AZURE PIPELINES

SUCCINCTLY

BY ANTONIO LICCARDI

SUCCINCTLY EBOOK SERIES

Syncfusion®

Azure Pipelines Succinctly

By
Antonio Liccardi

Foreword by Daniel Jebaraj

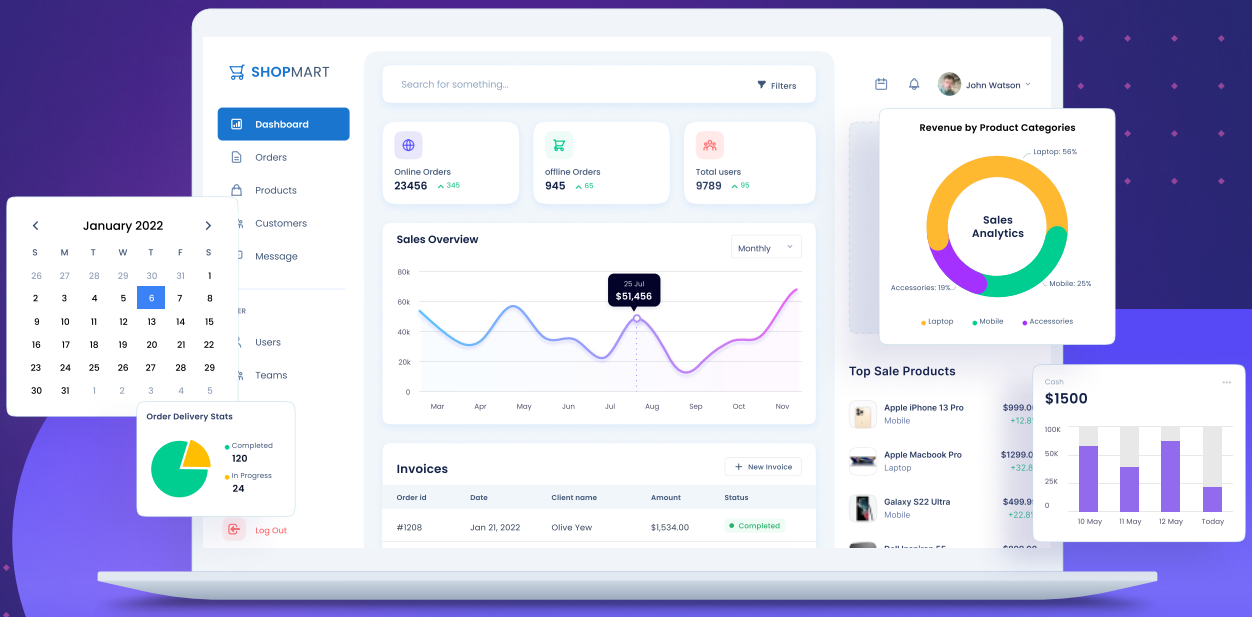
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

ISBN: 978-1-64200-213-3

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Graham High, senior content producer, Syncfusion, Inc.

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING

POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

4.6 out of 5 stars

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Syncfusion

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Hi! My name is Antonio Liccardi, and I work as a DevOps engineer and trainer at Blexin, a software company based in Naples, Italy. I have always been inclined to share my knowledge with others, which led me and some friends to start DotNetCampania, a .NET community, back in 2009. I'm a frequent speaker at Italian conferences, and I'm a member of the getlatestversion community. Recently, I started a YouTube channel and Twitch streaming (just in Italian for now). When I have free time, I like writing technical articles on the company blog or my blog. Since the beginning of my career, I have always been fascinated by software application lifecycle management. This book is an example of the passion that helped me to gain the knowledge I have today.

Let's connect!

I'm a great fan of continuous improvement—if you have any questions or feedback, please feel free to reach out me:

- Email: antonio.liccardi@blexin.com
- Twitter: https://twitter.com/turibbio
- LinkedIn: https://www.linkedin.com/in/antonio-liccardi

Acknowledgments

Writing a book is not an easy task: it requires lots of effort, especially during your spare time, which means that the people you love have to balance their lives according to you. I want to thank my wife, Marilena, for supporting me during the writing of this book.

I want to thank Michele Aponte: you are a great mentor and a friend to me. Make sure to check out his book published by Syncfusion: *Using .NET Core, Docker, and Kubernetes Succinctly*.

If you are thinking about writing a book, speaking at a conference, or even improving your speaking skills for a meeting, you need to read Lorenzo Barbieri's ebook *Public Speaking for Geeks Succinctly*. It helped me a lot.

Finally, I want to give a shout-out to the team at Syncfusion, particularly to Graham and Tres—thanks for the opportunity and your patience.

Introduction

The first time I worked on company software back in 2007, I found that sharing and maintaining code with my colleagues was very difficult. At the time, SourceSafe was well-known software that helped to achieve this task, but its configuration and daily use weren't always easy.

After some months, at the suggestion of my manager, I went to my first conference where I followed a few sessions. One session was about software named “Visual Studio Team System 2008.” I was speechless—that was the tool I needed to work better!

When I started to use it, I quickly realized that the software not only helped with versioning code, but also helped manage the entire lifecycle of a product. One component, in particular, was fascinating: the Team Build, which served as a workflow to build and create a deployment package for applications automatically.

Today, after some renaming due to marketing reasons, Team Build has become Azure Pipelines. In this book, I'm going to show you what it is capable of.

I hope you enjoy it.

Chapter 1 What is Azure DevOps?

Azure DevOps, formerly known also as Visual Studio Team Services, is the name of a set of services that help manage the entire lifecycle of an application. The product is composed of five services: Azure Boards, Azure Repos, Azure Pipelines, Azure Artifacts, and Azure Test Plans. Azure DevOps is available both online and on-premises, and the latter is named **Azure DevOps Server**. In the following sections, you will first sign up for an account, and then receive an overview of each service (excluding Azure Pipelines, which is extensively discussed in this book).

Sign up for Azure DevOps

The first step is to sign up in the Azure DevOps portal using a Microsoft or GitHub account. Open the Azure DevOps start page and select **Start free** or **Start free with GitHub** (Figure 1.1).

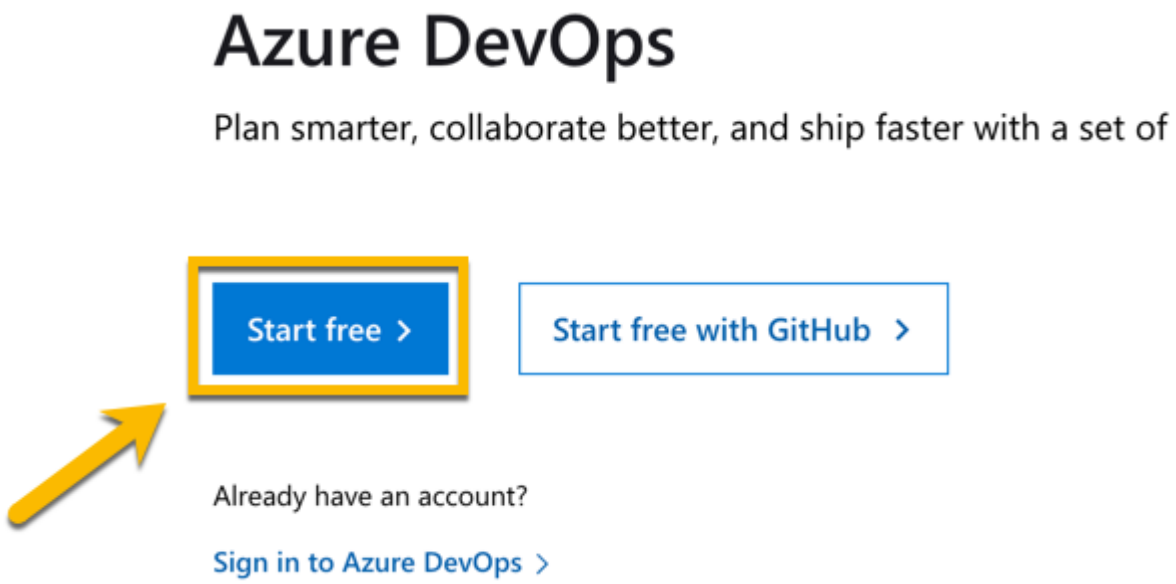


Figure 1.1: Azure DevOps start page

If needed, you can create a Microsoft account on this page for free (Figure 1.2).

Figure 1.2: Create a new Microsoft account

After logging in, you will be asked to select a country (Figure 1.3) for the new organization using the account name. An organization is simply a container for our projects.

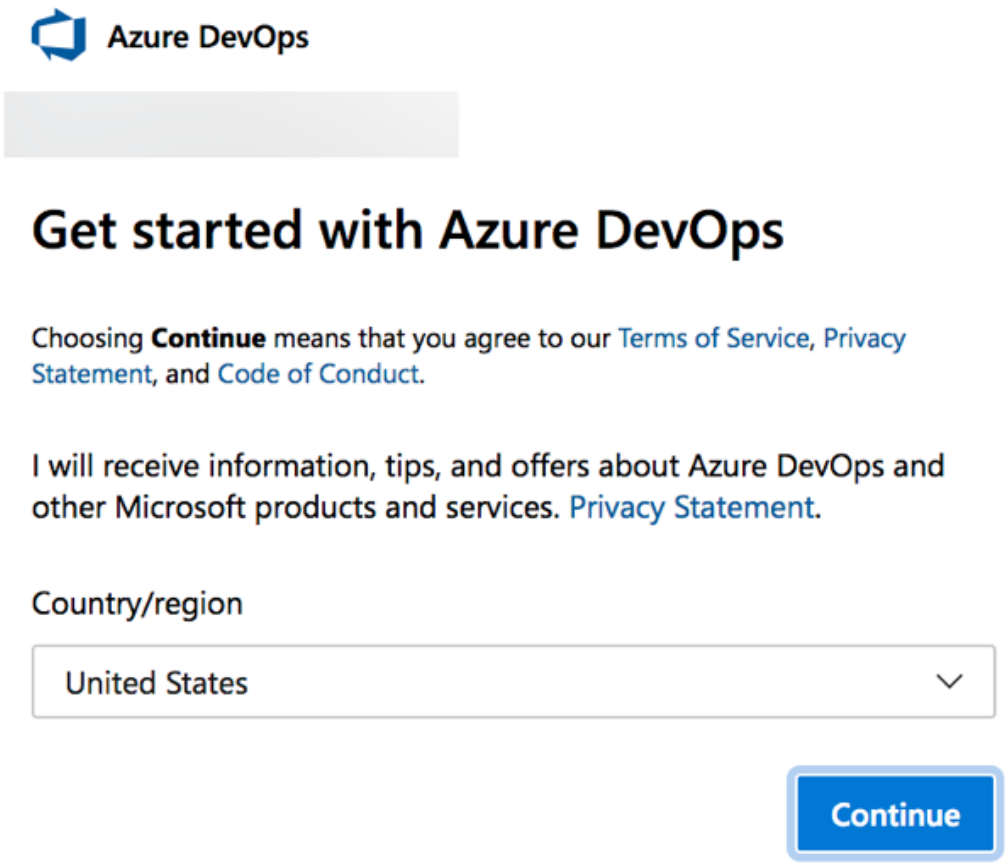


Figure 1.3: Country selection for an organization

When the organization is created, the page shown in Figure 1.4 appears. Here you can make the first project, give it a name, and choose whether it is public or private.

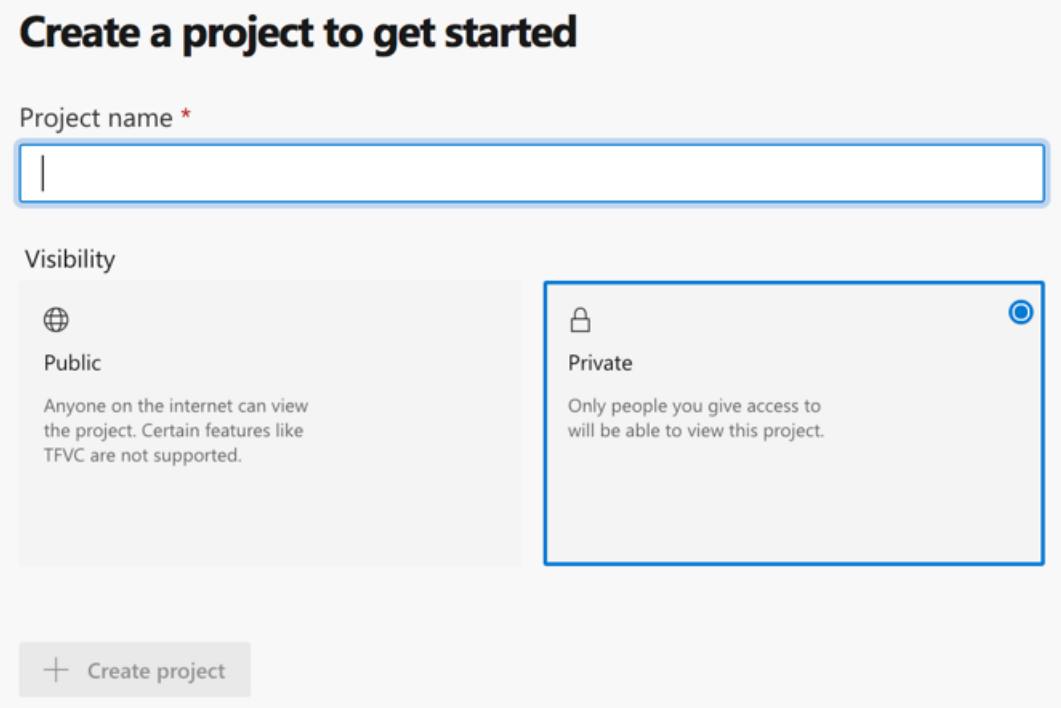


Figure 1.4: Create a new project

The project is located at **https://dev.azure.com/*{organization_name}*/*{project_name}***, (Figure 1.5) and has **Git** as the version control system and **Basic** as the process template (discussed later).

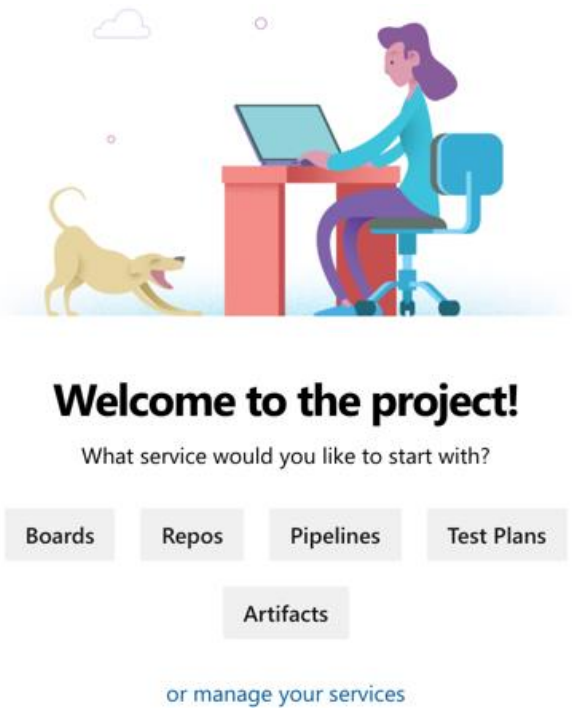


Figure 1.5: Landing page of a new project

There are several different settings when creating a new project:

- **Name**: The name of the project.
- **Description**: A description of the project.
- **Project Visibility**: The visibility of the project; can be public or private.
- **Process Template**: Helps organize the work. There are four process templates available: Basic, Agile, Scrum, and Capability Maturity Model Integration (CMMI).
- **Version Control System**: Supports both Git (unlimited repositories per project) and Team Foundation Version System (one repository per project).

Note: The Basic process is the most straightforward process available in Azure DevOps. It is used to track Epics, Issues, and Tasks.

Azure DevOps services

Let's take a look at the Azure DevOps services.

Azure Boards

Azure Boards allow us to organize and track the work related to a software project. Its main component is the **work item**, which is just an object that collects all the information regarding a new feature or a requirement for an application. The Work Items view lists all the work items in the project (Figure 1.6).

ID	Title	Assigned To
114	"Verify succesful login. Pre-requisite: Valid email and password"	Antonio Liccardi
73	A new test case	Antonio Liccardi
21	Ad-hoc promotions	Unassigned
42	Add categories	Unassigned
50	Add Terms and Conditions for Sale & Support in Layout.cshtml Page	Unassigned
53	Add wishlist page	Unassigned
38	As an admin, I should be able to update prices on ad-hoc condition	Unassigned
102	As a consumer, I want to be able to add a coupon code to my purc...	Antonio Liccardi
98	As a customer I would be able to browse the site	Antonio Liccardi
72	As a customer, I should be able to browse the website	Antonio Liccardi
100	As a customer, I should be able to log in	Antonio Liccardi
101	As a customer, I should be able to log in	Antonio Liccardi
25	As a customer, I should be able to put items to shopping cart	Unassigned
34	As a customer, I should be able to select different shipping option	Unassigned
27	As a customer, I would like to have a sort capability by price and ...	Unassigned

Figure 1.6: Work Items view

There are many types of work items, and they are related to the process selected in the project creation. What's interesting about work items is that they can track progress using a state model. In this way, each part of the team is updated about completed work or problems.

As seen in the previous section, there are four process templates in Azure Boards: Basic, Agile, Scrum, and CMMI. Figure 1.7 shows which types of work items are included with each process and their relationships.

Tip: You can find more information about process templates here.

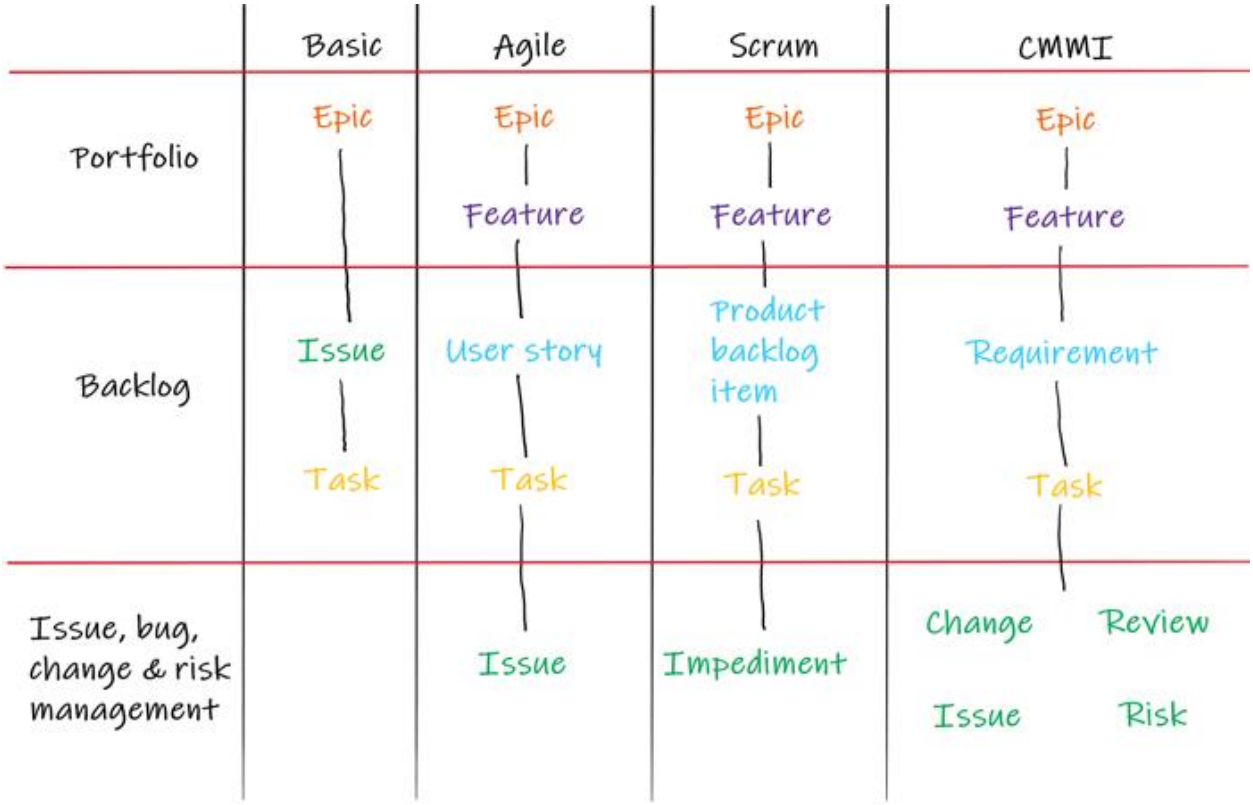


Figure 1.7: Types of work items

Note: In Agile, Scrum, and CMMI processes, it is possible to configure whether bugs are treated as a requirement or a task.

Work items in each process have their state workflow. When you double-click on a single work item, a window opens showing all the item's details, such as description, assignee, area, iteration, effort, full history, and even deployment details. In Figure 1.8, the work item detail page is related to a **user story** from the Scrum process template.

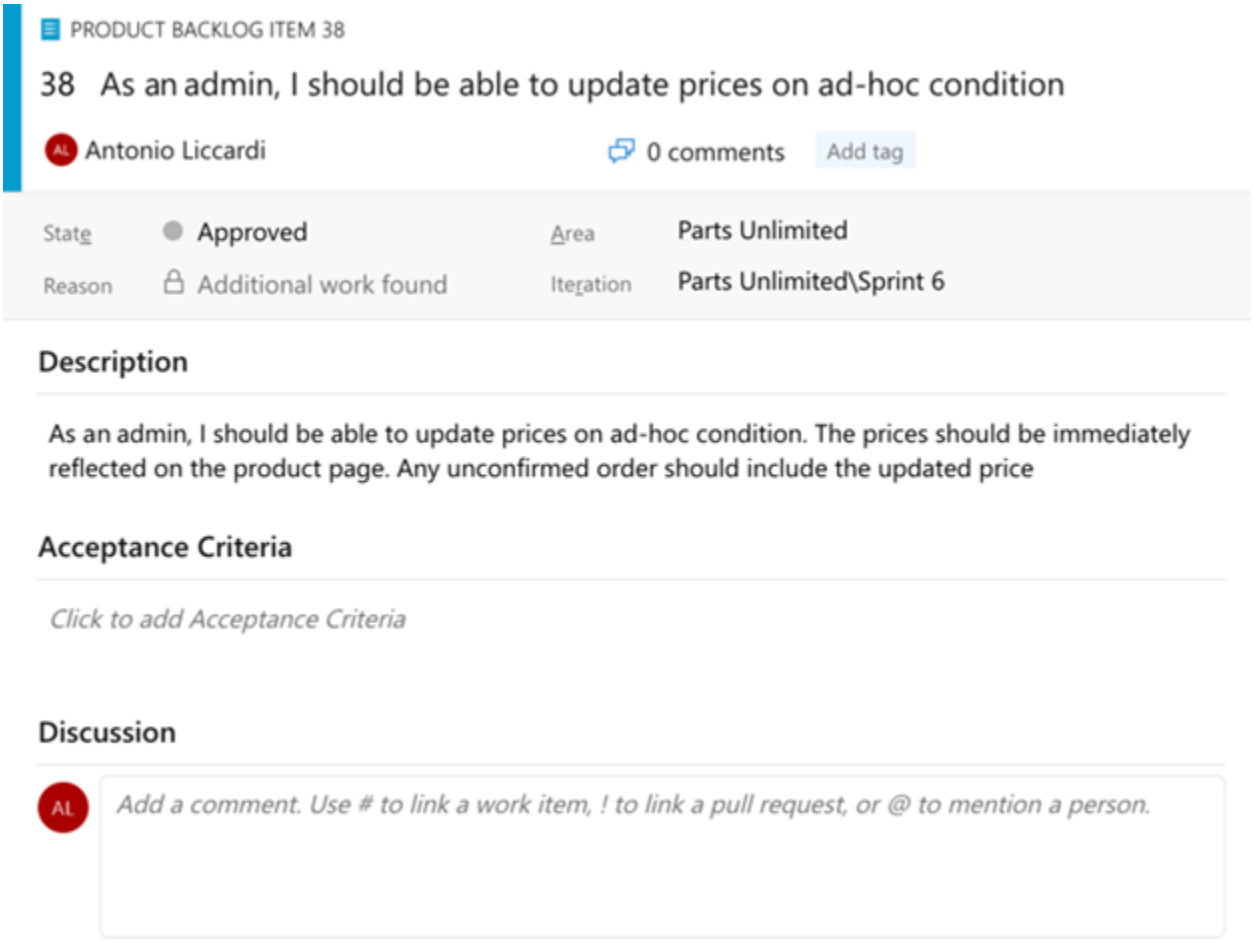


Figure 1.8: Work item page

Each process template is highly customizable: once inherited, work items can be customized by adding new fields or changing the state model. Many extensions are also available from the Marketplace for stepping up the customization level (Figure 1.9).

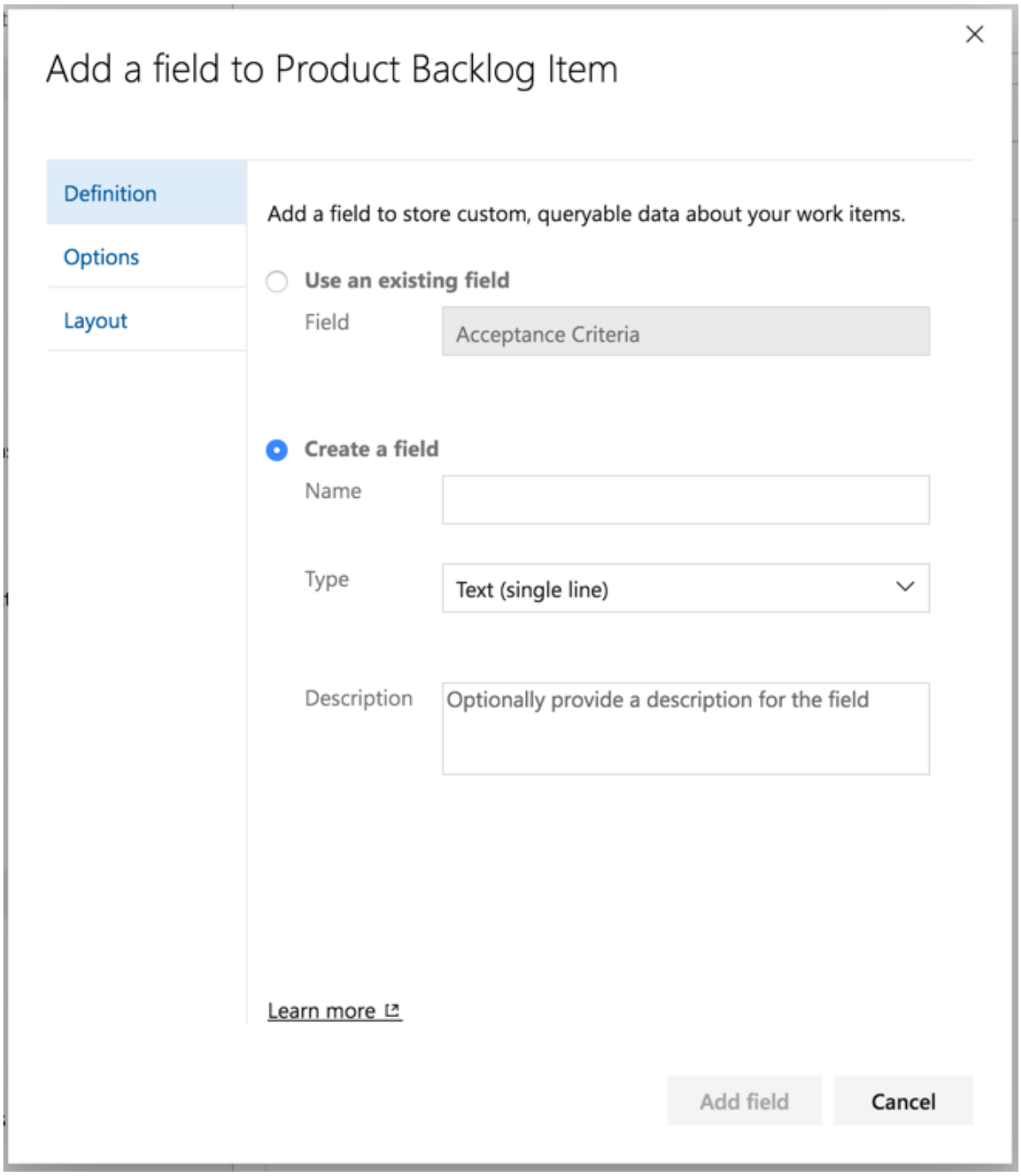


Figure 1.9: Process template customization

The first steps for organizing work are creating **areas** and **iterations**. Areas let us define area paths (including hierarchies) and assign teams involved during the development; iterations help to schedule work into time-framed containers (Figure 1.10). Once created, progress can be tracked using the Boards, Backlog, and Sprint sections.

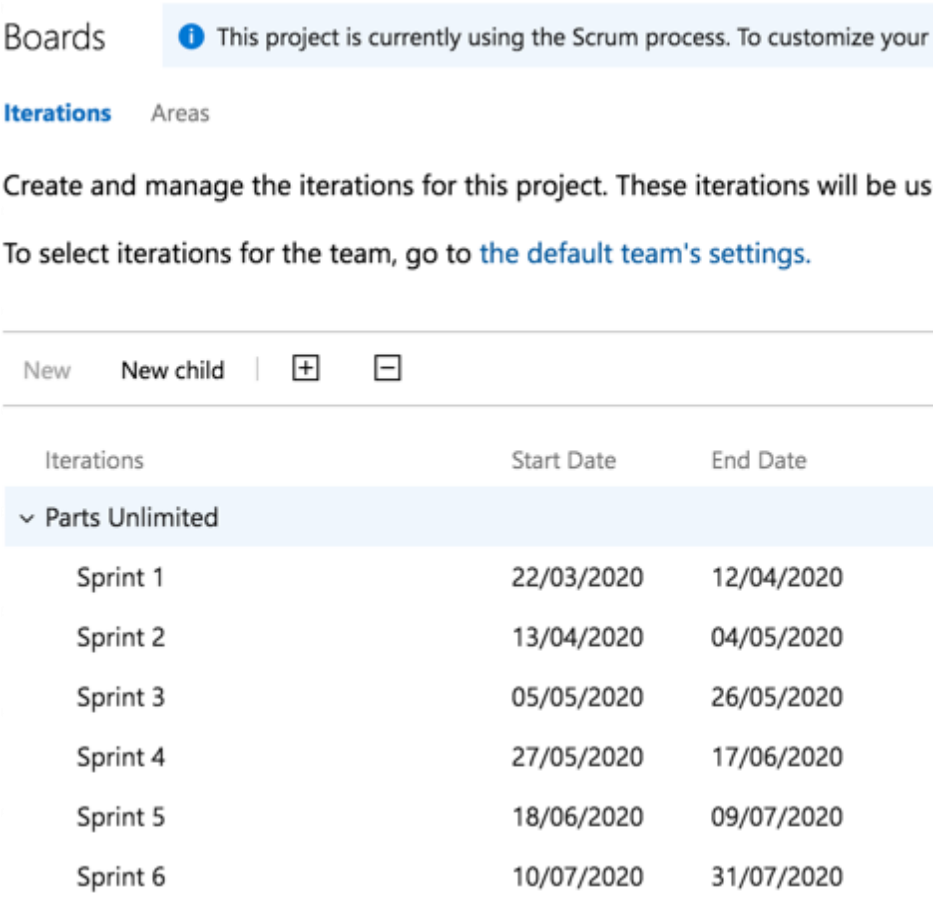


Figure 1.10: Areas and iterations

As the name implies, Azure Boards include a few types of boards. The main one is a *kanban* board (defined for each team), which makes it easy to get an overview of work in progress from different levels (portfolio management). Swimlanes are also available to help expedite work.

The customization level here is very high. It is possible to customize columns and cards, including fields and styles, and limit the work in progress (Figure 1.11).

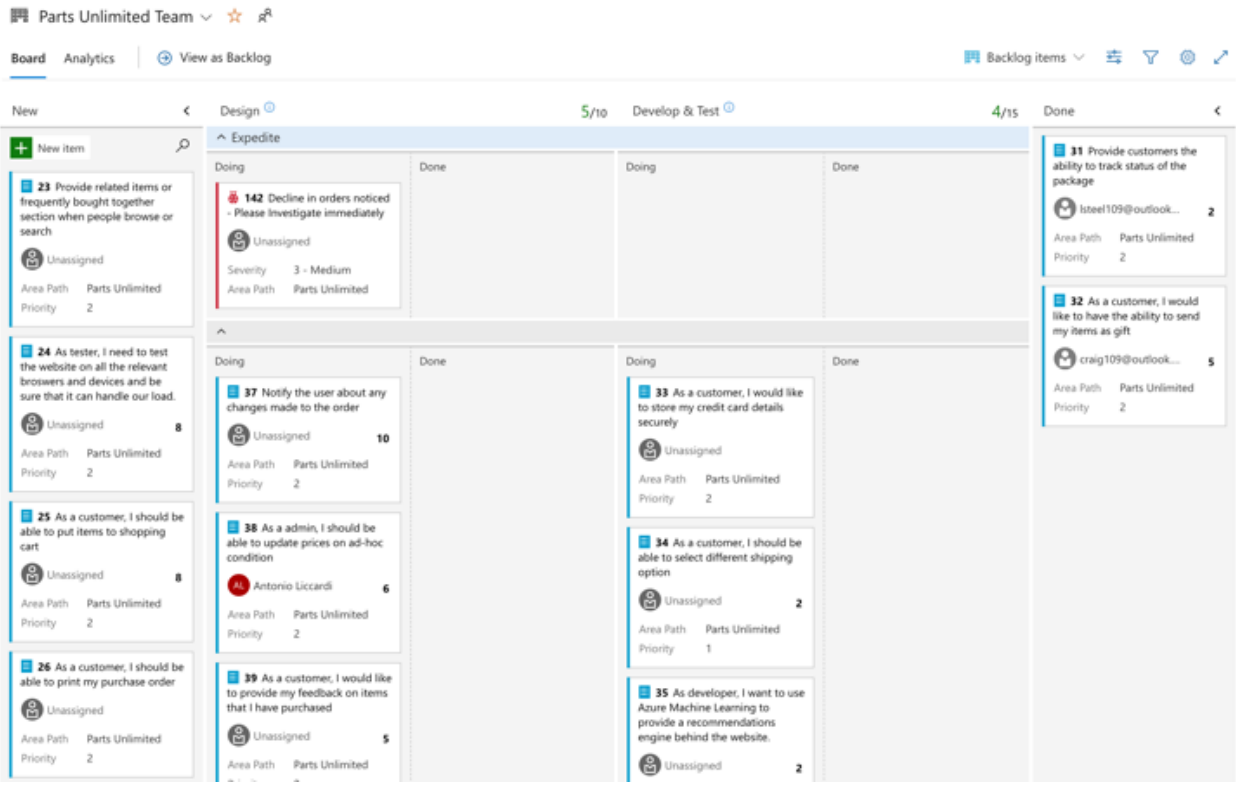


Figure 1.11: The kanban board

The Backlog section contains one or more backlogs, depending on how many teams are available. In a backlog, a user can manage relationships between work items, reorder them using drag and drop, or add them to a specific sprint (Figure 1.12).

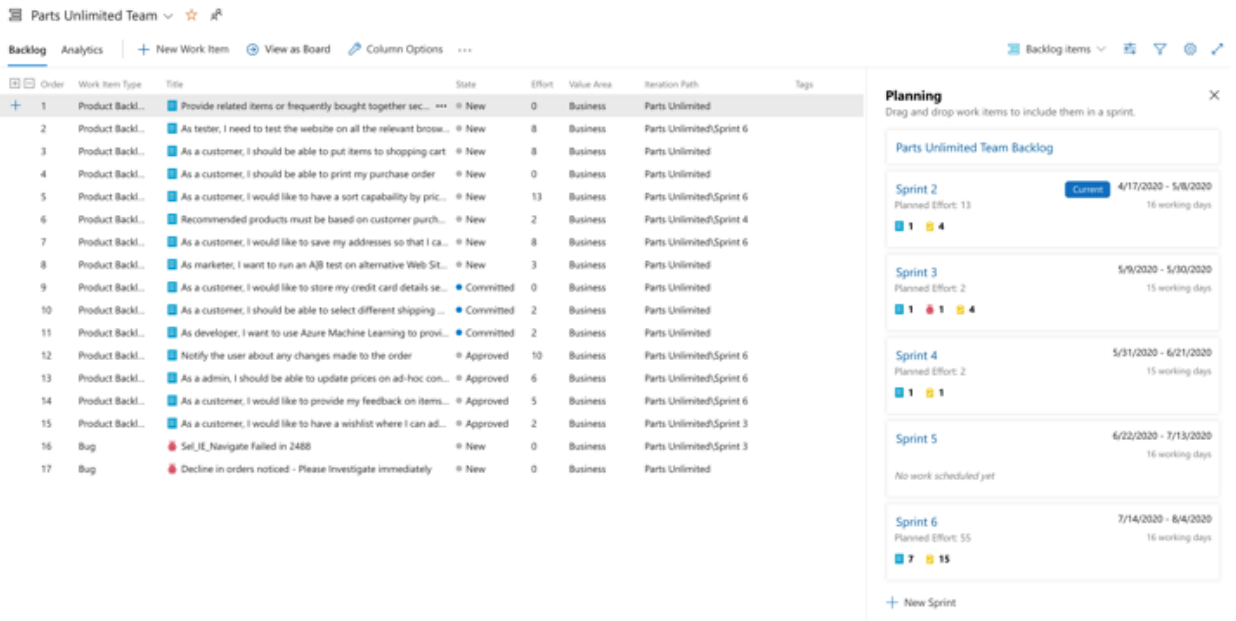


Figure 1.12: The Backlog section

The Sprint section gives us more detail about the work in progress in the current sprint, and has a Task board, a Burndown chart, an Analytics view, and a Capacity planning view. All the filters come in handy for adjusting what is shown (Figure 1.13).

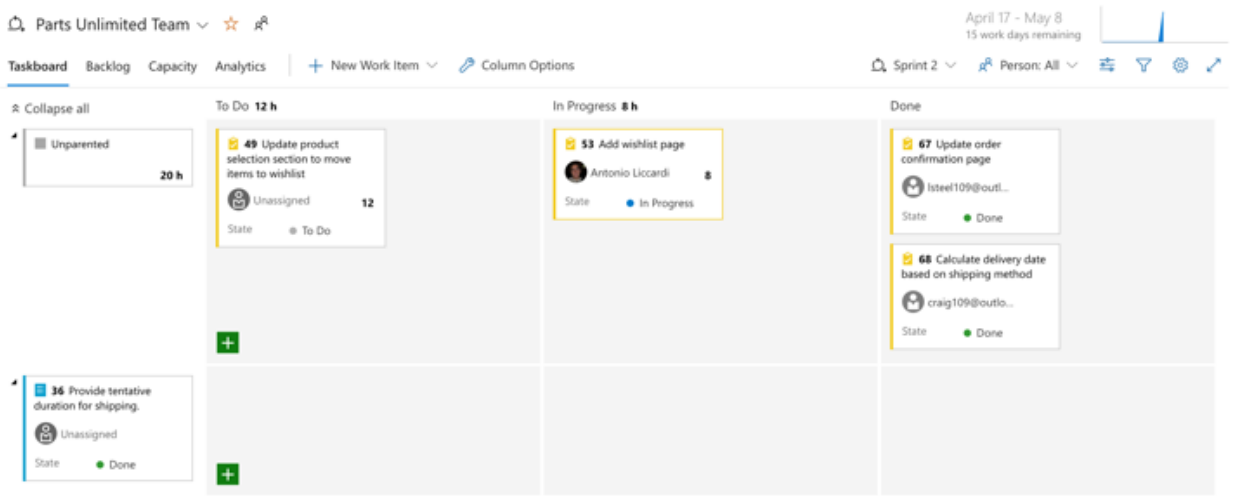


Figure 1.13: The Sprint section

Finally, Azure Boards has a powerful **query engine** that allows extracting data to track work and monitor KPIs (Figure 1.14). Results can be converted into **graphs** and pinned to **dashboards** in the Overview section (Figure 1.15).

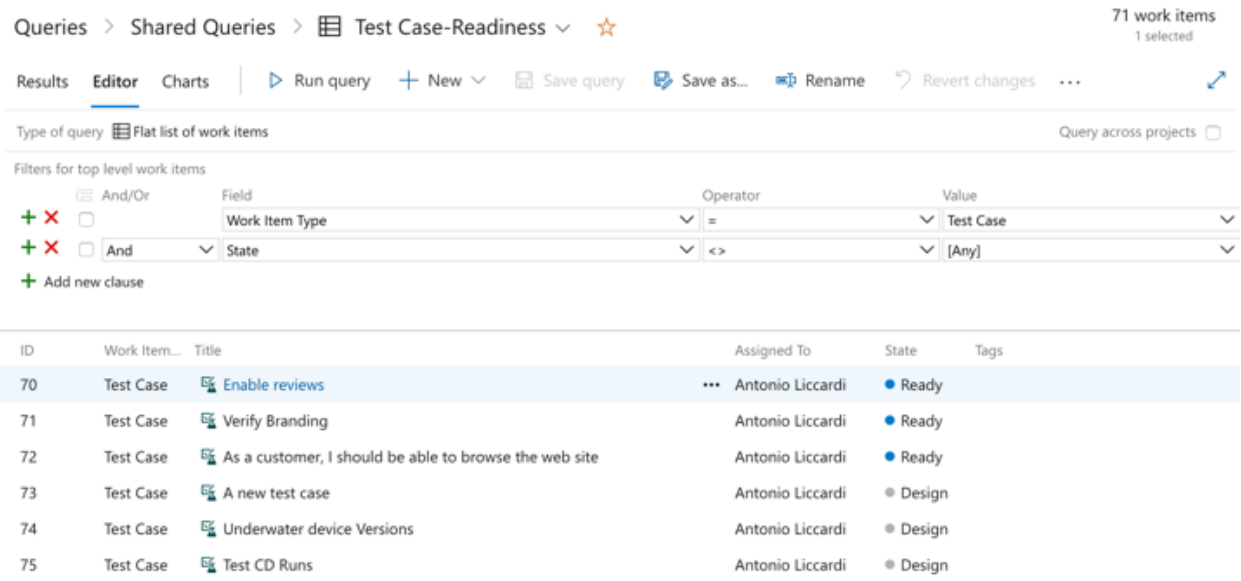


Figure 1.14: The query editor view

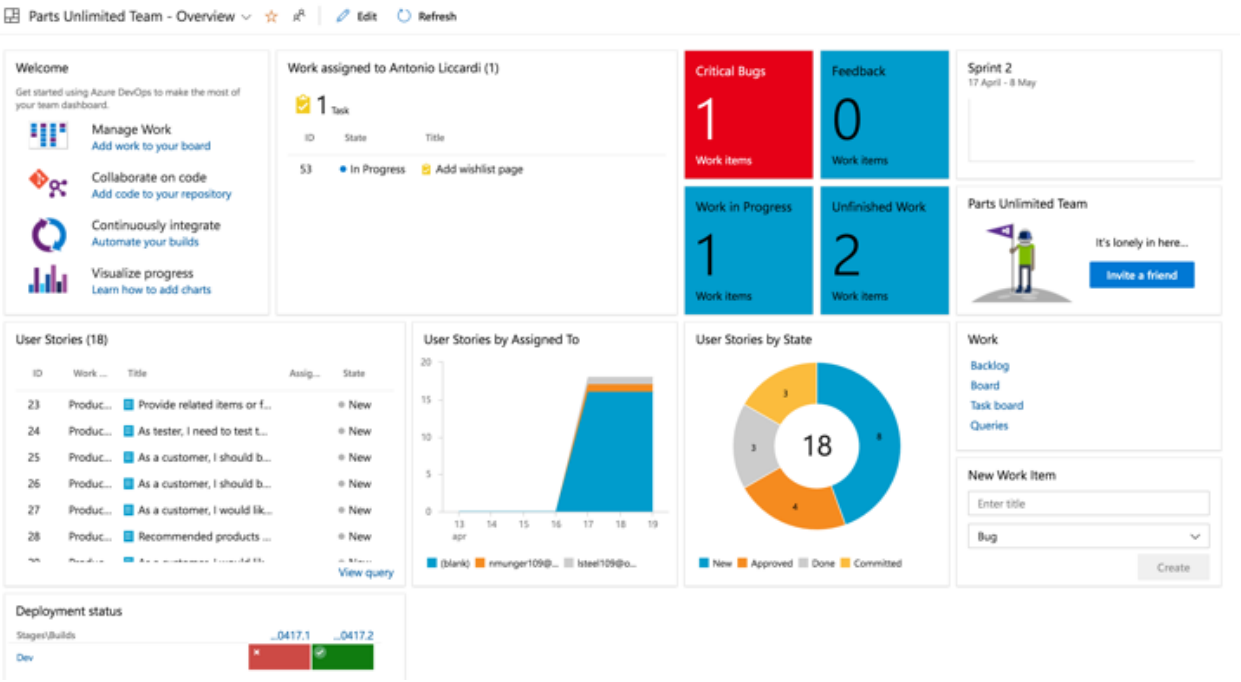


Figure 1.15: A dashboard example

Azure Repos

Azure Repos is another component of Azure DevOps. It makes version control systems such as **Git** (default) or **Team Foundation Version Control** (TFVC) available to track changes in your code.

Git is an open-source distributed version control system created by Linus Torvalds in 2005 to maintain changes to the Linux kernel, and its popularity and usage have increased over the past 10 years. Azure Repos provide unlimited Git repositories and implement a standard Git version. The user interface lets us fully manage the repository showing source code, full history, blame (annotation), and tags, and users can even create new files or folders. On the other hand, TFVC is a centralized version control system available in Azure DevOps from earlier versions, and Azure Repos allows one TFVC repository per project. Migration from TFVC to Git is also possible using the **import** feature from the user interface.

When using Git, Azure Repos provides a **pull request** (PR) mechanism to review code created by developers before the merge process (Figure 1.16). The user experience of the workflow is astonishing—reviewers can comment on code changes to give feedback to the developers, and can require mandatory work before the merge. Developers can respond to comments and mark the feedback as "resolved." Thanks to branch policies, a PR can trigger a build to compile and test the source code before the review, and in case of failure, a reviewer can reject it. Some actions can be automated (such as the autocomplete feature), and integrations with external services are available.

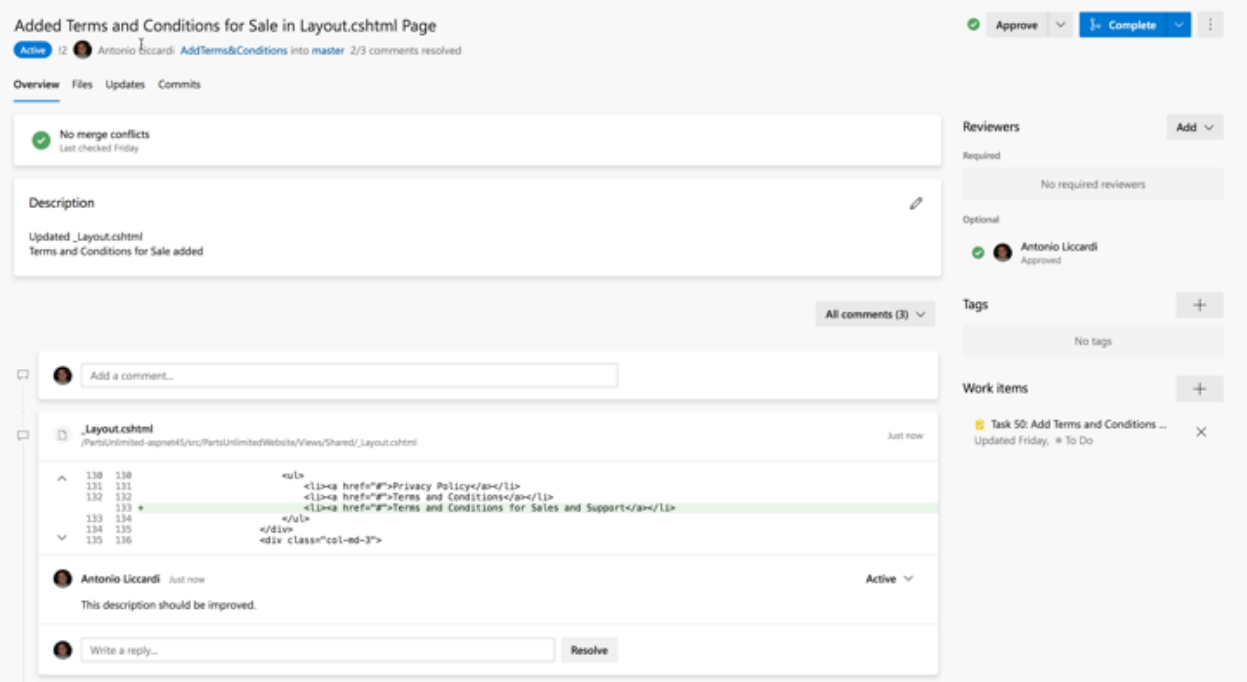


Figure 1.16: The pull request view

Azure Test Plans

In Azure Boards, test cases are a work item type that track testing done on software. They are created and managed on the board, and developers can specify precisely the steps and expected results that tests require (Figure 1.17).

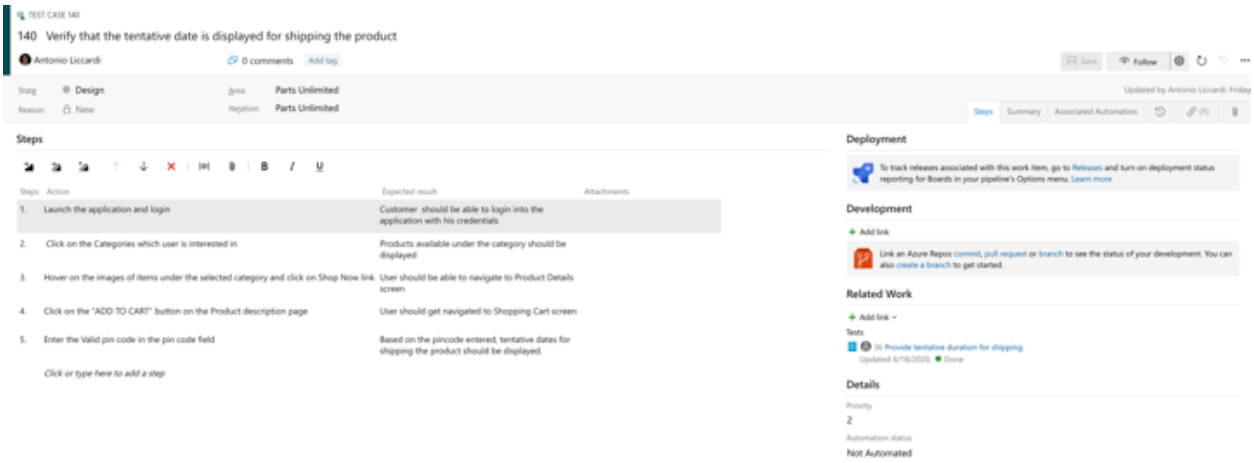


Figure 1.17: A test case

Sometimes, testing needs more insight into how developers should execute it, explaining dependencies, environments, and configuration. In this case, a service like **Azure Test Plans** is what you should use. Indeed, this component lets us plan and track different kinds of tests, such as **manual** or **user acceptance**. Also, thanks to the **Test & Feedback** extension, developers can practice **exploratory** testing, and stakeholders can share their feedback with us in a natural way (downloadable here).

Test plans are related to area paths and iterations, and they organize test suites using three different approaches (Figure 1.18):

- **Static suite**: Grouping test cases added manually.
- **Requirement based suite**: Grouping test cases related to work items such as Product Backlog items.
- **Query-based**: Grouping test cases starting from a search.

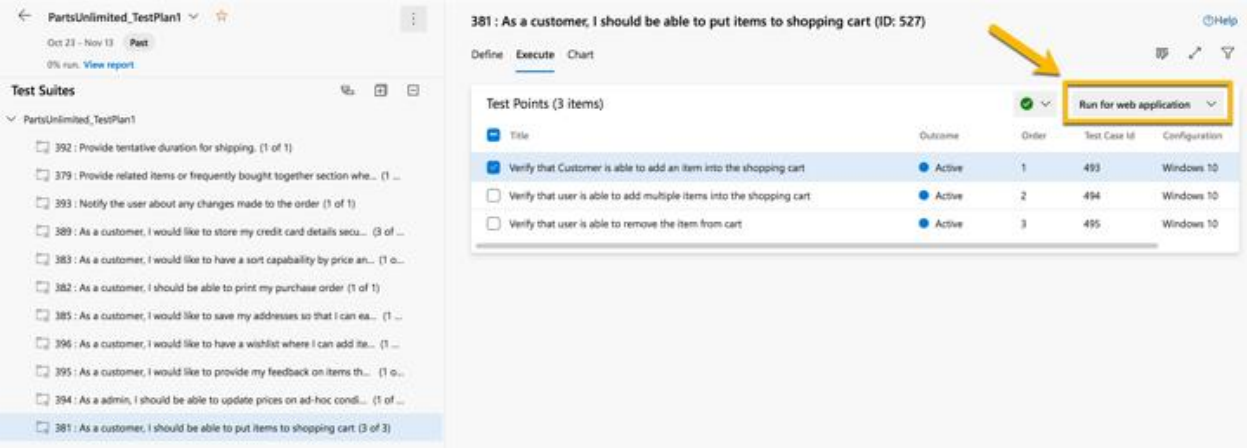


Figure 1.18: A test plan

A test case summarizes what to test. The test case window contains a section that lists all the mandatory steps to execute the test. Also, each test is related to a specific running configuration, which is customizable depending on parameters such as the system configuration (such as Windows 10 or Android devices).

You can run the test using the **Run for web application** button, which opens the Test Runner: this will recap all the steps to execute, and you can flag it as passed or failed. In case of failure, the runner lets us type comments, create a new bug (which includes information regarding the testing environment), take screenshots, or even take videos (Figure 1.19).

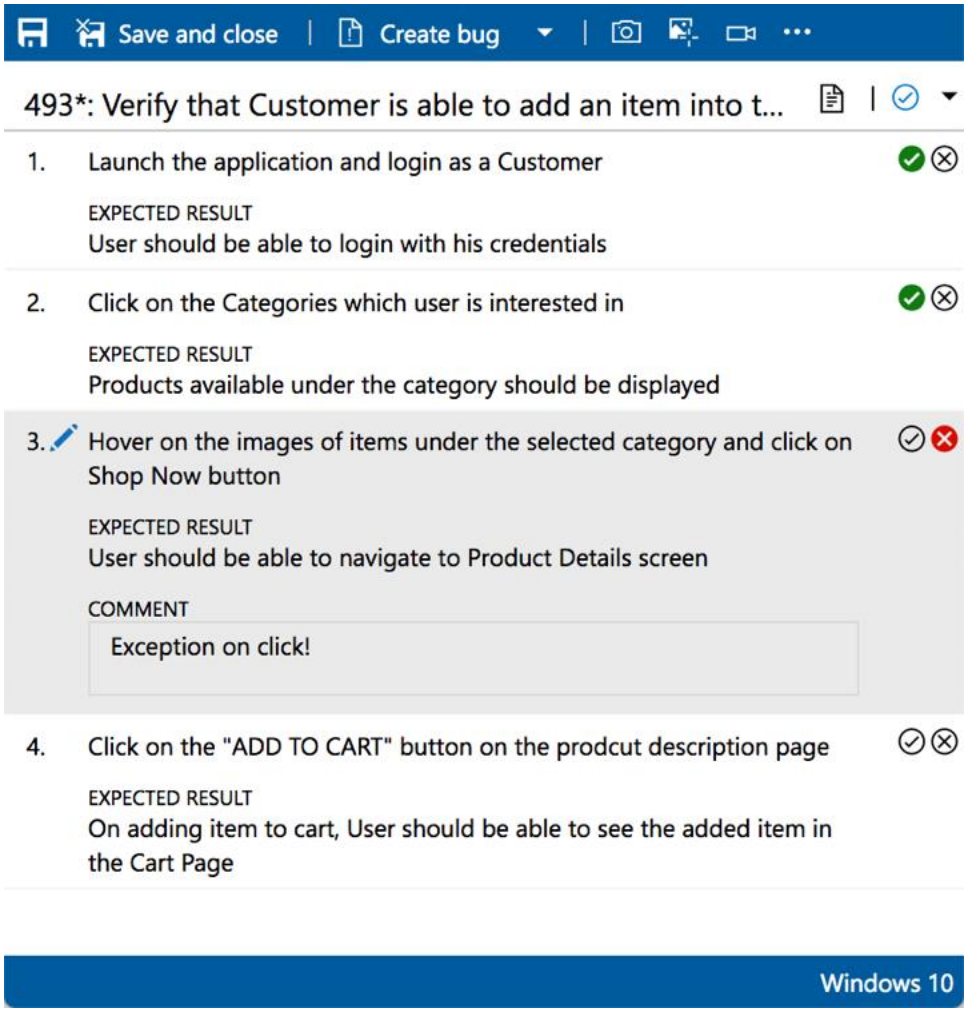


Figure 1.19: The test runner

The Runs section on the side menu recaps all the test sessions, and the Progress report section gives insights about runs (Figure 1.20). With such instrumentation, testing is an enjoyable experience!

Figure 1.20: The progress report

Azure Artifacts

Azure Artifacts is a service that helps create and maintain **NuGet**, **npm**, **Maven**, **Python**, and **universal packages**.

You can start using Azure Artifacts by creating a new feed that is scoped for an organization or a project (Figure 1.21). Then you can upload a package directly from a CI/CD pipeline. Administrators can set policies about who can read or upload packages. After a package is uploaded, developers can download it directly into their preferred IDE, such as Visual Studio, or by using a command line (Figure 1.22).

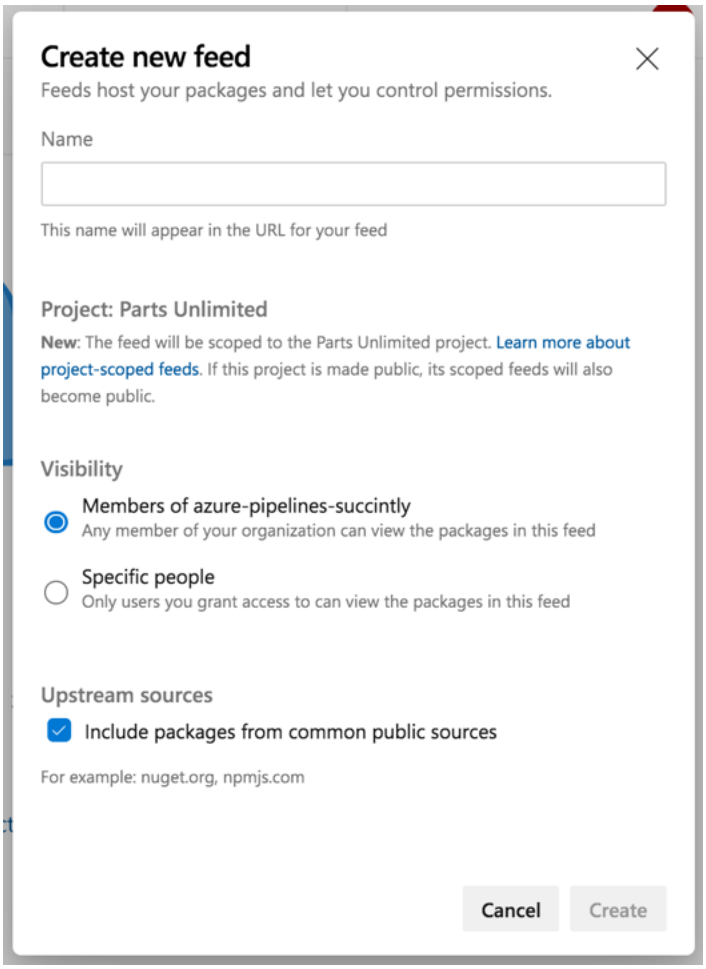


Figure 1.21: Create a new feed

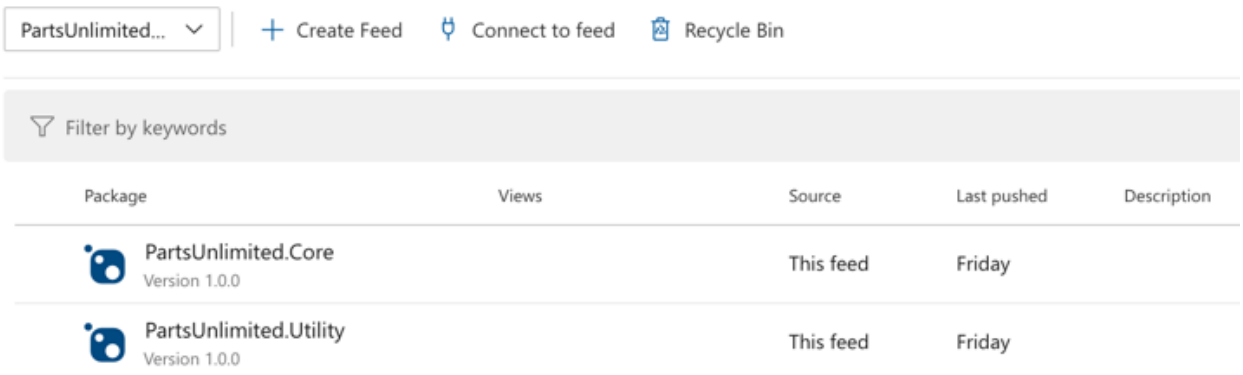


Figure 1.22: A NuGet feed with packages

Universal packages are a particular type of artifact that collects files into a single package that falls outside the scope of the other supported types.

Also, Azure Artifacts supports upstream sources, which combine our feeds with remote feeds such as nuget.org.

Azure DevOps Server

Azure DevOps Server, previously known as Team Foundation Server (TFS), is a platform that provides the same services as Azure DevOps Services, with some slight differences.

First, you have to set up an entire infrastructure to install the server, including a back end based on SQL Server. Scaling is different too: Azure DevOps Services uses organizations and projects to scale, rather than Azure DevOps Server, where scaling means adding more server instances (including their management). Other differences are related to user management and reporting.

The main reason that companies use an on-premises version is to keep their data in the same network as their infrastructure. A detailed list of all differences is available here.

Pricing

Azure DevOps uses a per-user pricing model, and it differs between the cloud and on-premises editions. It offers two plans: **Basic** and **Basic + Test Plans**, which differ for the integration with Azure Test Plans.

Both plans provide Azure Pipelines for **free** with the following limitations:

- One free Microsoft-hosted CI/CD (1,800 minutes free with one free parallel job).
- One free self-hosted CI/CD (one free parallel job with unlimited minutes).

You can purchase additional parallel jobs by paying a monthly fee, depending on how many parallel jobs you need. Regarding Azure DevOps Server, Microsoft provides a free version for individuals or small teams named **Azure DevOps Server Express**.

You can find additional information about Azure DevOps pricing here.

Preview features

Both Azure DevOps Services and Azure DevOps Server receive regular updates, even if the former receives them in advance of the latter because of its nature as a SaaS platform. Indeed, Microsoft usually updates the platform behind the scenes and gives us the ability to enable some **preview features**, supporting beta testing. Preview features can be enabled at the organization level or user level through the **User Settings** button close to the **Account Manager** button (Figure 1.23).

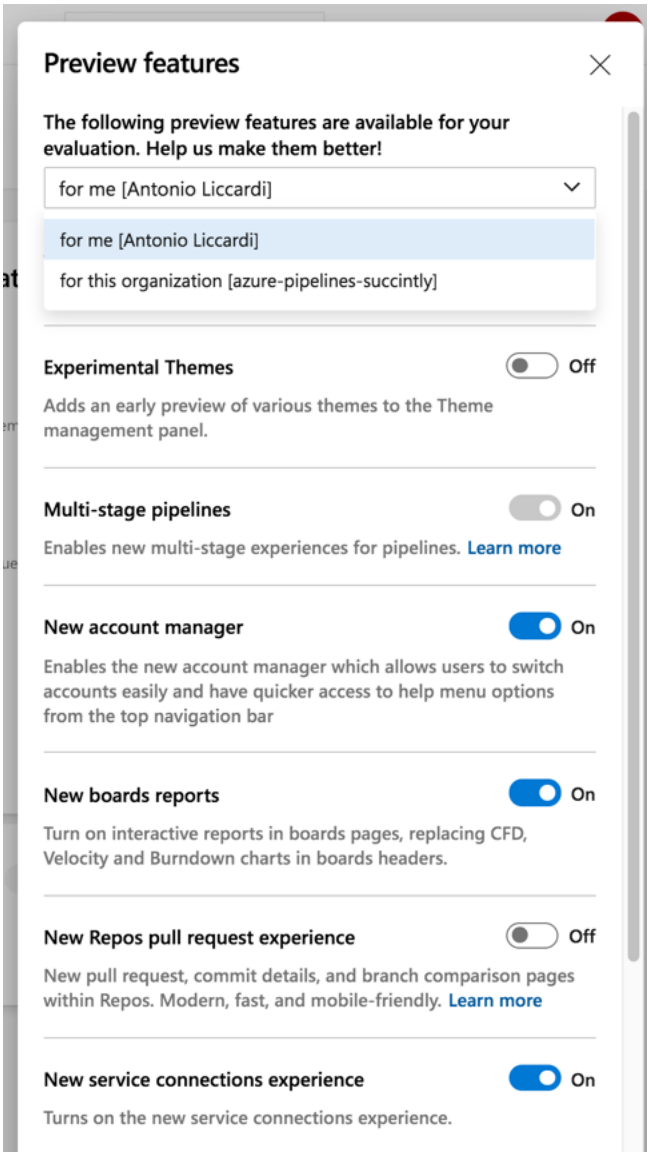


Figure 1.23: Enable preview features

Note: The preview features shown in this book could be released by the time of the publication.

Azure DevOps CLI

Like many tools or frameworks, Azure DevOps also has its own command-line interface (CLI), and it is based on the Azure CLI. With this tool, you can execute commands or tasks directly from a console window, in a quicker way compared to the user interface. A bonus is the automation of repetitive tasks. The installation instructions are available here, and they include links to the documentation and samples.

Azure DevOps documentation

Microsoft makes available excellent documentation for Azure DevOps, including:

- Full documentation.
- Learning path for certifications.
- Azure DevOps Labs, step-by-step tutorials on how to use features in Azure DevOps.

Chapter 2 Your First Pipeline

This chapter introduces the main topic of this book: Azure Pipelines. As you can imagine, this product is a key component of Azure DevOps and helps you build, test, and deploy your software. The basic concept is that a change in a code repository triggers a pipeline, which compiles and tests the software. This phase can generate one or more artifacts, which are the primary input for a release process comprised of multiple stages.

Azure Pipelines supports many programming languages and build platforms (such as Windows, Linux, and macOS). Also, release targets can vary depending on the scenario and include cloud providers or on-premises servers. Pipelines are defined using two approaches, user interface (classic) and YAML syntax, and this chapter introduces both.

Create a sample project

This section helps you create a .NET Core sample project, push it to a git repository hosted on Azure Repos, and use it as the input of a build pipeline.

In this scenario, I'm going to create a .NET Core web application because it is a technology that I'm familiar with, but the same concepts can be applied to any framework or programming language. Indeed, building a project on Azure Pipelines requires an agent that supports the platform you are using. The build process is just a collection of sequential steps, and depending on the platform, you may have specific arguments or more actions to include.

Creating an ASP.NET Core application is straightforward, and you can take a look at this page for help achieving the task. It doesn't matter if you use Visual Studio, Visual Studio Code, or even the .NET Core CLI; the aim here is to have a working sample project.

Note: In this sample, the version of .NET Core is 3.1.7.

Push the code to Azure Repos

In the previous chapter, you created a new Azure DevOps project. You can reuse that Git repository in this project to develop a build pipeline. Since the repository on Azure Repos is not initialized yet, you need to follow the instruction in the **Push an existing repository from command line** area, as shown in the following Figure (2.1).

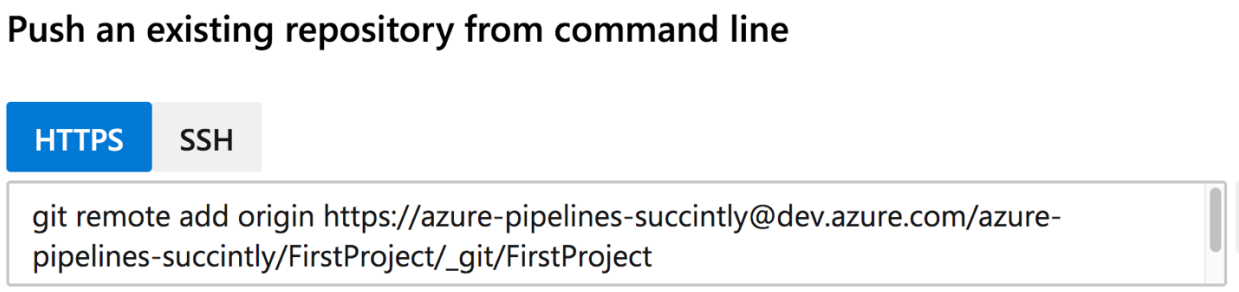


Figure 2.1: Push instruction for an existing repository

First, initialize a local git repository in the project folder, and then link it to the one available in Azure Repos using the following steps:

1. Open a command prompt and point to the local project folder.
2. Initialize a git repository with **`git init`**
3. Add all files to the git stage area using **`git add .`**
4. Commit the code typing **`git commit -m "project created"`**
5. Copy and paste the commands in Figure 2.1 into the terminal, then press **Enter**. This step could require authentication.

Now, the code is pushed into the remote repository (Figure 2.2).

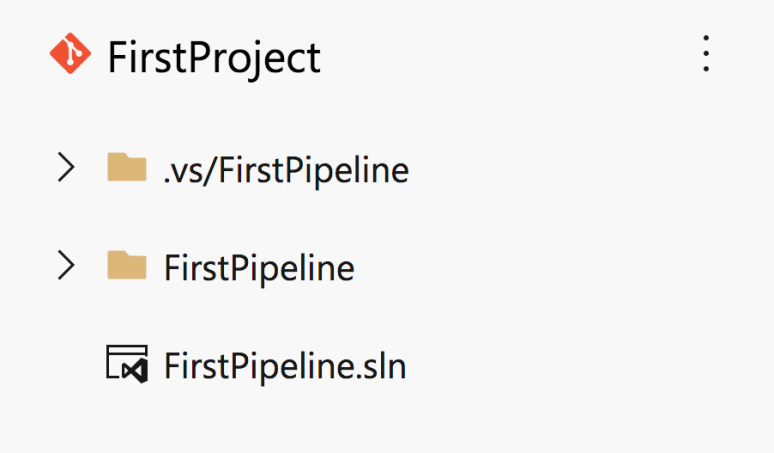


Figure 2.2: Code pushed to the remote repository

Tip: Please read this page if Git is missing in the terminal.

Note: Another possible approach is to first initialize the Git repository on Azure Repos, and clone it into a specific folder on the machine where you then create the .NET Core project.

It's time to create the first pipeline!

First pipeline using classic mode

As I've mentioned, there are two ways to create a pipeline in Azure Pipelines. This section explains how to create a pipeline using the user interface (also known as the **classic** mode).

The main difference between the classic and YAML modes is that with the former, you configure your pipeline entirely from the user interface whereas with the latter, you create a pipeline using YAML markup and save the definition into your repository.

Also, when using the classic mode, once the build and testing phase is completed (**continuous integration**), you should publish the artifacts to use them in a release pipeline. With YAML, you can leverage the multistage pipeline feature, which automatically retrieves the published artifacts in the release pipeline.

Let's start creating a pipeline definition using the following steps: in the side menu of Azure DevOps, select **Pipeline** > **Create Pipeline** > **Use the classic editor to create a pipeline without YAML**. In the page that appears, choose the source code repository storage location type by clicking **Azure Repos Git** and then **Continue** (Figure 2.3).

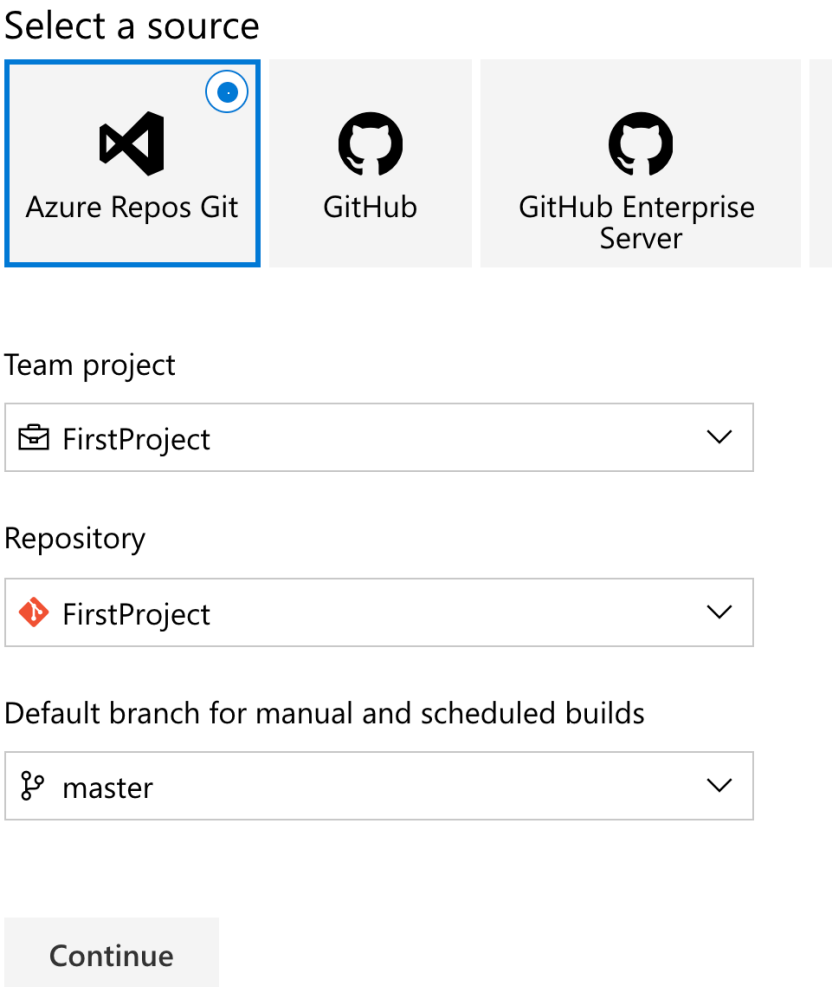


Figure 2.3: Build pipeline source types

In the search box of the window that appears, type **ASP.NET Core** and select the first option (Figure 2.4).

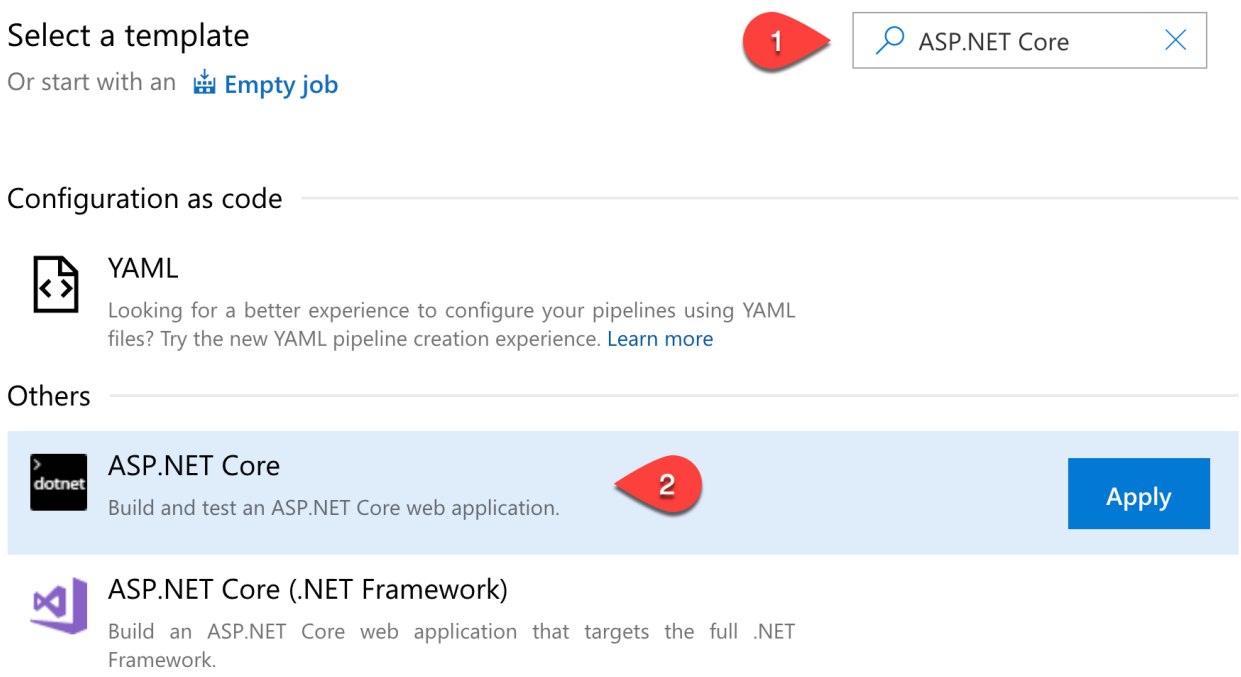


Figure 2.4: ASP.NET Core build template

Azure Pipelines prepares a build definition with all the steps to build and test your application. To follow good practices, let's give the build a comprehensible name, replacing the default name with **FirstProject CI**. Click the **Save & Queue** button to start the execution. The entire process can be observed using the progress page, which gives detailed information about the status and commands that are executed (Figure 2.5). Logs are stored for each build execution and are downloadable for further investigation.

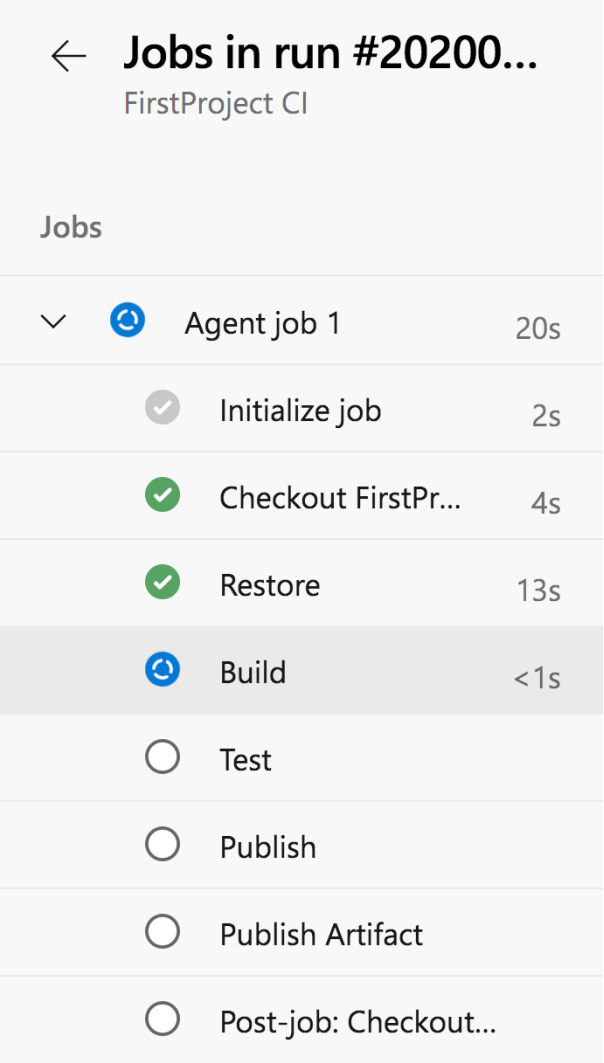


Figure 2.5: The build progress page

After build completion, a summary is available with all the information related to the execution (Figure 2.6).

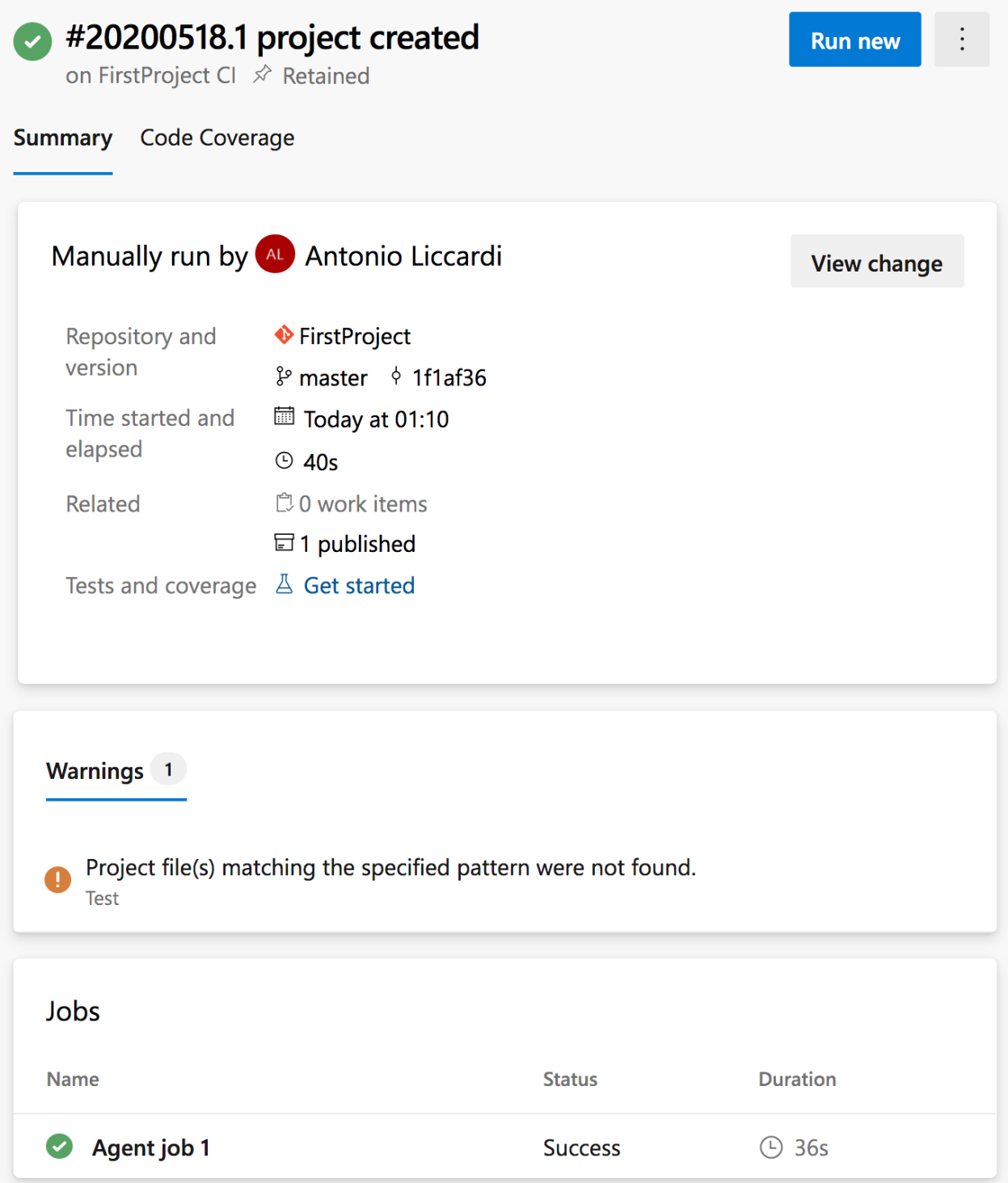


Figure 2.6: The build summary page

In case of success, the summary shows the artifacts published by the build process (in this case, the website publication package). You can browse the artifacts by clicking on them (Figure 2.7).

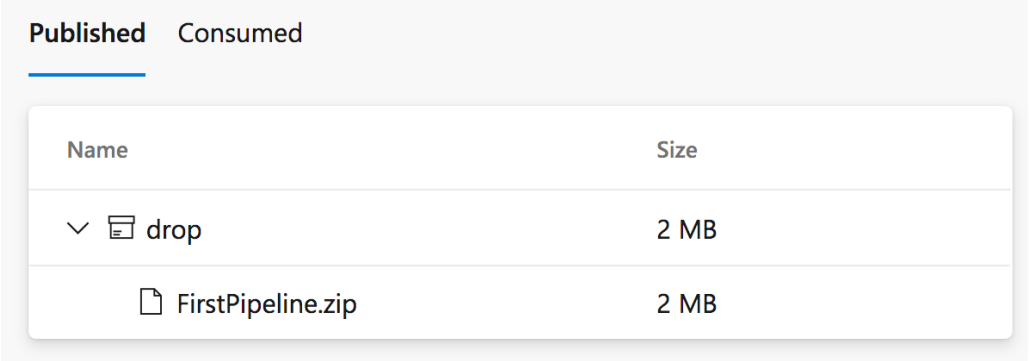


Figure 2.7: The artifacts explorer

A first pipeline using YAML

Using YAML for your build definition has lots of advantages. First, the pipeline definition is written using code, stored in the repository, and has a full version history. The pipeline is stored side by side with your infrastructure definition, which gives you a complete snapshot of your system, and you can restore it, pointing at any time in the past. Also, in complex infrastructure, having a pipeline as code can ease the review and merging phase, especially if multiple engineers create the pipeline. Engineers can submit their PRs, letting other people on the team review and comment in case of problems. Finally, it is possible to copy and reuse part of the definition to complete other pipelines.

If you are not convinced, let's see how it is easy to create a new YAML pipeline. As in the previous section, go to the side menu of Azure DevOps and select **Pipeline** > **New Pipeline**. Next, select **Azure Repos Git (YAML)**.

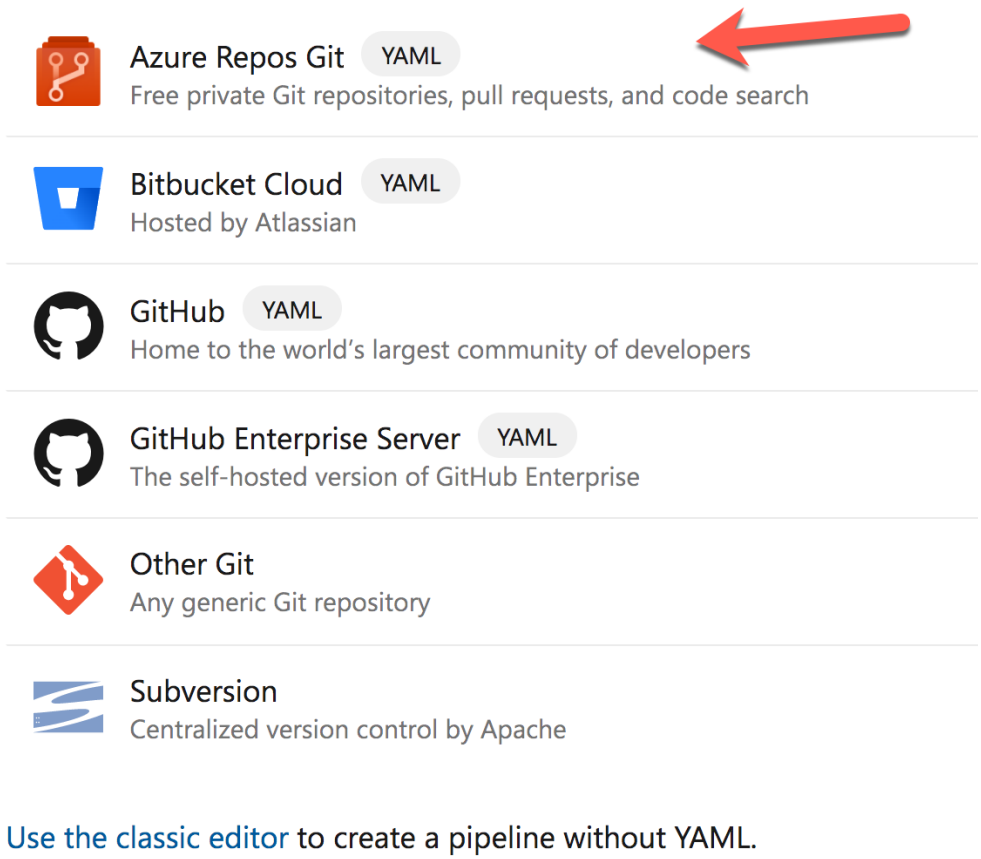


Figure 2.8: YAML pipeline creation

Select the repository and the ASP.NET template. Azure Pipelines creates a *ready-to-run* YAML build definition, which correctly compiles the application. Unlike the classic mode, just one step is missing in this definition, which is the publication of the build artifact. Indeed, to expose the artifacts, you need to modify the pipeline definition, adding a few lines of code.

First, place the cursor at the bottom of the code in the YAML editor. To get help, you can use the assistant on the right side to search for **Publish build artifacts**, and select the first item in the list (Figure 2.9).

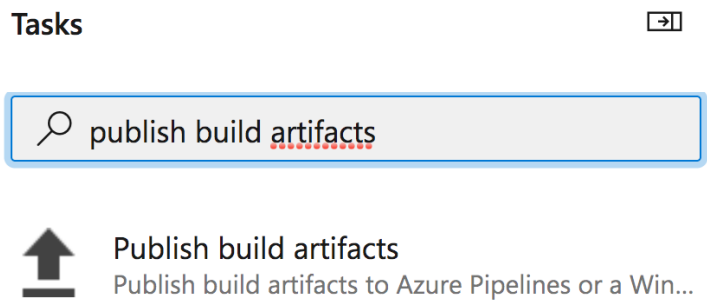


Figure 2.9: Searching tasks with the assistant

These are a few settings related to the task (Figure 2.10); for now, you can leave them with their default values and click **Add**.

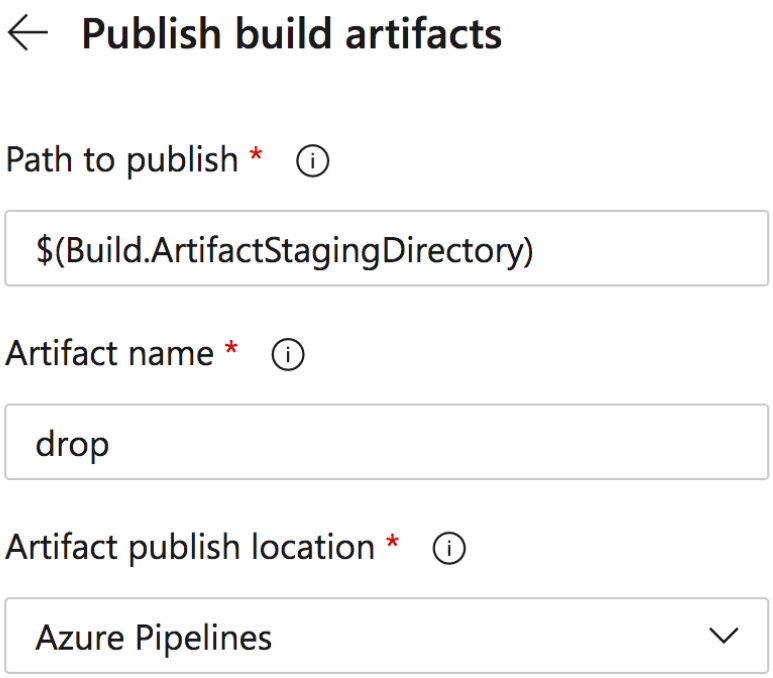


Figure 2.10: Publish build artifacts settings

This template introduces a snippet of code in the pipeline, which publishes the application deployment package. The following listing shows the entire YAML pipeline.

Code Listing 1

```
# ASP.NET
# Build and test ASP.NET projects.
# Add steps that publish symbols, save build artifacts, deploy, and
more:
#https://docs.microsoft.com/azure/devops/pipelines/apps/aspnet/build-
aspnet-4

trigger:
- master

pool:
  vmImage: 'windows-latest'

variables:
  solution: '**/*.sln'
  buildPlatform: 'Any CPU'
  buildConfiguration: 'Release'
```

```
steps:
- task: NuGetToolInstaller@1

- task: NuGetCommand@2
  inputs:
    restoreSolution: '$(solution)'

- task: VSBuild@1
  inputs:
    solution: '$(solution)'
    msbuildArgs: '/p:DeployOnBuild=true /p:WebPublishMethod=Package
/p:PackageAsSingleFile=true /p:SkipInvalidConfigurations=true
/p:PackageLocation="$(build.artifactStagingDirectory)"'
    platform: '$(buildPlatform)'
    configuration: '$(buildConfiguration)'

- task: VSTest@2
  inputs:
    platform: '$(buildPlatform)'
    configuration: '$(buildConfiguration)'

- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: '$(Build.ArtifactStagingDirectory)'
    ArtifactName: 'drop'
    publishLocation: 'Container'
```

Click **Save and Run**, and a popup window will appear. Here you can specify a commit message, a description, a branch to save the definition to, and even whether you want to open a pull request. By default, the pipeline file is named **azure-pipelines.yml**. I usually prefer to keep the definition in a separate branch until the pipeline is consolidated. The pipeline starts and, as in the classic mode, you have detailed logs.

In the code listing, you can see four main parts in the YAML definition:

- **Trigger**: Set the branch that is observed to trigger the pipeline.
- **Pool**: Define the type of agent that executes the build.
- **Variables**: Define the settings related to the build execution.
- **Steps**: Show the list of tasks that the build definition executes.

Depending on the template chosen in the creation, these configurations could change. Also, the code editor is fantastic: IntelliSense makes the typing smooth, and other utilities such as the command palette simplify the user interaction.

Tip: Even if, in this case, the trigger in the YAML is set to master, the actual branch considered during the run is the same as the one where the pipeline definition resides.

What's next

In this chapter, you had a taste of how easy it is to create a pipeline definition using Azure Pipelines. Of course, there are lots of new and unexplored concepts, which are illustrated in detail in the upcoming chapters.

Chapter 3 The GalaxyHotel Project Sample

In the previous chapter, you created a pipeline for a simple application to see the basic features of Azure Pipelines. In this chapter, I will introduce the main project of the book, the **GalaxyHotel** project, which we will use from now on to take a closer look at Azure Pipelines features. The approach I'm going to use is practical: using this sample project, I will create a build and release pipeline definition for each application, showing you features and best practices. This way, you have a list of recipes and scenarios to reference for your day-to-day work.

About the project sample

The solution used in this book is a system that helps manage a hotel. The system is composed of two web applications—one for the customers, and one for the helpdesk—and a couple of libraries. The applications simulate the booking of the room.

To deploy these applications, you are going to configure a specific pipeline for each project and see the features available in Azure Pipelines in detail.

All projects are based on the .NET Core Framework. They are written using C#, and are distributed under the MIT License on:

- GitHub
- Azure DevOps (full projects and pipelines)

Note: In the GitHub repository, you will find all projects in the same repository.

You can fork the repository or download it to try the example illustrated later. If you notice any problem during its use, please open an issue or contact me.

Part of the core architecture of the solution is inspired by the eShopOnWeb project. I consider this project an excellent example of how you can organize your code following modern app architecture principles.

The following table shows how projects are organized in the Azure DevOps project.

Table 3.1: Project organization in Azure DevOps

Project name	Repository name	Solution/project file
Shared Libraries	Libraries	GalaxyHotel.Libraries.sln GalaxyHotel.Core.csproj GalaxyHotel.Dto.csproj GalaxyHotel.Infrastructure.csproj GalaxyHotel.Infrastructure.IntegrationTests.csproj

Project name	Repository name	Solution/project file
		GalaxyHotel.Core.Services.Tests.csproj
GalaxyHotel Website	Website	GalaxyHotel.Website.sln GalaxyHotel.Website.csproj GalaxyHotel.UITests.csproj
GalaxyHotel Helpdesk	Helpdesk	GalaxyHotel.HelpDesk.sln GalaxyHotel.HelpDesk.csproj

To open and modify the projects, you need a stable version of Visual Studio, available here. Regarding the .NET Core Framework, install the version 3.1 available here.

Projects are split into multiple repositories for the following reasons:

- **Complexity**: To have the complexity of real-life scenarios, in which multiple applications are developed by various teams and can follow different workflows.
- **Software versioning**: So that each repository can have its own version number (I'm going to discuss this later).

The next chapters explain step by step how to build and deploy each project.

Tip: If you want to import these repositories into your Azure Repos account, you can follow this guide.

Azure resources

All the projects in the solution are deployed to Microsoft Azure, and you need an active subscription to create the resources. It is possible to create a free subscription (with a few limitations) here.

The following table summarizes the list of resources used in this book.

Table 3.2: Azure resources

Type of resource	Name	Notes
Resource group	succinctly	
App services	galaxyhotel-dev, galaxyhotel-qa, galaxyhotel	Runtime Stack: .NET Core 3.1
Azure SQL databases	galaxyhotel-dev, galaxyhotel-qa, galaxyhotel	Choose the less expensive plan, no elastic pool. Allow Azure Services to access the server.

Type of resource	Name	Notes
Microsoft Windows 10 virtual machines	helpdesk-test, helpdesk1, helpdesk2	Enable IIS and install the ASP.NET Core Hosting Bundle on each VM. One of these machines can also be used to execute UI tests in Chapter 5. A software requirement is the latest version of Google Chrome.

Chapter 4 Deploy the Libraries

Chapter 2 gave you an overview of how you can create a build pipeline using both the visual designer and the YAML pipeline. Starting from this chapter, you are going to reuse and improve that knowledge by creating a release pipeline for the libraries available in the GalaxyHotel project. The purpose is to explain which strategies you can use to deploy the packages used in other projects, and to apply a reliable versioning scheme to avoid dependency hell.

Build the GalaxyHotel libraries solution

The first step to building the libraries is to add a new YAML pipeline in Azure Pipelines. As explained in the previous chapter, the solution is GalaxyHotel.Libraries.sln. The steps to create a new pipeline are similar to those used in Chapter 2: go to the **Azure Pipelines** section in Azure DevOps, then go to the **Pipeline** section. Click **Create Pipeline** (Figure 4.1).

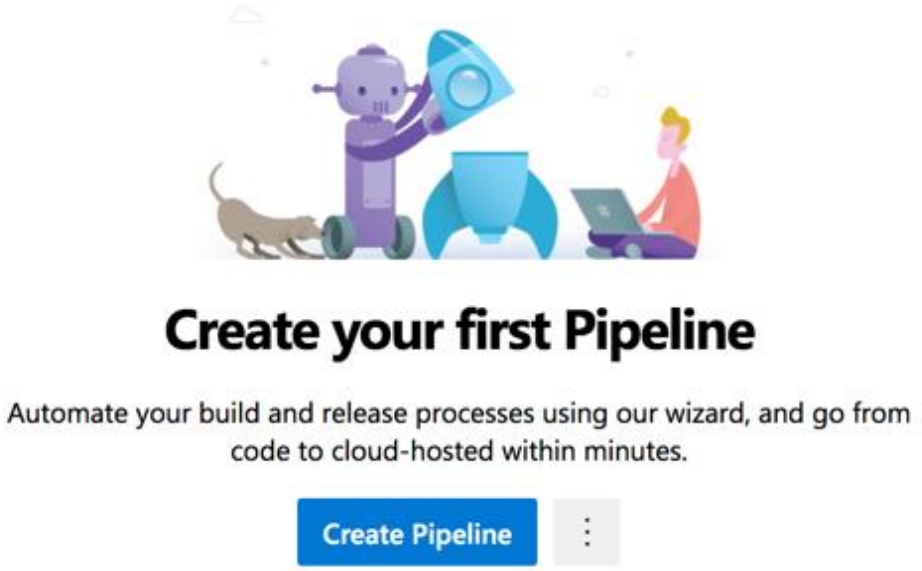


Figure 4.1: Create a new pipeline for libraries solution

Now, you have to choose which repository is the source of the pipeline. Select **Azure Repos Git**, and then select the **Libraries** repository (Figure 4.2).

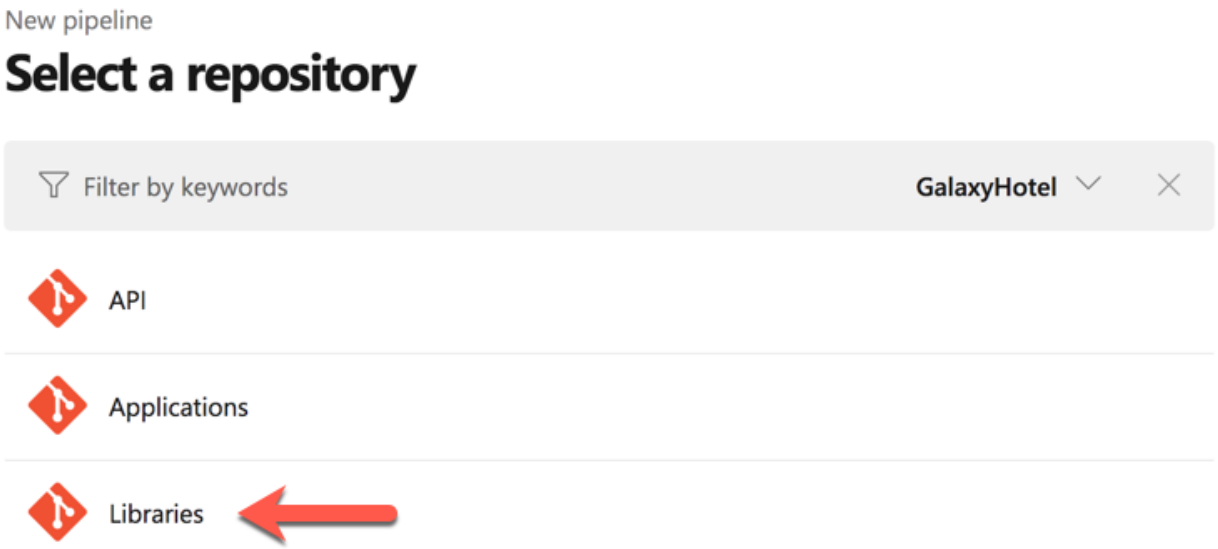


Figure 4.2: Selecting the source repository for a build

In the next step, choose the **ASP.NET Core** template from the list. The YAML created contains just the build task, but to create the NuGet packages, you need to replace it with the implementation in Code Listing 4.1.

Code Listing 4.1

```
trigger:
- master

pool:
  vmImage: 'ubuntu-latest'

variables:
  buildConfiguration: 'Release'

steps:
- task: DotNetCoreCLI@2
  displayName: Build libraries
  inputs:
    command: 'build'

- task: DotNetCoreCLI@2
  displayName: Test libraries
  inputs:
    command: 'test'

- task: DotNetCoreCLI@2
  displayName: Pack libraries
  inputs:
```

```
    command: 'pack'
    packagesToPack: '**/*.csproj'
    versioningScheme: 'off'

- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: '$(Build.ArtifactStagingDirectory)'
    ArtifactName: 'drop'
    publishLocation: 'Container'
```

This new workflow has four tasks: **`build`**, **`test`**, **`pack`**, and **`publish`** artifacts. The first two steps are pretty obvious. The third step creates a NuGet package for each library project using the nuget pack command and stores them in the **Build.ArtifactStagingDirectory** folder (which is the default folder for a build artifact). A predefined variable is set to identify this folder, and it points to a specific folder on the machine where the agent that executes the build process is hosted. Azure Pipelines has many predefined variables; you can find the full list here.

The last step in the YAML definition uses the same folder to expose the artifacts to the UI, or other pipelines or services. In this context, an artifact is the output of a build process.

> ***Note: For historical reasons, the artifact folder contains a subfolder named** drop. **It is possible to rename it according to your needs.***

Let's save and start the pipeline. Click **Save & Run** on the top of the page, which will open a side window as in Figure 4.3.

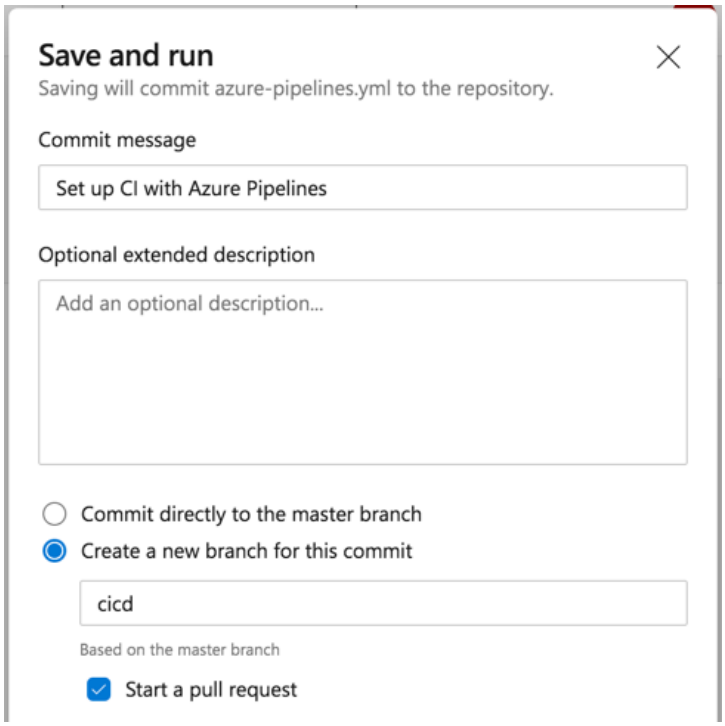


Figure 4.3: Save and run settings

Here you can define the details related to your run, add a new branch (which I always prefer), and create a new pull request. Click **Save & Run** at the bottom. The build starts executing all the steps defined in the YAML. Once the build is completed, the NuGet packages are available for use (Figure 4.4).

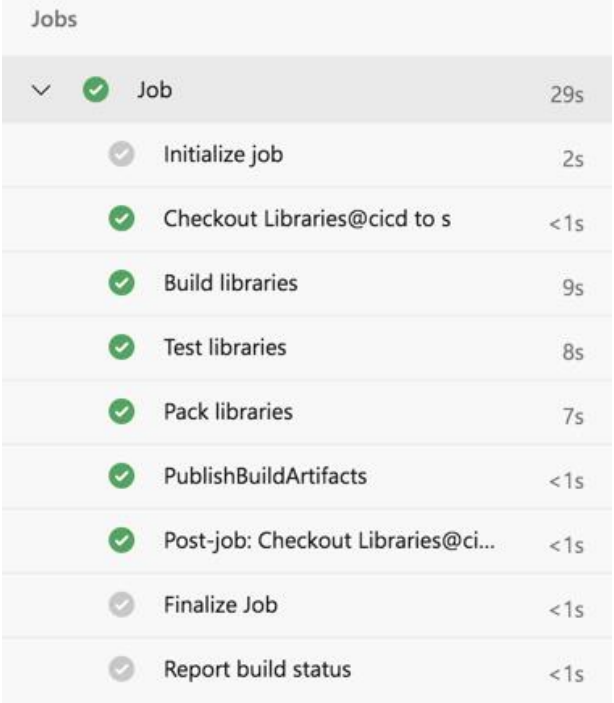


Figure 4.4: Build steps

In the build summary, more specifically in the **Related** section, there is a link that points to the published artifact (Figure 4.5).

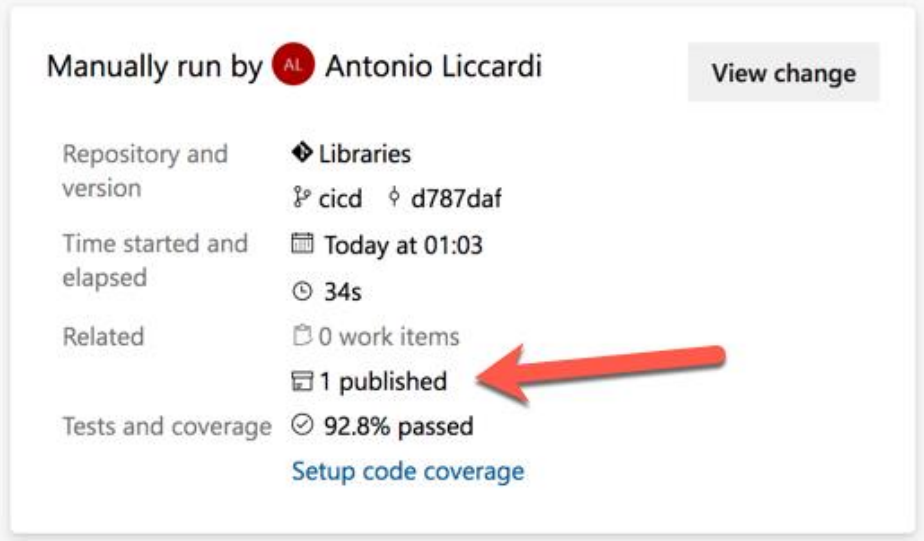


Figure 4.5: The build summary

Click it to open the **drop** folder, which contains the NuGet packages (Figure 4.6).

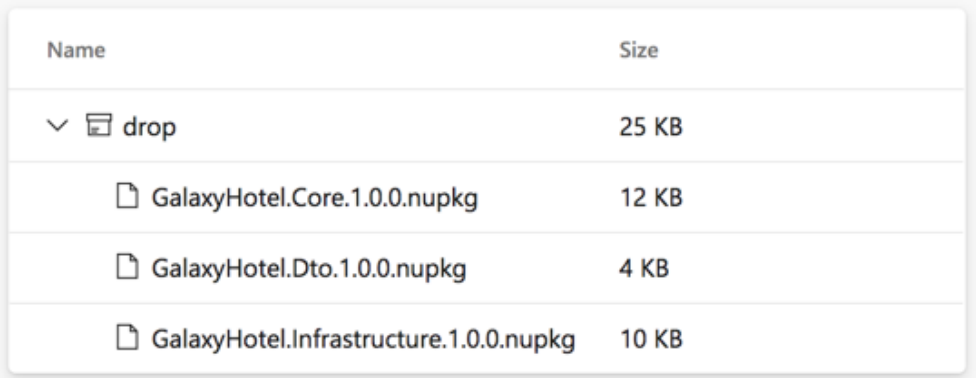


Figure 4.6: The Drop folder

The packages are ready to be deployed.

Deploy a NuGet package

Now that the packages are created, you are ready to deploy them. As explained in Chapter 1, **Azure Artifacts** allows you to host packages of multiple sources, and makes them available for developers or build pipelines.

Create a package feed

In the Azure Artifacts section, it is possible to create new feeds, depending on what you need to host. By default, a first feed is created during the project creation and has its name.

To create a new feed, click **Create Feed** and provide the following settings:

- **Name**.
- **Visibility**, allowing any member of your organization or just specific users.
- **Upstream sources**, including packages from common public sources.
- **Scope**, setting the scope to the current project or the entire organization.

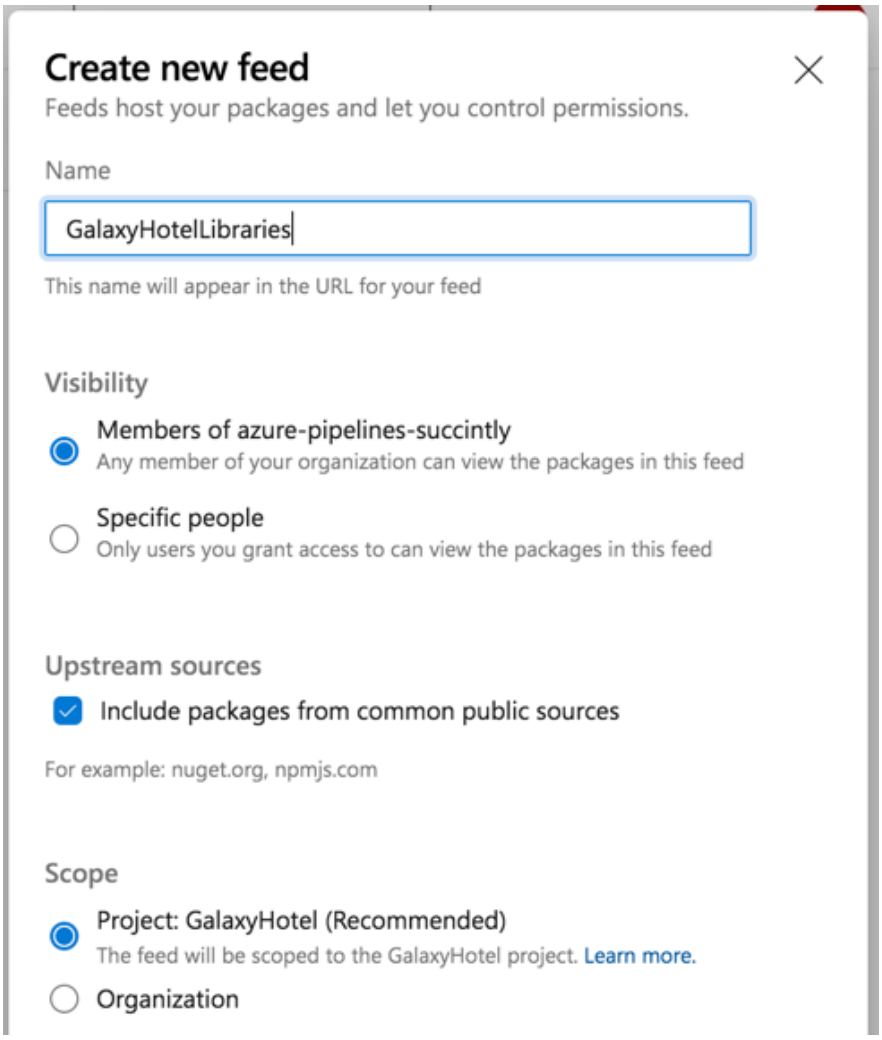


Figure 4.7: Create a new package feed

These settings can also be changed using the **Settings** wheel on the right side ( ), where you can specify the following (Figure 4.8):

- Name
- Description
- Deleted packages options
- Package sharing
- Retention policies
- Permissions
- Views
- Upstream sources

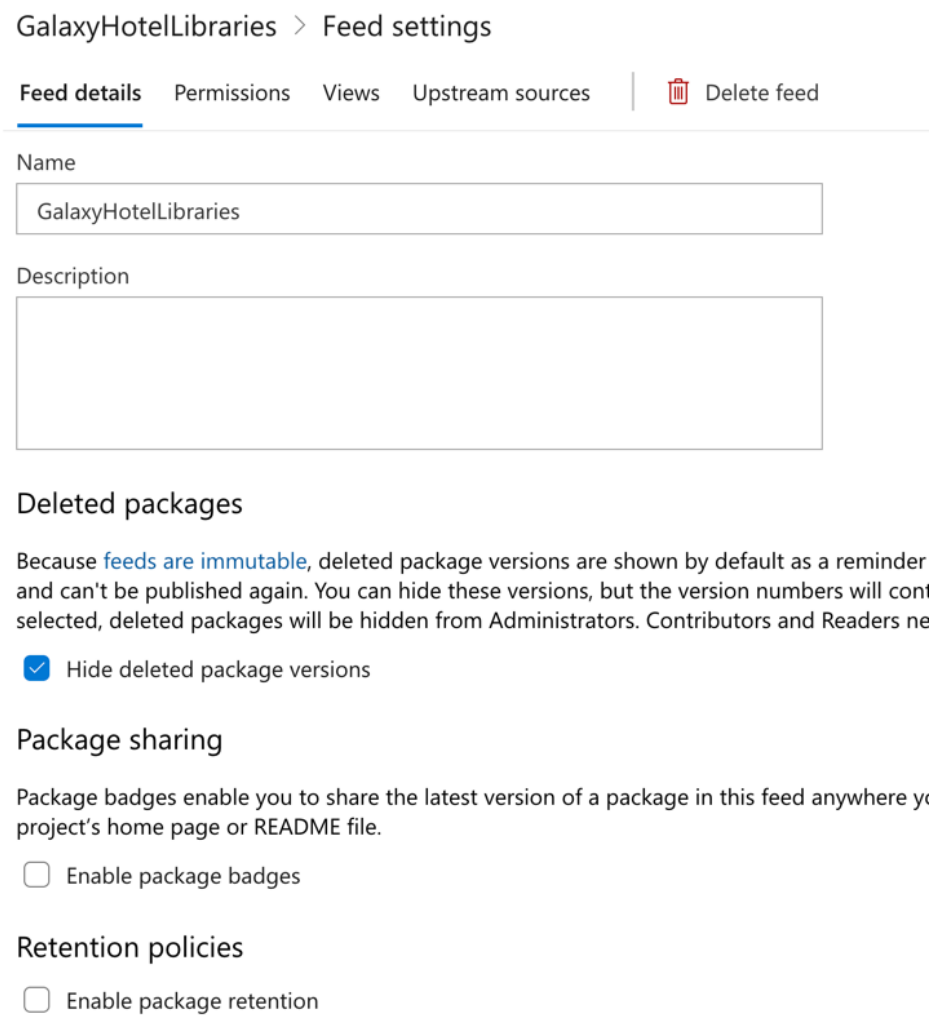


Figure 4.8: Feed settings

Tip: Instead of creating a new feed as explained here, you can also choose to rename the default feed with the name you prefer.

To use the feed, you can follow the instructions in the **Connect to feed** section, which are very detailed, and are articulated by the tool you are using (Figure 4.9).

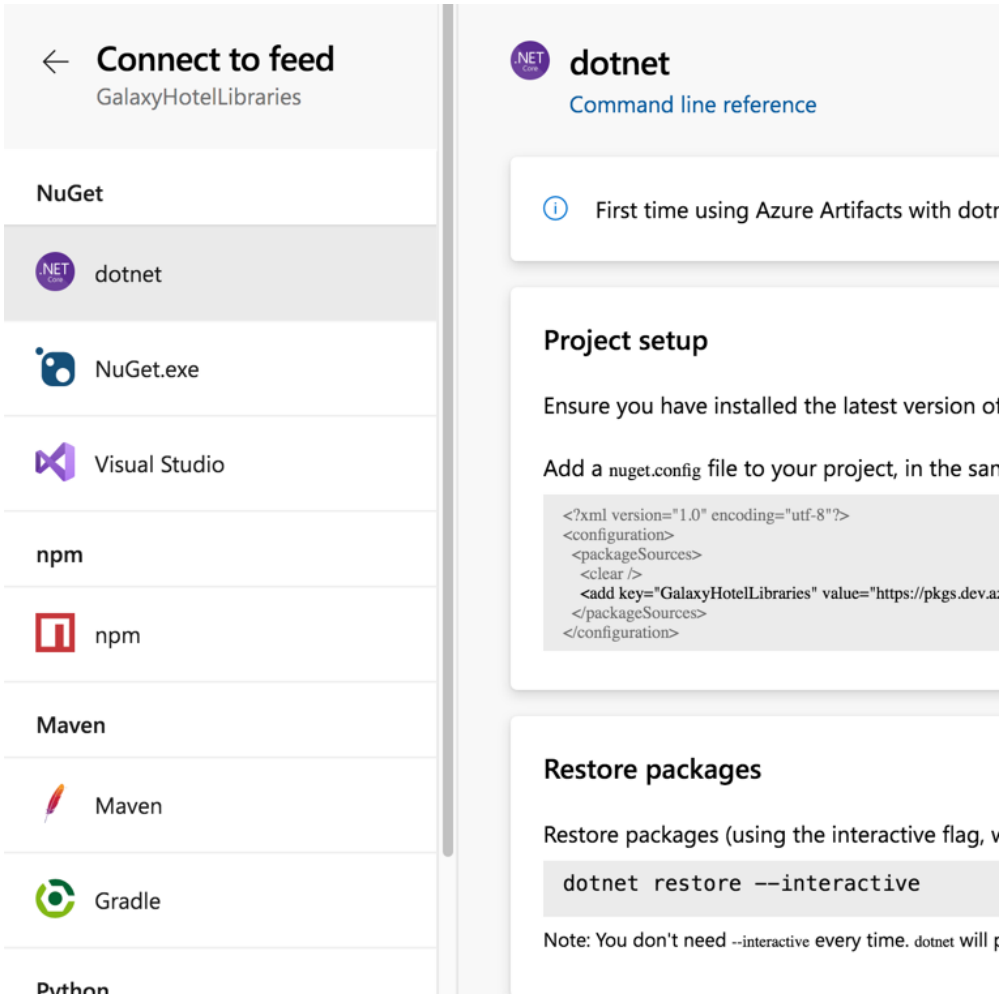


Figure 4.9: Connect to the feed

Deploy the package to the feed

Once the feed is created, you can update your YAML pipeline to deploy the packages. First, remove the **Publish build artifacts** task in the YAML, because it is not needed anymore. At the same position, with the help of the assistant, add a new .NET Core CLI task, assigning nuget push in the command field. In the **Target feed** field, select the feed created in the previous step. The task should appear as it does in Figure 4.10.

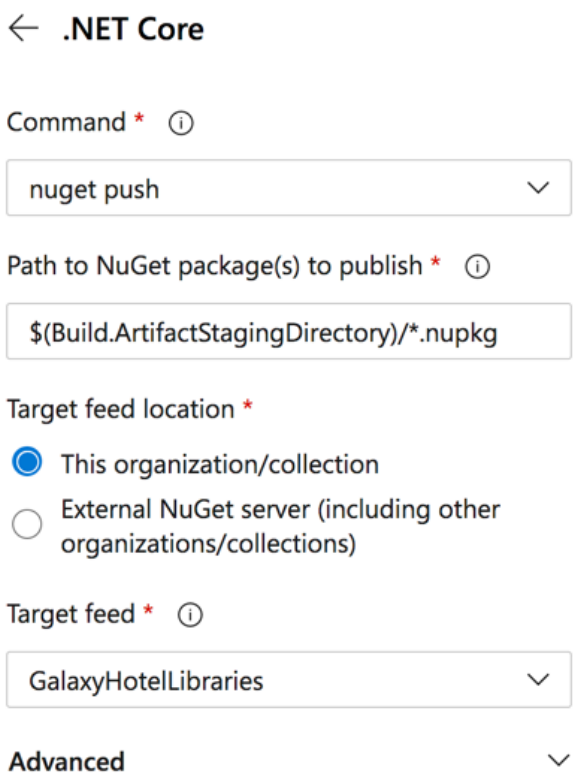


Figure 4.10: Adding the NuGet push task

Click **Add**, and Azure Pipelines generates the following YAML snippet:

Code Listing 4.2

```
- task: DotNetCoreCLI@2
  displayName: Push libraries
  inputs:
    command: 'push'
    packagesToPush: '$(Build.ArtifactStagingDirectory)/*.nupkg'
    nuGetFeedType: 'internal'
    publishVstsFeed: '<feed_guid>'
```

Save the pipeline definition and run the build. At completion, check the package deployed on the new feed (Figure 4.11).

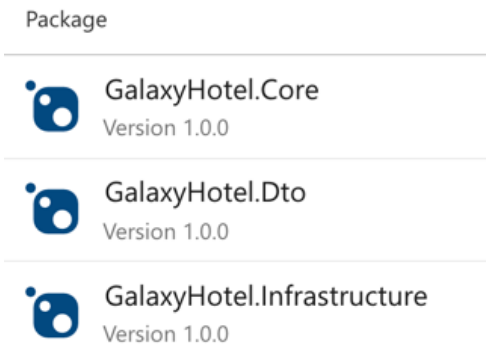


Figure 4.11: Packages deployed during pipeline execution

Now the branch **`cicd`** (continuous integration/continuous delivery) has completed its scope and is ready to be merged back on the **`master`** branch using the pull request created previously in the **Pull Requests** section of Azure Repos.

Once the PR is opened, click **Complete** > **Complete Merge**.

Versioning NuGet packages manually

You may have noticed that each deployed package has the version number 1.0.0. Where does this number come from? When you create a new package, the **`nuget pack`** command will assign a default version number if one is not specified. If you download one of the packages created in the previous paragraph and change the extension from **`.nupkg`** to **`.zip`**, you should see a **`.nuspec`** file that contains all the information related to the package, including the version number.

In many cases though, you need to use a different versioning scheme than the one assigned by default by NuGet. In my experience, the first challenge is defining a versioning strategy. As a matter of fact, without a well-defined strategy, developers could arbitrarily choose a new version number, which usually doesn't follow the rules. And this can lead to problems!

This is important because it can lead to *package immutability* and *dependency hell*. Also, having a well-defined version scheme helps identify releases for bug fixing.

Package immutability

Some package management systems, such as Azure Artifacts, have a strict policy about version numbers: once a package is uploaded, the release number becomes permanently reserved. *Immutability* is a core concept for packages: it ensures that a specific version of a library stays the same despite the time.

The main problem with updating a package with the same version is the cache. For example, when you retrieve multiple versions of the same package from NuGet, a local cache on your machine speeds up the process of downloading it, but it can use the cached version of the library. Also, consider that the same problem could arise during build execution. You can find more info about this topic here and here.

Dependency hell

The term “dependency hell” identifies a situation where you have multiple versions of a dependency in your code referenced by various projects or dependencies. Figure 4.12 shows an example of how different projects reference different versions of GalaxyHotel.Dto. In the worst-case scenario, your application could not compile or generate an exception during runtime.

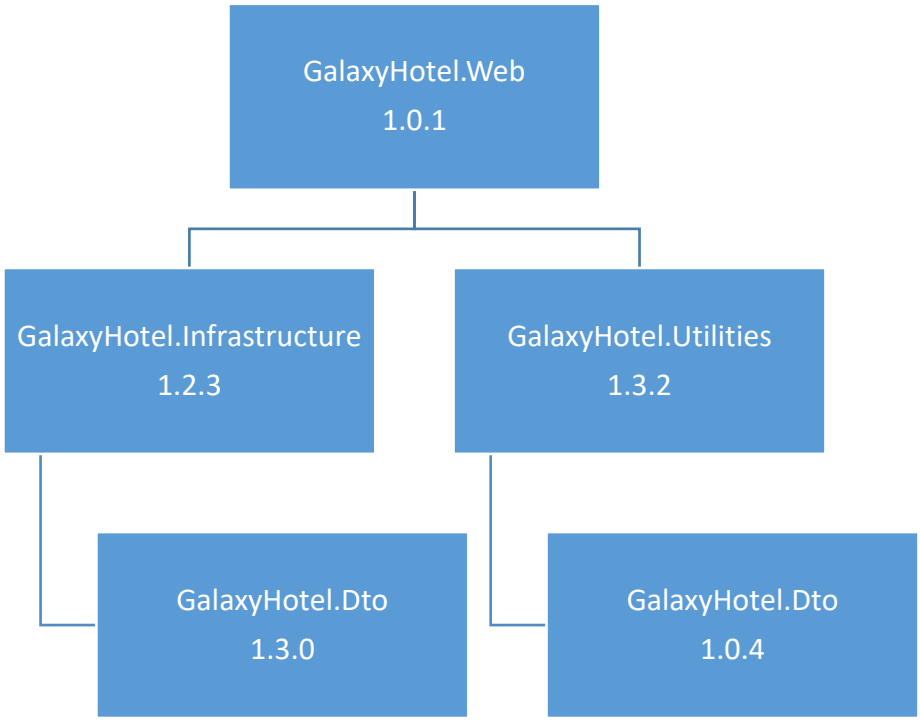


Figure 4.12: Dependency hell example

One way to overcome this problem in the .NET platform is the binding redirect feature, where you specify which version your application should use. Another approach consists of using the version ranges feature, where you specify which versions of your dependency are allowed, narrowing problems during compilation or runtime execution. These are some techniques that help solve these problems, but the important thing is to have a process.

Introducing SemVer

SemVer is a set of rules that help version your software. Practically, it is based on an ordered three-number version scheme where each number should be incremented depending on what changes you make in your code. The versioning scheme is based on the pattern **`MAJOR.MINOR.PATCH`**, and each number is changed based on these rules:

- **`Major`**: Breaking change with the past.
- **`Minor`**: Additional features that are compatible with the past.
- **`Patch`**: Bug fixes that are compatible with the past.

SemVer also includes a suffix that serves as pre-release and builds metadata. Here's an example of SemVer version **`2.0.0-rc.1+542`**, where:

- **`2.0.0`** is the version number.
- **`rc.1`** is the pre-release part.
- **`542`** is the build metadata.

The second and third bullets are beneficial to handling the dependency hell problem. Indeed, these numbers serve as intermediate versions dedicated to the testing phase of new features. In case of problems, you can change the pre-release and the build metadata part without changing the main version number. SemVer also works well with version ranges, where you can specify which versions of a dependency you want to reference despite being upgraded. All the rules related to the order of the numbers can be found here.

Due to its simplicity, SemVer is wildly adopted to version APIs, components, and software in general, and it's essential to understand that it is an arrangement between developers (not customers!) defining a standard process to version software.

Assign a version number using the .NET Framework

Both the .NET platform and Azure Pipelines provide many ways to assign a version number.

If you are using the using the .NET Framework (full, not Core), the **`AssemblyInfo.cs`** file is the first place to use. This class supplies three attributes to set:

Code Listing 4.3

```
// Version information for an assembly consists of the following four
// values:
//
//      Major Version
//      Minor Version
//      Build Number
//      Revision
//
// You can specify all the values or you can default the Build and Revision
Numbers
// by using the '*' as shown below:
[assembly: AssemblyVersion("1.0.1.0")]
[assembly: AssemblyFileVersion("1.0.1.0")]

// Use AssemblyInformationalVersion when you need to specify more info
about the version such as the build number, the date, or even the git
commit hash:
[assembly: AssemblyInformationalVersion("1.0.1.0-efae3b7")]
```

As shown in the code listing, there are three possibilities:

- **`AssemblyVersion`**: Defines the version of the assembly.
- **`AssemblyFileVersion`**: Defines the version for the Win32 file used by the operating system. It is optional, and can be different from the first one (more info available here).
- **`AssemblyInformationalVersion`**: Also optional, and can be used to specify a version not only consisting of numbers, but also suffixes, which include metadata such as the build number, the date, or even the **`git`** commit hash.

If you are using .NET Core, you can take advantage of the following properties in the project file:

- **`VersionPrefix`** (major.minor.patch[.build])
- **`VersionSuffix`** (alphanumeric [0-9A-Za-z-]*)
- **`Version`** (major.minor.patch[.build][-prerelease])

`VersionPrefix` and **`VersionSuffix`** are usually used together, and let you specify a pair of base version and metadata, such as **`0.1.0-beta-2`**, where the prefix is **`0.1.0`**, and the suffix is **`beta-2`**. If the **`Version`** field is set in the project file, the prefix and suffix are not considered.

It is also possible to specify the version using the .NET CLI, as in the following snippet:

Code Listing 4.4

```
// Setting the package version during build
dotnet build --configuration Release -p:Version=0.1.0-beta2

// Setting the package version during pack
dotnet pack -p:PackageVersion=0.1.0-beta2

// Setting the package version prefix during pack
dotnet pack --version-suffix "0.1.0"

// Setting the package version suffix during pack
dotnet pack --version-suffix "beta-2"
```

Visual Studio also provides in the project properties a simple way to manage the version in the **Package** section (Figure 4.13).

Figure 4.13: Package settings in Visual Studio

Assign a version number using NuGet

Using NuGet, you can assign a version number using the **.nuspec** file or the NuGet CLI (downloadable here). In the first case, you need to generate a **.nuspec** file for your project by running the command **nuget spec**. The result is similar to the following listing, where you can assign a new version value as well as all the package-related info, including the referenced assemblies.

Code Listing 4.5

```
<?xml version="1.0"?>
<package >
  <metadata>
    <id>$id$</id>
    <version>$version$</version>
    <title>$title$</title>
    <authors>$author$</authors>
    <owners>$author$</owners>
    <licenseUrl>http://LICENSE_URL_HERE_OR_DELETE_THIS_LINE</licenseUrl>
    <projectUrl>http://PROJECT_URL_HERE_OR_DELETE_THIS_LINE</projectUrl>
    <iconUrl>http://ICON_URL_HERE_OR_DELETE_THIS_LINE</iconUrl>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>$description$</description>
    <releaseNotes>Summary of changes made in this release of the
package.</releaseNotes>
    <copyright>Copyright 2020</copyright>
```

```
    <tags>Tag1 Tag2</tags>
    <dependencies>
      <!-- An example of dependency package with a version range -->
      <dependency id="ExamplePackage" version="[1.4.5,1.6)" />
    </dependencies>
  </metadata>
</package>
```

Then using the NuGet CLI, you can pass the **nuspec** as input to create your package: **nuget pack <project-name>.nuspec [-IncludeReferencedProjects]**.

The switch **IncludeReferencedProjects** will automatically include the referenced assemblies as part of the package.

Assigning a version number with the NuGet CLI is simple, as shown in the following command: **nuget pack <project-name>.nuspec -Version 0.1.0**

Assign a version number using Azure Pipelines

As you can see in Code Listing 4.1, the **pack** task contains a property named **versioningScheme**, which allows you to specify a version number for your package. The available options are:

- Off.
- Use the date and time.
- Use an environmental variable.
- Use the build number.

The third option requires an environmental variable in the build execution (more about this topic in the next chapter). The fourth option requires you to specify the value of the name setting in the YAML pipeline, as shown in Code Listing 4.6.

Code Listing 4.6

```
# Setting the build number (more info available here)
name:
$(TeamProject)_$(Build.DefinitionName)_$(SourceBranchName)_$(Date:yyyyMMdd)
$(Rev:.r)
steps:
- script: echo hello world
```

In the same task, it is possible to define the specific number using the additional build properties (**buildProperties**), as seen in Code Listing 4.7.

Code Listing 4.7

```
# azure-pipelines.yml build definition
```

```
- task: DotNetCoreCLI@2
  displayName: Pack libraries
  inputs:
    command: 'pack'
    packagesToPack: '**/*.csproj'
    versioningScheme: 'off'
    buildProperties: 'Version=0.1.0-beta2'
```

Versioning NuGet packages during build

As discussed previously, having a versioning strategy helps both during the development and usage of a library. But very often, a versioning scheme is paired with a branching strategy, especially if you are using Git. Having both mechanisms in place helps you to achieve a fully automated build and release process, thanks to a well-defined process. One positive note is that Azure DevOps already includes features and extensions to achieve automation, especially if using SemVer.

Git branching strategies

Branching in Git is a relatively simple process, and for this reason, you can easily apply a workflow in it using branches. In this section, I'm going to explain three common strategies: feature branching, GitFlow, and forking.

Feature branching, sometimes referred to as trunk-based or GitHub flow, is an approach in which all new development is done using a new branch defined as a feature branch. All feature branches start from the main branch, usually the **`master`**, and will merge back on it at the end of the feature development. The merging process is based on a pull request (PR) mechanism, in which a developer opens a PR and allows other developers to review the code, give feedback, and suggest changes. The deployment can happen before the merge, helping the test phase, or after the merge, bringing the feature online. It's sometimes helpful to have a feature toggle mechanism in place, letting users choose when a new feature is enabled. When using this strategy, you can quickly achieve **continuous delivery** by choosing the right time to deploy or enable a feature, delivering value to the users. You can find a good explanation of GitHub flow here.

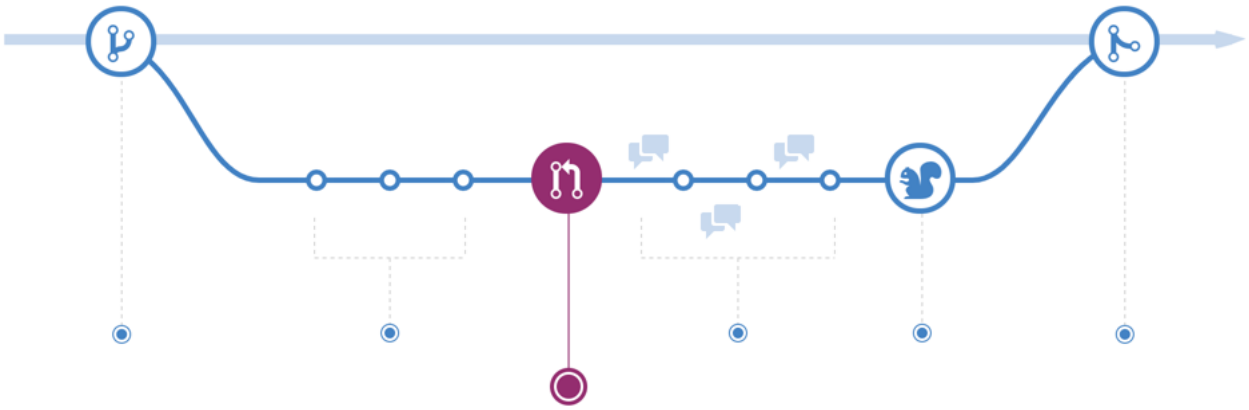


Figure 4.14: GitHub flow workflow

GitFlow is a workflow originally created by Vincent Driessen. It is mainly focused on the concept of release, and it is well suited for collaboration on large projects with release cycles. It shares the feature approach with the previous branching strategy, but increases the number of branches used, giving them meaning.

Development starts from the **`develop`** branch, which contains all the new features for the next software release. Feature branches are created from this branch, usually using the **`feature`** prefix. When ready to release, a new **`release`** branch is opened, and after testing, you can go to production, merging the release branch onto the **`master`** branch. But GitFlow also includes branches to manage bugs and hotfixes. These are all types of branches:

- **`develop`**
- **`feature/<feature_branch_name>`**
- **`release/<release_branch_name>`**
- **`master`**
- **`bugfix/<bugfix_branch_name>`**
- **`hotfix/<hotfix_branch_name>`**
- **`support/<support_branch_name>`**

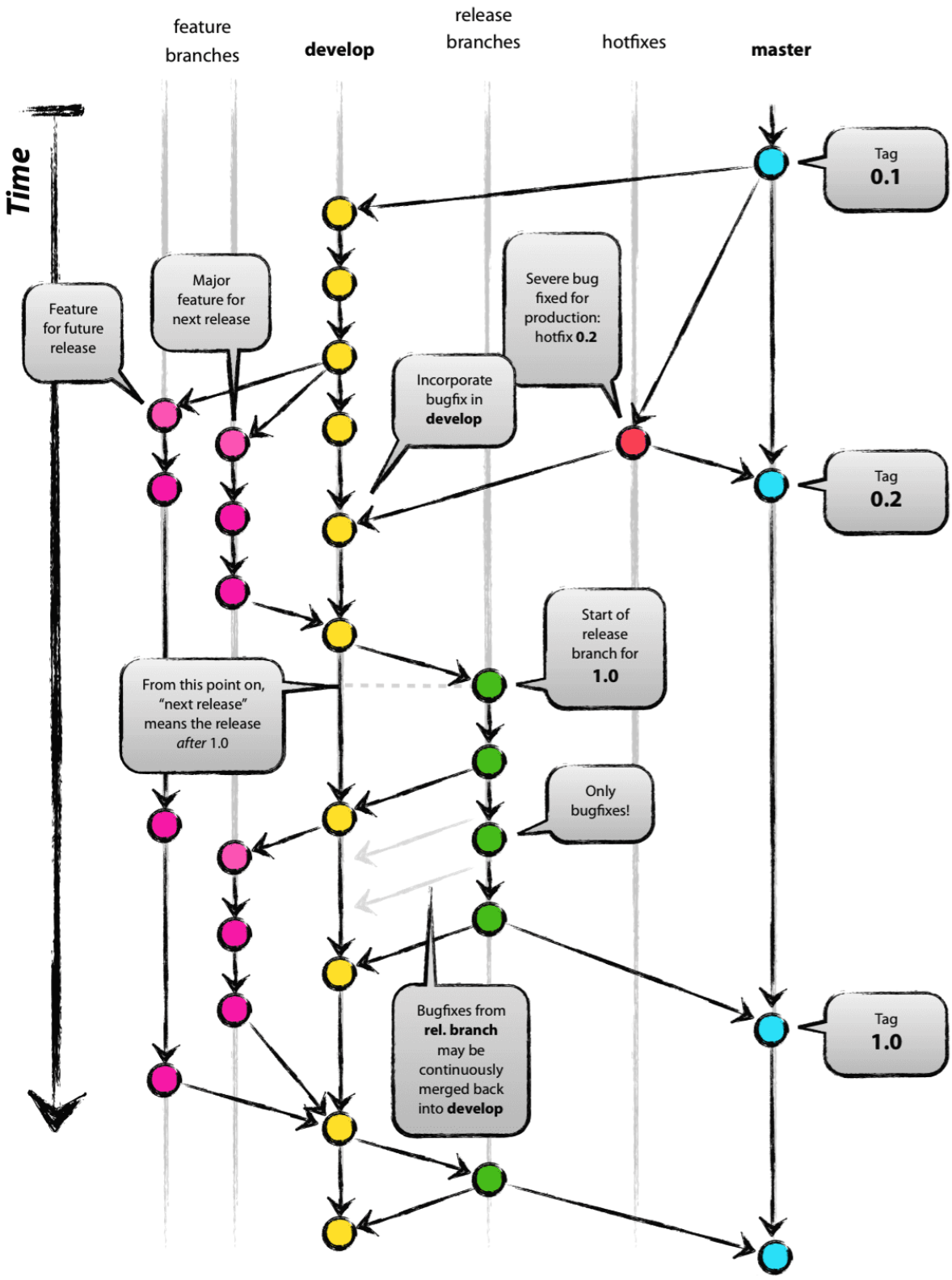


Figure 4.15: GitFlow branching overview

(Source: https://nvie.com/posts/a-successful-git-branching-model/)

Pull requests can also be used in this workflow strategy to review and validate code before merging on the develop or master branch. What is interesting is the **`git-flow`** CLI that comes with GitFlow, which helps create and manage (with branching and merging) all the branches listed before. An example of how to use it is available in the next section, and you can find a detailed reference here.

Finally, **forking** branching is mostly used in open-source projects or in organizations that promote the inner source approach. Developers make a copy of a Git repository they want to contribute to, and all changes are applied to their own repository. At the end of the development, they usually propose to promote their changes to the original repository using a pull request. In Azure Repos, you can fork a repository using the **Fork** button in the repo page, located in the three-dot menu near the clone button (Figure 4.16).

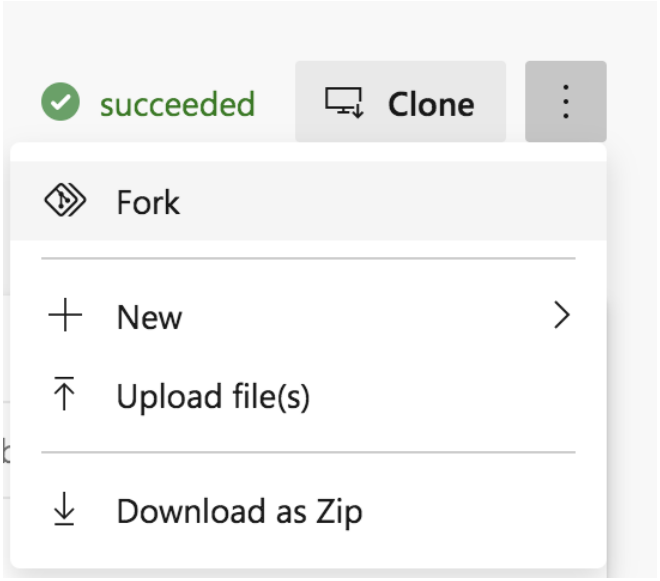


Figure 4.16: The Fork button

Using git-flow

As explained before, GitFlow provides a toolset to help developers managing branches. Usually, if you have Git for Windows or Visual Studio, there is a good chance that you already have **`git-flow`**. You can check by opening a command-line interface (cmd); go to your **Libraries** repo folder, and type **`git flow`**.

Tip: If you are not familiar with the command-line interface, you can download a visual editor such as SourceTree, the GitFlow for Visual Studio extension, or one of the available GitFlow extensions for Visual Studio Code.

Note: In a previous section, you merged the cicd branch to master using a PR. Remember to switch to the master branch locally and pull the new changes from the remote. Also, delete the local cicd branch, as explained here.

The following output should appear:

Code Listing 4.8

```
C:\azure-pipelines-succinctly\Libraries>git flow
usage: git flow <subcommand>

Available subcommands are:
   init      Initialize a new git repo with support for the branching
model.
   feature   Manage your feature branches.
   bugfix    Manage your bugfix branches.
   release   Manage your release branches.
   hotfix    Manage your hotfix branches.
   support   Manage your support branches.
   version   Show version information.
   config    Manage your git-flow configuration.
   log       Show log deviating from base branch.

Try 'git flow <subcommand> help' for details.
```

Typing **`git flow init`** will start the configuration wizard of GitFlow, asking questions related to the branch naming convention to use.

Code Listing 4.9

```
C:\azure-pipelines-succinctly\Libraries>git flow init
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
Hooks and filters directory? [C:/azure-pipelines-
succinctly/Libraries/.git/hooks]
```

In this case, I choose default values for my configuration. **`git-flow`** saves the newly created configuration in the **`config`** file inside the **`.git`** folder of your repository.

Note:** `git-flow` **configuration is stored on the local machine, and it is not pushed to the remote repository. Each developer should execute the** `init` **command to have the same configuration. To share the configuration, you can use a** `.gitflow` **file, as explained here.

To create a new feature, you can use the **`git flow feature start <feature_name>`** command.

Code Listing 4.10

```
C:\azure-pipelines-succinctly\Libraries>git flow feature start add-cart-
page
Switched to a new branch 'feature/add-cart-page'

Summary of actions:
- A new branch 'feature/add-cart-page' was created, based on 'develop'
- You are now on branch 'feature/add-cart-page'

Now, start committing on your feature. When done, use:

     git flow feature finish add-cart-page


C:\azure-pipelines-succinctly\Libraries>git branch
  develop
* feature/add-cart-page
  master
```

As shown in the code listing, **`git-flow`** creates a new branch with a **`feature`** prefix. You are ready to work, and at the end of the feature, the command **`git flow feature finish`** will merge the branch back on the **`develop`** branch.

Code Listing 4.11

```
C:\azure-pipelines-succinctly\Libraries>git flow feature finish
Switched to branch 'develop'
Already up to date.
Deleted branch feature/add-cart-page (was 16fafc7).

Summary of actions:
- The feature branch 'feature/add-cart-page' was merged into 'develop'
- Feature branch 'feature/add-cart-page' has been locally deleted
```

```
- You are now on branch 'develop'
```

To create a new release, type the **`git flow release start 1.0.0`** command: here, you are defining the **next version number** for your product (Code Listing 4.12). Using SemVer, you can easily apply a few rules to determine this number based on which types of features you are using. One question could be: Am I breaking anything with the past, or am I just adding a new feature?

Code Listing 4.12

```
C:\azure-pipelines-succinctly\Libraries>git flow release start 1.0.0
Switched to a new branch 'release/1.0.0'

Summary of actions:
- A new branch 'release/1.0.0' was created, based on 'develop'
- You are now on branch 'release/1.0.0'

Follow-up actions:
- Bump the version number now!
- Start committing last-minute fixes in preparing your release
- When done, run:

     git flow release finish '1.0.0'
```

After completing the testing part, you can close the release and merge on the master branch using the command **`git flow release finish '1.0.0'`**, and **`git-flow`** will save you from doing all the steps in the **`Summary of actions`** section:

Code Listing 4.13

```
Switched to branch 'develop'
Already up to date!
Merge made by the 'recursive' strategy.
Deleted branch release/1.0.0 (was 1f1b758).

Summary of actions:
- Release branch 'release/1.0.0' has been merged into 'master'
- The release was tagged '1.0.0'
- Release tag '1.0.0' has been back-merged into 'develop'
- Release branch 'release/1.0.0' has been locally deleted
- You are now on branch 'develop'
```

Note: During the command, you will be asked to add a git tag message.

After the first release is created, you can start opening new features and repeating the cycle again.

In conclusion, having a standard naming convention for branches and giving them a functional meaning provide a significant advantage in terms of automation and versioning, as explained in the next paragraphs.

Using GitVersion

Embracing a standard workflow like GitFlow allows you to leverage **GitVersion**, a command-line tool that generates a compatible SemVer number for the next release.

GitVersion can be installed on many operating systems and using many options, as explained here. I usually install it using chocolatey, but it depends on the situation.

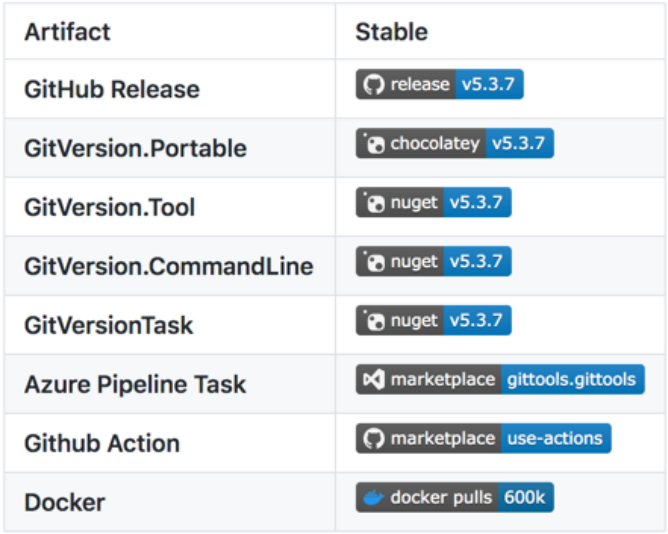


Figure 4.17: GitVersion installation options

Once installed, open the command-line interface, go to the repository folder, and type **gitversion**. The output should return a JSON payload similar to this:

Code Listing 4.14

```
C:\azure-pipelines-succinctly\Libraries>gitversion
{
  "Major":1,
  "Minor":1,
  "Patch":0,
```

```
  "PreReleaseTag":"alpha.5",
  "PreReleaseTagWithDash":"-alpha.5",
  "PreReleaseLabel":"alpha",
  "PreReleaseNumber":5,
  "WeightedPreReleaseNumber":5,
  "BuildMetaData":"",
  "BuildMetaDataPadded":"",

"FullBuildMetaData":"Branch.develop.Sha.09d4a7e4e0a5bcb739adddad84f0cf2b390
a63eb",
  "MajorMinorPatch":"1.1.0",
  "SemVer":"1.1.0-alpha.5",
  "LegacySemVer":"1.1.0-alpha5",
  "LegacySemVerPadded":"1.1.0-alpha0005",
  "AssemblySemVer":"1.1.0.0",
  "AssemblySemFileVer":"1.1.0.0",
  "FullSemVer":"1.1.0-alpha.5",
  "InformationalVersion":"1.1.0-
alpha.5+Branch.develop.Sha.09d4a7e4e0a5bcb739adddad84f0cf2b390a63eb",
  "BranchName":"develop",
  "EscapedBranchName":"develop",
  "Sha":"09d4a7e4e0a5bcb739adddad84f0cf2b390a63eb",
  "ShortSha":"09d4a7e",
  "NuGetVersionV2":"1.1.0-alpha0005",
  "NuGetVersion":"1.1.0-alpha0005",
  "NuGetPreReleaseTagV2":"alpha0005",
  "NuGetPreReleaseTag":"alpha0005",
  "VersionSourceSha":"8ac38e72c45a82fc72092c9c7c5ab7f9c86cb6c5",
  "CommitsSinceVersionSource":5,
  "CommitsSinceVersionSourcePadded":"0005",
  "CommitDate":"2020-07-11"
}
```

Within this payload, the **major**, **minor**, and **patch** version numbers are available, as well as a fully compatible **SemVer** value, which you can use to set the next release version of your software. GitVersion also provides a large number of key-value pairs, depending on what number you need to select (such as AssemblyVersion, AssemblyFileVersion, and so on). For example, a **NuGetVersion** is already available to assign the version number to a NuGet package.

As you may have noticed, I ran the **GitVersion** command on the **develop** branch: for this reason, the **SemVer** value includes the **alpha0003** pre-release part. By default, GitVersion generates a specific pre-release part based on which branch the command is executed. Also, the string **0003** is based on the number of commits on the develop branch. These default settings are configurable using the command **gitversion init**, which starts a wizard that lets you choose which workflow and strategy to use.

Code Listing 4.15

```
C:\azure-pipelines-succinctly\Libraries>gitversion init
GitVersion init will guide you through setting GitVersion up to work for
you

Which would you like to change?

0) Save changes and exit
1) Exit without saving

2) Run getting started wizard

3) Set next version number
4) Branch specific configuration
5) Branch Increment mode (per commit/after tag) (Current: )
6) Assembly versioning scheme (Current: )
7) Setup build scripts

>
```

Among the options you have:

- A wizard (2) where you can opt to select GitFlow or GitHubFlow workflow, and which increment mode you prefer (continuous delivery or continuous deployment).
- Set next version number (3).
- Configurations related to the branches (4 and 5).

Select option 2 to start the wizard and follow the instructions as seen in Code Listing 4.16.

Code Listing 4.16

```
The way you will use GitVersion will change a lot based on your branching
strategy. What branching strategy will you be using:

1) GitFlow (or similar)
2) GitHubFlow
```

```
3) Unsure, tell me more

> 1

By default, GitVersion will only increment the version of the 'develop'
branch every commit; all other branches will increment when tagged.

What do you want the default increment mode to be (can be overridden per
branch):

1) Follow SemVer and only increment when a release has been tagged
(continuous delivery mode)
2) Increment based on branch config every commit (continuous deployment
mode)
3) Each merged branch against master will increment the version (mainline
mode)
4) Skip

> 1

Questions are all done, you can now edit GitVersion's configuration further
```

Back in the main wizard page, you can use option 4 to change the pre-release tag of a specific branch (for example, **`rc`** instead of **`beta`** for release branches). At the end, option 0 saves the configuration in the **`GitVersion.yml`** file, as seen in Code Listing 4.17.

Code Listing 4.17

```
mode: ContinuousDelivery
branches:
  develop: {}
  release:
    tag: rc
ignore:
  sha: []
merge-message-formats: {}
```

Commit the file and push it to the remote repository.

Generate version number with GitVersion

Once the configuration is in place, it is possible to integrate GitVersion into the release pipeline created at the beginning of this chapter. GitVersion is available in Azure Pipelines thanks to the community contribution through extensions on the Marketplace.

Going back to Azure DevOps, you need to install the GitTools extension to use GitVersion in Azure Pipelines. Click **Get it free** to choose the organization, and then click **Install**. In the end, go to your pipeline and open the editor to edit it and make sure you are on the develop branch, as shown in Figure 4.18.

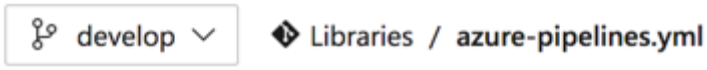


Figure 4.18: Select the branch that includes the GitVersion.yml file

In the assistant, type **GitVersion**. You should see two tasks as in Figure 4.19 (if not, the extension is not installed correctly).

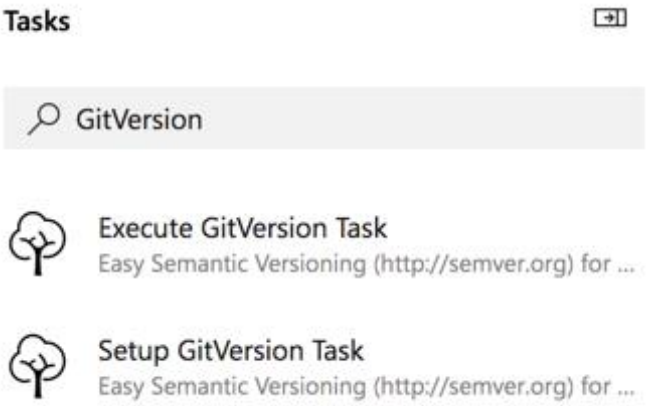


Figure 4.19: GitVersion tasks

Put the cursor just after the steps keyword and before the build task. Select **Setup GitVersion Task**, set the version as **5.x**, and click **Add**. This task will install GitVersion on the agent that executes the build.

Next, select **Execute GitVersion Task**, enable the **Specify Configuration file** flag, and enter **GitVersion.yml** in the **Configuration file** box. This task will:

- Generate a new compatible SemVer version number.
- Change the build name with the latest version number.
- Store all key-value pairs of GitVersion JSON payload as variables in the execution context of Azure Pipelines.

There are two additional changes to apply to the pipeline:

1. Add the trigger for the **`develop`** branch.
2. Configure the pack libraries task, setting a versioning scheme linked to the variable **`GitVersion.NuGetVersion`**.

Code Listing 4.18 shows the full updated YAML pipeline.

Code Listing 4.18

```
trigger:
- master
- develop

variables:
  buildConfiguration: 'Release'

steps:
- task: gitversion/setup@0
  displayName: Install GitVersion
  inputs:
    versionSpec: '5.x'

- task: gitversion/execute@0
  displayName: Generate a new version number
  inputs:
    useConfigFile: true
    configFilePath: 'GitVersion.yml'

- task: DotNetCoreCLI@2
  displayName: Build libraries
  inputs:
    command: 'build'

- task: DotNetCoreCLI@2
  displayName: Test libraries
  inputs:
    command: 'test'

- task: DotNetCoreCLI@2
  displayName: Pack libraries
  inputs:
```

```
    command: 'pack'
    packagesToPack: '**/*.csproj'
    versioningScheme: 'byEnvVar'
    versionEnvVar: 'GitVersion.NugetVersion'

- task: DotNetCoreCLI@2
  displayName: Push libraries
  inputs:
    command: 'push'
    packagesToPush: '$(Build.ArtifactStagingDirectory)/*.nupkg'
    nuGetFeedType: 'internal'
    publishVstsFeed: '834230a0-50c2-4971-954a-75eb9e3d47dd/dcb14a3c-e6ab-
48df-ac52-3ae79a5055f7'
```

Run the pipeline selecting the **`develop`** branch, and when it completes, you should see new packages with the version number created by GitVersion (Figure 4.20).

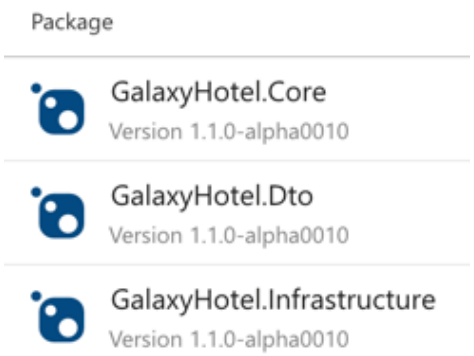


Figure 4.20: GitVersion applied to NuGet packages

With this mechanism in place, Azure Pipelines can directly version our software without us worrying about what the next version number will be. It is a reliable process and mitigates problems related to software versioning.

Feed views in Azure Artifacts

Feed views are a feature of Azure Artifacts that allow you to create release channels for your packages. By default, you have three views:

- **`@local`**
- **`@prerelease`**
- **`@release`**

The **`@local`** view is the default, and it contains all packages pushed to the main feed URL, as in the previous paragraphs. **`@prerelease`** and **`@release`** are two customizable views that can be used to promote packages based on quality. In this case, The **`@prerelease`** view is usually used when a package has a pre-release label in its version number. The package is then promoted to the **`@release`** view only after its validation, giving it an official release number (for example, **`1.0.0`** with no pre-release label).

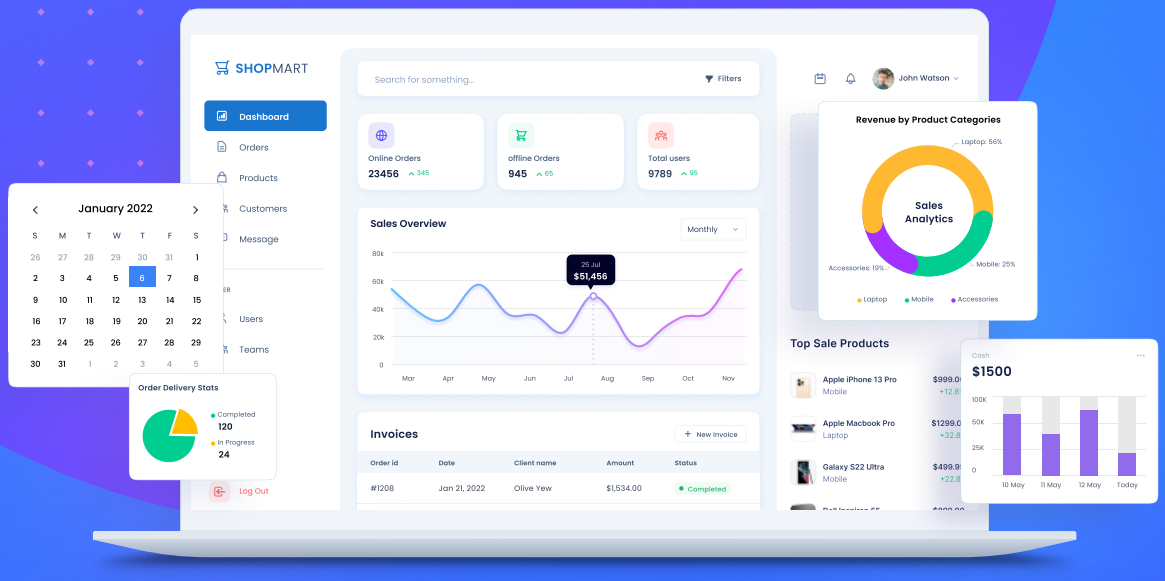
Feed views management is available in the settings of an Azure Artifacts feed, where you can set the name and specify which users can access a specific view. Each view has its own feed URL according to this template:

`https://pkgs.dev.azure.com/{yourteamproject}/_packaging/{feedname}@{Viewname}/nuget/v3/index.json`

You can promote the package from one view to another, as explained here.

THE WORLD'S BEST
UI COMPONENT SUITE
FOR BUILDING
POWERFUL APPS

4.6 out of 5 stars



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Apple IBM SIEMENS intel VISA

Syncfusion

Chapter 5 Deploy the GalaxyHotel Website

This chapter explains how to build and deploy the GalaxyHotel website. The website is built using ASP.NET Core and uses the libraries released in the previous chapter on the NuGet feed. The source code is hosted on a separate repository than the libraries' project, and the branching workflow used is feature branching.

The application is hosted on Azure App Service on multiple instances: Dev, QA, and Prod. For this reason, the deployment uses the **multistage pipeline** approach, which is based on YAML, and promotes new features to different staging environments. It is also possible to control the flow using triggers, conditions, and approvals among stages.

Note: To complete the examples in this chapter, you need an active Azure subscription. You can start for free with limited credit on this page.

Build the GalaxyHotel website solution

To build the application, you have to create a new build pipeline. The steps are similar to the build process designed in the previous chapter. In the **Pipeline** section, click **New Pipelines**, select the **Azure Repos Git (YAML)** option, and provide the repository. From the list of templates, select **ASP.NET Core**, which opens the editor where you can remove the build task already present. To compile the application correctly, follow the next steps.

Using the assistant, add a **`dotnet`** task as the first task of the pipeline to restore the solution packages. In the **Command** option, select **restore**. Then, in the **Use packages from this Azure Artifacts/TFS feed** option, select the **GalaxyHotelLibraries** feed, as seen in Figure 5.1, to restore all the libraries (NuGet.org and yours).

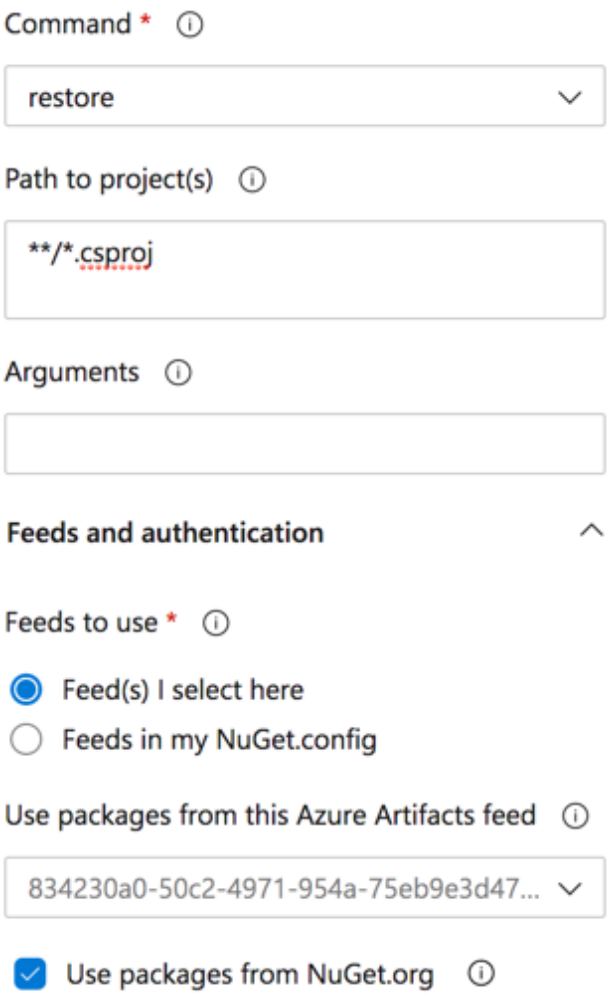


Figure 5.1: Restore packages from an Azure Artifacts feed

Then, add three more **`dotnet`** tasks: build, test, and publish. These are configured in the following Code Listing (5.1).

Code Listing 5.1

```
trigger:
- master

pool:
  vmImage: 'windows-latest'

variables:
  buildConfiguration: 'Release'

steps:
```

```
- task: DotNetCoreCLI@2
  displayName: Restore packages
  inputs:
    command: 'restore'
    feedsToUse: 'select'
    vstsFeed: '834230a0-50c2-4971-954a-75eb9e3d47dd/dcb14a3c-e6ab-48df-
ac52-3ae79a5055f7'

- task: DotNetCoreCLI@2
  displayName: Build the website
  inputs:
    command: 'build'
    projects: '**/*.csproj'

- task: DotNetCoreCLI@2
  displayName: Run tests
  inputs:
    command: 'test'
    projects: 'tests/**/*.csproj;!tests/**/*UITests*.csproj'

- task: DotNetCoreCLI@2
  displayName: Publish the website
  inputs:
    command: 'publish'
    publishWebProjects: true
    arguments: '--output $(Build.ArtifactStagingDirectory)'
```

Save the pipeline in a proper branch, run it, and check that everything works fine.

YAML template

Sometimes it is useful to have a snippet of YAML pipeline ready to use. Templates allow you to create reusable YAML code that can be referenced into a pipeline. More specifically, Azure Pipelines has two types of templates:

- **`template reference`** (also named **includes**): Includes external YAML content in the pipeline, easing the reusability.
- **`extends`**: Controls the flow in a pipeline, permitting only specific tasks or operations.

The syntax for both templates is shown in the following code listing.

Code Listing 5.2

```
# includes
steps: [ templateReference ]
parameters:
  key: value

# extends
extends:
  template: [ templateReference ]
  parameters:
    key: value
```

Code Listing 5.3 shows how to secure your pipeline using **`extends`**, allowing any task except the script one.

Code Listing 5.3

```
# template.yml
parameters:
- name: pipelineSteps
  type: stepList
  default: []
steps:
- ${{ each pipelineStep in parameters.pipelineSteps }}:
  - ${{ each pair in pipelineStep }}:
    ${{ if ne(pair.key, 'script') }}:
      ${{ pair.key }}: ${{ pair.value }}

# azure-pipelines.yml
extends:
  template: template.yml
  parameters:
    pipelineSteps:
    - task: FirstTask@1
    - script: this task will be skipped!
    - task: SecondTask@2
```

Tip: More information about securing a pipeline using `extends` is available here.

If you want to reuse the logic to build, test, and publish a web application in many pipelines, the GalaxyHotel website pipeline can be modified using a template reference. In large companies, templates are usually stored in a dedicated repository to version all templates. For simplicity, in this book, templates are stored in the application repository and eventually duplicated where they are needed.

Code Listing 5.4 shows the template for the build process.

Code Listing 5.4

```
# build-website.yml
parameters:
- name: librariesFeed
  displayName: 'Restore from Libraries NuGet feed'
  type: boolean
  default: true
- name: enablePublish
  displayName: 'Enable publish'
  type: boolean
  default: true
- name: publishArgs
  displayName: 'Publish Arguments'
  type: string
  default: ''

steps:
- ${{ if eq(parameters.librariesFeed, true) }}:
  - task: DotNetCoreCLI@2
    displayName: Restore packages
    inputs:
      command: 'restore'
      feedsToUse: 'select'
      vstsFeed: '834230a0-50c2-4971-954a-75eb9e3d47dd/dcb14a3c-e6ab-48df-
ac52-3ae79a5055f7'

- task: DotNetCoreCLI@2
  displayName: Build the website
  inputs:
      command: 'build'
```

```
      projects: '**/*.csproj'

- task: DotNetCoreCLI@2
  displayName: Run tests
  inputs:
      command: 'test'
      projects: 'tests/**/*.csproj;!tests/**/*UITests*.csproj'

- ${{ if eq(parameters.enablePublish, true) }}:
  - task: DotNetCoreCLI@2
    displayName: Publish the website
    inputs:
        command: 'publish'
        publishWebProjects: true
        arguments: ${{ parameters.publishArgs }}
```

The **parameters** section allows the template to receive values from the pipeline in which it is used. In this scenario, the template can be used when you want to build and test the application, or when you want to publish it.

Finally, Code Listing 5.5 shows how to reference and pass parameters to the template.

Code Listing 5.5

```
# azure-pipelines.yml
trigger:
- master

pool:
  vmImage: 'windows-latest'

variables:
  buildConfiguration: 'Release'

steps:
- template: build-website.yml
  parameters:
    publishArgs: '--output $(Build.ArtifactStagingDirectory)'
```

Run the build and check that everything works correctly using the template.

Deploy to the dev environment

At the beginning of this chapter, I mentioned that a multistage pipeline helps deploy an application to multiple environments. Although this feature has only recently been included in Azure Pipelines, it is becoming the primary choice when you want to deploy an application to various stages. It even includes features like approvals, checks, deployment strategies, and detailed history information for auditing. Plus, you can write the entire release pipeline in YAML, containing both build and deploy for specific environments. The pipeline can be versioned in the same repository as your application, and thanks to templates, you can also create reusable components for your deployments.

To deploy the artifact created in the previous section using the multistage pipeline, the build process in **`azure-pipeline.yml`** must change. Indeed, instead of defining steps, you should define **stages**. A stage is just a group of jobs that have the same target, such as **`Build`** or **`Deploy`**, and has the following syntax:

Code Listing 5.6

```
stages:
- stage: Build
  jobs:
  - job: BuildJob
    steps:
    - script: echo this is the build stage.
- stage: Deploy
  jobs:
  - job: DeployJob
    steps:
    - script: echo this is the deploy stage.
```

You can check out the full stage syntax here. Azure Pipelines also provides a special type of job named Deployment Job, which can be used when deploying an artifact. It offers a few benefits, such as:

- **Deployment history:** A detailed history of the pipelines for auditing.
- **Deployment strategy**: The way the artifact is deployed. At the time of writing, the available strategies are runOnce, rolling, and canary.

Deployment strategies are useful for facilitating the deployment phase, exposing the following lifecycle hooks: **`predeploy`**, **`deploy`**, **`routeTraffic`**, **`postRouteTraffic`**, **`on failure`**, and **`on success`**. The different strategies work as follows:

- **`runOnce`** executes hooks once.
- **`rolling`** updates the software version, replacing the current version with a newer one on a rolling set (a set of machines). It performs hooks on each machine where you can check if any deployment has failed, stopping deployment and starting a rollback phase using **`failure`** or **`success`**.
- **`canary`** lets you gradually deploy your changes to a small subset of resources based on the **`increment`** value. You can only use the canary deployment strategy for Kubernetes resources.

An example of deployment strategy syntax is available here.

Back to the GalaxyHotel project: the artifact will be deployed on an Azure App Service, and the YAML in Code Listing 5.3 should be modified to introduce stages. But before seeing the code, a target environment must be created.

Environments

Recently, Azure Pipelines introduced the concept of **environment**, which is a set of resources to use as a target for your deployment. At the time of writing, only two resources are available: Kubernetes clusters and virtual machines. An environment can be defined using the appropriate entry in the sidebar (Figure 5.2) and the **New Environment** button.

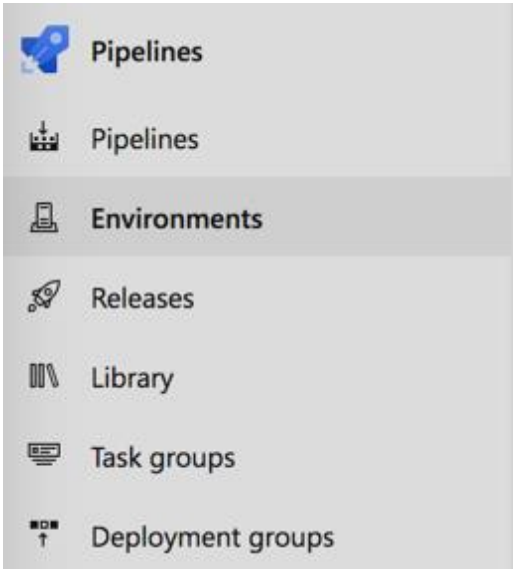


Figure 5.2: Environments section

A few advantages of environments are:

- Detailed history including information from multiple pipelines for related deployment (such as in microservices scenarios).
- Commits and work items deployed.
- Users and pipelines permissions to access a specific environment.
- Approvals and checks.
- Health information about resources.

To create a new environment, click **New Environment** and specify the following information (Figure 5.3):

- Name
- Description
- Resource Type (None, Kubernetes, or Virtual machines)

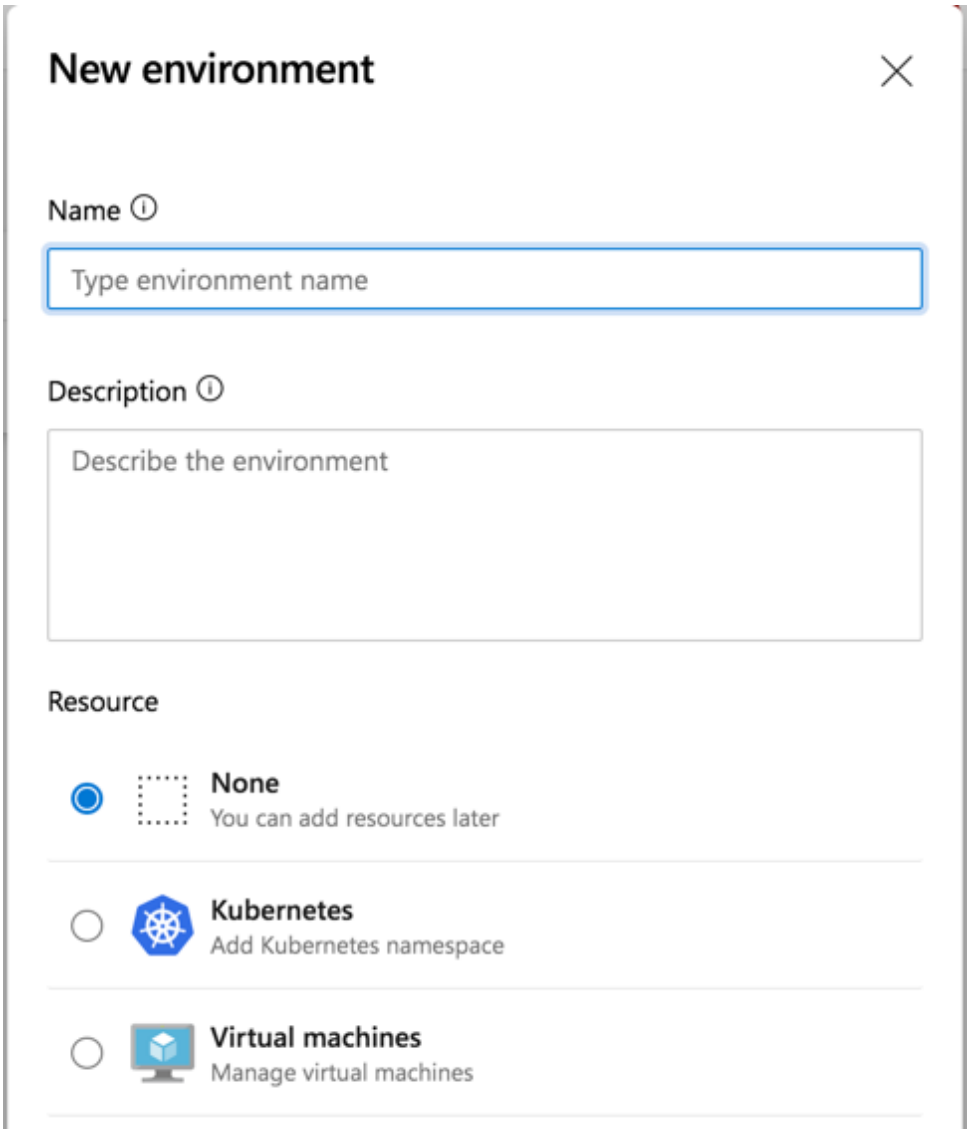


Figure 5.3: New environment creation popup

A fascinating concept is the **none** value: an environment can have no resources associated, but you can leverage the advantages listed earlier.

The GalaxyHotel application is deployed to three different environments: **GalaxyHotel-Dev**, **GalaxyHotel-QA**, and **GalaxyHotel-Prod**. These are not linked to a resource, and you can create them before changing the YAML pipeline (Figure 5.4).

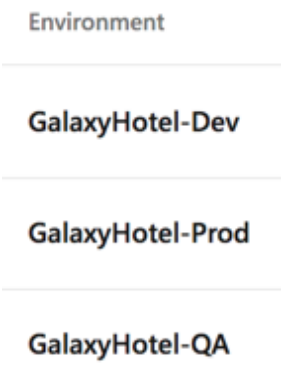


Figure 5.4: GalaxyHotel environments

Note: If an environment with a specified name is not found when a pipeline is run, the environment is automatically created.

It's time to update the YAML code.

Using stages in a YAML pipeline

The following shows a modified version of the YAML pipeline introduced in Code Listing 5.4, now with stages and environments.

Code Listing 5.7

```
# azure-pipelines.yml
trigger:
- master

pool:
  vmImage: 'windows-latest'

variables:
  buildConfiguration: 'Release'

stages:
- stage: build
  displayName: Build
  jobs:
  - job: Build
    steps:
    - template: build-website.yml
```

```
      parameters:
        publishArgs: '--output $(Build.ArtifactStagingDirectory)'
    - task: PublishPipelineArtifact@1
      inputs:
        targetPath: '$(Build.ArtifactStagingDirectory)'
        artifact: 'web'
        publishLocation: 'pipeline'

- stage: deployDev
  displayName: Deploy to Dev
  jobs:
  - deployment: deployDev
    displayName: Deploy To Dev
    environment: 'GalaxyHotel-Dev'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: AzureRmWebAppDeployment@4
            inputs:
              ConnectionType: 'AzureRM'
              azureSubscription: 'Microsoft Azure Sponsorship'
              appType: 'webApp'
              WebAppName: 'galaxyhotel-dev'
              packageForLinux: '$(Pipeline.Workspace)/web/**/*.zip'
```

There are two stages in the pipeline: **`build`** and **`deployDev`**. The build section is similar to Code Listing 5.3, but now it is included in a stage. The deployment part is a deployment job with a **`runOnce`** strategy, and it is linked to the environment created earlier. The artifact is retrieved using the **`Pipeline.Workspace`** variable, and it is deployed using an Azure App Service deploy (**`AzureRmWebAppDeployment@4`**) task on an Azure App Service.

To create a web app in an Azure App Service in a previously activated Azure subscription, you can follow this guide. If the account used for the Azure subscription is the same in Azure DevOps, there is a good chance that the subscription is already listed in the Azure App Service deploy task (Figure 5.5).

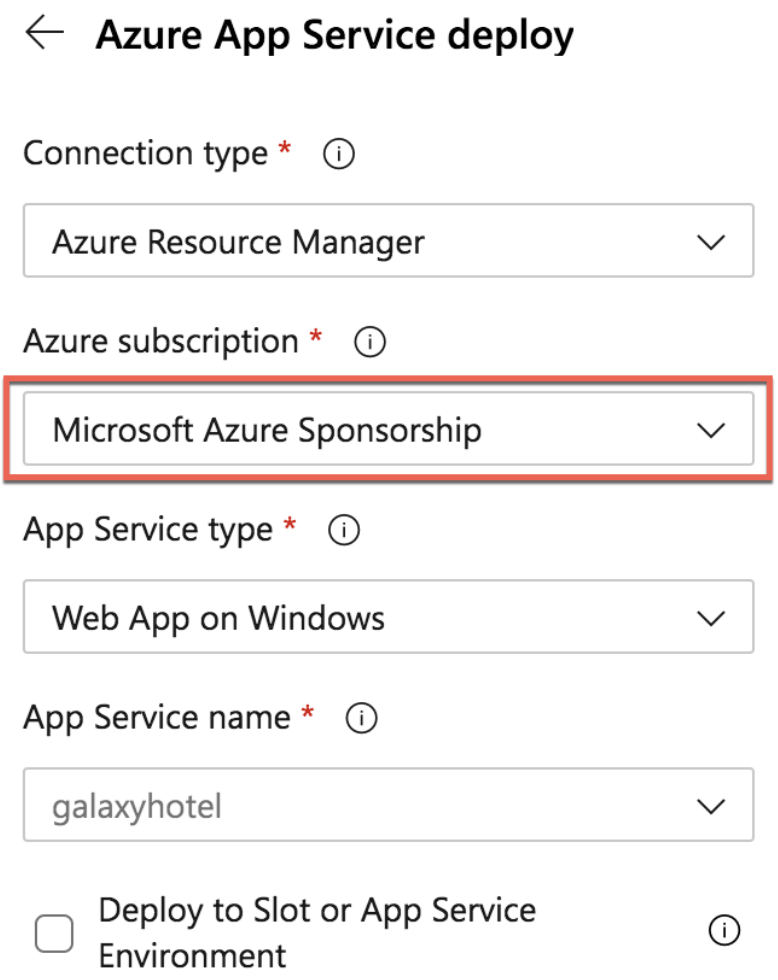


Figure 5.5: Azure App Service deploy settings

If it's not visible, you should use these steps to create a service connection, which is the way Azure DevOps connects to external services such as Microsoft Azure. Once connected, you can use it in your pipeline.

After that, you can save and run the pipeline. The first difference is visible in the Run Pipeline popup page, where the **Advanced options** contain two more settings: **Stage to run** and **Resource**. These help you override default settings at runtime. Also, when the pipeline starts, the summary page is different from the one seen in Figure 2.6: it shows both stages and jobs, as you can see in Figure 5.6.

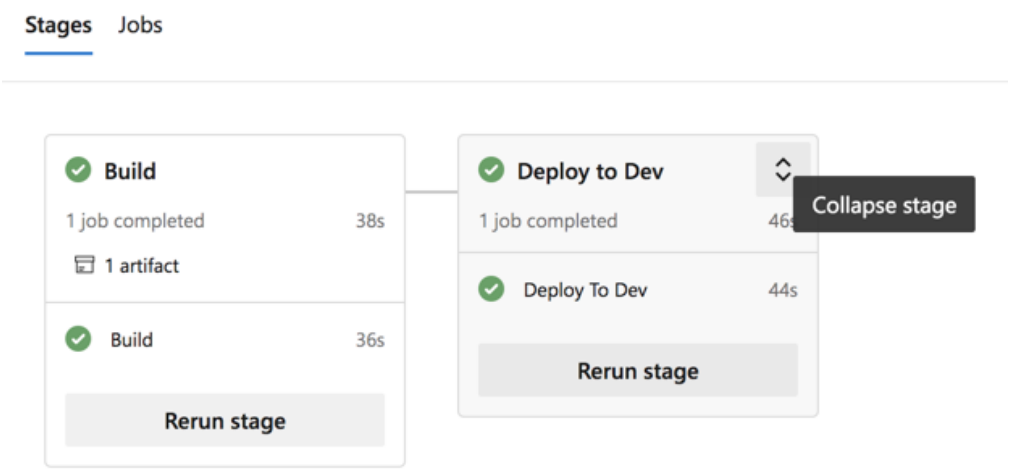


Figure 5.6: Multistage pipeline summary page

Each box in the figure is related to a stage, and by clicking **Expand stage** or **Collapse stage**, you can show or hide information regarding the artifact (build stage) or all the jobs executed. Clicking the boxes opens the jobs detail page, where all steps and logs are collected (Figure 5.7).

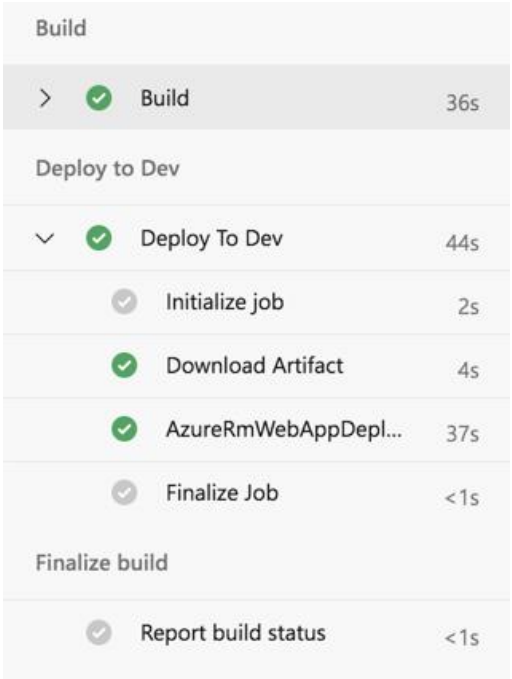


Figure 5.7: Multistage pipeline jobs details

As explained before, environments provide a set of information regarding pipeline execution. If you open the section, it shows details about the last execution (Figure 5.8).

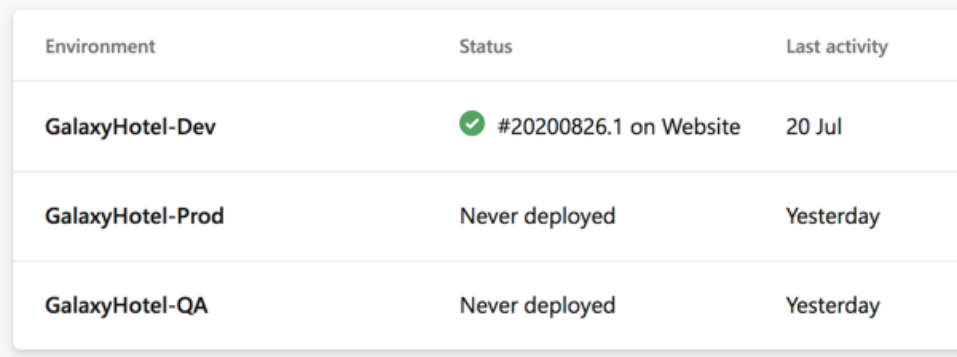


Figure 5.8: Pipeline execution in Environment page

Entering the environment details gives you the list of deployments (Figure 5.9). By clicking on each deployment, you can get insight into jobs, changes (or commits), and related work items (Figure 5.10).

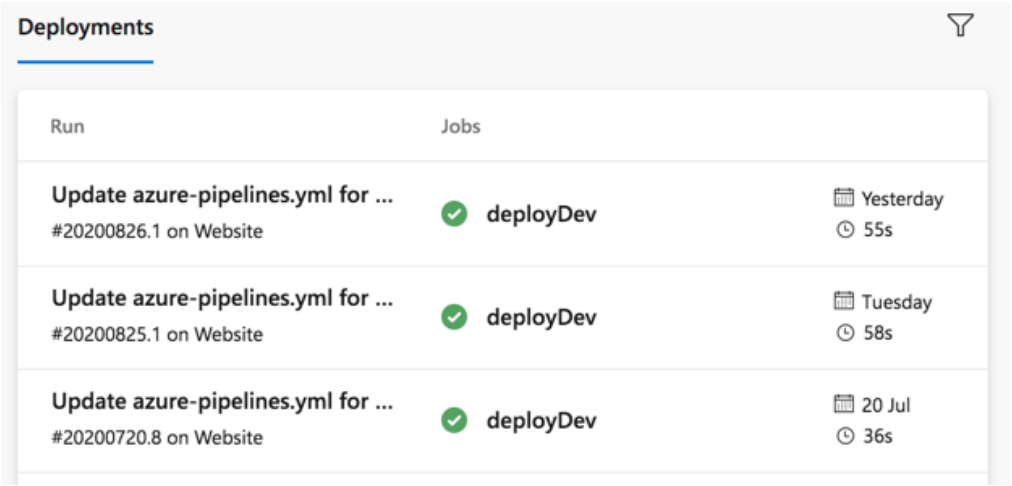


Figure 5.9: Deployments of an environment

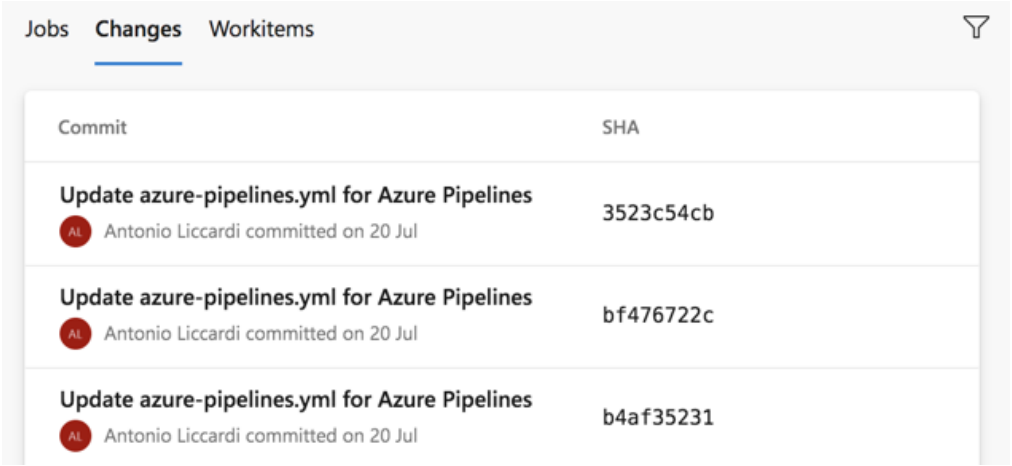


Figure 5.10: Changes in an environment deployment

Deploy to the QA environment

When ready, the artifact can be deployed to the QA environment with the same development process. In contrast to this environment, QA usually involves a manual or automatic testing phase. The manual testing can be done by testers using test plans and test cases. In this context, **Azure Test Plans** comes to our aid with a central place to trace everything (an example can be found here).

Automatic testing has lots of advantages because tests are repeatable and (often) faster than humans. But they have to be planned and implemented before integrating them in a pipeline, strongly depending on the type of tests to execute (UI, contract, and so on).

In either case, when deploying to QA, hosting environments should be prepared. For example, databases need to be filled with sample data or resources to be cleaned. Also, in this case, Azure Pipelines comes in handy thanks to a collection of tasks to achieve this scope.

Publish SQL script and UI tests artifacts

The GalaxyHotel solution includes a SQL script in the db folder to seed the database to execute the test in the QA environment. Moreover, UI tests written using Selenium are also available in a project of the same solution and must be copied on a VM provisioned to execute these tests. To make these components available in the QA stage, you should add in the **`build`** stage of the pipeline two more artifacts, as in the following code listing.

Code Listing 5.8

```
# azure-pipelines.yml
...
- job: Build
    steps:
    - template: build-website.yml
      parameters:
        publishArgs: '--output $(Build.ArtifactStagingDirectory)'
    - task: PublishPipelineArtifact@1
      displayName: Publish UI Tests artifacts
      inputs:
        targetPath:
'$(Build.Repository.LocalPath)/tests/GalaxyHotel.UITests'
        artifact: 'UITests'
        publishLocation: 'pipeline'
    - task: PublishPipelineArtifact@1
      displayName: Publish db artifacts
      inputs:
        targetPath: '$(Build.Repository.LocalPath)/db'
```

```
        artifact: 'db'
        publishLocation: 'pipeline'
    - task: PublishPipelineArtifact@1
      inputs:
        targetPath: '$(Build.ArtifactStagingDirectory)'
        artifact: 'web'
        publishLocation: 'pipeline'
...
```

After the **`build`** stage execution, there should be three artifacts (Figure 5.11).

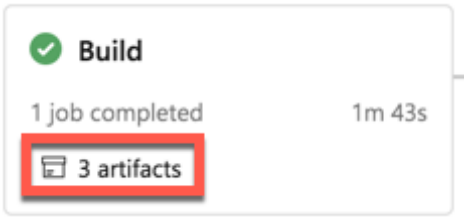


Figure 5.11: Multiple artifacts as build output

Prepare the GalaxyHotel-QA environment

Selenium is a robust framework for creating repeatable UI tests. It has drivers related to most common browsers and even a tool, the Selenium IDE, to record tests using a user interface.

The GalaxyHotel.UITests project contains UI tests based on Selenium to check GalaxyHotel website features. Thanks to the **Selenium.WebDriver** NuGet package, it is possible to reproduce a user's steps during their navigation on the website. It is straightforward to use, and the documentation provided is thorough. Code Listing 5.9 shows an example of a UI test written in C# using the Selenium driver.

Code Listing 5.9

```
[TestMethod]
[TestCategory("Chrome")]
public void BookARoomAndCheckForConfirmationTest()
{
    webDriver.Navigate().GoToUrl(baseUrl);
    webDriver.FindElement(By.LinkText("Book")).Click();

    var bookingConfirmationText = webDriver
            .FindElement(By.XPath("//h1")).Text;

    Assert.AreEqual("Room booked!", bookingConfirmationText);
}
```

In this example, a browser is opened to navigate to the GalaxyHotel page, find the Book button, click it, and then check if the booking confirmation page is opened.

During the QA stage, these tests have a few requirements that the Azure Pipelines agents cannot satisfy: they need to be executed on an agent that has a Chrome browser and the **`dotnet`** framework installed.

> ***Note: There are a few experiments done by the dev community using Docker and Chrome in headless mode, such as the one found here.***

To run the UI tests, you should add a virtual machine with the requirements listed earlier to the GalaxyHotel-QA environment. Go to the environment section in Azure Pipelines, click **Add Resource**, and select **Virtual Machine** as the resource type (Figure 5.12).

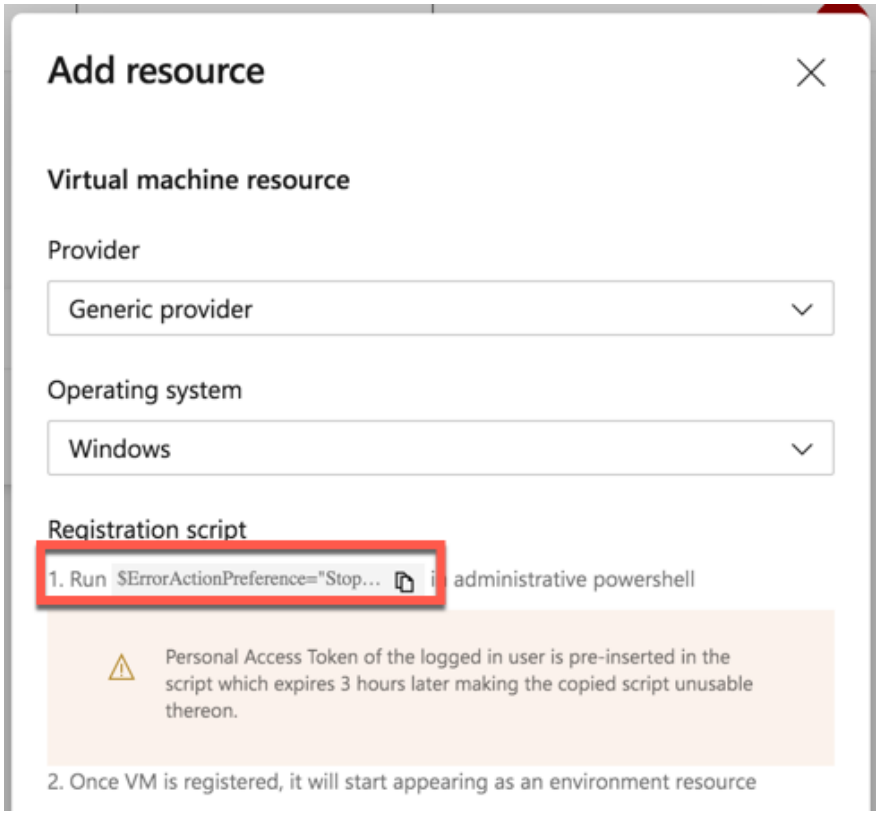


Figure 5.12: Adding a virtual machine

The **Add resource** popup contains the registration script for the provider and the operating system chosen. This script should be executed on the target virtual machine using an administrative PowerShell console, which asks for information such as the user account for the agent service and a list of tags (such as **qa**). After the execution, the environment page shows the linked resource (Figure 5.13).

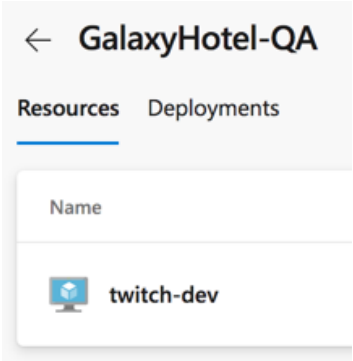


Figure 5.13: A resource linked to an environment

The environment is ready to execute automated tests.

Deploy to the QA environment and run tests

The QA stage is a bit different from the dev one. The YAML pipeline in Code Listing 5.10 is organized to:

1. Clean and fill the SQL Azure database.
2. Deploy the website on the Azure App Service designed for QA.
3. Run the automated UI tests from the VM linked to the environment.

Code Listing 5.10

```
- stage: deployQA
  displayName: Deploy to QA
  jobs:
  - deployment: deployQAWebsite
    displayName: Deploy Website to QA
    environment: 'GalaxyHotel-QA'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: SqlAzureDacpacDeployment@1
            displayName: Prepare the database
            inputs:
              azureSubscription: 'Microsoft Azure Sponsorship'
              AuthenticationType: 'server'
              ServerName: 'succinctly.database.windows.net'
              DatabaseName: 'galaxyhotel-qa'
              SqlUsername: '$(sqlAdmin)'
```

```
              SqlPassword: '$(sqlPassword)'
              deployType: 'SqlTask'
              SqlFile:
'$(Pipeline.Workspace)/db/SeedGalaxyHotelDbScript.sql'
              IpDetectionMethod: 'AutoDetect'
          - task: AzureRmWebAppDeployment@4
            displayName: Deploy website
            inputs:
              ConnectionType: 'AzureRM'
              azureSubscription: 'Microsoft Azure Sponsorship'
              appType: 'webApp'
              WebAppName: 'galaxyhotel-qa'
              packageForLinux: '$(Pipeline.Workspace)/web/**/*.zip'
  - deployment: runQATests
    displayName: Run QA Tests
    dependsOn: deployQAWebsite
    environment:
      name: 'GalaxyHotel-QA'
      resourceType: VirtualMachine
      tags: qa
    strategy:
      runOnce:
        deploy:
          steps:
          - task: PowerShell@2
            displayName: Override TestRunParameters
            inputs:
              targetType: 'inline'
              script: |
                [xml]$doc = Get-Content
$(Pipeline.Workspace)/UITests/GalaxyHotelTests.runsettings
$doc.RunSettings.TestRunParameters.ChildNodes.Item(0).value =
'$(qaBaseUrl)'
$doc.Save("$(Pipeline.Workspace)/UITests/GalaxyHotelTests.runsettings")
          - task: DotNetCoreCLI@2
            displayName: Execute UI Tests with Selenium
            inputs:
              command: 'test'
              projects: '$(Pipeline.Workspace)/UITests/**/*.csproj'
```

```
            arguments: '--settings
$(Pipeline.Workspace)/UITests/GalaxyHotelTests.runsettings'
```

The stage has two deployment jobs: one to prepare the database and the website, and one to execute tests on the VM, which depends on the first one (**dependsOn**).

In the **Prepare the database** step, the **db** artifact published earlier is used to retrieve the SQL script to fill the database. The script is executed using a **SqlAzureDacpacDeployment@1** task, which requires the database info and the location of the script. Because the username and the password are sensitive information, you should not include them in a pipeline, especially if it is versioned into a repository (this is one of the main principles of the DevSecOps culture).

For this reason, the pipeline has two variables: **$(sqlAdmin)** and **$(sqlPassword)**, created using the **Variables** button on top of the pipeline editor (Figure 5.14). Variables can be marked as **secret** to avoid security problems during pipeline management or execution.

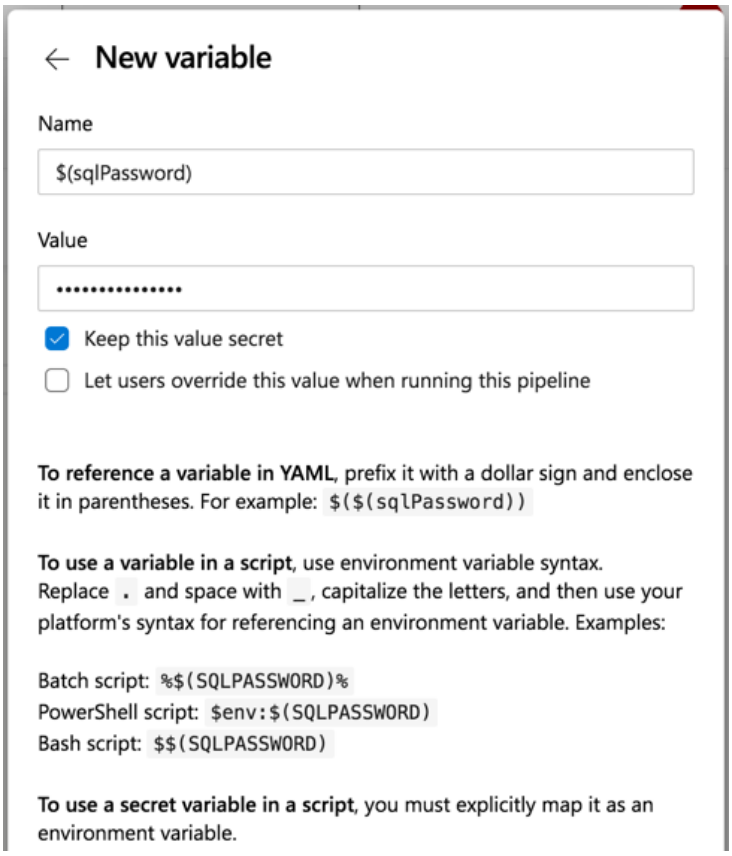


Figure 5.14: Creating a secret variable for the pipeline

The **runQATests** deployment job is linked to the GalaxyHotel-QA environment and uses the VM prepared for the tests by specifying:

- The **resourceType** parameter, set as **VirtualMachine**.
- The **tags** parameter, set as **qa**.

With these parameters in place, all the steps in the job are executed on the VM. During the execution, the **`UITests`** artifact is copied to the VM in the folder assigned to the agent in the provisioning phase (the PowerShell script executed earlier). In my case, the agent folder is located at C:\azagent\A4, and it has a few subfolders, such as _work, which contains the artifact during the pipeline execution (Figure 5.15). These folders have different scopes: for example, **`a`** is the **`Build.ArtifactStagingDirectory`**, and **`s`** is the **`Build.SourcesDirectory`**.

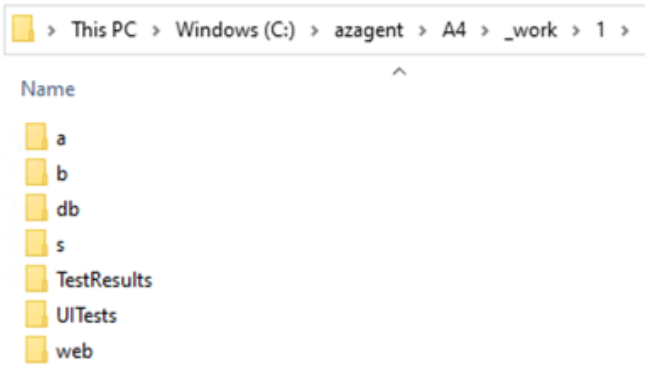


Figure 5.15: An Azure DevOps agent folder

In this folder, you can also see all three artifacts published by the build steps. If you don't want to download them all, you can opt to add in the YAML a **`download: none`** step, and then use a **`Download Pipeline Artifact`** task to retrieve the artifact for the current stage (Code Listing 5.11).

Code Listing 5.11

```
# azure-pipelines.yml
...
steps:
      - download: none
      - task: DownloadPipelineArtifact@2
        inputs:
          buildType: 'current'
          artifactName: 'UITests'
          targetPath: '$(Pipeline.Workspace)'
      - task: PowerShell@2
...
```

Now, there are three steps in the **`deployQA`** stage:

1. **`Download UITests artifact`**, which was explained earlier.
2. **`Override TestRunParameters`**, which is a PowerShell script that overrides the configuration in the **GalaxyHotelTests.runsettings** file. The **runsettings** file is a common way in the .NET Framework to pass parameters during test execution. The script will read the content of the file, and using an XPath expression, will replace the value of the **`baseUrl`**.

3. **`Execute UI Tests with Selenium`**, which takes the **GalaxyHotelTests.runsettings** file as input and executes UI tests using the **`dotnet test`** command (tests are written using Selenium).

During the pipeline execution, the **GalaxyHotelTests.runsettings** file will contain the **`baseUrl`** of the QA website (Code Listing 5.12).

Code Listing 5.12

```
<?xml version="1.0" encoding="utf-8"?>
<RunSettings>
    <TestRunParameters>
        <Parameter name="baseUrl" value="https://galaxyhotel-
qa.azurewebsites.net" />
    </TestRunParameters>
</RunSettings>
```

Save the pipeline to start it, and you can observe what happens in the detail page (Figure 5.16).

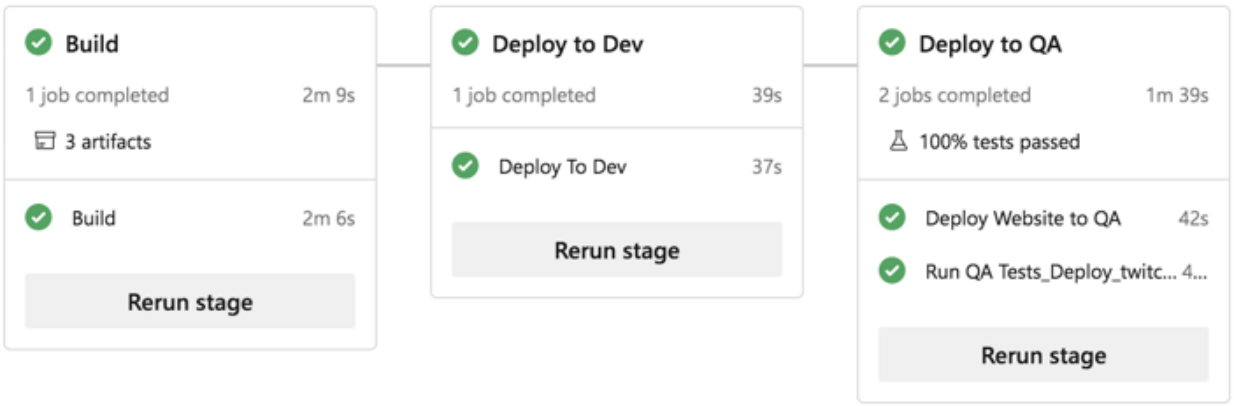


Figure 5.16: Run tests in the QA environment

The **Tests** tab on this page provides insight into the test run (Figure 5.17).

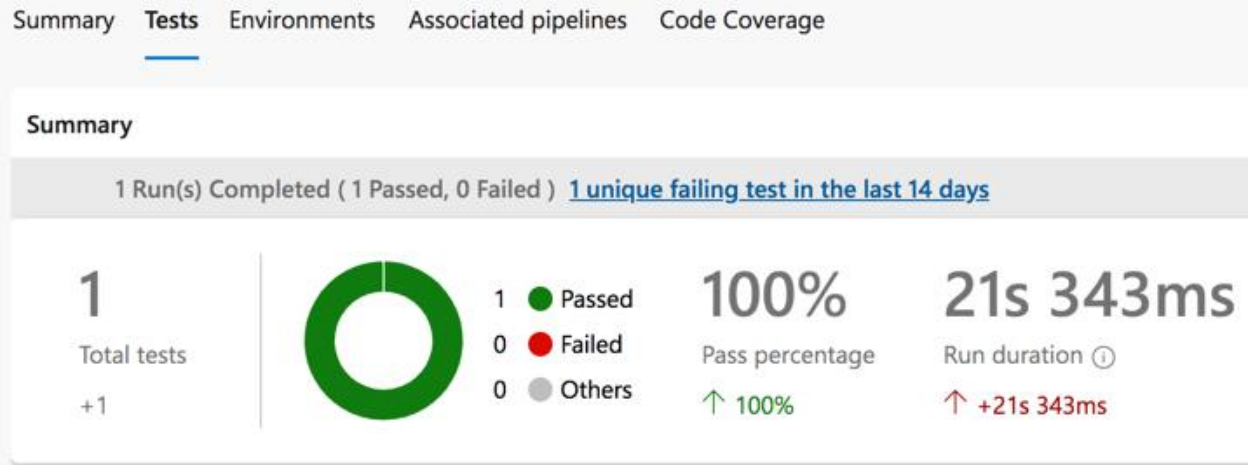


Figure 5.17: UI test run detail

Also, the logs of the test execution are available in the Jobs detail page.

Note: During the UI test execution on the virtual machine, a real instance of the Chrome browser opens and simulates the user steps.

The **Environments** tab shows pipeline details divided into stages (Figure 5.18).

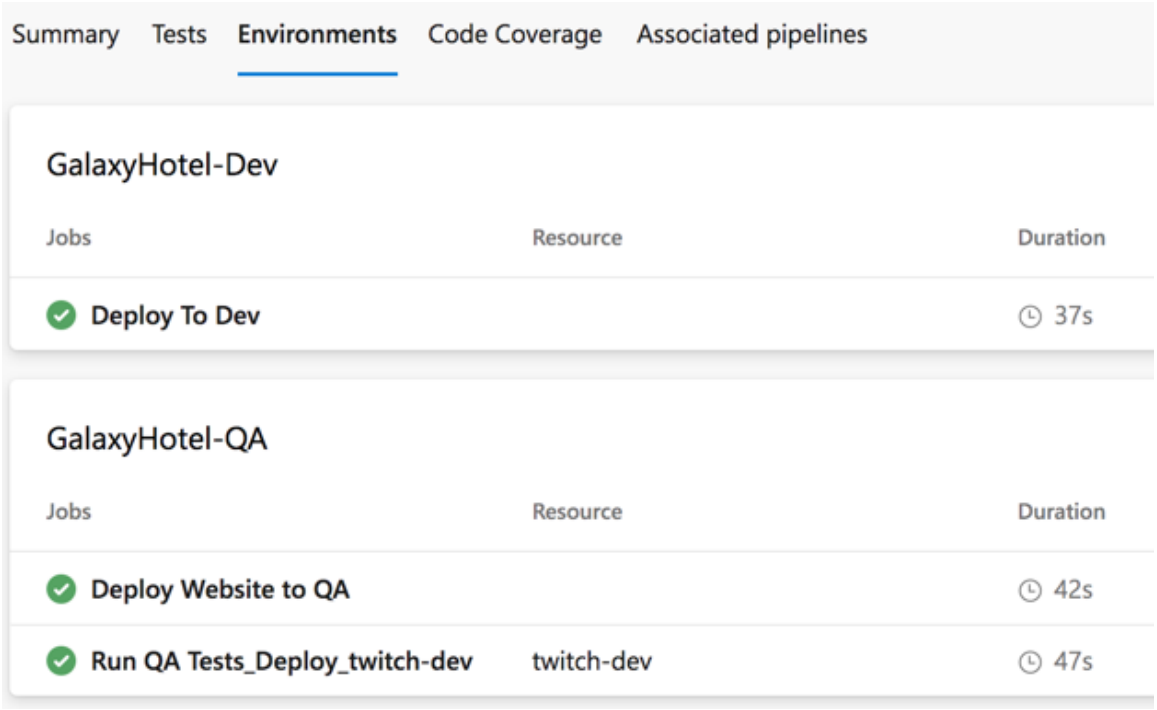


Figure 5.18: Environments impacted by the pipeline execution

It is interesting that in the QA stage, two different agents are used to deploy and test the software. Indeed, the concepts explored in this section can be applied to a lot of scenarios involving multiple deploys or actions in the same environment stage.

Enable approvals for QA environment

There is just one problem in the pipeline: deployment to the QA environment starts right after the dev one. But in the real world, QA deployment is usually done after developers have tested and approved their changes in the environment. For this reason, the deployment should be approved before being executed using **Approvals and checks**.

To enable them, open the GalaxyHotel-QA environment page, click the **More actions** button ( ), and select the **Approvals and checks** option. Click the plus button, and a popup window opens, which contains all the types of checks you can apply to an environment (Figure 5.19).

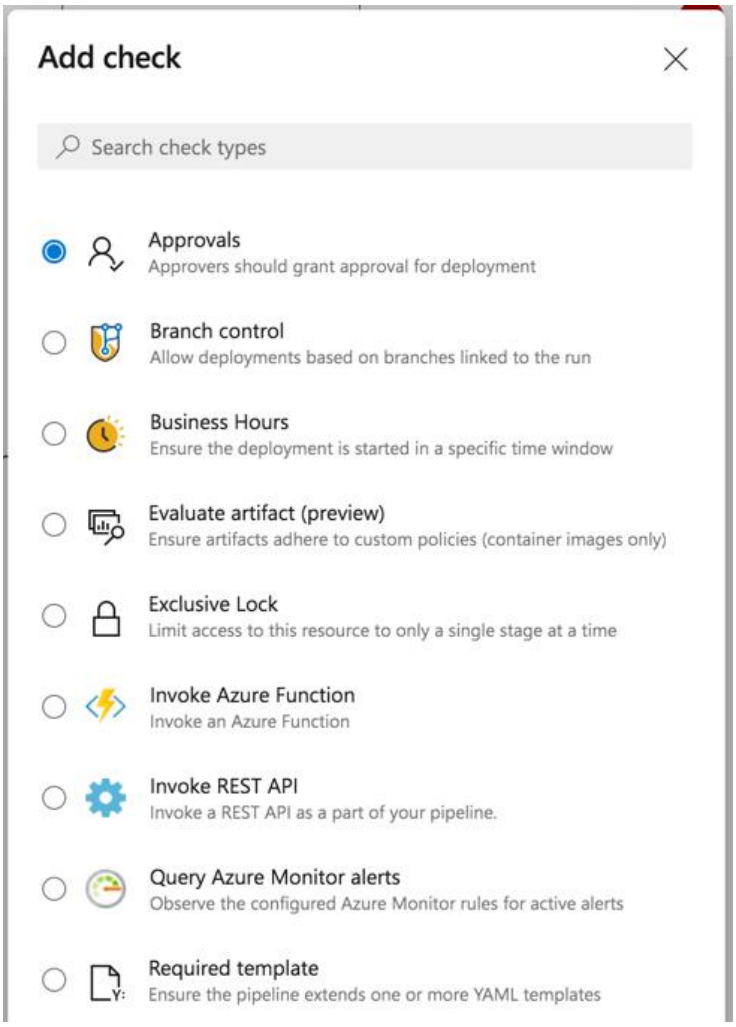


Figure 5.19: Approvals and checks list

The list provides a good number of items for the most common scenarios. For example, you can check for the branch related to the pipeline, the time of the deployment, or an alert on a monitoring system such as Azure Monitor. You can even invoke an Azure Function or a REST API. The result of the check controls the deployment started in a specific environment.

To add an approval for the GalaxyHotel-QA environment, select the first item and click **Next**. The popup lets you choose the approvers and set a few options as in Figure 5.20.

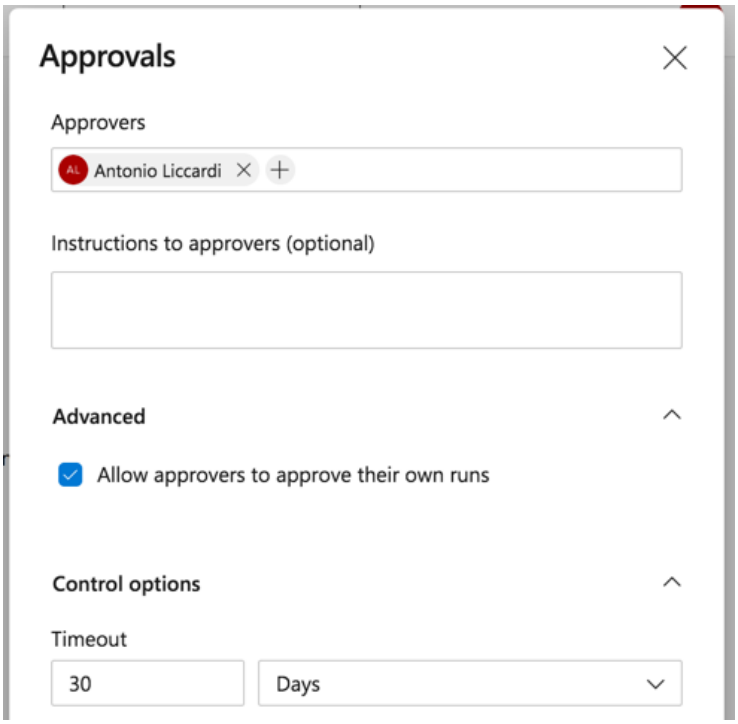


Figure 5.20: Adding an approver to an environment

Now, when you run the pipeline, an approver must approve the deployment. This is pointed out in the details page related to a run where the approval for a stage is marked with the Waiting label (Figure 5.21). To approve or reject the deployment, click **Review**, and then click **Approve** or **Reject** (Figure 5.22).

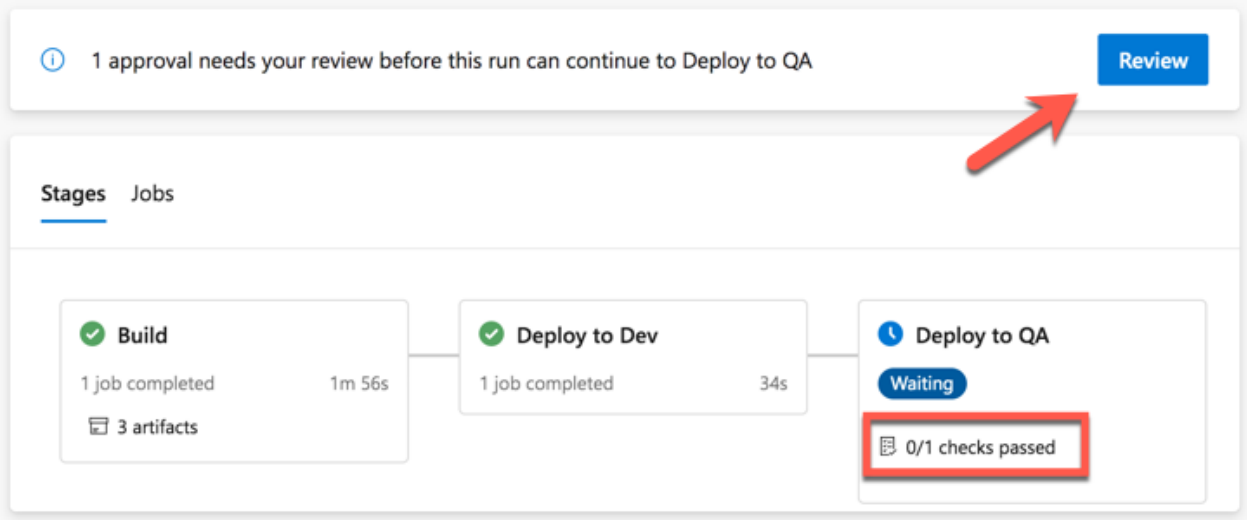


Figure 5.21: Deployment waiting for an environment approval

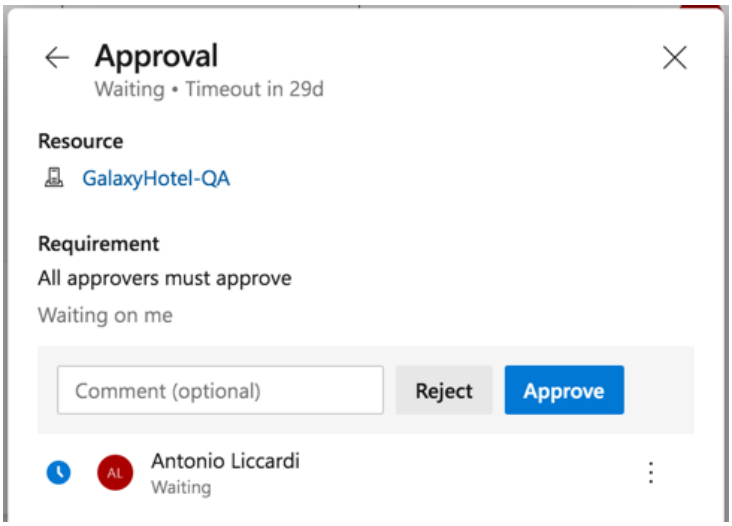


Figure 5.22: Environment approval page

Once approved, the deployment continues.

> ***Note: If you start multiple runs of the same pipeline containing an approval, the newer one will wait for the approval or rejection of the first one.***

Deploy to production environment

The last step to complete the pipeline is the deployment to the production environment. The YAML is similar to the **`deployDev`** stage. An approval is also set for the GalaxyHotel-Prod environment to decide when the software goes to production.

For the sake of completeness, Code Listing 5.13 presents the full pipeline for the GalaxyHotel website, including the production environment.

Code Listing 5.13

```
trigger:
- master

pool:
  vmImage: 'windows-latest'

variables:
  buildConfiguration: 'Release'

stages:
- stage: build
```

```
  displayName: Build
  jobs:
  - job: Build
    steps:
    - template: build-website.yml
      parameters:
        publishArgs: '--output $(Build.ArtifactStagingDirectory)'
    - task: PublishPipelineArtifact@1
      displayName: Publish UI Tests artifacts
      inputs:
        targetPath:
'$(Build.Repository.LocalPath)/tests/GalaxyHotel.UITests'
        artifact: 'UITests'
        publishLocation: 'pipeline'
    - task: PublishPipelineArtifact@1
      displayName: Publish db artifacts
      inputs:
        targetPath: '$(Build.Repository.LocalPath)/db'
        artifact: 'db'
        publishLocation: 'pipeline'
    - task: PublishPipelineArtifact@1
      displayName: Publish web artifacts
      inputs:
        targetPath: '$(Build.ArtifactStagingDirectory)'
        artifact: 'web'
        publishLocation: 'pipeline'

- stage: deployDev
  displayName: Deploy to Dev
  jobs:
  - deployment: deployDev
    displayName: Deploy To Dev
    environment: 'GalaxyHotel-Dev'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: AzureRmWebAppDeployment@4
            inputs:
              ConnectionType: 'AzureRM'
```

```
              azureSubscription: 'Microsoft Azure Sponsorship'
              appType: 'webApp'
              WebAppName: 'galaxyhotel-dev'
              packageForLinux: '$(Pipeline.Workspace)/web/**/*.zip'

- stage: deployQA
  displayName: Deploy to QA
  jobs:
  - deployment: deployQAWebsite
    displayName: Deploy Website to QA
    environment: 'GalaxyHotel-QA'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: SqlAzureDacpacDeployment@1
            displayName: Prepare the database
            inputs:
              azureSubscription: 'Microsoft Azure Sponsorship'
              AuthenticationType: 'server'
              ServerName: 'succinctly.database.windows.net'
              DatabaseName: 'galaxyhotel-qa'
              SqlUsername: '$(sqlAdmin)'
              SqlPassword: '$(sqlPassword)'
              deployType: 'SqlTask'
              SqlFile:
'$(Pipeline.Workspace)/db/SeedGalaxyHotelDbScript.sql'
              IpDetectionMethod: 'AutoDetect'
          - task: AzureRmWebAppDeployment@4
            displayName: Deploy website
            inputs:
              ConnectionType: 'AzureRM'
              azureSubscription: 'Microsoft Azure Sponsorship'
              appType: 'webApp'
              WebAppName: 'galaxyhotel-qa'
              packageForLinux: '$(Pipeline.Workspace)/web/**/*.zip'
  - deployment: runQATests
    displayName: Run QA Tests
    dependsOn: deployQAWebsite
    environment:
```

```
      name: 'GalaxyHotel-QA'
      resourceType: VirtualMachine
      tags: qa
    strategy:
      runOnce:
        deploy:
          steps:
          - download: none
          - task: DownloadPipelineArtifact@2
            displayName: Download UITests artifact
            inputs:
              buildType: 'current'
              artifactName: 'UITests'
              targetPath: '$(Pipeline.Workspace)/UITests'
          - task: PowerShell@2
            displayName: Override TestRunParameters
            inputs:
              targetType: 'inline'
              script: |
                [xml]$doc = Get-Content
$(Pipeline.Workspace)/UITests/GalaxyHotelTests.runsettings
                $doc.RunSettings.TestRunParameters.ChildNodes.Item(0).value
= '$(qaBaseUrl)'

$doc.Save("$(Pipeline.Workspace)/UITests/GalaxyHotelTests.runsettings")
          - task: DotNetCoreCLI@2
            displayName: Execute UI Tests with Selenium
            inputs:
              command: 'test'
              projects: '$(Pipeline.Workspace)/UITests/**/*.csproj'
              arguments: '--settings
$(Pipeline.Workspace)/UITests/GalaxyHotelTests.runsettings'

- stage: deployProd
  displayName: Deploy to Production
  jobs:
  - deployment: deployProd
    displayName: Deploy To Production
    environment: 'GalaxyHotel-Prod'
    strategy:
```

```
      runOnce:
        deploy:
          steps:
          - task: AzureRmWebAppDeployment@4
            inputs:
              ConnectionType: 'AzureRM'
              azureSubscription: 'Microsoft Azure Sponsorship'
              appType: 'webApp'
              WebAppName: 'galaxyhotel'
              packageForLinux: '$(Pipeline.Workspace)/web/**/*.zip'
```

Figure 5.23 shows a complete run from development to production:

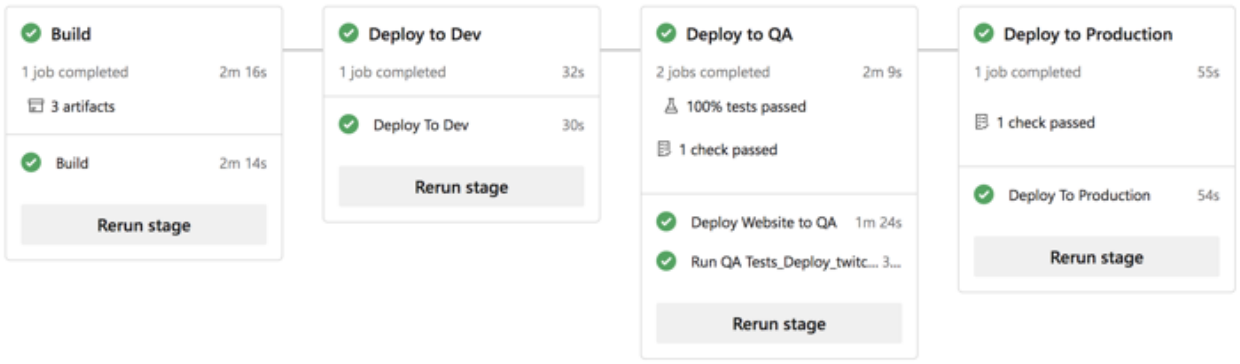


Figure 5.23: A full pipeline run, from dev to production

Figure 5.24 shows how it is pretty straightforward to understand which pipeline makes it to production on the runs list page.

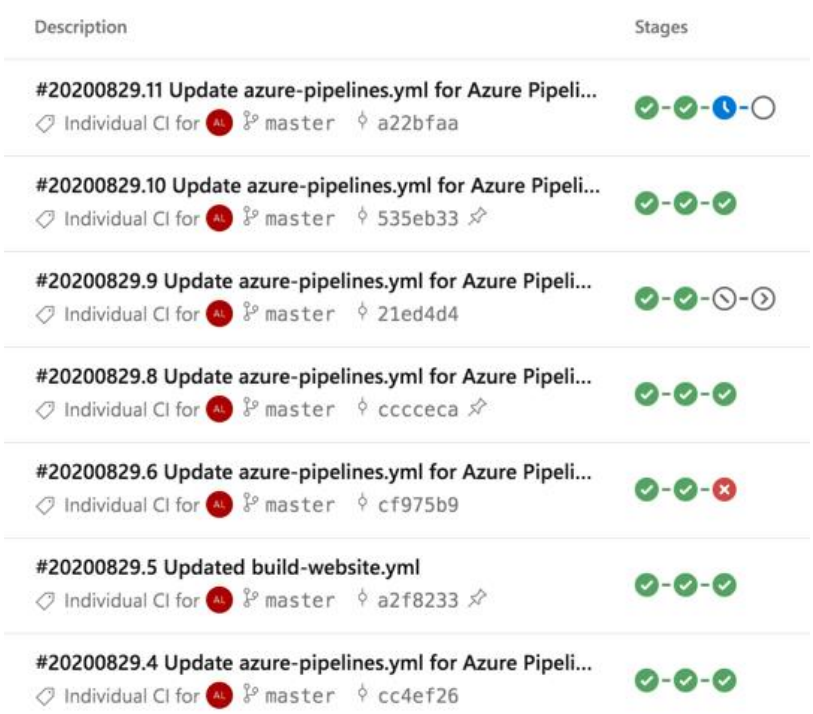


Figure 5.24: A list of runs including details regarding stages

Chapter 6 Deploy the GalaxyHotel HelpDesk

In this last chapter of the book, I'll explain how to build and deploy the GalaxyHotel HelpDesk project using the classic editor and the Release section of Azure Pipelines. Back in the day, Azure DevOps included just these options to build and deploy your projects; the YAML pipeline was introduced only in the last few years. Some features, such as the release part, are not implemented yet in the pipeline as a code approach.

The GalaxyHotel HelpDesk project is similar to the website (it is an ASP.NET Core web application). Because it should be used at the reception desk of the hotel, the application will be deployed on a simulated environment composed of:

- One virtual machine for the test environment.
- Two virtual machines for the production environment.

Both deployments are based on **deployment groups**.

Build the GalaxyHotel HelpDesk

To build the project, you can reproduce the same steps explained in Chapter 2. Open the **Pipelines** section, click **New Pipeline**, select **Use the classic editor** at the bottom of the page, and then set the repository properties as shown in Figure 6.1.

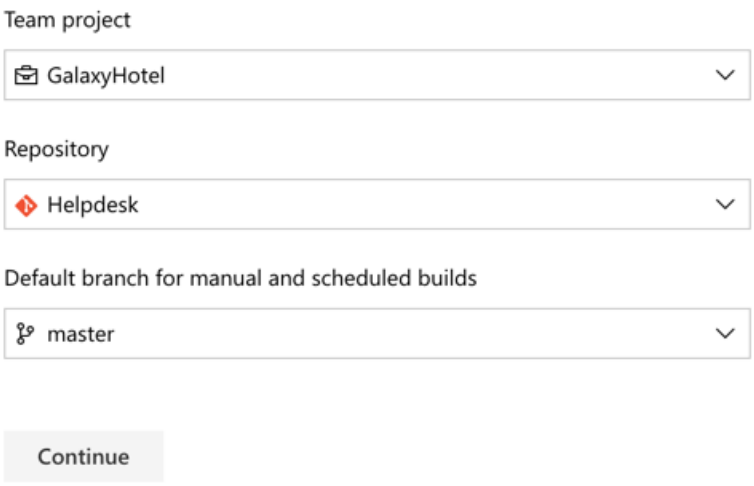


Figure 6.1: Pipeline repository settings for GalaxyHotel HelpDesk project

Azure Pipelines opens the build details page, showing a list of tasks that will be executed during the build run. Two settings need to be modified:

- The name of the pipeline should be changed to **HelpDesk**.

- The restore task should retrieve packages from the **GalaxyHotelLibraries** feed created in Chapter 4 (Figure 6.2).

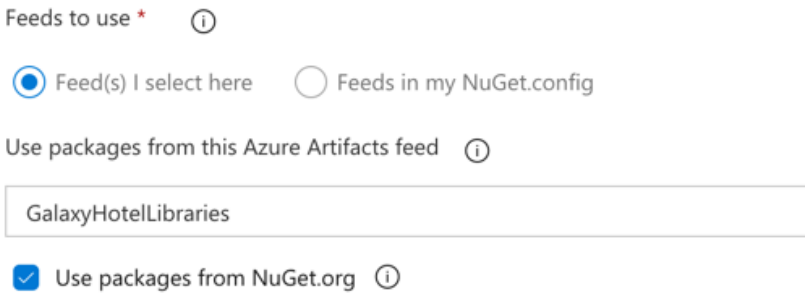


Figure 6.2: Feed settings for the build

You can click **Save & Queue** and check that the build executes correctly.

Build settings

The classic editor displays a list of tabs collecting all the settings related to the build.

Variables let you define custom variables to use during the build execution. A variable has a name and a value, and can be a secret (hiding its value). The syntax to get the value is the same as in the previous chapter: **`$(variable_name)`**. Variables can be grouped using the **Variable Groups** feature in the Library section and then linked to the build (Figure 6.3).

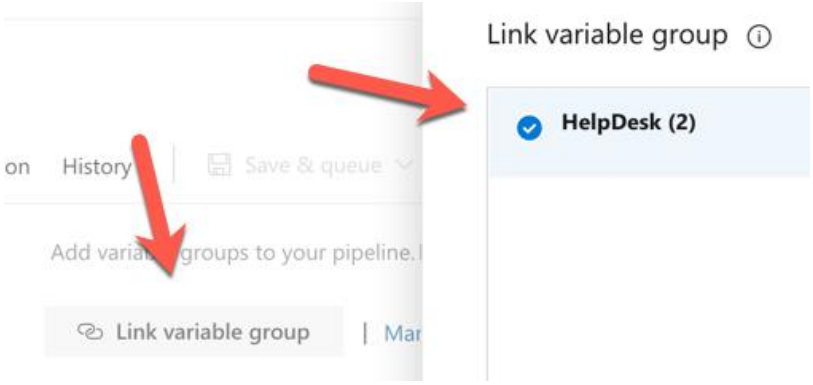


Figure 6.3: Link a variable group to the build

Triggers define when the build should start. The available options are:

- **Continuous integration**: The build starts right after a commit is applied to the repository.
- **Scheduled**: The build starts according to the date and time.
- **Build completion**: The build starts right after another build.

For each option, a **branch filter** can be applied, which helps when you want to start your build only for a specific branch. The branch name also supports wildcards, which are very useful for GitFlow (Figure 6.4).

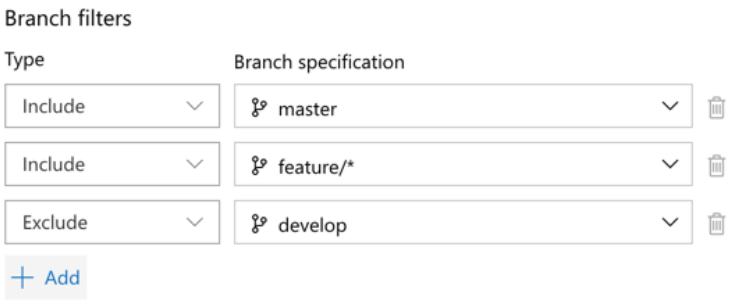


Figure 6.4: Branch filters

In the GalaxyHotel HelpDesk build, enable this option with a branch filter set only to **master**, and save the definition.

The continuous integration section also includes **path filters**, letting you specify which path should be observed or not to trigger the build (Figure 6.5).

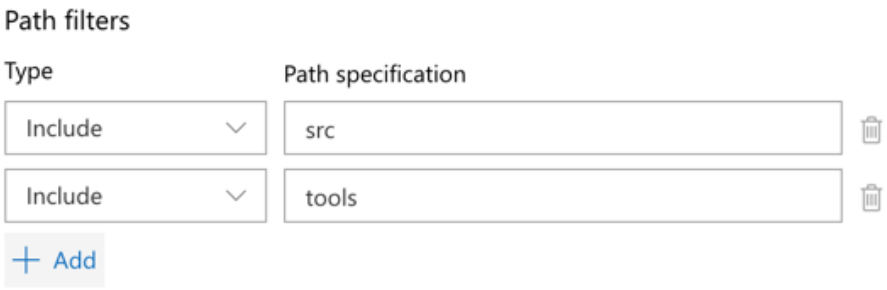


Figure 6.5: Path filters

The **options** tab provides lots of settings such as description, number format, link to a work item, enabling, status badge, scope, timeouts, and demands. Demands allow you to provide a required agent pool capability (more on this later). In the **retention** page, you can define how long artifacts, attachments, and build runs should be kept. Last, **history** lists all the changes applied to the build definition (including differences).

Agent pool and specification

In the Tasks tab, the build definition contains a section named **Agent job 1**, where the job tasks are listed. These are executed by an agent assigned to the build.

If you click the **Pipeline** bar on the right side, a section that includes details about the agent appears. As explained earlier, an agent is a process that executes the build, and an agent pool is a group of agents.

Azure Pipelines provides two types of agent pools:

- **Hosted** (Azure Pipelines): Provided and managed by Microsoft. They are free of charge until you reach 1,800 minutes of build run, or if you want multiple parallel jobs.

- **Private** (Default): Created and managed by the users. Usually, they are configured in on-premises infrastructure.

Hosted pools also have a list of **specifications** representing the operating system of the agent. Depending on your build requirements, these are the specifications available at the time of writing:

- macOS-10.14
- macOS-10.15
- ubuntu-16.04
- ubuntu-18.04
- ubuntu-20.04
- ws2017-win2016
- windows-2019

The full list of capabilities and included software related to each specification is available here. You can add new capabilities in the project settings page under the Agent Pools section.

Private pools are useful for on-premises scenarios using a self-hosted agent or when a hosted agent does not satisfy your requirements. Plus, a self-hosted agent does not have limitations.

Adding a new private agent is a relatively simple operation: in the **Agent Pools** section of your project settings, select **Default** > **New Agent**. A popup opens, giving you the instruction to download and install an agent on a machine (Figure 6.6).

Note: To install the agent, you must generate an authentication token, as explained here. Tokens allow external services to authenticate with Azure DevOps correctly. Each token has an associated list of permissions.

After the installation, the agent is listed in the Default pool page (Figure 6.7), where each agent can be updated (in case a new version is available) or deleted.

The agent detail page shows both jobs and capabilities (divided into user-defined and system). In this case, the capabilities are related to the software installed on the agent machine, as in Figure 6.8.

Capabilities play a fundamental role in Azure DevOps: they guide how an agent is chosen for a build. Indeed, the concept of **demands** explained earlier can be exploited to make sure that an agent has the requirements needed for a build.

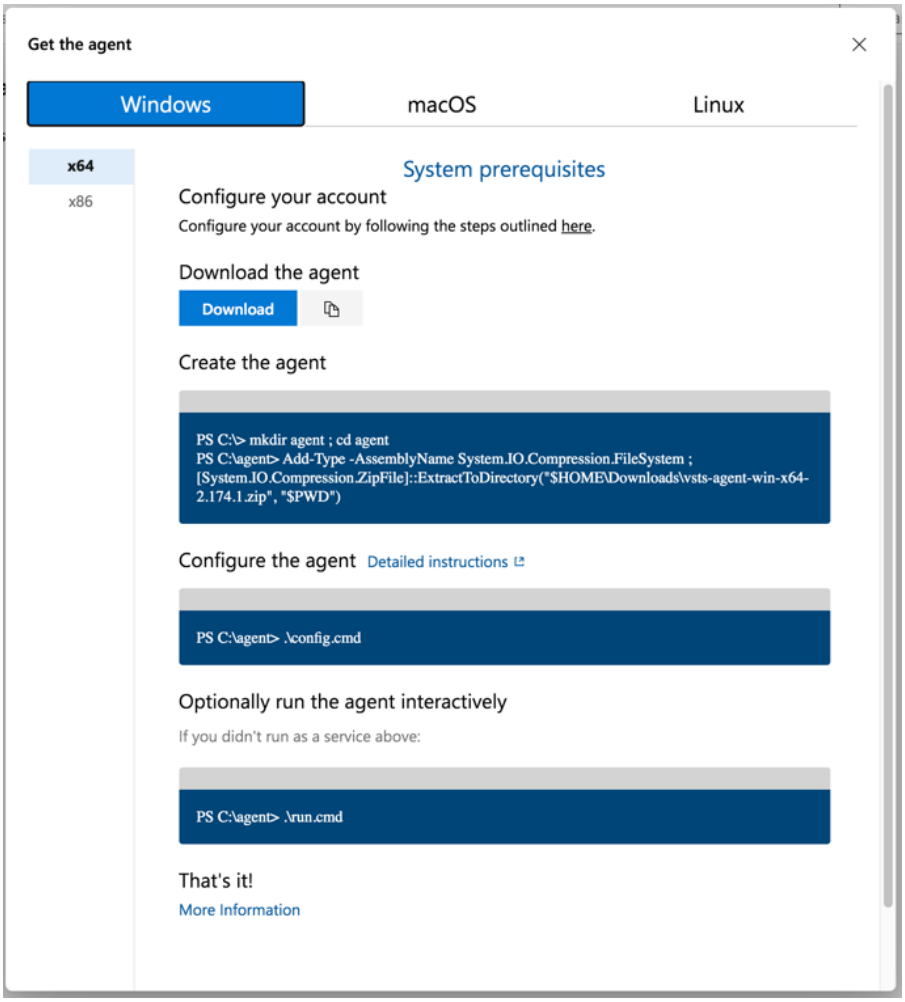


Figure 6.6: Create a new agent

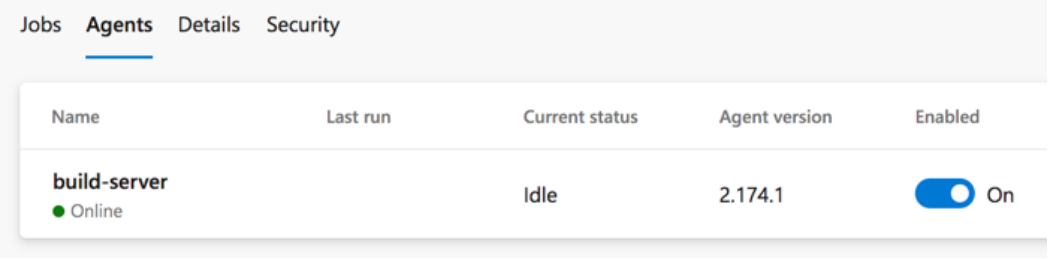
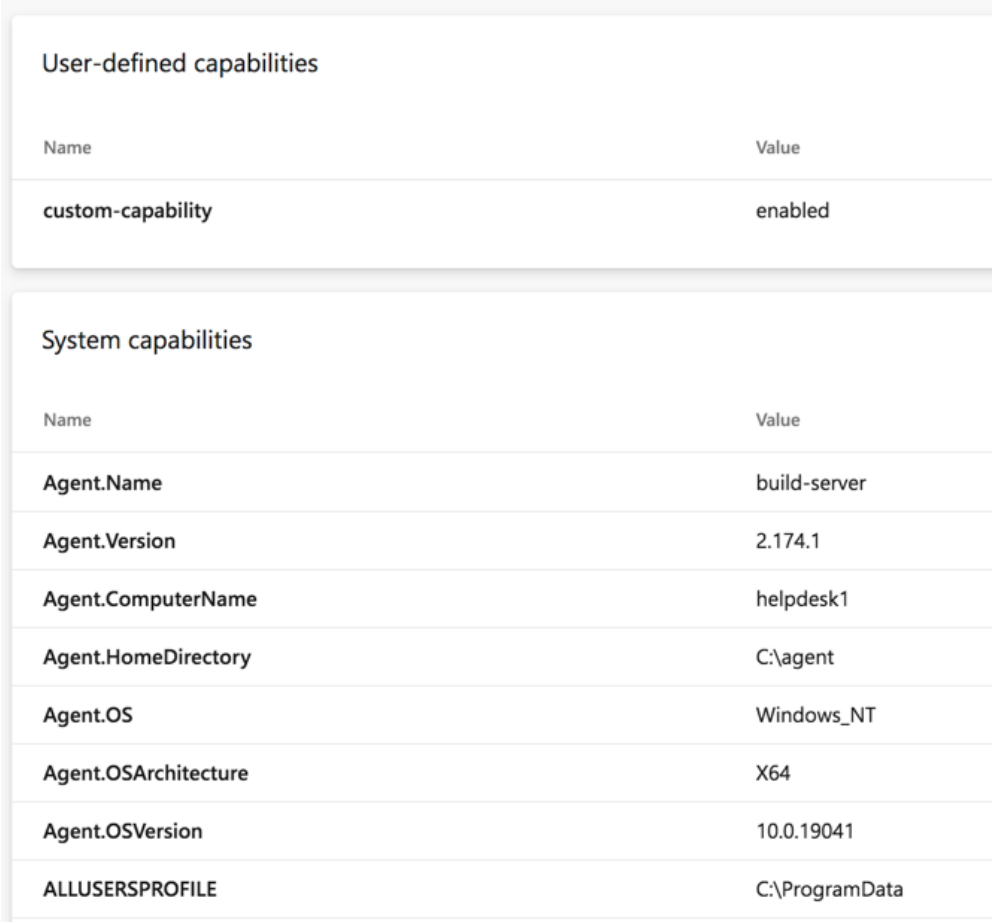


Figure 6.7: Private agents section



User-defined capabilities

Name	Value
custom-capability	enabled

System capabilities

Name	Value
Agent.Name	build-server
Agent.Version	2.174.1
Agent.ComputerName	helpdesk1
Agent.HomeDirectory	C:\agent
Agent.OS	Windows_NT
Agent.OSArchitecture	X64
Agent.OSVersion	10.0.19041
ALLUSERSPROFILE	C:\ProgramData

Figure 6.8: Capabilities of a private agent

Deploy the GalaxyHotel HelpDesk

This section will demonstrate how to provision the target machines and release the HelpDesk website.

Deployment groups

As explained before, the HelpDesk website will be deployed on three virtual machines. In this case, to make them reachable in Azure DevOps, you can leverage a **deployment group**, which is a group of related machines with an agent installed. These machines can be considered as a target for a release using a **deployment group job**.

You can provision a machine by going to the related section in Azure Pipelines and selecting **Add a deployment group**. Provide a name such as **HelpDesk-Test**, and click **Create**. A page with a PowerShell script opens. Select **Use a personal access token in the script for authentication**, click **Copy script to the clipboard**, and execute the script on the virtual machine created to host the HelpDesk website.

The script asks a few questions regarding the user account and tags (which are useful to distinguish machines based on capabilities). After the execution, the machine is listed in the Targets section of the deployment group page. The group just created will be considered as a test environment for the HelpDesk website. You should repeat the same steps for the **HelpDesk-Prod** deployment group, adding two more virtual machines for the production environment (Figure 6.9).

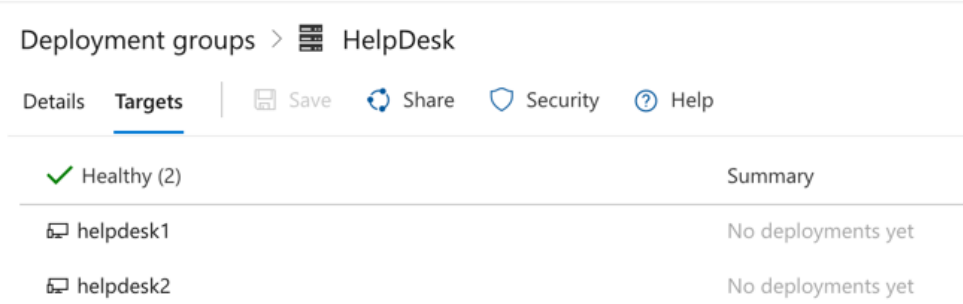


Figure 6.9: Targets in a deployment group

This page provides some insights about the status of the machines, tags, and related deployments. Deployment groups can also be shared with other projects.

Prepare the release

Once the deployment group is in place, you can target the virtual machines in the release.

Go to the release page in Azure Pipelines and click **New Pipeline**. In the popup, search for the term **IIS website deployment**, select **Apply**, and provide **HelpDesk-Test** as the name of the stage (Figure 6.10).

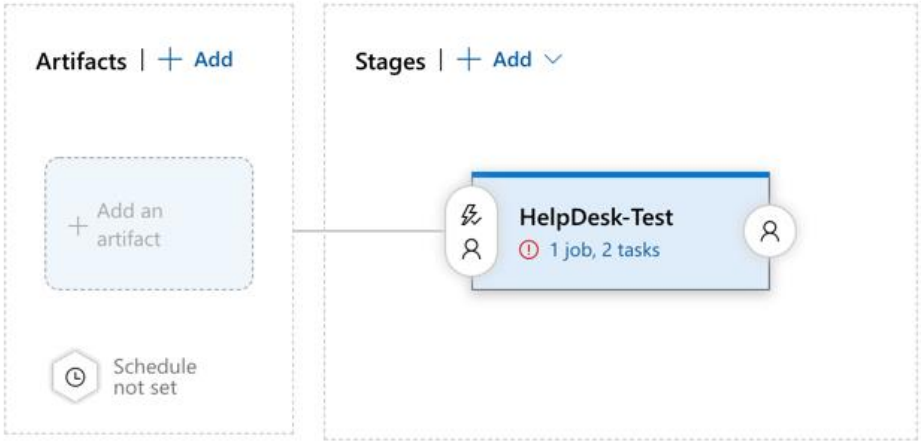


Figure 6.10: A release pipeline

Click **Add an artifact** and add the latest version of the build artifact created by the build definition **HelpDesk** (Figure 6.11).

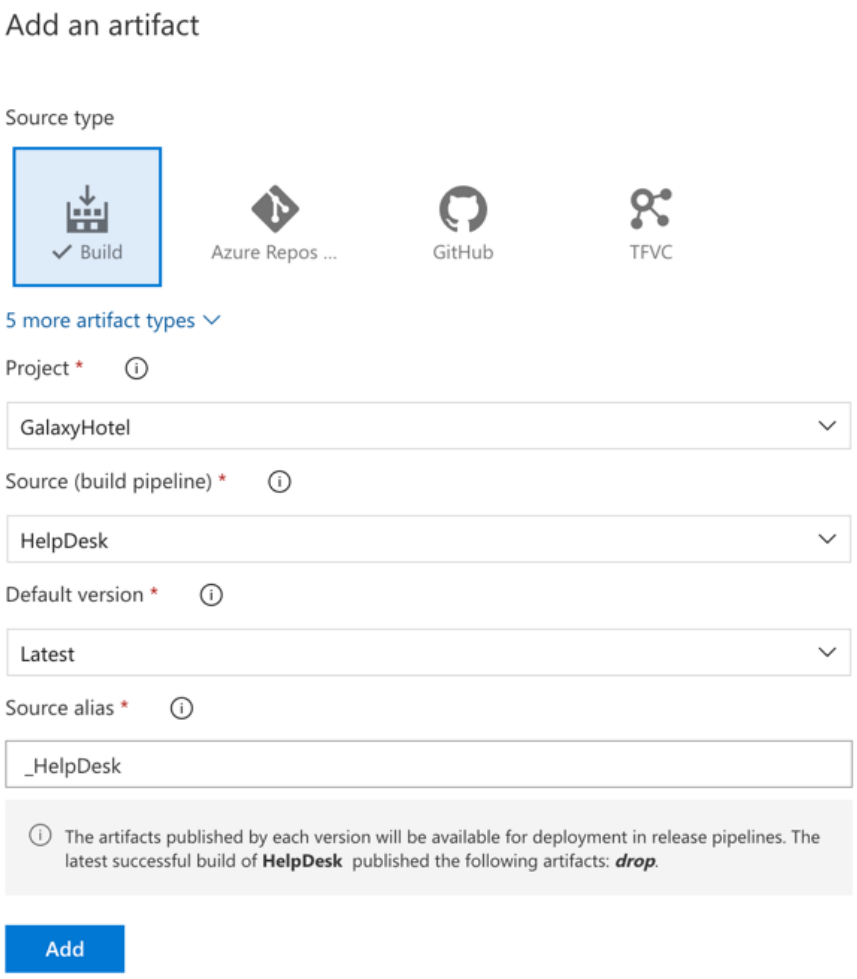


Figure 6.11: Release artifact popup

As you can see from the figure, releases support multiple types of artifacts such as Build, Azure Repos, GitHub, TFVC, and so on. The artifact can be originated from a different project or repository, and if necessary, select a specific version.

Once added, the **Continuous Deployment** trigger (lightning bolt icon) button appears. Select and enable the feature. In this case, a branch filter can also be applied (Figure 6.12). When you enable this flag, the release starts each time a new artifact is available.

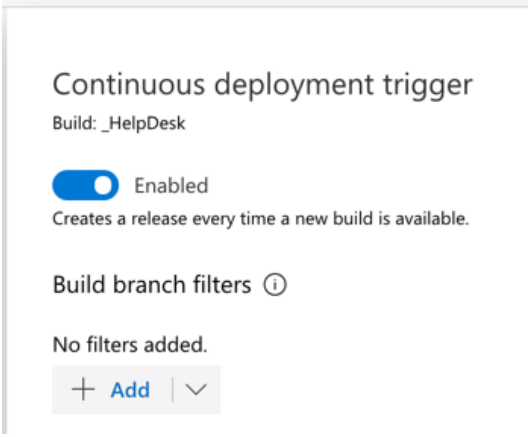


Figure 6.12: Enabling the continuous deployment trigger

Move the mouse over the **HelpDesk-Test** stage to reveal the **Add** and **Clone** stage buttons. Select **Clone** to create a new stage similar to the current one, and rename it **HelpDesk-Prod**. The new stage will appear on the right of the test stage, meaning that the production stage will start only after the previous one.

But what if two stages need to start together? And how can a user approve a release before deploying it? For these scenarios, Azure Pipelines makes available the **Pre-deployment** and **Post-deployment** conditions, which are configurable using the buttons at both sides of a stage (Figure 6.13). These options are one of the main differences from the multistage approach explained in the previous chapter. These conditions give you full control of what should be checked or approved before or after a release.

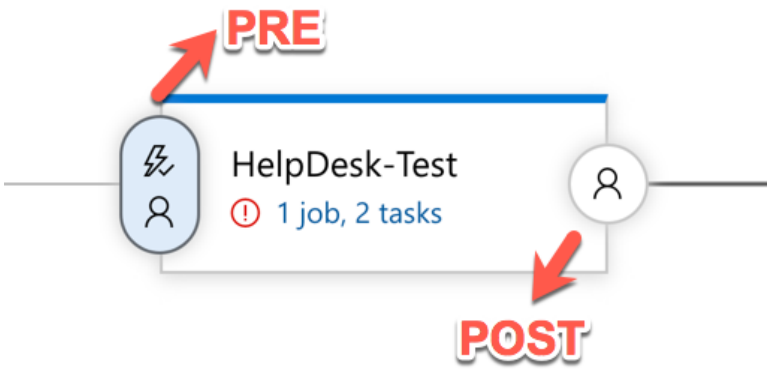


Figure 6.13: Pre-deployment and post-deployment buttons

For example, some possible scenarios are:

- Manually approve a release (pre).
- Make sure all bugs have been fixed before a stage (pre).
- Check a monitoring system to validate the release (post).

The list can go on and on, depending on your needs.

If you click the **Pre-deployment conditions** button on the **Test** stage, a popup opens with the following options:

- **Triggers**: Set what will start the release on this stage. Possible values are a release, another stage in the same release, or a manual release.
- **Artifact filters**: Manage multiple artifacts in the same release, and you want to start a stage depending only on a specific artifact.
- **Schedule**: Schedule a release, such as a nightly release.
- **Pull request deployment**: Start release-related artifacts derived from PRs.
- **Pre-deployment approvals**: Select an approver for the release.
- **Gates**: A powerful mechanism allowing checks on using external resources such as Azure Policy, Azure Monitor, Work Items (Query), Azure Functions, or REST API.
- **Deployment queue settings**: Define deployments' parallelism.

Post-deployment conditions are:

- **Post-deployment approvals**: Select an approver for the release.
- **Gates**: Similar to the pre-deployment conditions.
- **Auto-redeploy trigger**: Redeploy the release in case of failure, or when a new target is available in a deployment group.

For the HelpDesk website, you have to set the following conditions:

- **Pre-deployment**: Pre-deployment approval before the production stage (Figure 6.14).
- **Post-deployment**: Auto-redeploy when a new target is available (Figure 6.15).

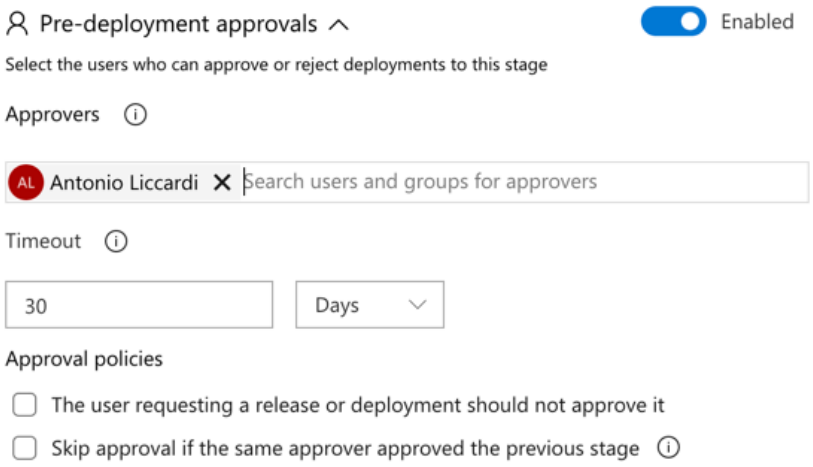


Figure 6.14: Setting approvers for a release

Figure 6.15: Setting a redeploy trigger

Before running the release, each stage should be modified to deploy the website. A warning icon advises that there is something not configured correctly. You can click on the task number link to open the stage details where a list of tasks is available. The editor is similar to the build definition, where each task has a specific pane with parameters (Figure 6.16).

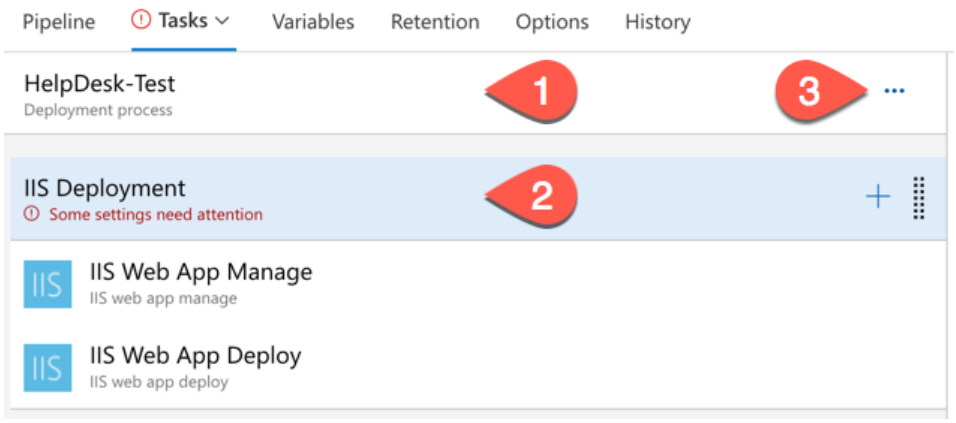


Figure 6.16: Stage details

The deployment process bar (1) lets you specify the stage name and a few settings linked to fields in the tasks. The IIS Deployment bar (2) represents a deployment group job, which executes a set of tasks on a deployment group. In this section, you can define the deployment group, a list of tags related to a specific machine (such as **`web`** or **`db`**), and whether the deployment should be applied in parallel or to one target at a time. Also, a release can have different phases by adding other types of jobs (3).

Tip: Among the types of jobs available, the agentless job is useful when you need manual intervention during a release.

Here is a list of changes we'll apply to the release:

1. In the IIS Deployment settings (2), select for each stage the related deployment group. Change the release name at the top of the page.

2. In each stage, the website will connect to a related database on Microsoft Azure. You should add a variable containing the connection strings for both environments. Azure Pipelines allows you to declare the same variable with different scopes. Go to the **Variables** tab, add two variables with the name **ConnectionStrings.DefaultConnection**, selecting a different scope for each (Figure 6.17). The variable name corresponds to the JSON path used in the next step.
3. Each deployment group job has two tasks: select the **IIS Web App Deploy** task, go to the section **File Transforms & Variable Substitution Options**, and type **appsettings.json** in the **JSON variable substitution** box. The option will replace the value of the connection string in the **appsettings.json** file with the one created in step 2.

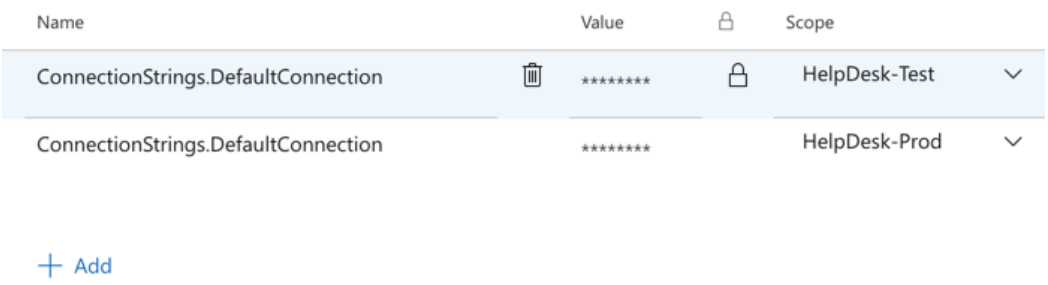


Figure 6.17: The same release variable with different scopes

Save the release and click **Create Release**.

Azure Pipelines will provide real-time updates on the release page, where you can select the stage to see the logs. At the end of the HelpDesk stage, the release stops, waiting for approval (Figure 6.18).

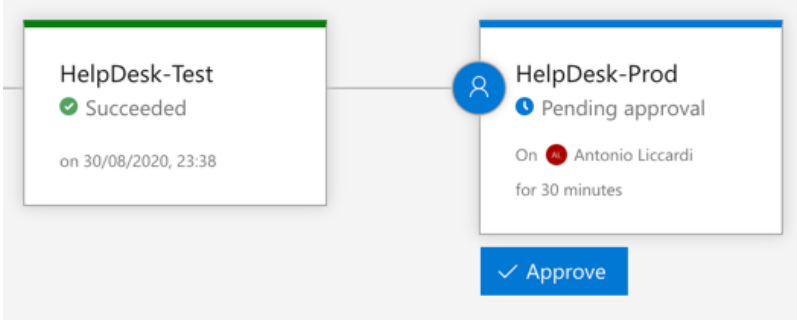


Figure 6.18: Waiting for release approval

After you approve it, the deployment on the production environment starts. If everything works correctly, the website is deployed on both machines. If you want to keep the release artifacts despite the retention policy, select **Retain indefinitely** from the release options (Figure 6.19).

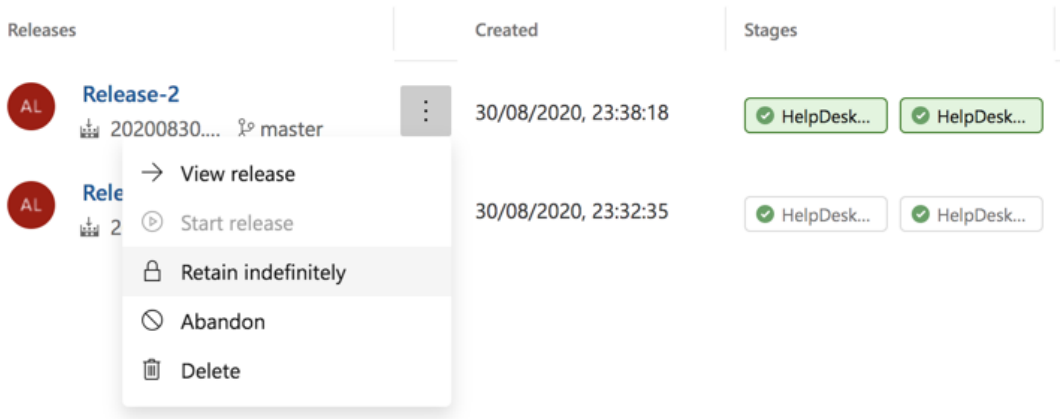


Figure 6.19: Retain a release indefinitely

Appendix A: Build and Deploy Containers

The rise of containers in the last decade has changed the way developers work. With the widespread use of Docker and Kubernetes, tools such as Azure DevOps are essential to facilitate the management of these technologies.

Azure Pipelines includes the following features:

- **Docker tasks**, to create and manage images using an image repository like Azure Container Registry (Figure A.1).
- **Azure Kubernetes tasks**, to manage deployment on Azure Kubernetes Service (Figure A.2).
- **Environments**, to manage and monitor clusters as an Azure Kubernetes Service (Figure A.3). The section provides detailed information about pods and deployments.

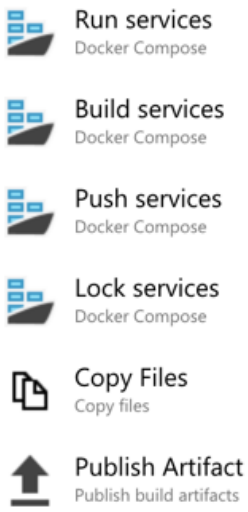


Figure A.1: Docker tasks in Azure Pipelines

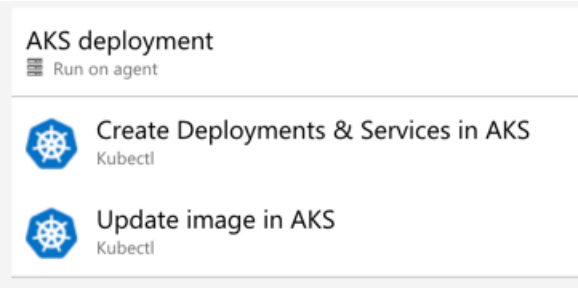


Figure A.2: Azure Kubernetes Service tasks in Azure Pipelines

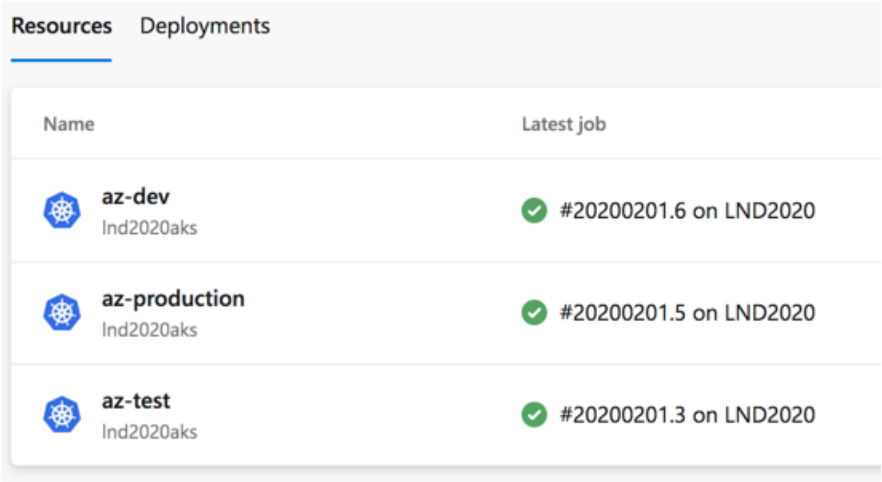


Figure A.3: An environment linked to an Azure Kubernetes Service

Appendix B: Security and Notification

In every organization, security plays a fundamental role: system administrators want to make sure that resources are not accessed by unauthorized users.

Azure DevOps takes security very seriously, allowing integration with Active Directory and defining a set of granular permissions for any user or group. The platform provides a collection of ready-to-use groups, simplifying user enrollment (Figure B.1).

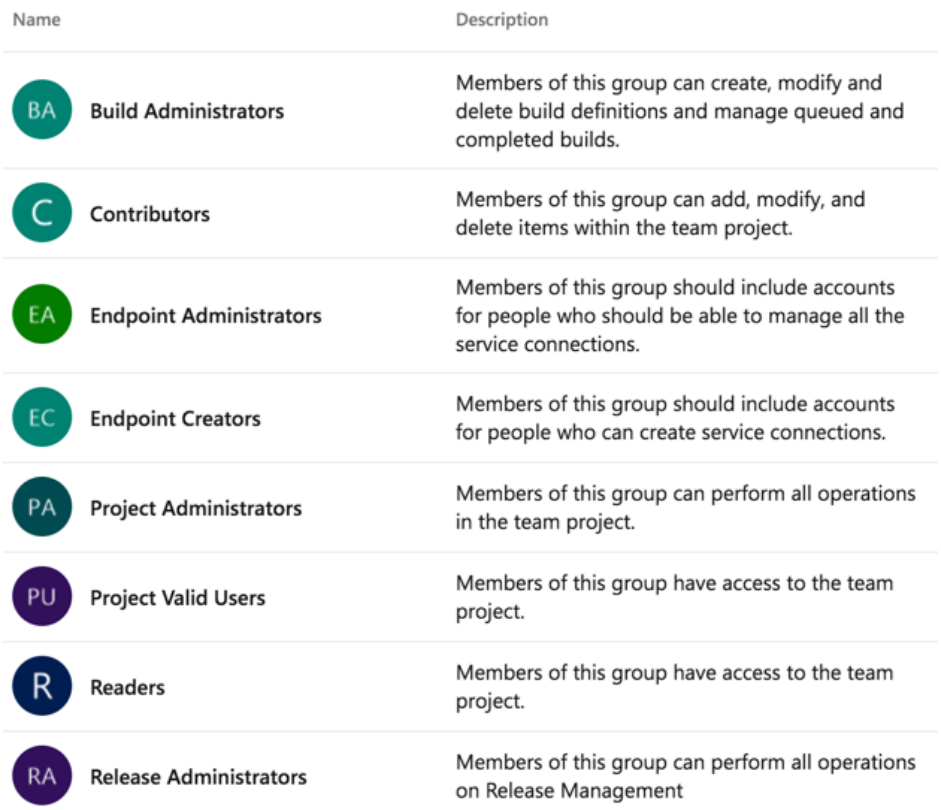


Figure B.1: Available groups in Azure DevOps

Administrators can also add a new group, clustering users based on requirements.

User access and permissions can be defined at:

- Organization level, granting access to specific projects.
- Project level, granting access to a specific section of the project.
- Section level, granting access to specific pages or features.

Each level has a predefined set of permissions, which can be granted or revoked. For example, at the project settings level, administrators define whether users can change project properties. At the repository level, administrators define whether users can create a branch or a tag (Figure B.2).

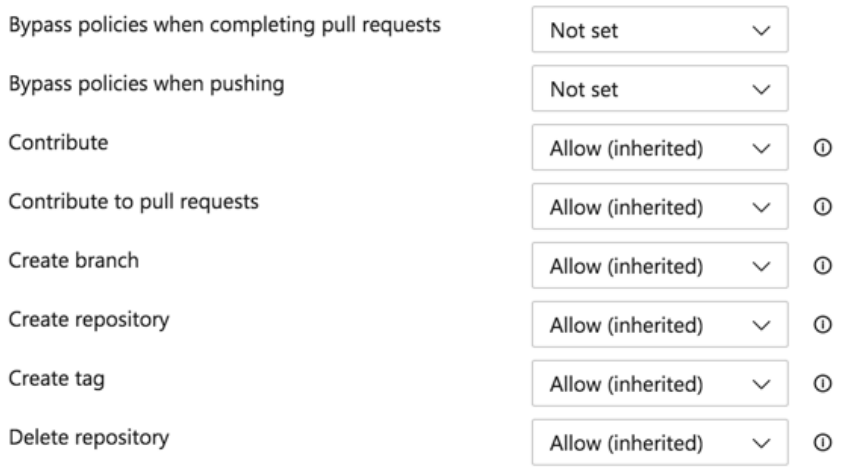


Figure B.2: A subset of repository permissions

The full list of permissions and groups is available here.

Azure DevOps also includes a **notification** system, which sends emails based on configurable events at the organization, project, or user level.

Administrators already have a set of preconfigured subscriptions related, for example, to builds or pipelines (Figure B.3).

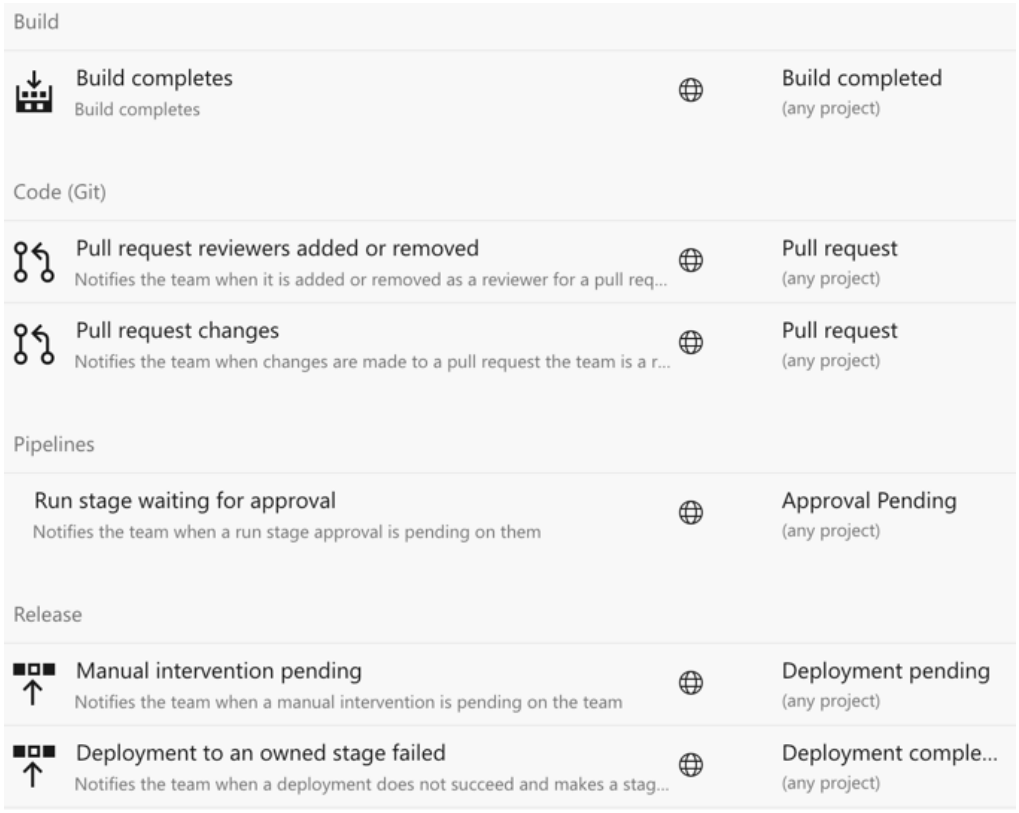


Figure B.3: Preconfigured notifications in Azure DevOps

You can add new subscriptions by using a template and configuring a set of parameters (Figure B.4).

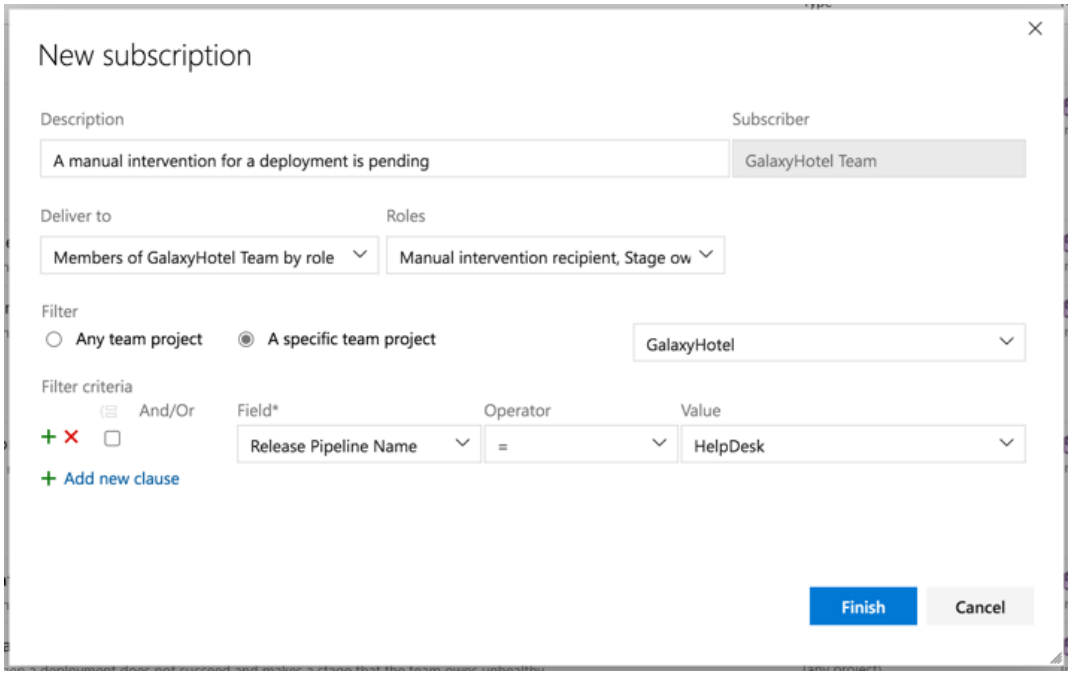


Figure B.4: New subscription popup