# Chapter 9: Deploying the release

Perhaps you have skipped ahead to this chapter rather than reading all previous chapters. That’s ok, but the other topics were intentionally placed ahead of this one. It is true that until we actually deploy bits to an environment, no one can use them. However, the deployment pipeline is where all prior techniques provide their value of squeezing out defects, so they are not promoted to our users. In this chapter, you will learn the model for designing your deployment pipeline, the types of environment to configure therein, and the types of activities necessary during deployment.

## Designing your deployment pipeline

In order to determine the proper structure of your deployment pipeline, you will have to decide how many environments to configure and the differences between them. Before we dive in to that topic, let’s consider some principles that will guide those decisions.

* **Build one, deploy many**  
  Regardless how many environments you have, you will deploy the same release candidate, produced from a single continuous integration build many times, once (at least) per environment type. Do not do anything that rebuilds or recompiles from source once a version enters deployment activities. If there is a problem anywhere, consider the release candidate dead, correct the problem, and proceed with a different versioned release candidate.
* **Do nothing on production for the first time**  
  Design the deployment pipeline so that every unique activity necessary in your deployment be performed in at least one pre-production environment before that activity is executed on production. For example, if your production environment runs on a web farm with several batch job servers and a large SQL Server cluster, it would not be prudent to have single-server configurations in all pre-production environments. Additionally, no files destined for production should be created or changed after the continuous integration build has packaged the release candidate. Everything necessary for the production deployment should be put into the release candidate packages at the end of the CI build. If you find something is missing, stop the line, add the missing piece to the code base, and let the CI build package up another release candidate that includes all that is necessary.
* **Shift left on pipeline capabilities**  
  In deployment of application components as well as configuration of settings and data, push logic into script files that are stored in the application’s Git repository. While the CI build configuration and the deployment steps will allow for running scripts that are stored in arbitrary locations, you create global and temporal dependencies through this tactic. Make sure that as many commands, scripts, and logic as possible are sources from the Git repository and the packages that make up the release candidate that is being deployed.

### Determining environments

Our industry has many terms for server environments. Everyone has Production. Also uses are the following.

* Local
* Sandbox
* Dev
* Integration
* Test
* User
* UAT
* QA
* QC
* Acceptance
* Staging

While there is no standard, the pre-cloud set that seems to be commonly known are Development, Testing, Staging, and Production. With the advent of easily creatable and modifiable environments, any notion of standardized environment names and stages has weakened. We will attempt to sidestep this confusion and talk about the purposes of various environments.

In a DevOps environment, you will never have fewer than three (3) deployed environments for the team. Consider the following table.



Table: The three distinct types of environments in a DevOps pipeline.

This table shows the three types of environments you will need when designing your deployment pipeline. You are free to have as many of each type as you like, but you will never have fewer than one of each of these three types. Everyone understands production. It exists for the people who derive value from using your software. The next environment before production is for any type of manual testing. We will call that UAT because it is focused more on the users than the engineering team. Finally, we have an environment that is only for automated verification of all kinds. To remove ambiguity from other environment names that have been used in the past, we call it the TDD environment, short for Test-Driven Development. In the TDD environment, no humans are allowed. If a human were to attempt to use this environment, they would find it being created and destroyed at a rate that precluded any valuable usage.

Let’s see some examples one could use when determining how many of each type to select.

**Production:**

In the case of production, you could choose to provision a dedicated production environment for each customer or have all of your customers use a single production environment. You

**UAT**

You can have a single manual verification environment if your organization is small. Or, you might have several different user or stakeholder groups that might benefit from having a dedicated environment of this type so that they can choose the cadence by which to accept the next release candidate that is ready. You might also decide to provision another environment of this type for exploratory testing of a particularly large data set – looking to verify if the system provides a snappy user experience even with a very large database.

**TDD**

This type of environment is suitable for complete automated construction and destruction. Every successful build should case a new deployment to this environment type. Because you may have many feature branches in play at a time, your CI build should be configured to be parallelizable – that is, multiple builds happening, one per active branch. And because each build causes a deployment to this environment type, you can have multiple instances of this environment being created at one time. For example, if you and a colleague each commit changes to your feature branch at the same time, you want the build, packaging, and deployment to your TDD environment to happen quickly without waiting on your colleague. You accomplish this by having the naming of the environment parameterized by your build or branch and creating an instance of the TDD environment dedicated to your build. Then, your acceptance tests execute (pass or fail), and the environment is destroyed.

### Assigning validation steps to environments

You are in control of how many actual environments to have in your DevOps pipeline. You will never have fewer than three but depending on how many of each type you choose, you may have more. It’s also your choice which environments to place in series and which to place in parallel. For example, if two stakeholder groups each need a dedicated UAT (manual verification) environment, you may decide that each can receive the new release candidate at the same time and work on validating it in parallel. In this example, you would provision two environments (or keep the environment around permanently) and deploy to each environment at the same time. From a process perspective, you would wait until each group had validated the release candidate before deploying to production. Here is a way to think about what types of activities might be appropriate for performing in each environment type.



Table: Each environment type is built for different deploy and validation steps.

The above table illustrates the deployment and validation steps that are appropriate for each environment type. As we move from automated validation (TDD) to manual validation (UAT) to production, we perform fewer steps. The design of the progression through environments is intended to front load as many validation checks as possible in order to find problems. “Shift left”[[1]](#footnote-1) is a statement of value that has grown in popularity within the DevOps community. The purpose of the Shift Left type of thinking is to design a process that finds as many defects as early in the process as possible. Both preventative and removal methods are used to cause the software product to be more defect-free the further down the process it progresses. In the previous table, the TDD environment includes a full spectrum of activities, from creating the environment from scratch to building the database from nothing, to running the full acceptance test suite. The UAT environment has some on-demand options for when you need to either recreate the environment or reload test data, but we will always need to deploy the new version of the application and migrate the database. This last point cannot be stressed enough. Manual validation environments should always be deployed completely automatically. This includes the database and data stores along with all the application components. This is practice for an unattended production deployment. We do not want to do anything in production for the first time. And we do not want any manual steps in the production deployment process.

### Deploying data changes across environments

Let’s face it. The database has some unique challenges in DevOps. Application components do not have any state. They can be destroyed and put back easily. Storage components must preserve data for years. When we discuss “the database”, the same thinking and principles will apply to any data store, whether it be a relational database engine, blobs, tables, json collections, or merely a directory of files on a network share. This data must be guarded and preserved through many, many deployments of application components. The schema, or the structure within which the data is organized, must be continually upgraded and modified while preserving the integrity of the database. Early in this book, we covered in detail the process of database migration tools. During deployment, you must think about the data needs of the different environment types. To move through this topic, let’s review the different types of data to be managed in our DevOps environment.

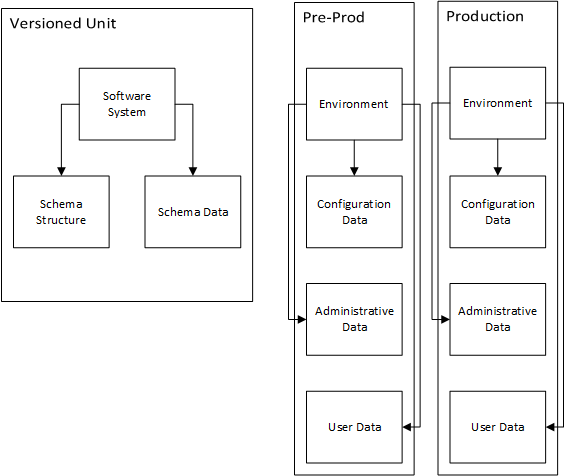


Figure: Each of the four unique types of data are managed in different ways.

In a software system, we can subdivide all data and data concerns into four distinct types. These types are managed differently in our DevOps pipeline. Let’s review them.

* **Schema**  
  The schema, or the structure of the data, is owned by the software system and should be exactly the same in every environment. This includes SQL Server stored procedures, views, indexes, functions, etc. The schema should be versioned and stored with the application code.
* **Schema data**  
  This data is architecturally part of your schema and should be the same across environments. For example, standard lists are in this category. These lists can populate drop-down boxes in your application. Common name prefixes (Mr., Mrs., etc) are a good example. These are defined during development. This schema data should be created and deployed with schema changes while being stored in the Version Control System along with the application code.
* **Configuration Data**Configuration data belongs to the environment itself. It should not be stored with the application code because it is potentially different from environment to environment. Some of it may be sensitive in nature, such as passwords, tokens, and credentials. Some of this configuration data might be in an XML or JSON configuration file. Other configuration data might be stored in a database table. The storage location does not change the nature of the data or that it changes environment to environment. Because of this, it should be deployed to the environment when the application and database are deployed. The automated deployment process should handle the process of retrieving the configuration data meant for the environment and deploying it properly, whether it be poking a string into an XML file or inserting a record into a SQL Server database table.
* **Administrative data**Administrative data is owned by the organization supporting the environment. A common example of this is top level user accounts or customer header records. In many applications, if not a single user account record exists, the software cannot do anything. At a minimum, a global administrator record might need to exist to enable functions to light up. Administrative data can differ by environment, but it doesn’t have to. Because it is determined by the organization supporting the environment, it could be the same across two environments and then differ on another. This data should be deployed to the environment automatically and should not be stored with the application code because it likely contains credentials.
* **User data**User data belongs to the users who create the data. It is different from environment to environment. This is the type of data you are most familiar with. It is constantly growing, constantly changing as people use the system. It should be preserved across deployments. All automated database migration processes and tools are designed to preserve the integrity of user data.

In the previous figure, separate the ways to handle data into Production and Pre-Production. The user data in our two pre-production environments (TDD, UAT) are not end user or customer data sets. In UAT, the set of user data has been built up or curated by a stakeholder group (or perhaps transformed from a production backup with sensitive information expunged). This set of data can be reloaded on demand from its source. In our TDD environment, the user data might be nothing. Because each automated acceptance test will be responsible for setting up the records needed to run the test, there might be absolutely no user data to deploy. And that would be just find. After the acceptance tests run, the database would contain quite a bit of user data because each test scenario and application transaction that executes will create user data.

### Choosing your runtime architecture

For any given application with many logical components, you will have several viable runtime architectures. In the 4+1 Architecture[[2]](#footnote-2), the physical layer is meant to depict the structure of how the application is running in hardware. Therefore, if we have decided that we are using Microsoft’s hardware, in Azure, in a particular region, we still must decide and specify which Azure services to use to run each component of our application. Consider our logical architecture.



Figure: The logical architecture of our application

Our application has three logical components, at its highest level along with an acceptance test suite that must be deployed somehow in order to execute against the application in the TDD environment. With this application, we have a great number of options when choosing how to design a suitable environment.



Figure: Each application can be deployed on a spectrum of environment types

Consider the spectrum of options here. We could choose any number of options for deploying our application. The options on the left give us more control but also more responsibility and maintenance. The options toward the right constrain the scope of the computing resources that we can control but also relieve us from more responsibility and maintenance. Because we control less of the computing environment, we are responsible for less maintenance. As the options move to the right, you have fewer APIs and resources available to your application. As an example, if your web application uses a custom font for rendering a screen, it will be incompatible with Azure App Services, which does not provide the ability for installing fonts on the underlying servers. But if your application only makes use of APIs available in that environment, it’s the most maintenance-free way to run your web application and off-line job.

We have already established that we will not be mounting physical servers into a cabinet in our own data center – but that option would run our application just fine. We could contract with a regional hosting company and ask them to provision some virtual servers for us. We would also provision some VMs in Azure. We would likely configure a few web servers, one or more servers to run the off-line job, and then we would need a SQL Server cluster for our database. We can likely use any server as a host for our acceptance test suite while it executes.

If we do not want to manage a server operating system, we can reach for containers or PaaS (Platform as a Service) in Azure. Windows containers are growing in maturity but retain some challenges. Linux containers are more mature and are an option if you are targeting Linux for your .Net Core applications. Progressing further than containers are the PaaS services such as Azure App Services. These can host web applications, off-line jobs, as well as a container image. If the Azure cloud had a personality, Azure App Services might say, “Don’t ask too many questions. Just give me your code. I’ll run it for you.” The industry is certainly moving from the left side of the spectrum to the right side. How far and how fast you and your team move are completely up to you and the software you are currently operating. For this book, we will be deploying to the runtime architecture shown here.



Figure: Physical architecture has been specified for our application.

For this book’s example application, we are choosing Azure App Services for the ASP.NET UI, which is a web application. The off-line job will be deployed as an Azure Function and hosted in App Services. The SQL Server database will be run in Azure’s SQL Database service. The acceptance tests will be deployed to the hosted agent provided by Azure Pipelines. The tests can execute from there.

## Implementing the deployment in Azure Pipelines

Now that we have decided our environments and the physical (or runtime) architecture for our application, the next step is to extend our pipeline from our continuous integration build and configure deployments across our three environments. Once it is configured properly, the overview will look like the following.

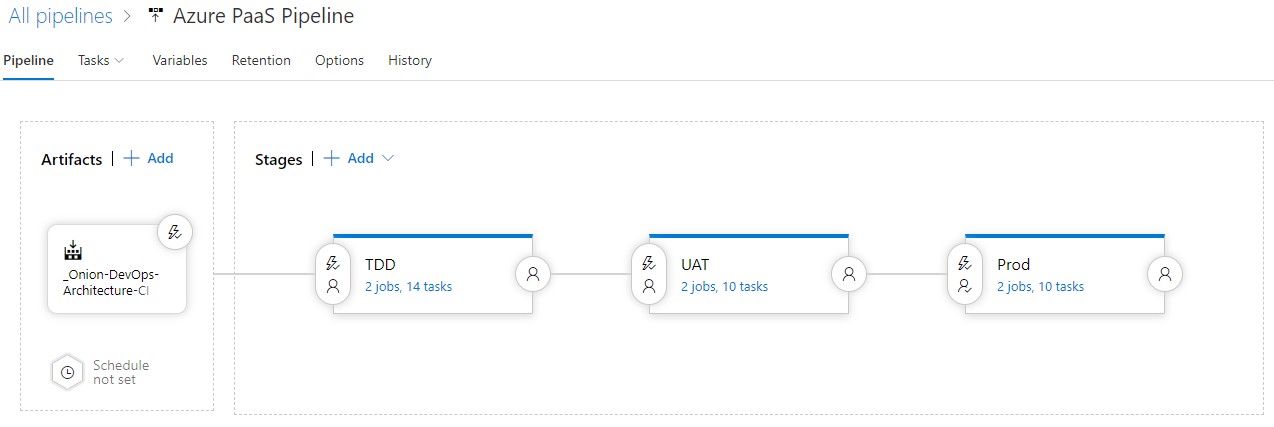


Figure: Our release configuration contains three environments and is triggered from the CI Build

We see that there are four key parts to our pipeline’s release configuration. Let’s take them one at a time.

* **Artifacts**  
  The release needs to know what artifacts are available to it. There are several options in the tool, but this is where you will specify the build configuration that represents the CI Build for your application. You will configure the release to automatically begin upon success of that build.
* **TDD Stage**The release can have multiple stages in series, in parallel, or both. This is the smallest, shortest pipeline you will have for any of your applications. The TDD stage corresponds with the TDD environment that is completely automated and where your automated full-system acceptance tests run.
* **UAT Stage**The UAT stage represents the deployment of the application to the UAT environment.
* **Prod Stage**The Prod stage represents the deployment of the application to the Production environment.

Next, we will move through each of the screens that need to have some configuration set for them.

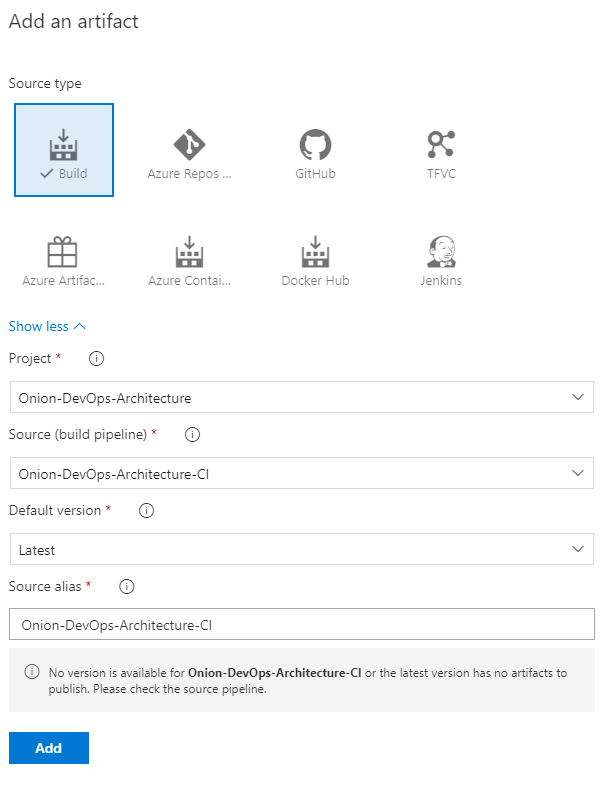


Figure: Specify the CI build that will be triggering the release.

The above figure contains the settings needed to wire up a CI build with an auto-triggered release. Use the “Latest” default version so that your release configuration works with any build from any branch that might be active. In this way, you can maintain a single CI build configuration and a single release configuration.

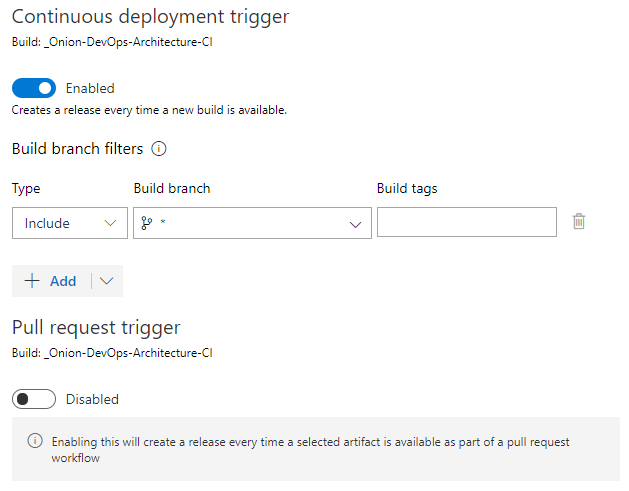


Figure: Enable continuous deployment to automatically trigger the release.

The property page of the build artifact contains two important settings. The first is to enable creating a release every time a new build is available. The second important setting is the Build branch filters. **While this might appear as a drop down, put the mouse cursor in it and type an asterisk (\*)**. This will ensure that builds that come from every branch will cause the release to trigger. You need this in order to deploy the pipeline to the TDD environment and run acceptance tests. Each stage/environment also includes branch filters so that you can exclude branch-based release candidates from progressing any further down the pipeline.

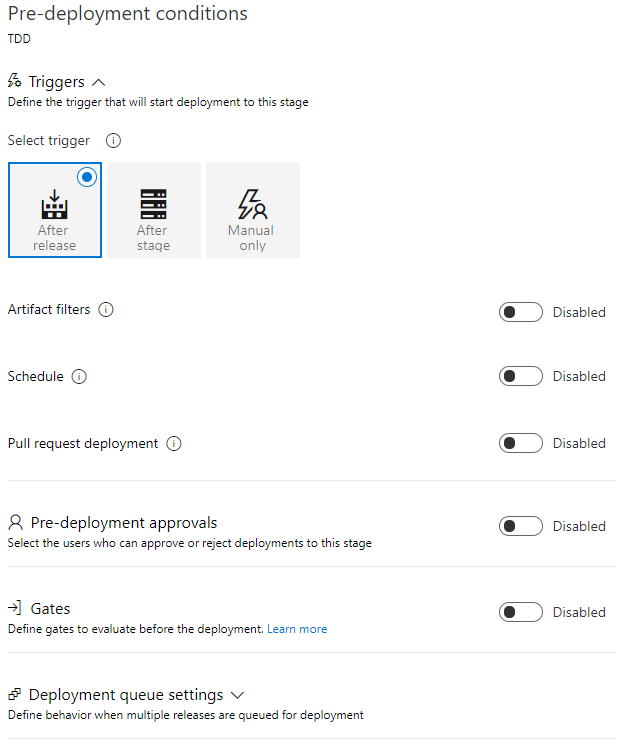


Figure: The pre-deployment conditions for the TDD environment.

For the TDD environment, leave all the settings at their defaults and make sure that this environment automatically triggers after release. This is the only setting you need. Azure Pipelines has filters and different logic points in several places, so unless you are changing it for a reason, leave the defaults as they are.

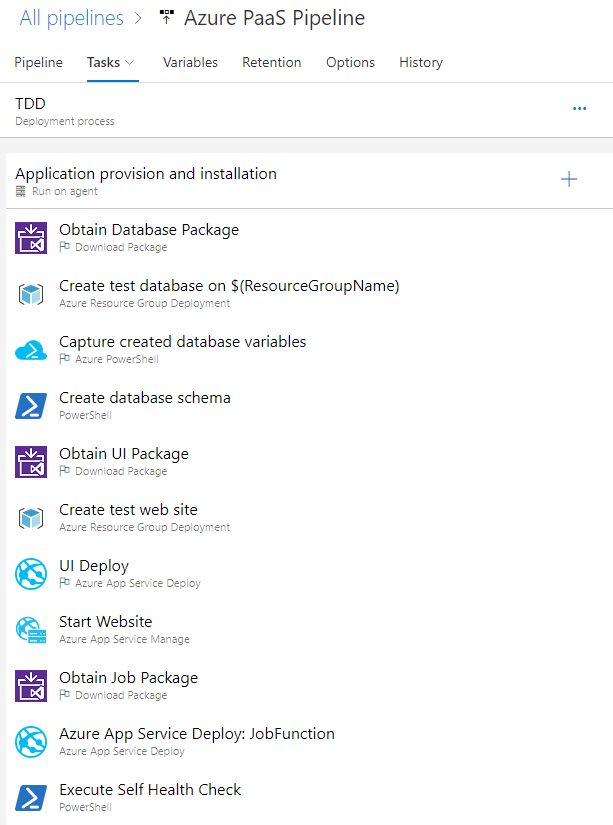


Figure: The TDD environment deployment process

When you configure the TDD environment, you’ll have three sections because our application has three components that are packaged. We have the website, the off-line job, and the SQL database. This portion retrieves the Nuget package for the application component being installed. It extracts and installs that component properly and moves on to the next component. At the end, you execute the health check that calls the appropriate URL or API so that the application can run the built-in routine that checks to see if everything has started and is online.

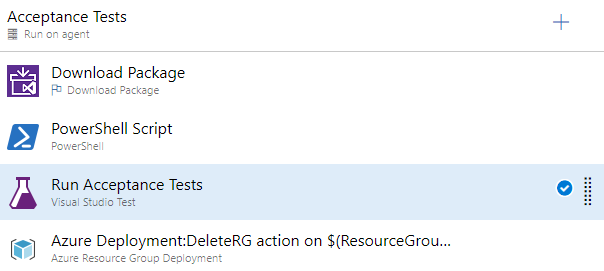


Figure: The second major task for the TDD environment is to run the acceptance tests

Each stage (think environment) can have multiple “jobs” configured. The second configured job is all about running some type of automated validated. Our application has a full-system acceptance test suite that uses Selenium to drive a web browser. The tests operate within the NUnit runtime and are executed through the VsTest adapter. Since our tests are packaged in a Nuget package and stamped with the same version number as the rest of the application, we use the same method to retrieve the right test suite for execution. After the tests run, pass or fail, we run an Azure Resource Manager deployment and destroy the resource group with the TDD environment that was created at this stage in our DevOps pipeline. That last step is important because you will be executing multiple build and release cycles per day per developer. If you don’t remove the resources in Azure that are created, you will run up an Azure bill; however, you are likely to hit Azure subscription limits, which will cause your deployment to fail with an error that will take time to debug.

Creating the TDD environment with ARM

The example application that comes with this book contains ARM (Azure Resource Manager) templates that will create the environments that are suitable for deployment. The TDD environment does not need environments that are scaled to the level that Production needs. Because of this, it can often work just fine to provision a scaled down environment by parameterizing the tier of resources to choose. For example, in our TDD environment, we create an App Service resource using the Free tier. We can do this over and over, run our acceptance tests, and validate that the new release candidate continues to function well. But we need to take care that we destroy the environment when we are finished with it. If we fail to do this, we won’t run up our bill because we are using the Free tier. But Azure has a limit of how many Free tier resources can be active at any one time. These limits are not guaranteed and can be changed at any time. Needless to say, if you are perpetually creating new environments without destroying them, you will encounter the limit regardless of what level it happens to be. Then your pipeline will begin to fail, and the error message may or may not lead you to the root cause quickly. The best rule of thumb is to clean up after yourself. For the TDD environment, this means destroying the environment and the end of the deployment process, after you have run the appropriate test suites.

While this chapter cannot highlight every setting in the release configuration’s steps, this book’s digital resources includes full exports of the builds and release configurations. In addition, you can refer to the Azure DevOps Services public project at <https://dev.azure.com/clearmeasurelabs/Onion-DevOps-Architecture>.

### Deploying an application component

Before we move on, let’s look in detail at the process of retrieving an application component from Azure Artifacts and deploying it. For this section, we are going to select the most complicated component of most applications. This is the SQL Database. In order to deploy the database, we will need to have all required assets available to us, and we will need to create the database itself in the TDD environment.

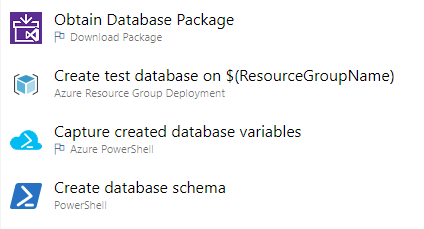


Figure: The four steps that make up a provisioning and deployment of the SQL database

The four steps in the above figure are responsible for on-demand provisioning of a SQL database and the creation of the schema. Below is the full YAML listing.

At the time of writing, YAML configuration is available for builds but not for release configurations.

steps:

- task: DownloadPackage@0

displayName: 'Obtain Database Package'

inputs:

feed: '<some guid>'

definition: '<some guid>'

version: '$(Build.BuildNumber)'

downloadPath: '$(System.ArtifactsDirectory)/packages

/Database-$(Build.BuildNumber)'

variables:

ResourceGroupName:

'$(System.TeamProject)-$(Release.EnvironmentName)-$(Release.ReleaseId)'

DatabaseUser: 'dbuser'

DatabaseName:

'db-$(Release.EnvironmentName)-$(Build.BuildNumber)-$(Release.ReleaseId)'

DatabaseEdition: 'Basic'

DatabasePerformanceLevel: 'Basic'

- task: AzureResourceGroupDeployment@2

displayName: 'Create test database on $(ResourceGroupName)'

inputs:

azureSubscription: '<redacted>'

resourceGroupName: '$(ResourceGroupName)'

location: 'South Central US'

csmFile: '$(System.ArtifactsDirectory)/packages

/Database-$(Build.BuildNumber)/DatabaseARM.json'

overrideParameters: '-databaseLogin $(DatabaseUser)

-databaseLoginPassword $(DatabasePassword) -skuCapacity 1

-databaseName $(DatabaseName) -collation SQL\_Latin1\_General\_CP1\_CI\_AS

-edition $(DatabaseEdition) -maxSizeBytes 1073741824

-requestedServiceObjectiveName $(DatabasePerformanceLevel)'

variables:

ResourceGroupName:

'$(System.TeamProject)-$(Release.EnvironmentName)-$(Release.ReleaseId)'

- task: AzurePowerShell@3

displayName: 'Capture created database variables'

inputs:

azureSubscription: '<redacted>'

ScriptType: InlineScript

Inline: |

$azureRmResourceGroupDeployment = Get-AzureRmResourceGroupDeployment

-ResourceGroupName "$(ResourceGroupName)" | Sort-Object Timestamp

-Descending | Select-Object -First 1

$azureRmResourceGroupDeployment.Outputs.GetEnumerator() | ForEach-Object {

$variableName = $\_.key

$variableValue = $\_.value.Value

Write-Host

"##vso[task.setvariable variable=$variableName;]$variableValue"

Write-Host "$variableName $variableValue"

}

azurePowerShellVersion: LatestVersion

- powershell: |

$env:DatabasePassword="$(DatabasePassword)"

& $(System.ArtifactsDirectory)\packages

\Database-$(Build.BuildNumber)\UpdateAzureSQL.ps1

workingDirectory: '$(System.ArtifactsDirectory)\packages

\Database-$(Build.BuildNumber)'

displayName: 'Create database schema'

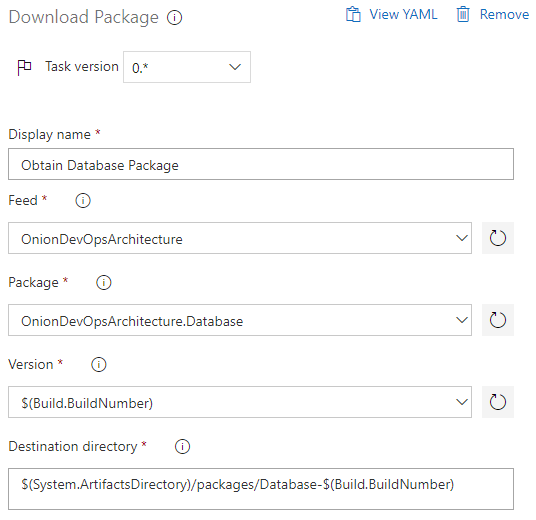


Figure: Download Package step configuration

The above YAML configuration can be quite cryptic if you haven’t worked with it much. This figure represents the customized properties of the Download Package task. This task downloads the specified Nuget package from Azure Artifacts and expands the contents into the Destination Directory specified. Keep in mind that we will have multiple releases happening at once on multiple branches, so we want to parameterize anything that creates environments so that we don’t accidentally create any global dependencies. In order to obtain the Nuget package for the release candidate we want, we specify the full name of the package: OnionDevOpsArchitecture.Database. Then, we must specify the Version that we would like to retrieve. In this context, we have the current $(Build.BuildNumber) available to us, so we specify that.

Because we configured our CI build as an artifact dependency, we have the variables of that build available to us in the release configuration. The variable we will use throughout the pipeline is $(Build.BuildNumber). Everything hinges on the build number.

If you are someone who likes to know exactly what is happening at all times under the covers, you may want to install the Azure Pipelines agent to your own workstation so you can closely observe the directories used and where files are placed. For historical reasons, the agent for most people will be downloaded in a file name “vsts-agent-win-x64-#-#.#.zip” where # is the latest version number. If you are experimenting, you are free to install as many instances of the running agent as you like. Make sure you extract and run the zip file to separate locations in order to do that.

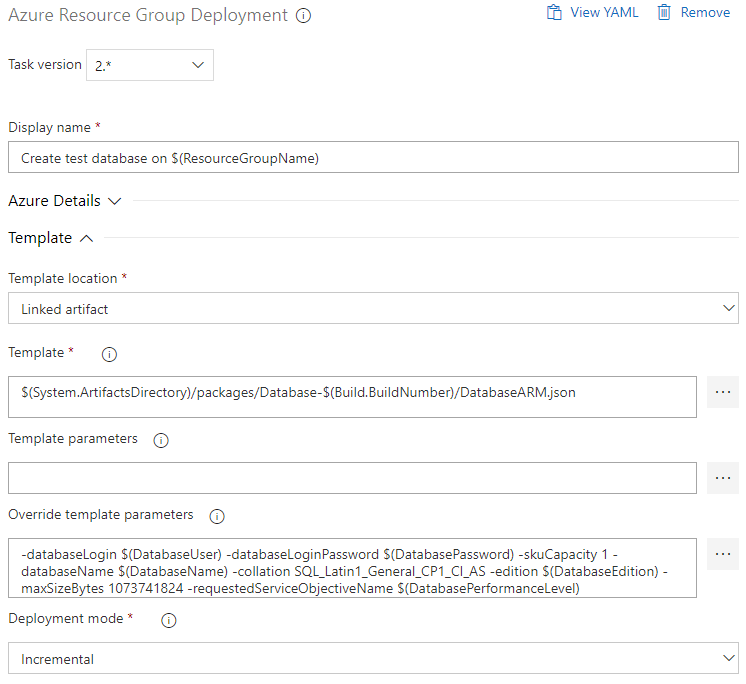


Figure: Creating the SQL Database in Azure uses an ARM template

The next step for our database deployment is to create the Azure SQL Database in our TDD environment. At the top, you will notice $(ResourceGroupName). We haven’t covered variables yet, but this variable is set to the value of

$(System.TeamProject)-$(Release.EnvironmentName)-$(Release.ReleaseId)

We do not want to hard-code these things. Many TDD environments may be provisioned simultaneously since feature branches make use of environment creation and acceptance test execution. We take enough other variables and construct a resource group name that is guaranteed to be unique. Every Azure resource we create for this versioned deployment in the TDD environment will go in this resource group. When we are finished, we destroy the resource group, and we can be confident that we have cleaned up appropriately. Notes that we embed three pieces of information in this variable:

* Team project name (by convention, we never put spaces in a team project name)
* Environment name
* Release ID (we don’t use BuildNumber since resource group names can’t contain dots)

As we specify the other properties, we didn’t have enough space to show the basic Azure settings, but you’ll specify your authenticated Azure subscription and the region you are targeting. Then you will specify the path to the ARM template file. This file was extracted by the OnionDevOpsArchitecture.Database Nuget package, so you have it available to you. This ARM template is stored in the Git repository and is owned by the “Database” Visual Studio project. You can reference the full file details by downloading the code that accompanies the book. This ARM template has some variables that have been externalized as parameters so that the deployment process can control the settings. You can see the “Override template parameters” text area. We are specifying several of these settings in order to make the ARM template generic and reusable across other applications that need an Azure SQL database. Through these variables, we can control the database edition, size, etc.

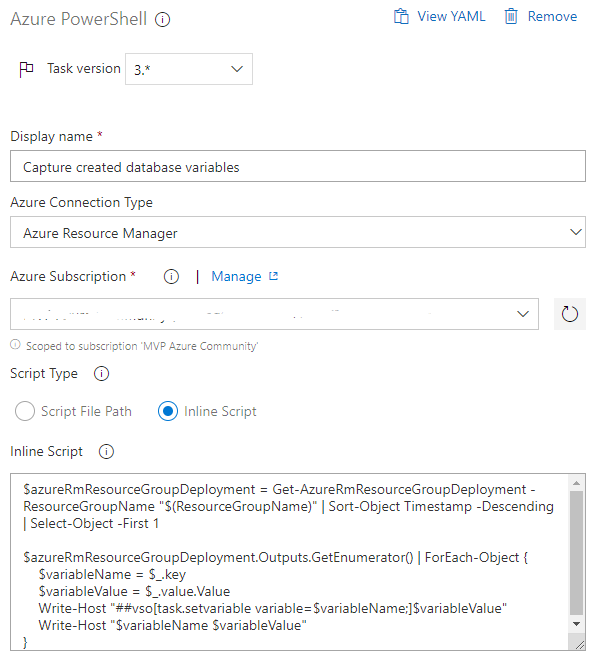


Figure: We capture output variables after our SQL database is created.

The next step is to capture the variables that we need in order to create our database schema. We have just created a new Azure SQL database, but we don’t know how to access it. A new SQL Database Server must be created to house any databases, and those always have a unique host name. From the execution of our ARM deployment, we loop through the outputs of the resource group deployment and capture them as variables that can be used in subsequent steps of our deployment. In this case, we will capture an output named “resourceGroupUniqueString”. Because of this, we now have the server name that can be used to execution our schema migration tool. We construct a variable $(DatabaseServer) by using the following value.

databaseserver$(resourceGroupUniqueString).database.windows.net

Azure SQL uses this pattern for host names to the database server. With this variable captured, we can proceed to access our newly created database server.

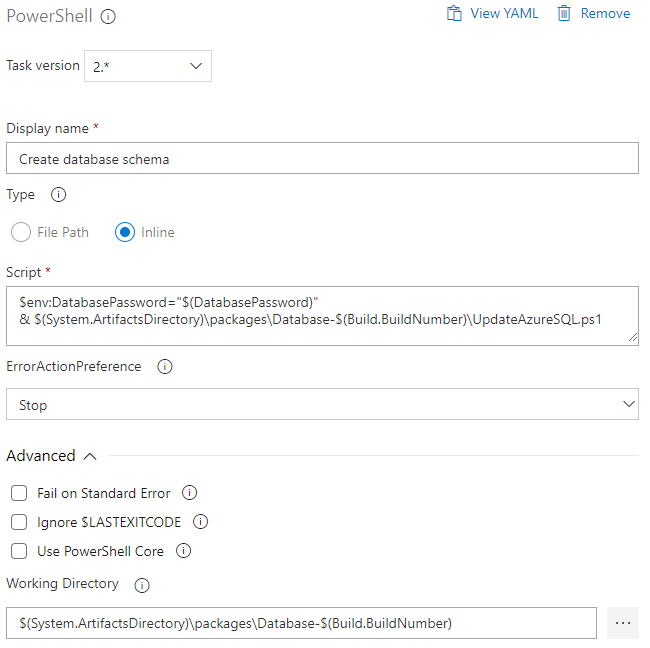


Figure: The step of our database deployment that creates the full database schema in the TDD environment

This PowerShell task is another example of “shift left” as we take the logic that needs to execute and push it into our Visual Studio solution. In order to run it, we need to make available the sensitive credential stored in the $(DatabasePassword) variable. Variables marked “secret” do not automatically become environment variables. Our PowerShell snippet explicitly makes it available as an environment variable to the current process. Other variables in plain text are automatically available as environment variables. The UpdateAzureSQL.ps1 file is part of the code and comes from the Database nuget package that was stored in Azure Artifacts with our release candidate. The contents of this \*.ps1 is below.

#

# UpdateAzureSQL.ps1

#

$DatabaseServer = $env:DatabaseServer

$DatabaseName = $env:DatabaseName

$DatabaseAction = $env:DatabaseAction

$DatabaseUser = $env:DatabaseUser

$DatabasePassword = $env:DatabasePassword

Write-Host "Executing & .\scripts\AliaSQL.exe $DatabaseAction $databaseServer

$databaseName .\scripts $databaseUser $databasePassword"

& .\scripts\AliaSQL.exe $DatabaseAction $DatabaseServer $DatabaseName

.\scripts $DatabaseUser $DatabasePassword

The PowerShell scripts is very straightforward. We pass in location and credentials in order to access the SQL Server and create or update a particular database name. Our database migration tool, AliaSQL[[3]](#footnote-3) accesses the script stored in the .\scripts\ folder and executes them in order to build the database schema.

Several automated database migration tools exist at the time of writing. The best commercial tool is Redgate’s SQL Change Automation (SCA). Other free OSS options are DbUp and Roundhouse.

One more setting that is important to running PowerShell scripts that are bundled in your release candidate’s Nuget package is the working directory. You can see in the above Figure, we set the working directory to be the directory where our Nuget package has been extracted. By doing this, the authoring and maintenance of the PowerShell script is simplified. It’s a normal assumption that the script would use relative paths to the path where it exists. By doing this, our script can work well in all of the places and environments where it may be executed. With the execution of this command, our TDD environment now has a complete SQL Server database with the full schema and schema data loaded. We have the full connection string, and it’s ready for use. The other application components follow the same pattern:

* Retrieve Nuget package
* Extract Nuget package in a working directory
* Poke any configuration variables
* Provision server/cloud environment
* Install application component
* Start application component

Now that you know the pattern to install your application, let’s turn to automatic validation in the TDD environment.

### Running test suites using a release configuration

Now that our application is deployed, the value of the TDD environment is the automatic execution of our acceptance tests. Our example application that comes with this book contains a number of simple acceptance tests, validating that we can add new Expense Reports and list them out on the screen. We use the NUnit test framework with Selenium’s web driver through the Chrome browser. Here is a listing of our test code.

using System;

using System.IO;

using System.Reflection;

using ClearMeasure.OnionDevOpsArchitecture.Core.Model;

using ClearMeasure.OnionDevOpsArchitecture.IntegrationTests;

using NUnit.Framework;

using OpenQA.Selenium;

using OpenQA.Selenium.Chrome;

using Shouldly;

namespace ClearMeasure.OnionDevOpsArchitecture.AcceptanceTests

{

public class GetAllExpenseReportsTester

{

private string \_appUrl;

private IWebDriver \_driver;

[OneTimeSetUp]

public void Setup()

{

\_appUrl = new DataConfigurationStub().GetValue("AppUrl",

Assembly.GetExecutingAssembly());

\_driver = new ChromeDriver(".");

new ZDataLoader().LoadLocalData();

}

[OneTimeTearDown]

public void Teardown()

{

\_driver.Close();

\_driver.Quit();

\_driver.Dispose();

}

[TestCase("000001", TestName = "Should add new expense report numbered '000001'")]

[TestCase("000010", TestName = "Should add new expense report numbered '000010'")]

[TestCase("000100", TestName = "Should add new expense report numbered '000100'")]

[TestCase("001000", TestName = "Should add new expense report numbered '001000'")]

[TestCase("010000", TestName = "Should add new expense report numbered '010000'")]

[TestCase("100000", TestName = "Should add new expense report numbered '100000'")]

public void ShouldBeAbleToAddNewExpenseReport(string expenseReportNumber)

{

void ClickLink(string linkText)

{

\_driver.FindElement(By.LinkText(linkText)).Click();

}

void TypeText(string elementName, string text)

{

var numberTextBox = \_driver.FindElement(By.Name(elementName));

numberTextBox.SendKeys(text);

}

Console.WriteLine($"Navigating to {\_appUrl}");

\_driver.Navigate().GoToUrl(\_appUrl + "/");

\_driver.Manage().Window.Maximize();

TakeScreenshot($"{expenseReportNumber}-Step1Arrange");

ClickLink("Add New");

TypeText(nameof(ExpenseReport.Number), expenseReportNumber);

TypeText(nameof(ExpenseReport.Title), "some title");

TypeText(nameof(ExpenseReport.Description), "some desc");

TakeScreenshot($"{expenseReportNumber}-Step2Act");

\_driver.FindElement(By.TagName("form")).Submit();

TakeScreenshot($"{expenseReportNumber}-Step3Assert");

var numberCells = \_driver.FindElements(

By.CssSelector(

$"td[data-expensereport-property=\"{nameof(ExpenseReport.Number)}\"]

[data-value=\"{expenseReportNumber}\"]"));

numberCells.Count.ShouldBeGreaterThan(0);

numberCells[0].Text.ShouldBe(expenseReportNumber);

}

private void TakeScreenshot(string fileName)

{

var chromeDriver = ((ChromeDriver) \_driver);

chromeDriver.GetScreenshot().SaveAsFile($"{fileName}.png");

TestContext.AddTestAttachment($"{fileName}.png");

}

}

}

We use a DataConfigurationStub() in order to clear out the database in the TDD environment and preload it with a few records. We use the same NUnit test to run six test cases. The steps his test progresses through are:

1. Navigate to the home page
2. Find and click on the “Add New” link
3. Find the Number text box and type in a value
4. Find the Title text box and type in a value
5. Find the Description text box and type in a value
6. Submit the form
7. Find the row of the table and the Number column
8. Make sure the value of the Number is the expected value.

This test follows the Arrange, Act, Assert[[4]](#footnote-4) convention that is a bedrock principle of test-driven development. As this test executes, it will open the local Chrome browser on the server and execute these steps. Let’s take a look at how this is configured in Azure Pipelines.

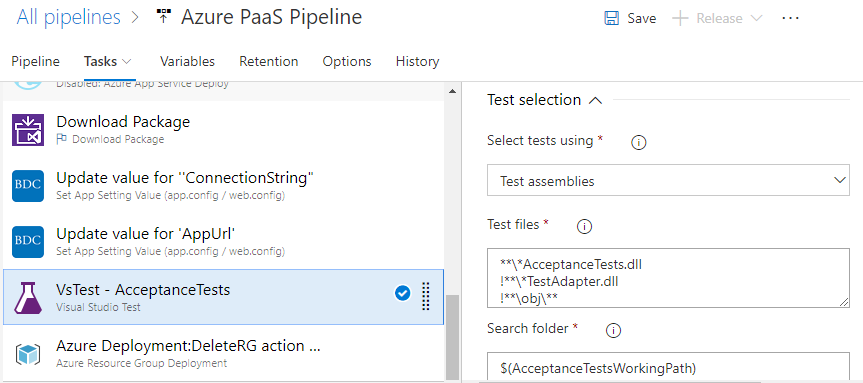


Figure: Full system acceptance tests are run just like any other NUnit/XUnit test suite.

Once we download the Nuget package, which contains our acceptance test suite, perform a few key steps.

1. We extract the package onto a working path on our Azure DevOps agent server
2. ConnectionString and AppUrl config settings are poked into the test suite’s configuration file[[5]](#footnote-5)
3. VSTest task is run against our \*AcceptanceTests.dll assembly, which contains our tests.
4. No fourth step – because we use VSTest, the test output is automatically captured by Azure DevOps as a test run.

Before we proceed further, let’s examine the Nuget package for our acceptance tests.

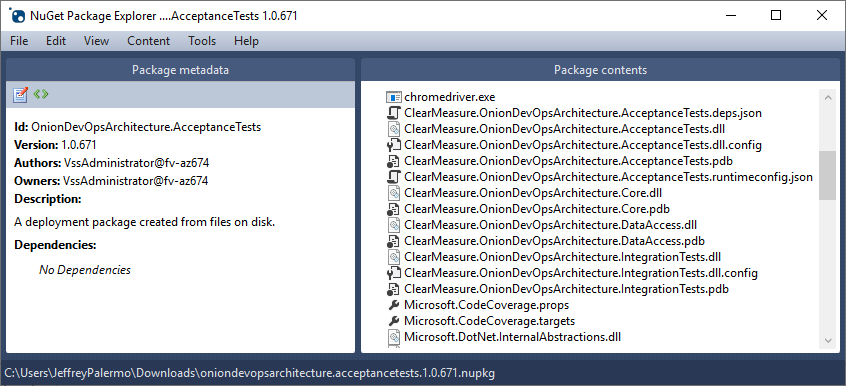


Figure: The acceptance test package contains the Selenium driver as well as the test assemblies.

Notice that chromedriver.exe is contained in the package as well as the test assemblies and the config files that go with them. There are more dependent assemblies that don’t fit in the screenshot, but everything necessary for the tests to run is here. Besides the NUnit test code and the VSTest Azure Pipelines task, we are also integrating some of the built-in features of Azure Test Plans. Notice in the previous code the line:

TestContext.AddTestAttachment($"{fileName}.png");

Azure Test Plans keep track of all test runs, the tests, and the results of each. And each test that is run can archive any arbitrary file attachment. In the case of full system acceptance tests that run through a browser UI, one of the most useful attachments is a screenshot of every screen the test sees as it runs. Our test scenario in C# is instrumented with calls to the ChromeDriver to take a screenshot, and then we save the file and attach it to the TestContext. When VSTest runs these tests, it collects all the information and archives it in Azure Test Plans. Let’s take a look at this one step at a time. First, we can see the results of our CI Build.

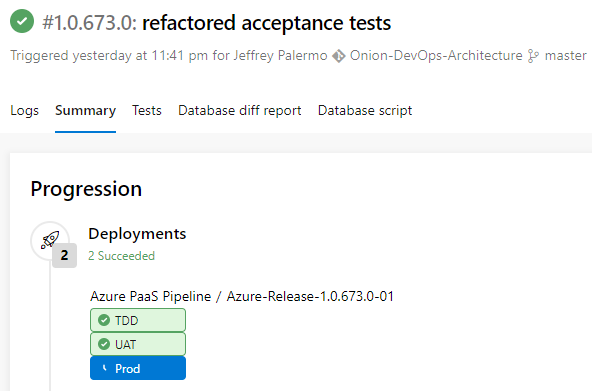


Figure: The build summary page shows that this build has been deployed across environments.

On this build summary page, we can see the successful build and that the deployments to TDD and UAT are successful. We can also see that the deployment to Prod is ready and waiting (but will not proceed without a manual approval).

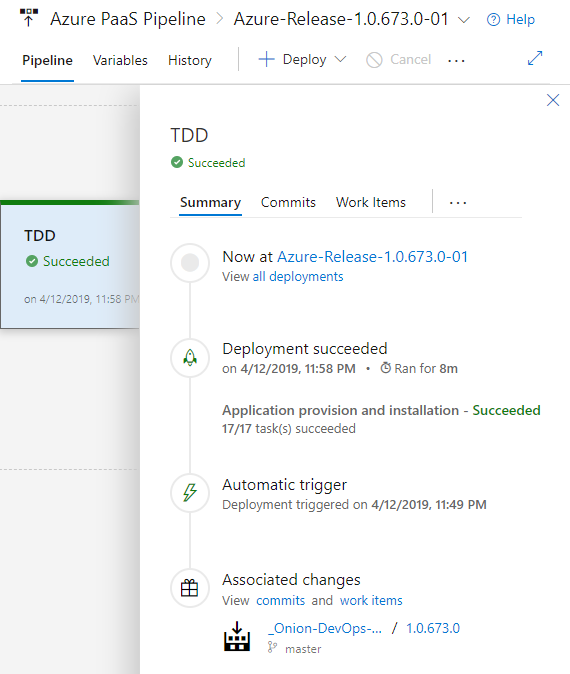


Figure: The TDD release view shows the top-level details of the TDD deployment.

When we click over to our TDD deployment from the build page, we can see information about the release and drill into each environment to see details about what has happened.

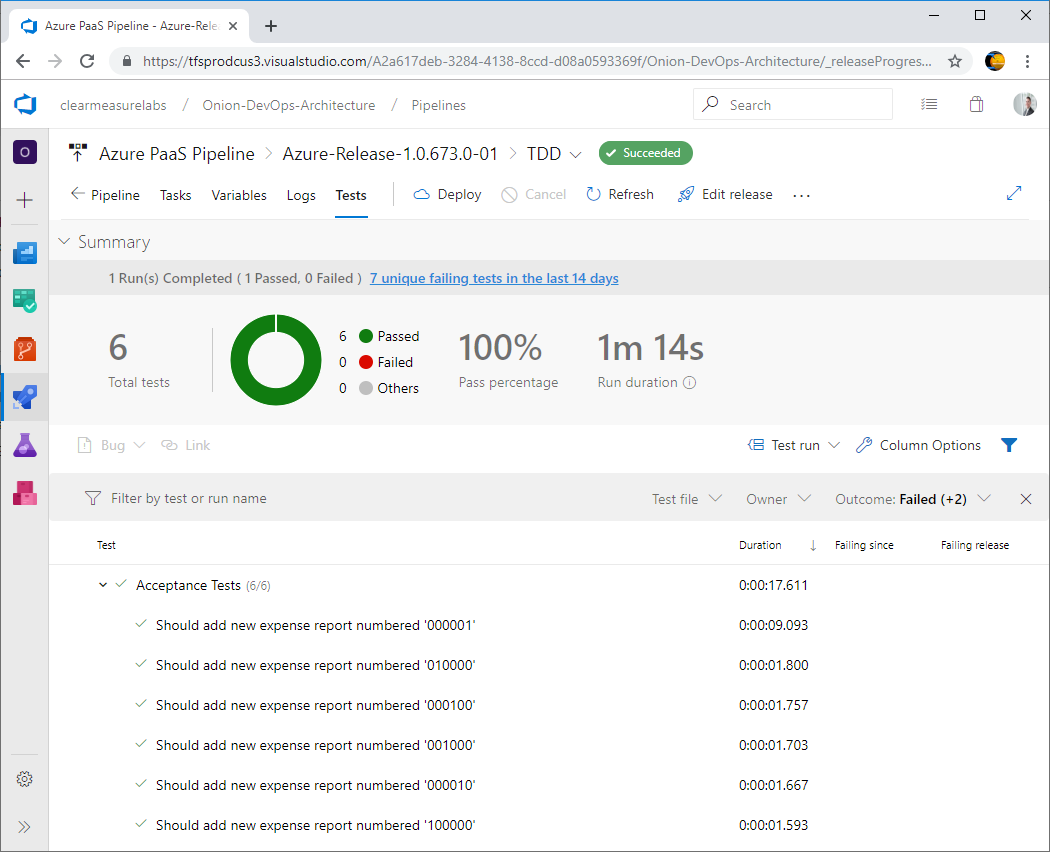


Figure: The Tests tab gives us access to the acceptance tests that have run.

Beyond looking at the Logs of your deployment, which is critical in debugging it until it works properly, the Tests tab is your access to the world of the acceptance tests. In our example, we have six tests. In business-critical applications, you will have over a hundred. A good rule of thumb is to ask yourself if every textbox and every button on every screen is being interacted with by your acceptance tests. You don’t want basic functionality gaps. You don’t have to look for every edge case, but you do want basic coverage. Next, let’s select and click on the last test in the list, “Should add new expense report numbered ‘100000’.

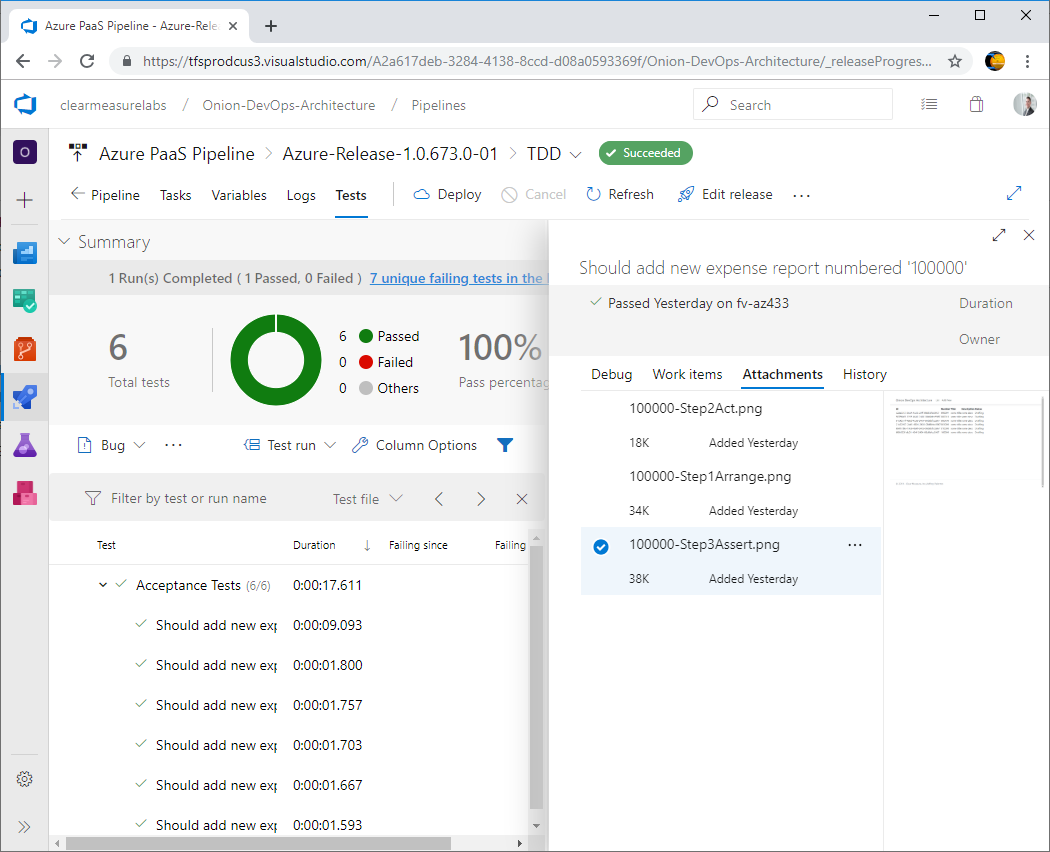


Figure: Selecting a test gives us more information about that test and the run of it.

While you can associate work items with tests and see the history of it, most of the time you want to run tests and just know that they passed. The value comes when a new commit breaks a test, and now you have to figure out why. This is where attachments come in. Because we took screenshots while the Selenium tests were running, we can refer back to them if something goes awry. You can see here in this figure that we can see a preview of what the screen looks like when we are running the asserts of the test. Let’s look at that screenshot more closely.

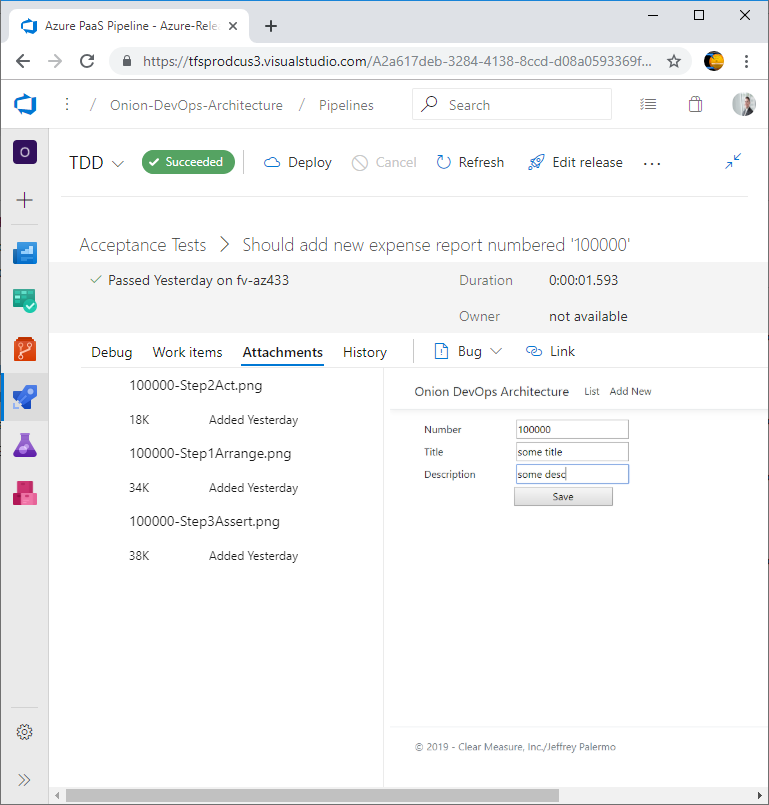


Figure: Expanding the attachment view shows a full preview with the ability to link a work item to log a bug.

With this capability, if the test fails, we can see what has changed on the screen. Because you are finding elements with CSS selectors or names, you may choose to store the full HTML page source as an attachment. It is completely up to you.

With Selenium and ChromeDriver, you can capture a screenshot of the page as your acceptance tests are running. Everything inside the window will be captured and stored as an attachment to the test run.

While Azure Test Plans is not in the scope of this book, the data on the test runs themselves enables more dashboard and analytics inside the Azure Test Plans product. As you accumulate more tests, you will likely start asking questions such as:

* Why do some types of tests seem more brittle than others?
* Why are some tests slower than others?
* Why did some tests get removed?
* Why is our ratio of tests to code changing over time?

Without these metrics, you don’t have the ability to consider these questions.

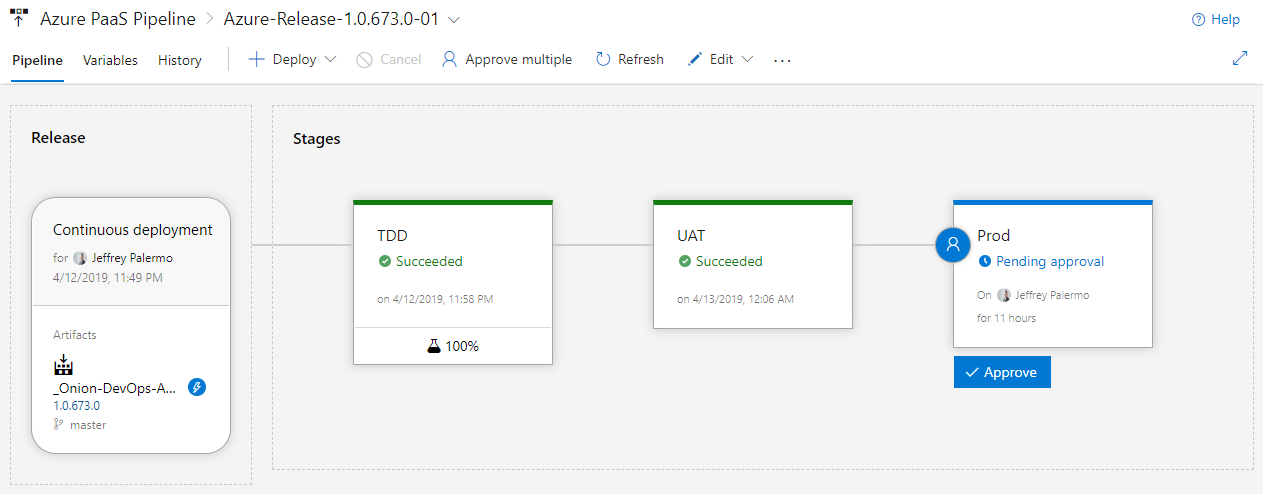


Figure: The complete pipeline looks like this when it is functioning properly.

Once your TDD environment is being created, configured, deployed, and tested, the other deployments in your pipeline follow suite. You can create and deploy as many environments as you like depending on the audience of people who need to participate in software validation and testing. You will see a 100% next to the beaker icon on the TDD environment. This signifies that a test suite ran and that all tests passed. It would show less than 100% if any tests were ignored.

### Differences in the UAT and Production environments

While the deployment process for the TDD environment should be the same as UAT and Prod, you will have some key differences in order to maintain all the branching capabilities for trunk-based development. First, the UAT state deployment needs to be configured to ignore release candidates generated by feature branches.

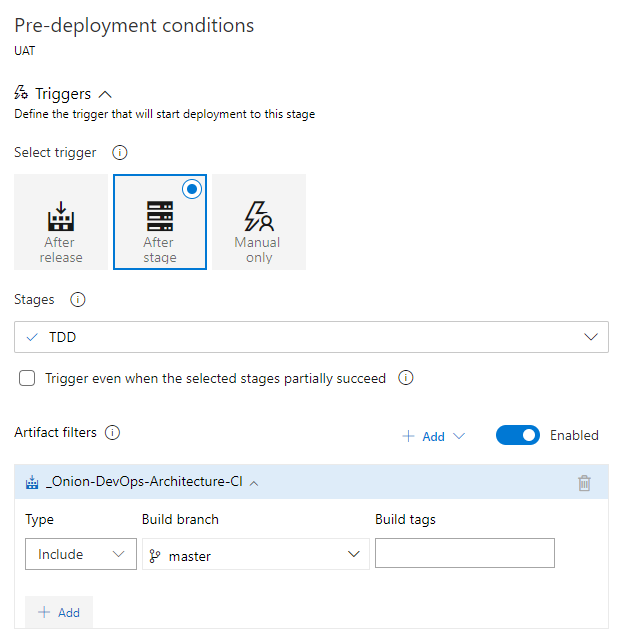


Figure: The UAT environment should only deploy release candidates generated by the master branch.

Every environment depends on the release candidates generated by a particular build. You can filter how a deployment is triggered by adding filters. In this case, we want the UAT environment to be automatically deployed when the TDD stage succeeds but only if the release candidate came from the master branch. Most of the time, stakeholders outside the development team want very stable release candidates that are fully integrated. It is your choice to modify this if it is appropriate for deploying some of these feature branch release candidates. Other differences in the UAT stage are not process differences but variable differences.

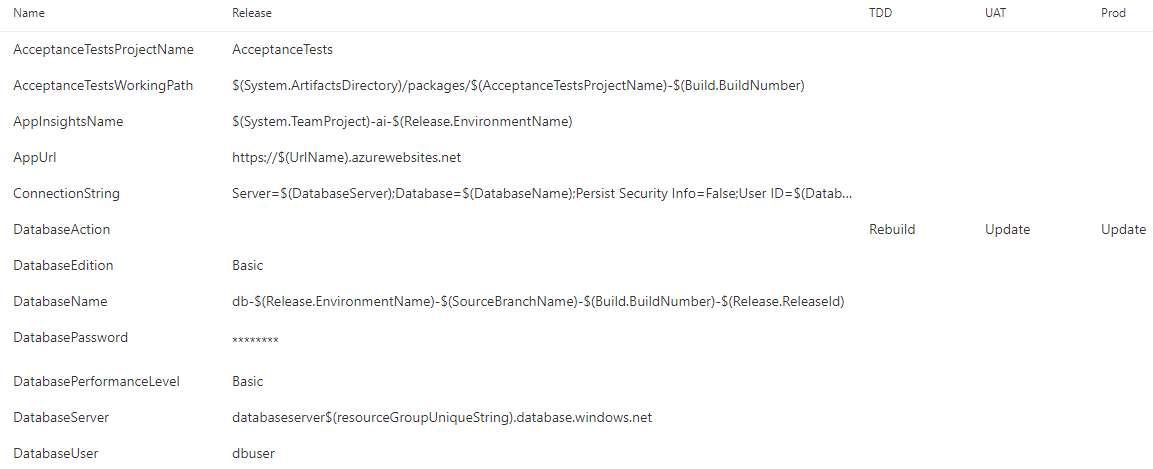


Figure: Variables can be different per stage or the same throughout the release.

The above figure is a subset of the variables used to deploy and test the application along our DevOps pipeline. Most of the variables tend to be the same in every stage because the environment name is able to be used to construct variable values. For example, take notice of the DatabaseName variable. No matter how many environments or how many feature branches are active any one time, we generate unique database names to prevent collisions. The environment name and even the branch name are embedded in the DatabaseName so that we can provision as many environments as needed. The DatabaseAction is different. The DatabaseAction variable has no default value for the whole release. Instead, we specify different values so that our process runs our database schema migration tool with the right command line arguments. In the TDD environment, we want a completely new database built from scratch. This proves that we aren’t relying on anything in the environment in order to have a database that can pass our acceptance tests. In the UAT environment, we do not rebuild the database. Rather, we preserve the database and data and update it by running only the \*.sql files that have not yet been run on that environment. This is signified by the “Update” value. When this completely successfully, we have a high degree of confidence that when we run the same routine on production, the data will be preserved, and the schema will be updated properly. Next, examine the different configuration for the Production deployment stage.

Because you’re going to be creating many releases, you need variable values that are going to be resilient to the repetitive nature of DevOps

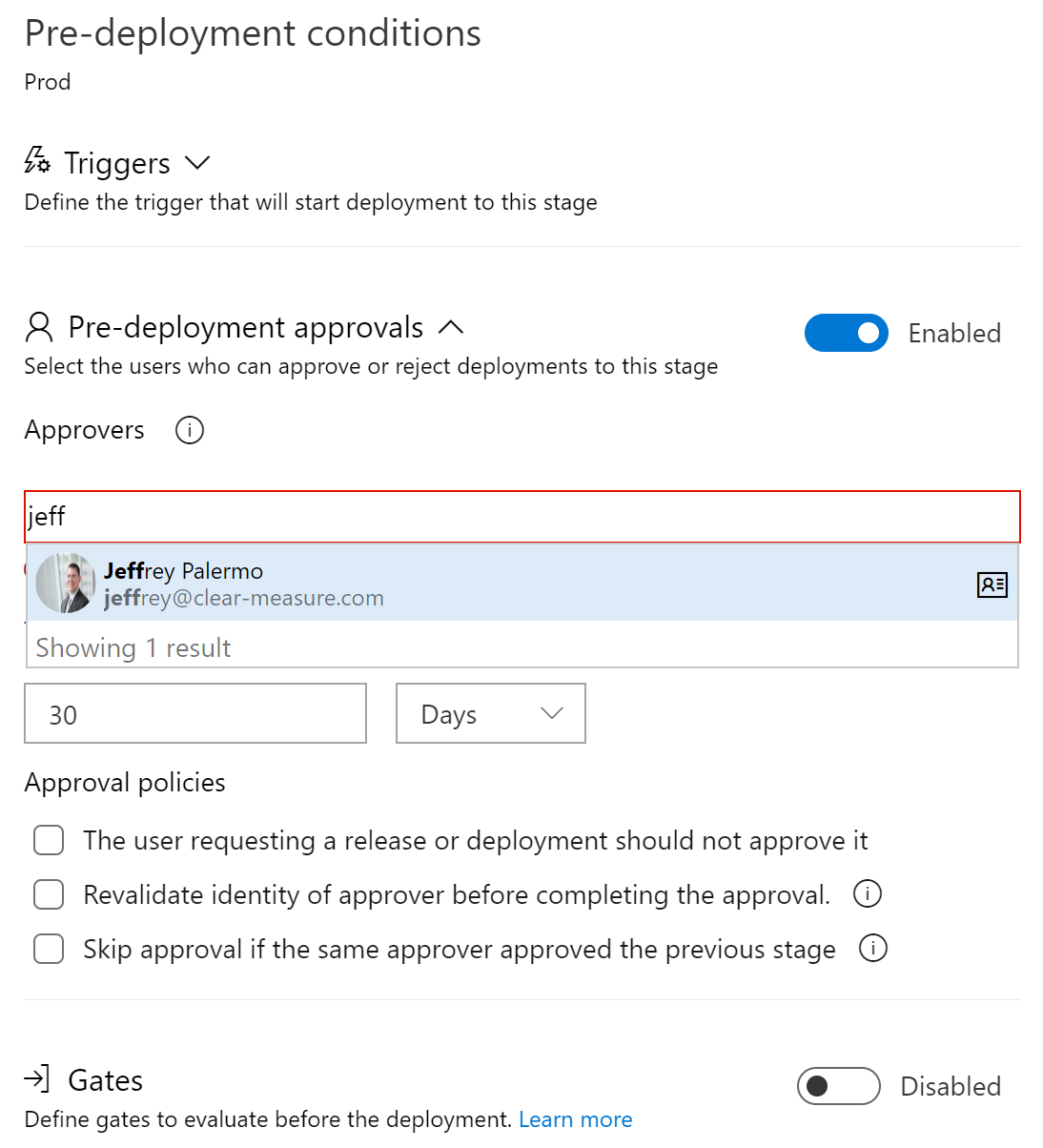


Figure: The Production configuration specifies an Approver.

The approver for a release stage can be an individual, multiple individuals or a group. You have several options. In addition, you can enable the Gates feature which provides the ability to build in some business logic to determine if the deployment should be allowed to proceed. The combination of configuration options provides a robust method by which to restrict the ability to automatically or manually, with approval, deploy to the Production environment. Upon approval and the satisfaction of any gates or filters, the deployment stage is queued.

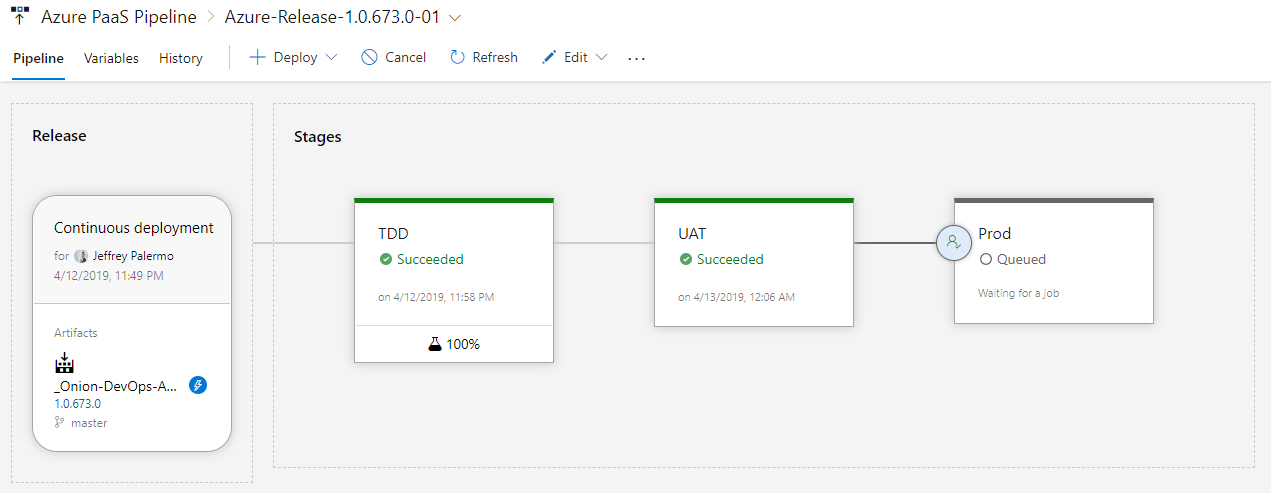


Figure: The Production stage will queue the deployment as soon as manual approval is given.

If a previous release has been placed in the approval queue, the new release will wait for it. If you see the above screen unexpectedly, look for a previous release that has not been approved. Chances are that some other releases are in the queue for this release stage.

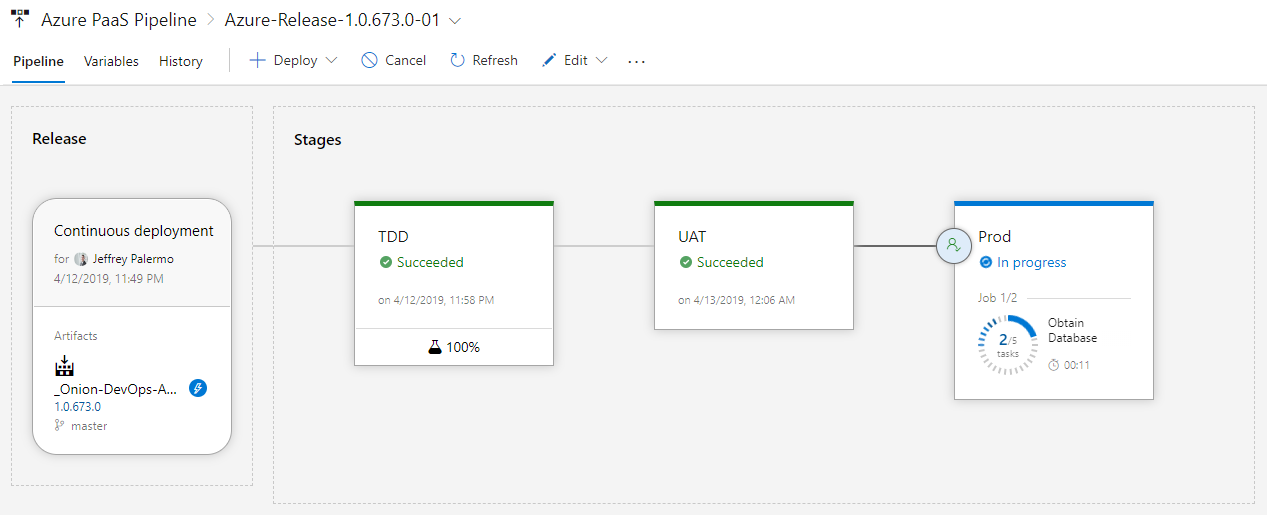


Figure: You can see the progress of the deployment when it is executing.

Upon approval of the production deployment, you will see the progress auto-updating on the screen. You can watch it as it executes or close the window and come back later. Because you have placed a call to the built-in application health check, you will know, if the deployment reports success, that all parts of the application along with their dependencies have been deployed and that everything is online and functioning properly.

## Wrap up

In this chapter, you learned how to design your deployment pipeline, and you saw the experience of executing a properly configured release configuration. It’s important to determine the correct number and type of environments. You also assigned validation steps to each environment including a built-in application health check. You learned about the different types of data that is deployed or provisioned with a deployment, and you reviewed the different options you have within Azure for running code in Azure PaaS services or others. Finally, you saw the various touch points in the release configuration, including the impact of variables on the execution of the deployment steps. And you learned how to integrate a full system acceptance test suite into your TDD environment, both for releases from feature branches as well as from master. Next, we’ll take a look at how to properly operate, monitor, and support our applications as they run in production and are used by our customers.

# Bibliography

Beck, K. (2002). *Test Driven Development: By Example.* Addison-Wesley Professional.

*Build & Release Tools from Benjamin Day*. (n.d.). Retrieved from https://marketplace.visualstudio.com/items?itemName=bendayconsulting.build-task

*JeffreyPalermo.com*. (n.d.). Retrieved from AliaSQL – the new name in automated database change management: https://jeffreypalermo.com/2014/01/aliasql-the-new-name-in-automated-database-change-management/

Kruchten, P. (n.d.). Retrieved from Architectural Blueprints—The “4+1” View Model of Software Architecture: https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf

*Shift Left to Make Testing Fast and Reliable*. (n.d.). Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/shift-left-make-testing-fast-reliable

1. (Shift Left to Make Testing Fast and Reliable, n.d.) [↑](#footnote-ref-1)
2. (Kruchten) [↑](#footnote-ref-2)
3. (JeffreyPalermo.com, n.d.) [↑](#footnote-ref-3)
4. (Beck, 2002) [↑](#footnote-ref-4)
5. (Build & Release Tools from Benjamin Day, n.d.) [↑](#footnote-ref-5)