Course Overview

Course Overview

Hi everyone. My name is Marcelo Pastorino, and welcome to my course, Deploying Data Pipelines in Microsoft Azure. I am a software developer and solutions architect with 20 years of commercial experience designing and developing software, services, and applications that run in the cloud, web, and mobile devices. Continuous integration, delivery, and deployment are a set of practices that allow software developers to continuously deliver value and also a great way to accelerate the feedback loop with customers. Did you know that that data engineers working with Azure Data Factory can take advantage of the same methodologies to deliver robust, well-tested data pipelines to production? In this course, you are going to learn about such practices and how to incorporate them into your Azure Data Factory pipeline creation process. Some of the major topics that we will cover include creating the infrastructure needed to support multiple deployment environments, integrating Azure Data Factory with a source controlled system, deploying data pipelines using Azure Data Factory visual tools and ARM templates, and also deploying data pipelines using a fully automated release pipeline in Azure DevOps. By the end of this course, you will have the skills and knowledge to apply CI and CD practices to your Azure Data Factory pipeline creation process. Before beginning the course, you should be familiar with Azure Data Factory and other Azure services, such as Azure Storage and Azure Key Vault, as well as having some familiarity with Git version control system. I hope you'll join me on this journey to learn with the Deploying Data Pipelines in Microsoft Azure course, right here at Pluralsight.

Getting Started Deploying Data Pipelines in Azure

Introduction

Hello, my name is Marcelo Pastorino, and I want to welcome you to my course, Deploying Data Pipelines in Microsoft Azure. In the first module of the course, we seek to answer two important questions. First, we want to understand what continuous integration and continuous delivery mean in the context of Azure Data Factory. Second, we want to learn the reasons why someone who's already creating pipelines in Data Factory should consider incorporating continuous integration and delivery to its workflow. Along the way, we discover two methods to add continuous integration and delivery methodologies to our pipeline creation process. In this course, I assume that you're well versed in Azure Data Factory. If you want to take this course but are new to Azure Data Factory or want to refresh your knowledge about it, I invite you to watch my course, Integrating Data in Microsoft Azure, first. Let's begin.

Learning the Advantages of Integrating GIT with Azure Data Factory

In traditional software development, continuous integration is a development methodology that involves integrating code changes into a shared repository on a frequent basis. There are many advantages to using this methodology, such as smaller code changes, early bug detection, faster release rates, and so on. Continuous delivery is the practice of ensuring that software is deployable at all times. At a very high level, the process looks like this. A developer makes a change to the source code, usually working in what's called a feature branch, which is a copy of the software being changes. Typically, only the developer working on the new feature has access to this feature branch. This is great, as the developer works in isolation from other developers, and the changes they are making to the version of the code in their own feature branches. Once the developer is satisfied with the changes, a local test is performed to make sure the code works as expected. At this point, the developer integrates the new code in the main collaboration branch, merging these changes with other changes made by other developers. The integrated code then can be pushed to other environments, usually called the staging or quality assurance environment, where other types of tests are performed to make sure the integrated code changes satisfy the requirements. The goal is to make sure that the code is shippable at all times. From here, the team may decide to promote the changes in this environment to the production environment on a manual or automated basis, depending on the followed methodology. Of course, this is a very rough description of the process, and it may change from team to team and based on the methodologies in use, but you get the idea. Software development teams have been taking advantage of continuous integration, continuous delivery, and continuous deployment for some time now to deliver better code faster. Wouldn't it be great if data engineers working on data integration pipelines could take advantage of similar methodologies to develop, collaborate, and deploy such pipelines? If you are a data engineer working on the Azure ecosystem and creating data integration pipelines with tools such as Azure Data Factory, now you can. In Azure Data Factory, continuous integration and delivery involves moving data factory pipelines between environments from the main collaboration environment, usually referred to as the development environment, to the staging environment and then to the production environment. But why would you like to do this? There are several advantages to using this methodology, like being able to collaborate with colleagues in the same data pipeline and testing data pipeline changes in different environments with different datasets and requirements. This methodology ultimately contributes to promote collaboration and deliver well‑tested data pipelines to production. As you already know, Azure Data Factory is a versatile service, and it allows us to achieve these goals in more than one way. We'll learn about this and more in the next clip.

Learning Two Different Ways to Tackle the Problem

It is good to know that we can achieve continuous integration and delivery in Azure Data Factory in two ways. One way is to use Data Factory visual tools integration with Azure resource manager templates. I call this the UI-based method. We can take advantage of the Data Factory visual tools capabilities to import and export and Resource Manager templates and a version control system such GitHub or Azure Repos to deploy pipelines between environments. This is the basic way to achieve continuous integration and delivery in Azure Data Factory, and it works great. The only caveat is that it is somewhat a manual process involving a few clicks here and there, but it is an excellent method if your team is not too large or you don't need a fully automated system. We learn how to achieve continuous integration and delivery in Azure Data Factory using the UI-based method later in the course. Now, the real beauty and appeal about continuous integration and delivery lies in the fact that the process can be fully automated. In this course, you'll learn how to automate the deployment of data pipelines between different environments, combining the power of Azure Data Factory and Azure DevOps. Whether you follow the UI-based or automated process, you can be sure that you'll end up with more resilient data pipelines in production.

Summary

In this module, we learned about continuous integration and delivery practices in traditional software development, and learned with these methodologies mean in the context of Azure Data Factory. We also learned about the advantages of having multiple Data Factory environments and how that can help us to create more resilient and well-tested data pipelines. In Azure Data Factory, we can achieve continuous integration and delivery in two ways. One is by using Data Factory UX integration with the Azure Resource Manager templates. I personally call this the UI-based method. The second way is by combining the power of Azure Data Factory and Azure DevOps to create a fully automated delivery pipeline that can deploy data pipelines automatically. In the next module, we create the data pipeline deployment infrastructure we'll use throughout the course. Stay tuned.

Creating the Data Pipeline Deployment Infrastructure

Introduction

In this module, we design and create some of the infrastructure we are going to use during this course. In Azure, we'll create the following resources, resource groups, storage accounts, Key Vaults, data factories, and Azure DevOps infrastructures such as repos, release pipelines, and so on. To implement continuous integration and delivery, we need to support different environments for different stages of the data pipeline development process. One of the critical factors in creating well-designed environments is creating one set of resources per environment. In this course, we are going to support three environments, development, staging, and of course production. The development environment is where most of our Azure Data Factory pipeline work happens. This is our collaboration environment where data engineers comes together and create pipelines collaboratively. Then we have the staging environment. Some people call it the quality assurance or testing environment. And we use it to test our data pipelines and make sure all works as we intend it to. Finally, we have the production environment. This is where the real data transformation happens and where the data pipelines we created will ultimately run. One thing to notice is that in real-world scenarios, these environments have different requirements. For example, we do not want to develop and test our data pipelines with production data or spend unnecessary time and resources in testing data pipeline functionality that could have been tested very quickly otherwise. To support this, we need to create a set of dedicated resources for each environment. So for example, we'll create three Azure resource groups, three Azure Storage accounts, three Azure Data Factories, and so on, one for each environment. All right, enough chitchat. Let's start creating some infrastructure, shall we?

Creating Azure Resource Groups

All right, welcome to the first demo of the course. In this clip, we create our first set of infrastructure, Azure resource groups. If you are not familiar with Azure resource groups or need a refresher, please check this link. I have logged into the Azure resource portal already, and I am looking at the resource group blade, ready to create our first group. To create one, please click the Add button at the top. As you already know, creating resource groups is probably one of the easiest things you can do in Azure, so not much explanation is needed during this process. What is imperative to understand is that each resource group will contain all the resources that belong to each environment. This is the resource group for the development environment. Okay, let's create the resource group by clicking on the Review + create button. This was fast. The Azure service already created the development resource group for us. Okay, very quickly, I am creating the other two resource groups. Please bear with me for a moment. This one is the one for the staging environment and will contain all the resources needed for us to test our data pipelines properly. In the next clips, we start to add resources to each resource group. For now, let's click on the Review + create button to finalize creating the resource group. Right, while the Azure service provisions the staging resource group, let's create the production one. Notice how the names of all resource groups follow a pattern and contain a reference to the environment at the end. I use the suffix dev for development, stg for staging, and prd for production. I will follow this pattern when I create other services in this course, as this strategy becomes really handy later when we start to automate things. Very good, we finished creating all three resource groups. A moment ago,

Creating Azure Storage Accounts

We created or set up resource groups, but they look painfully empty, so let's remedy that by creating one Azure Storage account per resource group. I start by navigating to the storage account blade in the Azure portal. As you can see, we have no storage accounts created at the moment. If you are unfamiliar with Azure Storage or need a quick refresher about this service, please check the link for more information. I want to start by creating the development environment storage account first. Notice how all services that I created belong to the same subscription called Evangeloper. All right, since the storage account belongs to the dev environment, let's associate it with the dev resource group. That just makes sense, right? Please notice the end of the storage account name. It ends in dev for development. The staging storage account will end on stg, and the one for production will end on prd. Of course, each storage account will be associated with the proper resource group, as we did with the development account. Another thing to notice is that every single service that I create is hosted at the same location, West Europe. For the Performance setting, I am leaving the default option. Now onto the Account kind setting. I am choosing BlobStorage for this one. The last setting I want to change is the Replication setting. For the purposes of this course, we don't need the powerful geo‑redundant storage that Azure provides. The locally‑redundant option is more than enough for our purposes. As for the rest of the options, I am leaving the defaults. Okay, I think that we are all set, and we should probably create the account at this point. The validation passed, and all we need to do is to click on the Create button to initiate the process. The Azure service is kind enough to work behind the scenes creating the storage account for us. This process usually takes just a few seconds. Once the deployment is complete, we need to access the resources, as we need to create a couple of storage containers. For that, click on Containers at the bottom of the page. As I said, we need to create two containers. The first one is called sink. I really don't want anyone who is not an authorized person to access this container, so I am selecting Private to restrict anonymous access. The second container is called staging, and it's also configured with a private access level. The data pipeline we create later in the course will move files from the sink to the staging container. All right, very good, we finished the creating the storage account for the development environment. The staging and production storage accounts are 99% exactly the same as the one we just created. I appreciate your time so much, so we won't repeat the same process all over again. There are only two settings that slightly change for each storage account. The staging storage account is named like this. Notice the stg suffix at the end of the name. The other difference is that the staging storage account is associated with the staging resource group, of course. That's all. We also followed the same logic for the production storage account, as you see on the screen, so this is the result. We have three blob storage accounts. All of them configure precisely the same and associated with a different resource group for the development, staging, and production environments. All right, in the next clip, we create an Azure Key Vault account for each environment. See you in a moment.

Creating Azure Key Vaults

Azure Key Vault allows us to safeguard keys and secrets used by your services in a very convenient way, and we use it for this purpose in this course. If you want to discover more about this service, please visit this link. In this demo, we are creating one Azure Key Vault per environment. Let's start with a key vault for the development environment. Once we input all the different settings needed to create this service, let's click on the Review + create button. The Azure service validates our request and creates the key vault account. This process usually takes only a few seconds. Later in the course, we create a secret and set access policies for each of the Azure Key Vault accounts we are creating. Let's repeat this process for the staging account. All settings are the same except for the resource group associated with the account and its name. I am clicking on the Review + create button to instruct the Azure service to create the service for us. All right, we followed the same logic to create the production account. As I mentioned earlier in the course, the goal is to create one set of services for each environment so we can development, test, and run our data pipelines with different sets of data.

Summary

In this module, we created a set of resources to develop, test, and run our data pipelines with different sets of data in three district environments. We started by creating three resource groups that bundle together the other services needed in each environment. We also created a naming convention for our services that will allow us to fall into a bit of success later in the course when we create automated deployment pipelines. Finally, we created blob storage and key vault accounts to store data and safeguard our service secrets, respectively. In the next module, we create three Azure data factories, one for each environment, configure them to access other services, and create our first data pipeline in the collaboration environment.

Creating Azure Data Factory Environments

Introduction

In this module, we continue creating the infrastructure needed to support our multiple environments to deploy data pipelines in Azure. Now it's time to create Azure Data Factories, configure them so they can talk with other services like Azure Key Vault, and build our first data pipeline in the collaboration environment. By the end of this module, we'll have everything in place to start deploying data pipelines in Azure. Let's begin.

Creating Azure Data Factories

If you're watching this course, the chances are that you're very well versed in all things related to Azure Data Factory. But if you happen to need a quick refresh about this service, I invite you to watch my course, Integrating Data in Microsoft Azure. You are looking at the Data factories page in the Azure portal. To create our first data factory, let's click on the Add button. I assume that you've created data factories before, so I won't bore you with the details. The most important things to notice are the data factory name and its suffix and the selected resource group. Okay, I am disabling Git integration on this screen, as I don't want to configure it right now. Don't worry; we do this later in the course. For now, let's create the data factory by clicking this button. The Azure service takes a few moments to create this service. This isn't as quick as creating our resource group, but the operation shouldn't take more than a few seconds. In the meantime, let's save some time and start creating the other two data factories. Here are the settings for the staging data factory. Please notice the name suffix and the resource group name. You know, we won't enable Git for this data factory at all. We'll only configure Git integration in the collaboration data factory later in the course. Okay, let's create the service. You know, time is money, so while the Azure worker services are busy at work creating this service, we are going to specify the details for the production data factory next. Here are the settings. Again, notice the name, the resource group, and the fact that the Git integration is not enabled for this data factory at all. Cool, let's create our last data factory. Our deployment is underway. Okay, let's refresh this screen. Hopefully, all three data factories have been created by now. They surely are. In the next clip, we configure our Azure Data Factories to access Azure Key Vault.

Configuring Azure Data Factory to Use Azure KeyVault

In previous clips, we created a Blob Storage accounts, Azure Key Vault accounts, and data factories. Now we are going to configure these services so they can talk to each other and also learn how to store sensitive information in our Azure Key Vault accounts. Let's start by navigating to the storage account page and accessing the development blob storage account. If we go to the Access keys settings page, we'll find the connection string needed to connect to this service. We are interested in the connection string for key1. We want to copy it to the clipboard, as we'll need it in a minute. All right, let's open the Key Vaults page and access the key vault associated with the development environment. We want to add a secret to this key vault, specifically the blob storage connection string we copied to the clipboard moments ago. But what's the purpose to store this connection string inside a key vault account? Right, let me explain. Later in the course, when we create our data pipeline, we need to create a data source to access data in our blob storage container. Azure Data Factory is kind enough to allow us to configure the resource directly, specifying the Azure Blob Storage connection string in the settings. Still, Microsoft does not endorse this practice, as it leaves the connection string exposed in plain text. We want to protect this information. A best practice is to safeguard the connection string encrypted in a service such as Azure Key Vault and provide access to it only to authorized services. That's what we will do. We want to secure the connection string so only our Azure Data Factory pipelines have access to it. Now that we have a better understanding of the use case, let's go ahead and create the new secret. We want to select the manual upload option and specify a name for it. I am choosing storage‑access‑key for the secret name. The name is really important, as we'll use it to get the secret later on. One other important thing to remember is that when we replicate this secret in other Azure Key Vault accounts, we need to use the same name. This will be very useful when we automate things later in the course. Okay, I am pasting the blob storage connection string in the Value field. Finally, I am clicking on the Create button to create the secret. Here it is. The next step is to give our development data factory access to this key vault. We can do this by using Azure service principals and managed service identity. Each Azure Data Factory has an associated service identity that we can use to allow access to other services. We can add the data factory managed identity in the Access policies settings page by clicking on the Add Access Policy link. Let's restrict Data Factory only to have access to Key Vault secrets by choosing this permission alone. By clicking on this link right here, we can specify which managed identity has access to this key vault account. I am interested in granting access to our development data factory only. Good, I am clicking on the Select button and adding the access policy. Before we move on, let's save it by clicking on the Save button on the top of the page. Sweet, our development environment Azure Key Vault is configured, and our Azure Data Factory should be able to query for its secrets. We need to repeat the process for the other two accounts. Let's navigate to the staging storage account so we can get its connection string. This connection string is of course different than the one for the development storage account; therefore, we need to create a new secret for it. Speaking about secrets, let's configure the staging key vault. The procedure is exactly the same as before. The secret name is also the same. What changes is the secret value. It stores the connection string for the staging blob storage account. Finally, we must configure the access policy in the same way as before, but this time we select the service identity for this staging data factory. All other settings remain exactly the same, as you can see on the screen. All right, let's save the access policy for the staging key vault then. Lastly, we need to configure the production key vault, but for the sake of brevity and to avoid repetition and also because there is no point in showing you the same steps again, I am going to skip this demo. It is enough to say that the production key vault contains a secret named storage‑access‑key that stores the value of the connection string for the production blob storage account. Also, the allowed service identity is the one for the production data factory, of course. Everything else was configured exactly the same. Very good. At this point, we have all in place to start creating our data pipeline. We'll do that next.

Creating an Azure Data Factory Pipeline

Okay, friends, now that we have all the infrastructure in place, let's create our data pipeline in Azure Data Factory. We are going to work in the development environment with the development data factory. In fact, all the work we perform in the data pipeline is going to be done in this environment. This is our collaboration environment. Let's click on the plus button and create a pipeline from a template. In the Filter box, search for move, and find the Move files template in the gallery. We'll use this template to create a pipeline that simply moves files from one blob storage container to another. In the DataSourceConnection drop‑down box, let's select New to create a linked service and a dataset to connect to our blob storage account. I am providing a descriptive name for the linked service. Let's call it StorageLS. As I stated, we need to connect to Azure Blob Storage account, so we need to select the appropriate connection type in this box right here. We have a connection string that allows us to access the blob storage account already secured by our Azure Key Vault service. We are going to take advantage of the data factory integration and connect to key vault to get the secret containing the connection string. To do that, we need to create another linked service to connect to Azure Key Vault. Let's name this new linked service AzureKeyVaultLS. As you know, the Azure Data Factory UX comes with fantastic integrations out of the box that allow us to select services with just a few clicks. Since I am working with the infrastructure that belongs to the development environment, it just makes sense to select the Azure Key Vault for this environment as well. Friends, that's all it takes to integrate Azure Key Vault in Data Factory. Super easy. Before we move on, let's test the connection and create the linked service. Okay, we are back creating the linked service for the blob storage account. Notice the key vault linked service already populated for us. Now we need to write the secret name in this box. Of course, we are interested in the storage access‑key‑secret that contains the connection string to the blob storage account for the development environment. Thanks to Azure Data Factory's integration with Azure Key Vault, we are not disclosing this information on this page. We don't even need to know what the connection string looks like, and we only care that there is a secret in Key Vault that contains this information. As for the rest of the settings, let's use the defaults and test the connection right away. Cool, the connection succeeded. Let's create it. The new linked service is already selected for the DataSource\_Folder and the file connections. We need to do the same for the data destination, as we want to copy files between containers in the same blob storage account. Awesome, we finished configuring the template. When I click on Use this template, the data factory service will create the linked services, three datasets, and one pipeline for us. Here's the pipeline with three actions designed to copy files from one blob storage container to another. If we click on the Parameters tab, we find two parameters, the source and the destination blob storage container names. If you recall from a previous clip, each Azure Storage account has two containers, one named sink and the other one staging. Then based on these settings right here, the pipeline moves files from the sink to the staging container in the blob storage account. Easy peasy. These are the three datasets that were created based on the template. All of them use the same linked service to access the blob storage container. Here are the two linked services we created moments ago. Notice the Azure Key Vault linked service URL pointing to the service associated with the development environment. And here's the blob storage linked service. It is using the native integration with Azure Key Vault to retrieve a secret named storage‑access‑key. As we know, the secret contains the connection string to the blob storage account associated with the development environment. Okay, we are ready to test this pipeline and make sure it works. You are now looking at Azure Storage Explorer. If you are new to this tool and wish to learn more about it, please use this link. In a nutshell, Azure Storage Explorer is a free tool to manage our Azure Cloud Storage resources. I already configured it to get access to our three Azure Storage accounts. This is the blob storage account for the development environment. It has two containers, sink, which currently contains 10 files, and staging, which is currently empty. The same goes for the blob storage account in the staging environment, the same containers, the same file configuration. And then we have the production storage account, configured exactly the same as the other two. In a real‑world scenario, these three environments would contain different amounts of data. Probably your production environment could contain thousands of files, but for our purposes, a few files are more than enough. Our goal here is to run the pipeline and move these files from the sink container to the staging container. Having said that and without further ado, let's run the development pipeline. If everything goes according to plan, the files you saw sitting in the sink container will be moved to the staging container. Good, it looks like the pipeline just finished executing. Let's refresh this view. If everything went according to plan, the files should be gone from this container. Sure, the files are gone, no trace of them here in this container at all. They magically reappeared in the staging container. Our pipeline moved all 10 files to this container as expected. Our development pipeline is working great. To finalize this demo, we are going to configure an automated trigger. I bet you've created triggers before, so there is no need for me to bore you with these details. All I want is this pipeline to run every day, 1 minute before midnight. As you know, we need to provide the parameter values here, so I am adding the source and the destination container names as we did before. With the trigger in place, all that is left for us to do is to validate and publish the changes. We have come a long way. Now we have everything in place to start learning about continuous integration and delivery in the context of Azure Data Factory.

Summary

Everything single action that we have performed so far during this course has been designed to allow us to fall into the pit of success when we start adding continuous integration and delivery to our workflow. In the next module, we take the first step to do so by configuring a source‑controlled system and integrating it in Azure Data Factory. The best is yet to come. Stay tuned.

Integrating Azure Data Factory Pipelines with Source Control Using Azure DevOps

Introduction

One of the key components in any continuous integration and delivery workflow is the source‑controlled system. In this course, we are integrating our Azure Data Factory with Azure Repos, and use Git to source control our data factory code. We start by creating a new project in Azure Repos to host our data factory definitions, and then we learn how to integrate both services easily. Azure Data Factory abstracts away the most complicated aspects of dealing with a source‑controlled system and Git in general, so we can concentrate on doing what we do best, creating powerful data integration pipelines in Azure.

Introducing Azure DevOps and Configuring Azure Repos

You are looking at Azure DevOps services, and you can access it by navigating to this URL right here. Azure DevOps allows developers to collaborate on code development and build and deploy applications. From the Azure Data Factory perspective, this service allows us to collaborate on data pipeline development and deploy these data pipelines to different environments like development, staging, and production. Azure DevOps is a mixture of features and services, and we will take advantage of two of them in this course, Azure Repos and Azure release pipelines. Azure Repos is a version‑controlled system that we use in this course to manage our code. Hold on. What code, you ask? I don't deal with code; I deal with data pipelines in Azure Data Factory. You are right, but Azure Data Factory infrastructure and data pipelines can also be expressed as code using ARM templates. Don't worry; this is, or at least can be, transparent for us. We learn more about ARM templates in the context of Azure Data Factory later in the course. Then we have Azure Pipelines. It is a cloud service that we use in this course to automate the deployment of our data pipelines to different environments such as development, staging, and production. If after this course you wish to review Azure DevOps documentation, please follow this link. I have an Azure account and organization already created, so now we have to create a new project. Let's click on the New project button and give this project a name. We can either host public or private repositories. They are just a collection of files, really. Our repository will host ARM templates, as I mentioned earlier. Today, I am creating a private repository, as I only want authorized users to have access to its contents. Finally, we'll leave the advanced default options as they are and click on the Create button. Here's our new project. There are different services that Azure DevOps offers. As I mentioned, we are interested in Repos to store our data pipeline code, and Pipelines to automate the deployment of our Azure Data Factory data pipelines. Very good, we have a new project on DevOps already set up. So far, so good. In the next clip, we integrate our Azure Data Factory with our new repository.

Integrating Azure Data Factory with Azure DevOps

We are in our development environment data factory, and we are ready to integrate this service with Azure Repos. The goal is to save the definition of the data factory as a series of files in our repository. Although we have three data factories, one for each environment, we only need to integrate Azure Repos with this data factory. This is the development environment, also called the collaboration environment. When we want to make changes to the data pipeline, we make them in this environment. Okay, let's do this right now. There are two ways we can achieve this, either by going to the main screen and clicking on the Set up code repository link, or in the pipeline itself by clicking on the Data Factory drop‑down box in the part at the top and selecting Set up code repository. We need to configure the repository settings, and we start by selecting the service that hosts our files, Azure DevOps, in this case. We have a fantastic integration experience that allows to authenticate and select the associated DevOps account automatically. Let's specify the project name that we want to connect to and the repository name that was automatically created for us in the previous clip when we created our project in DevOps. There is one important setting that we need to specify in this screen, and it's the collaboration branch. In source‑controlled systems, branching is the duplication of an object, such as a data pipeline definition file. It allows us to modify such objects in parallel and simultaneously in multiple branches. In simpler terms and for our purposes, branches allow multiple data engineers to work on the same data pipeline without interfering with each other's work. The collaboration branch is a specialized branch that Data Factory uses to merge the work of multiple data engineers in one place after they have performed their work. All right, as for the other options, let's work with the default values and click on the Apply button. Now that the integration between these two services is complete, Azure Data Factory is asking us to select a working branch, which is the branch that we want to do the work on. We can create new branches from this screen, but for now let's select master, or default collaboration branch. When we integrate Data Factory with a source control system, a second mode is activated. Besides the Data Factory mode, which is the default mode and the one we usually work with, we also have GIT mode, which allows us to work with our code repository directly. This is the mode that we want to work with when we integrate Azure Data Factory with a source control system. It is the mode that allows us to collaborate with colleagues and take advantage of our multiple environments. Please notice the Publish button at the top. When we are in Data Factory mode, we lose the ability to publish our data factory directly. Now this needs to be done from the GIT mode. By clicking on the branch drop‑down menu, we can switch between branches, create a new branch, or create a pull request based on the current branch. If you're not familiar with these concepts at the moment, don't worry. We review them later in the course. Let's switch to Azure DevOps for a moment and learn what has changed since we enabled our Git integration with Azure Repos. When we enabled the integration, Azure Data Factory created a series of files that represent the Data Factory infrastructure on pipelines defined as code. Then it saved those files in our Source control system, as you can see right here. When we modify our data factory using the Azure Data Factory visual tools as we are used to, Data Factory updates those changes in this repository as well. Data Factory saves these changes in a branch called master. This is the collaboration branch we set up during the service integrating a few moments ago. The master branch we saw moments ago in the data factory is the same branch you see right here in source control. We mentioned our templates a few time now, but what are they? Azure Resource Manager templates are JSON files that define infrastructure as code. In the context of Azure Data Factory, ARM templates allow us to represent the distinct objects that make our data factories such as the datasets, linked services, pipelines, triggers, and more. Many folks choose to write these files by hand. Still, data engineers working with Azure Data Factory can rely on visual tools that generate these templates automatically and transparently. When we combine them with a source control system such as Azure Repos, we have a very powerful recipe for success to enable continuous integration and deployments in our workflow.

Summary

In this module, we learned how to integrate Azure Data Factory with a source control system, specifically with Azure Repos. Azure Repos is a version control system that we use to store files that represent our data factory infrastructure and configuration. We learned about Azure Pipelines as well. It is a service that we'll use later in the course to automate the deployment of our data pipelines to different environments. We also learned about branching and the collaboration branch, called master, that Data Factory uses to merge the work of multiple data engineers in one place. Finally, we discovered ARM templates and how they fit in the context of Azure Data Factory. The stepping stone is in place. In the next module, we learn how to deploy data pipelines using Azure Data Factory visual tools.

Deploying Data Pipelines Using ARM Templates and Azure DevOps

Introduction

As we learned earlier in the course, there are two ways to deploy Azure Data Factory pipelines, the UI-based method and the automated method that we explore in detail in the next module. In this module though, we learn to deploy Azure Data Factory pipelines using the Azure Data Factory visual tools and ARM templates. I call this the UI-based method. In the next few minutes, we'll learn to create feature branches, pull requests, and export and import ARM templates, all without leaving the familiar Azure Data Factory visual tools that we are used to. Let's begin.

Understanding Azure Data Factory Continuous Ingratiation Life Cycle

This is the workflow we are following in this course to enable collaboration and to deploy data pipelines using the Azure Data Factory visual tools integration with ARM templates and Azure Repos. The first step to enable collaboration is to create a feature branch. We do this from the collaboration data pipeline, the one in our development environment. A feature branch in the context of Azure Data Factory is no more than a copy of the data pipeline represented as code where you can make the changes in isolation from other people working on the same data pipeline. Once you create a copy of the data pipeline, you can make your changes. For example, you could create a new linked service, add an activity to the pipeline, and so on. Once you finish with your changes, you could debug and test your data pipeline locally if you wish to do so. If you are happy with the changes you made, you can create a pull request to integrate said changes with the collaboration branch, in this case, the master branch. A pull request is a concept related to version control systems such as Azure Repos. It is a method for submitting your code changes like the ones you made when you updated the data pipeline. After you create the pull request, another team member reviews such changes and approves them so that they can integrate back to the collaboration branch, effectively merging your changes in the master data pipeline. It sounds complicated, but it is not. You will see that in a few minutes. Right, so after the pull request is approved, your data pipeline changes are merged with the collaboration data pipeline. Now it could be an excellent time to test these changes in the development environment to make sure the changes you introduced are working fine in that environment. After the test, the changes are published from the collaboration data pipeline using the good old Publish button. This process creates ARM templates containing the pipeline definition, as well as parameters used by the data pipeline itself. We can export this template and save it on disk. It is a ZIP file containing a few JSON files. We can then manually deploy the ARM templates contained in the ZIP file to our other Azure Data Factory environments. This deployment action creates the pipeline, linked services, and other services in the other data factory environments. The best thing is that the UI-based method relies on tools you're familiar with, right within the Azure Data Factory visual tools. In the next few clips, we put everything we learn in this clip into practice.

Creating a Feature Branch and Making Changes to the Pipeline

All right, time to get our hands dirty implementing the procedure we defined in the previous clip. The first step is creating our feature branch. Notice that we are currently working on the master branch, also called the collaboration branch. To create a new branch, let's click on the create branch menu option. Okay, the New branch window opens, so we can name it. Let's call this branch evangeloper, as suggested by the visual tools, and press the Save button. It is a good practice to name our branch after your username so others know what you're working on. Notice how the branch name in the branch drop-down menu changed. We are now working in isolation on the feature branch we just created. This branch is a copy of the development data pipeline defined as code on the master branch. Let's switch to Azure Repos for a moment and discover what changed here. For starters, there is a new branch in our source control system named evangeloper. This is the branch that we created moments ago using the visual tools. As I said, this branch contains a copy of the Data Factory definition files found in the master branch. Back on the Data Factory, let's add a new activity to the pipeline. Notice that we are still working on the evangeloper feature branch here. We are going to configure the validation activity to make sure that the source container in our Azure Blob Storage account is not empty when we run this data pipeline. On the Settings tab in the Dataset property, I am going to choose a dataset pointing to our storage account. If you recall, this dataset was created in a previous clip automatically by an Azure Data Factory template. Now we need to select the folder we want to check by specifying the folder path parameter value. There is a pipeline parameter called FolderPath\_SourceStore that contains the name of the folder we want to validate. This parameter points to the sink container inside the blob storage account. Very good. Now we need to instruct the Validation activity to check that the folder exists and whether it has items in it or not. We do this by setting the Child Items parameter to True. Last but not least, we need to link the Validation activity and the Get Metadata activity so that we query for a list of files only if the folder exists and the file count is greater than 0. While we work on our feature branch, it's always a good idea to debug or test the pipeline to make sure the changes we're introducing work as expected. Let's debug the pipeline and make sure the Validation activity we added does not introduce a defect. This operation is going to take a few seconds. Now that the debug service finishes executing, let's check the activity outputs to make sure that both the Validation activity and the Get Metadata activity were able to check for the source folder existence and acquire a list of files, respectively. Very good. It looks like it's working as expected. You know, at this point, if we were working with a regular data pipeline, we would be tempted to publish the changes. If we tried to do so from here, we'd quickly find out that publishing is not possible, and it makes sense. When we are in Git mode, we can only publish changes from the collaboration branch, the master branch, in our case. What we can definitely do though is saving our changes. Saving means writing the changes made to our data pipeline in our repository. Let's switch to Azure Repos for a moment. If we open the feature branch and then the pipeline folder, we find the MoveFiles.json file. This file represents the pipeline definition expressed as code. When we saved the pipeline, the changes we made were saved to this file as code. We can clearly see this by opening the history page for this file, look for the last commit, open it, and see the changes that we made, expressed as code in this file. Here's the code that expresses the dependency between the Validation and the Get Metadata activities. If we scroll down a little bit, we find the definition of the Validation activity, its settings, and so on, expressed as code as well. We created a new feature branch, modified the data pipeline, and saved the changes to our code repository without leaving the comfort of the Azure Data Factory visual tools. In the next clip, we'll learn how to create and approve pull requests, as well as merge and publish our changes.

Creating a Pull Request and Publishing Changes

We created our feature branch, made our changes, and now we are ready to integrate this new functionality back into the main data pipeline. We need to create a pull request. In the branch selection drop-down menu, click on the Create pull request option. This opens a new tab in the browser, and we are redirected to Azure DevOps to finalize the creation of that request. Notice that we are merging the changes made in the feature branch called evangeloper to the master branch that contains the code for the collaboration data pipeline. If we scroll down, we can see the changes we are merging. They correspond with the changes made to the pipeline in the last clip. Okay, we can add a detailed title and a description so the person in charge of reviewing these pull requests has a clear idea of what we are trying to accomplish here. Then we can click on the Create button to create the pull request. Good, the pull request was created. At this point, we can walk away from the computer, drink a coffee or a cup of tea, and wait for the person in charge of this code review to approve the changes. In this demo, we are approving the pull request ourselves. In practice, the person reviewing a pull request shouldn't be the same as the one requesting it though. To complete a pull request, we have a few merge strategies. We selected the full merge, no fast-forward strategy, and click on the Complete merge button. Notice the Delete evangeloper branch after merging checkbox. This instructs the service to delete our feature branch after the merge. Excellent, our pull request was approved, and the changes we made were merged with the main data pipeline in the collaboration environment. Since we are done working with this tab now, let's close it. Back in the Data Factory portal, we can confirm that the changes made to our feature branch are present in the collaboration branch. Also notice that since we deleted the feature branch after the merge, the evangeloper branch is no longer available. Azure Data Factory is asking us to select a new working branch. Let's select master then. We are back in the master branch. The evangeloper branch is gone. We can double-check this by opening DevOps and making sure the branch is gone from the source control system too. All right, back to ADF. Now it's time to properly test the data pipeline in the development environment and make sure the changes we made work as expected. Since we are working with the infrastructure in the development environment, the files will be copied between containers in the development blob storage account. The pipeline's running, and the activities are under way. If all goes according to plan, we'll end up with 10 files in the development environment staging container. Let's see the results. Here's the sink container. When I refresh, the files should be gone. Now let me check the staging container. Very good. There are 10 files in this container moved by the data pipeline successfully. The pipeline works fine. At this point, we need to publish the changes. We can do so because we are now working on the collaboration branch. Remember that publishing can only happen in this branch. When I click on the Publish button, the services publishes the data pipeline and also generates ARM templates with the data factory definition expressed as code. It saves these files in our source control repository in a special branch called adf\_publish. Let me show you this new branch in Azure Repos. This branch is created the first time we publish changes from our collaboration data factory and updated every time we update our data pipeline. Let's investigate its contents. There are two essential files here, the Data Factory template definition, and the Data Factory parameters definition file. Join me in the next clip to learn more about these files. It is time we learn how

Exporting ARM Templates from the Collaboration Pipeline

to export ARM templates using the Azure Data Factory visual tools. The goal is to use this template to deploy the data pipeline we created in our development environment to the staging and production environments. In Azure Repos, let's open the adf\_publish branch and reveal the data factory template definition and the data factory parameters definition files. These files were generated in a previous clip when we published the data factory and represents the data factory and its components, properties, and parameters. This file is the ARM template for the data factory itself, and this other file represents the parameters used by the data factory. These are the files that we are exporting in a moment and will be used to deploy the data factory to our other environments, staging and production. Let's switch to the Data Factory portal and learn how to export these ARM templates, shall we? This drop-down menu in the top bar allows us to perform the import and export process. When we click on the Export ARM template menu option, the service creates an archive containing the JSON files that we just saw moments ago in Azure Repos. Let me extract this file to show you its contents. As you can see, it contains the same JSON files stored in the adf\_publish folder in the source control repository. Let's open the Data Factory template file for a moment. It is an identical copy of the files stored in our source control system. The same goes for the parameters file. It contains the parameters used by the data factory, and it is the same file as the one stored in source control. All right, in the next clip, we deploy the data pipeline to the staging environment.

Deploying to the Staging Environment

All right, friends, this is the moment we've been waiting for. We are deploying the data factory created in the development environment to the staging environment. The objective is to test the data pipeline to make sure it works as expected before we move it to production. I've already opened the development data factory, the staging data factory, and the production data factory, so it's easy for us to jump between these three environments when we need to. For now, let's open the staging data factory. It is currently empty. There are no pipelines, datasets, or linked services created yet. Let's change that by deploying the data pipeline using the ARM templates we generated in the previous clip. When we click on the Import ARM template menu option, we are redirected to the custom deployment tool provided by the Azure portal. We can use this tool to deploy Azure services based on ARM templates by clicking on this link. Let's click on the Load file button and select the Data Factory ARM template we downloaded in the previous clip. Here's the template. We've seen the contents of this file a few times already. To continue, we need to click on the Save button. In this screen, we want to customize the deployment by providing the values for a few parameters that are a part of the Data Factory ARM template itself. We already know that this template represents the data factory we created in our development environment, including its parameters. We want to deploy the same data factory on the staging environment and, at the same time, change the parameters to reference the service in the staging environment. The first thing to change is the resource group. We want this deployment to be part of the staging resource group. Next is the Factory Name. Finally, we want to use the staging environment Key Vault service. Let's reference the correct endpoint by changing the service URL here. The other parameters are just fine. We don't need to change them, as they work in all environments. Before we click on the Purchase button, we need to agree to the terms and conditions. Perfect, the deployment is under way. We manually deployed the data factory from the development to the staging environment by simply using the visual tools to import ARM templates. It's really that simple. Okay, I am getting rid of this \_\_\_\_\_ tab, as we don't need it anymore, and getting back to the staging data factory tab. Look at these numbers right here. When I click on the Refresh button, they will change. Now we have a copy of the exact same pipeline we created in the development environment, deployed on our staging environment. The pipeline is here, the datasets are here, and the linked services and the trigger are present as well. It is the exact same pipeline with one exception. This data factory points to the services in the staging environment. For example, now we are using the staging key vault instead of the development environment service. This happens because we changed the parameter values when we imported the ARM template moments ago. The same goes for the storage account. We are using the staging version as well. I want to prove this to you. Let's switch to the Azure Storage Explorer and open the sink container on the staging environment. The staging folder is currently empty. I am going to run the pipeline, and if everything goes as planned, the pipeline will move the files between those two containers. In the meantime, let's monitor the pipeline execution. Good, now let's check the results. If we refresh the sink folder, we effectively see that the files are gone for good. Now let's open the staging folder and make sure that the files were moved successfully. It worked. We deployed the data factory successfully between environments using ARM templates and the Data Factory visual tools. Now that we were able to test the pipeline on the staging environment successfully, let's deploy it to production.

Deploying to the Production Environment

Our hard work has paid off, and our well-tested data pipeline is ready. Now is time to deploy it to production. The process is almost identical as before, so let's cut right to the chase. We start by clicking on the Import ARM template button. Once the custom deployment page opens in a new tab, we have to click on the Build your own template link as we did before. In the next screen, we need to click on the Load file button and select the ARM template we used to deploy to the staging data factory. Here's the template loaded into the editor. Now we need to replace the data factory parameters with values that reference the production environment services we want to use. We start by selecting the production resource group. Next is the data factory name, and here we merely ought to replace the suffix dev for prd. Last but not least, I need to take care of the Azure Key Vault URL, as we want to use the one created for the production environment as well. We are almost done. Let's agree to the terms and click Purchase so the service can deploy the new services used in the ARM templates we provided. If I now refresh the production data factory, I should see the new pipeline and the other components already deployed. Apparently, the deployment went okay, although there is only one way to know for sure. So we are running the pipeline in a moment. Before we do that though, I want to check the linked services and make sure they point to the right production services. All seems to be good. As I said, only one way to know. Let's run this thing. This is us running the data pipeline in production, so let's imagine for a second that we are moving 10 PB of data between those two Azure containers instead of just 10 files. All right, everything is in place. Let's run our production pipeline, shall we? With only 10 files to move, the operation usually takes only a few seconds. In the meantime, let's monitor the pipeline execution to make sure all goes as planned. Nice. Let's switch back to the Azure Storage Explorer tool and check that the pipeline effectively moved these files between the containers in the production account. As you can see, the pipeline did its job, and so did we by learning to deploy data pipelines to Azure by using ARM templates and the Azure Data Factory visual tools.

Summary

Congratulations, you deployed your first data pipeline in Microsoft Azure. We started by learning about feature branches and how they represent a copy of the collaboration data pipeline where we can make changes in isolation. We also learned about pull requests, a concept related to version control systems, such as Azure Repos, that allows us to submit our data pipeline changes for other team members to review, approve, and merge with the collaboration branch. Finally, we learned about ARM templates, how to import and export them, and, most importantly, how to use them to deploy data pipelines between different environments by using the Azure Data Factory visual tools. You know, we could really call it a day. After all, we are achieving our goal. But there is one other thing we can do to make it even better. We can automate the whole thing. In the next module, we take this workflow to the next level by automating the continuous integration and delivery of Azure Data Factory pipelines using Azure DevOps.

# Implementing Continuous Integration and Delivery of Azure Data Factory Pipelines Using Azure DevOps

## Introduction

This is a very exciting module. We are about to create an automated continuous delivery and deployment pipeline for our Azure Data Factory data pipeline. We are making use of **Azure Pipelines**, a cloud service that is **part of the DevOps family of services** that allows us to **build and deploy our code automatically**.

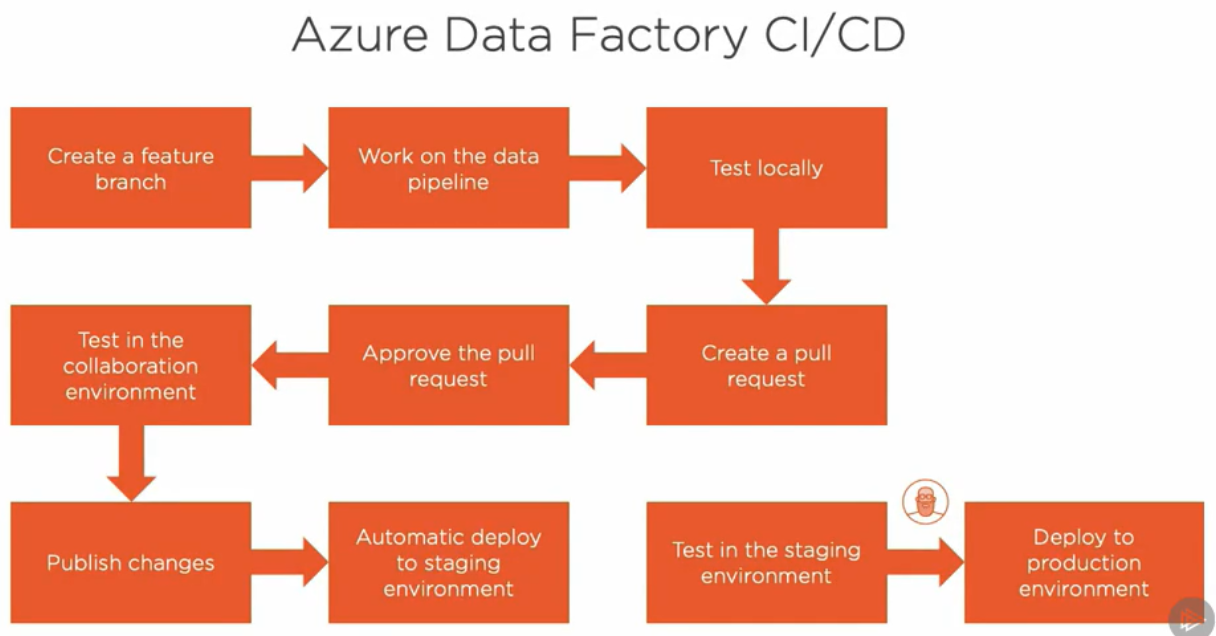
The goal is to automate the manual process we created in the last module so when **we publish changes** to our Azure Data Factory data pipeline, our **DevOps pipeline kicks in and deploys the data pipeline automatically to other environments.**

Takeaways:

1. Azure pipeline (part of Azure DevOps family).
2. Azure pipeline can be used to automate continuous delivery and deployment of our Data factory pipelines.
3. Trigger: Us publishing changes to Development Data factory. Changes get automatically deployed to other environments.

## Getting Started with Azure Data Factory and CI/CD Using Azure DevOps

Here's a visual representation of the automated continuous integration and deployment workflow that we are creating in this module.



It starts in the same way as the UI-based workflow by **creating a feature branch**.   
In this branch, we make the necessary changes to our data pipeline, and when we are ready to **integrate the changes with the collaboration environment,** **we create a pull request**.

Once our changes have been reviewed, approved, and merged with the development environment data factory, **we publish the changes**.

In a previous module, we learned that this section triggers **the creation of ARM templates that are stored on the adf\_publish branch on Azure Repos.** This part of the workflow works in the same way as the UI-based workflow. When the ARM template in the **adf\_publish branch is updated**, the Azure DevOps **deployment pipeline** that we create in this **module is automatically triggered**, deploying the data factory pipeline to the staging environment.

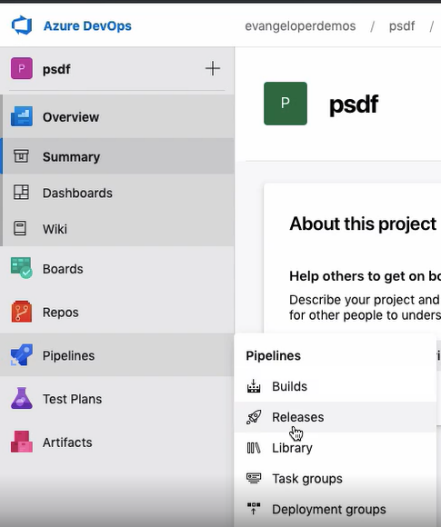
After we test in the staging environment and make sure all works as expected, an **authorized user deploys** the data factory **to the production environment with a simple click of a button**. I like to do things this way to control that the deployment to production is done by an authorized user in a controlled manner only when the team is satisfied that the changes do not have a negative impact and enough testing has been performed in the staging environment.

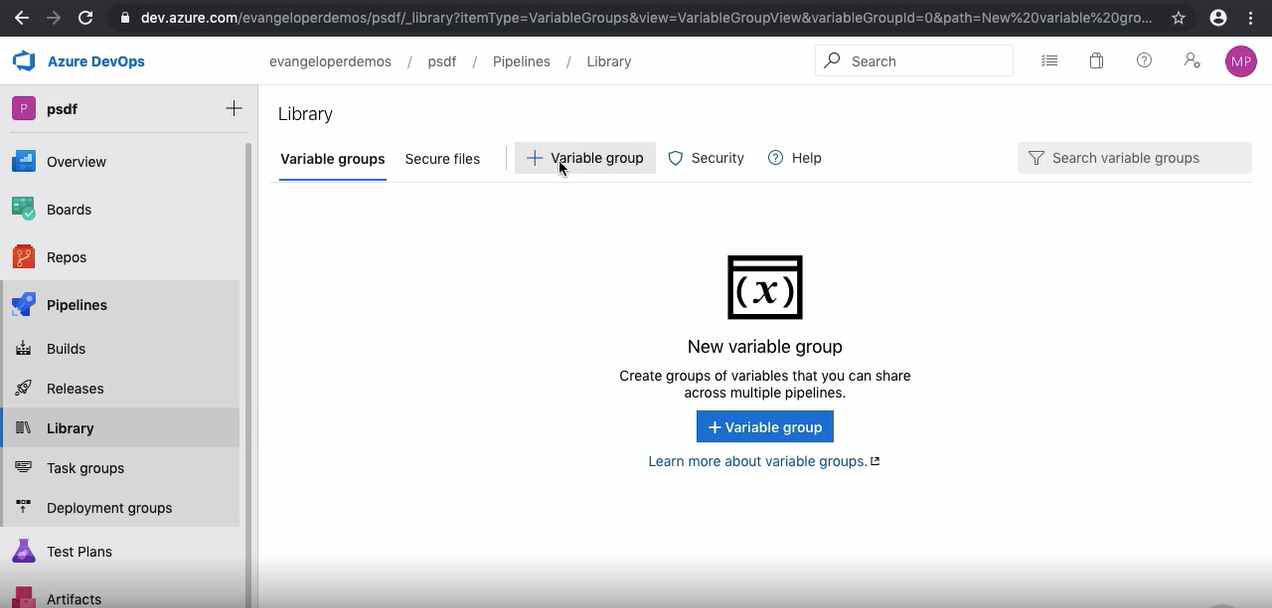
Takeaways:

1. To integrate our code with collaboration environment, we create pull request.
2. Publishing changes to ADF, creates ARM templates under the branch adf\_publish in Azure Repos.
3. When ARM templates under adf\_publish are updated, the Devops deployment pipeline is triggered.
4. Deployment to production is done by authorized user in controlled manner. All he does is simply click button.

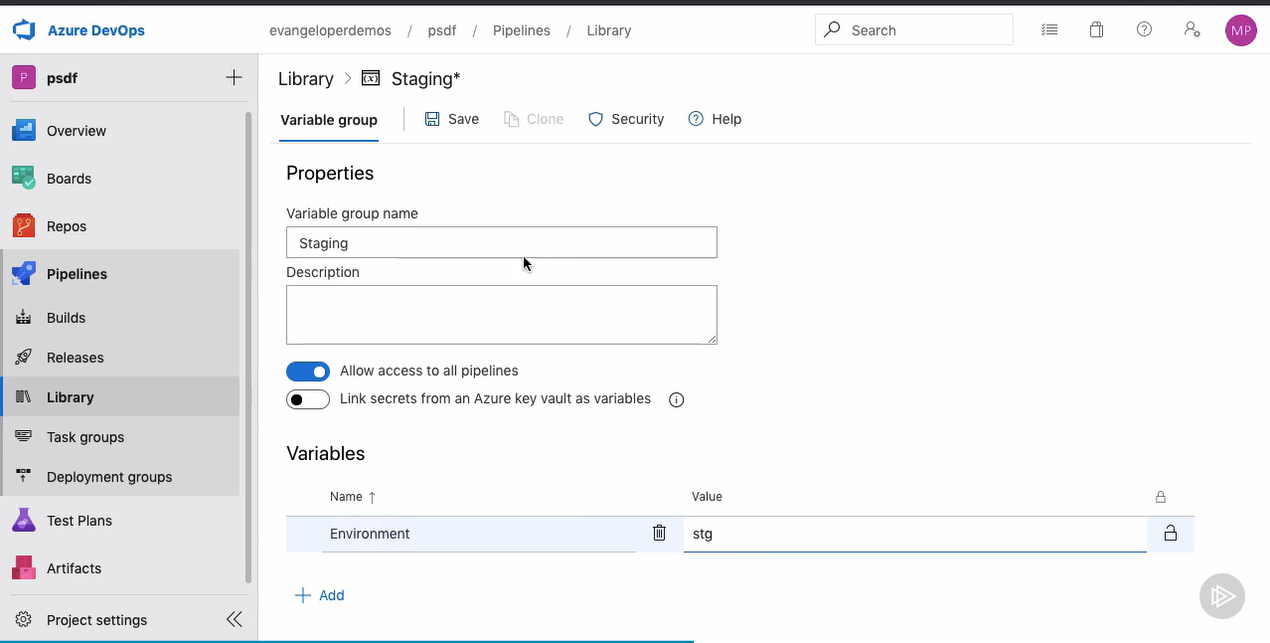
## Creating Variable Groups in Azure DevOps

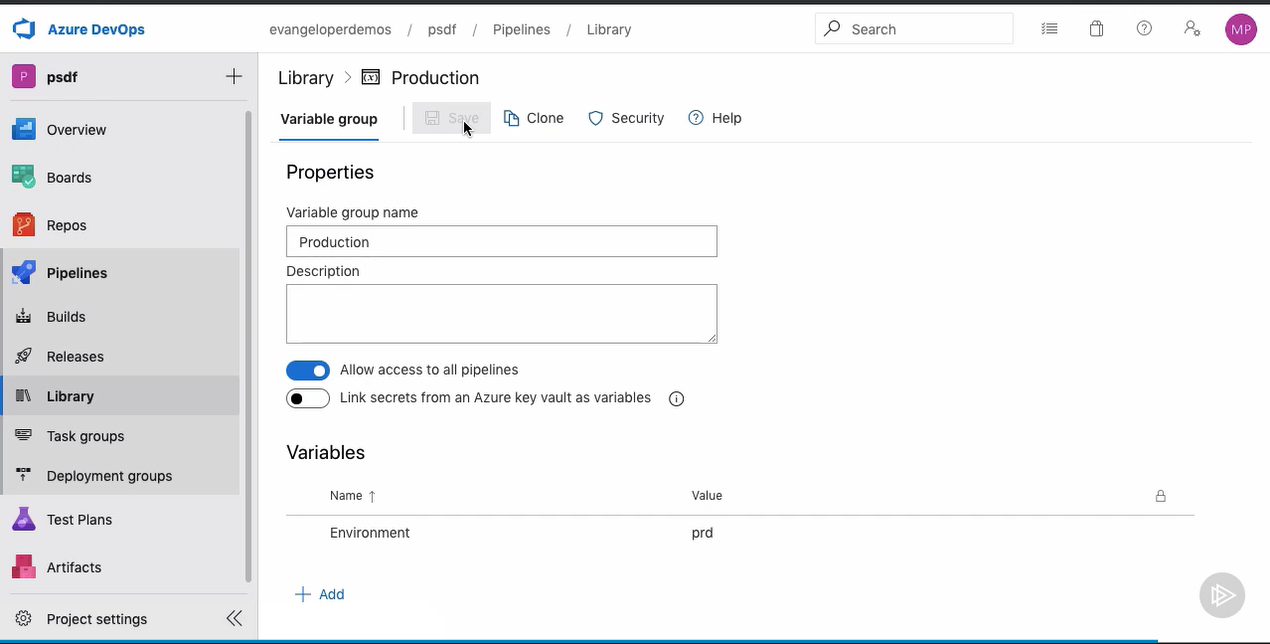
We are back in Azure DevOps, and this time we are going to work with a new feature called Pipelines.

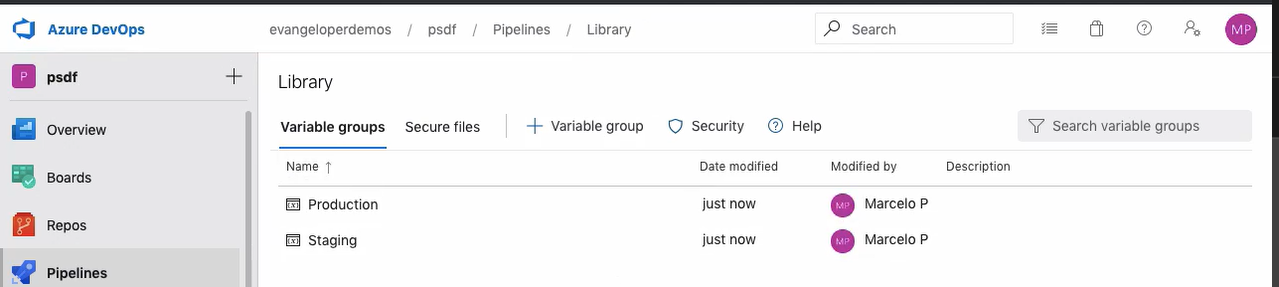
All right, we are going to start by defining a variable group(under Library). They are simply a collection of values and are useful to share those values across multiple pipelines. To create a variable group, we need to click on the Library menu option and then on this button on the top.



We are creating two variable groups in this demo, one for the staging environment and another one for the production environment. In this section right here, we create the variables that belong to the group. Let's create a variable called Environment and set the value to stg for staging.

Let's save it and quickly create one more variable group. Let's call this one production. We need to create a variable with the same name here and assign prd as its value.

Later on, we use the Environment variable defined in each group to help us select the appropriate services for the different environments. The use of these variables and the variable groups becomes more evident when we use them in later clips. Up next, we create the actual deployment pipeline, define stages, and much more.



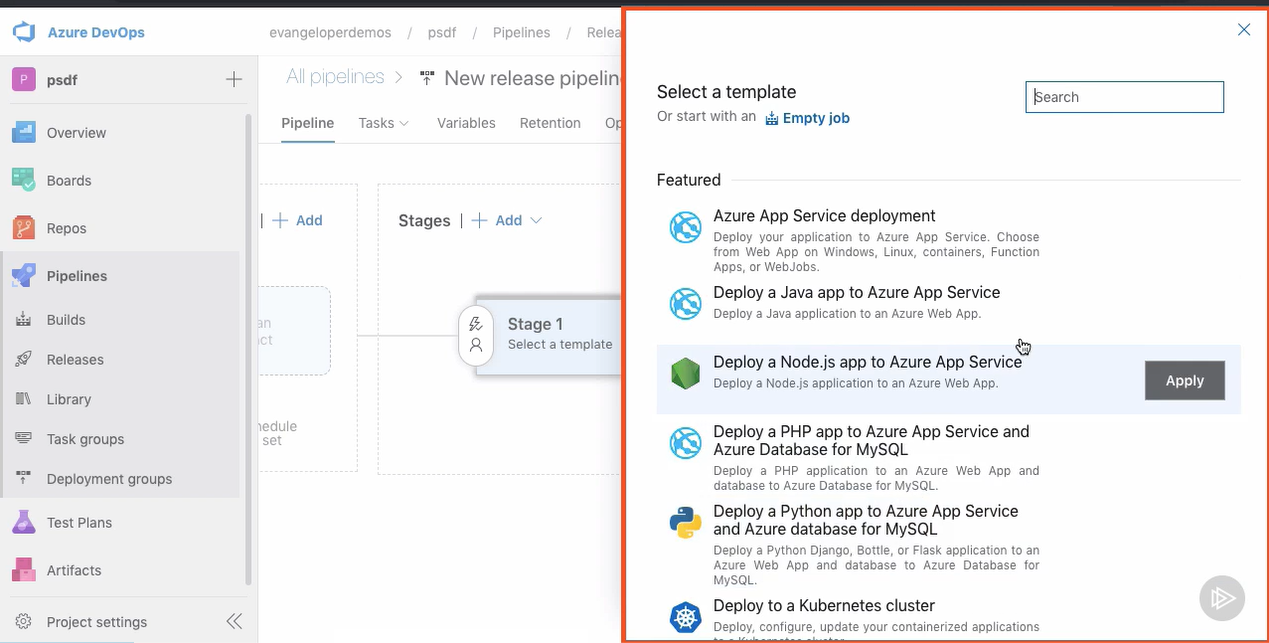
Takeaways:

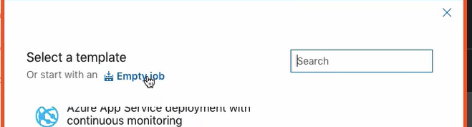
1. Under Azure pipelines, we have option of creating Builds, Pipelines.
2. Under library we create Variable groups- one for each Environment.
3. Under both the variable groups, we create variables with same name, but different values depending upon the environment in which we plan to use.

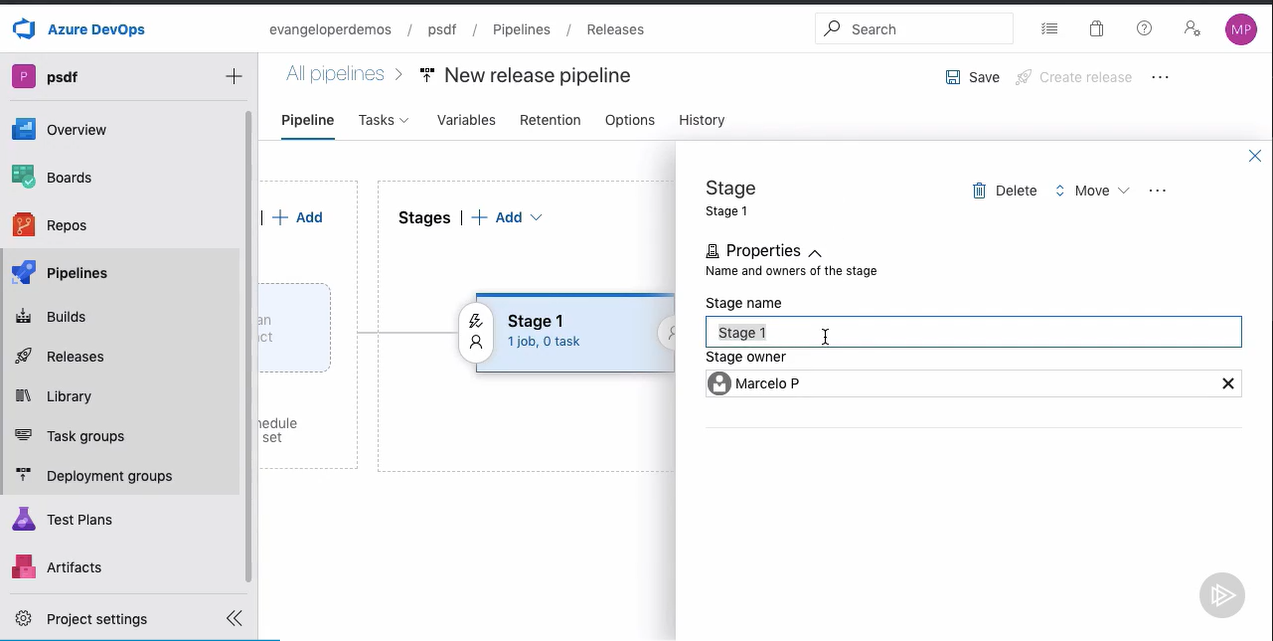
## Creating a Release Pipeline in Azure DevOps

Azure DevOps has **build** and **release** pipelines (2 types of pipelines). To automate the deployment of our Azure Data Factory pipeline, we'll create a **release pipeline**.

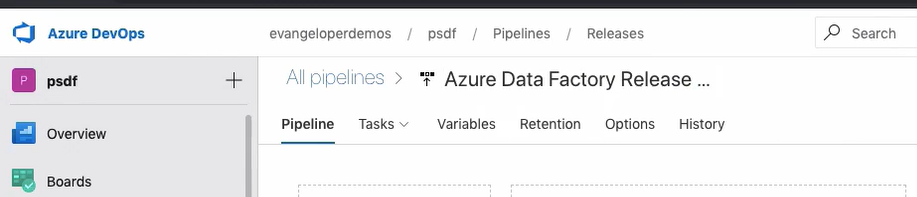
Let us start by clicking on the New pipeline button to open the template selection wizard. Azure DevOps comes with a **bunch of release templates** that we can use **to speed up the creation of our pipelines**, but this time we are creating one from scratch by selecting the Empty job option at the top.



The **first thing that we need to do is create a stage.** A stage is a local boundary in a pipeline where we add related jobs to perform a common goal, for example, releasing the data pipeline to the staging environment. We are **creating two stages in our pipeline today, one that releases our data pipeline to staging and another one that releases it to the production environment.** Let's name the stage Staging, and close the window.

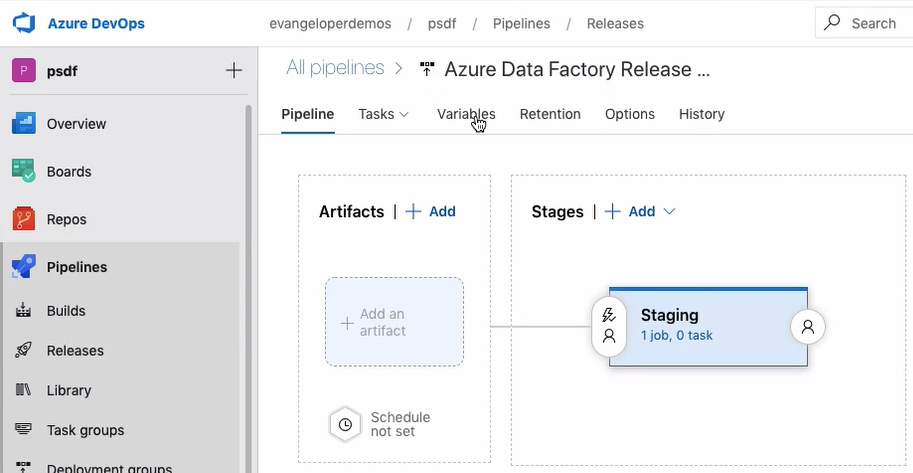


And before we move forward, let's also give the actual DevOps pipeline an appropriate name.

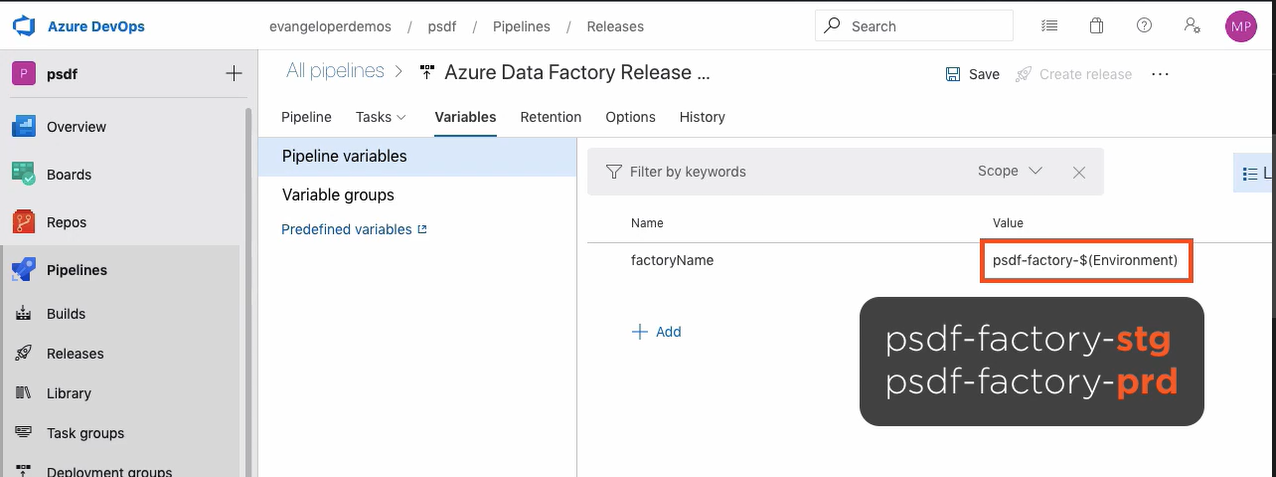


For convenience's sake, I am opening another tab to access the files in the repository. I want to have easy access to the ARMTemplateParameters file to get ahold of a couple of parameters and its values. Let's copy this variable name right here.

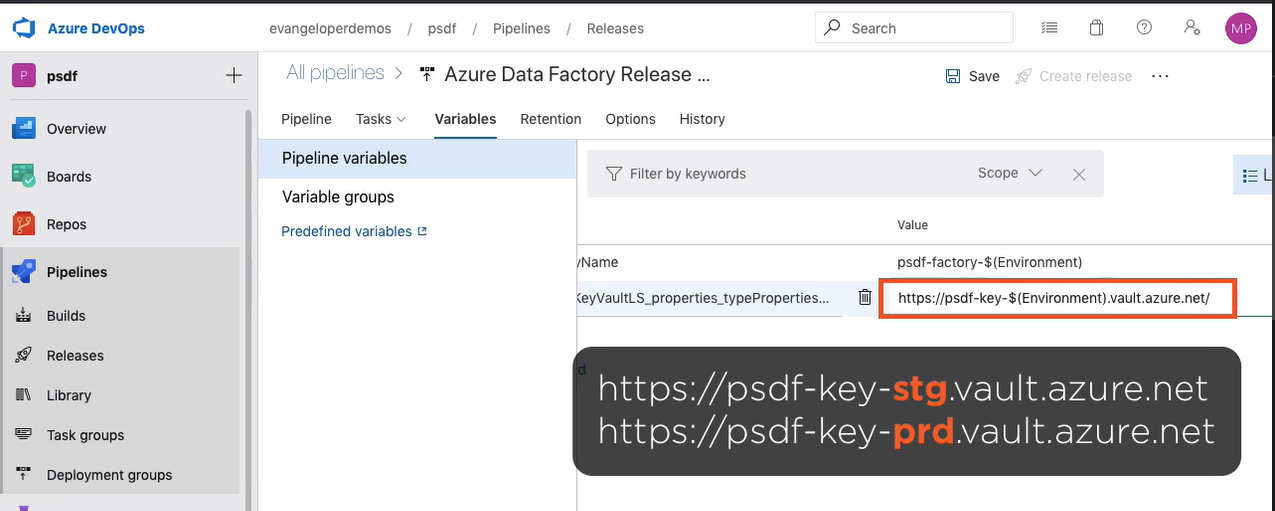
Back in the pipeline now, we need to create a couple of pipeline variables.



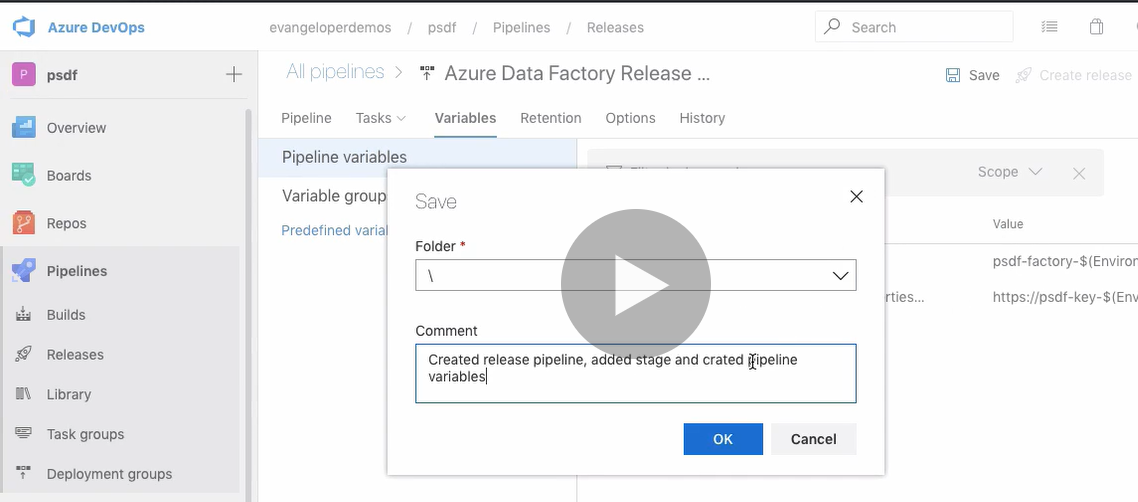
The first variable is called factoryName. It represents the parameter you just saw in the parameters file. Let's grab the factoryName value from this file and paste it as a value for the pipeline variable. It currently indicates the name of the development environment data factory. We want this value to change based on the value of the variable name Environment that we've creating earlier inside the variable groups. The usefulness of this little trick becomes more evident later on when we clone the pipeline and link the variable groups to release stages. For now, it's enough to know that this value right here will change to stg or prd, based on the pipeline stage that it's running.



There is another parameter that we need to create a variable for. This parameter holds the value of the URL pointing to the development environment key vault service. Notice how I'm using the exact parameter name as the variable name. I am replacing the dev string in the URL for the Environment variable as well. This URL needs to point to either the staging environment or the production environment key vault service, depending on the stage the pipeline is running.



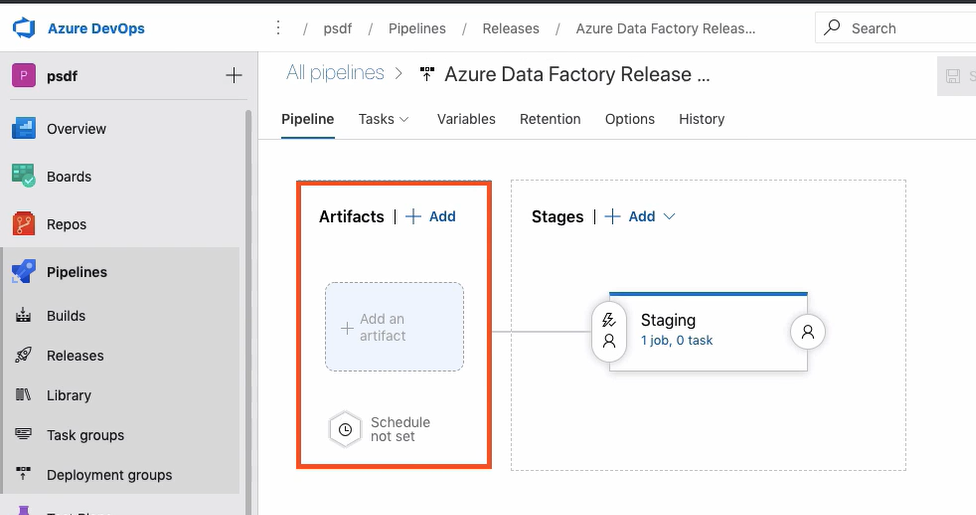
Very good, let's save the pipeline and add a little comment.



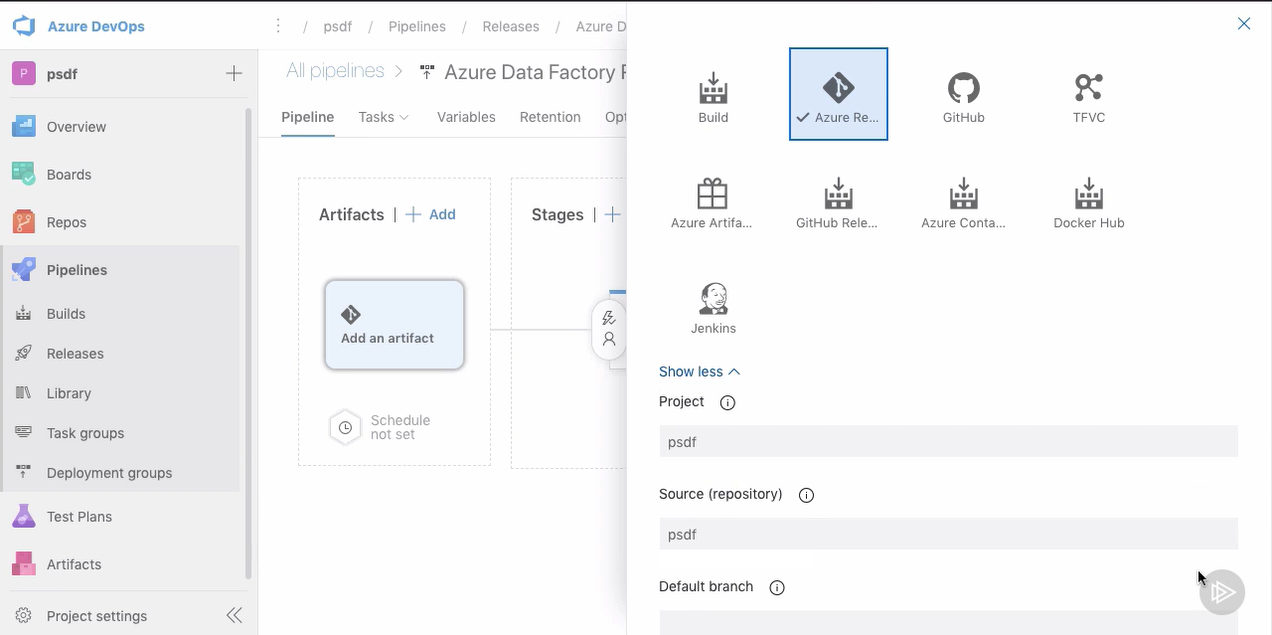
In the next clip, we learn how to add artifacts to the pipeline.

## Selecting an Artifact

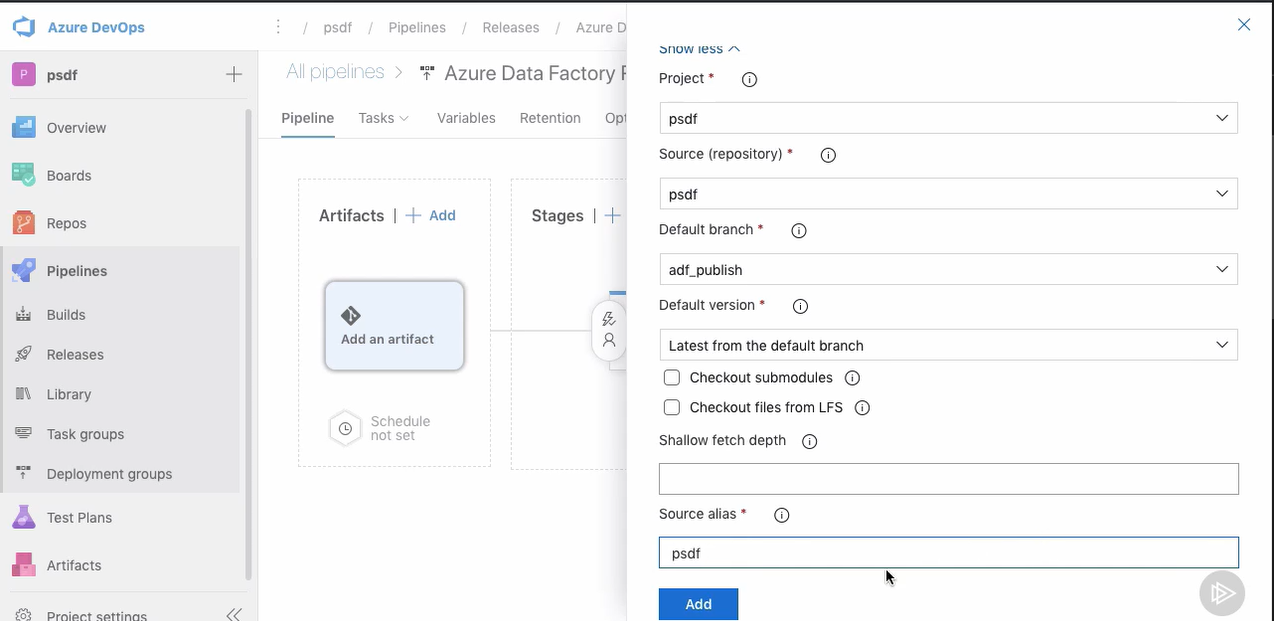
In Azure Pipelines, we **can consume many different types of packages and artifacts**. For our purposes, we are interested in **consuming the ARM templates that** we generated when we published the collaboration data factory.



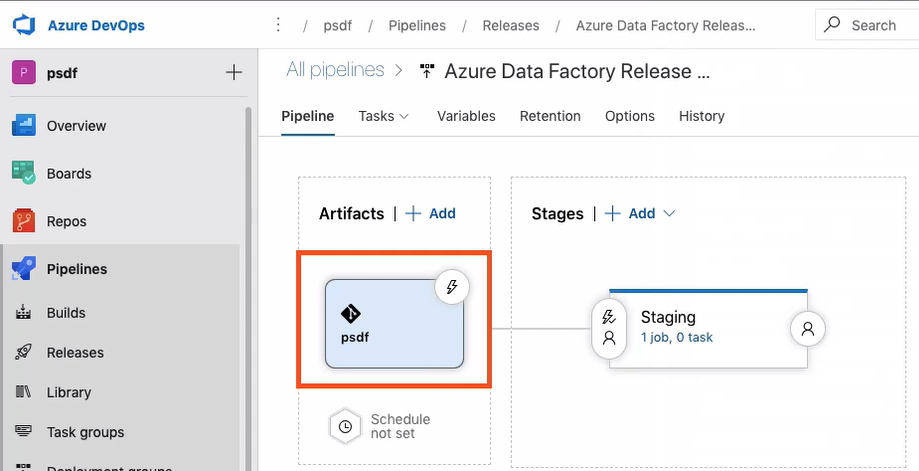
These ARM templates live inside Azure Repos of course, so we need to select that option to get started. Now we have to select the project name and the proper source code repository from the corresponding drop-down menus.



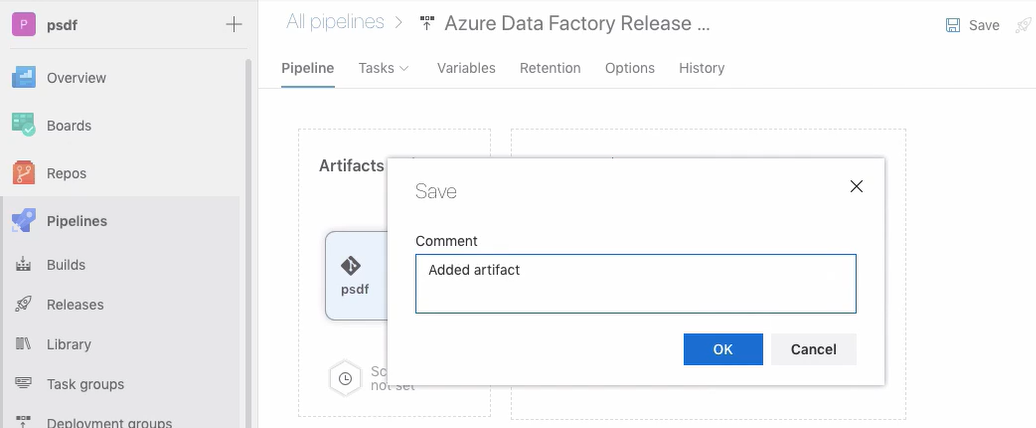
This populates the list of branches from where we can choose the adf\_publish branch generated during the publish process. As for the default version, make sure you choose the latest version from the default branch option. The rest of the options remain the same, so let's add the artifact.



As with the UI-based method, the ARM template files that define our Azure Data Factory pipeline will be used in this DevOps pipeline to re-create the data factory automatically in other environments.



Let's save the pipeline changes one more time and add a descriptive comment.



Now that we configured the artifact, let's create the steps needed to deploy the pipeline by adding the appropriate tasks to the staging pipeline stage. We'll learn how to do that next.

## Adding Tasks to the Pipeline Stage

We organize our pipeline into jobs. Every pipeline has at least one job, which is a series of tasks that run sequentially as a unit. In the next couple of clips, we create a series of tasks that help us deploy our data pipeline to our different environments. Azure DevOps comes with a series of predefined tasks that allow us to connect to various services and deploy our code to a myriad of places. The first task that we are adding is called Azure Key Vault and allows us to access secrets from this service. Let's configure it. First of all, let's name our task. We need to select the Azure subscription the key vault belongs to. We created all three of them under the Evangeloper account. DevOps has excellent integration with other Azure services, and from this window, we can see a list of all the key vault accounts under the subscription. Let's pick any of them and change the name's suffix for a reference to our variable called Environment. Remember, its value changes to either stg or prd, depending on the stage that's running. For the staging stage, its value will be stg. When we clone this stage to create the production stage, its value will be prd. Okay, let's leave the other default values and save the pipeline, always adding a descriptive comment, of course. Now let's go ahead and add another task. I am searching for ARM and selecting the Azure resource group deployment. This task allows us to use the ARM templates that define our Azure Data Factory pipeline and deploy it to one of our environments. The first thing that we need to do is to select the version of the task. At this time, the current version is number 2, so that is the one that I am using. Then we need to add a proper name and select the correct Azure subscription. The Azure resource group deployment task is useful in a variety of scenarios. While we are interested in the first option, creating and updating resource groups, now let's pick any resource group from the list. We are replacing the name's suffix for this variable as well. Another setting we need to configure is the location. As you know, I am deploying the services using this demo to Azure datacenters located in West Europe. When we work with Azure resource group deployment tasks, we need to choose the file that defines the infrastructure we want to deploy. First, we have to set its location. In our case, the template is coming from the linked artifact we configured earlier. In the Template setting, click on the ellipses to open an overlay window to select the ARM template to use in this deployment. Here's the data factory ARM template we've seen many times before. Let's select it and click OK. Now we need to repeat the process and select the parameters file. When we click on the ellipses to override the template parameters, we are presented with a list of parameters found in the parameters file. We are overriding some of them to parameterize the task. So when we clone this stage and create one for the production environment, we don't have to change a thing. It will work out of the box. Let's change the factoryName parameter for this variable right here. If you recall, we created this pipeline variable a couple of tips ago and set this value to it. Basically, this translates to this value for the staging environment, and this other value for the production environment. We have to parameterize this parameter as well, replacing its hard value for this variable right here. The variable will hold one of these two values, depending on the stage that its running. Perfect, I am saving these settings. Now let's configure the deployment mode. It specifies how the existing resources and the ones specified in the template are handled. The default mode is incremental, and it's the one that I'm selecting. In this mode, the Resource Manager lives and change resources that exist in the resource group but are unspecified in the template. If you're interested, you can learn more about Azure Resource Manager deployment modes in this link right here. Okay, we are ready to save the pipeline. The release pipeline is shaping up nicely, but it's yet incomplete. We still need to add a few more tasks. You know, if we were to run the deployment pipeline right now, it would fail, as we have an active trigger in our data pipeline. We need to find a way to deactivate the trigger before the deployment takes places. This and more next.

Adding Pre and Post Deployment PowerShell Scripts

If we are redeploying a data factory and it contains active triggers, the deployment pipeline might fail. As part of their Azure Data Factory CI/CD documentation, Microsoft provides a handy script to stop and restart triggers automatically. Here's the script URL. It's also included in the course Git repository and as part of the course code. This is the script I was telling you about. It is a PowerShell script. Once we supply some parameters, it takes cares of the hard stuff for us. We need to upload it to a repo in the source control system so it's visible to the deployment pipeline. We have to make sure that we select the adf\_publish branch, and then we click on the Upload files button. Here's the script. Let's select it and click the Open button. Now let's commit this file to the source repository. Here it is. It is the same script I was showing you earlier. Okay, let's go back to our deployment pipeline. To continue creating the pipeline, we need to click on the Edit button to add two more tasks in the staging stage. We are dealing with a PowerShell file, so it makes sense to look for PowerShell in the list of tasks. Here's one called Azure PowerShell, and I have a hunch this is the one we need. I want to run this task before the deployment task so it can stop any active triggers and prevent the deployment pipeline from failing. Let's begin by selecting the correct task version. We are working with the latest one, version number 4. It is also a good practice to give the tasks a descriptive name. This one will do. As with other tasks, we need to select the subscription. I happen to have two subscriptions available, but as you know, we are working with a subscription named Evangeloper. Very good. Now we need to configure the script that this task is going to run. We can decide between adding a script inline or providing a path for a script file. We are choosing the latter option in this demo. Here's the script file we uploaded moments ago. Now we need to supply the script arguments. We can add them one by one on this screen, but I am going to paste them here to save some time. Now, if we click on the ellipses again, we can have them formatted nicely in a table where we can edit them if we need to. This is the path to our ARM template containing the data factory code. This is the ResourceGroupName and the DataFactoryName, and these are two handy flags that allow us to reuse this script for pre-deployment and post-deployment purposes. This flag indicates whether the script is running before or after the deployment. And when we set this flag to true, the script deletes unused resources. Okay, let's accept these values. Last but not least, we need to set the PowerShell version running this task. Since we are using hosted agents, I prefer to set this setting to Latest installed version. By the way, an agent is simply a machine running these tasks. Okay, now we can save it. This script is set to run before the deployment, and based on the supplied parameters, it's going to stop active scripts only. We need to clone the task by clicking on the right mouse button and selecting Clone task, and then move it to the bottom of the list so it runs after the deployment has taken place. Now we need to make just a couple of changes here. First, the name. Let's remind our future selves that this is the post-deployment task. Then we need to change a couple of script argument values. This is a post-deployment script, so let's change this flag to false. By doing so, the script will reactive the stop data factory triggers. Then change the deleteDeployment flag to true so the script deletes unused resources after the deployment. That's it, friends. Now we are ready to save the deployment pipeline for good. This is our complete pipeline. First, we connect to the staging Azure Key Vault service. Then we run some pre-deployment scripts to prevent the deployment pipeline from failing. This task is the one deploying the data pipeline to the staging environment, and then this last task reactivates the data pipeline triggers and cleans up unused resources. So far, we have been creating a stage that deploys the data pipeline to the staging environment. Now we need to repeat the process and create one for the production environment. We'll work on it next.

Cloning Stages, Linking Variable Groups, and Configuring Permissions

I am creating another stage, this time to deploy our data pipeline to the production environment. We could start by scratch and repeat the process we did for the Staging stage, but we can also clone the current stage and save ourselves about 95% of the work. We used a few variables to parameterize the staging stage, and this is going to pay off now, as the number of changes we need to make is really minimal. Let's name this stage and call it production. This stage contains the same set of tasks as the Staging stage, same configuration, same everything. As a matter of fact, we don't have to change a thing. We are falling into the pit of success by design. The secret is in the variable groups and the environment variable we created earlier in this module. Let's go to the Variables tab and then click on the Variable groups option in the left menu. We have two variable groups, one named Staging and another one named Production. Inside each of these groups we created a variable called Environment and assigned it a different value on each group. In the staging group, we assigned the value stg, and in the Production group, we assigned the value prd. We also have two stages, also conveniently named Staging and Production. Now we are going to link variable groups to stages, and you guessed it right, we are linking the production variable group to the Production stage, and the staging variable group to the Staging stage. Here's the relationship between variable groups and stages. Okay, let's save the pipeline and continue. When the deployment pipeline runs the Staging stage, the Environment variable will be assigned the stg value; therefore, the staged tasks will point to the Data Factory and Key Vault services in the staging environment. The same goes for when the pipeline runs the production stage. The services used will be those in the Product environment. There is one more thing we need to take care of. We need to allow the data factory account access to the Key Vault service. Let's switch to the Azure portal and navigate to one of the data factory accounts. Once in there, let's open the Access control page and click on the View role assignments. Here's the name of the principal we are looking for. Let's copy the name, navigate to the Key Vault services page, and open the staging key vault service. We have to configure the key vault access policies to allow our data factory access to this service. To do that, click on the Add Access Policy link. We are interested in granting two secret permissions, Get and List. This allows Data Factory to list and get key vault secrets. Now we need to grant access to the application principal. In the Select box, let's paste the principal name we got moments ago and select it from the list. That's it, friends. Let's save the access policy to finish. We need to repeat the same process to grant our data factory access to the production environment key vault. Let's select the Access policies link in the left menu and click on Add Access Policy link. We have to set the exact same permissions in the Secret permissions box. We are interested in the Get and List permissions. As for the service principal, we search and select the same account as before. That's it. Our deployment pipeline is almost complete. We still need to configure conditions to trigger our staging deployment pipeline automatically and create a manual approval process to deploy the data pipeline to our production environment. Let's take care of that next.

Enabling Continuous Deployment Trigger and Pre-deployment Conditions

I want to deploy the data pipeline to the staging environment automatically every time the artifact changes. We know that ARM templates compose the artifact and that they are updated every time we publish the data pipeline in the development environment. Therefore, every time we publish the data pipeline, the deployment pipeline kicks in and deploys the data factory to the staging environment automatically. To enable the continuous deployment trigger, we click on this little icon right here. The process is pretty simple. We need to enable the trigger and select the filter. Every time there is a change in this branch, the deployment pipeline kicks in. That's it. I want to follow a different strategy to release the data factory to production. I want to be able to release the data pipeline manually. Why? Because I want to make sure that what goes into production has been vetted and tested in the staging environment before we release the changes to our main environment. This deployment needs to be approved or rejected only by an authorized person. For demo purposes, I will select myself. All right, we just finished setting up our automated deployment pipeline. Now we need to put it to work by testing the complete workflow.

Implementing a New Feature in the Development Environment

I want to test our deployment pipeline by creating a new data pipeline feature in the development environment and push the change first to staging and then to production. Let's start by creating a feature branch to isolate our code changes from the work of others. This action also created a branch in our Git repository in Azure Repos, as we've seen before. If this were a real-world project, we would be adding a significant feature, but for demo purposes, it suffices to add something very basic, such as a simple Wait activity. Our goal is to force a change in the data pipeline so that we can take the change from our development environment all the way through the production environment using our automated release pipeline. After giving the Wait activity a proper name and selecting the correct settings, we can save the changes. You know, testing the change locally while we work in our feature branch and before we attempt to merge these changes with the collaboration branch is always a wise practice. We could do so by either debugging or manually triggering the data pipeline. Since this is such an innocuous change though, I will skip the tests this time. Okay, time to merge our change with the main collaboration branch. We need to review the change and create a pull request. Once the pull request is created, a colleague must review the change and either approve it or reject it. For the purposes of this demo, we can approve and merge the changes with the main collaboration branch ourselves. Let's switch to the development data factory and see if the change is reflected there. Once our pull request was merged, our feature branch was deleted from the source control repository, and now Data Factory is asking us to select a new working branch. Let's select the master branch. Our new feature is showing up in the development data factory. We need to test the data pipeline in the development environment and make sure the integration of our feature works as expected. Here's the Azure blob storage container for the development environment. We are moving files from the sink to the staging container. Let's trigger the data pipeline and see how it works. I am going to bend time a little bit and accelerate the process. All right, the pipeline finished executing, and all actions were triggered successfully. As you can see, the data pipeline moved the files between the containers as expected, and we can now confirm that the data pipeline change is working good after the integration in the main collaboration branch. Our new feature works as expected in the

Deploying to the Staging Environment

development environment. Let's push this change to the staging environment and then to the production environment. To trigger the automatic deployment to staging, we need to publish the pipeline to the development environment. By publishing the data pipeline changes, the system generates ARM templates and saves them in the adf\_publish branch in our source control repo, effectively triggering the pipeline execution. I am switching to DevOps to take a look at our release pipeline. Look at this. A new release has been created, and it's currently executing the staging stage. Let me click on it so we can see the details. These are the tasks we created earlier. The connection to the Key Vault service succeeded, and now the pre-deployment script is executing. Now is the turn for the data pipeline deployment task. The deployment pipeline is currently deploying the data pipeline to the staging environment. This operation usually takes just a few seconds to complete. Now the final task is executing. The post-deployment script is reactivating our data pipeline trigger in the staging environment. Great, the deployment succeeded. Here's the staging data factory pipeline. The pipeline changes, including the Wait activity, were successfully deployed to this data factory environment. At this point, we want to test the data pipeline to make sure all looks right before we deploy it to the production environment. Here is the staging blob container. The sink folder contains only 10 files, but in a real-world scenario, we would want to test with a more significant data sample. I am triggering the pipeline execution, making sure the data pipeline behaves as expected in this environment as well. While the pipeline runs, let's monitor its execution in the Monitor panel. We are only moving 10 files, so the process should not take very long. Okay, the pipeline executed successfully. Let's see the results. The files are gone from the sink folder and present in the staging folder. So far, so good. The quality assurance team is happy with the results, and they have given us the green light to deploy the changes to production. We'll do that next. The team is happy with

Deploying to the Production Environment

the changes we introduced. It was tested in the collaboration environment and also in the quality assurance environment. The new data pipeline feature is ready for production. Let's push the changes to this environment then. One way to initiate the process is by clicking on the Production label, right here. As you can see, this release hasn't been deployed to production yet. Let's click on the Deploy button to get the ball rolling, shall we? Earlier, we configured a preapproval process for our production deployments. Since we set my user as the authorized person, I can click on the Approve button and initiate the deployment to the production environment. Excellent, the deployment is now started. The first task in the deployment pipeline has completed successfully, and the PowerShell pre-deployment task is now executing. Deploying the data factory takes about two minutes, so using my magic powers, I am going to bend time a little bit and accelerate the process. Now the deployment has succeeded. I am going to double check by opening the production data factory and making sure that the Wait activity is showing up in the data pipeline. It surely is. This is the moment we've been working so hard for. We need to run the pipeline in production. Here's the production container sink folder and the staging folder. In a real-world scenario, we would probably be looking at thousands of files, but in this demo, we only need just a few to make sure that our pipeline works. Okay, let's trigger the pipeline execution one last time, now in the production environment, and monitor the process so we can make sure all goes according to plan. The data pipeline is connecting to the production environment blob storage container, getting a list of files from the sink folder and moving the data to the staging container. Let's refresh the sink blob container. All files are gone, nice. Now let's click on the staging container to make sure the data pipeline moved the files successfully. Excellent. We created a new feature in the collaboration environment and deployed the change between our environments using the automated deployment pipeline we built in this module.

Summary

I am so glad you made it to the end of the course. Thank you. This module in particular has been so much fun, as we learned to create an automated deployment pipeline for our Azure Data Factory data pipelines. As you already know, continuous delivery and deployment workflows contribute to foster collaboration and deliver well-tested data pipelines to production. I hope you enjoyed the course as much as I enjoyed sharing this knowledge with you. Until next time.