, also using Kusto.

Receive Alerts

So far, in this chapter, you have explored how to interact with log data in the ADF UX monitoring experience and how to do so with greater flexibility using a Log Analytics workspace. Both of these approaches require proactive monitoring of the factory activity – a lighter-touch approach is to receive notifications automatically under prescribed conditions, requiring you to intervene only in cases of failure. The following sections introduce mechanisms available to issue *alerts* automatically in response to configured patterns of metric or log data.

Configure Metric-Based Alerts

Data factory metrics can provide useful indicators about overall system health. Configuring alerts based on metrics allows you to discover deteriorating system performance in a timely manner, before it becomes critical. In this section, you will configure an alert suitable for notifying administrators of a high pipeline failure rate.

1. Open the resource blade for your data factory in the Azure portal, then in the *Monitoring* section of the sidebar, select *Alerts*.
2. The *Alerts* page lists alerts that have been raised for the data factory – currently none. The conditions under which an alert is raised are defined using an *alert rule*. Click + *New alert rule* to create a new rule.
3. An alert rule has three components: its *scope*, *conditions*, and *actions*. The scope of the new alert – the resource being monitored – is your data factory and will be preselected. Its conditions define the circumstances under which an alert will be raised. In the *Condition* section, click *Add condition*.

4. On the *Configure signal logic* blade, choose “Failed pipeline runs metrics” – a chart appears automatically, displaying the chosen signal’s recent history. Under *Alert logic*, set the *Threshold value* to 0. The *Condition preview* should now read “Whenever the total failed pipeline runs metrics is greater than 0.”
5. Under *Evaluated based on*, set the *Aggregation granularity* to “5 minutes” and the *Frequency of evaluation* to “Every 5 minutes.” This combination of settings means that, every five minutes, Azure Monitor will count the number of pipeline failures that occurred in the previous five minutes. If that number exceeds zero, an alert will be raised. Click *Done* to create the condition.
6. The alert rule’s actions describe what Azure Monitor will do when the alert is raised. Actions are defined in an *action group* – a reusable collection of *notifications* (e.g., emails or SMS messages) and automated system *actions* that can be defined in a variety of ways (such as an Azure Function or a Logic Apps workflow). Click *Add action groups* to create a new action group.
7. In the *Select an action group...* blade, click + *Create action group*. Select the subscription and resource group that contain your ADF instance, then under *Instance details*, enter the value “NotifyMe” for both the *Action group name* and *Display name*.
8. Click *Next: Notifications* to move on to the *Notifications* tab. Select “Email/SMS message/Push/Voice” from the *Notification type* dropdown, then complete at least one option on the displayed blade. In practice, your choice of notification type will be determined by the urgency of the alert you are creating, but note that not all options are available in every Azure region. Click *OK* to close the blade.
9. Supply a *Name* for the new notification, then click *Review + create*. (For the purpose of this exercise, you need not create any additional notifications or actions.) Click *Create*.

10. Finally, back in the *Create alert rule* blade, complete the *Alert rule details* section. Specify an *Alert rule name* of “High failure rate,” select the same resource group as your data factory, and choose the *Severity* level “Sev 2.” Ensure that the *Enable alert rule upon creation* checkbox is ticked, then click the *Create alert rule* button to create the new rule.

Tip Severity levels range from “Sev 4” (least severe) to “Sev 0” (most severe) and have the following meanings: “Verbose” (4), “Informational” (3), “Warning” (2), “Error” (1), and “Critical” (0).

The failure rate being detected here may not resemble a production workload, but is chosen to illustrate the approach – you will now start to receive alert messages whenever the “High failure rate” alert condition is met. The “RunEvery8Minutes” trigger calling the failing pipeline does not run in every five-minute window, so this may not be immediate. Additionally, new metric-based alerts take a few minutes to come into effect in Azure.

The approach taken here to configuring alerts uses the Azure portal. Metric-based alerts for Azure Data Factory can also be created and managed from the *Alerts & metrics* page of the ADF UX monitoring experience.

Configure Log-Based Alerts

Receiving alerts in response to overall system health is useful from the general perspective of managing your Azure Data Factory service, but you may also have more specific notification requirements. For example, if a certain pipeline performs a particularly important data transformation process, you may wish to find out as soon as possible if it fails. In this section, you will construct an alert using a custom log query to detect failures of a specific pipeline.

1. Open your Log Analytics workspace in the Azure portal, then in the *Monitoring* section of the sidebar, select *Alerts*. As before, click *+ New alert rule* to create a new rule.
2. Accept the default *Scope* identifying the Log Analytics workspace, then click *Add condition* to create a new alert condition.

3. Notice that the set of signals displayed in the *Configure signal logic* blade differs from previously – the available signals of type “Metric” now relate to Log Analytics metrics, and a new signal type of “Log” is present. Choose the “Log” signal named “Custom log search.”
4. As the name suggests, “Custom log search” enables you to create alerts based on the results of custom log queries. Scroll down the blade to the *Search query* pane and enter the Kusto query given in Listing 12-1.

Tip Notice that Listing 12-1 has no terminating semicolon. Custom log search reduces its query result set to a count of matching rows by piping the query into Kusto’s count operator. If the input query is explicitly terminated, appending “| count” results in an invalid expression.

5. Set the *Alert logic’s Threshold value* to zero, and specify evaluation based on a *Period* of 10 minutes and a *Frequency* of 5 minutes. This means that, every five minutes, the custom log query will be executed over rows with a “TimeGenerated” value within the prior ten minutes.
6. Click *Done* to add the new alert condition, then click *Add action groups* to add an action group. Select the “NotifyMe” action group you created earlier.
7. Name the new alert rule and set its severity to “Sev 1,” then click *Create alert rule*.

Listing 12-1. Kusto query returning recent failed runs of a specific pipeline

```
ADFPipelineRun  
| where PipelineName == "ImportSTFormatFile"  
| where Status == "Failed"
```

The new rule reports an error on more specific conditions: whenever the named pipeline fails. Windowed queries like this cannot be guaranteed to catch every error, because there is a delay between events occurring in ADF and their being received by the Log Analytics service. The alert frequency is intended to allow detection of failures occurring in the previous five minutes. Its period of 10 minutes allows more time for late-arriving errors, but failure records that take more than 10 minutes to arrive will not cause an alert.

The alerts you configured in the preceding sections were based on metrics emitted directly by your data factory and on logs sent to Log Analytics. Your ADF diagnostic setting also sends data factory metrics to Log Analytics, so you could choose to build ADF metric alerts there instead. The disadvantage of doing this is that it subjects metrics to the same latency effects as other log records. Note also that ADF metrics sent to Log Analytics appear as log records in the AzureMetrics log table – metrics reported in Log Analytics are those emitted by that service and not by Azure Data Factory.

A more direct approach to raising alerts is to create an Azure Function that performs notifications. While creating Azure Functions is outside the scope of this book, doing so allows you to issue notifications from within an ADF pipeline, for example, by connecting an Azure Function pipeline activity via a Failure activity dependency.

Deactivate ADF Triggers

When you have finished exploring monitoring features, deactivate the two triggers you activated at the beginning of this chapter. Remember to publish your changes to deactivate the triggers in the published environment.

Chapter Review

In this chapter, you used factory resources developed in earlier chapters to generate a workload, then monitored it using a variety of measures and tools:

- Logs and metrics as indicators and measures of system activity and health
- The ADF UX monitoring experience and Log Analytics to inspect and analyze logs and metrics

- Azure Monitor's alerting support to receive notifications of potential issues indicated by log and metric analyses

These tools enable you to gain insight into the workload and health of your Azure Data Factory instance, allowing you to detect and resolve issues as they arise.

Key Concepts

Key concepts introduced in the chapter include

- **Pipeline annotation:** A label, added to a pipeline, that appears in the log of subsequent pipeline runs and can be used to filter or group log data. Multiple annotations can be added to a pipeline.
- **Trigger annotation:** A label, added to a trigger, providing functionality analogous to a pipeline annotation.
- **Activity user property:** A name-value pair, added to a pipeline activity, that appears in the log of subsequent pipeline runs. Multiple user properties can be added to an activity. The Copy data activity supports two auto-generated properties that identify its runtime source and sink.
- **Azure Monitor:** Monitoring service used to collect, analyze, and respond to data from Azure resources.
- **Metric:** Automatically maintained count of a given system property over a period of time, emitted to and logged by Azure Monitor.
- **Log Analytics:** Azure Monitor component that enables sophisticated analysis of system logs and metrics.
- **Log Analytics workspace:** Identified Log Analytics provision, to which Azure resource logs and metrics can be sent for analysis and longer-term storage.
- **Diagnostic setting:** Per-resource configuration information identifying log data and metrics to be sent to other storage services, for example, a Log Analytics workspace.
- **Kusto:** Query language used to interrogate data stored in Log Analytics and Azure Data Explorer.

- **Tabular expression statement:** Kusto query expression that returns a result set. Every Kusto query must contain a tabular expression statement.
- **Log Analytics workbook:** A notebook-like interface for querying Log Analytics data, allowing code and text cells to be interleaved to create narrative reports.
- **Azure Data Explorer:** Analytics service for near real-time, large-scale analysis of raw data. Like Log Analytics, the service accepts read-only queries written in Kusto.
- **Alerts:** Azure Monitor supports the raising of alerts in response to configured metric or custom query output thresholds.
- **Alert rule:** Information that defines an alert – its scope (what to monitor), its conditions (when an alert should be raised), and its actions (who to notify and/or what to do when an alert is raised).
- **Signal:** Measure used to evaluate an alert condition.
- **Action group:** Defines a collection of notifications and actions, used to specify the action component of an alert rule.

For SSIS Developers

Many of the logging and metric concepts presented here will be familiar to SSIS developers. Azure resource metrics are very similar to the notion of Windows performance counters and serve a comparable purpose. Just as SSIS emits performance counters specific to Integration Services, so does Azure Data Factory emit ADF-specific metrics.

Automatic logging of activity will be familiar, particularly to users of the SSIS catalog, and automated log truncation may be a welcome change. Conversely, if long-term storage is required, ADF requires additional functionality in the form of Log Analytics or blob storage for indefinite retention. Users of the SSIS catalog used to being able to query and analyze SSIS logs will find comparable functionality offered by Log Analytics queries and workbooks.

Index

A

Acme Boxed Confectionery (ABC), 28
Aggregate transformation, 208, 210, 214, 215
Alerts, 329
 log-based, 325–327
 metric-based, 323–325
Amazon Web Services (AWS), 25
Apache Parquet, 73
AutoResolveIntegrationRuntime, 37, 217–221, 223, 239
Azure Active Directory (AAD), 3, 20
Azure Data Factory (ADF)
 ARM template, expsort, 259
 ARM template, import, 260, 261
 Azure portal, 2–4
 blade, 8
 create account, 2
 create resource group, 4–6
 create resources, 7
 definition, 1
 deployment parameters, 262, 263
 feature branch workflow
 definition, 272
 JSON, publish resources, 275
 powershell, 275, 276
 resource dependencies, 277
 utilities, 274
 git repository, link
 Azure repos, 13–15
 data factory, 16, 17

instance, 278
 ADF UX, 253
 ARM, 279
 pipelines, 253
 publish resources, 254, 255
 run pipelines, 255, 256
management hub, 21
pipelines, 1
product environment, 257, 258
publishing
 ARM templates, 263
 DevOps service connection, 264, 265
SSIS developers, 22
UX
 definition, 9
 navigation header, 10
 navigation sidebar, 11, 12
 web-based IDE, 17, 18
Azure Data Factory User Experience (ADF UX), 2, 9, 21, 79
Azure DevOps pipelines
 add deployment task, 268–270
 definition, 264, 265
 trigger automatic deployment, 270–272
 YAML, 265, 266
Azure DevOps Services, 13
Azure Event Grid, 290, 304
Azure Integration Runtime, 217
 AutoResolveIntegrationRuntime, 217, 218

INDEX

Azure Integration Runtime (*cont.*)

- create, 219, 220
- databricks cluster TTL, 220, 221
- geography of data movement, 221
- identify the copy, 221, 222
- revise the sink, 222, 223
- use, 221

Azure Key Vault, 119

- ADF linked service, 118, 119
- create, 114–116
- create secret, 120
- grant access, 116, 117
- linked services, 119, 120

Azure Pipelines, 263, 264, 267, 280

Azure portal, 2, 3, 5, 7, 11, 18, 20

Azure Resource Manager (ARM) templates, 259, 279

Azure SQL Database

- create database, 45–49
- create database object, 49, 50
- definition, 45
- pipeline
 - create/run, 56, 57
 - database linked service/dataset, 51–55
 - DelimitedText file dataset, 55, 56
 - multiple files, 59–61
 - truncate, 61, 62
 - verify results, 58, 59

Azure Storage

- create account, 23–26
- definition, 23
- explore storage, 26, 27
- upload sample data, 27, 28

Azure-SSIS Integration Runtime, 231

- create, 231–234
- deploy, 234, 235
- run in ADF, 236, 237
- stop, 237, 238

B

Blob storage, 37, 230, 239, 242

C

Collection reference property, 7

Conditional activities, 157

- divert error rows, 157–160
- load error rows, 161
 - if condition activity, 162–164
 - new sink dataset, 161
 - revise source dataset, 161
 - run pipeline, 164, 165
- switch activity, 165–167

Continuous integration and continuous delivery (CI/CD), 264, 280

Copy data activity

- datasets, 78
- features, 45
- JSON dataset, 74
- map source/sink schemas, 63–66
- Parquet dataset, 74, 75
- performance settings
 - degree of copy parallelism, 77
 - DIUs, 76, 77

semi-structured data,

- Azure SQL DB
 - create pipeline, 68
 - features, 67
 - JSON file format, 68
 - report files, 67
 - schema drift, 70, 71
 - schema mapping, 68, 69
 - set collection reference, 69, 70
 - type conversion, 72
- SSIS developers, 81
- storage formats/services, 78, 79

- transformation pipeline, create/run, [75, 76](#)
- Copy data tool, [23, 28, 29, 31, 32, 43, 45](#)
- ## D
- Data factory metrics, [323, 327](#)
- Data flow activity
- debugging, [182, 183](#)
 - execute, [198](#)
 - create pipeline, [199, 200](#)
 - execution output, [200, 201](#)
 - log completion, [201, 202](#)
 - filter transformation, [188–190](#)
 - run pipeline, [181](#)
 - transformation, [184–188](#)
- Data integration units (DIUs), [76, 78](#)
- Datasets parameters, [121–123](#)
- Dependency condition interact, [149](#)
- combine condition, [149](#)
 - completion condition, [151, 152](#)
 - errors raised, [156, 157](#)
 - failed condition, [150](#)
 - multiple activities, [151](#)
 - pipeline outcome, [152–155](#)
 - skipped condition, [150](#)
 - variable activity, [149](#)
- Derived Column transformation, [194–196](#)
- Dimension maintenance data, [210, 211](#)
- ## E
- Escaping @, [108](#)
- Event-based trigger
- create, [290–293](#)
 - grid resource provider, [289, 290](#)
 - run, [293–295](#)
 - trigger-scoped system variables, [295](#)
- ## F
- Factory logs
- activity user property, [312](#)
 - inspect trigger runs, [308, 309](#)
 - pipeline annotation, [312, 313](#)
 - pipeline runs, [308–311](#)
 - user properties, [313, 314](#)
- `$$FILEPATH`, [117](#)
- ForEach activity, [169–172, 175–177, 179](#)
- ## G
- Global parameter, [139–142, 259, 260, 262, 270, 277](#)
- Google Cloud Platform (GCP), [25](#)

INDEX

H

Handy Candy data, 241, 245

I, J, K

Infix operators, 107, 111, 190

Infrastructure as a service (IaaS), 20

initial @ symbol, 88, 108

Integrated development environment (IDE), 9, 21

Integration runtime (IR), 217, 238

Interchangeable datasets, 122

Interim data types (IDTs), 72, 81

Iteration activities, 167

ForEach, 169–172

metedata, 167–169

parallelizability, 172–175

Until, 175–176

L

Lighter-touch approach, 323

Linked service parameters, 127

create, 127–131

increase reusability, 131, 132

JSON configuration, 129

parameterizing ADF datasets, 135–137

user new dataset, 132, 133

Log Analytics, 316

definition, 319

queries, 319–321

workbooks, 321, 322

Logs and metrics, 316–318, 320, 323,

327, 328

blob storage, 318, 319

diagnostic settings, 316–318

log analytics, 319–323

Lookup activity

breakpoints, 98, 99

configure, 96–98

create database objects, 94–96

run pipeline, 101, 102

Stored procedure activity, 100, 101

values, 100

Lookup transformation, 191

add, 193, 194

add data stream, 191–193

M, N, O

Monitoring

deactivate ADF triggers, 327

factory activity, 307, 308

factory logs, 308

factory metrics, 314, 315

P, Q, R

Parameterized dataset

create, 123, 124

reuse, 126, 127

use, 124, 125

Parameters, 113

Per-file pipeline, 145, 146

failure, 147, 148

parameters, 149, 150

Pipeline, 1, 19, 21, 23, 35, 83

activities toolbox, 35

datasets, 34, 35

debug mode, 40, 41

definition, 36, 42

execution results, 42

factory resources, git, 39

integration runtimes, 37

- linked services, 33
 - Pipeline annotation, 311–313, 328
 - Pipeline parameters, 133
 - activity, 137–139
 - create, 132
 - definitions, 134
 - parallel, 139
 - run, 135–137
 - pipeline().property, 105
 - Platform as a service (PaaS), 20, 45
 - Power Query
 - ADF benefit, 241
 - ADF create mashup, 241–243
 - ADF features, 250
 - definition, 241, 251
 - editor, 243, 244
 - Handy Candy data, 245–248
 - pipeline, 248–251
 - PowerShell module, 280
 - Product dimension, 202
 - create datasets, 204
 - create table, 203
 - maintenance, 203
 - aggregate, 208, 209
 - exists transformation, 209, 210
 - use locals, 205–207
- S**
- Schedule trigger
 - advanced options, 288, 289
 - create, 281–283
 - deactivate, 287, 288
 - definitions, 284, 285
 - monitor, 286
 - publish, 285
 - reuse, 283, 284
 - Select transformation, 196, 197, 214
- Self-hosted integration runtime, 224
 - create, 224, 225
 - create data factory, 225, 226
 - link, 226, 227
 - use, 227
 - copy file system, 229–231
 - create file system, 228, 229
 - create link, 227, 228
 - Server Management Studio (SSMS), 18, 45, 49
 - Sink transformation, 197, 198, 210, 211, 213, 214
 - Software as a service (SaaS), 20
 - SSIS package parameters, 142
 - String interpolation, 107
- T**
- Tenant, 3, 4, 11, 13, 14, 16, 20
 - Trigger
 - definition, 281
 - pipelines, 302–304
 - publish automatically, 302
 - run, 303
 - SSIS developers, 305
 - types, 281
 - Trigger annotation, 313, 328
 - @triggerBody() function, 295
 - Tumbling window triggers
 - advanced features, 301, 302
 - create copy pipeline, 297–299
 - definition, 281, 295
 - monitor trigger runs, 299–301
 - prepare data, 296, 297
- U, V, W, X, Y, Z**
- Until activity, 175, 176