AZURE DEVOPS

SUCCINCTLY

BY SANDER ROSSEL

SUCCINCTLY EBOOK SERIES

Syncfusion

www.dbooks.org

Azure DevOps Succinctly

By

Sander Rossel

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

ISBN: 978-1-64200-204-1

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.

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

Sander Rossel is a Microsoft-certified professional developer with experience and expertise in .NET and .NET Core (C#, ASP.NET, and Entity Framework), SQL Server, Azure, Azure DevOps, JavaScript, and other technologies. He has an interest in various technologies including, but not limited to, cloud computing, NoSQL, continuous integration/continuous deployment, functional programming, and software quality in general. In his spare time, he writes articles for MSDN, CodeProject, and his own blog, as well as books about object-oriented programming, databases, and Azure.

Chapter 1 What Is DevOps?

By now you've probably heard the term "DevOps" (before downloading this book, of course). The word DevOps is a combination of the words "development" and "operations," which gives you an idea of what DevOps entails. DevOps is a set of practices that focus on collaboration between developers and operations (or system administrators). This may also include testers and security, and basically any party that is involved in delivering software to customers. The goal of DevOps is to release more-reliable and better-quality software at a faster and consistent pace.

Azure DevOps used to be named Visual Studio Team Services (VSTS) and Visual Studio Online (VSO), and is the cloud successor of Microsoft Team Foundation Server (TFS), now named Azure DevOps Server. Azure DevOps helps teams by implementing the technical side of DevOps. In this chapter we're going to explore DevOps, what it is, and how it can help companies produce better software, faster.

The history of DevOps

The term DevOps was first coined over a decade ago. Back then, different teams were often completely separated from one another. Developers would write software and hand it over to testers, who would test the software and either give the results back to the developers, or give a go to operations. Operations would then deploy the software.

It was not uncommon for these teams to be at separate locations and have no contact with each other—one team would do their job, and then throw it over the fence. The different teams would often have varying, conflicting goals and metrics of success. For example, developers need to deliver a feature as fast as possible, testers need to find as many defects as possible, and operations needs to have as much uptime as possible (and an update requires downtime). The result, of course, was that software was often buggy and delayed, and teams were unhappy and frustrated.

With new technologies such as virtualization, cloud, and containerization, as well as various automation tools on all levels of the development lifecycle, the expectations of both IT professionals and customers changed. Everything must work seamlessly, without downtime, and on demand. For developers, this means they need instant access to the various resources needed to run their code. Back in the day, this was not possible, as various teams were separate entities who protected their own domains. The result was unhappy customers.

The first DevOps conference, named devopsdays, was held in Ghent, Belgium in 2009. The first "State of DevOps" report was published in 2012. The report has been published yearly since 2014, when DevOps adoption started accelerating. DevOps is now a trend in the industry, inspiring conferences around the world. With DevOps practices and the right supporting tools, teams can deliver value to customers in a matter of minutes. A change request from the business can be in production minutes after a developer checks in the code. When fully realized, DevOps enables non-disruptive innovation at scale, anywhere in the world.

The basics of DevOps

As you may have concluded from the history of DevOps, it's not so much about tooling as it is about company culture. One where everyone works together to bring value to customers. In the broadest sense, it is the process of improving coordination and collaboration to produce better, more reliable products, and to achieve business goals more quickly and reliably.

By practicing DevOps, companies improve collaboration between teams, but the teams change as well. Instead of having teams defined by the work that they do, teams are defined by the result they deliver. So instead of having a team of developers and a team of testers, each team now has a developer and a tester, because they are both necessary to achieve the goal. Next to developers and testers, teams often comprise a Scrum master, product owner, business analyst, system administrator, and security engineer. Teams can change and people can be on multiple teams.

By adopting Agile methodologies, such as Scrum, teams can deliver value even more quickly, and at the same time increase stability and quality. The basic idea behind Scrum is that small teams of about four to seven people work in sprints of two or three weeks. During a daily Scrum meeting of about 15 minutes, they discuss their progress. A few times a week, the team takes time to look at open user stories and tasks that need to be carried out, estimate whether the story is clear enough to be carried out, and assess the amount of work necessary to perform the task. A product owner sorts the stories by priority, and because a team knows how much work they can do per sprint, the top *x* stories can be planned for the next sprint. At the end of each sprint, the teams do a review of the sprint to see what went well and what could have been done better. This way, teams continually strive to improve themselves.

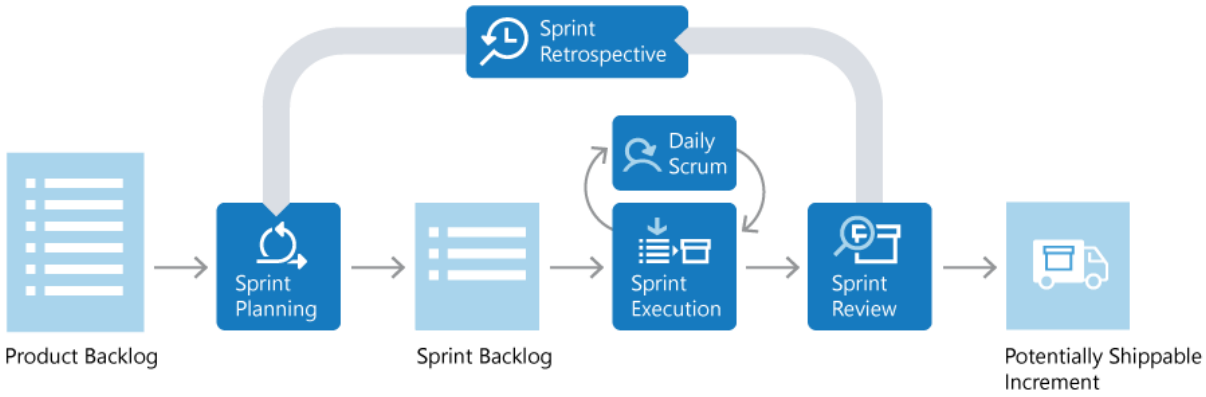

Figure 1: The Scrum Lifecycle. Sourced from [Microsoft Azure documentation](#).

This is a paradigm shift from the old Waterfall method, where specifications were collected up front and passed on to developers, who would then write code for months or even years straight, before ever delivering anything. When the team finally delivered, the specifications were often outdated and the software did not satisfy current needs.

There are other Agile methodologies, like Kanban, but Scrum is one of the most popular today. Like DevOps, these methodologies aim to deliver higher-quality software faster, and at a predictable pace. Agile methodologies and DevOps complement each other. Where Scrum can be an implementation of the cultural aspect of DevOps, DevOps tooling has a focus on automated software builds, tests, and deployment. And that tooling is where Azure DevOps comes into play, as we'll see in the remainder of this book.

Automation

While a certain company culture is required for DevOps to succeed, many people focus on the technical aspect of DevOps. The key word here is *automation*. When a developer and a system administrator work together, it suddenly becomes possible for code to be shipped automatically, using automated tools that are already at the developer's disposal. That's where Azure DevOps comes in, but other tools like GitHub, GitLab, Jenkins, and Bamboo can do the same thing.

Imagine that your code is under Git source control. The moment you check in new code, a build is automatically triggered on the server. When the build passes, you're sure you didn't commit any typing errors, omit references to local packages, or anything else that would prevent your coworkers from building the source locally. After the build is passed, your code is automatically tested using the unit tests you, your coworkers, and your testers wrote. Once those tests are passed, your code is deployed to a staging environment where testers can do some manual tests, like acceptance tests. After the testers give their blessing, you can one-click deploy the software to a production environment with little or no downtime.

The process of automatically building and testing your code is called continuous integration, or CI for short. The automatic deployment after that is called continuous delivery. If all manual steps are eliminated and the code goes from a commit to production fully automated, we're talking about continuous deployment. Both continuous delivery and deployment are abbreviated as CD. In practice, most people mean continuous delivery when talking about CD. Very few companies feel comfortable with the thought that code goes into production completely automatically. You will often see the term CI/CD, meaning teams do both, although you can't have CD without CI. CI/CD is also sometimes used interchangeably with DevOps.

With tooling such as automated builds and releases, version control and work tracking, teams have everything they need to collaborate on software projects. Azure DevOps has support for all of them. Where other tools do one or two of those things, Azure DevOps is a complete solution.

The cloud

With the emergence of cloud technologies in the past few years, DevOps has become increasingly more popular, especially CI/CD. It's never been easier to deploy various resources using just a script. Whether that's JSON, PowerShell, or a CLI script, it's now possible to deploy a brand-new website with little or no manual intervention.

In fact, Azure DevOps itself is a cloud solution. It's the successor to the on-premises Team Foundation Server (TFS). As can be expected from a Microsoft product, Azure DevOps has out-of-the-box integration with .NET, .NET Core, and the Azure cloud. In the next chapters we're going to create an Azure account and deploy a website there, but we'll also deploy locally to our own computer. In the cloud, however, we see development and operations really coming together. Gone are the days when you needed physical access to a server to deploy software on it. And since everything can be deployed using scripts, developers can write the deployment procedures. Operations, or system administrators, can help in this endeavor, but may feel more comfortable in providing access and monitoring cloud resources. Of course, when it comes to networking, like limiting access to your public cloud to your own internal network, it becomes apparent how teams with both Dev and Ops can really work together.

With both cloud and DevOps, new architectural patterns have become possible, most notably microservices. With microservices, an otherwise large application is divided into a set of smaller subsystems where each subsystem can function completely on its own. For example, an application can have a product module, a sales module, a stock module, and an invoice or financial module. Going into the details of microservices is outside the scope of this book, but it is important to note that these services all have their own databases and possibly run on different servers. The advantage to having microservices is that scaling is easier than with one large application, or a monolith. Especially when done in the cloud, scaling becomes massive and automatic. Also, when one service fails, parts of the system can still be used if other services continue to work. The downside is that you have a lot of deployments. Instead of deploying one huge application (which is usually just a few files anyway), you now have to deal with tens or even hundreds of releases. Releasing everything manually would be a full-time job, but with CI/CD this process is completely automated. Containerization tools such as Docker and Kubernetes help to manage these complex deployment scenarios.

Culture

We have already been over this, but I can't stress this enough: DevOps is about company culture. However, forming cross-functional teams and calling yourself Agile is not going to cut it. That is something I come across often—companies calling themselves Agile while they're just unorganized, or saying they do DevOps while they are just running a CI pipeline. That is not DevOps. DevOps is about changing the company culture and having the tools to support that culture.

When teams are cross-functional, they have to take shared ownership of the code and the environment. Usually, when a bug is found in production, finger-pointing begins. The developers will blame the testers, testers will blame developers or system administrators, and system administrators will blame everyone else. After a team takes shared ownership, it is not anyone's fault; it is the team's fault, and the team will have to solve the issue. It is a shared responsibility. And suddenly there are fewer conflicting interests.

In DevOps it is not important who gets the blame. It is important that issues get fixed as soon as possible, even when the original author has a day off.

Of course, this change in culture does not stop with the technical team. Bugs in software are often blamed on users asking for impossible features, or on management cutting time or budgets. However, blaming is not part of DevOps. If users and management are involved in the development process, impossible features can be discussed with users earlier, and management knows about time and money constraints before they result in bugs and unhappy users.

The mindset in a DevOps environment should not be that of name and shame, but of blameless and collaborative problem solving. By adopting Agile methodologies with cross-functional teams, everyone is involved in estimating and planning work.

Where does Azure DevOps fit into this?

Azure DevOps supports teams that work in a DevOps environment. DevOps allows companies to create teams and projects and let them work independently of each other. With Azure Boards, teams can create and prioritize user stories, estimate efforts, and plan sprints. Repos allows teams to put code into Git source control, link commits to user stories, and enforce code reviews. By linking commits to stories, you will more easily be able to see how far a user story is completed or why bugs happen.

Pipelines are an important aspect of Azure DevOps. In fact, I have worked in teams that only used this feature. With pipelines, teams can create CI/CD pipelines. It is easy to automatically trigger builds or releases on code commits.

In a build pipeline, code can be pulled from Azure Repos, GitHub, Bitbucket Cloud, other Git sources, and even Subversion. You can build, test, and package code in pretty much every language. Needless to say, .NET and .NET Core are supported out of the box. You can also build Java projects using Maven or Apache Ant, or build Go applications. Tools such as Grunt, Gulp, and Docker are also directly available. For JavaScript projects, there's npm support. Even Apple is supported with Xcode and Xamarin.iOS (as well as Xamarin.Android, of course). Your builds can run on Windows, Ubuntu, or MacOS, but you can use your own device as well. Furthermore, pretty much anything is possible using the Bash shell, the Windows command line, or a Python script.

Once your build is finished, you can deploy your code using a release pipeline. Your pipelines can have multiple stages, like development, test, and production. There is plenty of support for Azure deployments. You can also deploy databases, for example: SQL Server with a DACPAC or SQL script. You can deploy using Docker and Kubernetes. Even if something is not available in your pipelines, chances are you can add it from the Marketplace. For example, you can add SonarQube, Yarn, AWS Toolkit, Terraform, or Google Play.

A less-used feature of Azure DevOps is test plans. This is where testers can create manual test plans, run them, and create test reports. Tests can be created for specific versions of your software or for sprints.

Last, Azure DevOps offers artifacts. These are reusable software packages. Supported package management tools include NuGet, npm, Maven, and pip.

Summary

DevOps is a mindset that is all about collaboration and shipping high-quality software at a predictable pace. Azure DevOps offers a complete package for developers, operations, and business users to support collaboration and automation. In the remainder of this book, we are going to explore the various features Azure DevOps has to offer.

Chapter 2 Configuring Azure DevOps

In the previous chapter, we looked at the theoretical foundations of DevOps. We have also seen how Azure DevOps can help teams adopting DevOps. In this chapter we are going to create an Azure and Azure DevOps account and configure Azure DevOps.

While Azure and Azure DevOps are free to get started, you do need to enter your credit card information when creating an Azure account. This is very unfortunate for people from countries where credit cards are not as accepted as in the United States. In fact, I got my credit card just so I could create an account.

Creating a Microsoft account

Let us start by creating a Microsoft account. If you already have one, you can skip this section. Browse to this site to either sign in or create a Microsoft account. Click **Sign In**, which will allow you to create a new account and give you some extra options. You can sign up using a personal account by clicking **No account? Create one!** You can also create a new Outlook account by clicking **Get a new email address**.

Alternatively, if you don't have an account yet and want a new email address, the easiest thing to do is to create an Outlook account here. I will explain this a bit more in the next section.

By clicking on **Sign-in options**, you can also sign in using a personal GitHub account with OAuth or a security key.

Note: Microsoft has really messed up account creation in the past. Personally, I still have an email address that is used for both a personal and a work account, and I have no way of seeing which account I am logged into. Sometimes I have to switch accounts, which is just logging in again with the same email address. To fix this issue, Microsoft has disabled registration with a work or school email address. In practice, this means any email from a domain that is configured in an Azure Active Directory (Azure AD or AAD). So, if you have an email address myname@mycompany.com and want to use it at this point, you may be out of luck. No worries though; you can still use it later. You can read more about this issue here.

Creating an Outlook account

I am going to sidetrack a little here to look at the Outlook account creation. I created one just for this book. It is mostly straightforward, save for some minor details. The CAPTCHA is all caps and no space (even if the characters are on separate lines). Microsoft asks for your phone number, which is used for account retrieval or two-factor authentication (2FA). They are sending you an access code, so you have to have access to the entered phone number.

After the account creation, you may get an error page. Just wait a minute or so and try logging in again. Make sure you first log out with any other Microsoft accounts you may have. You should now have access to your Outlook mailbox.

Tip: If you are using this account just for the sake of having a Microsoft account, you may want to forward your email to your main email account. That way, you will never miss important emails, like Azure updates or issues with your account. Click the cog in the upper-right corner and click** View all Outlook settings **at the bottom. You may have to verify your account. Notice that they are first asking you for the last four digits of your phone number. I was waiting for them to send me a code, but that is the next step. If you read the text it is obvious, but you may not expect it. Once you have verified your account, go to** Forwarding **and enter your preferred address.

Creating an Azure tenant

Next, we are going to create an Azure tenant and subscription. An Azure tenant is basically a container for your users, accounts, and subscriptions. A subscription is a service within Azure that allows you to create (paid) Azure resources. Obviously, if you already have an Azure subscription, you can skip this section.

Browse to this website and click **Start free**. You can only use the free service if you do not have another Azure subscription associated with your Microsoft account. In the next window, you have to sign in with a Microsoft account.

When you are logged in, Microsoft will take you to the identity verification page. Enter or verify your information. Next, you will need to authenticate by phone. After that you need to enter your credit card information (again, this is unfortunate, especially since it is not even charged, but for now it's a necessity). In the agreement, you have to explicitly agree to the subscription agreement, offer details, and privacy statement. You also have to explicitly opt out of the Microsoft newsletter with tips on Azure, pricing updates, and offers from Microsoft and partners.

After you have agreed, you will be taken to a page to register for a demo, or whatever offer they have available at the time you register. Skip it and go straight to the Azure portal, either by clicking the on-screen button or by browsing to the Azure portal. Here you will see your new tenant and subscription.

Azure subscriptions

In the Azure portal, use the hamburger menu at the top right, go to **All services**, and search for **Subscriptions**. Click **Subscriptions**. This should give an overview of your subscriptions, of which you currently have one: the **Free trial** subscription.

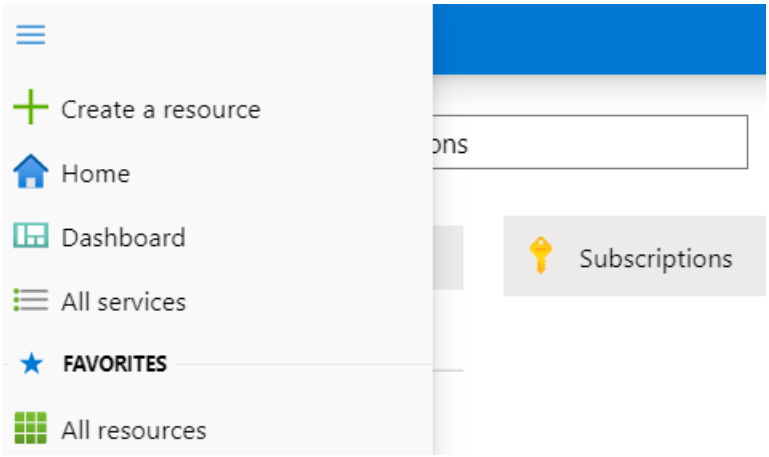

Figure 2: Find Azure Subscriptions

In the Subscriptions screen, you can upgrade your subscription to a paid one (or pay-as-you-go). Do not do this yet, because you will miss out on the free stuff. The free subscription offers a couple of popular services for free (at certain tiers), like 5 GB of Blob storage and file storage, and 250 GB SQL databases, as well as $200.00 or €170.00 credit to use for paid services. It is important to know that Azure bills according to usage. So, if you create a virtual machine that costs €100.00 a month, but you delete it after half a month, you are only billed €50.00. If you keep track of your expenses and delete resources once you are done with them, €170.00 can last you quite some time. Some Azure resources are always free, like serverless offerings (see my other book, *Azure Serverless Succinctly*), the free tier of App Service, and Azure Active Directory.

Azure Active Directory

The heart of your Azure account lies in the Azure Active Directory, or AAD. This is where you can find the name and ID of your Azure tenant. When you use the top-right hamburger menu, it should be in the list of favorites. It is also at the top of your home screen by default. You will be taken to the Azure AD overview. The user-friendly name for your directory is "Default Directory," which you can change. If you signed up using an Outlook account, the name of your tenant will be something like [*youremail*]outlook[*number*].onmicrosoft.com—not a nice name, and you cannot change it. You can create another tenant and pick your name, but that is not in the scope of this book. However, if you do want to experiment with this, you can click **+ Create a directory** at the top. There is also a **Delete directory** button, which you can use to delete your directory later if you want to.

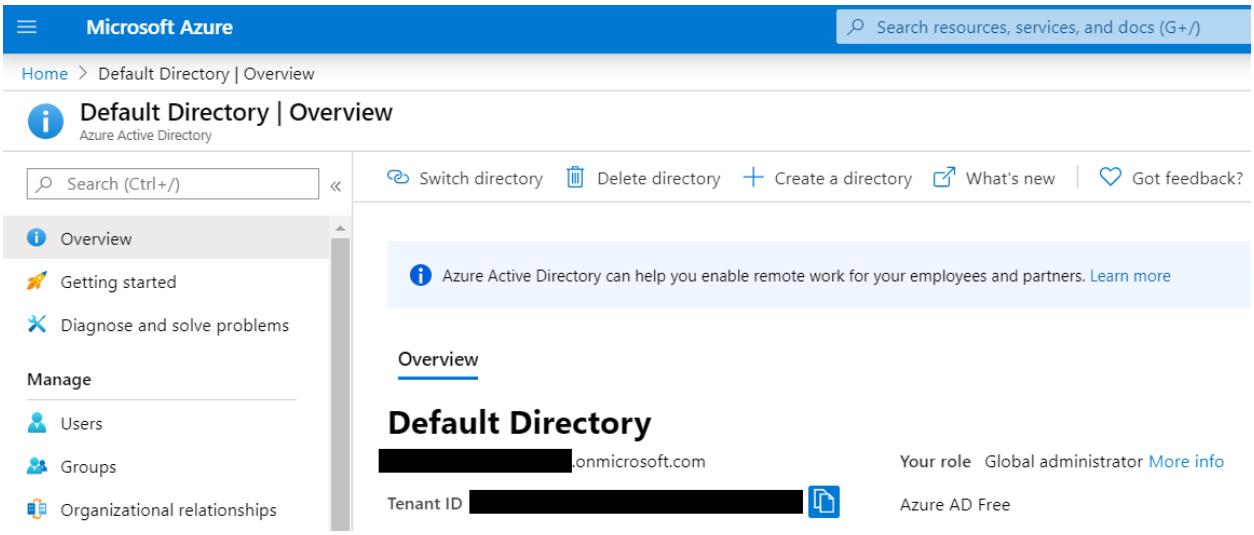

Figure 3: Azure AD Overview

If you go to **Properties**, you can change **Default Directory** to **Succinctly Directory**, just **Succinctly**, or whatever you like. You can also change your notification language, your technical contact email address, and your privacy information. Do not forget to save if you change anything.

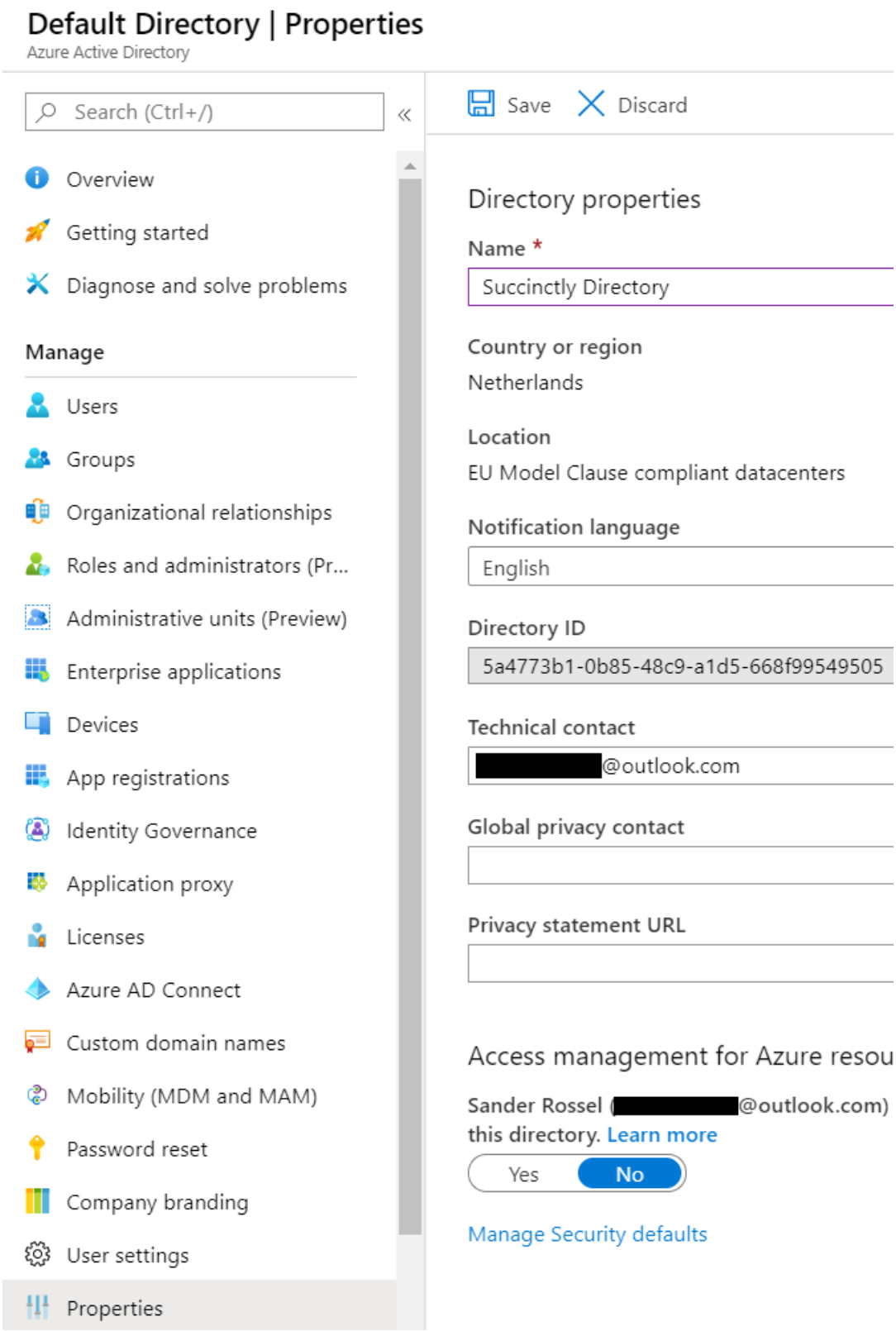

Figure 4: Azure AD Properties

Last, you can view all your users by going to **Users**. Users that we add in Azure DevOps will be added here as well. We will add some users later.

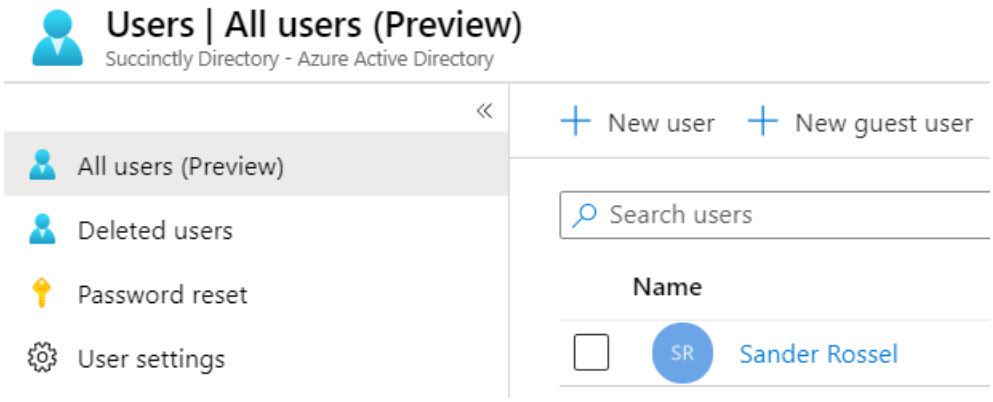

Figure 5: Azure AD Users

As you probably already guessed, Azure has much more to offer, but these are the important services that we will need in the remainder of this chapter. Azure AD is especially important, and we are going to see a lot of it.

Creating an Azure DevOps account

Next, we can create an Azure DevOps account. Now that we have a Microsoft account, this should be a lot easier. Go here and click **Start free**. Alternatively, you can click **Start free with GitHub**.

Since you already have an account that is connected to Azure, Azure DevOps just asks you to verify a few things. Again, opt out of the promotional offers, enter your country, and click **Continue**. Optionally, you could switch your directory if you have multiple, but we are not doing that now.

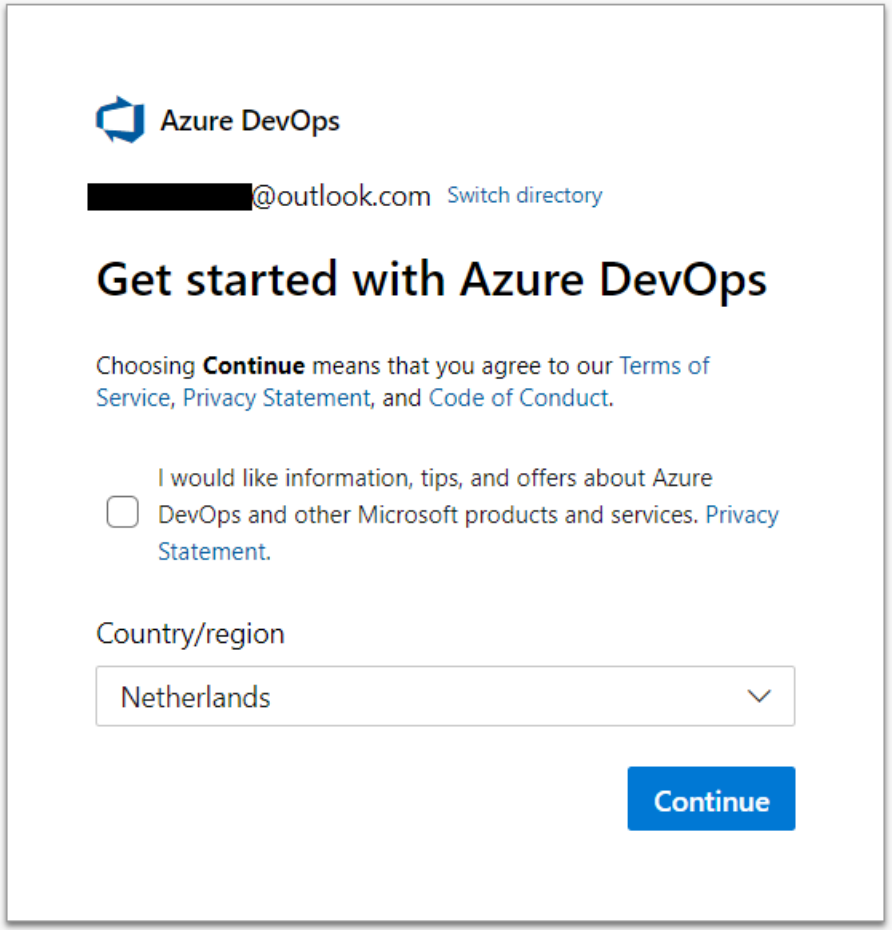

Figure 6: Creating an Azure DevOps Account

The next screen takes you to your default organization, which is named something like [*youremail*][*number*]. Most of the screen is dedicated to project creation. Here, you can enter a project name and description, and specify whether it is a public or private project, whether you want to use Git or Team Foundation Version Control (TFVC), and your work item process (basic, Agile, Scrum, or CMMI). Those last two options are under **Advanced**, and you should leave them for now.

You should not use TFVC unless you have a specific need. It is an older version control system that is centralized. Git is a distributed version control system, and the de facto standard in the industry. For more information on the two, click the question mark icon. The workings of the different source control systems are not in scope of this book.

We will look at the different work item processes in the next chapter.

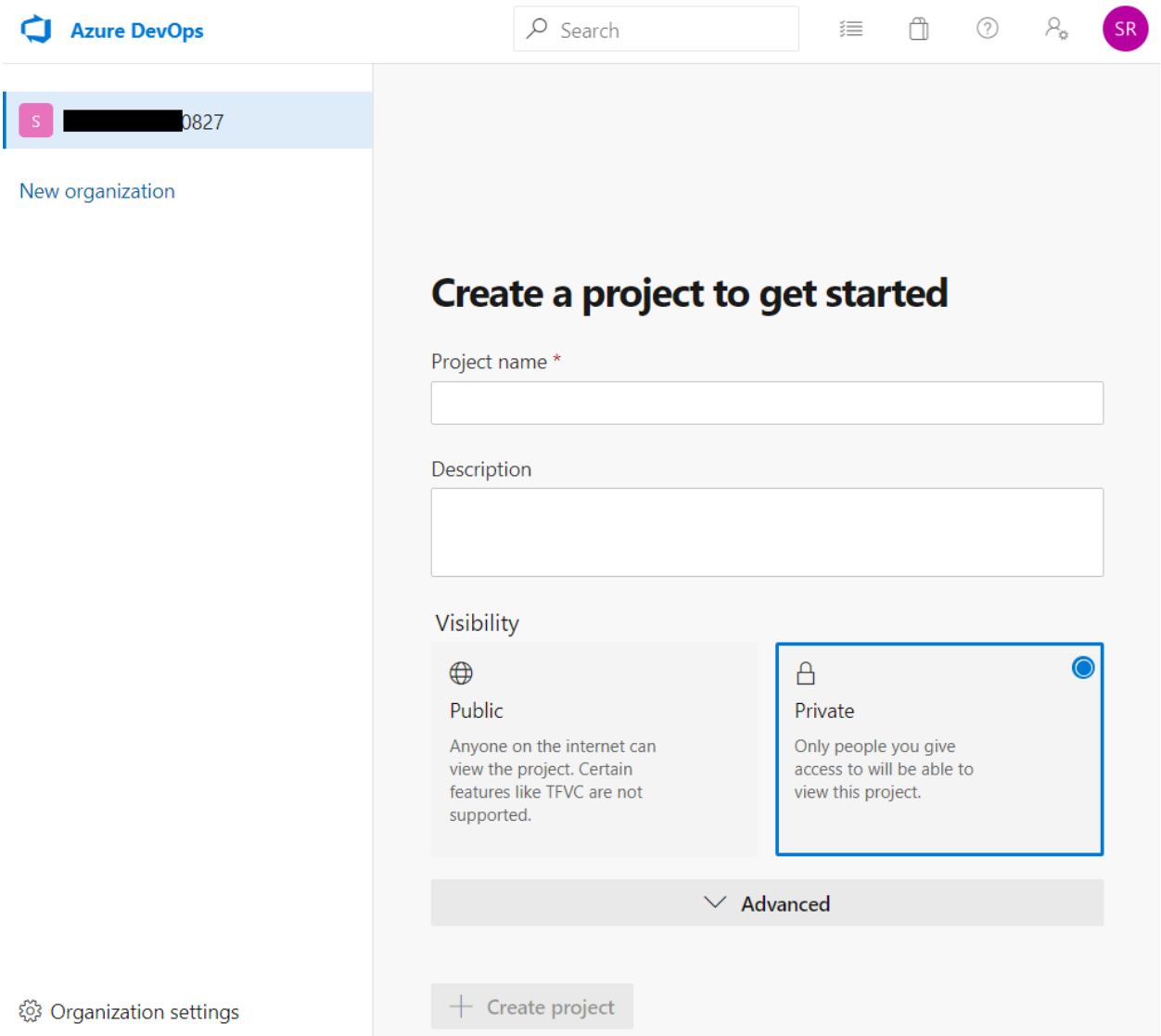

Figure 7: Azure DevOps Home Page

That's it—you now have your very own Azure DevOps environment. You may create a project now, but we will be doing that later in this chapter as well. In the remainder of this chapter, we are going to focus on organization and project configuration.

Managing your DevOps organizations

Before we create a project, let us look at some organization settings. You can find the organization settings in the lower-left corner. The first thing we want to do is change our company name. The Name field only allows alphanumeric characters, no spaces or special characters. You can enter uppercase and lowercase letters, but I prefer all lowercase myself. I am going to change the company name to **succinctlyexample**. The organization name must be unique, or you will get a message that the name is unavailable. I am also going to change my time zone to **UTC+01:00** for the Netherlands (Amsterdam). You can add a privacy URL and an organization description if you like. When you save, you will have to retype your organization name.

Changing your organization name has a great impact on your projects and teams. Not now, because you do not have anything yet, but it will in the future. Maybe you have noticed, but the URL of your Azure DevOps environment is https://dev.azure.com/[*yourorganizationname*]/. Changing the organization name changes the URL of your DevOps environment! That means everyone in the entire company has to change every DevOps URL they have. You need to change documentation, bookmarks, Git repositories, possible feeds to NuGet, other packages, and everything else that somehow links to your Azure DevOps account. Changing the name and getting a new URL may take a while, so your new environment may not be directly available. If you have some projects, expect some downtime. So, the little warning before saving is in order.

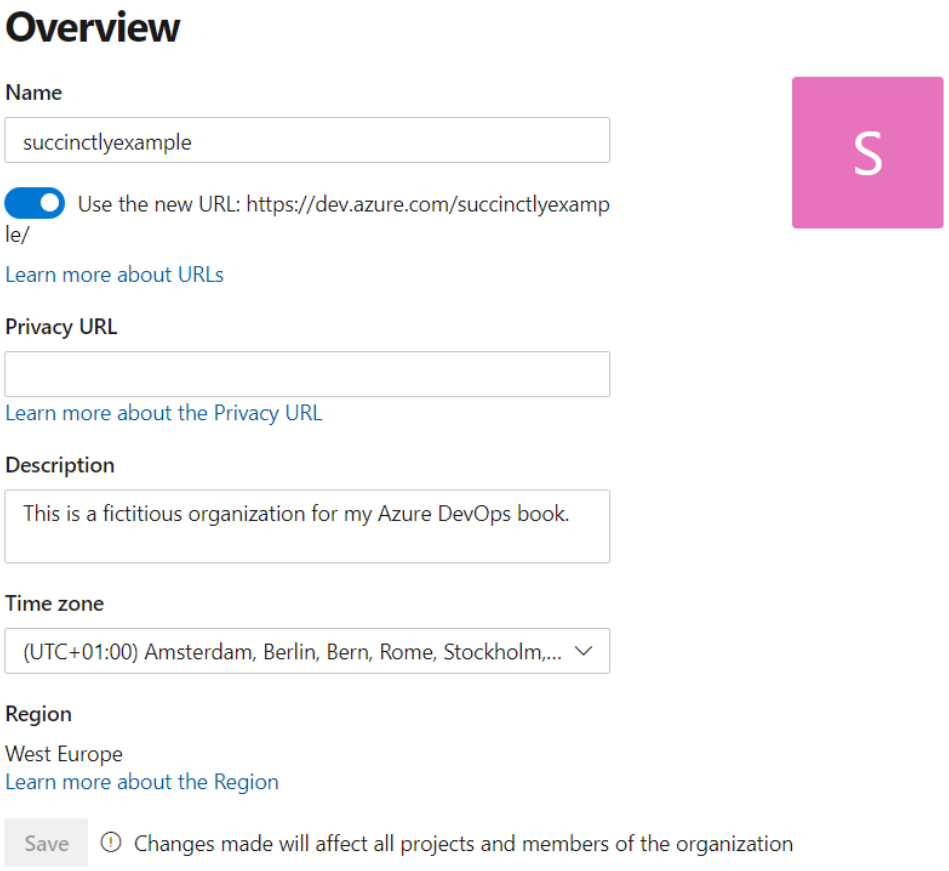

Figure 8: Azure DevOps Organization Settings

Another important setting you can view here is the region of your company. Mine is currently West Europe. It was set automatically. For some reason, we were not able to set it at organization creation, but if you would create a new company now you can, as we will see in a bit. Region is important because every region has its own privacy laws. For example, a lot of European companies do not want their data to leave Europe because Europe has strict privacy laws. My customers often ask me explicitly where their data is stored. The weird thing about region is that it cannot be changed here. Instead, you need to go to the DevOps Virtual Agent. There you need to select the quick action **Change Organization Region**. There have been reports that the Change Organization Region feature does not work properly in some scenarios. One alternative is to create a new organization, but this usually is not feasible. So hopefully the Virtual Agent will work for you, or you will have to contact Microsoft support.

Another setting that can be changed here is the organization owner, but that is only possible when you have at least one other user in your company. An organization owner has all permissions in an organization, so be careful who you give ownership to. Of course, not everyone can change the organization owner.

You can also delete an organization. This will irrevocably delete all your boards, repositories, pipelines, test plans, and artifacts. So do not ever use this self-destruct button unless you are very sure you can delete an organization, like the test organization we are using now.

Before we look into any other settings, let us look at how to create a new organization where you can set the name and region on creation.

Creating a new organization

Go back to the main page of Azure DevOps, https://dev.azure.com/succinctlyexample (of course, your organization name will be different). Now, at the left, underneath your organization name, you can click **New organization**. You will be taken to the same screen as before. However, when you click **Continue**, you will now be taken to a second screen where you can pick an organization name and region. After that you will be taken back to the home page of Azure DevOps, and you will now have two organizations on the left side of the screen.

Other than that, this organization is the same as the other one. Keep both organizations for now, but pick one that you will be using for the remainder of the book.

Not all your organizations will show up in the organizations list on the left side. We are going to connect our environment to our Azure AD later, and only organizations in the same Azure AD will show up here. It is somewhat confusing and annoying, but that is how it currently works.

Creating a project

Creating your first project is easy. Just go to the home screen and fill out the project name, which can contain uppercase and lowercase letters, numbers, and some special characters (I have named mine **Sample project**), and an optional description. Leave the visibility on **Private** and leave the advanced settings alone. Then click **+ Create project**. Your project will now be created, and you will be taken to the project page. The project overview page is still empty now, but we can fix that later.

For now, go back to the organization page by clicking on the Azure DevOps logo at the top left, or in the breadcrumbs at the top. The organization page looks different now. It shows your projects, of which you now have one. You can also switch to **My work items** and **My pull requests**, which we will create in later chapters. You can create new projects on the top right.

Every project can have multiple boards and repositories, so you really have to plan how you will be creating your projects. For example, your company may be creating an ERP application and a CRM application. You can create two projects: one for ERP, and one for CRM. Both projects will probably have multiple code repositories. Perhaps you have some shared services between the two, so you can create a third project named **Core services** or whatever. Or maybe those will be separate repositories in the ERP project. You can use separate boards in both projects, and you can have one board that both project teams use, or separate boards in the same project. You can also just have a single project and keep all the repositories for both the ERP and CRM in there. If you have multiple customers for whom you create customer-specific products, put those in separate projects. Do not mix different customers in the same project. Also, if you have separate teams, give them their own projects. So, if one team only did the ERP and another other team only did the CRM, give them separate projects.

I will give you an example of my own company, JUUN Software. I have one organization that I use for a couple of projects. Each project is for a different customer. They are relatively small projects for customers who have no IT department or on-premises infrastructure. Some projects have multiple repositories for different parts of the software. I also have a customer who wants access to the source code I write for them. They basically hired me to set up their Azure environment and write software for them, but they want to be able to expand the team or continue the project when I'm no longer involved at some point in the future, as they do have their own IT department. I set up a separate organization for them, which has multiple projects.

In the past, I have worked for a company that always kept all source code in-house and never shared with their customer. They had one organization and maybe hundreds of projects of which sometimes tens of projects belonged to the same customer.

On top of that, you can assign users to projects individually, so project access is another variable to consider when creating new projects.

As you can see, different requirements require different strategies.

User management

It is time to add another user to our project. For this, you will need a second email address. You probably already have one (maybe a work address or a second private address), but if you do not, create one. This can be another Outlook account, a Gmail account, or even a work account that is already registered in an Azure AD somewhere.

The easiest way to add a new user is by going to your organization settings and clicking **Users** in the menu on the left. On the right, click **Add users**. Now, simply add one or more email addresses of the users you want to add. Your access level can be Basic, Stakeholder, or Visual Studio Subscriber. You can add a total of five free, Basic users. Basic users have full access to all Azure DevOps features, such as Boards, Repos, and Pipelines. Stakeholders have limited access, but are always free. They have full access to Pipelines, no access to Repos, and limited access to Boards. You can find more information on stakeholder access here. Visual Studio subscribers already pay for access through a professional or enterprise subscription. They have full access, but do not take up a free user slot.

You can also add users to projects if you have any. If you select your sample project, you can also add your new users to a group in your chosen projects. The groups are Project Readers, Contributors, and Administrators. These groups are self-explanatory, so I will not say too much about them. Just be careful who you give administrator access to. A contributor can access everything in a project, but cannot set certain settings (but still more than you might want them to).

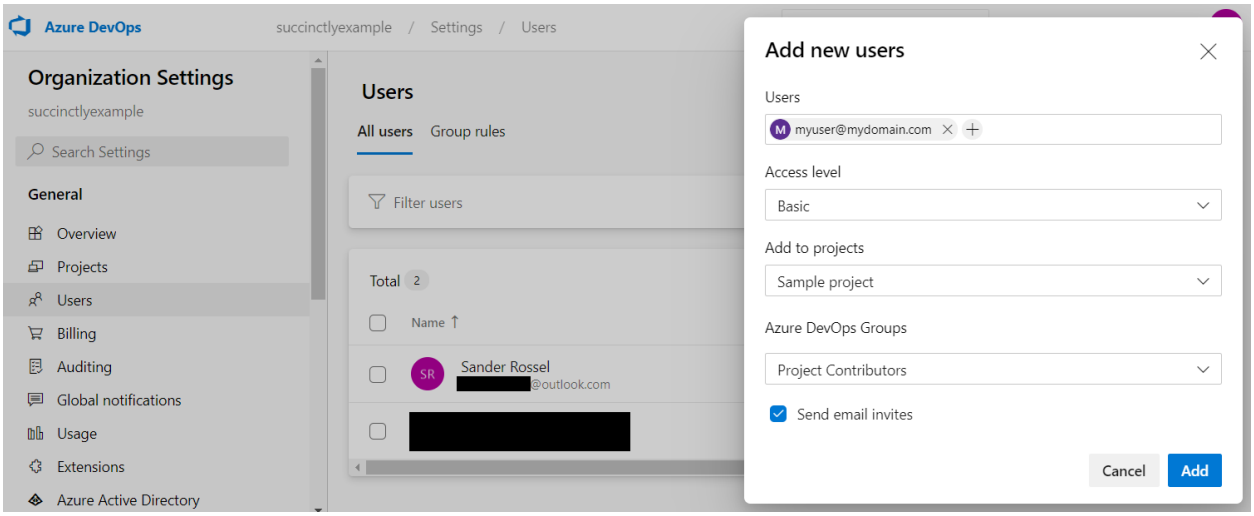

Figure 9: Add a New User

Add a user with Basic access and add them to the Contributor group for your sample project. Leave **Send email invites** selected. You should receive an invitation in your email shortly (be sure to check your spam folder as well). Click **Join now** in the email to get started.

We are going to leave the new user for now, and continue with our current user.

The new user is added to the user overview. You can click the three-dot button (**…**) at the end to change their access level for the company or for individual projects, resend the email invite, or delete the user. If you log in with the new user, you can still access the user overview, but you cannot access the three-dot button to manage a user or invite new users.

Billing

As mentioned, you can add five free users to your account, and unlimited stakeholders and Visual Studio subscribers (because they already pay for an account). However, you get more than just five users. Later, when we are going to add CI/CD pipelines, we need build time as well. The free tier offers 1,800 minutes (30 hours) running time per month and one self-hosted agent. Jobs can only run sequentially. So, if you want to build and release two projects, you will have to wait for the first build, then the second build, then the first release, and then the second. You may be tempted to think a build only takes a few seconds at a time, but unfortunately, they take up minutes on Azure DevOps because the code has to be downloaded every time, as well as any packages that would be cached on your own machine. A typical build can require minutes at a time, and for complex builds and releases you are looking at 10 minutes each. That means that with a single job at a time, you will have to wait for maybe half an hour for two builds and releases to complete. Next to that, you also get artifacts and cloud-based load test limits.

If you want to scale up the number of users, your build time, or the number of parallel jobs, you can do so in the Billing settings. Here you can view your free users, build minutes, self-hosted agents, artifacts, and cloud-based tests, as well as your paid ones. To set up billing, click **Set up billing** and you will see the Azure subscription that is linked to your account. We are currently on a free trial subscription that has a spending limit, so we cannot set up billing. I am not going to ask you to remove the spending limit or upgrade to a paid account since we will not need it, but if you have one it is as easy as selecting an Azure subscription, which is linked to your account. After the subscription is connected, you can just add a number of users, parallel jobs, agents, etc.

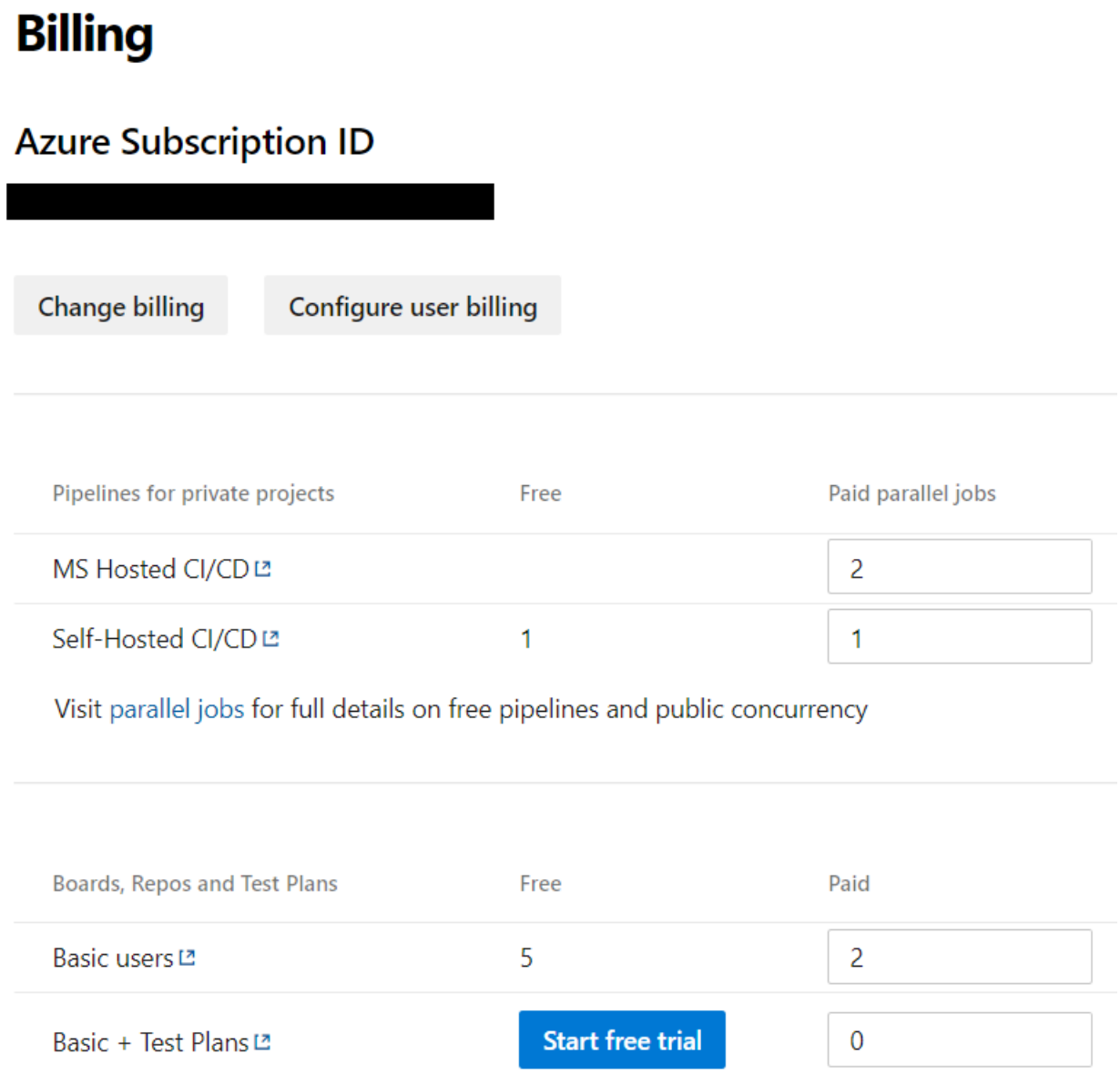

Figure 10: Azure DevOps Billing

Your users can be billed for a single organization or for multiple organizations. So, either a user is billed per organization, or a user in multiple organizations is only billed once. The first is useful when you have only a few users that work for multiple organizations; the latter is useful when many users access multiple organizations. The multi-organization billing applies to the Azure subscription, so you do not get five free users. That also means that all the organizations this user is a part of need to be part of the same Azure subscription. You can change this setting under **Configure user billing** once billing is set up.

You are billed daily, so if you add users and remove them after two weeks, you will only pay for those two weeks. The prices per added resource per month used to be displayed here somewhere, but that has been removed. However, you can view the pricing details here. For the Basic plan, the first five users are free and each additional user is $6 per month, while an extra parallel job costs $40 a month.

To change your billing or remove it entirely, click **Change billing** and either select another Azure subscription, or click **Remove billing**.

Connecting an Azure AD

We have seen that your Azure subscription is already used in billing scenarios. However, you can also connect an Azure AD to your organization. You would want this so you can manage users in one place (Azure AD), although you still have to assign their access rights in Azure DevOps by assigning roles to users. In the organization settings, click **Azure Active Directory** in the menu on the left. The following screen cannot be simpler since it only has one button, **Connect directory**. Click that button and you will be able to choose a directory. If you followed along with this book, you will only have one choice, which in my case is the Succinctly Directory.

You will get a warning though: 1 out of 2 members is not part of the Azure AD and will lose access to the Azure DevOps organization. We are going to re-establish access for that user in a second. For now, connect to the Azure AD.

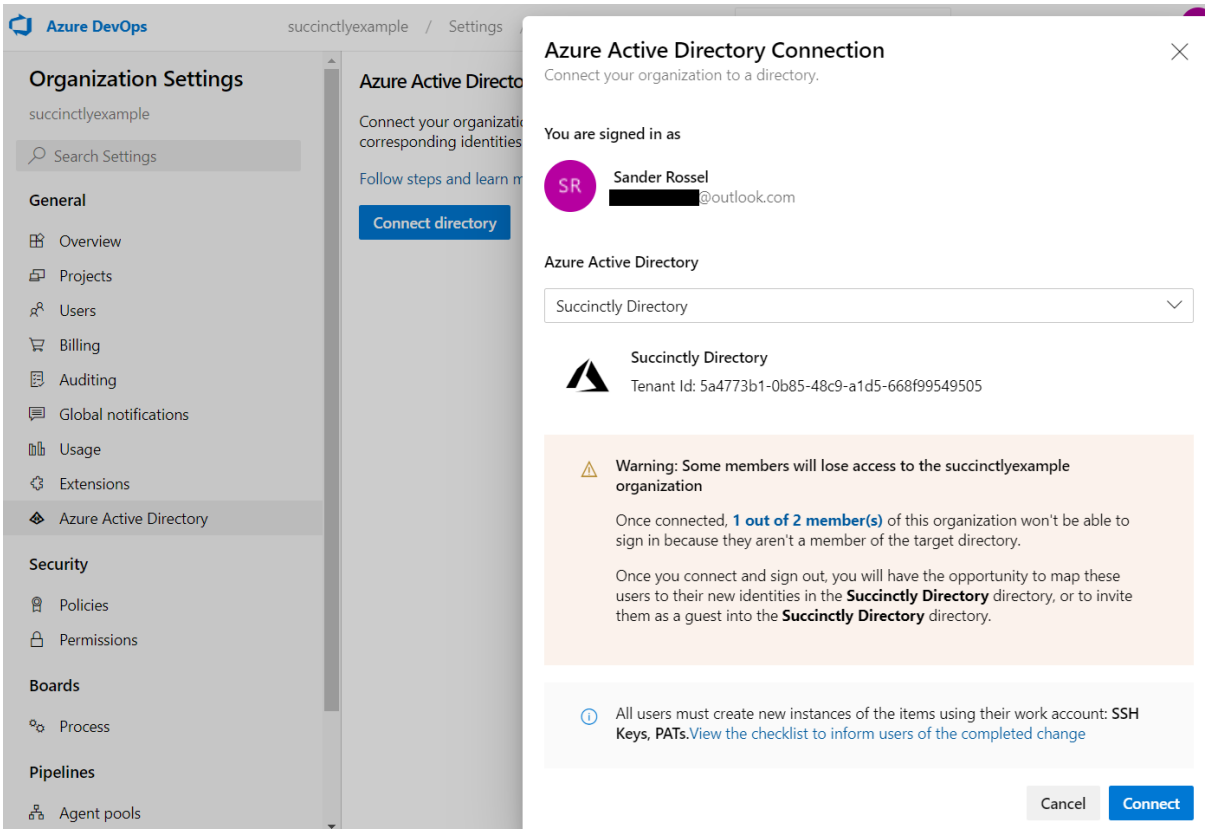

Figure 11: Azure AD Connection

After connecting you will be forced to sign out. For some reason, I was not able to sign back in for a few minutes, so be a little patient. Be sure to sign out completely before signing back in. When you are able to sign back in, you will be asked to verify your identity, and then you will be given access to the organization again. We can now see what I told you before—the other organization that we created is not in the list of organizations anymore because it is not connected to the same Azure AD. It gets a little messy here. On the top right, click your initials and click your account (right above **Sign in with a different account**). You will briefly see **Switch identity**, and you will be prompted for your password again. You will then be taken to an overview of all your organizations, which only shows the other organization! You can click it to enter it. Now go back to your main organization, and you will be denied access. You have to click **Sign out and login with a different account** and log in to exactly the same account. Good job, Microsoft! Did I mention they messed up accounts?

Unfortunately, our second user just lost access completely. Now you can do two things. Enter the user into Azure manually, or delete the user from the organization and re-invite them. First, let us add them into Azure manually. Go to https://portal.azure.com/ and navigate to Azure AD, and then to users. There, click **+ New guest user**. Enter a name and email address, and click **Invite**. They should now get another invite. They must then click the link in the email and provide our Azure AD access to their account. After that they can log in to our Azure DevOps organization again after confirming their identity.

For the second method, we must either delete this user or add a third user to our organization. Deleting the user requires a little extra explanation. In addition to deleting the user in Azure DevOps, we must now also delete the user from Azure AD manually. To delete them from Azure AD, simply go to your Azure AD users again, click the user, and click **Delete** in the menu bar at the top. If you also deleted the user in Azure DevOps, then the user is now officially a stranger again. You can now add the user to DevOps again, just like we did before. The user will receive another invite, but it looks a little different this time. That is because it is an Azure AD invite, just like when we added the user to Azure AD manually. If the user clicks on the link in the mail and accepts the terms, this user is now added to Azure AD and to Azure DevOps.

Another method of adding users is to first add them to Azure AD, which will often be the case if you synchronize Azure AD with your on-premises AD. When the user is already in Azure AD, you can add them to Azure DevOps as well, except this time you get a little auto-complete when entering their email address. Other than that, it is the same process.

Project settings

Next, we are going to look at some project settings. These settings can be set for individual projects. Browse to your project and click **Project settings** in the lower-left corner (in the same place as your organization settings). You will be taken to the overview page where you can change the name and description of a project as well as the visibility. You can also change the available services here. If you decide not to use test plans, you can completely disable it here and declutter the UI a bit. Also, you can add project administrators and completely delete the project.

We will look at teams and permissions in the next section.

You can also add service hooks that allow you to integrate with other services. You can notify various services of events that happen within your project, like code being pushed, a build triggering, or a work item being updated. Not every service supports all events. Other than that, service hooks are outside the scope of this book.

The last setting I will describe here is that of dashboards. It is possible to add dashboards for your project. There are only three settings and they are well described, so I will leave it at the mention of them. However, go back to your project overview page and you will see **Summary**, **Dashboards**, and **Wiki** in the menu. By going to **Dashboards**, you can add widgets like assigned work items and burndown charts. Just click **Edit** at the top to add some widgets, and simply drag them in the boxes. You can also create new dashboards by clicking on the down arrow next to **Sample project Team - Overview**. You can now view various projects statistics, like failing builds and deployments, or burndown charts, with a few clicks.

Last, you can add a wiki by going to the menu item with the same name. Here you can add pages on important issues in your project, like application architecture, code structure, best practices, or jargon that everyone should know about. The actual creation of a wiki is not in scope of this book.

Security

Security is a critically important aspect of Azure, Azure DevOps, and any cloud service. Azure DevOps has some security settings as well. In the organization settings menu, you will see a **Security** heading with two subitems, **Policies** and **Permissions**. Policies are organization-wide settings. You can click on the little link icons to get more information for each setting. You might want to disallow public projects, so no one accidentally creates a public project with sensitive customer data. Turning off external guest access can also be a good policy. This forces you to add all new users in your (Azure) AD first so people cannot use personal emails. This is not very practical when you work with a lot of contractors, but may add extra safety. Personally, I never needed these policies.

The Permissions tab is a little more interesting. Here, you can view, change, add, and delete groups, and assign users. A group allows or denies certain permissions to the users that are assigned to the group. There are a couple of built-in groups, of which **Project Collection Valid Users** and **Project Collection Administrators** are the most important. The first gives the least access and contains every other group, and by extension every other user, while the latter is reserved for administrators. When you click **Project Collection Administrators**, you will notice that you cannot edit it. The **Project Collection Valid Users** can be edited as you like. It is advised to not edit the built-in roles, and to create new roles if these built-in ones are insufficient. Keep in mind that these roles give or deny access on the organization level.

When you click **Project Collection Valid Users** and then check the **Members** tab, you will see all other roles in your organization, as well as the **Project Valid Users** role on the **Sample project**. If you click the role and check out the members, you will notice the **Readers**, **Contributors**, and **Project Administrator** roles that we could set when adding new users. If you drill down further to the **Contributors** role and its members, you will find the user that we added. You can change these roles by changing permissions, adding a group image, or changing the name or description. You can also delete these groups. Unless you know what you are doing and really want to spend some time figuring this out, I suggest you leave it alone. At most, add a new group.

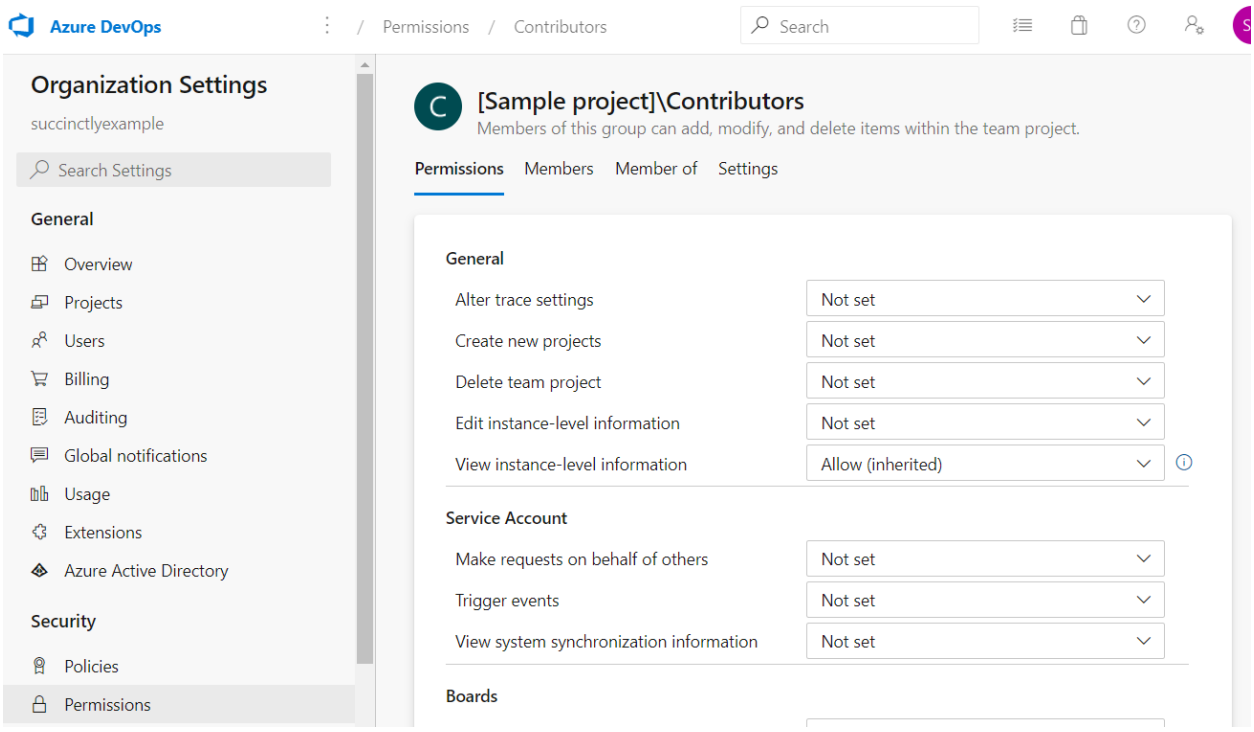

Figure 12: Contributors Role

If you go back to the permissions screen of the organization, you will see a warning that you only have one Project Collection Administrator, and that you should add a second one to reduce the risk of losing access. You can add your other user by clicking the message, or you can add it to the group directly. To do this, click on the group, go to members, and click **Add** in the upper-right corner. Simply find your user and click **Save**. If you now log in as the other user and go to the user overview, you will find that you now have the rights to add and change users. Other than that, this user can now disconnect Azure AD or connect another one, add and change groups, and even remove you as an administrator and owner. Basically, this user has unlimited power—so be extremely careful who you assign to this role!

Notice that by drilling down the members of members of roles, you will eventually end up in roles that are specific to your projects, like the Contributor and Reader roles. You can find these project-specific roles by browsing to your project settings. Here you can find **Permissions** as well, under **General** settings. Next to groups and users, this level also has teams. You can find **Teams** in the menu as well, right above **Permissions**. A team is a collection of users that you can assign to a role. Every user in that team will then have the same roles, unless someone in the team has some specific roles assigned to the user account.

While permissions are important, this is also advanced stuff, and most teams can use the default provided roles. I'm not going over how to create custom roles, but feel free to experiment with it at your leisure.

Other settings

We have looked at the most important settings. Some settings that I have not mentioned yet are the auditing and usage logs under organization settings. This is where you can view who did what and when. This is particularly important when something goes wrong and you need to identify the users involved. There are also global notifications under organization settings, and notifications under project settings. These are the same, but in a different scope. This is where you can configure when certain notifications are being sent. I would leave that alone as users can override them individually as well (in the upper-right corner, next to your initials, under **Notifications**). Speaking of which, that is where you can change other personal settings as well, such as your theme, time zone, and profile. (I think it is nice to add a picture of yourself there so you do not get all those boring initials.) I will explain the personal access tokens, SSH keys, and alternate credentials briefly in a later chapter.

Another important organizational setting I have not discussed is extensions. Here you can browse the Marketplace for extensions other people created that should make life easier while using Azure DevOps. I would not advise you to add unofficial extensions to Azure DevOps, but some of them can be handy. There are extensions for Slack integration, SonarQube, and something as simple as code search. Some are official Microsoft extensions; others are official extensions from other companies, and some are made by users who are not backed by any (large) company. Most are free, but some are not. Browse at your own leisure and use at your own risk. Simply click on an extension and click **Add for free** or **Get**.

Other settings that you may be wondering about, like repos and pipeline settings, will be discussed in the upcoming chapters.

Summary

Azure DevOps offers plenty of possibilities for project, user, and group management. By using groups and permissions, you can secure your environment. By creating different projects or even different organizations, people and teams can work independently according to their own needs. By connecting to Azure, you can use Azure AD for user management and your Azure subscription for billing for necessary upgrades. Azure DevOps can provide value for teams of any size.

Chapter 3 Boards

In the previous chapter, we created a DevOps account, an organization, and a project. A project comes with a default Kanban board. In this chapter we are going to use and configure various boards and see how they can be used by teams to improve productivity.

Terminology

The first thing we must establish is some terminology. If you go to your project and click **Boards** in the menu, you will be taken to a page with recently updated work items. Of course, this page is empty because we have no work items yet.

We can create a new work item by clicking **+ New Work Item** at the top. We then get to pick between Epic, Issue, or Task.

We will get to their meanings in a second, but first let us look at what other types of work items we have. Go back to the organization page and create a new project. This time, open the advanced settings and choose **Agile** as your work item process. Name it **Agile project**. Do the same for **Scrum**. When you look at the different work item types for the different processes, you will see a lot more types.

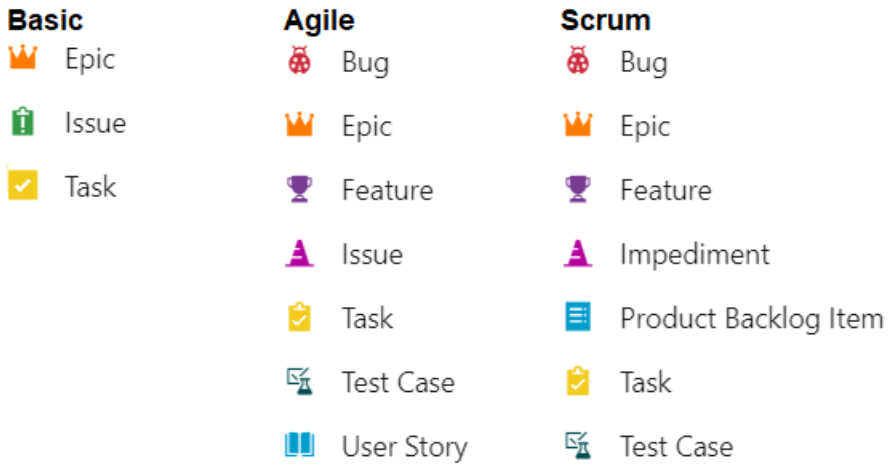

Figure 13: Work Item Processes

The CMMI (Capability Maturity Model Integration) type has some extra types as well, but I am not going to discuss CMMI in this book. Instead, we will focus on Basic, Agile, and Scrum. I am going to discuss the methods next, but this page has some great visuals to better understand the various methods.

Note: The Agile and Scrum methodologies are not in the scope of this book. Instead, I will give you a basic understanding so that you can put the Azure DevOps tooling into perspective. You can read other books on Agile and Scrum, and even

become certified, so that should give you a bit of an idea of the breadth of these methodologies. That said, I have experienced that every organization uses Agile and Scrum differently. Some organizations are just messing around and calling it "Agile," and others do it purely to satisfy stakeholders because Agile and Scrum are hot and happening. Some teams have the certifications and follow the rules religiously while other teams interpret the rules more loosely. Whatever you do, the information in this chapter should help you become proficient with Azure DevOps Boards, whatever methodology you (strive to) use.

Backlogs

Before we continue, let us discuss backlogs for a minute. A backlog is simply a list of work that needs to be done. Project managers, product owners, and/or business analysts write epics, issues, and other work items, depending on their process, and place them on a backlog. There is a hierarchy between the various work item types. In the Basic process, epics contain issues, and issues contain tasks. Say you have a product and you want to add a web shop; this could be an epic. Another epic is that you want a wiki for your product. These epics are placed on the backlog and can be prioritized.

Within the epic you have features and/or issues. These can be prioritized as well. After that, the developers know what they should work on next, and can start creating tasks for work they think needs to be done. A task can be simple, like "On the homepage, add a link to the web shop." The effort required to complete these tasks or issues can then be estimated. Once everything is estimated, a manager should have insight into how much effort a feature or epic is going to be.

This process should also make clear how many developers can work on a single task. If your team has four developers and only two can effectively work on the web shop, the other two can work on the wiki simultaneously.

Estimating

Individual issues, and especially tasks, should not be too much work. The more work, the harder it is to give accurate estimates. If your estimate is too big, consider breaking up an issue into multiple issues. Estimates are often given using points equal to the Fibonacci sequence (the next number is the sum of the previous two, so 0, 1, 1, 2, 3, 5, 8…) or in t-shirt sizes (XS for extra small, S for small, M for medium, and so on). In theory, your estimates should be relative to some base issue. Pick a small issue and say, "this is a 2 or an S" and then estimate whether the next issue is less work (a 1 or an XS) or more work (3, 5, M, L). In practice, people often measure in hours, for example S, or 2, equals two hours, but this is not how it should work. Of course, if it works for you, be my guest.

Sprints

All processes support sprints. A sprint is a period, usually two or three weeks, in which work items will be completed and deployed. With a full backlog, teams can plan their sprints. The top-priority issues will be placed in a sprint, after which teams can estimate the effort it takes to complete an issue or the tasks in an issue. Teams will estimate how much work they can do in the next sprint. If people are on vacation or there is a holiday coming up, you know you will be able to deliver less work that sprint, and you can plan accordingly. If your estimates are accurate, teams will be able to predict how much work they can finish in a sprint. So, if they estimate the top 10 issues and the next sprint has room for the top five issues, the next five can be moved to the sprint after that. Teams can now give estimates, like the web shop epic will take three sprints to implement.

Once a sprint is planned, the team commits to it and it is more or less set in stone. Of course, if any issues turn up, like bugs in production, there should be room to solve those as quick as possible. This will probably lead to fewer points or shirt sizes being fixed that sprint, which helps in keeping track of trends. In some cases, sprints are dropped entirely because something else just took priority. This could be the result of some crisis (COVID-19, at the time of writing), a critical bug like a data breach, or a change in law. In my experience though, it is usually the result of bad management.

By working in sprints, you can plan ahead, but more importantly, you can plan for change. You only plan two or three sprints ahead, so if priorities or specifications change, you can pick up these changes in two or three weeks at most, depending on the length of your sprint. It is recommended to finish your current sprint, so you are not left with half-finished work. Also, constantly changing priorities can cause stress, which has a detrimental effect on the team and office atmosphere.

To sum it up, we have work items, which are placed in backlogs, estimated, and then planned into sprints. Now let us look at the individual processes. While they have a lot of similarities, there are some important differences between them as well.

The basic process

The basic process is a simple one. It has three work item types: epic, issue, and task. As stated before, there is a hierarchy between them. An epic has issues, and an issue has tasks.

Note: The term "epic" is derived from Agile terminology. Originally, an epic was two or more Agile stories (which makes sense), but at some point, the meaning of the two terms switched, leading to a non-obvious meaning of epic.

Let us start by creating an epic. In Azure DevOps, go to your work items of the first project we created. Click **+ New Work Item** and select **Epic**. In the screen that opens, enter **Read Azure DevOps Succinctly** as the title, assign yourself, and then click **Save**. If you now go back to your work items, you should see the epic in the overview.

Next, create a new Issue. Enter **Read Chapter 3** as a title and assign yourself. On the right side, look for **Related Work** and click **+ Add link** and choose **Existing item**. Select **Parent** for **Link type** and select the epic we just created for **Work items to link**. By linking the issue to the epic, we are saying that this issue is a part of the epic. You can mess this up by making the epic part of the issue instead, so be sure to make the proper link type. We can circumvent this issue by creating issues directly in epics, like we are going to do next with a task. It is good practice to add a description to your card unless the title makes everything clear to everyone, but it hardly ever does.

If you work in an Agile or Scrum process, you should also enter the Acceptance Criteria, that is, when a story is considered done. For example, Acceptance Criteria can be “A user can register using email, password is stored encrypted with salt.” Do not enter any effort or story points yet; those should be discussed with the team when the items are planned for the next sprint. You get updates on items you created or are assigned to. If you want to get updates on other items, you can click **Follow** in the upper-right corner. Click **Save** again and return to the work items.

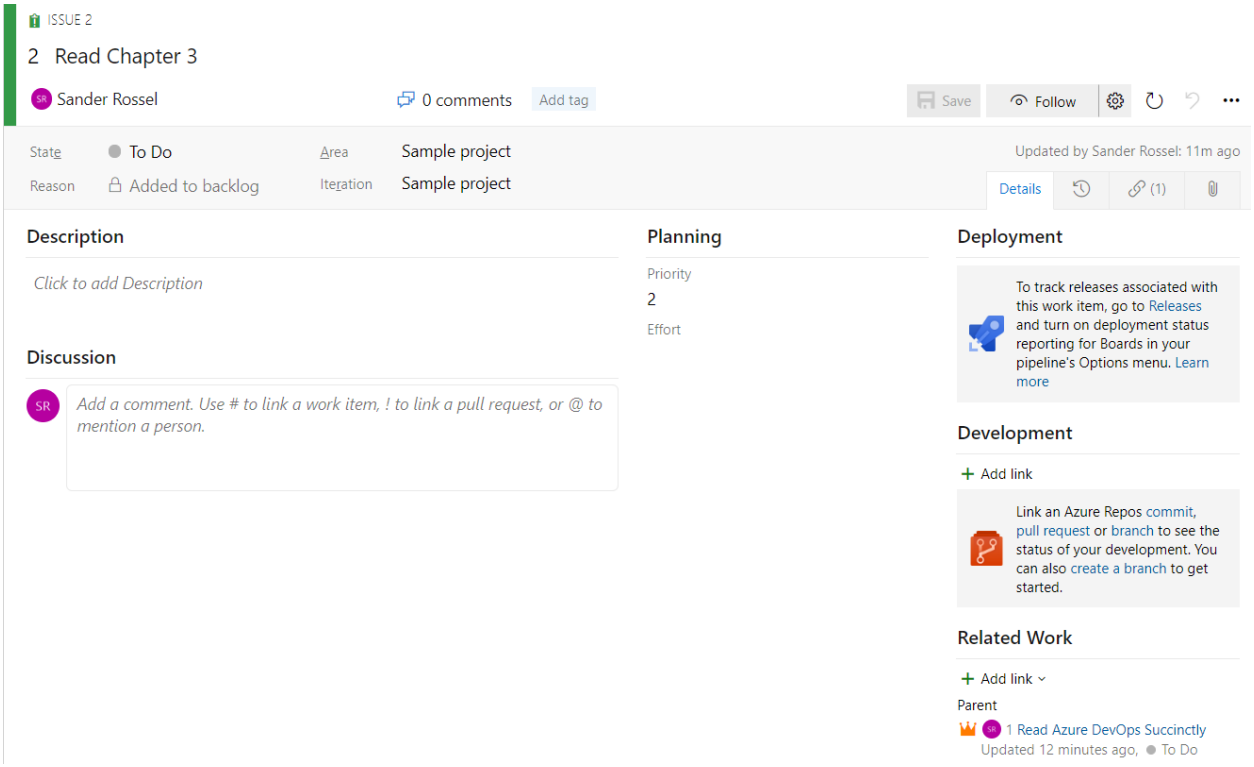

Figure 14: Issue Creation

Next, we are going to create a task as a part of the issue, but we are going to do that a little differently. Open the issue again and add another link, but this time choose a new item. The link type will be Child, and the work item type will be Task by default. Enter **Read terminology** as a title. A new window will now open for the task creation. Everything is fine for now, so just save and close.

If you now look at the work items, you just see a list with the three items we just created. However, when going to **Backlogs** in the menu on the left, you will see your issue, and when you click the little arrow in front of the title, it will show the task as well. In the upper-right corner, you can set the view from Issues to **Epics**, and the epic will be shown too.

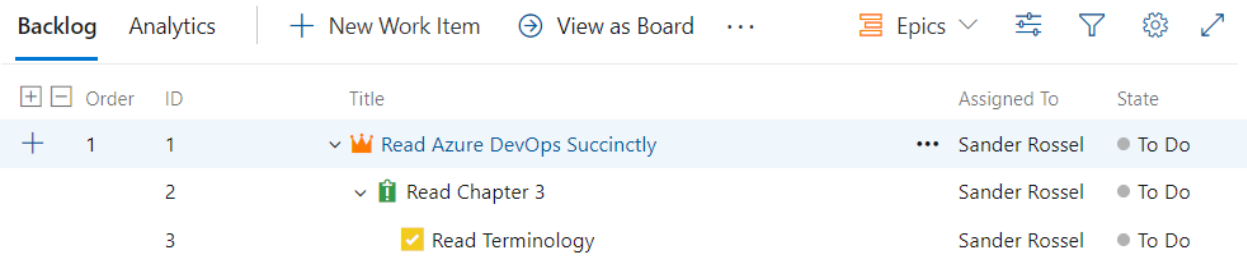

Figure 15: Backlog

Next, click **Boards** in the menu. This is where you will be spending a lot of time when developing. You see three columns: To Do, Doing, and Done. You can view issues or epics on the board, although I always prefer issues because you will be working on issues, not epics. The squares (or rectangles) of your issues are called cards. The issues have tasks that you can show or hide. You can create new issues and tasks on this board as well.

You will probably discuss with your team who is going to work on what issue, but sometimes everyone is busy, and some issues are still left untouched, and you can pick an issue to work on. To let the team know you are working on an issue, simply click and drag it to the next lane. You can then check off the tasks that you have completed.

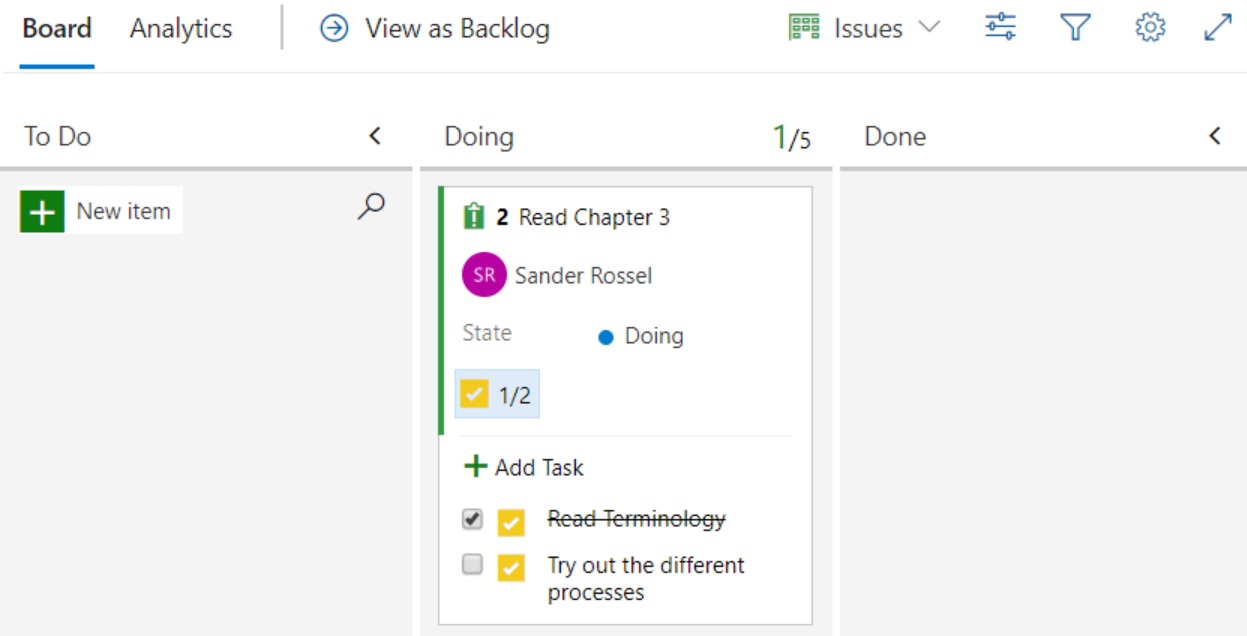

Figure 16: Work Items on a Board

You may notice 1/5 is in the Doing column. Generally, a team should not have more than five developers, and each developer should not work on more than one item at a time. Of course, when you are done with the issue you can move it to the next column.

This example has three columns, but you will often see more columns. For example, you may see a Test column where developers move their finished issues so testers can pick them up and move them to a Testing column. I have also seen a Deployed column, which indicated whether the issue was deployed to production. The number and name of columns is not set in stone, and you can add or remove them as you wish, which we will see later. In fact, you can already see different columns in the Agile and Scrum processes.

If you open a card, you can start a discussion. For example, if you have a question for someone regarding this card, you can ask your question and mention them using @. For instance, "@Sander Rossel Do you want this in blue or in pink?" Mentioned people will get an email. The mentioned user can then respond, "I want this in pink" and it is now documented for everyone to see. You can also use # to link to other work items or ! to link to pull requests, which we will discuss in the next chapter.

You can use this board as you see fit. Maybe you do not need epics or tasks and simply want to work with issues only. Feel free to do so.

Because you have some work items assigned to you, you can now go back to the organization home screen and select the **My work items** tab. You should see the epic and issue assigned to you as well as the status and your activity, like "Updated the Read Chapter 3 issue."

The Agile process

Let us look at the Agile process next. This works mostly the same as the basic process in that you can make work items with a certain hierarchy, put them in your backlog, show them on the board, and drag them between columns.

At the top of the hierarchy, we still have the epic. An epic consists of features, which consist of user stories. The user story is equal to the issue from the basic process. Tasks remain the same, under User Stories. The issue in the Agile process is something different, which we will look at in a bit. A user story can have tasks, bugs, and tests.

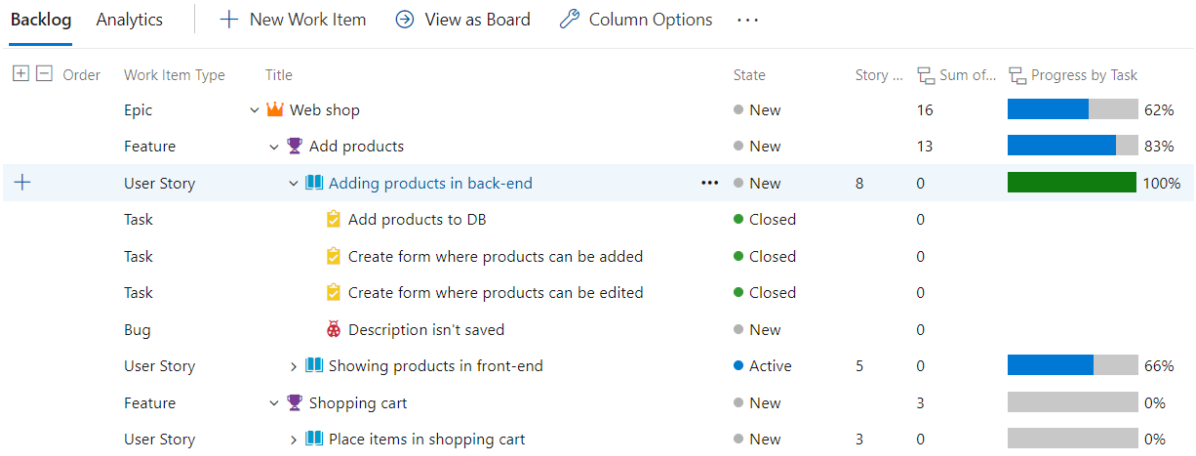

Figure 17: Agile Backlog

Tests and issues are not shown in the backlog, but tests are shown on the board. Tasks, bugs, and tests are part of user stories, and are shown in the user story card. Creating a test also creates a test plan, but we will get to that in another chapter.

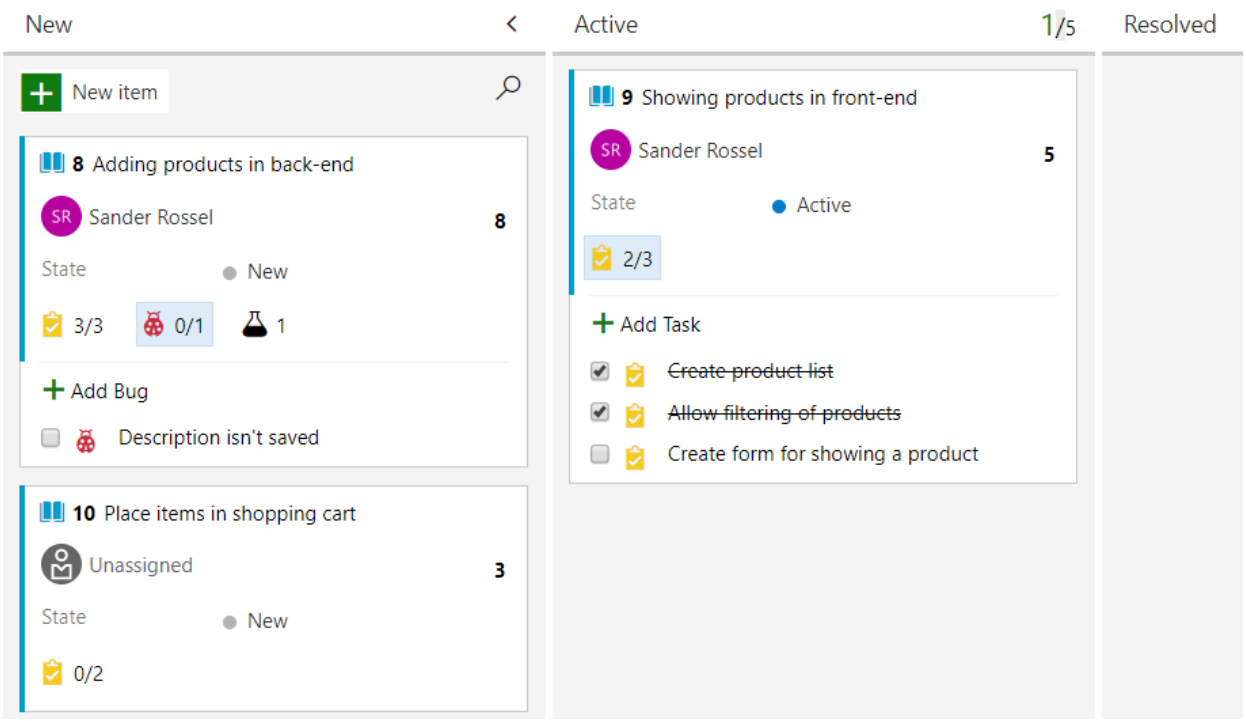

Figure 18: Agile Board

Queries

We still have the issue work item. An issue is something that blocks planned work or a release, such as explicit permission required from a manager, or waiting for a supplier. In this example I have created an issue that the server capacity is insufficient for the new load we expect from the web shop. The problem is, we cannot see the issue in the backlog or on the board. For this, we can create queries. A query is a filter over your work items. For example, you can have a query for the current sprint, for just your items, for all resolved items, or for items that are (not yet) estimated. Queries are useful to get various overviews and can be personal or shared with the team.

Go to **Queries** in the left-hand menu. Your default view will be **Favorites**, but if you switch to **All** you will see two personal queries, **Assigned to me** and **Followed work items**. Open **Assigned to me** and see the results. Clicking on **Editor** shows the actual query, which is **Assigned To | = | @Me**.

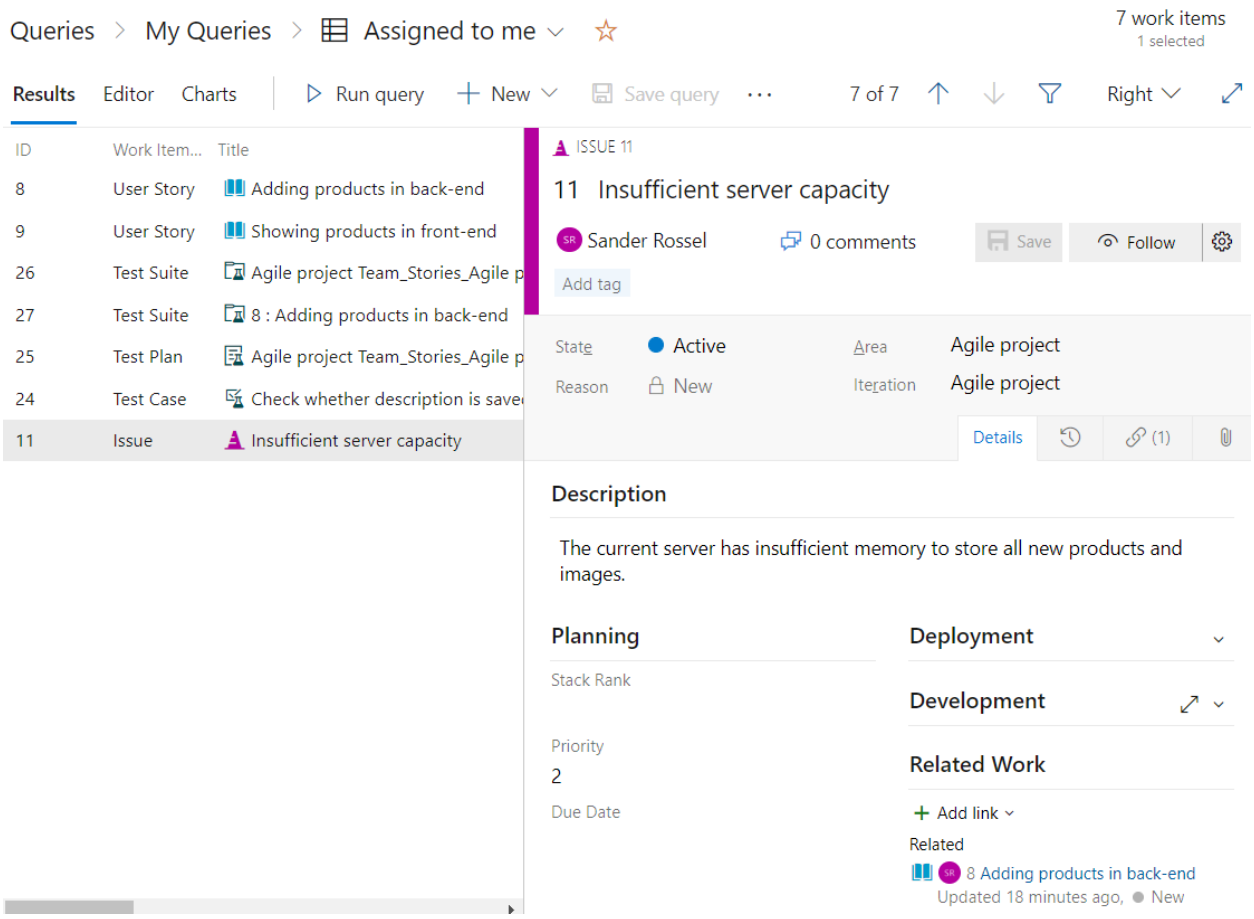

Figure 19: Assigned to Me Query

Go back to the query overview and click **+ New query**. The default query is any work item type and any state, but let's change any work item type to **Issue** and run the query. You should now see your issue. However, we also want to filter out any issues that have been solved, so change the state filter to **State | Not In | Closed,Completed,Inactive,Removed,Resolved**. Save the query in the **Shared Queries** folder and name it **Active Issues**.

The Scrum process

Last but not least, we have the Scrum process. The Scrum process looks a lot like the Agile process. In fact, just switch "user story" with "product backlog item" and "issue" with "impediment," and you are pretty much there. There are some differences though. The first is that a bug is now managed with requirements instead of with tasks. That means a bug gets a separate card on the board and is visible in the backlog.

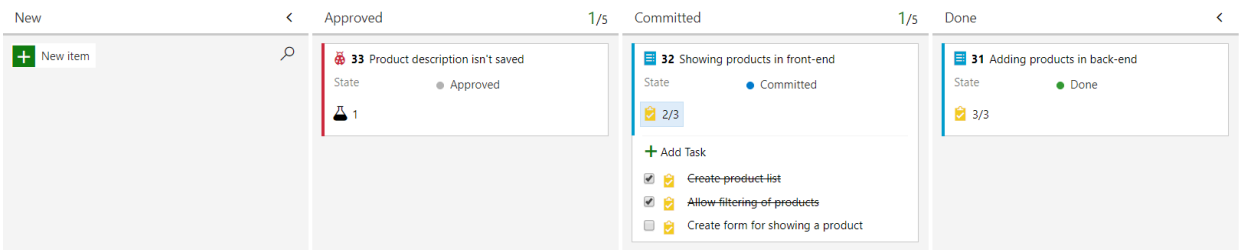

Figure 20: Scrum Board

In this example, the bug relates to the **Adding products in back-end** work item, but since it is planned separately, we can close the work item. As such, it is not shown in the backlog anymore.

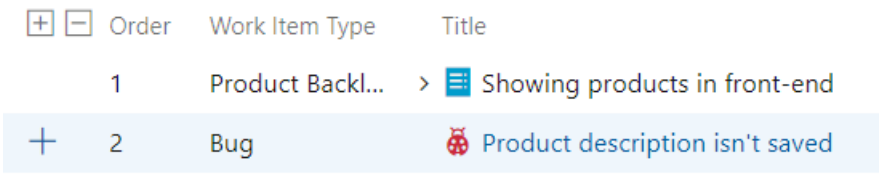

Figure 21: Scrum Backlog

Again, impediments are not shown on the board or on the backlog, but you can make custom queries to find them.

Customizing the board

We have now seen the Azure DevOps board defaults, but you can customize the board, items, and backlog. Go to your board in the Scrum project and click the cog icon in the upper-right corner; this should open the settings page.

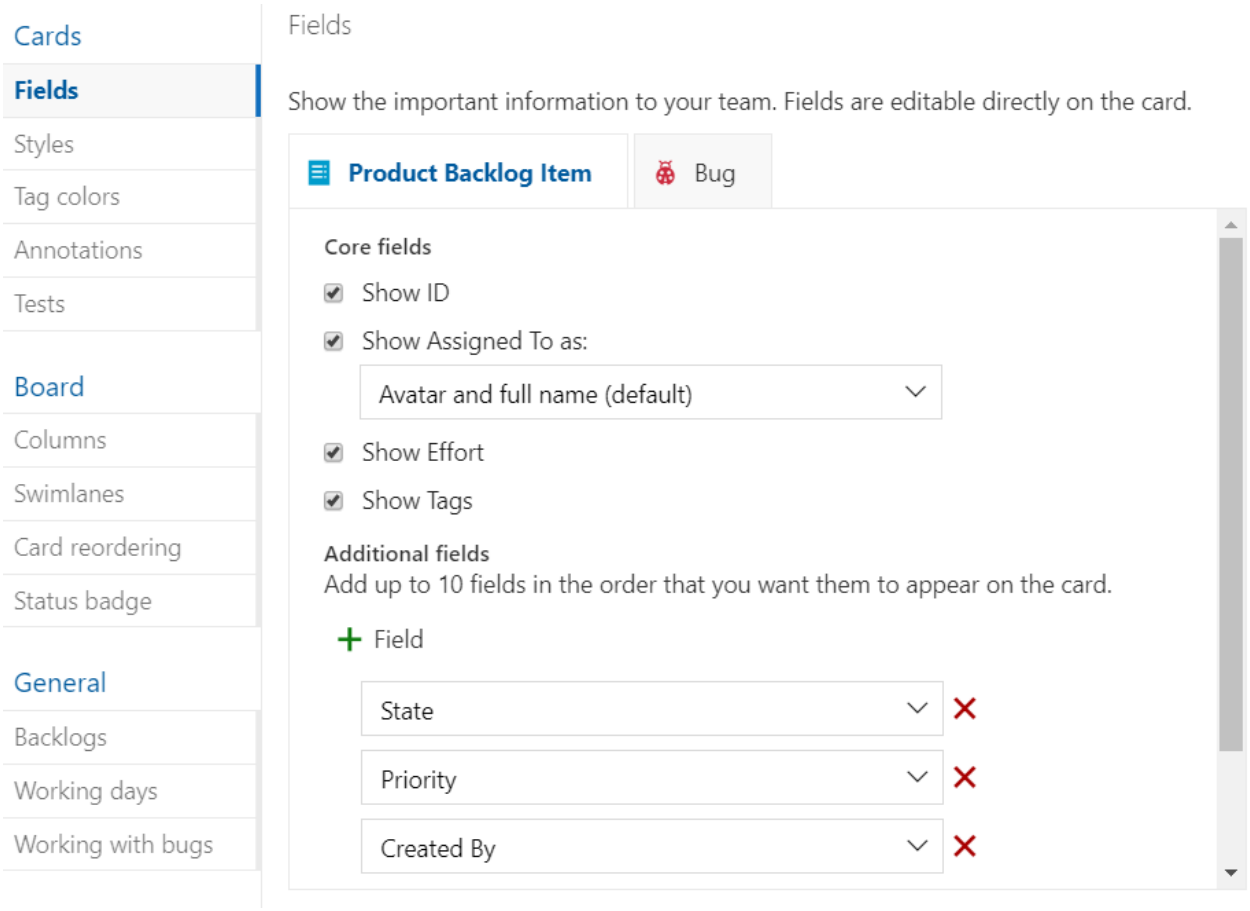

Figure 22: Board Settings

First, you can customize cards. You can change the information that is shown on your cards. These can be issues, user stories, or product backlog items and bugs, depending on your type of project. For example, you could add the fields Priority and Created By. Next, you can change the style of your card if some criteria are met. For example, enter rule name **Red for blocked**, pick some red color for **Card color**, and in the criteria add **Tags | Contains | Blocked**. Now go to a card on the board and edit it. Next to title, you can add a tag, so add **Blocked**. The card should now be red. You can also add colors to specific tags. We already added a Blocked tag, so add a tag color for Blocked, and give it any color you like so it stands out. Under **Annotations** you can specify if you want tasks and tests to be visible on the board. I will return to test settings later.

You can also customize the board itself. You can add and delete columns. For example, add a column and name it **Testing**, set the WIP limit to **0**, and set the state mapping for both bugs and product backlog items to **Committed**. Drag the column between Committed and Done. Swimlanes are like columns, but horizontal. There is a default lane that must always be there, but we can add a new line. For example, instead of creating a column called Testing, we could have added a Testing lane. That way when developers are done working on an item, they can move the item to the Testing lane, and testers will know they can start testing the item. Card reordering defines whether items on the board are ordered according to the backlog, or whether the backlog is ordered according to the board. I will come back to the Status badge later.

In the **General** settings we can set the backlog navigation levels, working days, and how we handle bugs. We have seen the **Bugs are managed with requirements** setting in the Scrum process, and the **Bugs are managed with tasks** setting in the Agile process. By selecting **Bugs are not managed on backlogs and board**, bugs become invisible, much like issues in Agile or impediments in Scrum.

Working in sprints

Now that we are comfortable with using the board and backlog and creating work items, we can start planning our work. The Scrum project will have six sprints planned by default. You can view these sprints by going to the **Sprints** menu item. You should see the taskboard, which says “You do not have any work scheduled yet.” Go to your **Backlog** (in the left-hand menu, not the top menu), and you should see your sprints on the right side. Simply grab a **Product Backlog** item and drag it into **Sprint 1** on the right. Alternatively, click the three-dot button (**…**) when hovering over a work item, select **Move to iteration**, and select **Sprint 1**. Make sure you have one or two open tasks in the item. Now, if you go back to **Sprints** and check your backlog there, you should see the work item in that sprint. The Sprint 1 card on the right should also show the planned effort in that sprint. The real value is in the taskboard, though. This should now show you all planned work items and their tasks, which you can now drag between columns individually. As a developer, this is where I spend most of my time.

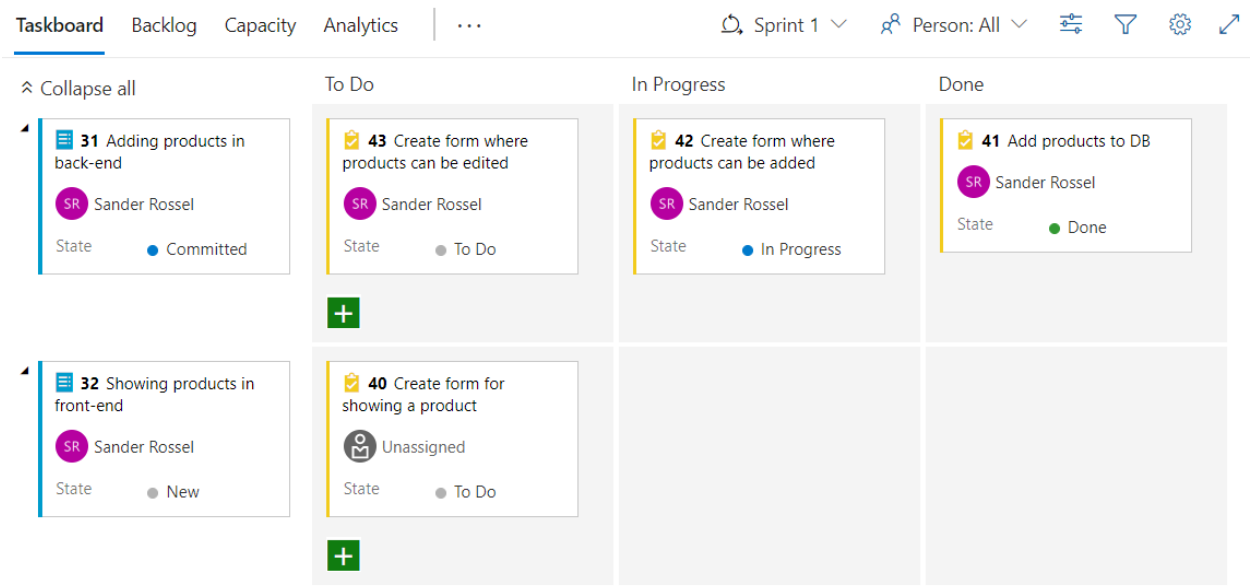

Figure 23: Sprint Taskboard

You can sort items on the taskboard by sorting them in the (sprint) backlog. In the top-right corner, you can switch between sprints. You can also create new tasks on the taskboard directly.

Under the **Capacity** menu, you can plan your team's capacity, like how many hours every team member can spend on various tasks, such as development and design. This helps you in planning work and makes for more accurate reporting. Enter **8 hours** for **Development** and click **Save**.

Under **Analytics**, you can view your **Burndown Trend**. This is the amount of effort left in a sprint. Unfortunately, the page is not currently showing us anything. We can fix that by giving our sprints start and end dates. Go to your project settings and navigate to **Project configuration**. You will see your sprints, and if you hover over Sprint 1 you should see **Set dates**. Click that and set the start day to a few days back, and set the end date at two weeks after the start date. Do the same for Sprint 2, and set the start date a day after the end date of Sprint 1. For example, my first sprint starts on a Monday and ends two weeks later, also on a Monday. My second sprint starts on Tuesday and ends on Monday two weeks later. Every next sprint then starts on a Tuesday and ends on a Monday. You are free to set different start and end dates, like maybe Friday and Monday or Tuesday and Wednesday.

Go back to your burndown chart; if you have entered your team's capacity, it should show your remaining capacity per day. It will also show that you have some items that are not estimated. These are your tasks that have not been estimated yet. Normally, you would do that up front, but let us estimate them now. We are going to fake some estimates, of course, so let us say 8 per task entered in **Remaining work**. Set the **Activity** to **Development** except for one, which you can set to any other activity.

If you now go back to the chart, it looks a bit weird. First, we said our capacity was 8 hours a day. Then, we dragged some tasks into the sprint that were not yet estimated. Last, we estimated the tasks, but only for three or four days of work in a 10-day sprint. The only way to fix this is by setting the start of the sprint to today, so it does not look like you added lots of work halfway in the sprint (although this will also happen in practice). It may take a few minutes for the work to show up in the burndown chart.

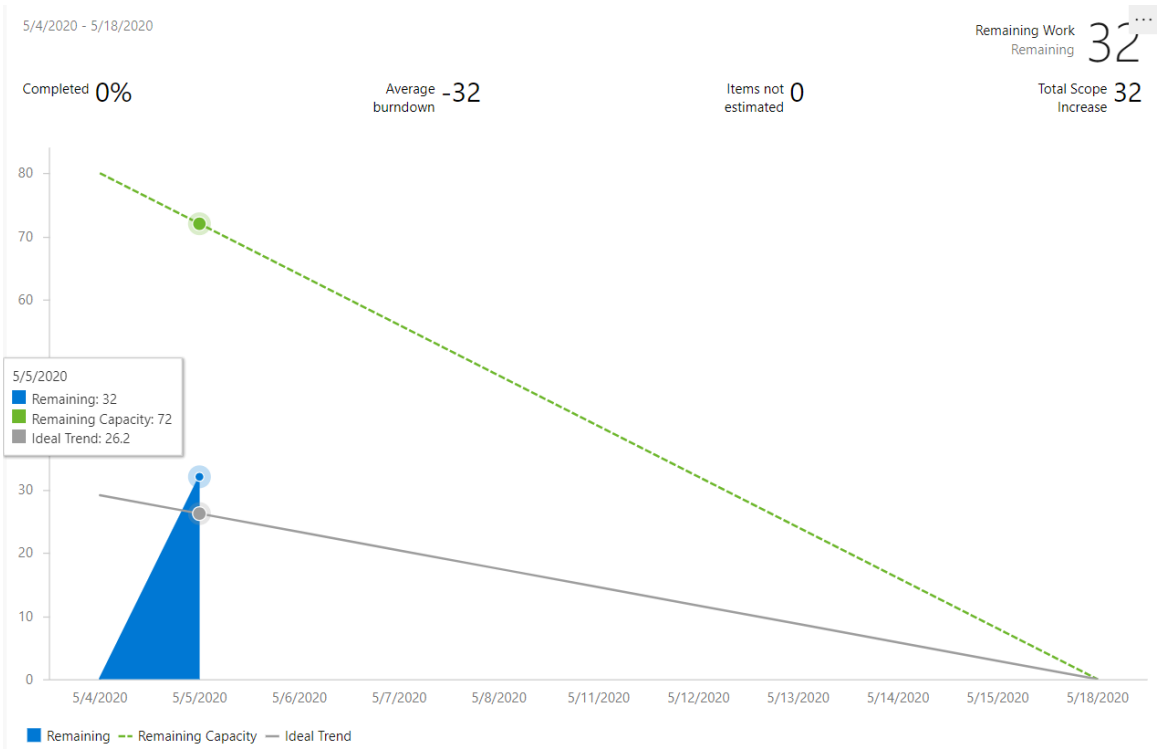

Figure 24: Burndown Trend

In a perfect sprint, the green, gray, and blue lines are equal, but I have never seen that happen. The blue line often starts a bit above the green line, planned by either an optimistic or an oppressed Scrum team, and the gray line starts a bit above green too, of course. After that, the blue line goes down quickly because people do the easy tasks that have been overestimated first. After that people realize they need something done in this sprint, so the blue line goes up a bit again. Then the blue line starts to go down slowly because people have issues with code, and tasks take longer than expected. At the end of the sprint, people quickly commit their work and update their tasks, so the blue line drops to near bottom, but not completely. The remaining work is taken into the next sprint. I am just saying, you are having a good sprint if the lines are near each other.

In the sprint planning, you can create child items. This is useful if you estimate that you need multiple sprints for a specific feature. You can create an iteration of a month and create two sprints within that iteration.

Sprints automatically end and start at the specified dates.

Customizing your process

It is possible to customize your process. Go back to organization settings and find **Boards - Process** in the menu. Here you will find your processes and the fields that are used in those processes. You cannot customize the built-in processes, but I would not recommend it anyway. Instead, hover over a process, click the three-dot button and select **Create inherited process**. Inherit the **Basic** process and name it **Basic incl. Supplier**, and in the **Description** put something like **A simple, basic process that supports collaboration with suppliers**. Now click on the name of the newly created process, and you will see the various item types. Click on an item type to study how it was set up. Then go back and click **+ New work item type** and enter **Supplier Issue** in the title. You can also change the icon and the icon color if you like.

In the next form, you can customize your task any way you want. Hide **Deployment**, **Development**, and **Related Work** (in the right column) from the layout. Click **Add a field** in the middle column and name it **Estimated Delivery Date**, and then select **Date/Time** as **Type**. Now change the name of the new middle column to **Planning**. Add another new field and select **Use an existing field**, and then select **Acceptance Criteria**. This field will be placed in the first column, under Description.

You can add new field groups, add multiple pages of fields, and create rules and states. You can create other fields too, like **Ticket ID**, and add them to existing work item types (in inherited processes), like issue or task. Any new field you create becomes visible in the **Fields** tab under the **Process** settings. You can delete any custom fields there, but they will also be removed from your work items.

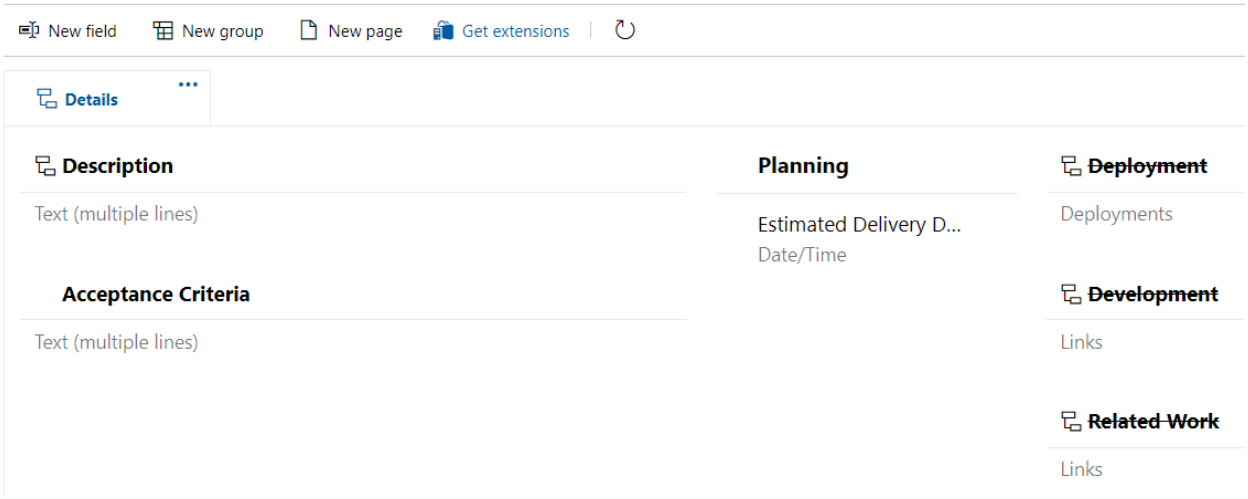

Figure 25: Custom Process

Once you are done customizing your new work item, create a new project and select the **Basic incl. Supplier** process. Go to your project board, add a new work item, and select the **Supplier Issue**. You will see it has a description, acceptance criteria, and an estimated delivery date. Unfortunately, like impediments, the new work item type does not show up on the boards or backlog, but we can change that too. Go back to your custom process and click the **Backlog levels** tab. Under **Requirement backlog**, edit the **Issues** backlog and then add the **Supplier Issue**.

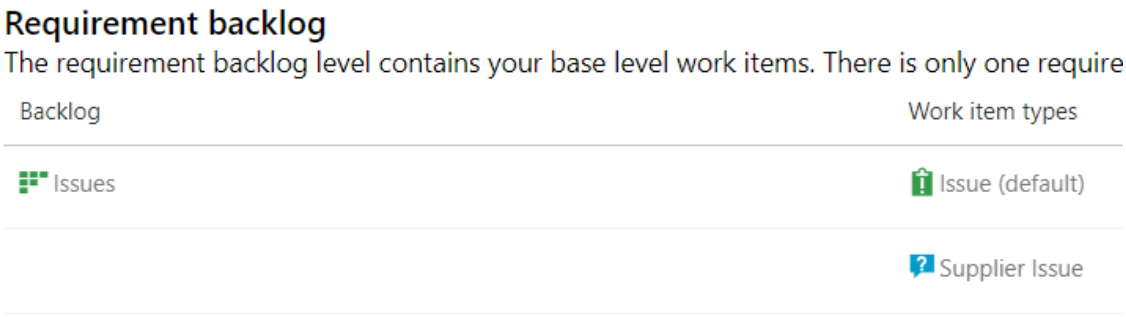

Figure 26: Custom Work Item as Requirement Backlog

Now it should be shown on your board and on your backlog. You will recognize it by the different icon.

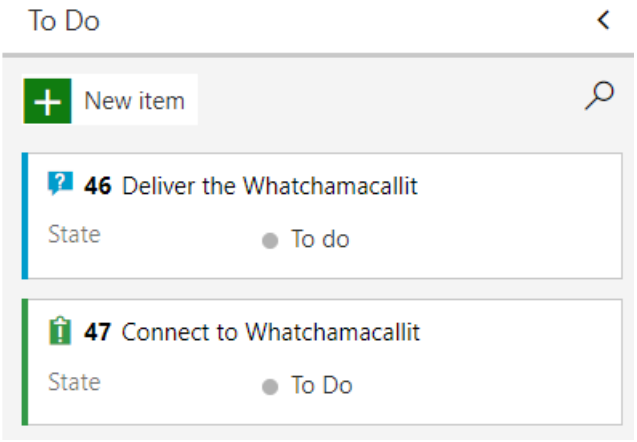

Figure 27: Custom Work Item on the Board

If you like, you can also create a new top-level portfolio backlog, so that is the same level as an epic. To do this, just go back to the **Backlog levels** and click **+ New top level portfolio backlog**. More interesting, perhaps, is to create a new item on the iteration backlog level, so at the same level as a task. If you do this, you can even create a new custom task directly from your boards.

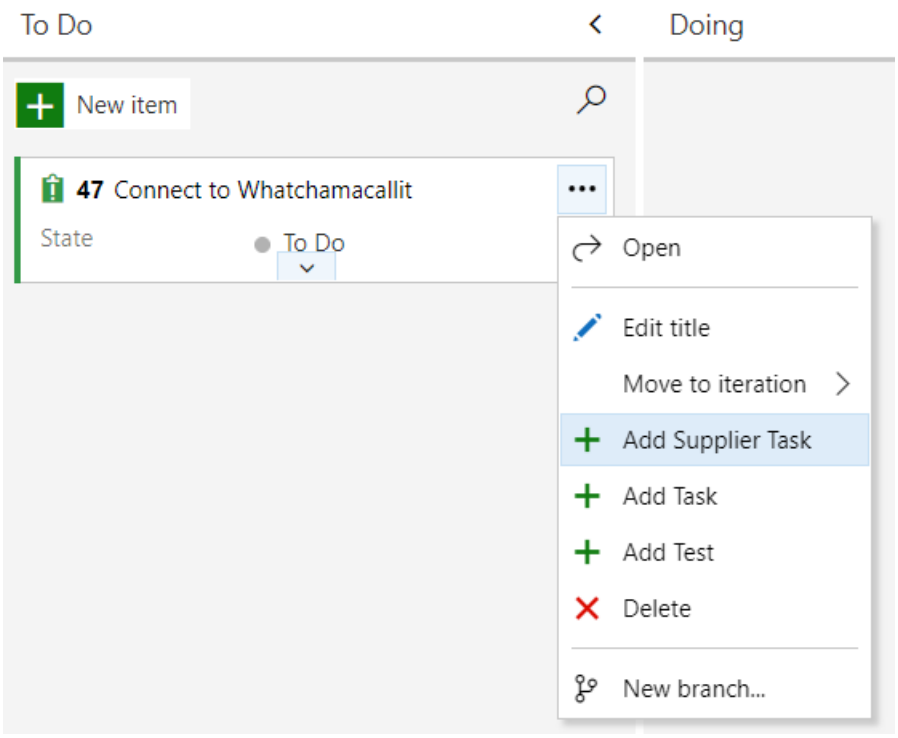

Figure 28: Add Supplier Task

Migrating processes

Now that you have your new process, you probably want to migrate your Scrum project to the new process. To do this, go to the **Process** settings in the **Organization** settings. Click the **Basic** process and go to the **Projects** tab. Here you will see the projects with that project type listed. Click the three-dot button and select **Change process**. Select the new project type and click **Save**. You cannot just migrate from any project type to any other project type. For example, if you moved from Basic to Scrum, you would lose all your issue work items. Azure DevOps gives you a warning when this happens. So, think ahead before selecting a work process.

Multiple boards

It is possible to use multiple boards in a single project. You can do this by creating teams within a project. We have already seen this briefly in the previous chapter. Go to your project settings and create a new team named **Back-end Team**. If you now go to **Boards**, you can select your team board at the top left of the board. When creating a new work item, you can set the **Area** to either **Scrum project** (the default team) or **Back-end Team**. The same goes for your backlogs.

The biggest difference is in the planning of the sprints. I will not say too much about it, but in your project settings you have your boards **Team configuration**. Here you can plan iterations and specify team-specific board settings. The team can be selected at the top of the screen (in the breadcrumbs section). For example, if your organization works 24/7, you may have a weekend team that works on Saturday and Sunday. You cannot have different processes per team in the same project.

Summary

In this chapter we used boards to plan our work. We were able to create various work items under different process types. By viewing them on the board and in the backlog, we were able to plan and estimate our work. By working in sprints, we get more reliable reporting in the form of a burndown chart. Using the sprint taskboard makes it even easier to focus on the current sprint. By customizing your processes, you can add fields that are important to your specific workflow.

Chapter 4 Repositories

In the previous chapters, we familiarized ourselves with Azure DevOps and planning work with Azure Boards. However, although you can plan all you want, ultimately a team will have to deliver software. With Azure DevOps Repositories (or Repos for short), teams of any size can share code, implement versioning, and review each other's work. In this chapter we are going to take a closer look at Azure DevOps Repos.

Working with Repos using Visual Studio

I have already briefly touched upon the options for source control in Azure DevOps. Because Git is by far the most commonly used source and version control system today, I am not going to discuss Team Foundation Version Control. Also, this book is not so much about Git, so while we will use it throughout this chapter, I am not going to explain the Git basics.

Azure DevOps projects can contain multiple repositories. That makes sense because, especially with microservices, projects can consist of multiple applications or services. Using Git, every (code) project gets its own repository. A default repository is created when you create a new project in DevOps.

Open a project and browse to **Repos** in the left-hand menu. Here you will see your default repository or the repository you opened last. You can create or delete repositories at the top of the Repos screen by clicking the repository name. The default repository is empty, and we cannot do anything with it yet. Luckily, DevOps has a nifty **Initialize** button that creates a README.md file and/or a .gitignore file. You can pick the type of .gitignore file you want. There are loads, like VisualStudio, UnrealEngine, Node, Java, Android, and much more. Select the **VisualStudio** one, because I want to create a small C# project later, and click **Initialize**. This initializes the repository and shows you the contents of the README.md file, including the Markdown markup. You can browse other files on the left. Clicking a file shows you its contents and allows you to edit and rename files, but I do not recommend using this overview to do that.

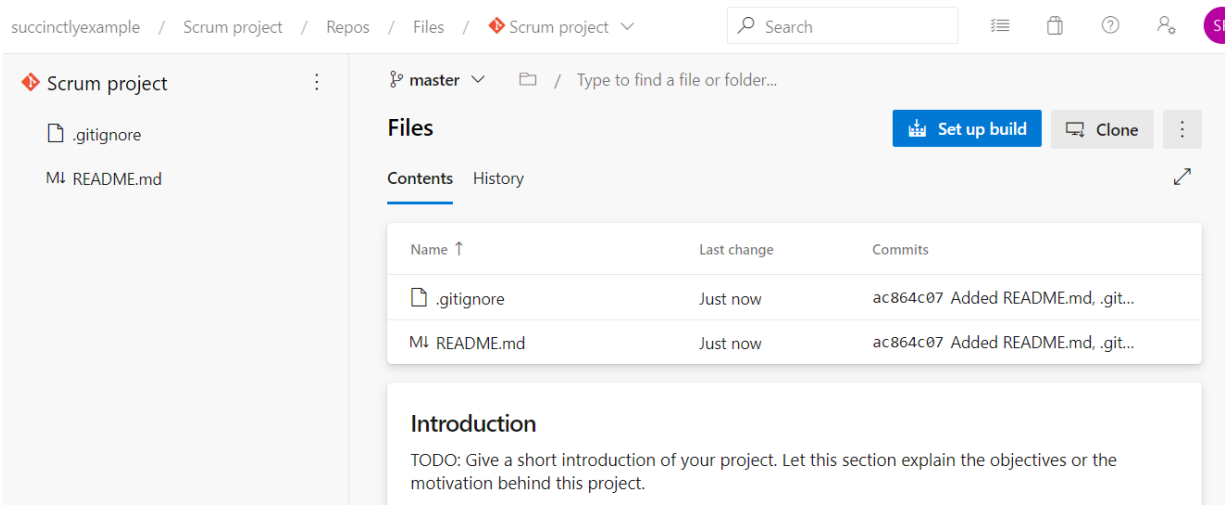

Figure 29: The Default Repository Page

We can now clone our repository. At the top right is a **Clone** button, which lets you clone your repository directly using various applications such as Visual Studio, Visual Studio Code, Android Studio, Eclipse, and others. If you would like to use another application that is not listed, you can use the HTTPS or SSH URI to connect however you like.

> ***Note: In the following section, we are going to use Visual Studio and .NET Core. If you are using Azure and Azure DevOps, but not Visual Studio or .NET Core, or if you just want to get through this chapter quickly, you can just commit a simple text file with a simple text, like** Hello Git**. However, in the next chapters we are going to use our .NET Core project to create an automatic build and release to Azure.***

Unfortunately, cloning directly with Visual Studio does not seem to work for me, so I am going to clone from within Visual Studio instead. In Visual Studio, open **Team Explorer** (in the menu under **View**). There, click the green power plug icon and then **Manage Connections** > **Connect to a Project**. You can now add an account, which could be the Outlook account that we created in Chapter 2, or any other account that you use to sign into Azure DevOps. This will list your Azure DevOps organizations as well as the projects and Git repositories. When you connect from here, everything works as it should.

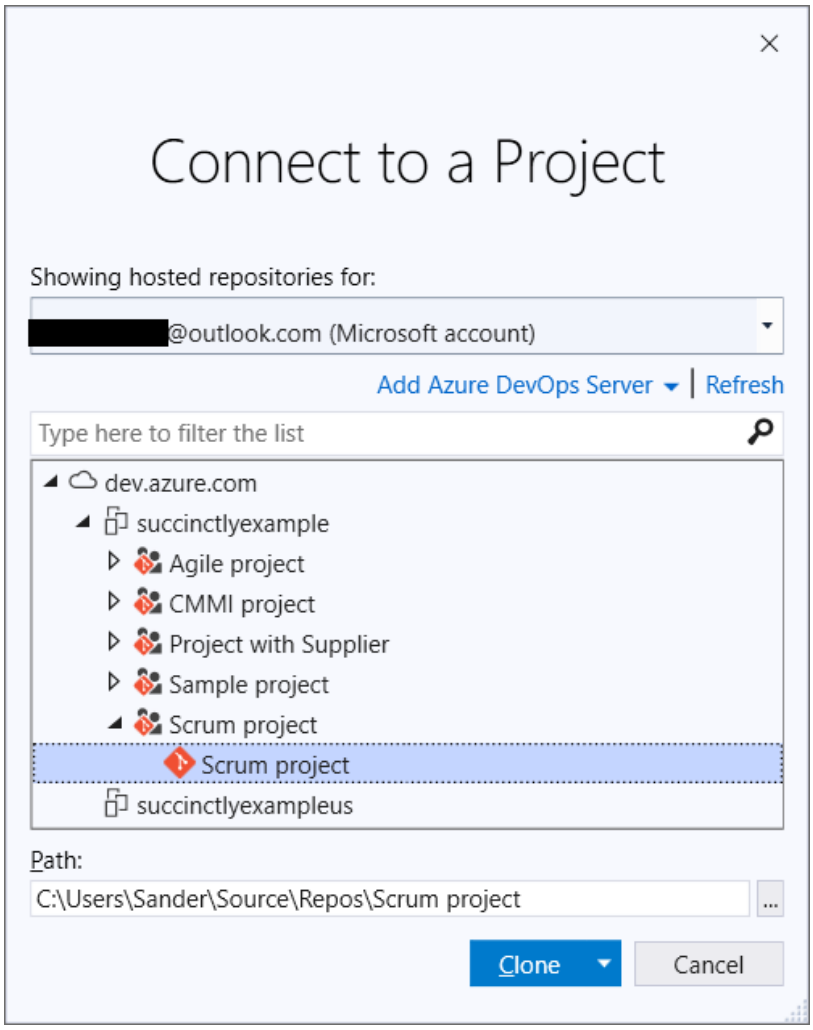

Figure 30: Connect to a Project Using Visual Studio

When you clone the repository, Visual Studio will open the folder and show the .gitignore and README.md files. Now, using Visual Studio, create a new ASP.NET Core Web Application with Razor Pages and save it in the folder where you cloned your repository. You can leave all the defaults (the **No Authentication** and **Configure for HTTPS** options selected; the **Enable Docker Support** and **Enable Razor runtime compilation** options cleared). Press F5 to debug the application to make sure you get the default web application page.

When everything works as expected, we can commit and push our code back to Azure DevOps. Go back to your Team Explorer and click the **home icon**, and then click **Changes**. You will see all the files that were added to your repository except the ones that are excluded by the .gitignore file, like the bin and obj folders. Click **+** next to **Changes** where it shows all your added files. This will stage your files, meaning they are ready to be committed. Enter a commit message (for example, **My first commit**), open the **Commit Staged** drop-down button, and click **Commit Staged and Push** instead. This will push your changes back to Azure DevOps.

Now, go back to Azure DevOps and check the **History** tab.

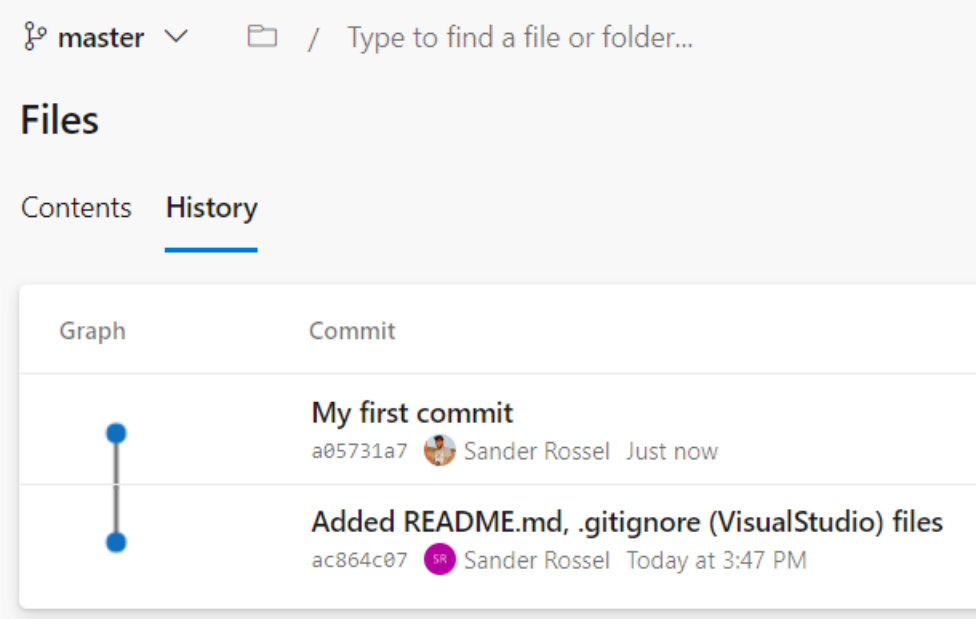

Figure 31: Repository History

That was quite a bit of work, but we are going to use this later, so it is not for naught.

Another method of viewing your history is by going to **Commits** on the left-hand menu. You can click any commit to view what has been changed in that commit. You may be wondering why the screenshot shows SR in the initialization commit and my picture in the commit from Visual Studio. The answer is that they are two different accounts with the same name.

There is also a Pushes menu item that shows your pushes. Notice that this is not the same as a commit. Depending on how you work it can often be the same, but you can also make multiple commits and one push.

Branching

One important aspect of Git is branching. Git has made branching much easier than that of its Subversion predecessors. There are two types of branches, local branches and remote branches. Local branches are branches that you created on your computer, but merge back to a remote branch before you push them. A remote branch is a branch that is created on the server, in this case Azure DevOps, and that you can share between coworkers. You can start with a local branch and then push that to Azure DevOps (the remote), making it a remote branch. Of course, we can also create new branches in Azure DevOps directly.

Go to **Branches** in the menu, and you will see a list of your own branches, all branches, and stale branches. Stale branches are branches that have not seen a new commit in over three months. In the top-right corner, click **New branch**. In the pop-up window, enter **develop** as the name and base it on the master branch. Leave **Work items to link** empty and click **Create**. You have now created a new branch that can be checked out by the team.

When you hover over a branch you will see the three-dot button (…); click on it to see other actions that you can do with a branch. Here you can create new branches off an existing branch, delete branches, view files and history, compare branches, and set branch policies. I will talk about these policies later. First, let us make a commit on our new development branch.

In Visual Studio, go to your **Team Explorer** again and click the **home icon**. Now, click **Branches**, open **remotes/origin**, and then double-click **develop**. If you do not see the branch, go back to **home**, go to **Sync**, and click **Sync**. This will synchronize your repository with the remote. When you are on the develop branch, go to your code and make a change. For example, go to **index.cshtml** and change **Welcome** to **Welcome develop**. Commit your changes as you did before.

Now, in Azure DevOps go to **Branches**. You can already see that the develop branch is one commit ahead of the main branch (in the Behind | Ahead column). Click the three-dot button on the develop branch and click **Compare**. You should now see your new change as a difference between the two branches.

Merging

No doubt you want to merge this change back to the master branch. You can do this locally and then push to the remote, but you can also do this using Azure DevOps. To do this, you have to create a pull request. In Azure DevOps, you will see the message **You updated develop 1m ago [Create a pull request]**. Click **Create a pull request** or, alternatively, if you do not see the message, go to branches, click the three-dot button on the **develop** branch, and click **New pull request** there.

In the pull request form that opens, you can set a title and description for your pull request, as well as link to work items. You can also set reviewers, people who must approve your changes before they are merged to master. Leave everything as it is and click **Create**. Next, you will be taken to a page with the pull request. Instead of accepting it right away, click **Pull requests** in the menu. Here you will see your pull request as well. Go back to the organization page and click **My pull requests**. It will be visible there as well. I will go into pull requests in more detail later, but for now open the pull request and click **Complete**. In the complete form that opens, clear the **Delete develop after merging** option and then click **Complete merge**. After that, you can close the pull request by clicking **Close** where Complete was.

Now, go back to the master branch and view the commits. It will show you a graph with the merge clearly visible.

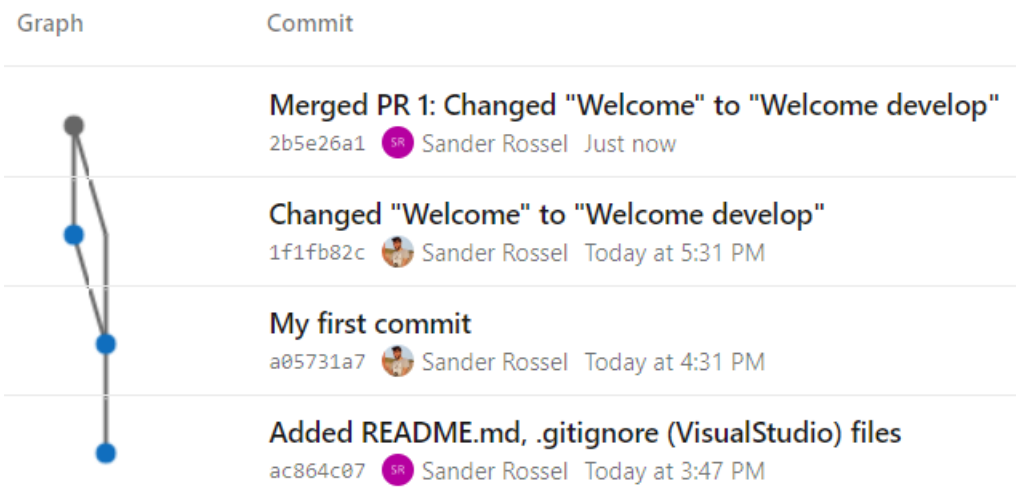

Figure 32: A Merged Commit

Next, let us look at some branch policies that will force us to create pull requests and prevent us from approving our own changes.

Branch policies

When you have a branching strategy, such as GitFlow, you want people to work in a specific way. Often, you will have one branch for development and another for deployment, which is often the master branch. The deployment branch should be the one that goes to production, and should not change until a new deployment is done. That way, you always know what code is running on production, and you can find and fix bugs without having to worry about new code. For this reason, you can set branch policies.

Go to your master branch and click **Branch policies**. You will be taken to a new form where you can set various properties on your master branch. At the top is the text **Protect this branch**. Protecting a branch means that you cannot directly commit on this branch. Instead, you will have to create a pull request, which I will talk about in a moment. A protected branch also cannot be deleted.

The first four checks on this page are all about protecting your branch. Enabling any one of these protects your branch, and will require people to create a pull request. A pull request is simply a check before merging one branch back to the other.

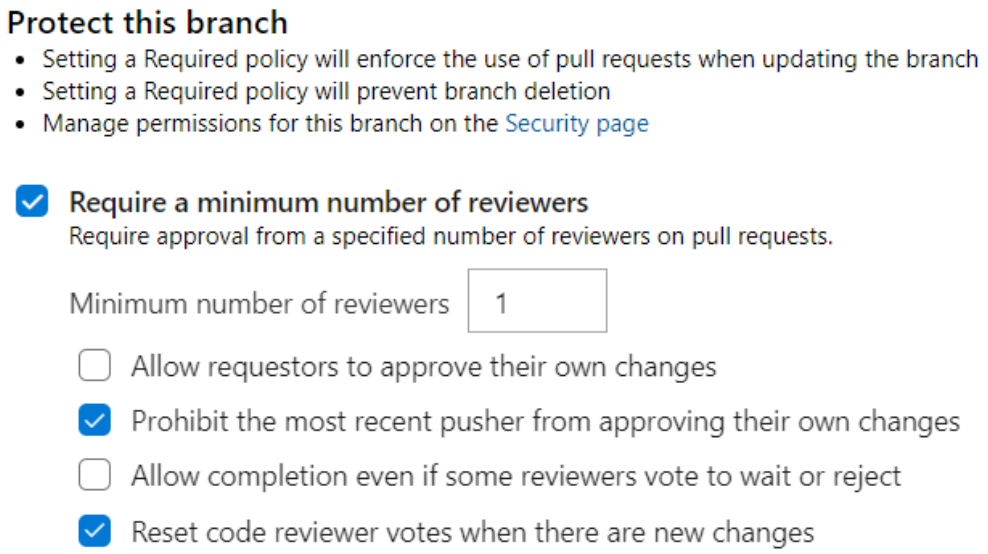

Figure 33: Protecting Your Branch

We have briefly seen a pull request before we completed it ourselves. Completing your own pull request kind of defeats the purpose of having pull requests. Here, you can set a minimum number of reviewers and whether the author can approve their own changes. The minimum number of reviewers depends on your team. If you have a team of two, setting the minimum to three is a sure way of never getting any work done. Also, when people make a vote to accept or reject your changes, you can make a change and add that to the pull request. You can set whether this will reset votes. Also, if someone else on the team makes the change, you can prohibit them from approving their own change. Last, if two people approve your change, but one person rejects it, you still have enough approvers to complete the pull request. You can set whether a reject overrides all approvals.

You can also require people to link to a work item in a pull request. That way, code that is merged back into master can always be traced back to a work item. You will know who requested the change and why (assuming your work item is up to par).

People can comment on pull requests, for example: “Why did you use recursion instead of a for loop?” or, “This code is technically correct, but the casing is not according to our guidelines.” You can require these comments to be resolved before a pull request can be accepted.

Last, you can limit the amount of merge types. In our pull request, we chose to have a merge commit even though it was not necessary. Since we did not have any other commits on our master branch, we could have fast-forwarded, but that would obscure the fact that there was once a merge. So, you may want to prohibit some merge types.

Other policies include the **Build validation**, which I am going to discuss in a later chapter, **Require approval from additional services**, which is out of scope for this book, and **Automatically include code reviewers**. That last policy is interesting, so we are going to set it. This simply enables you to add some default reviewers in your pull requests. This can be your lead developer, for example, or someone from QA. We have one other person in our organization, so we are going to add that account as a required reviewer. Click **+ Add automatic reviewer** and find the other account. You can add filters for specific reviewers, like ***.html;*.css;*.js** for your team's lead front-ender. We are leaving the filter blank to add the reviewer to all pull requests. Click **Save**. By hovering over the new reviewer, you can edit or delete them. You can also (temporarily) disable the reviewer.

Whenever you clone the repository, the master branch is checked out and you would be inclined to start building and commit back to the master branch. When you check the branch overview, you will notice that the master branch has a "default" label. It would make a lot more sense now to have the develop branch as our default branch. To change this, go to your project settings and then to **Repositories**. Click your branch, and then select **Set as default branch**.

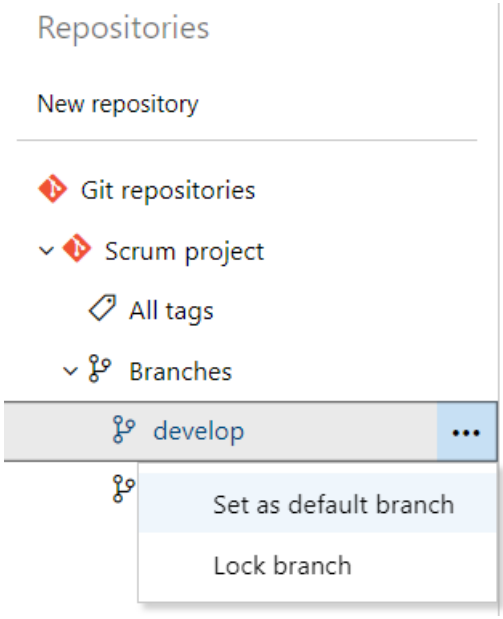

Figure 34: Set Default Branch

If you go back to the branch overview, you will see that the default label moved to the develop branch. If you now clone the repository you will get the develop branch by default.

To test all of this we just have to create a new pull request, which we are going to do in the next section.

You can find other policies under project settings by going to **Repositories** and then the **Policies** tab. Here, you can limit file size and maximum path length, block pushes that contain reserved names, and block pushes from authors whose email does not match a specific pattern.

Pull requests

Time to create another pull request. Make another change; for example, change **Welcome develop** to **Welcome pull request**, and commit and push it to Azure DevOps. Then create a pull request like you did before.

Tip: Did you accidentally commit and push to master? You will get an error message saying the remote repository rejected your push. You now still have a local commit, but no way to push it. You can right-click your branch in Visual Studio, and then click** Reset **>** Keep Changes (--mixed**). This will revert your commit but keep your changes. It is possible that you will get conflicts, which you will have to fix before continuing. You can then switch to the correct branch and create a new commit.

Go to **Pull requests** in the DevOps menu and click **New pull request**. The next window looks the same as before. You will notice that your other account was not added as a required reviewer, but do not worry about that. You could change the description if you wished. I sometimes do that when I want an opinion of some coworkers, like "*I have used x to solve the problem, but maybe y would be better, what do you guys think*?" Once you have created the pull request, your other account should receive an email that a pull request was created and that the account was added as a reviewer. In the pull request you will also see that the reviewer was added, and that the reviewer must approve. You can still approve it yourself, but if you set the policy that authors cannot approve their own work, this will do nothing except show others that you are approving your own changes. Others will get an email saying that you approved though, so it can be a nice reminder when people do not review your changes in due time.

Next to the Approve button is a **Set auto-complete** button. You can click the drop-down arrow on this button and select **Complete**, but you will get an error that this is not possible. You can also abandon the pull request here, which is basically a delete, or mark it as a draft if you still wish to change it. The auto-complete option works quite well, and personally I like to use it as much as possible. Normally, people would have to approve your changes, and then you would have to return to your pull request and manually complete it, like we did last time. However, when you set auto-complete, your pull request is automatically completed when all conditions are met, and everyone approved. Note that you can still get merge conflicts by the time everyone approved. You would have to fix those first by updating your branch, fixing the merge conflicts, and then pushing them back to your merge conflict.

Try making another change and pushing it. I changed **request** to **requests** so that it now says **Welcome pull requests**. Your Azure DevOps pull requests will update automatically and show a message that new changes were pushed. All reviewers will get another email notifying them about this change.

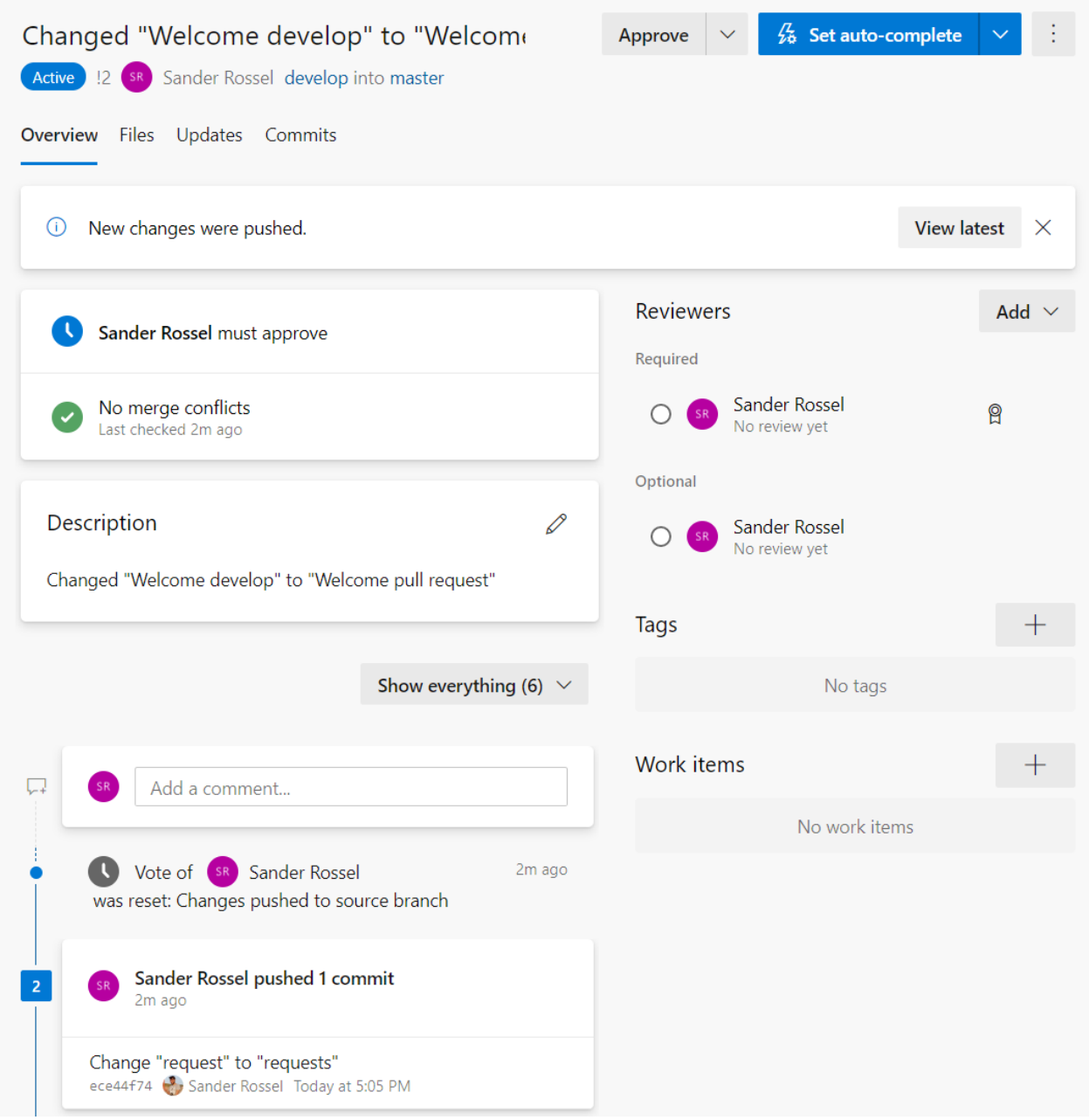

Figure 35: Pull Request

Set the auto-complete, which is the same as manually completing. Notice that this time you cannot delete the develop branch because it is the default branch. If you are working with feature branches and you are done with a feature, it is good practice to delete the branch, so you do not get hundreds of stale branches. Once you have set the auto-complete, you can log in with your other account.

As a reviewer, you can cancel the auto-complete and manually complete the pull request instead. Of course, you can cast your vote, approve, approve with suggestions, wait for author, or reject. You can view commits and pushes specific to this pull request. You are probably most interested in the Files tab. Here you can compare the old code with the new. The default view is to have the changes inline, but you can change this to side-by-side in the top right, which is my preferred view. You can comment on code by clicking on the comment icon in the margin. The author will get an email and can respond to your comment. When a comment is answered, you can resolve the comment and it will be closed. There is a policy that requires all comments to be resolved before you can complete a pull request. When you are done reviewing the changes, click **Approve**, and the pull request will complete and close automatically (if you have set that option).

Branch security

Sometimes it can be good if someone can override certain settings. Imagine being the only one in the office because people are on vacation or on sick leave, but you really need to get some code out. Obviously, the code reviewers are not going to do a review today, but you cannot wait until next week. Using roles, you can set certain permissions for groups or people. Go to your branch overview and click **Branch security**, right under Branch policies. You can find it under your project settings as well, under **Repositories** and **Security**. There, you can set project-wide security settings as well as branch-specific security settings.

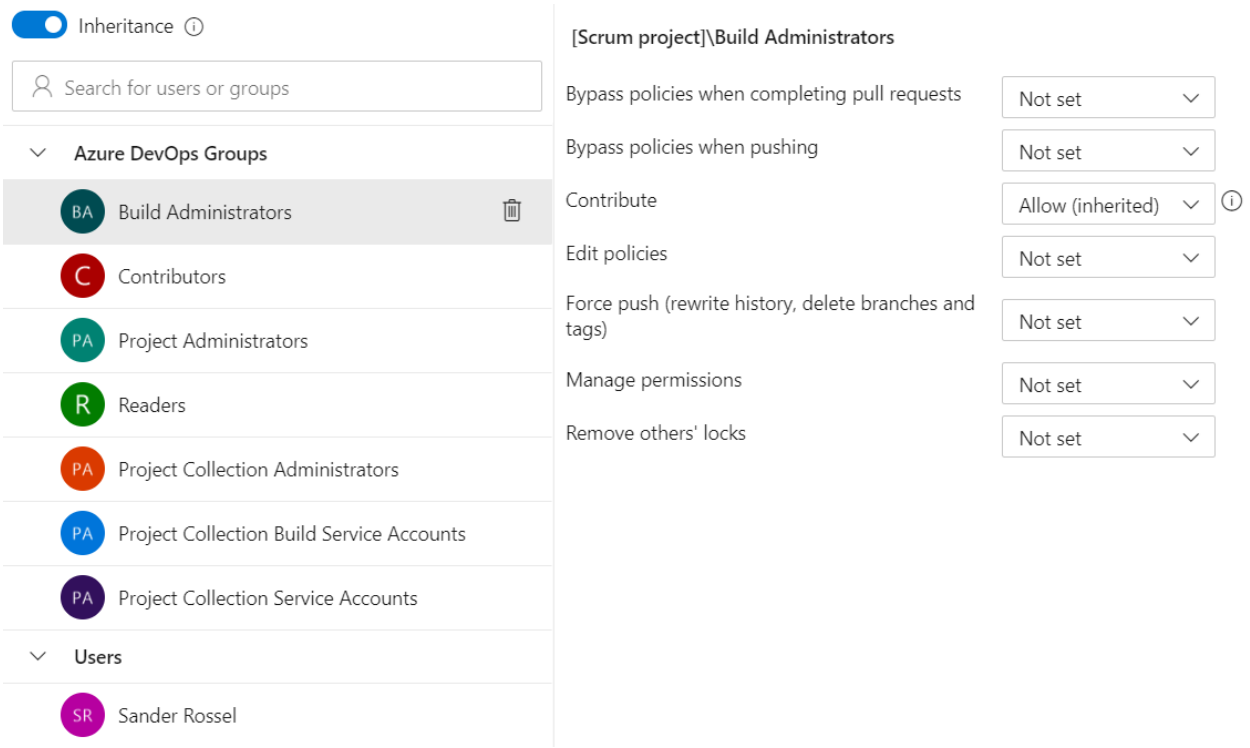

Figure 36: Branch Security

As always, you should be careful who you give permissions to. One permission to look out for is "Bypass policies when completing pull requests," which you can set on every level, including for specific users (at the branch level). Other permissions include reading, creating branches, deleting repositories, editing policies, and forcing pushes. As with project security, these groups are inherited. You can allow or deny a specific permission, or you can leave it on **Not set** and set it at another level.

Tags

Last, you can view your Git tags in Azure DevOps. Like branches, you can create tags locally and push them to DevOps, or you can create them in DevOps directly. A tag is a pointer to a specific point in the Git history. They are often used to denote a specific version, like v1.0 or beta-0.9.1. A tag can be viewed as a branch that does not change. Once the tag is created, it gets no further commits. You can create a tag by going to **Tags** in the menu and then clicking **New tag** in the upper-right corner. You have to give it a name and a description and specify which branch, other tag, or specific commit you want to base your tag on.

Another way to create a tag is by going to your commits and selecting **Create tag** from the inline menu on any commit. This will create a tag on that specific commit. Going to your tag then shows the code as it was in that specific commit.

Using tags to order your work can really help in keeping your number of branches down. For example, instead of creating a branch v1.0, you can create a tag. If v1.0 has a bug, you can branch from the tag and then merge it back into your current branch or release.

Summary

Azure DevOps offers everything you would expect from Git and CI/CD tooling. You can have branches, pull requests, code comparisons, code reviews, and tags, and you can link back to work items. In the next chapters, we are going to automatically build and deploy the code that we keep in our Azure DevOps repos.

Chapter 5 Build Pipelines

In the previous chapter we used Azure DevOps to save our code in Git source control. In this chapter we are going to use Azure DevOps pipelines to build our code. By automatically building your code, you are immediately notified when your code breaks a build in a way that prevents deployment. While planning work and using Git is important in any environment, the biggest strength of DevOps, for me, lies in automated builds and deployments. It reduces risk, saves time, and allows you to deploy at any time with minimal downtime, if any.

Classic build pipelines

Let us start by creating our first build pipeline. Our goal is to be able to build our software so that we know our source has no incorrect syntax (it happens), packages are available, and that any automated tests succeed. Go to **Pipelines** > **Create Pipeline**.

You will now be asked where your source code is: Azure Repos Git, Bitbucket Cloud, GitHub, another Git repository, or Subversion. However, we are not going to answer that. Instead, we are going to click **Use the classic editor to create a pipeline without YAML**. We will get back to this later, but I think the classic editor is more suitable for learning purposes.

In the classic editor you are once again asked to select a source, where you can select the same sources as before. You will notice you can select other projects than the one you are in now, so that is an option too. However, we will just keep our current project and default repository. The default branch for manual and scheduled builds is interesting since we have two branches, develop and master. Basically, we want both to build as soon as something is committed to them. After all, both must always be deployable, where develop is ahead of master. Keep develop so we will at least know our latest commit builds as it should.

Next, you get to select a template. At the top is the YAML template that we just opted out from. Featured templates include .NET desktop, Android, ASP.NET, Docker Container, Maven, and Python package. However, we are going for the **ASP.NET Core** template. If none of the templates satisfy your needs, you can select **Empty job** at the top and start from scratch.

Once you select the ASP.NET Core template you will be taken to the pipeline editor. The pipeline editor shows an agent job, which is basically a set of tasks that are executed on a single server. In the agent job you see the steps Restore, Build, Test, Publish, and Publish Artifact.

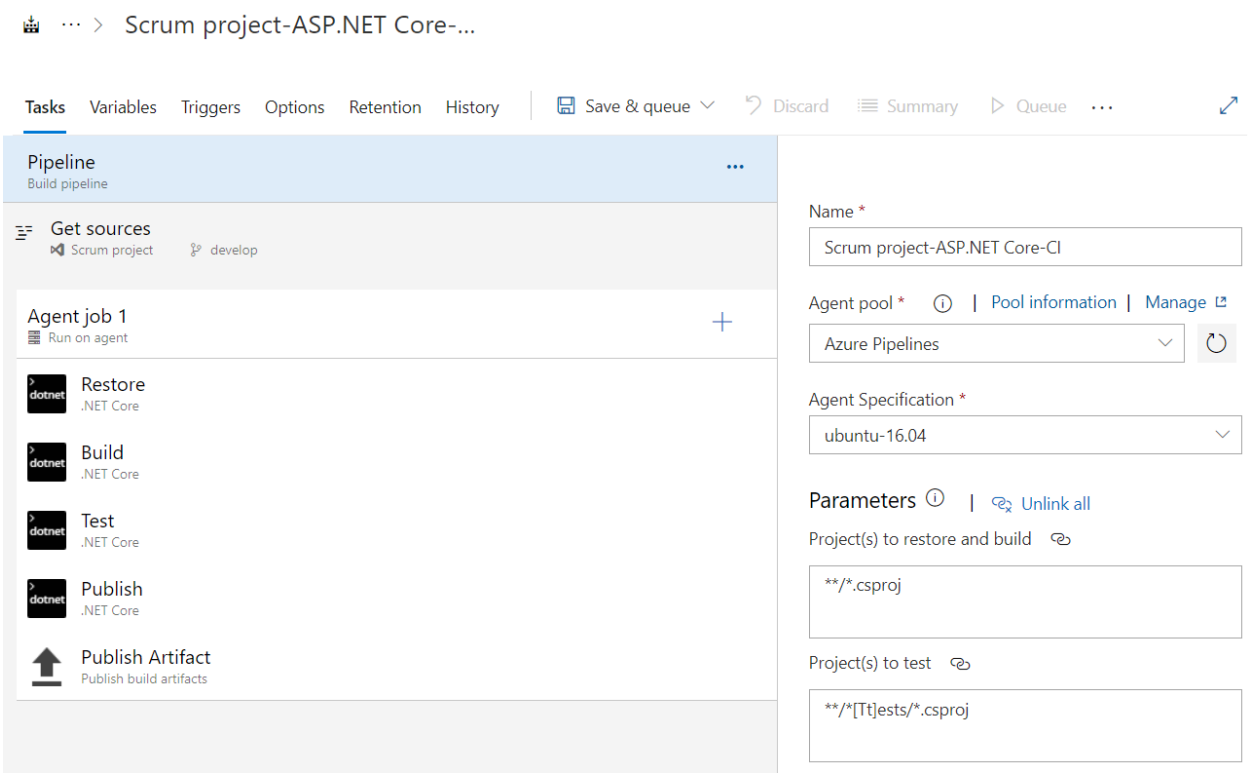

Figure 37: Pipeline Editor

Let us look at the various levels in your pipeline: the pipeline, jobs, and tasks.

Pipeline settings

Before you continue, you probably want to change some pipeline settings. First, the name of your pipeline. For some reason I often forget this, and I am left with a generic pipeline name. Name it something like **WebApplication1 Build** (if you did not rename your web application). Next, you can choose an agent pool, either Azure Pipelines or privately hosted agents. We do not currently have a privately hosted agent, so **Azure Pipelines** is our only option. That means your build is executed on a server hosted by Azure DevOps rather than your own. You can also pick an agent specification. You can pick macOS, Ubuntu, or Windows. The list is not long, but you should be able to build most of your projects on one of those. For everything else, you have private agents. Change the agent to **windows-2019**. These agent settings are inherited by your jobs, but can be overwritten. On a side note, I never need more than one job. The parameters are a bit difficult to explain now, but we will get back to them later.

If you click **Get sources**, you will find your Git settings that you just entered and some additional settings that I will not go into.

Agent settings

On **Agent job 1** you can configure some settings, like parallelism, demands, and overwriting the agent pool. A demand is something that applies to your self-hosted servers. Imagine you have a server that has a third-party application installed that is necessary to build your source (for example, a license for a library you are using). You can tag your private server with **third-party license** and then demand here that the pipeline agent have that specific tag. Or maybe you use it for a different version of Visual Studio. For example, the **visualstudio** label must have a value of **2015** or **2019**. We will see this later.

You can add extra agents by clicking the three-dot button (**…**) on your pipeline. You can add an agent job (running on a server) or an agentless job. The agentless job is limited in what it can do, but it can delay execution, invoke REST APIs or Azure Functions, and publish to Azure Service Bus, to name a few. Personally, I never needed these, but I wanted to point them out. You could add a second agent if you want to build on multiple platforms, for example Ubuntu for Android and macOS for iOS, if you have a mobile app.

Tasks

Below your agent, you will find tasks. In our current template we have four .NET Core tasks, which run the dotnet command-line interface with a specific command, and a Publish build artifacts task, which will create an artifact of any build output. We can add tasks if we want by clicking **+** on the agent job. If you do this, you will see all tasks, but you can filter by Build, Utility, Test, Package, Deploy, or Tool tasks, as well as download more tasks through the Marketplace. Here you can find tasks that can run Bash scripts, batch scripts, use cURL to upload files, sign Android APK files, build Docker images, run Grunt or Gulp for JavaScript, and much more. I have not counted them, but including the Marketplace there are probably hundreds of tasks that you can use. Needless to say, I am not going to discuss them all.

Next, let us examine the Restore task. You can see that even though it is named Restore, the task is of type .NET Core, just like the Build, Test, and Publish tasks. This task runs the dotnet command-line tool with the **`restore`** command and no arguments. You can set additional settings like feeds to use, but the defaults will do for now. This task will now perform **`dotnet restore`** on your project as if you were running it from the command line.

Linking settings

The interesting part to notice here is the Path to project(s) settings. You cannot change this, and clicking the little chain icon reveals that it is linked. This is linked to the parameters we saw in the pipeline settings. If you want to change this value, you have to either unlink it or change the value in the pipeline settings. If you opt for the latter, you will change the value for every other setting that it is linked to. The Arguments setting is not linked, so click on the little **i** icon next to it and click **Link**. In the pop-up that opens, edit the display name so that it is easy to recognize and enter some value. Then click **Link**.

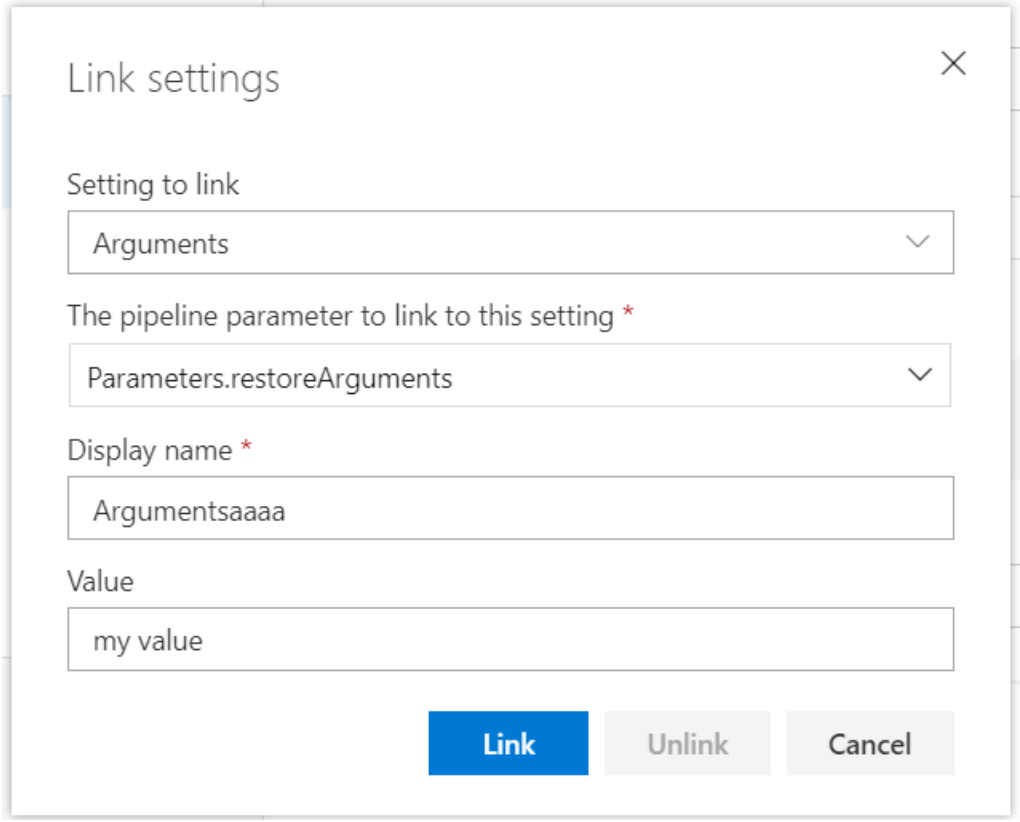

Figure 38: Link Settings

You will now see that the argument setting is linked as well. When you go back to the pipeline settings you will see your **Argumentsaaaa** parameter with **my value**. Clicking on the chain icon tells you where it is used. You can now link this to other fields as well. Go back to the **Restore** task, click the chain icon, and click **Unlink**. This will unlink the field and remove the parameter. You can now edit the value again, so make it empty.

The Build and Test tasks work pretty much the same, except the command is different and they get an (unlinked) argument. These arguments come from a variable that we will look at in a moment.

Artifacts

Before we continue, I would like to discuss the Publish and Publish Artifact tasks. The goal of your pipeline is to create an artifact that you can deploy. First, we will need to publish the software, and then we can make an artifact that contains the published software.

The Publish task is another dotnet CLI command, but this time it is the **`publish`** command, which will publish your software so that it is ready for deployment. In the task you will see a check box labeled **Publish web projects**, which is on by default. With this setting enabled, the task will publish all projects that either have a web.config or a wwwroot folder. If you have a web project without either of those, or you simply do not have a web project, this step will currently publish nothing. In this case we are fine, but if you have any other project type, for example a Console app, be sure to deselect this. You will then have to enter the path to your project, which is linked to ****/*.csproj** by default, but that means all your projects will be published, while you only want your entry project to be published (all other referenced projects will be included in the published folder). So, you will have to unlink the value and enter something like **MyProject/MyProject.csproj**, or keep using wildcards and enter ****/MyProject.csproj**.

Also, if you do not want your published project to be zipped, make sure to clear the **Zip published projects** check box.

The next task creates your artifact, which is simply any output of this pipeline. The Publish task publishes your project to a default folder, which is accessible using the **build.artifactstagingdirectory** variable. In this task you will see that the **Path to publish** is set to **$(build.artifactstagingdirectory)**. The $() means that anything between parentheses is a variable that is to be interpreted. You can name your artifact, but the default name is **drop**. You can also specify a location, either **Azure Pipelines** or **A file share**, which makes you enter a path. Keep all the defaults.

We are not done yet, but before we continue let us click **Save & queue** at the top and see our pipeline in action!

Running a build

To sum up what we have done, we picked the ASP.NET Core template, changed the name of our pipeline, and changed the agent specification to **windows-2019**. That is all you need to do before saving and queueing.

When you click **Save & queue** at the top of the editor, you will be presented with a prompt. You can enter a save comment, like **changed the publish path**. More importantly, you can change the agent pool, agent specification, which branch you would like to build, any demands, and any variables. Leave everything as it is and click **Save and run**.

This should start the build. You will see a summary of your run, as well as your jobs and their status. Your build will be queued and as soon as an agent becomes available, your tasks start running. Keep in mind that you have only one hosted agent available, so if a build or deployment is running, another build or deployment will have to wait for the other to finish. You can buy more agents in the billing section of the organization settings. If you pay for one parallel job, you will be able to run two pipelines simultaneously, and so on.

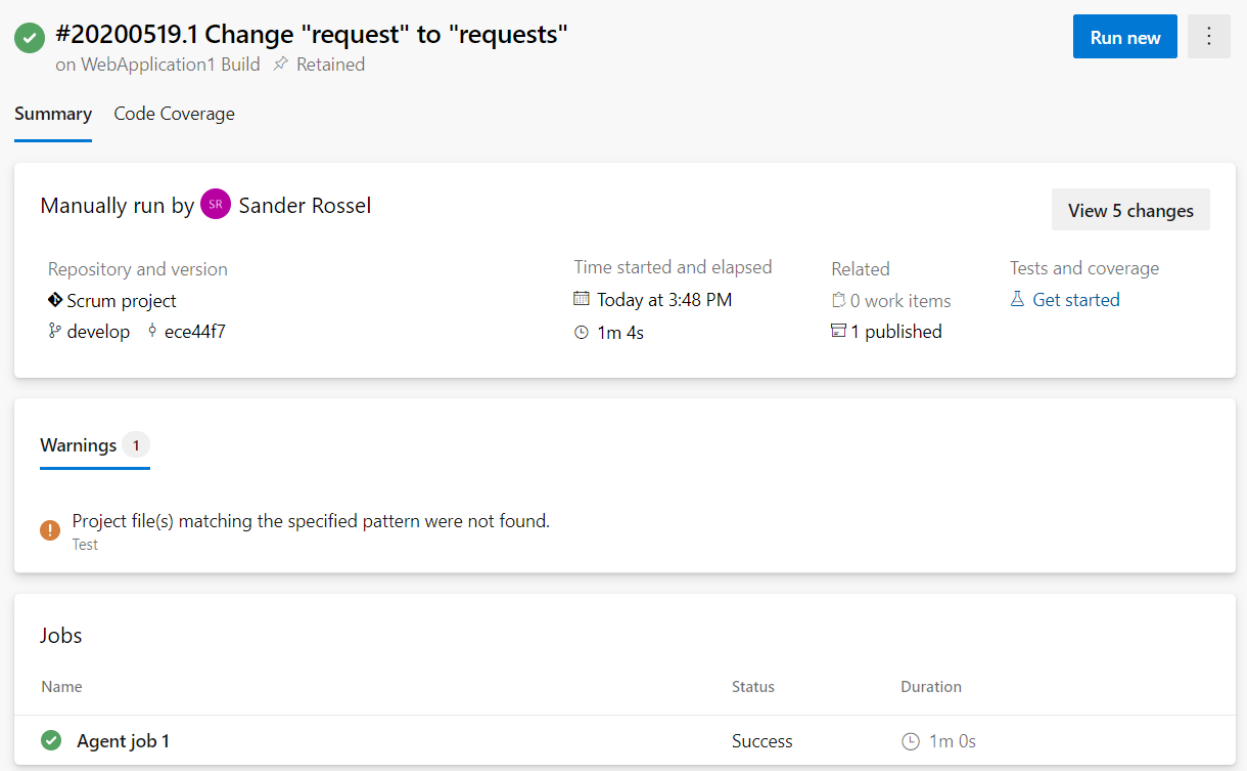

Figure 39: Build summary

When the build is finished you will see a warning that no tests were found, which is true and as planned, so we can ignore this warning. You will see that the job succeeded, but you can drill down to the individual tasks by clicking on the task. In the next form you can see the tasks and logging of the tasks. This is invaluable when a job fails and you want to know why.

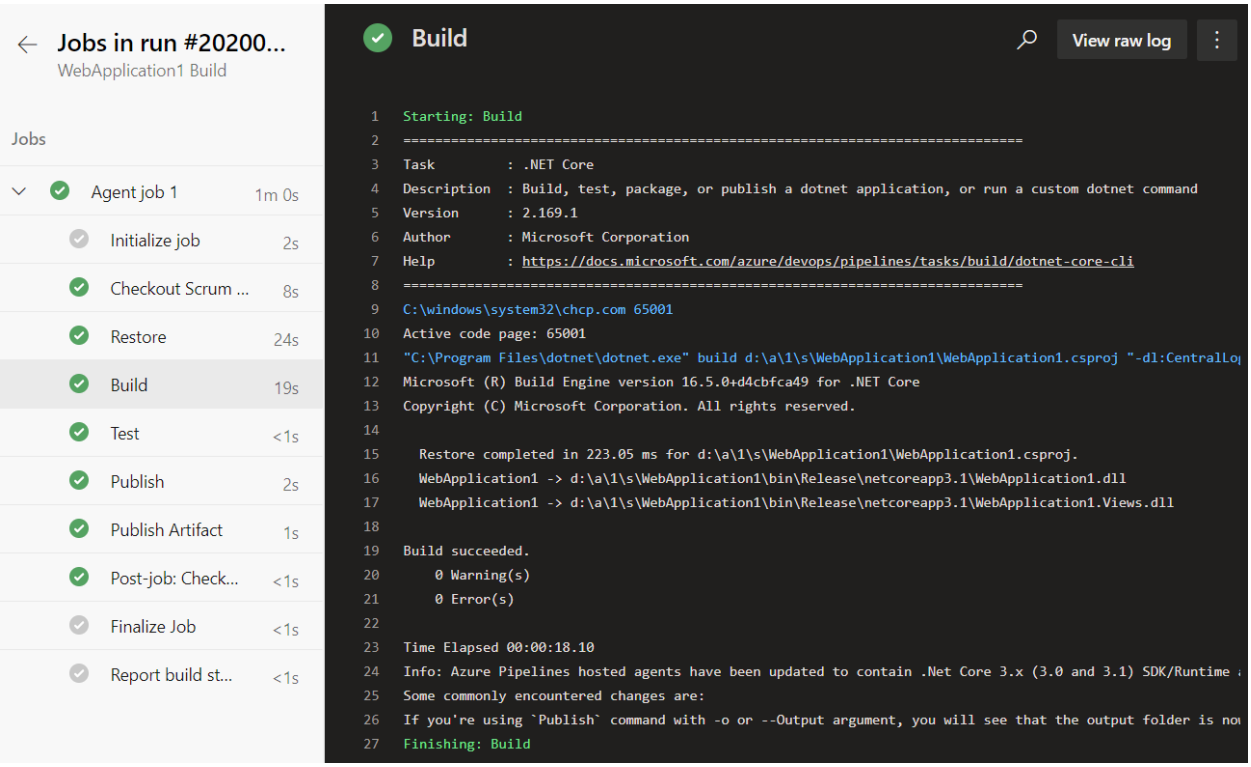

Figure 40: Build output

We can also view the artifacts this build has produced, if any. Go back to the summary and click **1 published**. You will now see your **drop** artifact, and if you click it you will see it contains a zip file named **WebApplication1.zip**. You can download it to view the contents. If you disabled zipping in the artifact task, you will see all the files here and you can browse them. We will need this artifact later if we are going to deploy this project.

Variables

Let us go back to the designer. When you go to your pipeline overview, you can hover over it and click the three-dot button, and then click **Edit**. This is also where you can start a new run or delete a pipeline. You can also click the pipeline, which will give you an overview of the individual runs and an Edit button in the top right.

Examine the Build task. You will notice that it has an argument**, —configuration $(BuildConfiguration)**. If you look at the top, you will notice a few extra tabs, one of which is **Variables**. This is where your variables are stored. You can use it as an alternative for linking values. You can add new variables and make them settable at queue time, which means you can change them before you trigger a manual build. An important feature is a secret variable. If you need any passwords in your build that you do not want to display as plain text, you can add them as variables, enter the passwords, and then click the little lock icon. The value will now be shown as ********, and after saving your pipeline you cannot unlock it to see the value. The only way to change the value is by entering it again. The real value will be used in your jobs of course, but any job output will be censored.

On the left you will see pipeline variables, variable groups, and predefined variables. We are currently looking at pipeline variables. You can follow the predefined variables link for a list of variables you can use (like **Build.ArtifactStagingDirectory**). Variable groups are useful, but we will use those when we are going to set up deployment pipelines. They allow you to predefine groups of variables that can be used across builds and deployments. These variables can even be stored in Azure Key Vault so you can use them across applications.

Continuous integration

This is what it is all about—continuous integration! Go to the **Triggers** tab in your build pipeline, and select the **Enable continuous integration** option. This will trigger a build every time you push to the develop branch. You can change the branch or add a branch. You can also exclude branches instead. You can add a path filter, so you can exclude some paths from triggering a commit. You can batch changes while a build is in progress. So, if two teammates make a commit while the pipeline is building, only a single new build is queued. On one hand, this speeds up the building process, but on the other hand, if that build fails, you do not know which of the commits made it fail. So that is a little trade-off.

On the left side, you can also add schedules. For example, you could schedule a build for every morning at 5:00 AM. When you develop on the develop branch, you could schedule builds for the master branch so you are sure it still builds as it should. If you do, be sure to clear the **Only schedule builds if the source or pipeline has changed** check box. By building every day you can be sure no external dependencies broke, like NuGet feeds and packages. You can also chain builds, but since we have no other builds at this time, we cannot configure this.

Save the pipeline, make a commit to your repository, and see the pipeline getting triggered automatically. I am skipping other settings in the build pipeline editor. You get the idea, so you can play around with them. Let us look at private agents next.

Private agents

In our free account, we have room for one private agent. An agent can be a Windows, Linux, or Apple device, so we have some choice there. We are going to make our own computer into an agent. A private agent has a few benefits over hosted agents. First, a hosted agent is always a fresh machine. That means you will have to wait for Azure DevOps to find and clean a machine, which can take time. Second, because it is a fresh machine, you will always have to download your source and restore your packages, which takes time. The benefit of that is that you can be certain your software builds on any machine. With a private agent you do not have to clean your source in between builds, so downloading and restoring can be a lot faster. Your queueing time is probably also faster, so all in all your builds will speed up considerably. You can always choose to clean up your source code on a private agent, too; it is a setting on the agent in your build pipeline. Another huge benefit to private agents is that they are your own servers, so you can install whatever you want. If you need some custom third-party software installed for source code to run, you can.

Installing the agent

Go to your project settings and find your **Agent pools**. Here you can create new pools and manage existing ones. You have two pools already: Azure Pipelines, which is your Azure DevOps hosted pool, and Default, which you can use for private agents. You can add new pools, which can be interesting if you want to limit access to specific pools. You can also add a pool that contains an Azure virtual machine scale set, but that is in preview at the time of this writing and not in the scope of this book. We are going to use the Default agent pool. Click it, and you will find some information on the pool, but it is mostly empty since we do not have an agent in the pool yet. Click **New agent**, and you will see a pop-up that guides you through the setup process. I am going to install the agent for Windows.

Unfortunately, the pop-up gives you just enough information to get started, but not enough to finish it. The first step is simple; download the agent by clicking **Download**. Make sure you put it in your **Downloads** folder.

Next, open Windows PowerShell and browse to the folder where you want to install the agent. For example, entering **`cd C:\`** will install the agent in **C:\agent**. Next, follow the next step in the pop-up.

Code Listing 1

```
mkdir agent ; cd agent
Add-Type -AssemblyName System.IO.Compression.FileSystem ;
[System.IO.Compression.ZipFile]::ExtractToDirectory("$HOME\Downloads\vsts-
agent-win-x64-2.168.2.zip", "$PWD")
.\config.cmd
```

If you did not place your download in the Downloads folder, you will have to change the path in PowerShell.

Next, PowerShell will ask for a URL. Use the URL of your DevOps organization, so for me that is https://dev.azure.com/succinctlyexample/.

Creating an access token

Next, PowerShell will ask for an authentication type. The default is PAT (Personal Access Token) so we will use that. We do not have a PAT yet, but we can create one. Go back to Azure DevOps and in the top right, click the little person with the cog (next to your picture or initials) and then select **Personal access tokens** from the menu. You probably already have some tokens that you were unaware of because the system created those for you.

Click **+ New Token** > **Show all scopes** (at the bottom). The only scope we need is **Read & manage Agent Pools**. Give your token a name (for example, **Private agent token**) and click **Create**. You will now get your token, and you will never see it again—so be sure to copy it. Paste it in Notepad or something just to be safe.

Now that you have your token, you can enter it in the PowerShell prompt.

The next questions are about your pool and agent name, and whether you want to run your agent as a service. Just pick the default options so it will not run as a service, and will not start at startup.

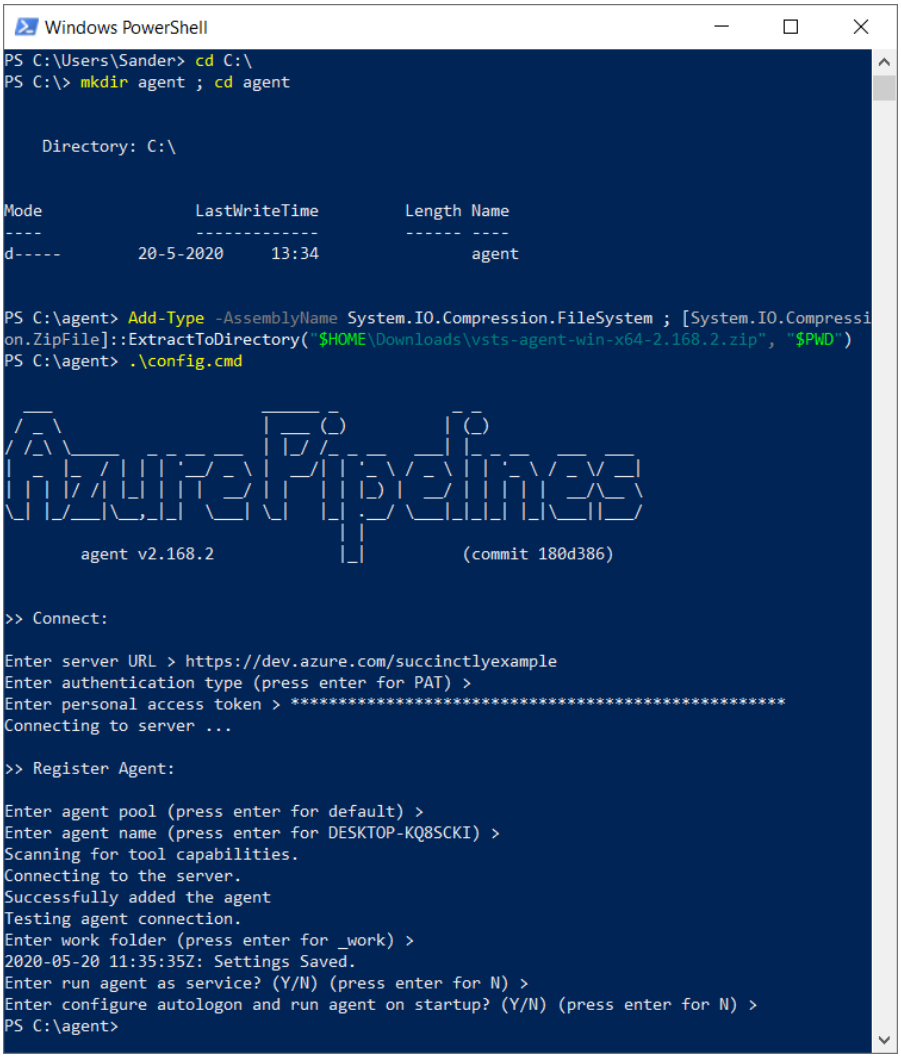

Figure 41: Private agent configuration

You now have a private agent installed.

Running locally

Now go back to your agent pool in Azure DevOps. You will see your agent in the agents list, but it shows as offline. Because the agent is not installed as a service, you have to run it manually. Run it by running **run.cmd** in your agent folder. The status in DevOps will update to online and the status will be idle.

Edit your build pipeline so that it uses your private agent instead, and save and queue a build. The status in your agent overview should change to **Running build x**, and in your local command prompt you should see **Running job: agent job 1**. The build should finish successfully. You can browse to your agent folder and check out the _work folder for your source and artifact (in subfolders 1\s and 1\a).

When you run it again, you will see the speed gain. The hosted build took 1m 4s for me; the first local build took 46s, but the second local build only took 21s. That is three times as fast for a simple project.

You can delete the agent in your agent overview in the agent pool settings. Your pipeline will still have the Default agent pool, so it will fail if you run it. You can simply delete the agent folder on your computer. If you installed it as a service, you should uninstall the service first.

YAML pipelines

We have seen how to build and run pipelines, but we have yet not seen the YAML pipelines that are now default. YAML is a markup language (although YAML stands for “YAML Ain't Markup Language”) that allows you to build pipelines as code. The benefit of using code instead of a designer is that you can now treat your pipeline as any other code file, complete with source control and everything. The downside is that it is a lot harder to write, although copying and pasting becomes easier.

Create a new build pipeline and this time select **Azure Repos Git** when it asks where your code is. Next, select your repository. After that, you can pick a template again. You can choose **Starter pipeline** for a minimal pipeline, or any of the other predefined YAML templates. When you click **Show more** you will find the ASP.NET Core template again. When you click that template, you will be presented with a YAML file that is pretty readable, at least now that we can make a visual representation in our head.

Code Listing 2

```
# ASP.NET Core
# Build and test ASP.NET Core projects targeting .NET Core.
# Add steps that run tests, create a NuGet package, deploy, and more:
# https://docs.microsoft.com/azure/devops/pipelines/languages/dotnet-core

trigger:
- master

pool:
  vmImage: 'ubuntu-latest'

variables:
  buildConfiguration: 'Release'

steps:
- script: dotnet build --configuration $(buildConfiguration)
```

```
  displayName: 'dotnet build $(buildConfiguration)'
```

You will immediately see the problem with YAML. I do not want the latest Ubuntu image; I want the windows-2019 image. We have to make an educated guess and change **ubuntu-latest** to **windows-2019**, and hope that it works. The only reason we know windows-2019 exists at all is because we played around with the designer first. Also, change the trigger to **develop**.

Once you click **Save and run**, you will have to enter a commit message. That is because the YAML file is committed to your main branch. Alternatively, you can commit it on a new branch. Click **Save and run** and watch the build, which should succeed. Do not forget to pull the commit in your local repository. You will notice an azure-pipelines.yml file in the root of your repository. The new build pipeline will have the same name as your repository.

You can create multiple YAML pipelines for a single repository. You will get an azure-pipelines-1.yml file (or -2.yml, etc.) and a new Repository name (1) pipeline will be created. You can rename pipelines once they have been created by clicking on the three-dot menu. When you edit a pipeline in Azure DevOps, you can also associate the pipeline with another YAML file by clicking the three-dot menu in the upper-right corner and going to **Settings**. You can delete a pipeline, but this will not delete the YAML file.

Luckily, once the pipeline is created and you have your YAML file, there is an editor in Azure DevOps that makes it a lot easier to add steps to your file. Azure DevOps does not add a publish task by default, so we will have to add our own. Edit your pipeline and find the **.NET Core task** in the task list on the right. When you click it, you will get an editor like what you are used to. Pick the **publish** command, and you will get the **Publish web projects** and **Zip published projects** check boxes, just like before. When you click **Add**, a complete task is added to the YAML.

Code Listing 3

```
- task: DotNetCoreCLI@2
  displayName: Publish
  inputs:
    command: 'publish'
    publishWebProjects: true
    arguments: '--configuration $(BuildConfiguration) --
output $(build.artifactstagingdirectory)'
```

Notice that only the non-default settings are added to the YAML. I added the **`displayName`** manually.

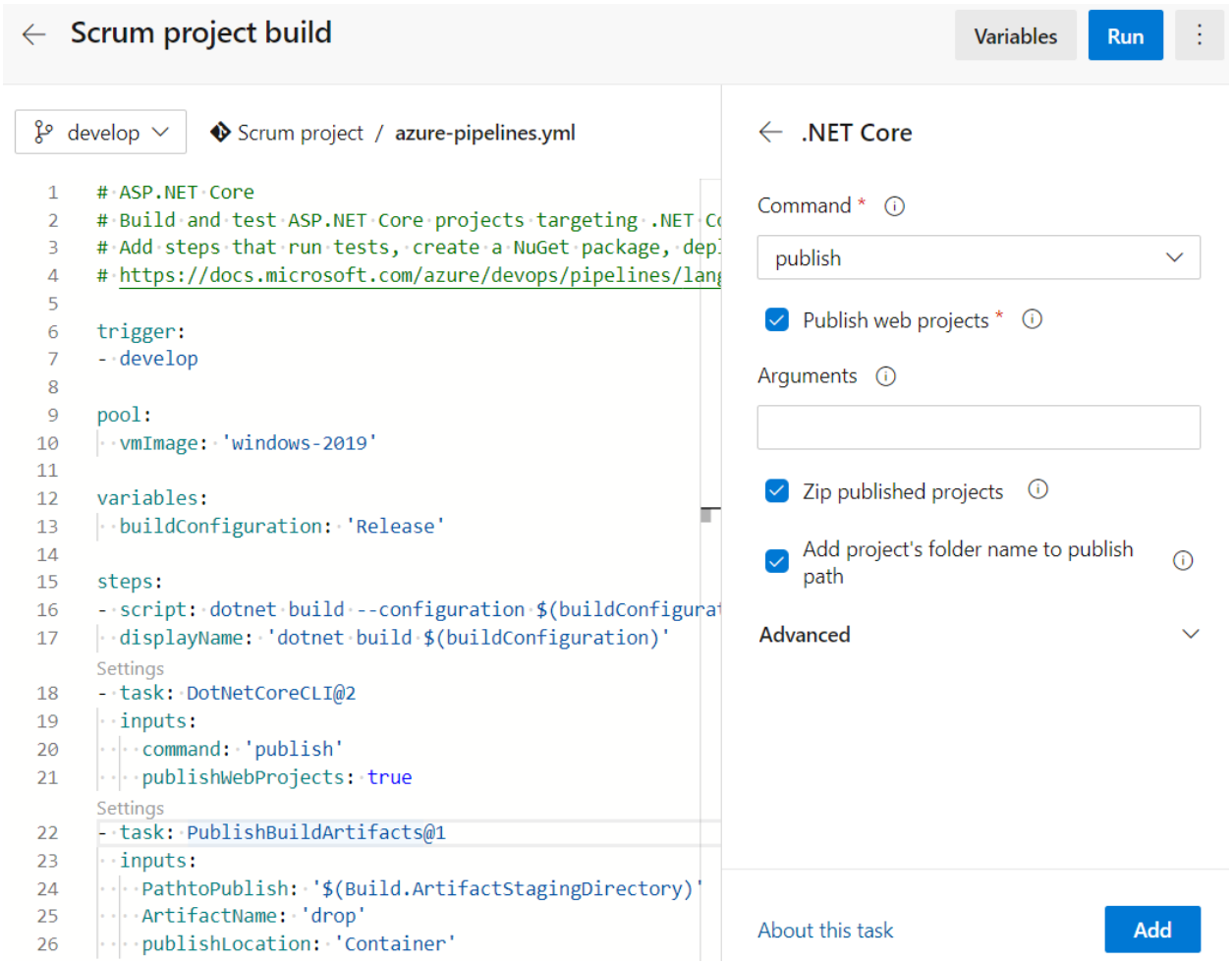

Figure 42: YAML editor

We will also have to add the **Publish build artifacts** task to the YAML, which you can add in the same manner.

Code Listing 4

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

When you save the YAML file, you will make another commit to the develop branch and the build is automatically triggered.

Everything should work as it did, but your pipeline is different now that you build using a script rather than a task that you can configure using the editor. Personally, I just want the same pipeline we had before. Luckily, this is not difficult. Go back to your old pipeline and edit it. In the agent settings, click the **View YAML** button in the top right (next to **Remove**). When you click this, you will get your YAML output. It will give you some warnings that not all parameters are present in the pipeline, but we can fix that. You can add additional variables to your YAML script, like the **`buildConfiguration`** variable.

If you need to add variables that can be set at queue time or are secret, you can add variables in the top right of the editor, next to **Save** (or **Run**). These variables are not added to the YAML file.

The complete script should look something like the following code.

Code Listing 5

```
trigger:
- develop

pool:
  vmImage: 'windows-2019'

variables:
  buildConfiguration: 'Release'
  restoreBuildProjects: '**/*.csproj'
  testProjects): '**/*[Tt]ests/*.csproj'

steps:
- task: DotNetCoreCLI@2
  displayName: Restore
  inputs:
    command: restore
    projects: '$(restoreBuildProjects)'

- task: DotNetCoreCLI@2
  displayName: Build
  inputs:
    projects: '$(restoreBuildProjects)'
    arguments: '--configuration $(buildConfiguration)'

- task: DotNetCoreCLI@2
  displayName: Test
  inputs:
    command: test
    projects: '$(testProjects)'
    arguments: '--configuration $(buildConfiguration)'
```

```
- task: DotNetCoreCLI@2
  displayName: Publish
  inputs:
    command: 'publish'
    publishWebProjects: true
    arguments: '--configuration $(buildConfiguration) --
output $(build.artifactstagingdirectory)'

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

You may now choose to edit the YAML using the Azure DevOps editor, or edit it directly in your local repository. You can find the YAML schema in the documentation.

Security

You probably know the drill by now, but you can manage security for your pipelines in the same way that you can manage it for your projects and repositories. Go to your pipelines overview, click on the three-dot menu, and select **Manage security**. This will open a pop-up where you can set permissions for actions such as deleting build pipelines, deleting individual builds, stopping builds, and viewing builds. Again, you can set these for groups or for individual users.

Summary

With build pipelines you can configure automated builds. In this chapter we worked with an ASP.NET Core project, but we have seen support for JavaScript, Python, Java, Go, and other languages as well. When the hosted build agents cannot satisfy your building needs, you can always install a private agent. With YAML you can configure your pipelines as code, making them manageable in your source control management.

Chapter 6 Release Pipelines

In the previous chapter, we looked at build pipelines. In this chapter we are staying with pipelines, but we are going to construct release pipelines. A lot of what we saw in the previous chapter can be used in this chapter. Similarly, some topics from this chapter can be applied to the previous, but I have left them for this chapter because they are more relevant here.

The pipeline editor

Go to **Releases** in the left-hand menu and click **New pipeline**. The first step in the editor is selecting a template. We get no YAML choice this time. Next to the **Empty job** at the top, you will find a lot of Azure App Service deployments. There are default templates for deploying Java, Node.js, PHP, Python, Go, and Ruby on Rails applications to Azure App Services. You will also find Azure Service Fabric and Azure Functions deployments, and IIS and SQL database deployments. That is all very Microsoft and Azure centered, but you can deploy to AWS, Google Cloud, other platforms, and your own servers as well. For now, choose the **Azure App Service deployment** template, which will deploy a .NET application to Azure. Next, you can enter a stage name and owner. Change the stage name to **Development** (the D in DTAP).

You now have a form on the left side with an empty rectangle for artifacts, and on the right side a rectangle for stages, with another rectangle for the Development stage we just created. Before we forget, change the name of your pipeline. Click **New release pipeline** at the top, and change the name to something like **Scrum project release**.

Next, click **+ Add an artifact**. You can now pick a build, an Azure DevOps Repository, a GitHub repository, or a TFVC (Team Foundation Version Control) repository. By clicking **5 more artifact types**, you can choose additional artifact types, Jenkins among them. Since we have a build, we are going to keep that. In the drop-down you can select your build, which should be **Scrum project build** (if you picked the Scrum project's default repository for the YAML build in the previous chapter). You can then choose a default version, which defaults to the latest. You can deploy specific versions, but we want the latest, so keep that. You can also enter a source alias, which is the name of the artifact in the deployment pipelines. This is usually your repository name prefixed with an underscore, so enter **_Scrum project build**. Click **Add** and the artifact will be added. This pipeline now has access to any artifacts the selected build produces. That is why it was so important to publish an artifact in the previous chapter.

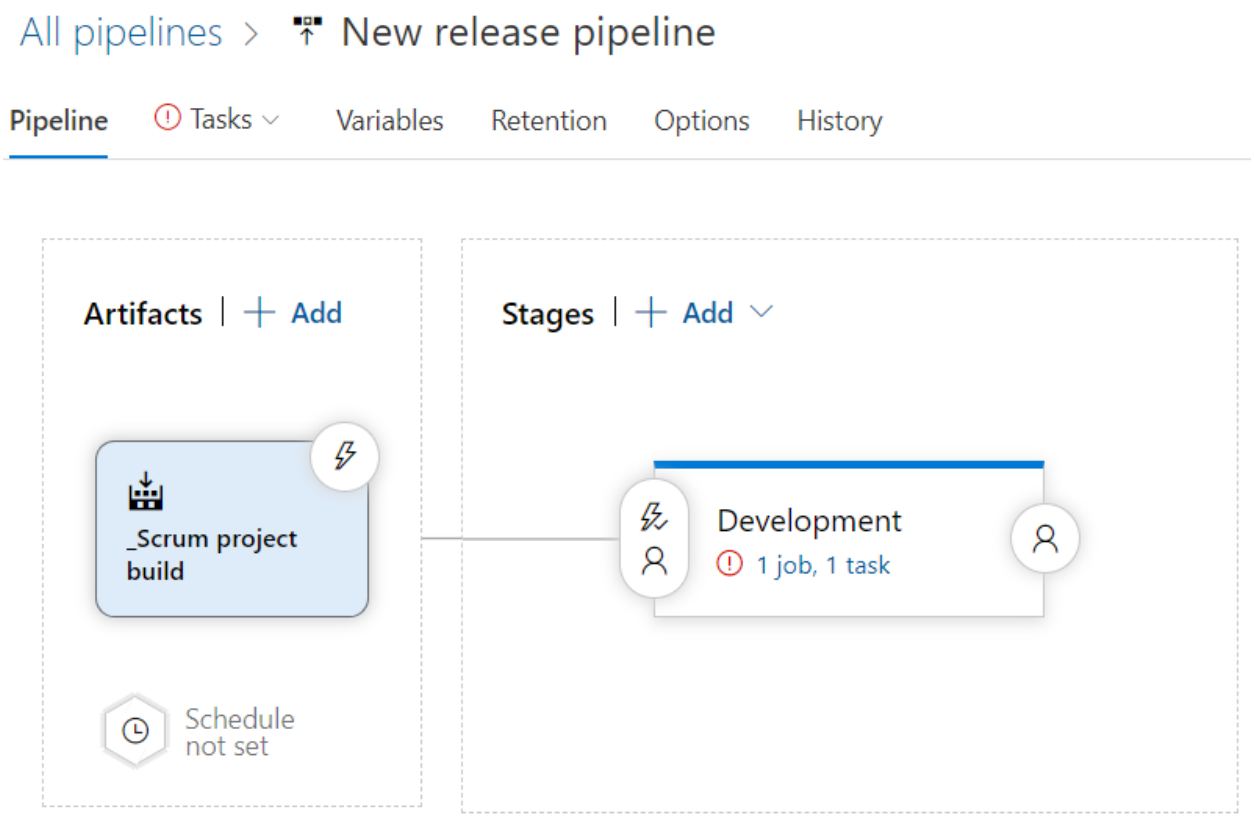

Figure 43: New release pipeline

As you can see, there is a red exclamation point next to your tasks. Click **Tasks** at the top, or **1 job, 1 task** on the Development stage. This will take you to an editor that is not unlike the classic build pipeline editor. You will see some stage settings and parameters. These are linked parameters, which work the same as in the build pipelines. You will need to specify an Azure subscription and an app service name. The subscription is easy; just open the drop-down and you will see all subscriptions that you have access to with your current account. That should be your free account that we created at the start of this book. When you select the subscription, you have to authorize it before you can use it. You can do this by clicking the **Authorize** button that appears after choosing your subscription. The app service name is a bit of an issue since we do not have an app service yet.

Ignore the app service setting for now, and look at the other settings. We have the app type, **Web app on Windows**, which is fine for us. You will notice that the agent settings are mostly the same as in the build pipeline. The **Deploy Azure App Service** task is where the magic happens, though. There is the subscription, app service type, and app service name that are linked to the stage variables, so we have seen those. Next, there is a **Deploy to Slot or App Service Environment** setting. This is specific to Azure, so it is not in the scope of this book. The same goes for the **Virtual application setting**. The next setting, **Package or folder**, is relevant for us. This is the file that the task copies to your App Service. The task expects this to be a zip file, which is also the default setting in the build pipeline.

When you look at the Additional Deployment Options, you will see a **Select deployment** check box. This is not selected by default, meaning the task will decide for you based on the files it gets. When you select it, you will get to choose from Web Deploy, Zip Deploy, and Run From Package. You can also set some additional settings, like if additional files need to be deleted from the App Service, if you want to take the App Service offline, and if you want to exclude your App_Data folder. You do not need these settings now, but I wanted to point them out to you because you may need them in some cases.

Another interesting section is Application and Configuration Settings. In fact, .NET Core loads the config.Environment.json configuration file based on the ASPNETCORE_ENVIRONMENT environment variable. We can set that here. Click the three-dot button (**…**) next to the App settings and click **Add**. Now, enter **ASPNETCORE_ENVIRONMENT** for name and **Development** for value. This should make sure that variable is added to your App Service environment.

Unfortunately, we cannot save our pipeline until we fix the App Service name issue. Go to your Azure portal (by browsing to https://portal.azure.com). There, click **+ Create a resource**, search for **Web App**, and create one. In the settings blade that pops up, click **Create new** for **Resource Group** and name it **development**. The name of your web app should be something unique, like **sander-succinctly-dev-app** (use your own name). For **Runtime stack**, select .**NET Core 3.1 (LTS)**. Choose the region that is closest to you. Click **Next** and disable **Application Insights** on the next tab. Then click **Review + Create** > **Create**.

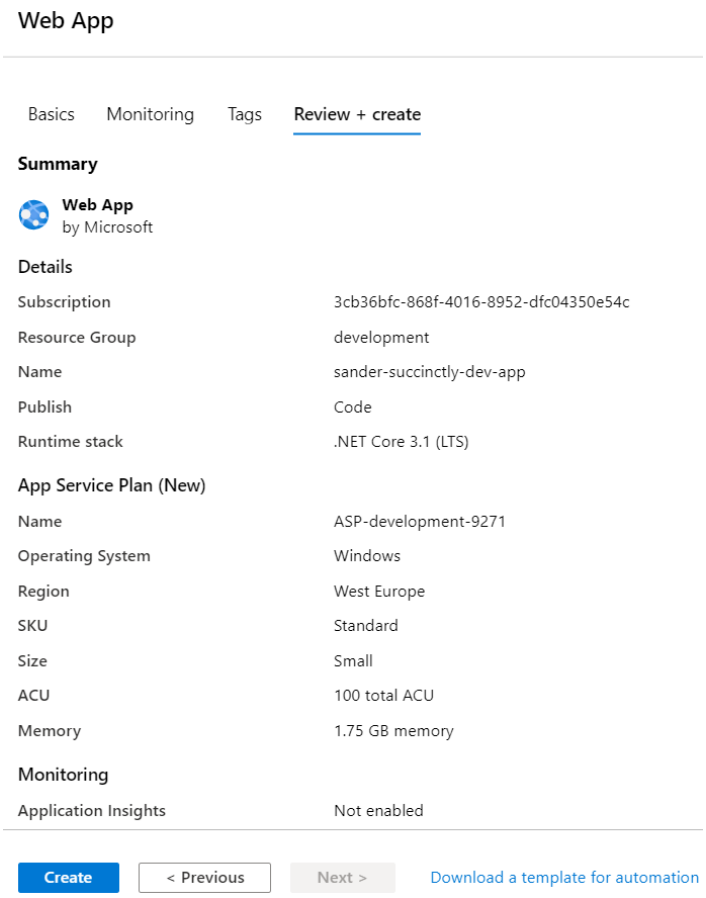

Figure 44: Web App summary

Once the web app is created, you will see a **Go to resource** button, which will take you to the web app settings in Azure. Click the URL at the top of the **Overview** page, which is open by default. If you cannot find it, the URL should be https://[*your web app name*].azurewebsites.net. If all went well, you should see a default web app page.

Now, go back to your release pipeline, open the **App Service name** drop-down, and select your web app. Next, click **Save** at the top right to save the pipeline, and then click **Create release**. In the blade that opens, click **Create**. A message should appear on the top of the screen, which you can click to go to your release overview.

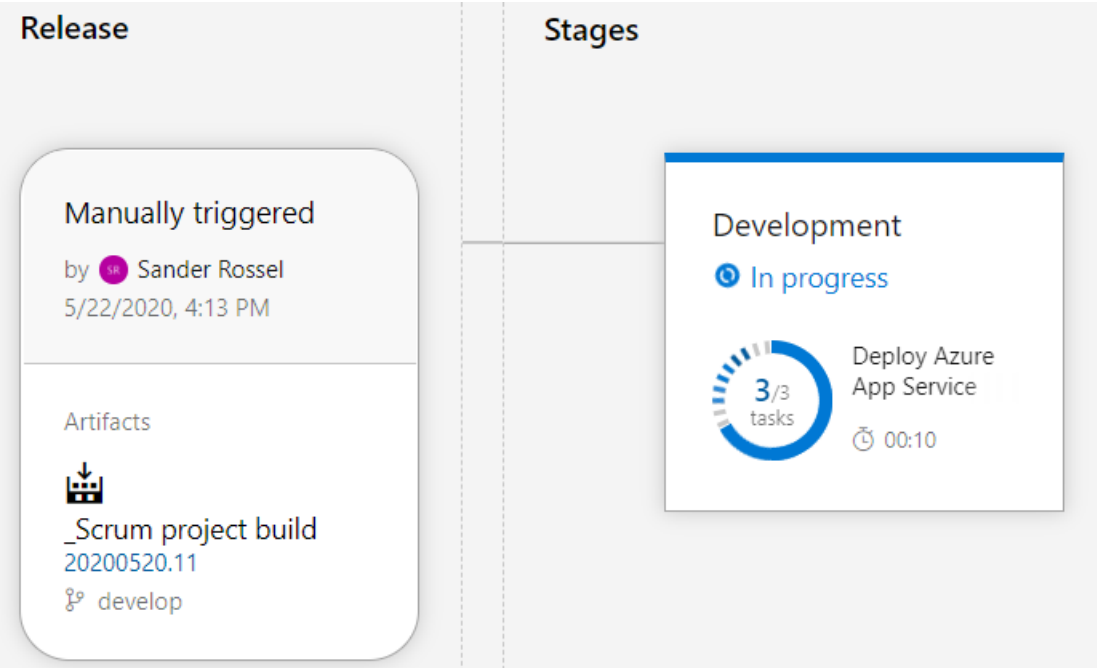

Figure 45: Release overview

Hover over the stage and click the **Logs** button, which opens a logging screen similar to the build logging. When the release is finished successfully, go back to your web app URL and refresh the page. It should now show you the ASP.NET Core default project page.

If you go to the Azure portal, find the web app and go to **Configuration**, where you will also find the ASPNETCORE_ENVIRONMENT setting that we added.

Deploying Azure resources

We have now automated the deployment of the software, but not the environment. Azure DevOps can deploy Azure resources using ARM (Azure Resource Management) templates, PowerShell, or the Azure CLI. First, let us deploy the resource group using the release pipeline.

The easiest way to create a new resource group using Azure DevOps is by using the Azure CLI. Go back to your editor and click the **+** button on the **Run on agent** job. You can now add a task. These are the same tasks that we saw in the build pipeline, but some of them make a lot more sense in a deployment. Go to **Deploy** and find the **Azure CLI** task. If you cannot find it, use the search field at the top right. Add the task and drag it above the Deploy Azure App Service task.

In the Azure CLI task, you will need to specify your Azure subscription again. This time, when you open the drop-down, you will see **Available Azure service connections**. I will get back to that in a minute, but for now select the connection and not the subscription. For script type, select **Shell**, and change **Script path** to **Inline script**. The script is a one-liner.

Code Listing 6

```
az group create --name "development" --location "West Europe"
```

If you save and run now, you will find that the release does nothing. That is because the deployments are incremental and so can be run over and over. Only changes will be deployed.

Deploying ARM templates

Deploying the web app is a bit trickier. You will have to create an App Service and an App Service plan, which determines your memory, cores, features, and costs. We went with the default before, so Azure generated a name for the plan. First, let us see how we can create both. You can use the Azure CLI again, but I want to use an ARM template instead. ARM stands for Azure Resource Management. These templates are written in JSON, and you can generate templates by going to your Azure resource and going to **Export template**. Unfortunately, these templates are not always pretty, and you usually have to edit them manually. Microsoft also has a GitHub repository with example templates. One of these templates does almost what we want: the 101-webapp-basic-windows template. We can do three things: add an additional GitHub artifact (you need a GitHub account for this), create a new repository, and add the file there (or add the file to our existing repository). I always prefer to have a separate repository with all my ARM templates.

So, go to your Repos and create a new one named **ARM Templates** with a VisualStudio gitignore. I do not usually do this, but add a new file and name it **WebAppTemplate.json**. Then copy the contents of the file in GitHub to your DevOps file. Unfortunately, the template does not work because the location variable is invalid, and we also do not want to create a new App Service plan; we want to use our existing one. So, remove the **location** parameter and change the parameter references to **[resourceGroup().location]**, add an **appServicePlanName** parameter, and remove all variables. Also replace the variable references to their respective parameters. That is a bit complicated, but the result should look as follows.

Code Listing 7

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-
01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
```

```
  "parameters": {
    "webAppName": {
      "type": "string",
      "metadata": {
        "description": "Base name of the resource such as web app name and
app service plan"
      },
      "minLength": 2
    },
    "sku":{
      "type": "string",
      "defaultValue" : "S1",
      "metadata": {
        "description": "The SKU of App Service Plan, by default is Standard
 S1"
      }
    },
    "appServicePlanName": {
      "type": "string"
    }
  },
  "resources": [
    {
      "apiVersion": "2018-02-01",
      "type": "Microsoft.Web/serverfarms",
      "kind": "app",
      "name": "[parameters('appServicePlanName')]",
      "location": "[resourceGroup().location]",
      "properties": {},
      "dependsOn": [],
      "sku": {
        "name": "[parameters('sku')]"
      }
    },
    {
      "apiVersion": "2018-11-01",
      "type": "Microsoft.Web/sites",
      "kind": "app",
      "name": "[parameters('webAppName')]",
      "location": "[resourceGroup().location]",
      "properties": {
        "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', parameter
s('appServicePlanName'))]"
      },
      "dependsOn": [
        "[resourceId('Microsoft.Web/serverfarms', parameters('appServicePla
nName'))]"
      ]
    }
```

```
  ]
}
```

Now go back to your pipeline and add a new artifact. Because we do not have to build the JSON template, we can just add an Azure Repos Git type directly. Select the repository, the master branch, and the **Latest from the default branch** default version.

Now, go back to your tasks and add an **ARM template deployment** task. Make sure it is between your Azure CLI task and the Azure App Service build. In the ARM template deployment task, we have to select the available Azure connection again, as well as a subscription. Leave the action as **Create or update resource group**. For **Resource group**, choose **development**. Select the location you used for your web app in Azure. In the **Template** field, you can browse your artifacts for the ARM JSON template, so click the three-dot button and select it. Skip the **Template parameters** field because we do not have a separate parameters file. In **Override template parameters**, click the three-dot button and enter the name of your web app (for me, that is sander-succinctly-dev-app) and the name of your App Service plan, which you can find in Azure. Click **Save** and start a new release. If everything went well, the release succeeds, and nothing happened. Check Azure to see that no additional resources were created and that your web app still works.

Adding a new stage

At this point, we should be able to release the entire development environment from scratch—that is the resource group, the web app, and the App Service plan. Let us confirm this by creating a new environment. Go to your pipeline and click the **Pipeline** tab. If you hover over the **Development** rectangle, you should see two buttons, **Add** and **Clone**. Click **Clone** and rename **Copy of Development** to **Test**. This should create a new stage that is linked to the Development stage. To remove the link, click the little lightning icon on the **Test** stage and change **After stage** to **Manual only**. If you always want to release the Test stage after a build, you can also select **After release**.

When you now hover over the **Tasks** tab at the top, you will be able to choose between **Development** and **Test**. Go to the tasks of your Test stage. We now have to change everything that relates to a specific environment. For example, the name of the resource group should be test and not development. I have a better alternative, though.

Variables

Click the **Variables** tab. This looks the same as the build variables. Add the following variables.

Table 1: Development variables

Name	Value	Scope
Environment	Development	Development
ResourceGroupName	development	Development
ServicePlanName	[Name of your plan]	Development
WebAppName	[Name of your web app]	Development

Now replace all those values in your tasks with the variable. In your deployment process parameters, replace your App Service name with **$(WebAppName)**. In the Azure CLI task, replace **development** with **$(ResourceGroupName)**. In your ARM template deployment, set **$(ResourceGroupName)** as the value in the **Resource group** field, and replace the web app name and the service plan name in your template parameters. And in the **App settings** of your Azure App Service Deploy, replace **Development** with **$(Environment)**.

All these variables will now be replaced with their values at release time. Now go back to the **Pipeline** tab and delete the **Test** stage by clicking on it and clicking **Delete** in the blade. Now clone the Development stage again, again renaming the clone to **Test** and setting it to **Manual only**. The new Test stage will now also reference the variables, but if you go back to the **Variables** tab, you will see that all variables have been clones too. That means you only need to set the values for the new variables, and you are good to go! So set the values to the following.

Table 2: Test variables

Name	Value	Scope
Environment	Test	Test
ResourceGroupName	test	Test
ServicePlanName	test-plan	Test
WebAppName	sander-succinctly-test-app	Test

Save your changes and create a new release. When creating the release, you now see two stages. Click **Development** to cancel its selection and click **Create**. Your release will not start the Development stage now because you have deselected it. The Test stage will not start either, because you have to manually start it. So, go to your release overview and deploy the test stage manually. When all goes well, the release succeeds, and you should see a new test resource group in Azure with an App Service and an App Service plan. Browsing to the new web app should show your web app. Creating an Acceptance and/or Production environment should now be easy. The new stages are shown in your releases overview.

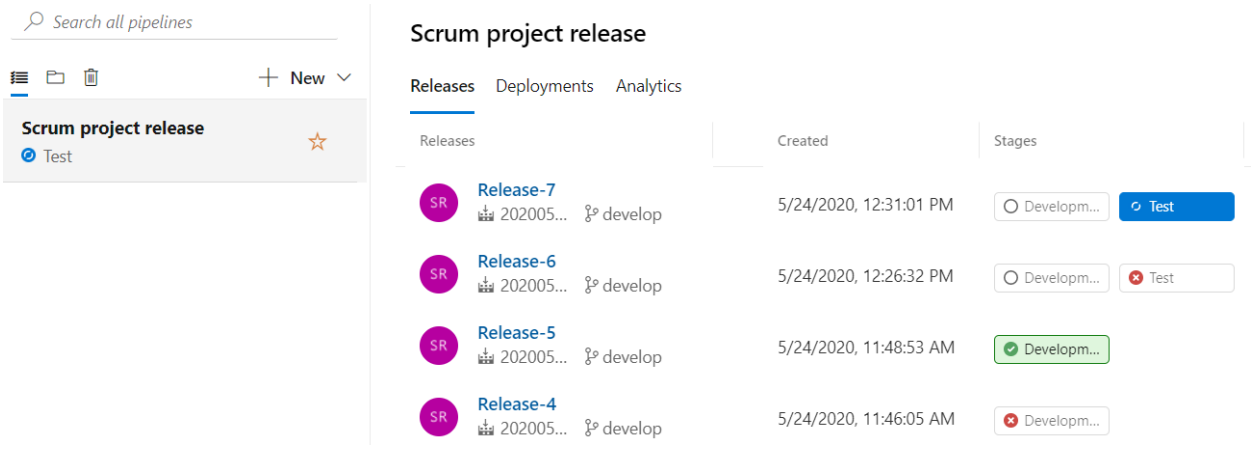

Figure 46: Releases overview

You can have different tasks per stage in your release pipeline. The clone can be a handy starting point, but they do not have to be the same.

Approvals

One thing you may want to add to your stages is a pre- or post-deployment approval. You can go from one stage to the next fully automated, or you can trigger the next stage manually, but either way you may want to limit who can trigger releases. When you go to the pipeline editor, you can click on the little person icon on the left or right of a stage rectangle. When you do, a blade opens, and you will be able to enable pre- and post-deployment approvals.

Service connections

By building our pipelines, we created an Azure connection. This was done for us automatically, but we can also create one manually or edit the automatically generated one. Creating one ourselves goes a bit outside the scope of this book, but I want to point it out. Go to your project settings, and then go to **Service connections**. This is where you should see a list of your connections. We currently have one, Free Trial ([*subscription ID*]). When you click it, you can edit it with the button in the top-right corner. You cannot do a lot with it, and I suggest you only change the name and description, because if you mess this up you can reset all connections in all your pipelines. Any GitHub connections will be shown here as well.

Go back to the Azure portal and check your Azure Active Directory. Go to **App registrations**, and you will see a registration that has the name of your DevOps organization and project with an ID. This is how Azure and Azure DevOps are linked together. By deleting the registration here, you invalidate the link. If you want to manually create a service connection, you will also have to create an app registration here.

Variable groups

One cool feature of Azure DevOps pipelines is variable groups. This is a predefined collection of variables that you can use in build and release pipelines. What is even better, is that you can link it to an Azure Key Vault, which gives you the ability to manage secrets in Azure, complete with Azure AD access management, and then use them in Azure DevOps.

To create a variable group, go to **Library**, under **Pipelines**, and click **+ Variable group**. Name the variable group whatever you like, for example **Secrets development**. You cannot have multiple stages in a variable group, so if you want some stage-specific secrets, you will have to create multiple variable groups. You can now add a variable (for example **SqlServerPassword**), give it a value, and click the lock icon. Save the variable group.

Now, go back to your release pipeline and edit it. Go to the **Variables** tab and click **Variable groups**. You can now link a variable group. You can link it to either the release scope (meaning the variables and their values are shared across stages) or a specific scope, for example **Development**. If you link the variable group to Development, the **SqlServerPassword** variable is now accessible from the Development stage tasks. The value must still be changed in the variable group. If you delete the variable from the group and you still use the variable in your release, the release will break. If multiple pipelines use the **SqlServerPassword** variable and you change your password, you only have to change it in the variable group and all pipelines will use the new password.

Linking an Azure Key Vault

Azure Key Vault is a great way to manage secrets. I use it for my Azure applications all the time. Go to your Azure portal, find all services, and search for **Key Vault**. Create a Key Vault in your **development** resource group and name it [*your name*]**-succinctly-kv-dev**. Naming is a bit of a hassle because it cannot be more than 24 characters, and has to be unique across Azure. Again, choose the region closest to you. Click **Review + create** > **Create**. Once the Key Vault is created, go to the resource, go to **Secrets**, and click **+ Generate/Import** to create a secret. Create a **SqlServerPassword** secret and give it any value. Go back to your variable group and enable **Link secrets from an Azure key vault as variables**. This will give you a warning that your current secrets will be erased, but continue anyway. You now have to select a subscription and a key vault, and authorize it. Once it is authorized, you can click **Add** and select the secrets you want to link. Select the **SqlServerPassword** secret you created, and it will be linked. The SQL Server password is now managed in Azure Key Vault.

When you go back to your Key Vault and go to **Access policies** and refresh it, you will see your DevOps application registration now has Get and List permissions on your secrets.

Task groups

Another nice feature of Azure DevOps is task groups. They are especially useful in release pipelines where YAML is not an option. Go back to your release pipeline editor and go to your tasks. The Azure CLI task to create a resource group is pretty neat. You may want to reuse that, but you do not want to remember or look up the correct CLI commands to create one. Instead, you can create a task group. This groups one or more tasks together so that you can reuse them in other pipelines. Right-click the task and select **+ Create task group**. You are prompted to give the task group a name, description, and category, and to configure parameters. Name it **Create resource group** and delete the default value for the **ResourceGroupName** parameter. Then click **Create**. Your task is now converted to a task group, and you have to enter the **ResourceGroupName** parameter. Give it the value **$(ResourceGroupName)** and save.

Right-click the task group and select **Manage task group**. This will open your task group, which has the Azure CLI task. There are two caveats. First, the variable **ResourceGroupName** will be used as the public display name for the variable in your task group. **ResourceGroupName** is not a nice name for a variable, so change it to **Resource group name** so it looks good in the UI. The second is the Azure subscription, which is not a variable at all. Trying to make this variable may prove difficult, but if you use the variable **$(AzureSubscription)**, you will find that it works. The name will be **AzureSubscription** instead of **Azure subscription**, but the description is copied to your variable.

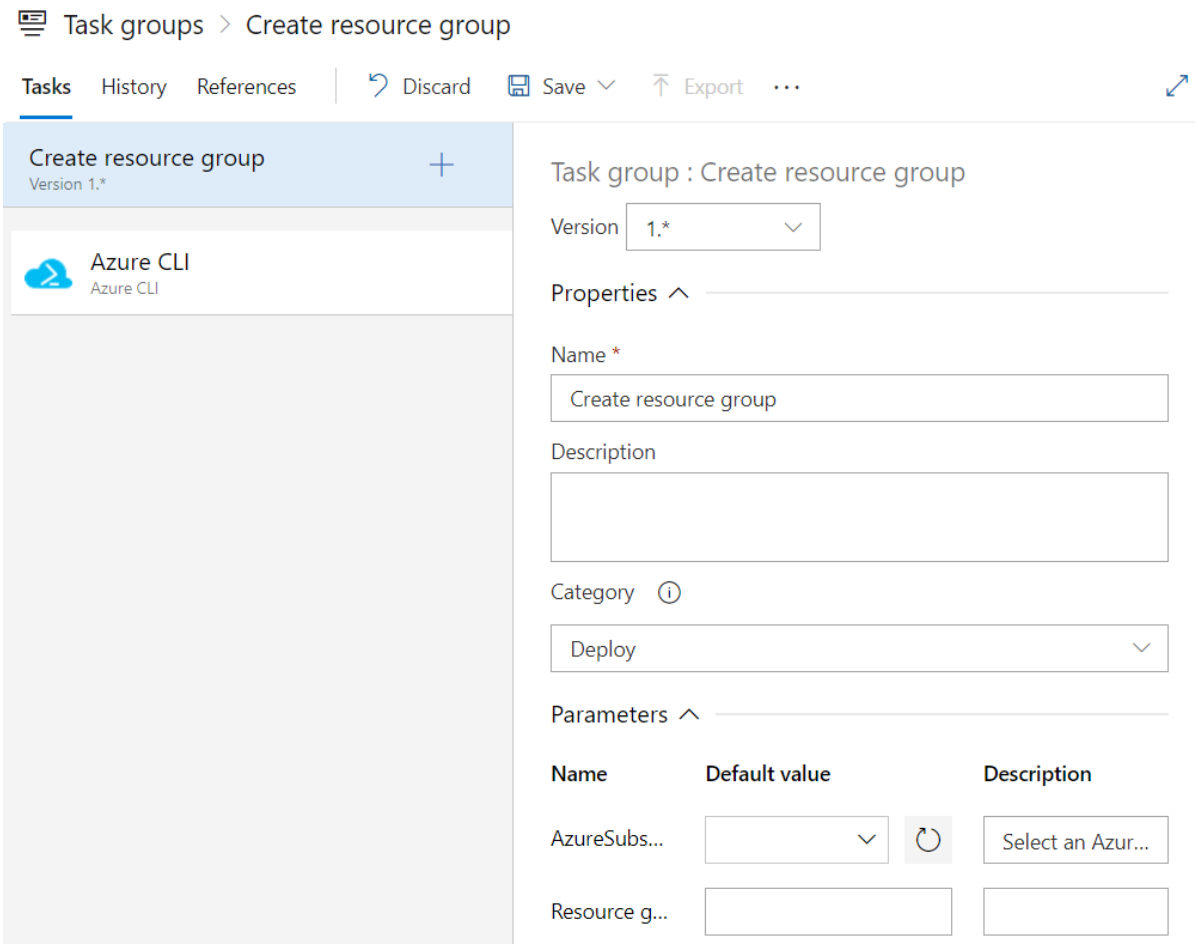

Figure 47: Task group

Save the task group and go back to your release pipeline where it is used. You can find this under the **References** tab in your task group. You will find that the pipeline broke because the task group definition broke. Add the Azure subscription and resource group name, and it should work again.

To prevent your pipeline from breaking, you can save a task group as a draft. Simply click **Save** and select **Save as draft**. This will create a draft that you can use in pipelines. However, when you publish the draft it will be deleted, so it is not recommended to use the draft. Once you have a draft, you can publish the draft. The **Publish draft** button is next to the Save button, and it is only visible when you have a draft. When you publish, DevOps will warn you that your previous task will be overwritten unless you check **Publish as preview**. So, check the box and click **Publish**. You will now get a version 2.*. When you go back to your pipeline, you will now be able to select a new version, which is labeled **Preview**. You can publish the preview by going back to your task group and clicking **Publish preview**, which is at the same location as the **Publish draft** button. Once you have published the preview, you have a definitive version 2.*.

You can select multiple tasks and make them into task groups, you can edit additional tasks to a task group, and you can even use task groups inside your task groups. Your task groups will be added to the list of tasks, so you can add them to any pipeline by simply adding a task and finding your task group.

Deployment groups

Besides deploying to your cloud environment of choice, you can deploy to your own on-premises servers. To do this, you have to create a deployment group and register a server to that group. So, go to **Deployment groups** in the left-hand menu and click **Add a deployment group**. You can now enter a name for your group (for example, **succinctly-servers**) and a description. The idea is that you can group servers into logical groups for deployment, but I am assuming we only have our non-server laptop or desktop available.

Once you have created the deployment group, you will be taken to the Group Management page. Here, you will find a PowerShell script that you can run on your computer. Copy the script, run a PowerShell prompt as administrator, and paste the code. When it asks you for an authentication type, go for the default, which is **Personal Access Token**. For the next step we need to create another access token, so go to your access tokens and create one that has **Read & manage** access on **Deployment Groups**. Enter the generated token in your PowerShell prompt and proceed with all the defaults. When you return to your deployment group in Azure DevOps, you will find your computer in the **Targets** tab. Clicking it allows you to add custom tags.

For simplicity and clarity, my example is going to be a bit contrived. To do any real deployments to your computer is not feasible, and not in the scope of this book. Go to your release pipeline and create a new stage. Now, remove the agent job that is created by default. Instead, click the three-dot button on your stage and click **Add a deployment group job**. In the job, add a task and search for the **Copy files** task. In the task, enter **$(System.DefaultWorkingDirectory)/_Scrum project build/drop/** in the source folder (or browse using the three-dot button) and any local path in the target folder, for example **C:\Temp\Deployment**. The local path does not have to exist; the task will create it for you. Then simply save and run the stage. The stage will run on your local computer and when it is done, you will find your artifact in the Deployment folder.

Using deployment groups allows you to run releases on your own servers. For example, when you have websites running in IIS, you can simply copy your files to the corresponding IIS folder. You can also run batch or PowerShell scripts on your machine to do additional tasks. With the **Required tags** property in the job, you can filter your servers, so you are certain that an application is released to the correct server.

Creating a deployment group will also create a deployment pool, which can be found in your organization settings. To completely remove this, you have to delete your computer from the deployment group, delete the deployment group, and then delete the deployment pool. You can also delete the Azagent folder from your computer, which is in C:\ by default.

Environments

Last, but not least, are environments. This is by far the least-obvious feature of Azure DevOps. When you go to **Environment** from the menu, you can create an environment with a name and description, and that is about it. You can add a Kubernetes namespace or a virtual machine resource, but whatever you're supposed to do with it is not obvious as this point.

Remember how you cannot use YAML for release pipelines? This right here is the solution. A build supports YAML, but does not have stages, while a release has stages, but not YAML. Environments have both. We are already halfway to using them. The trick is to create a YAML file like you do for a YAML build pipeline, but format it slightly differently.

First, you have to define stages. Within those stages you define jobs. The jobs contain the steps. Also, make sure you use the **Publish Pipeline Artifacts** step, and not the **Publish build artifacts** step. The trick is in the deployment stage. You specify a deployment, and in the deployment you specify an environment. The environment is created when it does not exist. Next to the environment, a deployment has a strategy, which defaults to **runOnce**. Under **runOnce** you get your deploy and preDeploy nodes, which contain steps.

A complete YAML file that builds the software and deploys the artifact to an (already existing) app service looks as follows.

Code Listing 8

```
trigger:
- master

stages:
- stage: Build
  variables:
    buildConfiguration: 'Release'
    restoreBuildProjects: '**/*.csproj'
  jobs:
  - job: Build
    pool:
      vmImage: 'windows-2019'
    steps:
    - task: DotNetCoreCLI@2
      displayName: Build
      inputs:
        projects: '$(restoreBuildProjects)'
        arguments: '--configuration $(buildConfiguration)'
    - task: DotNetCoreCLI@2
      inputs:
        command: 'publish'
        publishWebProjects: true
        arguments: '--configuration $(buildConfiguration) --
output $(build.artifactstagingdirectory)'
    - task: PublishPipelineArtifact@1
      inputs:
        targetPath: '$(build.artifactstagingdirectory)'
        artifact: 'drop'
        publishLocation: 'pipeline'
- stage: DeployDev
```

```
  variables:
    appName: 'sander-succinctly-dev-app'
    environment: 'Development'
  jobs:
  - deployment: Development
    pool:
      vmImage: 'windows-2019'
    environment: 'succinctly-webapp'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: AzureRmWebAppDeployment@4
            displayName: 'Deploy Azure App Service'
            inputs:
              ConnectionType: 'AzureRM'
              azureSubscription: 'Free Trial (3cb36bfc-868f-4016-8952-
dfc04350e54c)'
              appType: 'webApp'
              WebAppName: $(appName)
              packageForLinux: '$(Pipeline.Workspace)/**/*.zip'
              AppSettings: '-ASPNETCORE_ENVIRONMENT $(Environment)'
```

If you need an additional stage, like test, acceptance, or production, you can simply copy the **DeployDev** stage and paste it at the bottom. You can then change the name and variables to reflect the new stage.

When you save and run the YAML file, you will see the pipeline running in the build pipelines, and you can find it in your environments.

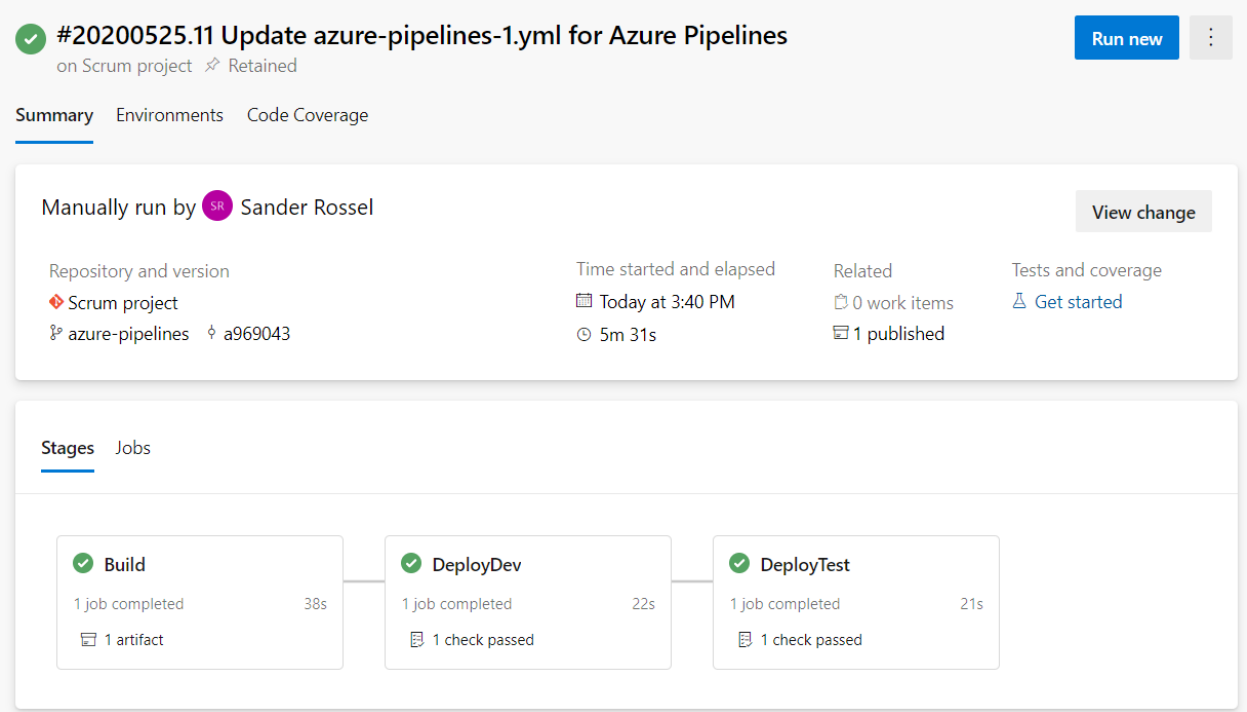

Figure 48: Multistage pipeline

Unfortunately, these environment pipelines, or multistage pipelines, are quite new and do not have all the functionality you would expect—most importantly, manually triggering stages. It is a common scenario to automatically build and deploy to a development and/or test environment. After a few days you do a manual release to a user acceptance environment, and a few days later you do a manual release to production. There are some techniques to work around this limitation, like working with conditions, approvals, or using different environments. But all of them are workarounds, and for that reason I still prefer the classic release pipelines. Having a manual trigger is a much-requested feature, so hopefully Microsoft will implement it soon.

You can set approvals in multistage pipelines, but this is not obvious, and applies to all deployments in a single pipeline. When you go to an environment, click the button in the top-right corner, and you can select **Approvals and checks**. When you add an approval, all deployment stages have to be approved before they run.

Summary

Using Azure DevOps, you can create release pipelines using a relatively easy editor. Releasing your applications to Azure is supported out of the box. Secret management using variable groups and Azure Key Vault adds an extra layer of security. Using deployment groups allows you to deploy to your own servers. With environments, YAML is supported for releases as well.

Chapter 7 Test Plans

In the previous chapter, we looked at planning work and releasing to various environments using pipelines. An important aspect of all software development is testing. We have seen tests on boards, and when creating pipelines we can take into account that software has to be tested before it gets released to the next stage. In this chapter we are going to look at Azure DevOps Test Plans, which help testers manage their test plans.

> ***Note: You may have noticed Azure DevOps has a menu item named** Load tests**. At the time of this writing it is still available, but it is deprecated and will be unavailable soon. Maybe the menu item will no longer be available when you read this, but in case it is, you now know you should not use it, and I am not going to discuss it in this book.***

Enabling Test Plans

When you click **Test Plans** in the menu, you will either be taken to your test plans, or you will see a getting started page. It depends on what you did in Chapter 3, "Boards." If you created any test tasks, you would see them here. If you did not, you will see a welcome page that explains how test plans work. I recommend you go to a project that has no tests yet so you can explore the welcome page, which has some good information. Make sure to watch the Test & Feedback browser extension video. It is less than two minutes, and I will not be discussing the extension in this book.

> ***Note: If you already have an Enterprise, Test Professional, or MSDN Platforms subscription, you already have full access to all testing capabilities, and this section does not apply to you.***

As you can read on the welcome page, you have to upgrade to get full test management capabilities. To get a 30-day free trial, go to **Billing** in your organization settings. Here you can click **Start a free trial**. You do not have billing set up to use this. During the trial all your Basic users will have full access to Test Plans. After the trial expires, you will need to buy Basic + Test Plans and assign it to specific users for them to continue using it.

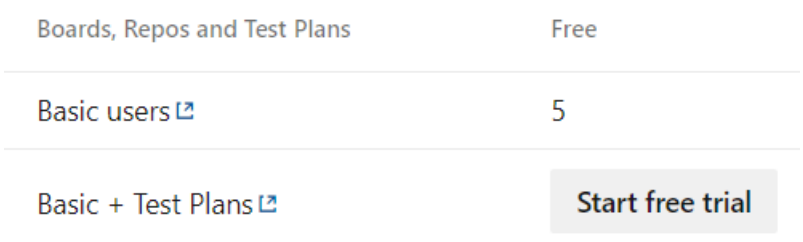

Figure 49: Enable Test Plans free trial

When you go back to your test plans, you should not see the welcome page anymore. Instead, you will be able to create a new test plan, and you will find some new items in the menu as well.

Before we continue, I would like to take a quick look at the free offering. We cannot disable the free trial, so for this organization, we are stuck. However, it is easy to quickly create a new organization with a basic project. Simply go to the overview page of your organization and click **New organization**. I have named my new organization testplanstest, but the name has to be unique, so choose any unique name (for example, by adding a number or your name). Next, create a project with the Agile work item process. We now have two organizations, one with full test management capabilities, and a free one with limited test management capabilities.

Creating a test plan on a free account

This section is for the free offering, but it also works on paid accounts. Go to your Boards and create two new product backlog items. Create three test tasks for each of them.

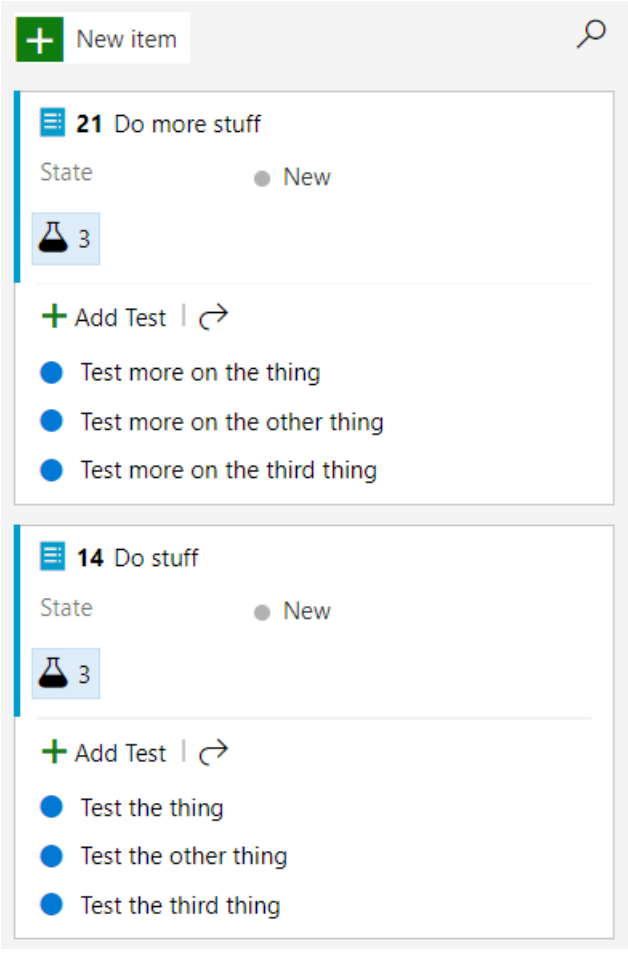

Figure 50: Some tests on a Scrum board

Now, when you go to your test plans, you will see a test suite with some tests on the board. Here you can pass or fail tests. You can do this manually or run automated tests. You can run tests manually and just set whether the test has passed or failed. You can also select **Run for web application**, **Run for desktop application**, or **Run with options**. For desktop applications, you need to download a test runner. When you select **Run with options**, you get some additional options, like automated testing with a release stage, and testing with Microsoft Test Manager 2017 Client or 2015 or earlier, which are not in the scope of this book. **Run for web application** allows you to add comments and attachments to tests and create bugs or add to existing bugs. You can also record your screen or add screenshots, but you will need the extension I mentioned earlier. You can do all of this directly in the test task or using other applications, like the Snip & Sketch tool that comes with Windows 10. When you do any of these test runs, you can review them later under **Runs** in the menu.

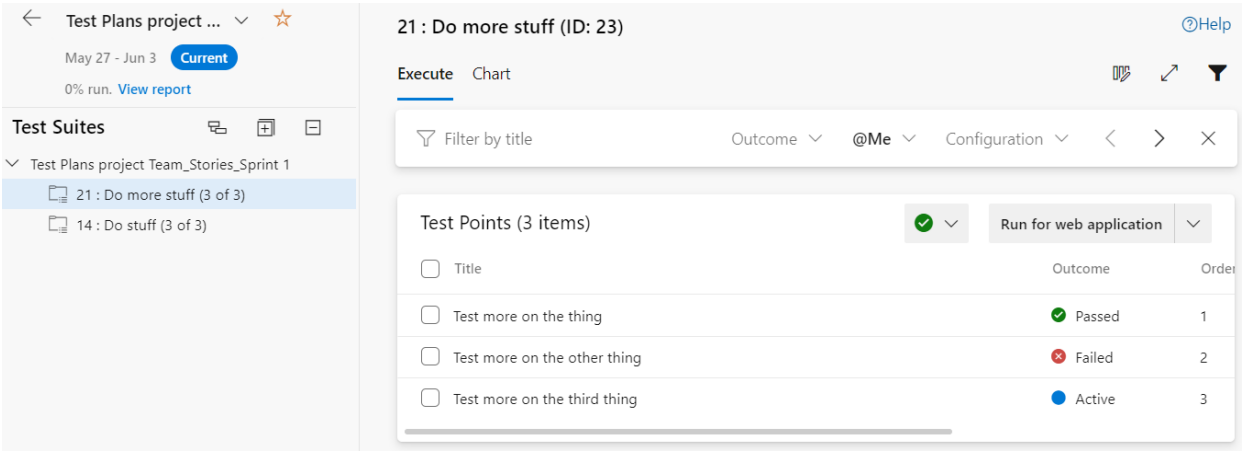

Figure 51: Test suite with tests

This is basically what test plans are about: organizing your tests in a central location and directly giving feedback to developers through Azure DevOps Boards.

Charts and reporting

Another cool feature is charts. Next to the **Execute** tab, you have a **Chart** tab. On the individual work item level, this is not truly relevant for anyone else but the tester. However, on the test suite level, you can get some cool insight for the team. You can create test case charts and test result charts. The test case charts are not going to work in our little project because we do not have sufficient users and work items. You could use these charts to see who created the most tests or has the most tests assigned. You can create pie, column, bar, line, and other types of charts. For example, I set the state of two tests to **Ready** and one to **Done**. Configuring a column chart on state would look as follows.

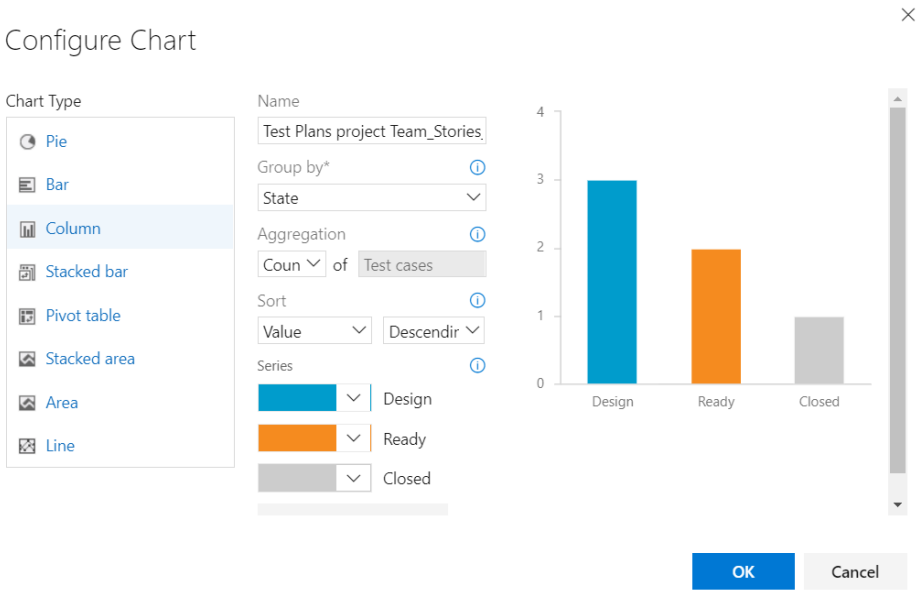

Figure 52: Configuring a column chart on state

While the number of test cases can be interesting at times, you probably want to know the results of your tests. Click **+ New** and choose **New test result chart**. In the window that opens, select **Outcome** in the **Group by** field. Click **OK** to save the chart. You now get a pie chart that gives you an overview of the outcome of your tests. When you hover over the chart, you get a three-dot button (**…**) in the top-right corner of your chart. You can edit or delete the chart or add it to your dashboard so the whole team can track tests.

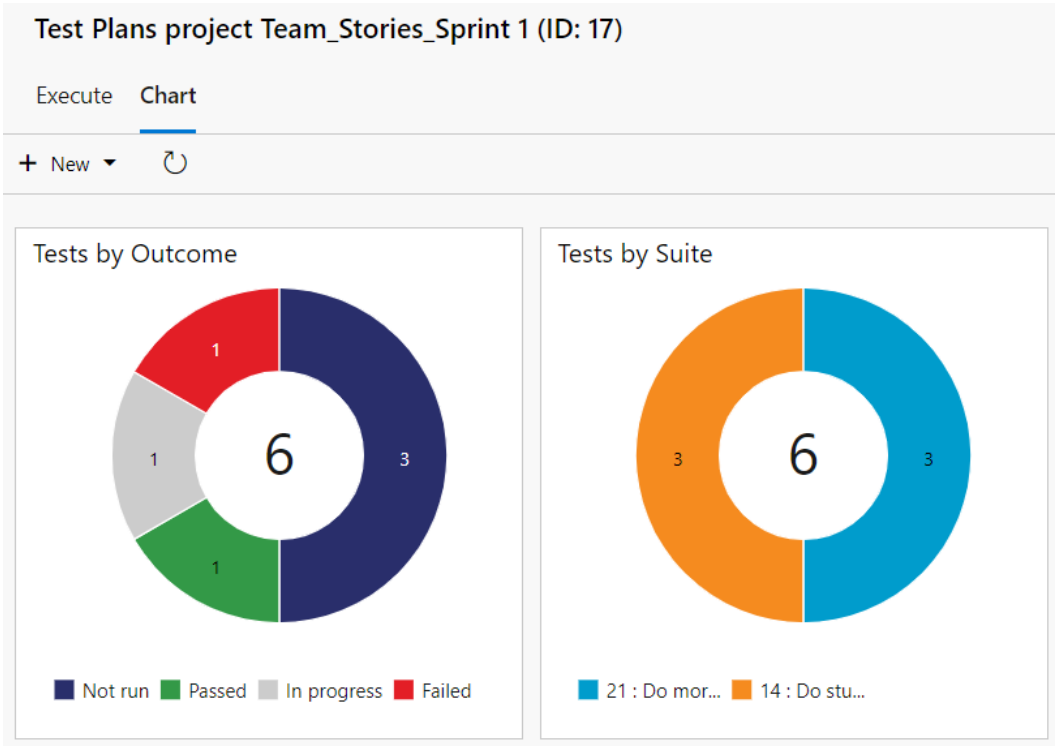

Figure 53: Test charts by outcome and suite

Charts are great, but for a quick summary of your test progress, you can simply go to the **Progress report** from the menu. This will show you a summary of the number of open tests, passed tests, failed tests, and the percentage of tests that have been run.

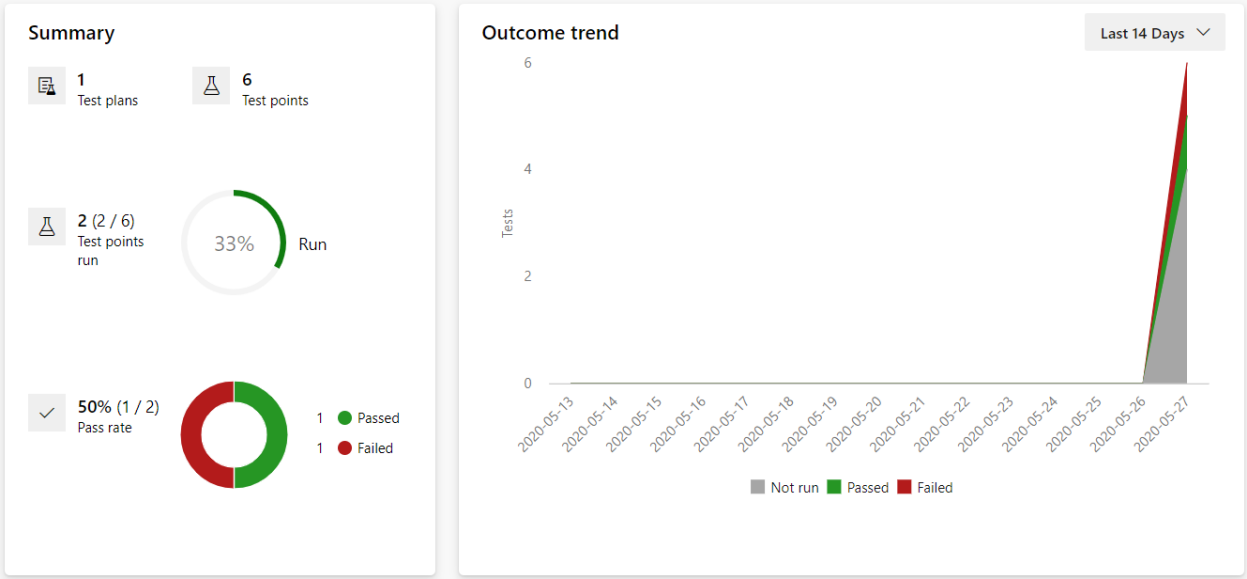

Figure 54: Progress report

With the progress report and your charts, you should have all the insight you need.

Creating a test plan on a trial or paid account

When you are on a trial or paid plan, you can create your test cases directly from your test plans instead of relying on a board task. Go to your organization with tests enabled, and go to a project where you have some test tasks. Now go to **Test Plans**, and you should see a difference already. Where the free plan has the **Execute** and **Chart** tabs, the paid plan also has a **Define** tab where you can create new test tasks. In case you do not have a plan yet, or you go back to the Test Plans overview, you will see that you have a new button here as well, **+ New Test Plan**. Click the button and enter a name, for example **Succinctly Test Plan**.

In the test plan you can create a new test case. This is the same as adding a task on the board, so that has been covered already. You can also add an existing test case, which will open a pop-up window where you can define a query to find existing test cases. You can then select these tests and add them to your current test plan.

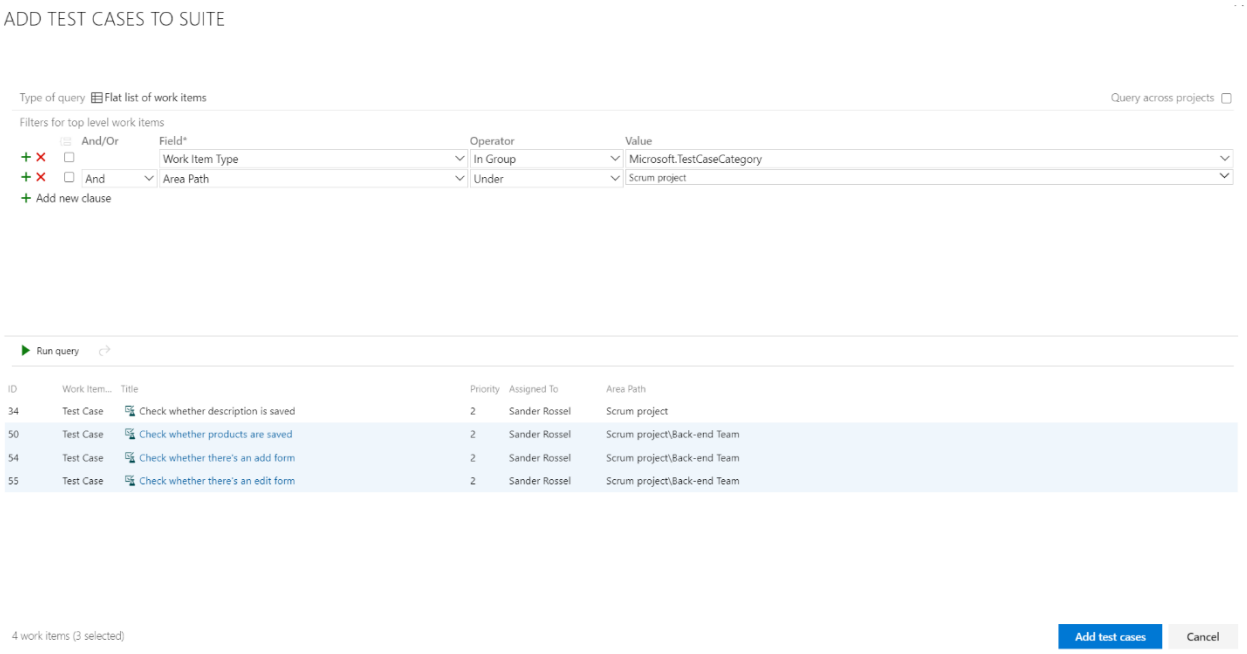

Figure 55: Add test cases to suite

Another way to add test cases to your test plan is by using the grid. Using the grid view, you can add multiple tests in an Excel-like interface. The tests and steps you enter are made into tasks when you save.

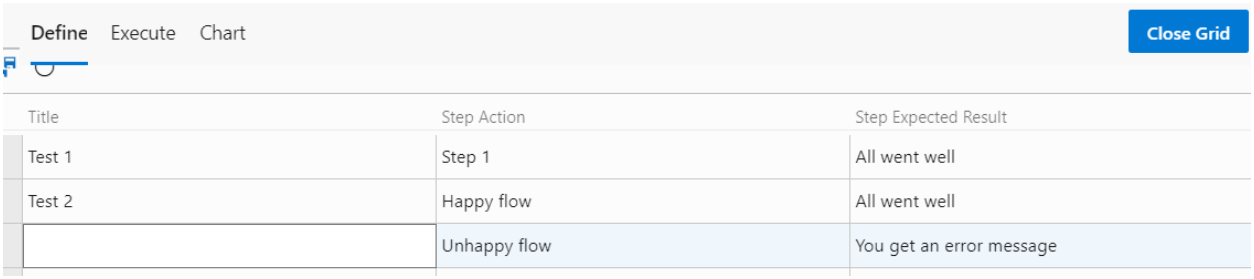

Figure 56: Add test cases using grid

In this example, Test 1 is created as a separate task with a single step, Step 1. Test 2 is created as a task with two steps, Happy flow and Unhappy flow.

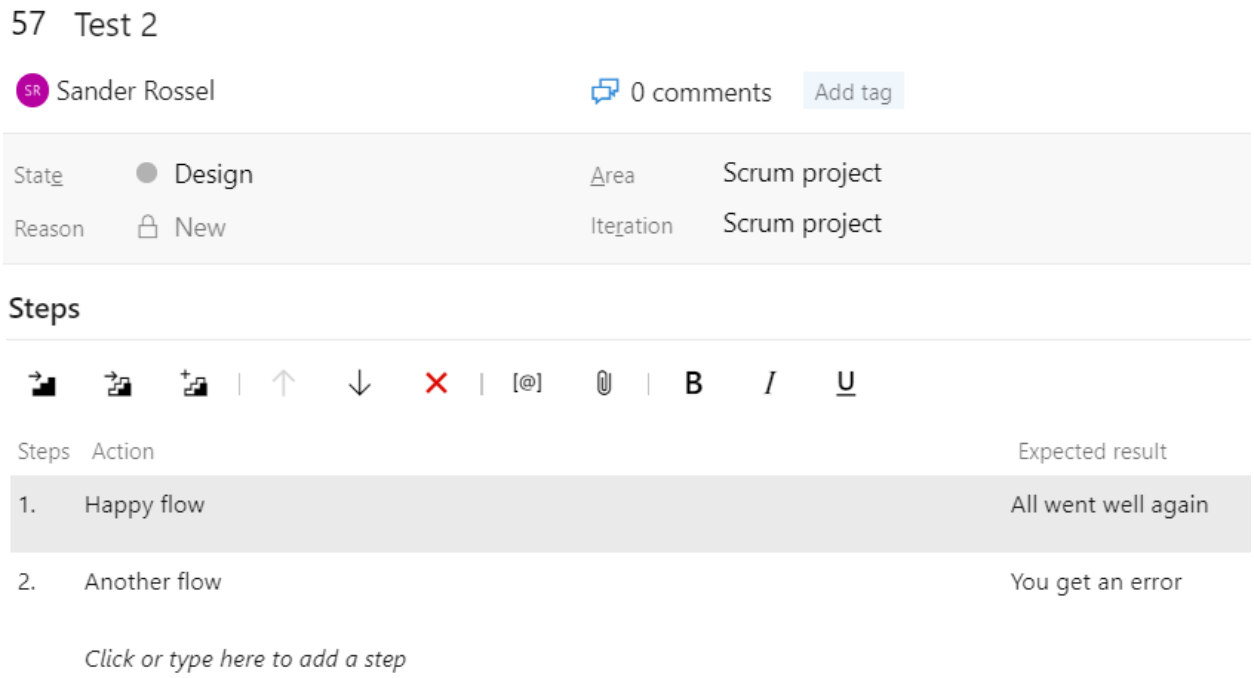

Figure 57: Test task with steps

Whatever method of test creation you choose, creating your test plans and test cases using the paid tools is somewhat easier than in the free version. It gives you more control over what you create and where.

Parameters

Another useful addition to the paid tools is parameters. Using parameters, you can specify a set of values you want to use in your tests. For example, you could have a test where someone needs to order some items. It is important that the number of items is at least one. You also want to check if the name works correctly.

When you go to **Parameters**, you can create a new parameter set. Click the **+** button at the top left to create a new set. You now get another grid, where you can edit column headers and fields. The column header is the name of your variable. The cells below it are possible values. So, enter **quantity** and **name** (by double-clicking on a column header) for column names and **0**, **1**, **256**, **-1**, an **empty space**, **Sander**, and **null** for values.

It is not immediately obvious, but you can remove columns by hovering over them and clicking the cross that shows up. You can add columns with the **+** button in the corner. The **Save** button icon shows two floppy disks instead of one (and I really do not know why).

In the **Properties** tab, you will find a work item-like interface. That is because your parameter set really is a work item. If you go to your work items, you will find it there, but it does not show up on your boards. It has a name, state, iteration, linked work items, and everything. This is also where you can delete the parameter set, in the menu at the top right.

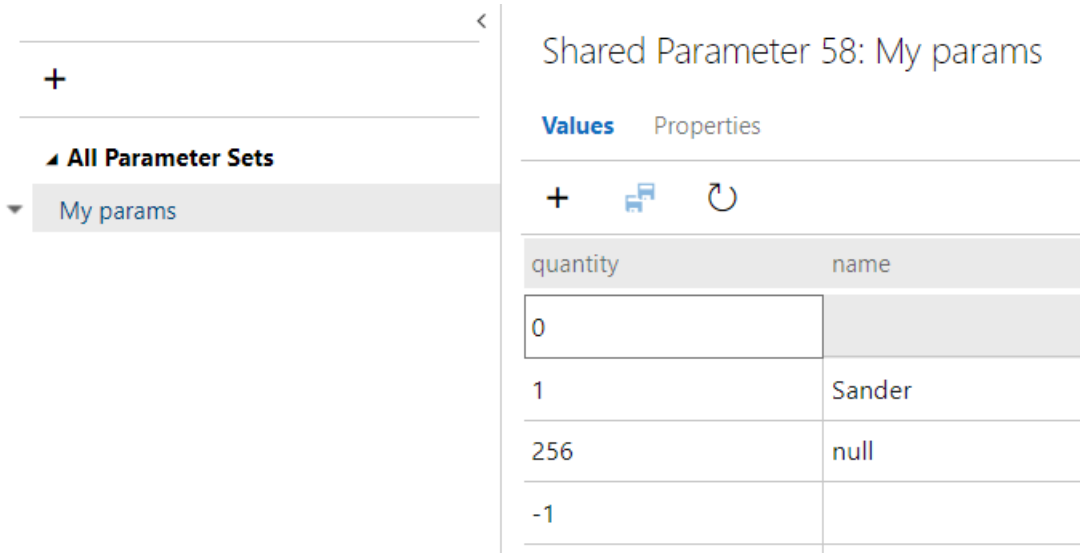

Figure 58: Parameter set

Now, go back to your test task and change the action of a step to **@name orders @quantity items**. This will automatically reference your parameter set to the task and show the values. These parameters are handy to keep around so you know what values you tested with, but they can also be used in automated tests.

Configurations

Another addition to paid testing plans is configuration. Here, you can specify a set of hardware and system variables. One configuration and two configuration variables are predefined. The variables are your browser and operating system, and the configuration is Windows 10. You can edit the predefined variables and configuration. For example, you can add **MacOS** and **Linux** as values to the **Operating System** variable and remove **Windows 8**.

Click **+** and then **New configuration variable**. Enter the name **Graphics card** and the values **AMD Radeon** and **Nvidia GeForce**, and then click **Save**. Click the **+** button again and click **New test configuration**. Enter the name **Nvidia GeForce** and add the configuration variable **Graphics card** that we just created. Assign the value **Nvidia GeForce**. Create another test configuration and do the same for the AMD Radeon.

Go back to your test plan and click the three-dot button that shows up when you hover over your test suite, and then click **Assign configurations**. You can now clear **Windows 10** and select **AMD Radeon** and **Nvidia GeForce**. When you save your new configurations and go to the **Execute** tab, you will see that your test cases doubled, because all tests now have to be tested on both configurations. It would make sense to assign configurations for all major browsers or all major operating systems if you are developing a cross-platform application.

Summary

With Azure DevOps Test Plans, you can track your tests, generate reports and charts, and share them all with your team. When the entire team has insight into the test results, testing becomes a team effort. Managing tests with variables and configurations can make your work as a tester a lot easier. The easy integration with your boards gives direct feedback to developers, so failed tests can be fixed as soon as possible.

Chapter 8 Artifacts

We have now seen most of what Azure DevOps has to offer. One feature that is often overlooked is artifacts. You can create your own NuGet, npm, Maven, and Python feeds, and push any kind of package to DevOps. This allows you to share and reuse your packages from a centralized (private) repository.

The default feed

When you browse to **Artifacts** in DevOps, you will see your default feed that was created for you. This feed will be accessible for you to publish and restore packages from. You will see a **Connect to feed** button that you can click for information on how to publish and restore packages for various package managers. We are not going to use this feed though. Your default feed is scoped to your organization, while new feeds will always be scoped to your project. Organization-scoped feeds are always private, while project-scoped feeds are public if the project is public. The URL for connecting to your feed will also be slightly different because your project-scoped feeds do not have the organization name in the URL. An organization-scoped feed is also visible in all your projects, while a project-scoped feed is not.

If you want to remove the default feed, you can. In the top-left you will see the name of your current feed, and using the cog in the top-right, you can edit and delete your feed. When you delete a feed, it will be moved to the bin, where you have to delete it again for permanent deletion. If you change your mind, you can also restore your feed from the bin.

Creating a feed

To create a new feed, click **+ Create Feed**. Enter a name for your feed, like **succinctlyfeed**. Next, you need to think about the visibility of your feed. You can make it visible to everyone in your Azure Active Directory, to members of your organization, or only to specific people. For this example, choose members of your organization.

The next option is whether you want to enable upstream sources. These are packages from common public sources. So, if your package needs another package from another source, that package is downloaded to your feed and accessible from there. This allows you to use only one feed, and if the package is removed from the source feed, you will always have it backed up in your own feed. Of course, if a package is pulled from the feed, you should consider looking for an alternative to receive updates, if this is important for that package. The downside is that it takes up space, and you can only store up to 2 GB for free. Above that, you will be billed, or, if billing is not set up, you will not be able to upload any more packages. For now, disable upstream sources because it creates some noise, and we do not want that.

When you create the feed, be sure to select it in the upper-left corner.

Connecting to your feed

Before we can do anything, we will need to connect to our feed. We have various ways to do this.

Visual Studio

The easiest and most straightforward way is to connect using Visual Studio. Open any project in Visual Studio and from the top menu, go to **Tools** > **NuGet Package Manager** > **Package Manager Settings**. This will take you to the general NuGet options. Select **Package Sources**. Click the **+** button to add a new source, and name the source **succinctlyfeed**. Now, go to Azure DevOps, select **Connect to feed**, and then select **Visual Studio**. This will show you the name and source. Copy the source and paste it into the **Source** field in Visual Studio, and then click **Update**. Go to **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution**. You can select your new feed in the upper-right corner. Click the **Browse** tab, and you will be prompted for your Azure DevOps credentials. After you enter them, you will have access to your feed. Your account will be added to your Visual Studio accounts, which can be accessed in the upper-right corner.

.NET Core CLI

Another method of adding a NuGet source to either Visual Studio or our solution or project, is to use the .NET Core CLI tool. Open a command prompt and use the **`dotnet nuget`** command.

Code Listing 9

```
dotnet nuget add source [your source] --name succinctlyfeed --username
[your email] --password [your password]
```

This will add a source to your global Nuget.Config file, which you can find at %APPDATA%\NuGet\NuGet.Config. You will see it has a **packageSourceCredentials** element.

You can also add the feed at the solution level. To do this, create a new file in your solution folder and name it **NuGet.Config**. It needs at least a root element, so edit the file and add the following.

Code Listing 10

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
</configuration>
```

Now use the same command as before, but with an extra **`--configfile`** option.

Code Listing 11

```
dotnet nuget add source [your source] --name succinctlyfeed --username
[your email] --password [your password] --configfile path\to\NuGet.Config
```

You can find more information on the nuget.config file here.

Personal access token

We can also use a PAT instead of our username and password. Go to DevOps and create a new access token. Give it any of the authorizations for the Packaging scope. You can now use the token to add your source. You can use any username.

Code Listing 12

```
dotnet nuget add source [your source] --name succinctlyfeed --username
[whatever you like] --password [your token]
```

This also works on the solution level.

Now that we can connect to our source, let us see how we can publish packages.

Publishing packages

In this section we are going to create a NuGet package and publish it to our feed. First, we have to create a package. Open Visual Studio and create a new **Class Library (.NET Standard)** project. Name it **SuccinctlyPackage** and enter a property in the default **Class1** (for example, a **Name** property). Also make sure the class is public. This will be our package.

Code Listing 13

```
namespace SuccinctlyPackage
{
    public class Class1
    {
        public string Name { get; set; }
    }
}
```

Next, open the project file and add a **GeneratePackageOnBuild** element to the **PropertyGroup**. The entire .csproj file should look as follows.

Code Listing 14

```
<Project Sdk="Microsoft.NET.Sdk">
```

```
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
    <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
  </PropertyGroup>

</Project>
```

Now, when you build the project, a .nupkg file is created in your bin\Debug folder (or bin\Release when you make a release build).

The next step is to publish it to our feed. This should be as simple as doing a **`dotnet nuget push`**, but unfortunately, it is a little more difficult. The CLI tool will add the credentials to the push request, but you will get an error stating "The request to the server did not include the header X-NuGet-ApiKey, but it is required even though credentials were provided." We can now add a PAT to our CLI command, or we can use the Azure Artifacts Credential Provider. The PAT is easy enough if you already have one.

Code Listing 15

```
dotnet nuget push path\to\SuccinctlyPackage.1.0.0.nupkg --source
succinctlyfeed –-api-key pat [your token]
```

This should work, but entering your token every time you do this is tiresome at best, so let us look at an alternative.

Azure Artifacts Credential Provider

As an alternative to the PAT, we can use the Azure Artifacts Credential Provider. You can go back to **Connect to feed** in DevOps and select either **dotnet** or **NuGet.exe**. This will show the message "First time using Azure Artifacts with NuGet.exe on this machine? Get the tools." Click the button and you will see a blade with "Download and install the credential provider." That link will take you here. Personally, I find the manual installation the easiest. Download the latest version of the .NET Core Credential Provider and copy the **plugins** folder to **%UserProfile%\.nuget**.

If you now try to push again, use the **`az`** value for your **`--api-key`** option and specify the **`--interactive`** option.

Code Listing 16

```
dotnet nuget push path\to\SuccinctlyPackage.1.0.0.nupkg --source
succinctlyfeed –-api-key az --interactive
```

You will now have to copy a link from your command window, paste it in a browser, and then copy a code from your command window and paste that in the browser as well. You then have to log in to your account, and the command prompt will continue. You can then close the browser window. After you have used the Credential Manager once, it will remember your credentials, and you do not have to do this again.

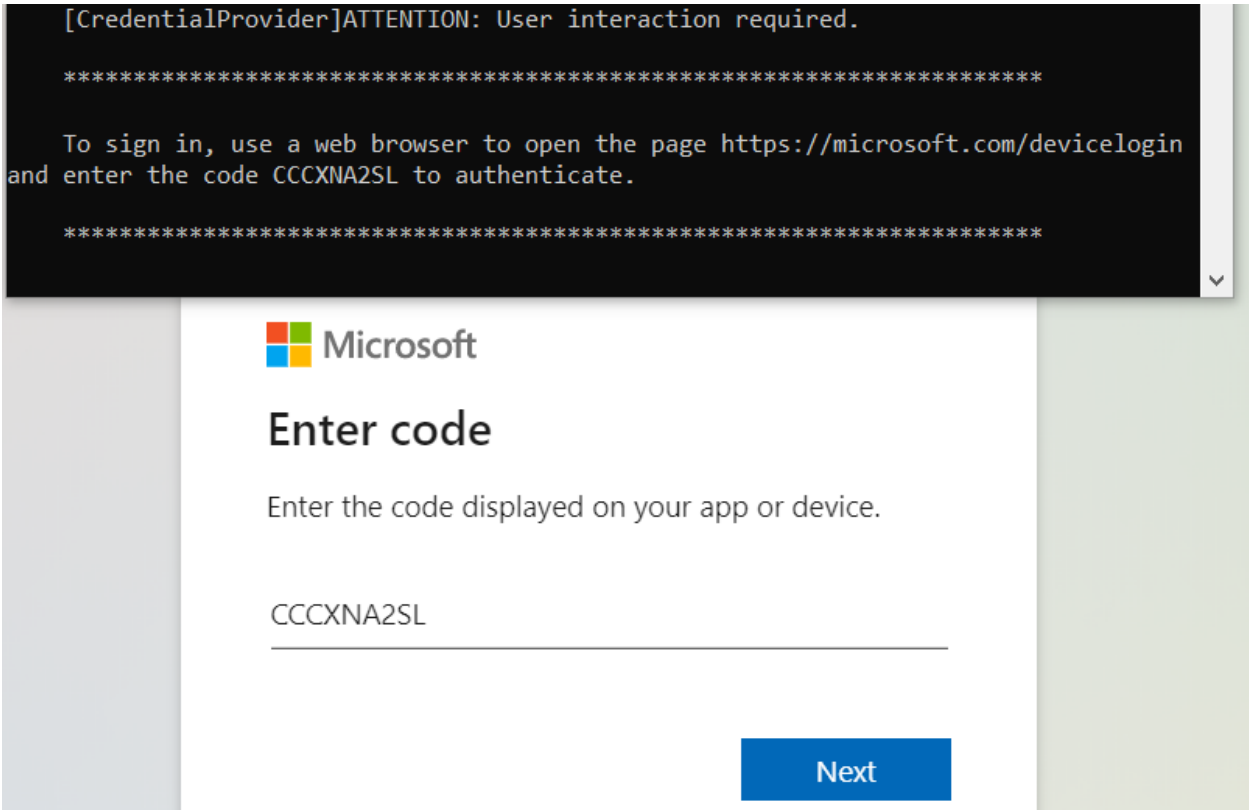

Figure 59: Azure Artifacts Credential Provider

Whether you have used the PAT or the Credential Manager, your package should now be successfully published to your feed in DevOps.

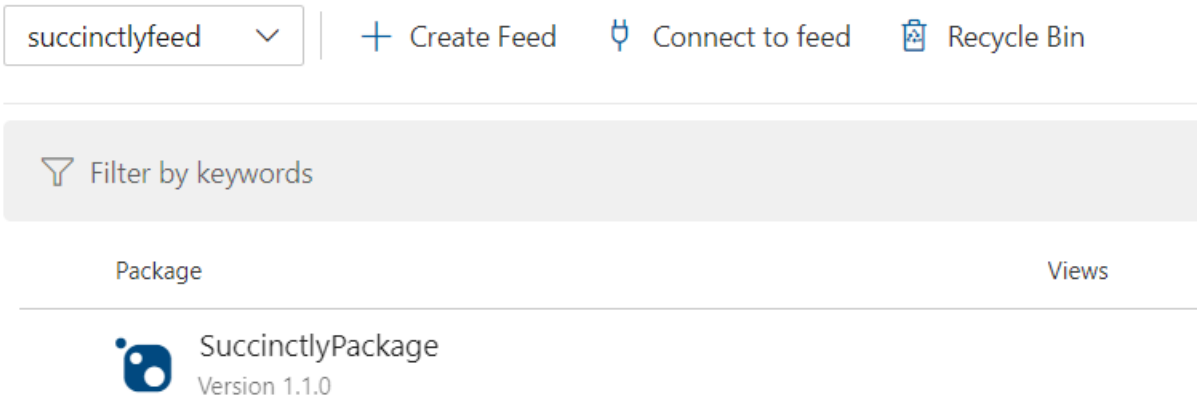

Figure 60: Published package

You cannot publish identical packages twice, so when you need to publish again for whatever reason, you will need to create a new version of your package. If you go to your package project in Visual Studio, right-click the project and click **Properties**, and then go to the **Package** tab, you will find the package version, as well as other properties you can set for your package.

You can now view all sorts of information about your package in DevOps, like authors, versions, how often it was downloaded, how many users your package has, and more. You can also delete a version, but beware that this will not let you republish the same version again.

Publishing using pipelines

Publishing packages manually requires quite a bit of work, so automating this task is not a bad idea. I have created a repo and pushed my code there. Next, I simply created a pipeline and copied our previous work, except this time I do a NuGet push instead of publish and publish artifact. Just be sure to change the path of the packages to **`push`**, because your .nupkg file is not in the artifact staging directory. Instead of building, you can also use the **`dotnet pack`** command, which gives you some extra options and will place your .nupkg file in the artifact staging directory. It is not strictly necessary though. The classic designer is intuitive enough, and the YAML for the **`nuget push`** command looks as follows.

Code Listing 17

```
- task: DotNetCoreCLI@2
  inputs:
    command: 'push'
    packagesToPush: '**/*.nupkg'
    nuGetFeedType: 'internal'
    publishVstsFeed: '[feed id]'
```

The term **`publishVstsFeed`** is a bit of legacy from the time when Azure DevOps was still named Visual Studio Team Services (VSTS). If you build this, you will get an error that the package already exists. You should update the package version and the build will trigger again, but this time it will succeed, and you will get a new package.

If you want to be able to rebuild the pipeline without it failing, for example because you also have a regular project in your repository, you cannot use the .NET Core task. I am not sure, but it may be a bug that the **`nuget push`** command does not allow for additional arguments (it is the only command of the .NET Core task that does not allow arguments). Instead, you can use the NuGet task, which is almost the same, except it has an extra property, **Allow duplicates to be skipped**.

Code Listing 18

```
- task: NuGetCommand@2
  inputs:
    command: 'push'
    packagesToPush: '**/*.nupkg'
    nuGetFeedType: 'internal'
    publishVstsFeed: '[feed id]'
    allowPackageConflicts: true
```

For some reason, this does not add the option **`-SkipDuplicate`**, but simply ignores the response 409 (Duplicate).

Restoring using pipelines

When you add a reference to your package in your web app by adding the feed to Visual Studio, you will get a build error saying “error NU1101: Unable to find package SuccinctlyPackage. No packages exist with this id in source(s): NuGetOrg.” That is a clear enough error message, so let us fix it.

We have two options: use a NuGet.Config on the solution level instead so any pipelines know where to look, or add an additional feed. Since we explicitly did not use the NuGet.Config, let us look at the additional feed.

Code Listing 19

```
- task: DotNetCoreCLI@2
  displayName: Restore
  inputs:
    command: 'restore'
    projects: '$(restoreBuildProjects)'
    feedsToUse: 'select'
    vstsFeed: '[feed id]'
```

That should fix your restore step, and it will now build correctly again.

Summary

Using artifacts, you can host your private packages, whether they are NuGet, npm, or Maven packages. Other packages are supported as well. By using pipelines, you can publish your packages automatically so everyone in the team can easily deploy fixes or changes to existing packages.

Conclusion

In this book, we have seen almost everything Azure DevOps has to offer. Starting with accounts, organizations, projects, and settings, we can create a DevOps environment and get our team up and running. Boards allows us to plan work with the team. Working with sprints and story points should give us insight into how much the team can work on in each sprint. This allows you to better estimate upcoming work. Using repos and pipelines enables teams to work together on code, and to build and publish that code automatically. If anything fails, the whole team will be notified, and the error can be fixed when it happens. By using Git branching strategies, you can always deploy to production. Testers can use Azure Test Plans to plan and manage tests. The outcomes can be shared with the team so that developers and testers can work together more closely. Finally, with artifacts, teams can share packages in any language and use them in their projects.

Next to the Azure DevOps environment, we have also seen some Azure resources and how they work together with Azure DevOps, most notably Azure Active Directory (AAD) and Azure Key Vault.

Discussing everything Azure DevOps has to offer is an impossible task. Pipelines alone have hundreds of tasks that support multiple languages, such as C#, JavaScript, Python, and Go. Next to Azure, you can release your software to AWS, Google Cloud, or your on-premises servers using deployment groups.

This book should have given you the tools and insight to go beyond what this book covered. With Azure DevOps, you have the tools to collaborate on high-quality software!