OPTIMUM SETUP

Getting the most out of Octopus Deploy to optimize your team's continous delivery pipeline.

Octopus Deploy

CONTENTS

Authors:
Ryan Rousseau, Bob Walker & Derek Campbell.
Published May 2019, Brisbane Australia.

> We want to help you make better decisions about your configuration so that Octopus Deploy can scale with your company.

Introduction

Everyone who works at Octopus Deploy likes to think our product is super easy to use and is very easy to set up. Underneath the covers, Octopus Deploy is a complex product. It was designed to help reinforce best deployment practices such as building artifacts once, using the same deployment process for each environment, or restricting access to production machines to name a few. At the same time, Octopus Deploy offers you flexibility in how you meet your company's needs. That flexibility can be a double-edged sword.

The authors of this book focus on customer success at Octopus Deploy. We meet with customers to ensure that they are getting to most out of Octopus Deploy. We discuss what is and isn't working for them and offer recommendations. We have seen some interesting configurations with Octopus Deploy. In some cases, bad practices have been going on for so long that changing them takes quite a bit of effort. That is what drove us to write this book. We wanted to take all those lessons we have learned and share them with the community.

Our goal for this book is that you can set up Octopus Deploy to scale as soon as possible. Anytime you start learning something new, you go through a series of steps. First, you start out by learning the basics. You create a proof of concept to see if it works for your basic needs. After that, you move onto a pilot project or a pilot group and use it day to day. You remove short-cuts

made early on. Now your proof of concept is becoming the working solution. After that comes adoption, where everyone in the company starts using it. This is where a wrong configuration or two becomes a standard. Changing standards takes more time as usage grows. It goes from being a one week project to taking months or even a year.

We are always improving the onboarding process. Our documentation has a great getting started guide. But those items give you "just the facts." Imagine if the getting started guide prompted you every time you did something that is "not recommended." That would get old very quickly. The intended audience for this book have set up a proof of concept and would like some recommendations on configuration. We want to help you make better decisions about your configuration so that Octopus Deploy can scale with your company. Our hope is that anyone with Octopus Deploy experience, from beginners to experts, will gain something from this book.

We are going to take a fresh install of Octopus Deploy and configure it to scale in this book. We are going to start at the very beginning, setting up your infrastructure, and proceed on with configuring environments, lifecycles, retention policies, projects, security, performance, and maintainability. We wrote it to be read from cover to cover, but you can pick your favorite topic out and jump right to it. We don't believe that what we recommend will solve every possible scenario you run into in your day to day life. We want you to use this as a starting guide to meet your needs.

Finally, this book does have a lot of overlap with our getting started guide and our on-boarding process. We are going to assume you have some familiarity with Octopus Deploy. You know what a tentacle is and you know how to add steps into your deployment process. If none of those terms are familiar to you, or if this is your first time using Octopus Deploy, we recommend that you read our getting started guide and create a proof of concept using our on-boarding process.

"We are going to start at the very beginning, setting up your infrastructure, and proceed on with configuring environments, lifecycles, retention policies, projects, security, performance, and maintainability.

2

TERMINOLOGY

Terms

Let's start by defining some terms and what they mean in the Octopus world. These terms might match up with terminology that you use. They might differ slightly. But it'll help to have a shared glossary as we work through the upcoming chapters.

Target / Deployment Target / Machine

These are the machines and services that you deploy your software to. They might be physical machines, virtual machines, or PaaS targets in the cloud.

In early versions of Octopus, we used the name Machines instead of Deployment Targets. After we added Targets like Azure Web Applications and Kubernetes, Machine became a misnomer. You might be deploying to a machine but you might also be deploying to an API. Deployment Target describes both usages.

Environment

An Environment is a group of deployment targets. Your Development environment might be one server that hosts a web application, a service, and a database server. Your Test environment may include a web application server, a service server, and a database server. Your Production environment might have multiple web application servers, service servers, and databases. With Octopus Deploy, you have the freedom to model your environments to match your company's infrastructure.

Package

A Package is an archive (zip, tar, NuGet) that contains your application assets. Octopus works by deploying your Packages to Deployment Targets.

You can host Packages in external repositories or the built-in repository. Octopus can integrate with external repositories like Artifactory and NuGet.

Project

Projects define your deployment process, configuration variables, and the deployment lifecycle. Think of it as a blueprint for releasing your Packages to Deployment Targets.

Release

A Release is a snapshot of your deployment process, configuration variables, and software packages. Releases are created from Projects. Releases are deployed to the Environments defined in the Project Lifecycle.

Deployment

A Deployment is the application of a Release to an Environment. Octopus executes the deployment process steps to transfer, configure, and install your packages.

The same release can be (and should be) deployed to multiple Environments. You should deploy the same binaries to each of your environments using the same process.

Lifecycle

A Lifecycle defines which environments that you can deploy a release to and in what order.

3

Configuring your

ENVIRONMENTS

Environments

Environments are the backbone of your deployment pipeline. It is what you move your code through. Before you configure anything else, you need to configure your environments first.

The most common setup is four environments. Those are Dev, Test or QA, Staging or Pre-Production, and Production. We've seen these at previous jobs and customers that we talk to. It makes sense for this to be a common setup. Dev is for developers to experiment on. It is very much in flux, and you expect it to go up and down quite often. Quality assurance test functionality in the Test. You use Staging as a final "sanity check" before deploying to Production. Production is ... well, production. It is what your users connect to.
We didn't design Octopus Deploy to force people to use a set of predefined environments. Some companies only have three environments. Others have many more. Not everyone uses the same naming for their environments. One person's Test is another person's QA. It is important to us that our customers can define and name their environments.

In this section, we will walk through our recommendations for configuring your environments to better prepare you to scale up and out your Octopus Deploy instance as you add more projects. We will also walk through a couple of common scenarios we have seen and how to work through them.

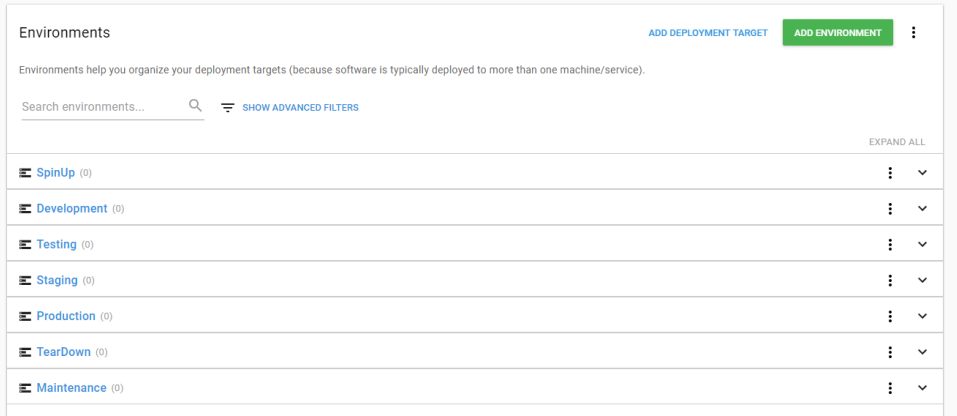
Environments Configuration

We recommend configuring your environments to match your company's terminology. Keep it general where possible. Think of how you want to phrase it during a conversation with a non-technical person. "I'm pushing some code up to Dev" or "I'm deploying my app to Production" makes a lot more sense than "I'm pushing to Dev Omaha 45." What does Omaha mean? The data center? Where did 45 come from?

> A good sign that you have well-modeled environments is that they are easy to explain. If it takes longer than a few seconds to explain your environments then that is a sign that you need to make some changes.

Keep the list of environments under a dozen or so. Have the standard four or five environments, such as Dev, Test, Staging, and Production. For dynamic infrastructure and maintenance, you can also add SpinUp, TearDown, and Maintenance. Those environments will help when it is time to build up infrastructure, tear down applications, or perform some scheduled maintenance tasks like taking a backup of logs in production.

Keeping the number of environments low helps with configuring lifecycles, channels, and security. They also keep your dashboard easy to follow.



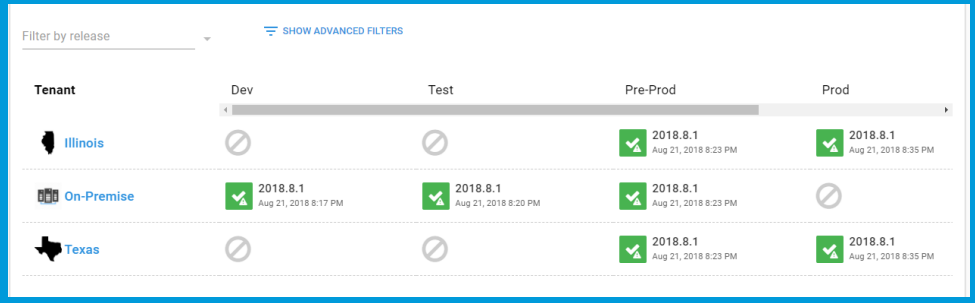
Don’t worry about the order of the environments or adding in machines yet. That will come a bit later. For now, we want to focus on creating our environments.

Multiple Data Center

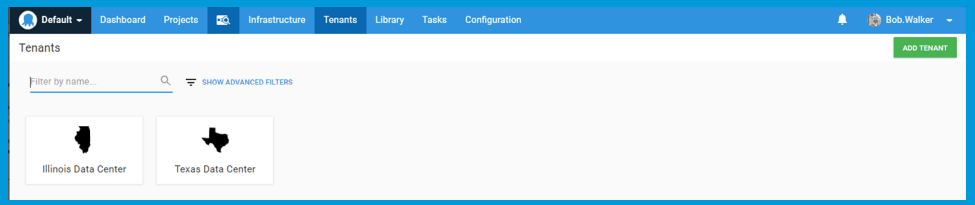
In the world of the Azure, AWS, Google Cloud, and other cloud providers, it is becoming common to deploy to multiple data centers. In some scenarios, you need to deploy the software in specific intervals. For example, you might deploy to a data center in Illinois first before deploying to one in Texas.

The temptation is to name your environment “Production [Data Center]” or “Production Omaha.” You would do this because you might not want to deploy to all data centers at the same time. Or you want to know what version of the code is in each data center. This does not scale very well, unfortunately. Every time you add a new data center, you will need to adjust many different things. You would have to add a new Environment. Then you would add that Environment to a Lifecycle. You would need to update variable scopes. Those are a few of the areas that you would have to manage.

A scenario that we have seen is customers deploy to an on-premise data center for dev, test, and staging, but production is hosted in data centers in Illinois and Texas. Before pushing to production, they run some sanity checks in a staging environment in Illinois and Texas. If you create an environment per data center, you would have seven environments when you only need four.



Because we don’t have any targets or projects set up at this particular time, this is rather easy to do. For now, we will add two new tenants to Octopus Deploy. We do this by clicking on the tenant link at the top of the screen and then clicking add tenant in the top right corner.



Don’t worry! We will be coming back to Tenants in a later chapter when we start setting up our first multiple data centers project. Just know that they are there when you need them.

> Adding images to your tenants makes them easier to find. You can do this by clicking on the tenant and selecting settings link on the left. On that screen, you can upload an image for a tenant.

Multiple Customers

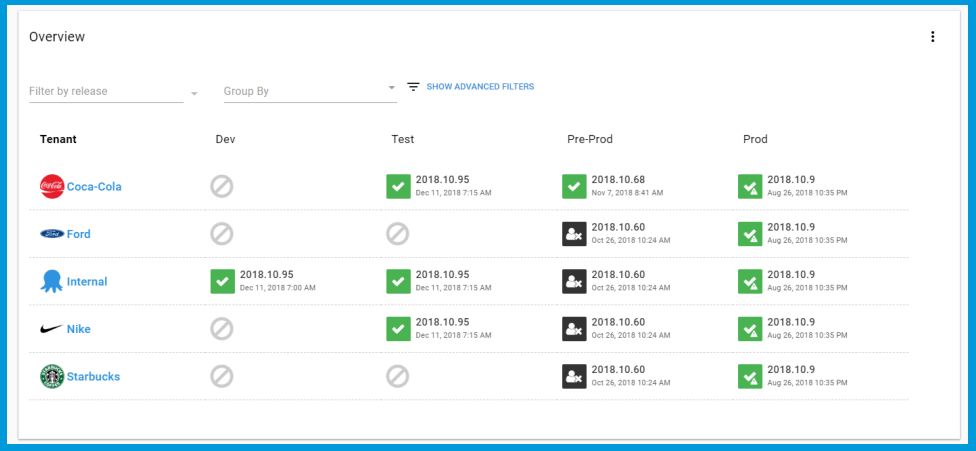
In the same vein of deploying the same project to multiple data centers, a lot of our customers deploy the same project to multiple clients.

Each of their customers gets their own set of machines and other resources. Again, you might be tempted to configure a unique set of

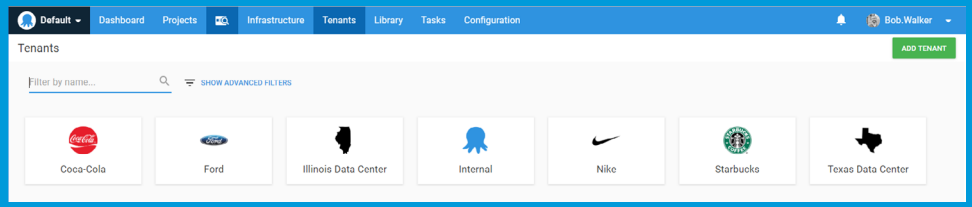
environments for each customer. You could create “Dev [Customer Name],” “Staging [Customer Name],” and “Production [Customer Name].” This will work for the first dozen or so customers but again it doesn’t scale very well.

Imagine if we had five clients, an internal testing customer, Coca-Cola, Ford, Nike, and Starbucks. The internal customer deploys to all the environments, dev, test, staging, and prod. Coca-Cola and Nike have resources in test, staging, and production while Ford and Starbucks only have resources in staging and production. If you create an environment per tenant, you would have 14 environments. And that is only for five customers!

This is where the multi-tenancy comes in. It allows you to keep the number of environments low while creating a unique workflow per client.



For now, we are going to create those five customers that we talked about in this example: Internal, Coca-Cola, Ford, Nike, and Starbucks.



In a later chapter, we will walk through configuring a suite of projects to deploy to multiple customers using the multi-tenancy feature.

Conclusion

In this chapter, we walked through of how to set up the backbone of Octopus Deploy, the environments. The major point to remember is to keep the number of environments small and leverage tenants to handle deployments to different data centers or customers. In upcoming chapters, we will be walking through setting up lifecycles and retention policies. Having a small number of environments will make that easier.

4

LIFECYCLES

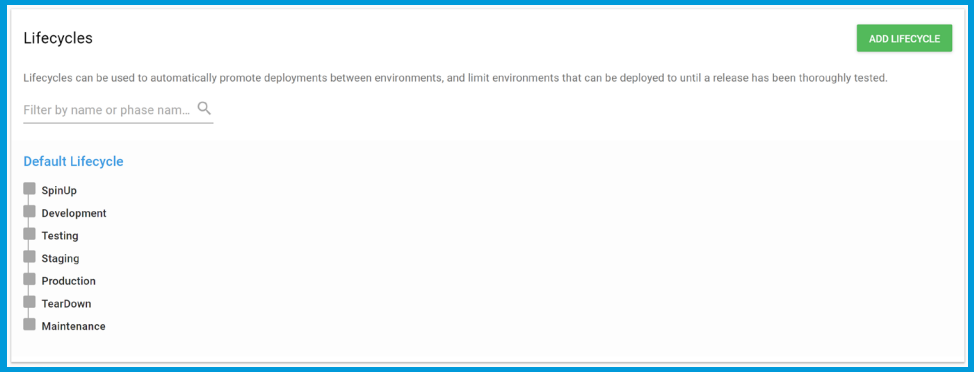
Lifecycles

Now that we have some environments, now it is time to set the order we can deploy our applications. Lifecycles let you define a promotion path for your applications. You can also use them to automate deployments and set retention policies.

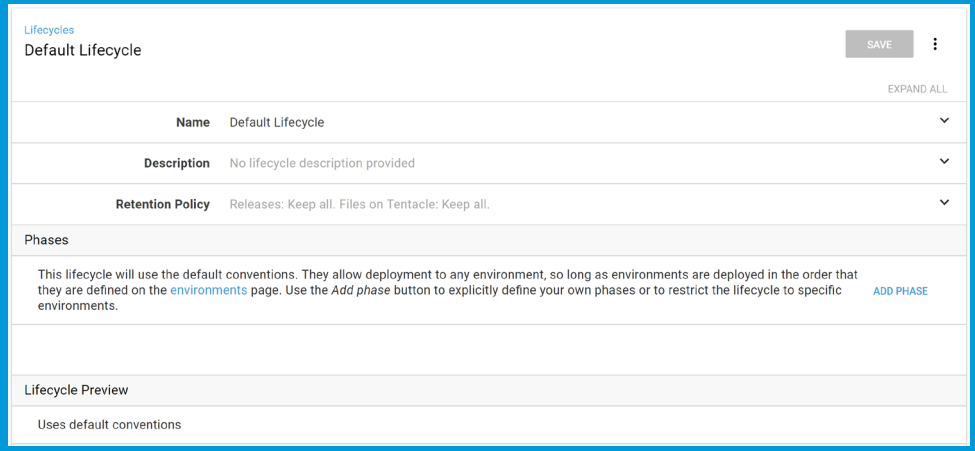
Early in your Octopus journey, you likely won't even know that lifecycles are there. The Default Lifecycle will handle most cases for small or simple configurations. Let's talk about the Default Lifecycle and why we recommend updating it early on.

The Default Lifecycle

Navigate to Library > Lifecycles. You'll see a single lifecycle named "Default Lifecycle." Let's take a look at ours.



We can see that the environments we created earlier listed in order. These are Phases created implicitly by the Default Lifecycle. By convention, the Default Lifecycle will create one phase per environment. Those phases will be in the order that the environments are listed in on the Environments page. If we view the Default Convention, we can see this information in the Phases section.



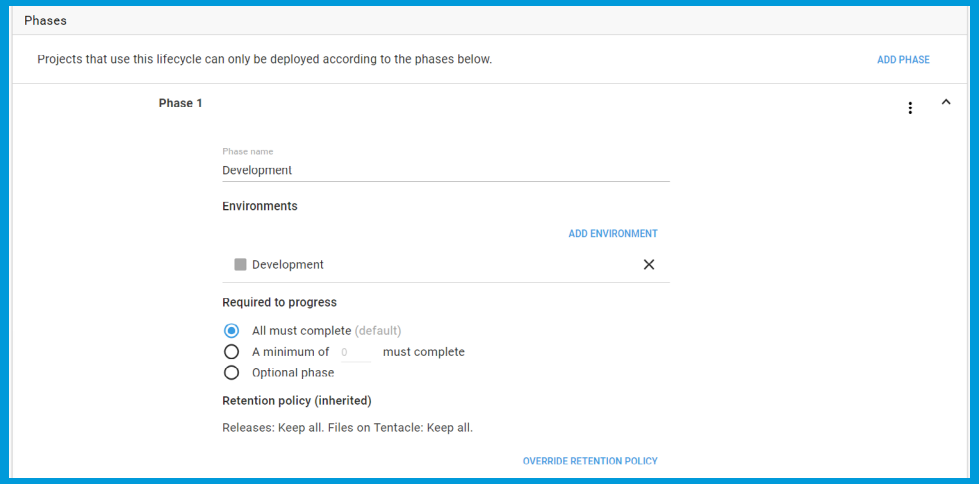
Phases

A phase represents a stage in your deployment lifecycle. You deploy to Phases in order and must have a successful deployment to move to the next phase. For example, there must be a successful deployment to the Development phase before we can proceed to the Testing phase. Phases can also include multiple environments, but we're going to focus on single environment phases in this chapter.

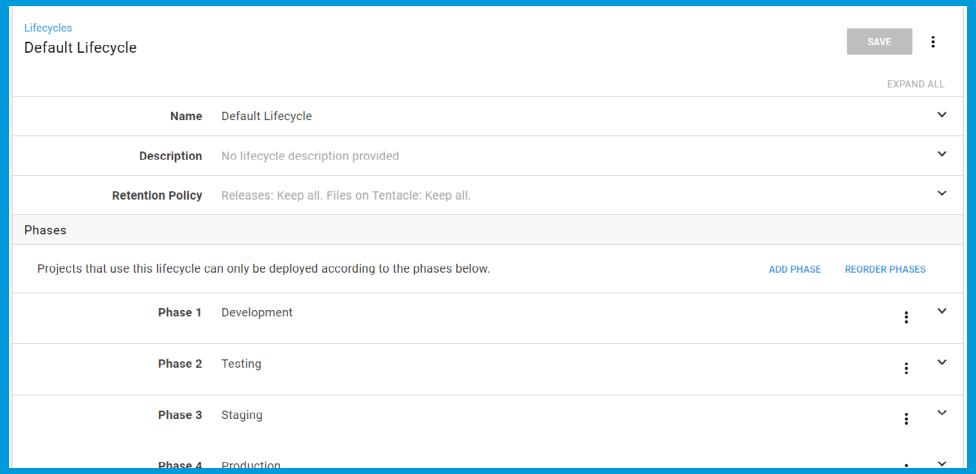
Let's create a Development phase for our lifecycle. We will give it the name "Development" and add the Development environment to the phase.

> Phase names usually match the environment it contains. While this is a good practice, it is not a rule.

We'll leave the radio button set to the default **`manually deploy`** on the add environment screen. We'll also leave the **`Required to progress`** and **`Retention policy`** set to the default values. We'll be covering these later.



Let's repeat the process to create phases for Testing, Staging, and Production.



Now we have an explicit default lifecycle for deploying our applications.

Other Lifecycles

Hotfix Lifecycle

What happens when there's a critical bug in production and you need to deploy a fix fast? Deploying to a Development and Testing environment might take too much time. But you still need to follow good deployment practices and validate the change before pushing to production. We often see hotfixes go directly to a Staging environment and then to Production after testing.

For this, we recommend creating a hotfix lifecycle. Our hotfix lifecycle will only have two phases, Staging and Production. Yours might be different to account for your internal policies and how your team decides to handle hotfixes. In a later chapter, we'll look at how to configure your project to use this lifecycle in addition to the default. F or now, we'll be content with setting it so that it's ready to go later.

Infrastructure as Code Lifecycle

We didn't include SpinUp, TearDown, or Maintenance in the default or hotfix lifecycles. We are going to create a lifecycle for our Infrastructure as Code projects that includes SpinUp and TearDown. We'll cover the use of this lifecycle later, but for now, create the lifecycle with a SpinUp and TearDown phase. There is a twist in with the SpinUp phase as we're going to make it an optional phase.

Optional Phases

One of the settings when creating a phase is the **`Required to progress`** setting. The default value is **`All must complete.`** All environments in the phase must be deployed to successfully before we can promote to the next phase.

The second choice is to set a minimum number of environments that must be deployed to successfully before promoting to the next phase. That is handy for cases where you have two or three QA environments, but only one is used

for each release of your application. You can configure your Testing phase such that it contains three environments, but only one needs to be successful before you can promote to the Staging environment.

The last choice is to set the phase to be entirely optional. The phase can be skipped, and you can deploy directly to the next phase after it.

Because errors and issues can arise when building up infrastructure, especially when setting up the IAC scripts for the first time, we want to be able to deploy directly to TearDown so that we can start fresh. As you're setting up these phases, make the SpinUp phase optional.

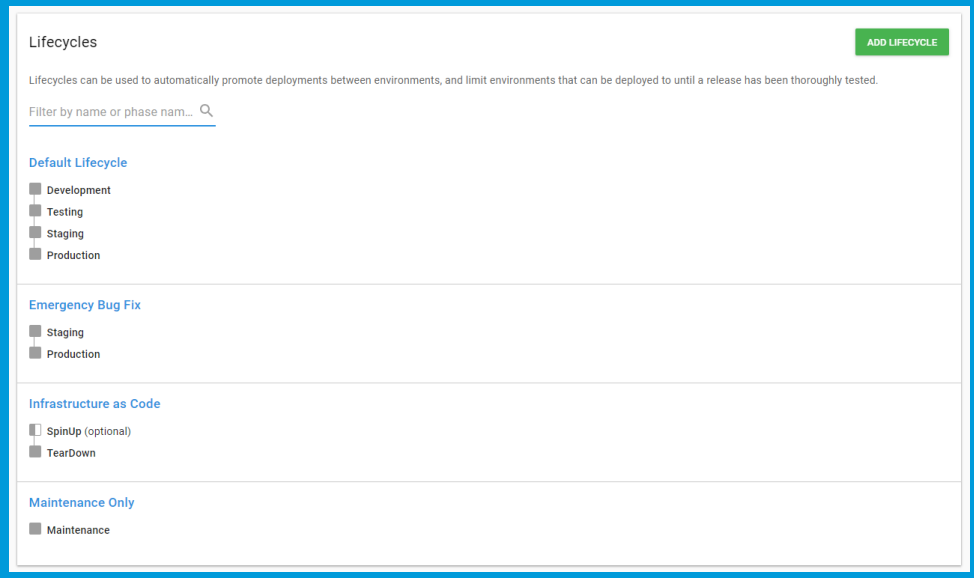
Maintenance Only Lifecycle

The last lifecycle we created was a Maintenance lifecycle. The Maintenance lifecycle and environment will be used for projects that run maintenance tasks such as backups or software upgrades. It can be used for any tasks that you want to run regularly and want the same benefits that Octopus provides for your application deployments.

Even though we have grouped the machines for these tasks in the Maintenance environment, you could also split them up into the Development, Testing, Staging, and Production environments if you want to run the tasks for machines in those environments at different times.

Whichever you choose, you know the steps now so go ahead and create that lifecycle with the phases that you want.

Conclusion



In this chapter, we configured our default lifecycle. We also created some auxiliary ones to set ourselves up for future success. In the next chapter, we'll walk through retention policies and how to keep your server from being cluttered with old releases and packages.

5

RETENTION POLICIES

Retention Policies

When we review a customer's Octopus Deploy configuration the first thing we ask to see is the retention policies. Setting appropriate retention policies is an easy win, and they are usually not changed from the default during a trial and not revisited after setting up a production configuration.

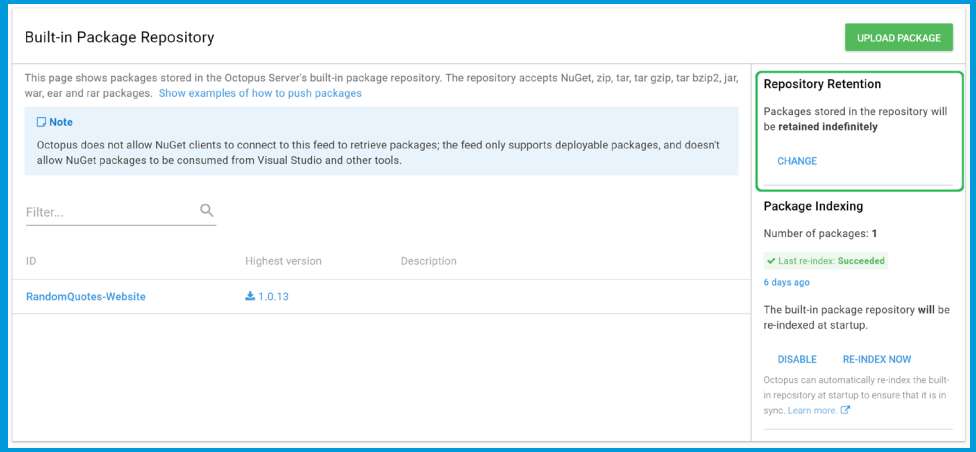
Without setting retention policies, Octopus will keep all releases created and packages uploaded indefinitely. If you have small packages or don't release frequently, you may never notice any adverse effects, but as your usage grows, you might run into disk space or performance issues as you server turns into a hoarder.

We haven't covered what packages and releases are yet, but it's okay to set these values now and revisit them once you have a better understanding of what they are.

Package Retention

First, we'll look at the built-in package repository retention policy. If you're not planning to use the built-in repository, then you don't necessarily need to set this, but it's good to go ahead and set it so that you won't have to remember it if you do start using the repository in the future.

You can see this policy by navigating to Library > Packages and looking for Repository Retention on the right side of the page.



Let's click CHANGE and set this to something less than forever.

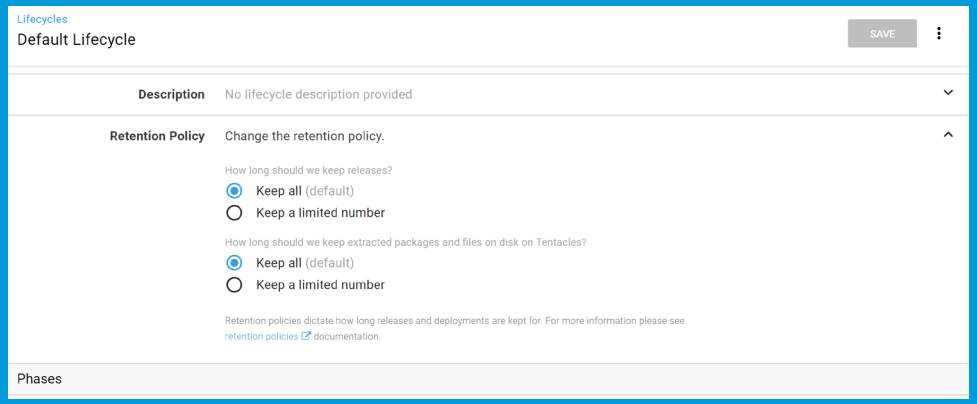
Choosing the `A limited time` option will allow you to select the number of days to keep a package in the repository. The default value is 30, but you can choose something shorter or longer based on your needs. Normally we opt for a shorter length of time, something close to 7 days.

It's important to note here that only packages that are not associated with releases will be cleaned up. That means that even if a package is older than the value you choose if it is attached to an existing release, it won't be cleaned up until that release is also cleaned up.

Which leads us to release retention policies!

Release Retention

Ok, we're going to take a quick trip back to the Lifecycles for the next bit. Navigate back to the Default Lifecycle and expand the Retention Policy section.



You can see that this is set to keep all releases forever and also to keep all extracted packages and files on disk on Tentacles, a Deployment Target that represents a Windows server.

Let's change this to something a little tidier. You have the choice to clean up after a specified number of releases or a specified number of days. If you're not sure what value to pick at this time, a good value to start with is to keep the last three releases for both the releases and the extracted packages.

There is one nuance to release cleanup. If the release is displayed on any dashboard, either the main dashboard or the project overview screen, it will not be cleaned up even if it matches the rules. You don't have to worry about a recent release in the Staging environment being deleted before it can be promoted to Production. So if you see that a release isn't being cleaned up, check the dashboards to see if it's being displayed.

Lifecycle Policy

We just set the retention policy for the lifecycle as a whole. Each phase in the lifecycle will use that retention policy. It also means that any releases that are created but not deployed to an environment will also follow this policy.

Why would you create a release and never deploy it? You probably wouldn't, but maybe your build server is creating releases as part of its process, or you have Automatic Release Creation configured on a project. Since it can happen, it's good to have this policy set to clean up any non-deployed releases.

Phase Policy

When a release moves into a phase, it will take on the retention policy for that phase. Any release deployed to only Development and no farther will use the Development phase's policy while releases that are deployed to Production will use the Production phase's policy. Right now, these are all set to the same policy, which is to inherit the lifecycle's policy.

Let's set the retention policy for the individual phases. We'll skip the Development phase as the lifecycle default of keeping three releases is perfect for our usage. You can keep the Testing phase as the default or something slightly longer like five releases if you'd like.

We prefer to keep the Staging and Production retention policies in sync. It is preferred for them to be time-based instead of keeping a set number of releases. The amount of time to keep them will vary depending on how frequently you release and if you have any regulatory obligations. A good starting point is 30 days, but you may choose 90 days, 365 days, or even choose to retain them forever if that is a requirement.

Changing Retention Policies on Existing Octopus Servers

If you have an existing Octopus Server with a large number of releases and are planning to set some retention policies going forward, we highly recommend starting with a large retention policy and adjusting it down to what you need.

For example, if you have 12 months worth of releases now, perhaps set the retention policy to keep 11 months worth of releases. The Octopus server will apply these retention policies periodically. After it has cleaned up the oldest releases, you can change the policy to keep ten months of releases. Rinse and repeat until you have reached your desired retention policy. You can apply this method with the number of releases instead of the time-based setting.

Conclusion

In this chapter, we covered retention policies for packages and lifecycles as well as covered the difference between the lifecycle policy and policies at the phase level. We set up some nice defaults that won't come back to bite you months or years down the road. Future you will thank you!

In the next chapter, we will jump into setting up a project and a deployment process!

6

Finally it's time to set up some...

PROJECTS AND DEPLOYMENT TARGETS

In previous chapters, we set up the necessary infrastructure scaffolding for our Octopus Deploy server to help it scale. In this chapter, we will take that infrastructure and use it to set up a simple project. We know...we know...this is something you already did in your Proof of Concept. You already know how to set up a project. The goal of this chapter isn't to rehash that. Instead, it is to provide you with a different way of thinking about projects.

Project Recommendations

Octopus Deploy was built with the core concept consistency across all environments. The process used to deploy to the Development servers is the same process which used to deploy to Production. You can disable and enable specific steps, but it is the same process. The end goal of this is to make going to production a non-event. Because by the time that project is deployed to production that process has been tested many times. In our case, it has been tested at least three times, in Development, Testing, and Staging.

Knowing the underlying concept of Octopus Deploy is consistency; here our are recommendations.

Keep your projects simple
One of our favorite programming maxims is the single responsibility principle. A class/function/method should do one thing and one thing well. The same holds true for projects. Projects should be straightforward and should deploy a component of an application.

For example, your project has a Windows Service, a UI, and a database. The temptation is there to create a single project to deploy all three components at the same time. The project wouldn't be very complicated. Four or five steps if you include a manual intervention. However, consider this, how often do you deploy a UI change which doesn't require a database change? What if you could make a small UI fix without having to worry about deploying your database and windows service? What if you could add an index into your database and push that to production without having to worry about deploying your UI or Windows service? Think about how much faster you could respond to customer feedback.

Each component should have its own project. Unique projects will give you the flexibility to deploy the pieces of your application only when a change has occurred.

Octopus Deploy provides a mechanism for a project to call other projects. That feature will allow you to set up an orchestrator, or what we like to call a traffic cop, to deploy your projects in a specific order.

Projects should be responsible for setting up what it needs to run
Imagine you are working on a greenfield application for six months. It only exists in your Development and Testing environments. Now it is time to deploy to staging. The web admins have set up a web server running IIS for you using a base image. The DBAs have created an account for the application to use. What about the configuration? What should the database name be?

> This will also allow you to isolate your code in your source control repository or have separate builds for each component. This way your CI/CD pipeline only has to build and deploy something which changed rather than building and deploying the actual application. An added benefit to that is reducing the build and deployment times.

When you set up a project's processes you should work under the assumption the key base applications are present (.NET Framework, IIS, SQL Server, WildFly server, Oracle Database, and others) but they have never been configured for your application. Assume SQL Server is there but the database

has never been created. Assume IIS is there but the web application has never been configured.

Going back to earlier, when it is time to deploy to staging you should only need to verify the servers are there and hit the deploy button. The project deployment process will take care of the rest. As a bonus, if a new server is added, you can deploy to that new server without having to worry about the configuration. All the servers will be configured the same.

Take advantage of the run conditions
Almost everyone is familiar with the environment run conditions. Run a step in production only. Alternatively, don't run this step in development or testing. However, there are other run conditions. Only running when the previous step was successful, only running on failure, always running, and only run when a variable is set to true.

Those additional conditions are advantageous. You can configure a step to send a slack notification when a failure occurs. You could set a manual intervention to only happen if you are deploying during business hours. You can also configure steps to run in parallel with one another.

These conditions allow you to have a degree of control with your deployments.

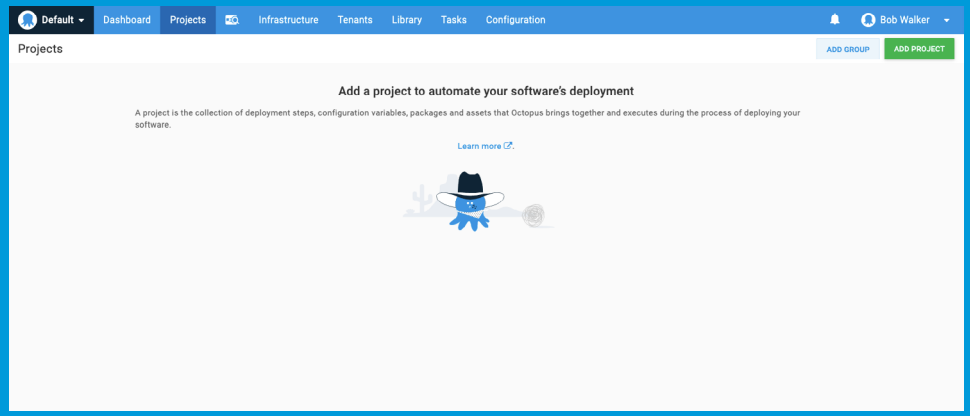
Every component's deployment should be automated
A typical scenario we see is application deployments are automated, but the database piece is not. That is still a manual process. What ends up happening is on the night of deployment to production a DBA has to run the scripts. After they finish then the automated process can be kicked off. Because this is manual, there is a good chance the scripts were not included in the deployments to dev or testing or staging. Without prior testing, the likelihood of success goes down and the deployment time goes up.

Essentially there is this great automated process which takes a few minutes to finish, but it is dependent on a manual process which takes anywhere from 10 minutes to an hour to complete. Every component of the application needs to be automated — even the database. Octopus Deploy integrates with many database deployment tools to help with this sort of automation.

Setting up the project

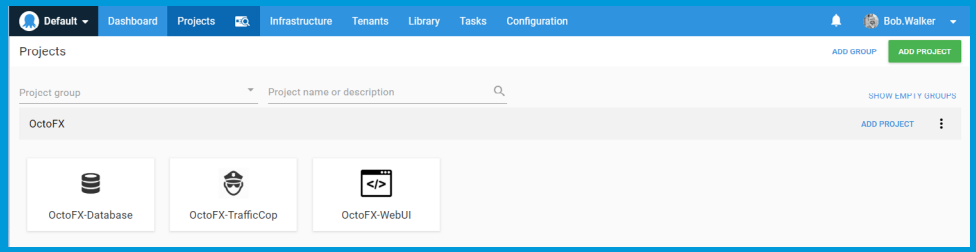
Let's configure a project using all those principles. We are going to be deploying a sample application, OctoFX. It is a small ASP.NET application with a database and a user interface. When we are finished with this setup, we will have three projects. One project will deploy the UI, and another project will deploy the database, and a traffic cop project to coordinate those deployments.

Let's first get the project scaffolding in place. Start with creating a project group called "OctoFX."



Project groups are a great way to organize your deployment projects. They have many uses, not only do they visually separate the projects, but you can also configure the dashboard to hide/show specific project groups as well as configure permissions to restrict access to them.

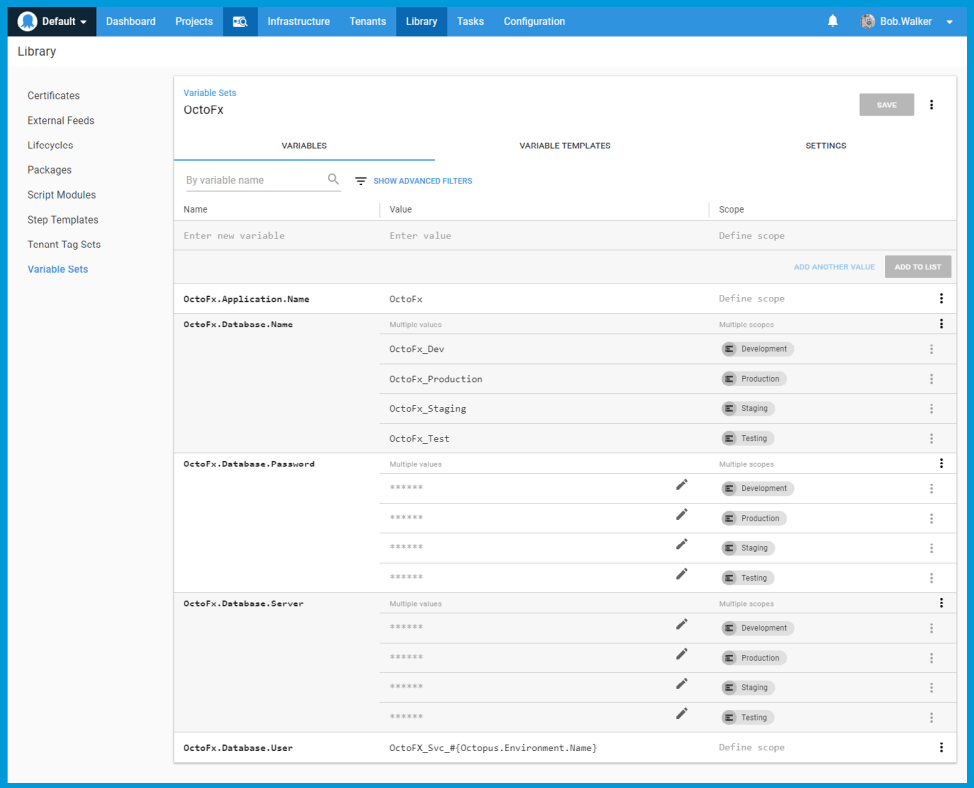
That group looks a little empty. Let's add in the three projects we discussed earlier.



Just like with tenants, adding an image to your project is a useful way to set them apart from other projects visually. In addition to supporting .jpg and .png files, we also support .gif files. Which means you can have an animated icon to add a little flair to your Octopus Deploy instance!

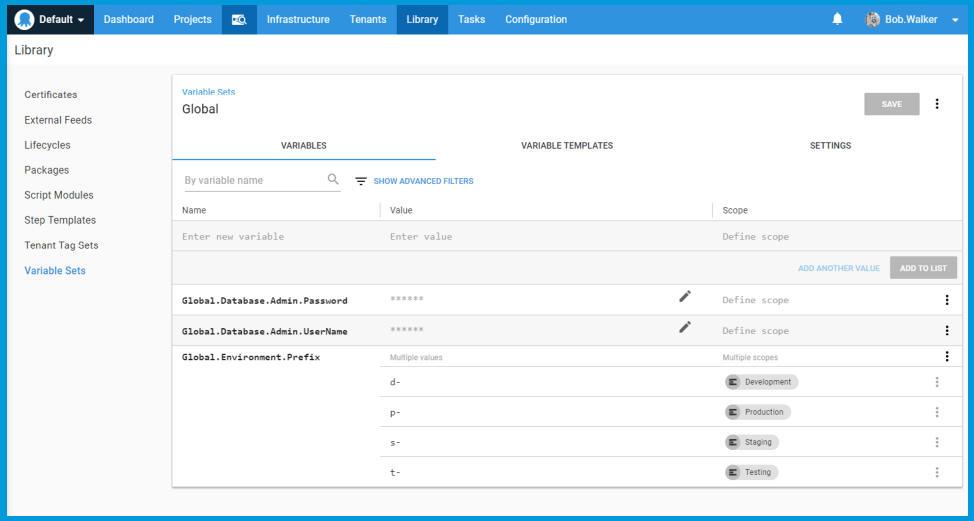
Sharing variables between projects

We have the three projects set up, but there is going to be a need to share some common variables between them. Some which come to mind right away is the SQL Server to deploy or connect to, the database name, the application name. To accomplish this, we are going to create a library set for this specific application.



A project can reference 0 to N number of library sets. Variable naming is significant. A good practice is to use a NameSpace style syntax on naming, [LibrarySetName].[ComponentName].[SubName]. Project variables can then be called [Project].[ComponentName].[SubName]. Detailed variable names give you the ability to distinguish in the project steps and logs which variable is from a library set and which is from the project.

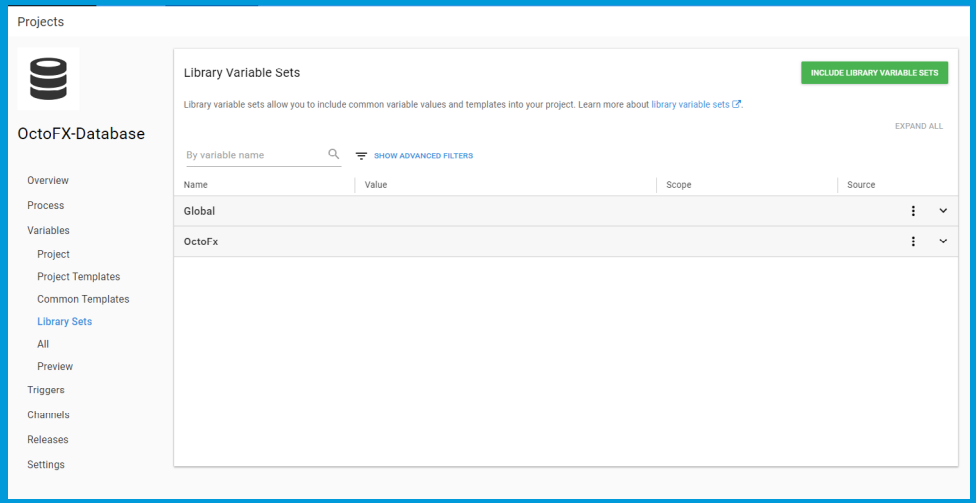
It is also a good idea to have a couple of other library sets to handle some non-project specific values. For example, global and another to store any infrastructure as code or IaC variables. The same naming convention applies as the project specific variable set example, just replacing “OctoFx” with “Global.”



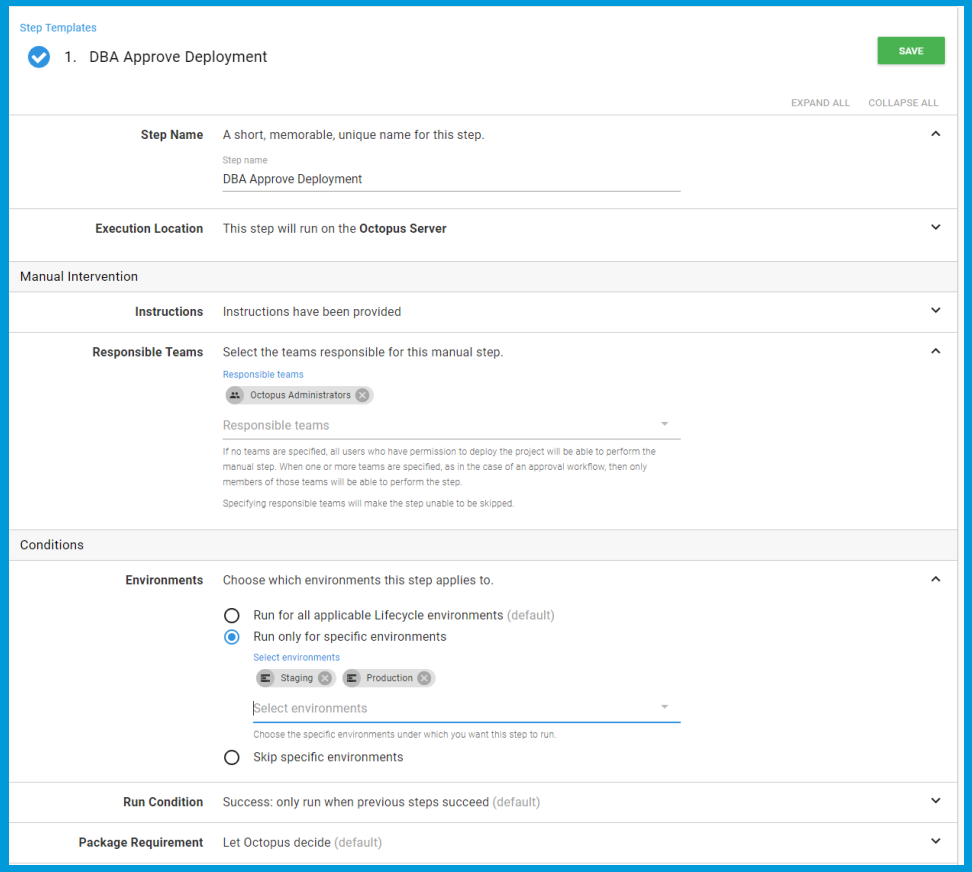
OctoFX-Database Project

The first project we are going to configure is the OctoFX-Database project. If we follow the recommendations from earlier in this chapter, we are going to assume that the SQL Server is running but this database and required user does not exist. We are going to have steps in place to check to see if the database exists as well as the necessary user for the environment. If they don’t exist, then we will need to create them. Also, we want to build some trust in the process; this can be done by having a manual intervention for a DBA to approve.

Before adding in steps to the process, we need to add a reference to the library variable sets we created earlier.



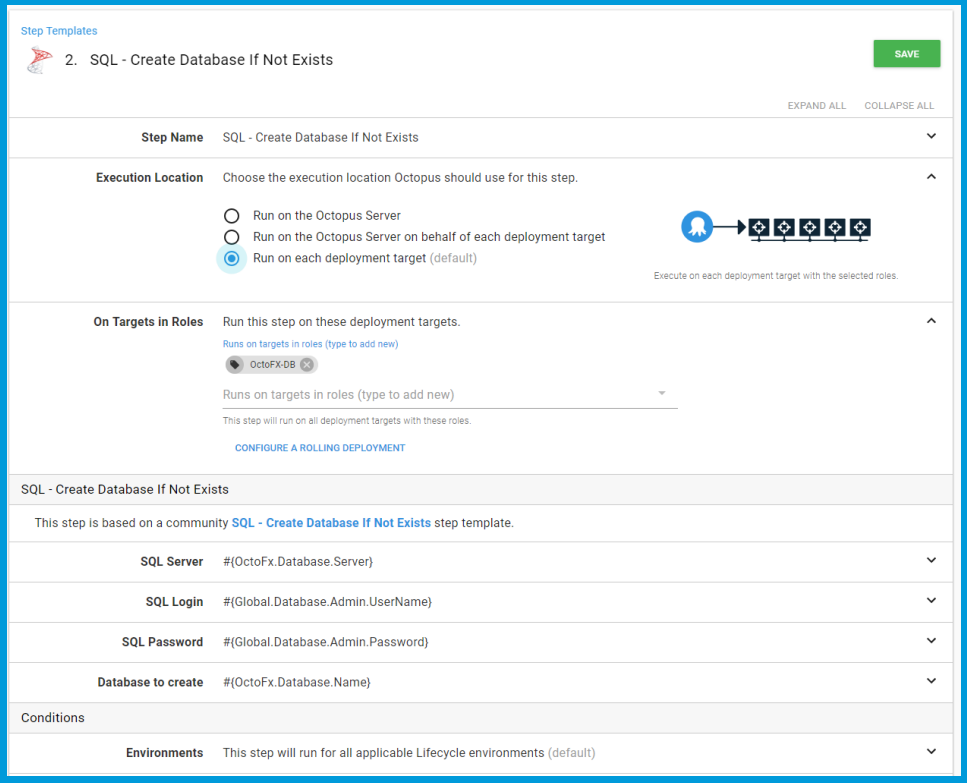
Next, we are going to add in the manual intervention for the DBAs to approve. If you are configuring a fresh instance of Octopus Deploy you likely haven’t had the chance to configure a DBA team yet. That is okay, for now, just put in Octopus Administrators or a team which already exists. We will get to configuring teams in a later chapter.



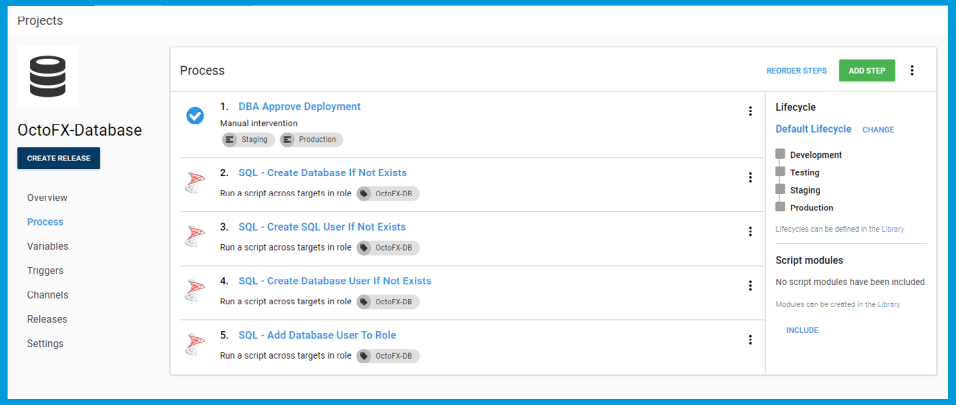
> This project deploy a database package using DBUp, a free database deployment tool. Some tools provide the ability to generate a difference report before deployments which Octopus can store as an artifact and a DBA can download and review. In that case, it makes more sense to have the manual intervention occur after that report has been generated.

Many community step templates have been created to help with some of this database scaffolding. We are going to be using the SQL - Create Database If Not Exists step template to create the database if it doesn't exist. We are going to be using variables out of the library sets we brought in previously.

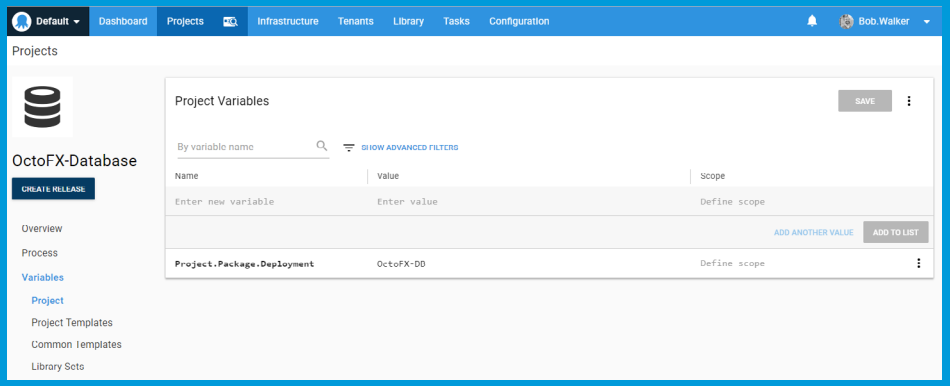
For now, we are going to execute this script on a tentacle with the role "OctoFX-DB." Later in the book, we will convert this over to using workers.



There are few more maintenance tasks to add, such as creating the SQL Login if not exists, assigning that user to the database and assigning them to a role. Keep in mind, all of the steps being added are occurring before an actual deployment happens. Without even doing a deployment we have added in five steps. That is to deploy the database. Imagine if this project also deploys a website, a windows service and other components. The project would become very hard to manage.

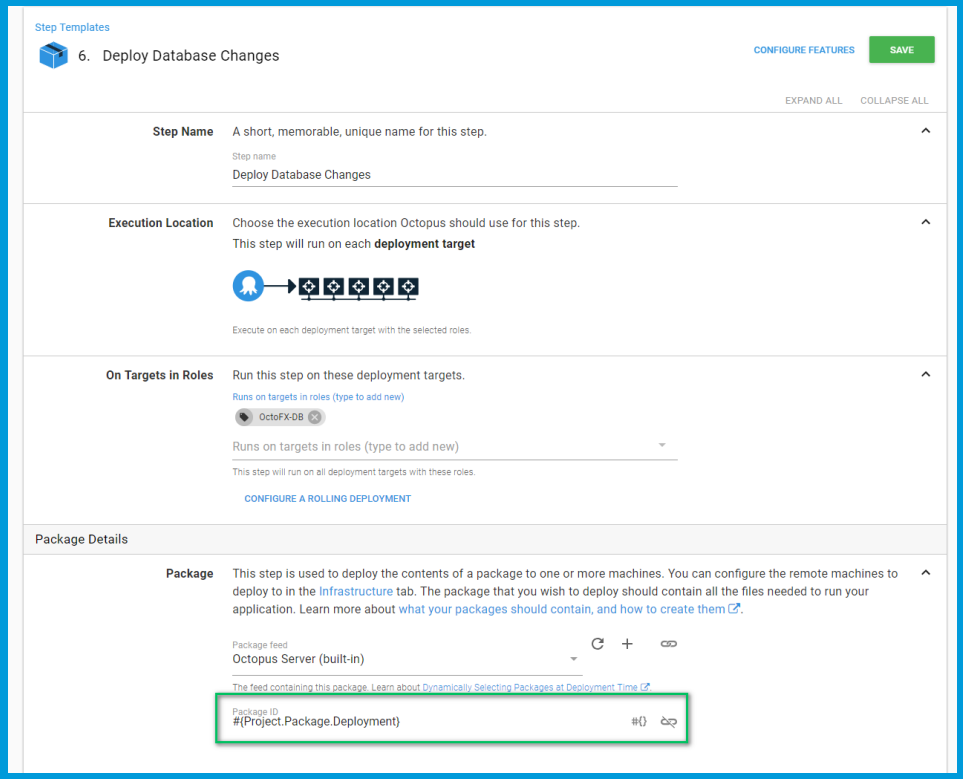


Now we are ready to configure the database deployment. As you most likely learned when creating a PoC or other deployment projects, you need to package up the database into either a NuGet or a Zip file. In this book, we haven't configured anything to push the packages to Octopus Deploy's internal package feed for it to deploy. However, the deploy a package step requires us to specify a package. The way we are going to short-circuit this chicken/egg scenario is by using variables. In a later chapter we will worry about getting the packages uploaded, but for now, we want just to set a variable.

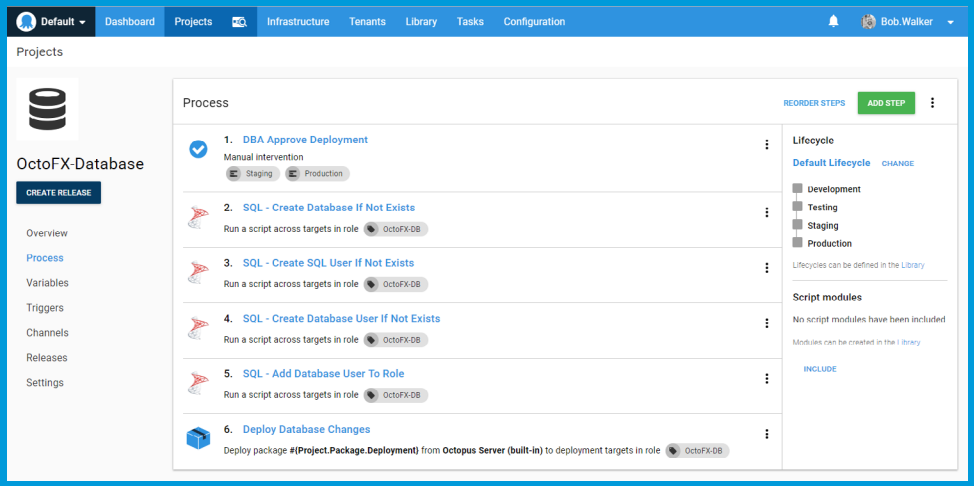


> Using a variable to reference a package also makes it easier to clone this project to use with another application.

Now that we have our package referenced as a variable, we can add the step to deploy the package and run the scripts on the database.



Finally, we are done with the database deployment process. The process itself is relatively simple; there might be some more approvals or some additional steps you will want to add. Again, the idea is to keep all the database deployment work in this specific project.



Don't spend too much time on the actual steps in the process. The major takeaways from this are the database project is responsible for everything required to create, configure and deploy a database. You might be using a different tool (like Redgate or RoundhousE) to do your deployments which include some additional features.

OctoFX-WebUI Project

Now it is time to move onto deploying the UI. Unlike the previous section, we will not be walking through all the necessary steps you need to do to configure your project. We will follow the same rules as before; the project will do all the work required to deploy the web application as if it was the first time.

That being said, do not forget to reference the variable sets in this particular project just like you did with the database deployment project.

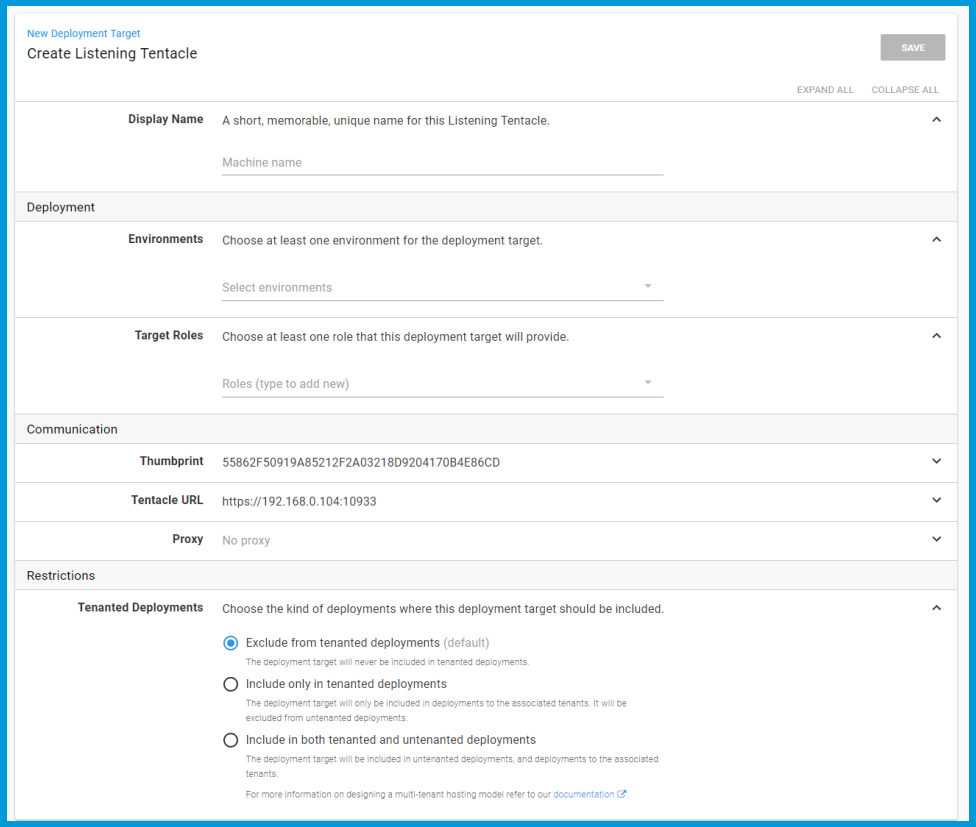
7

Everything you wanted to know about

DEPLOYMENT TARGETS AND ROLES

(but were afraid to ask)

We have environments and project processes configured. Now we need to configure a couple of deployment targets. Deployment targets are what you will deploy the code to. Originally, deployment targets were Windows VMs running the tentacle windows service. The Octopus Deploy server connects to that tentacle and instructs it to do work. But a deployment target is not just a tentacle. As time has gone on we have added more and more deployment targets types. Now there are Windows Targets, SSH Connections (for Linux machines), Azure Targets, Kubernetes Clusters, Offline Drops, and Cloud Regions. That list keeps growing and growing. We are willing to bet that by the time you read this book that list will have changed.
When you were setting up a proof of concept or a pilot project chances are

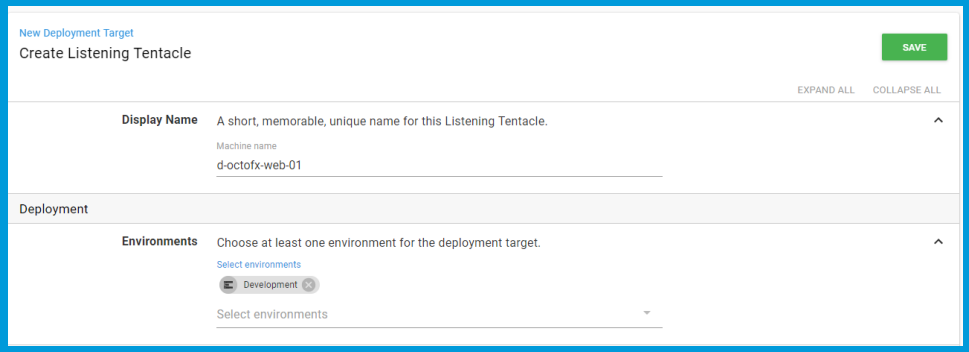


you didn't think to the add target form. We know we didn't. We wanted to get to defining our deployment process. We didn't give much thought to roles, machine policies or tenants. Unfortunately, that is where we see a lot of misconfiguration.

Naming

Naming. Easy to learn. Hard to master. A good machine name will describe the machine and its purpose in a succinct manner. You might have your own internal naming conventions for VMs. If that is already in place, use that. Naming conventions tend to fall apart with PaaS targets such as Kubernetes, Azure Web Apps and Service Fabric clusters.

If each application gets its own set of resources or its own machine, a good naming guideline to follow is [EnvironmentPrefix]-[AppName]-[Component]-[Number]. Or [EnvironmentPrefix]-[AppName][Component][Number].
For example, the machine hosting the OctoFX Website in Dev would be **`d-octofx-web-01.`**



If you have a web server hosting many websites a succinct name is important. For internal websites on a test machine, a good name would be **`t-internal-web-01.`**

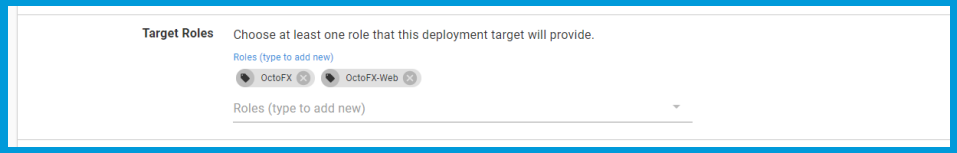
Target Roles

Target Roles (roles) are a little trickier and tend to trip up a lot of people. If you remember when we created the deployment process we had to pick a role for the step to run on. During a deployment, Octopus Deploy will grab all machines with that role and run the deployments on those machines.

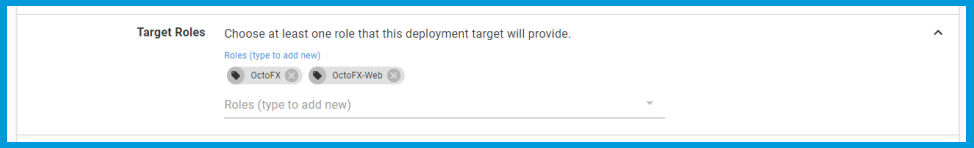
All too often we run across scenarios where a customer has created a role called IIS-Server because they only had one or two servers hosting all their web applications. This worked fine until they decided to move a small subset of applications to a new server. They tried using the same role, IIS-Server but when they went to deploy they had all the projects being deployed to both the new and the old servers.

The problem is the role of IIS-Server is too generic. Some of the customers we have talked to think that a machine can only have a single role. But in reality, roles are nothing more than tags. You can have 1 to N number of roles assigned to a specific target. Because of that, we recommend using specific roles.

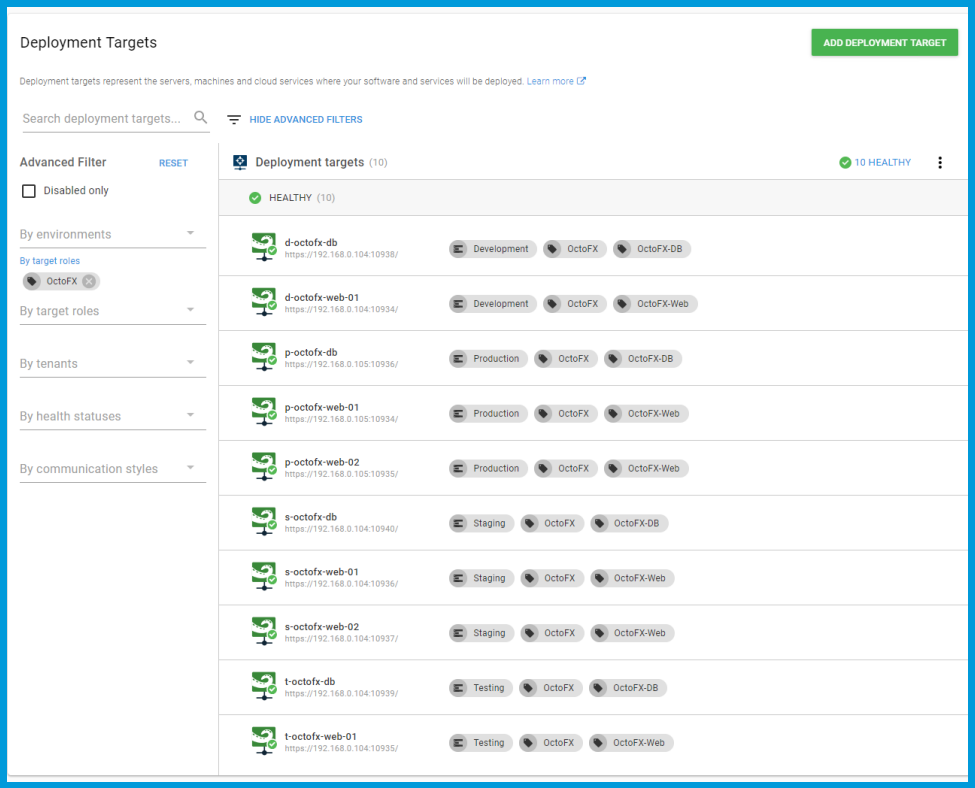
For the machine deploying to OctoFX-Web, we recommend the roles **`OctoFX`** and **`OctoFX-Web.`**



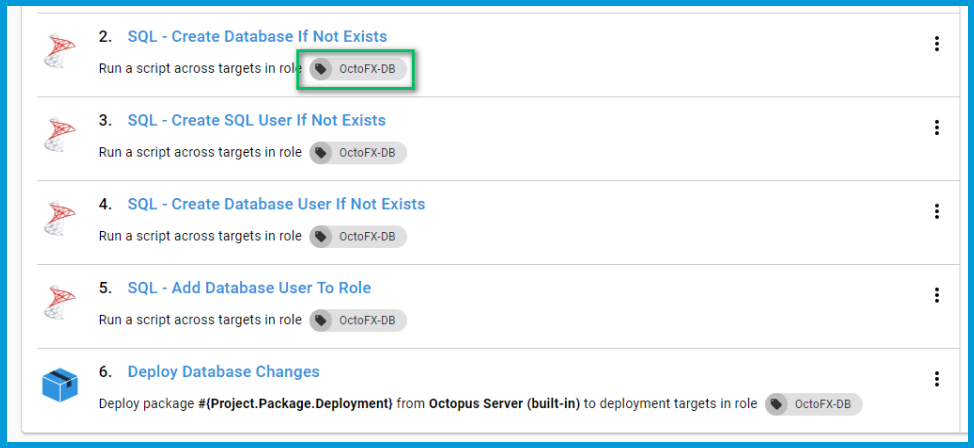
For the database, the roles would be **`OctoFX`** and **`OctoFX-DB.`**



The reason we recommend including the role OctoFX is for server organization and searching. Using the filtering functionality on the deployment target page you can see all the machines for OctoFX.



The roles `OctoFX-Web` and **`OctoFX-DB`** are what is being used in the deployment process.



If you wanted to keep track of all the IIS Servers in your infrastructure then, by all means, add the role `IIS-Server`. Don't have any steps in your process target that role.

In the event you have several applications hosted on the same target then you can add a role for each application.

It does feel a bit tedious to be adding roles to an existing target, but it makes changes in your infrastructure much easier to change. For example, two new servers are created for a subset of applications. You only need to remove the roles from the old servers and add them to the new ones. You don't have to adjust your deployment process or create a new release. In fact, once the roles are added you can re-run the deployment for the environment and the new machines will get the latest and greatest code. In later chapters, we will discuss triggers on how to automate this.

> Adding dozens upon dozens of application roles to a single target is a good litmus test the server is doing too much. If you have the resources consider splitting up the server.

Tenants

The last piece of the add deployment target form is the tenant's section. We will be covering tenants in later chapters. The major takeaway for this chapter is this will allow you to specify if the target is for tenant deployments. If the deployment target is for a specific tenant then it would make sense to adjust the name [EnvironmentPrefix]-[TenantName]-[AppName]-[Component]-[Number]. Use d-ford-octofx-web-01 if we wanted the machine to deploy the OctoFX website for Ford to Dev.

It is possible for a deployment target to be used for both non-tenant deployments as well as tenant deployments. This is typically done for virtual machines which host lots of applications and tenants. Most often in a development or testing environment. This is not something we would recommend in production, but it is possible. It would make more sense in the lower environments when the resources tend to be more limited.

Conclusion

We now have deployment targets configured and ready for the code to be pushed to them. By using specific roles we have also positioned ourselves so if we do make changes to the infrastructure we don't have to make changes to our projects. We now have targets to deploy to, in our next chapter we will talk about how to get the code out to them.

8

LET’S DEPLOY SOME CODE

In previous chapters we got the environment scaffolding put together, then we created some projects for our application. We wrapped it up by setting up deployment targets for the projects. After all that (whew!) it is finally time to deploy some code!

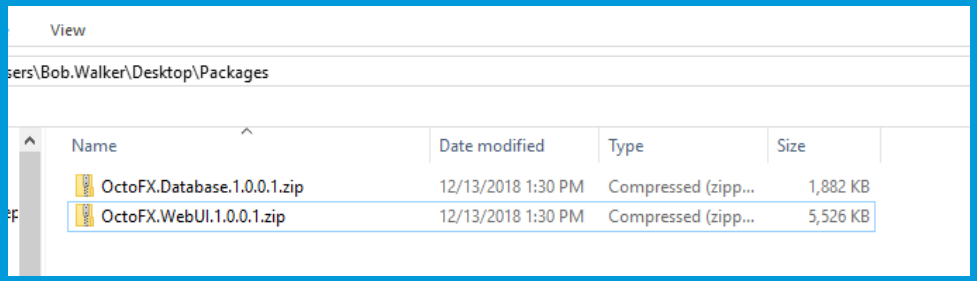
> Typically the previous chapters will take anywhere from 10 to 20 minutes to set up and configure. It does seem like a lot at first. The overall goal of the previous chapters was to provide you the foundation to get all the scaffolding set so you can scale your Octopus Deploy.

Octopus Deploy will deploy code using what is known as a release. Releases are snapshots of your project's process, variables, and packages. You deploy that release to the various environments and servers. The idea behind a release is the core concept of Octopus Deploy, you build your code once, and you deploy that same code to all environments. The same is true of the release. The release is created once, and that same release is deployed to all environments.

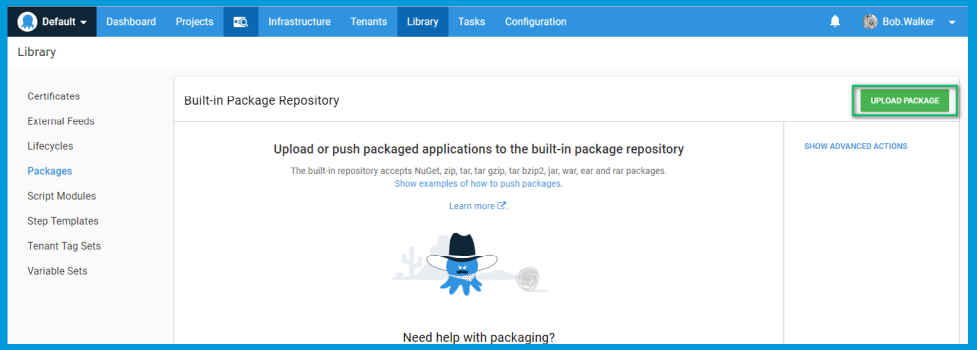
Creating the first release

At this point, we do not have our build server integration set up. Which is okay, we don't need to do that at this very second. An easy way to test our deployment process for the Web UI is to go to a development web server and zip up the application folder. Zipping up the folder will let us test our deployment process without having to worry about configuring the build server and getting that connection wired up.

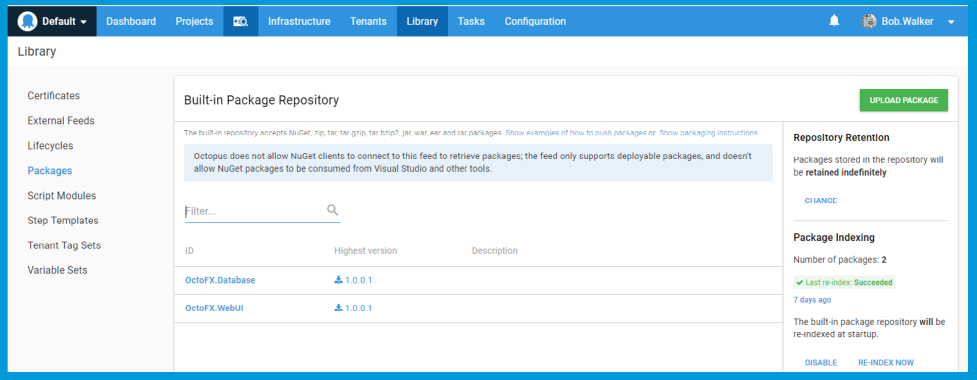
For this first release, I have created two packages, one for the website project and another for the database project. For the website, we went out to the dev server and packaged up the site into a zip file.



Now we need to upload them to the Octopus Server. Go to Library -> Packages and clicking the upload packages button in the top right to upload packages to the server.



Now we have some packages to deploy.



Before diving into the release let's talk about everyone's favorite subject, version numbers.

Version Numbers

The technology you're working with will, in some cases, determine the type of versioning scheme that you choose. We recommend using Semantic Versioning for your applications unless you are deploying artifacts to a Maven repository, in which case you will need to use Maven Versions.

Consider the following factors when deciding on the versioning scheme you'll use for your applications and packages:

1. Can you trace a version back to the commit/check-in the application/ package was built from? For example, We stamp the SHA hash of the git commit into the metadata component of the Semantic Version for Octopus Deploy which makes it easier to find and fix bugs. We also tag the commit with the version of Octopus Deploy it produced so you can quickly determine which commit produced a particular version of Octopus Deploy.

2. Can your users easily report a version to the development team that supports #1?

3. How easy will people understand the changes that have been made to the software? For example: bumping a major version component (first part) means there are potentially breaking changes, but bumping a patch (3rd part) should be safe to upgrade, and safe to rollback if something goes wrong.

4. Does your tool chain support the versioning scheme? For example: Octopus Deploy supports Semantic Versioning, which enables enhanced features like Channels.

The Octopus Deploy ecosystem includes a wide variety of external services which care about versions, with some of them being quite opinionated in their versioning implementations, with potential inconsistencies amongst them. Rather than implementing a "lowest common denominator" approach, we've taken a "string-based" approach. You can leverage the idiomatic/natural versioning schemes of your target ecosystem.

Octopus Deploy supports regular Semantic Versioning or SemVer. A typical example would be: **`1.5.2:`**

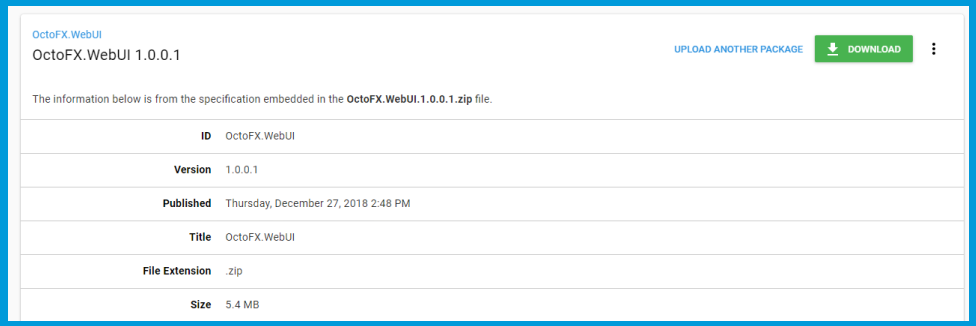
- Major
- Minor
- Patch

We also support pragmatic versioning, which is including a fourth number, 1.5.2.3. Not every version can fit into three numbers.

- Major
- Minor
- Patch
- Build

Octopus Deploy also supports pre-release tags, for example, **`1.5.2.3-rc.1.`**

As you can see after uploading the packages, the package versions are **`1.0.0.1.`** Octopus Deploy read the version number from the file name.



Strictly speaking about SemVer 2.0, a version like 1.5.2-rc.1 is considered a "pre-release" and 1.5.2 would be considered a "full release." In practice, these concepts carry weight when you are talking about hierarchies of application dependencies like standard NuGet packages or NPM dependencies. This kind of strict semantic versioning enables dependency management tooling to interpret what kind of changes each package version represents. For example, they can automatically protect your software, by preventing accidental upgrades to pre-release versions, or versions that might introduce breaking changes.

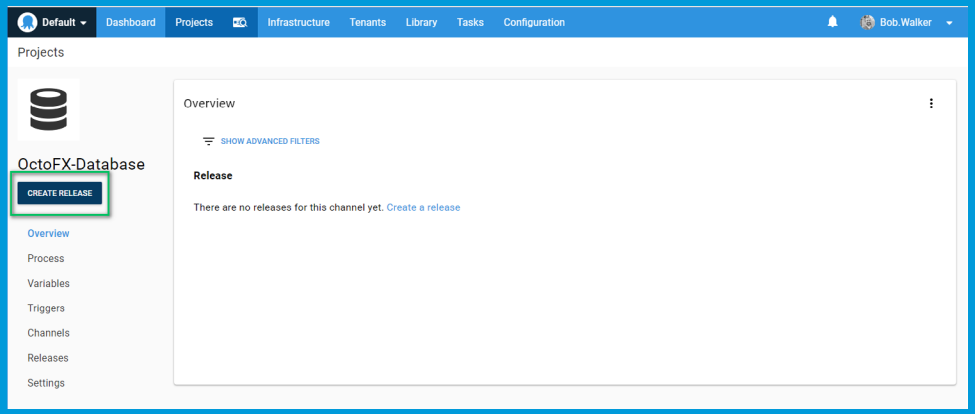
A typical Maven version string is split into five parts:

- Major
- Minor
- Patch
- Build number
- Qualifier

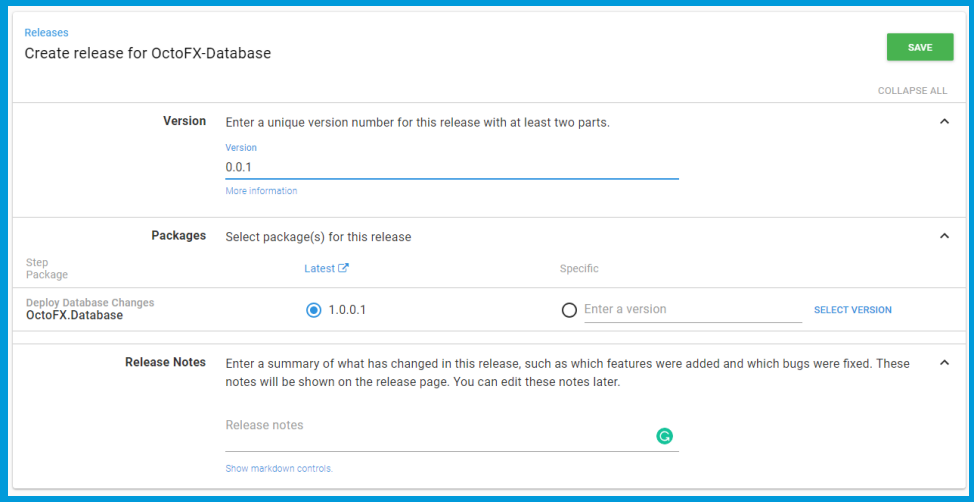
The Major, Minor, Patch and Build number are all integer values. The Qualifier can hold any value, although some qualifiers do have special meaning.

Creating the release

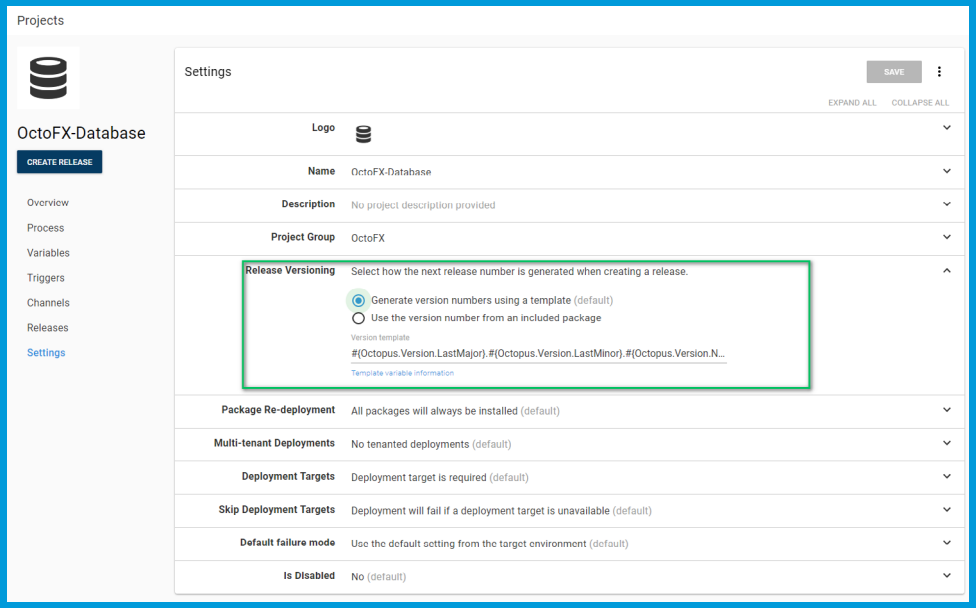
We are going to create a release to deploy the database changes. There is no sense in deploying code without a database. Creating a release is done in the UI by going to the project page and clicking on the create release button.



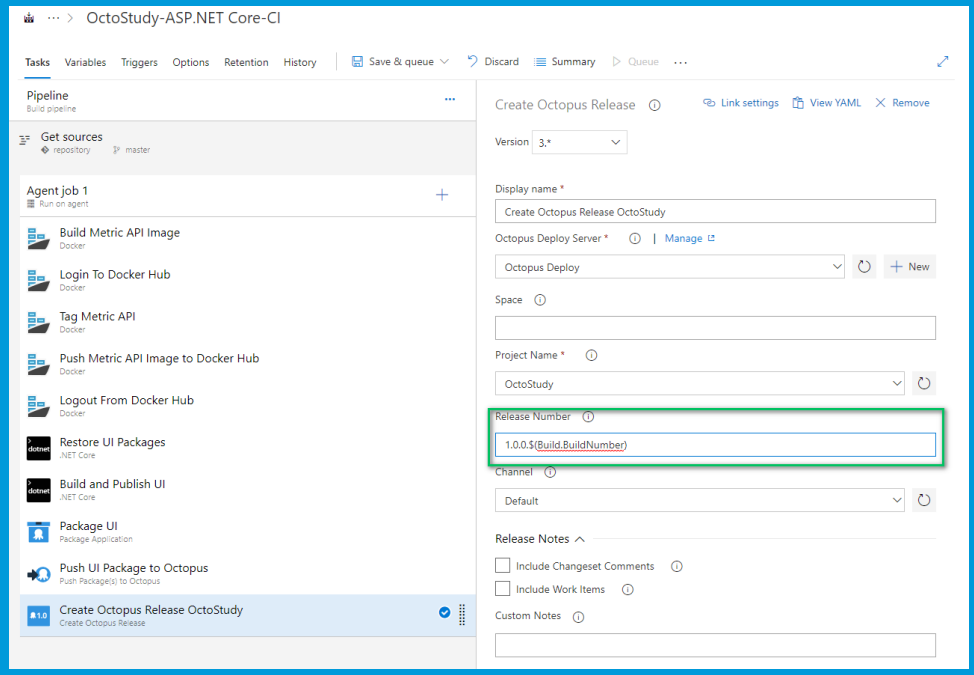
Before clicking on the save button, we want to pause on the screen and walk through what each value means.



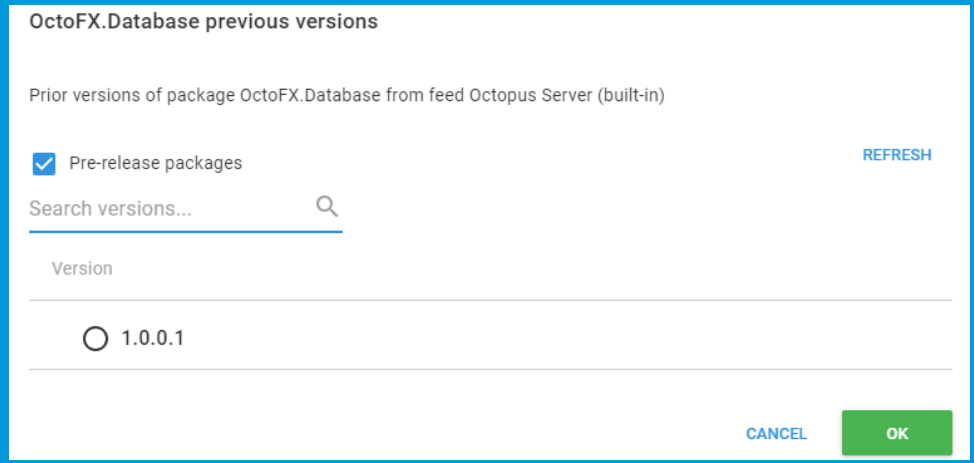
The first option is the version number for this release. By default, Octopus Deploy is using a version number defined by the template which you can configure by going to the settings screen on the project.



We recommend leaving that as is. When you set up the integration with the build server, you can configure the plug-in to specify the build number when creating a release. When manually creating releases through the UI (typically done to test out changes to the process) the default option will create a new version for you with requiring you to have to make a bunch of changes to the version number.



The next option on the release creation screen is the package selection section. By default Octopus Deploy will select the latest version it can find. However, there are cases when you want to choose a specific version. To do that you can click on the select version button.

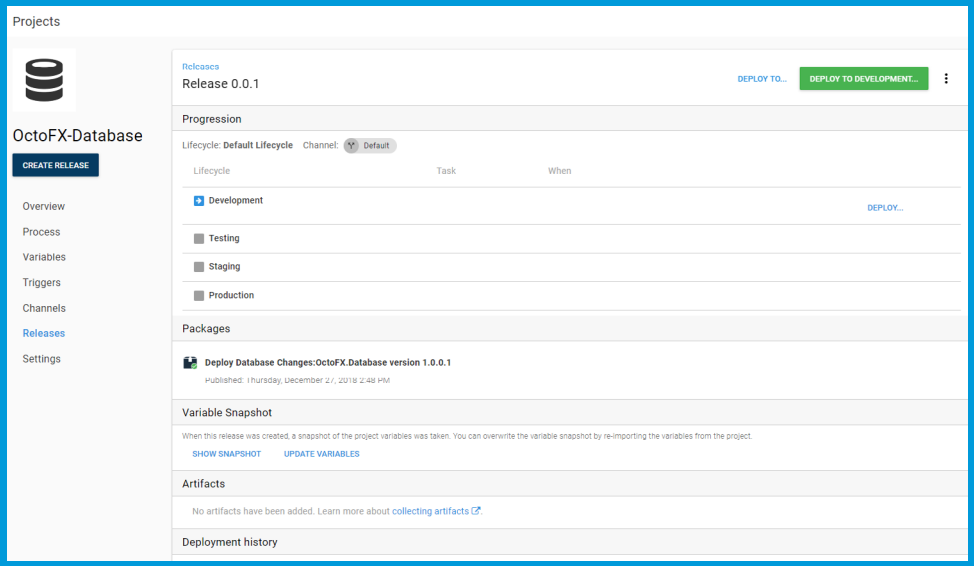


Finally, there are release notes. The release notes support markdown syntax. Because we are just testing out this deployment process for the first time, we are going to skip entering in anything into the release notes. We recommend including as much information as you think as necessary for each release. Some ideas include:

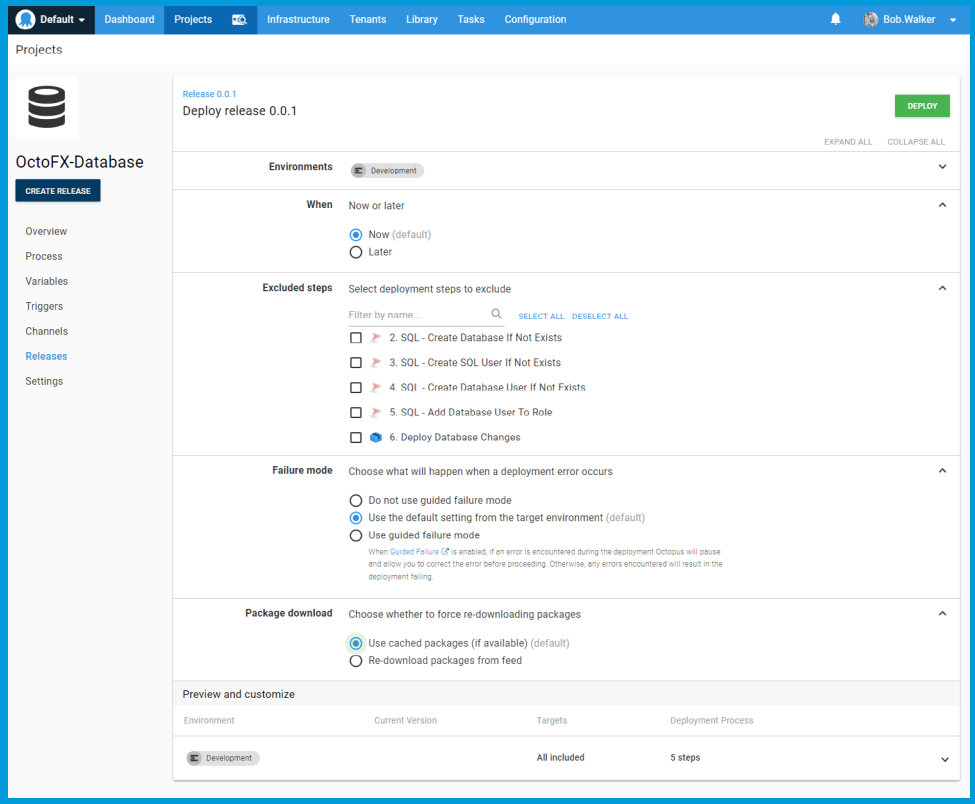
- All Git Commits since the last deployment
- Links to your internal tracking system such as Jira.
- A detailed description of changes
- Git Commit Hash which triggered the deployment

Deploying the release

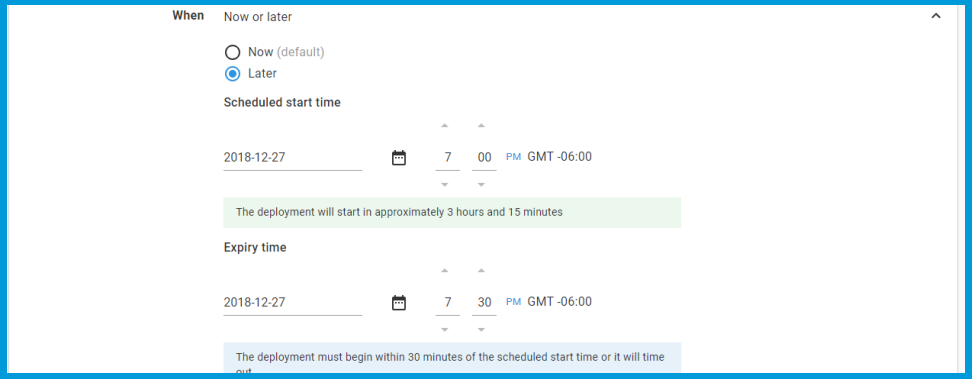
Clicking the save button on the release will take you to the release overview screen. From here you can view much information about the release, such as what environments it has been deployed to, what variables have been snapshotted, what packages are being deployed, any artifacts created as well as the deployment history.



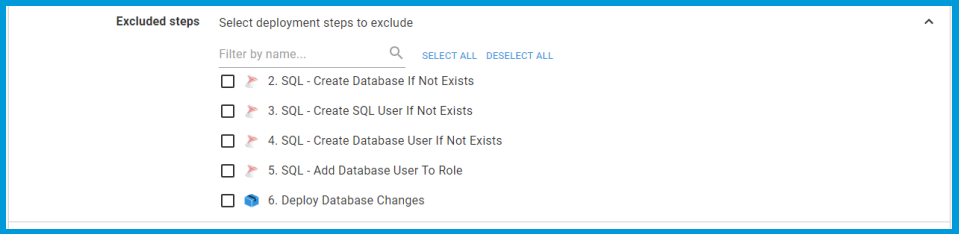
We don't need to do anything on this screen, so we can proceed to the deploy to development by clicking on the **`Deploy to Development...`** button. When you do that you will be taken to another screen to select some options. Just like the release creation screen we want to take a moment to walk you through each of the options.



The first section we will look at is allowing you to schedule when the deployment will occur. Scheduled deployments enable you to have a web admin or a DBA schedule a deployment to production at 7:00 PM and not have to be online when the deployment occurs. If they are like most web admins and DBAs, we have worked with they are busy. They only want to be involved with a deployment when something terrible happens.

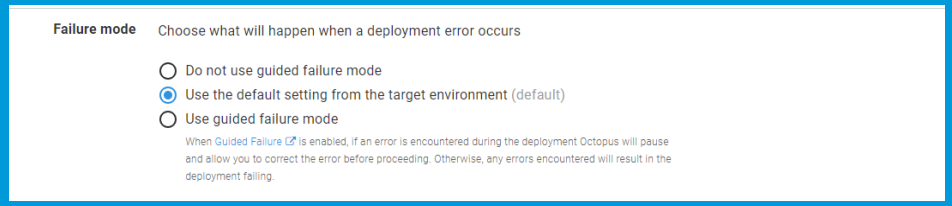


The next section allows you to configure if any steps are disabled with this deployment. We don't recommend using this feature for regular deployments. However, if you are testing something out and it keeps failing on a specific step then disabling steps will decrease the turn around time.



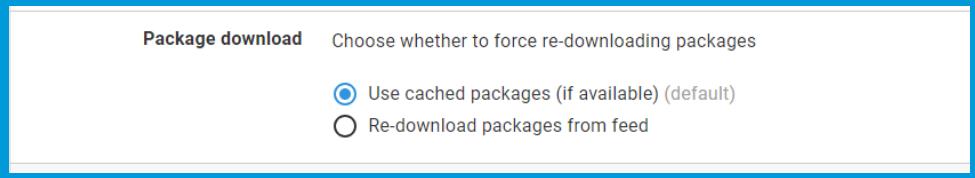
> Excluding steps using this interface provides an excellent alternative to disabling the steps in the process. Excluding steps here allows you to turn off steps without changing your process, which requires a new release to be created.

The next section configures how the Octopus Server will handle a failure. You have a couple of options when it comes to failures. You can fail the deployment. Alternatively, you can tell the Octopus Server to pause and wait for input when a failure occurs. Guided failures are useful when you have a deployment step which randomly fails, and you want to be notified of it so you can choose what to do with the step (skip it, retry or abort the deployment).



> After a process is configured and tested the chances of it failing are very slim. Typically we see a failure occur because permissions on a database got changed, a file share isn't available or something else directly unrelated to the deployment. Those types of failures require an external change (fix the database permission) and then a retry will work.

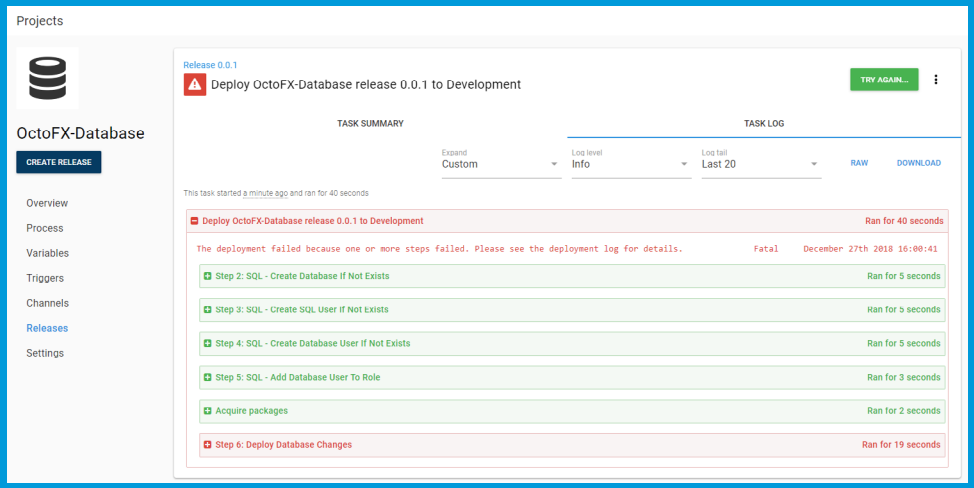
The final section is the package download option. We recommend leaving this option alone. The only reason you would want to force a download of the package is for debugging purposes. Just like any other cache, using the package cache should speed up the overall deployment process.



Because this is the first time we are running this deployment process we should leave all those options alone and click on the **`deploy`** button at the top right corner of the screen.

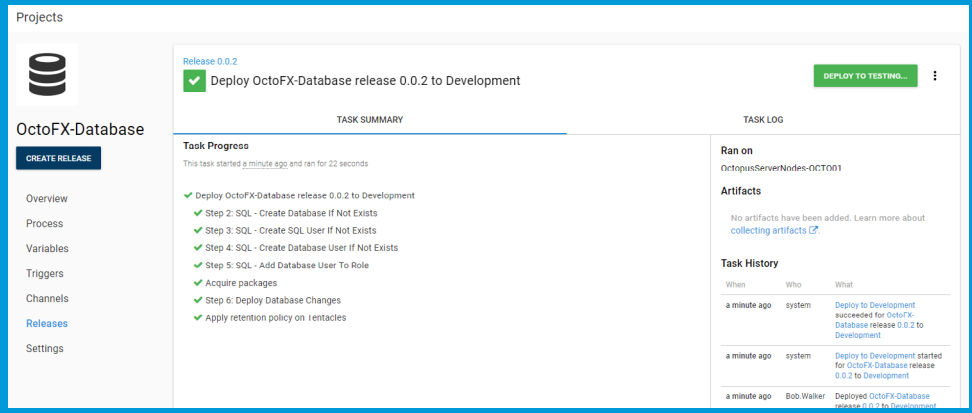
First Database Project Deployment

Unsurprisingly, the deployment failed. A real surprise would be if the deployment were successful on the first try. Almost every project fails on the first release. Do not get frustrated when that happens. Believe us; it happens to everyone.

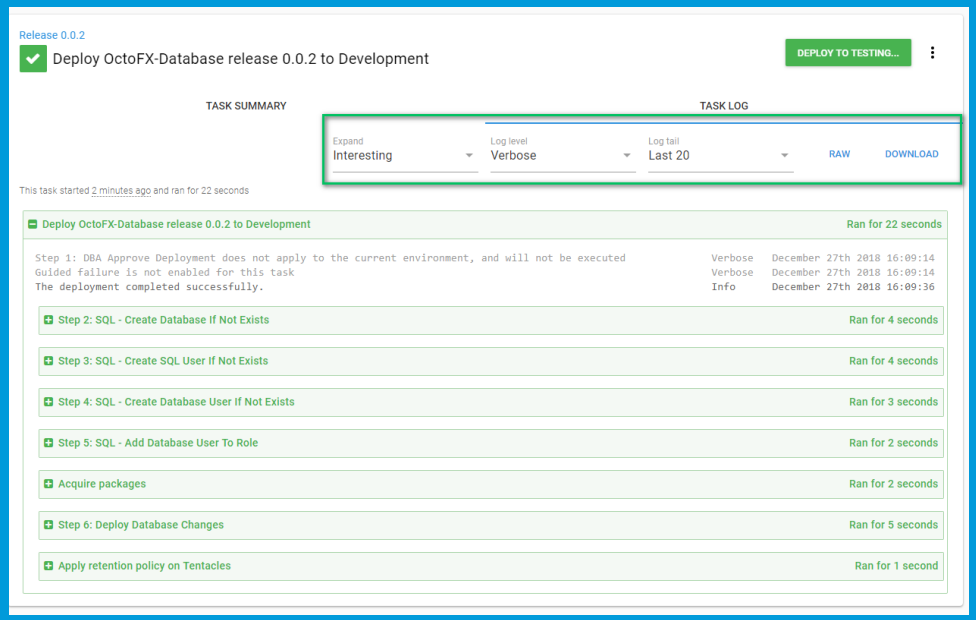


By clicking on the error message on the overview screen, we can dive right into the error message and see where the failure happened. It is making mention of the unable to find the instance to deploy to. Based on the fact the deployment was successful in the previous steps, this leads us to believe there might be a problem with some variables that were configured.

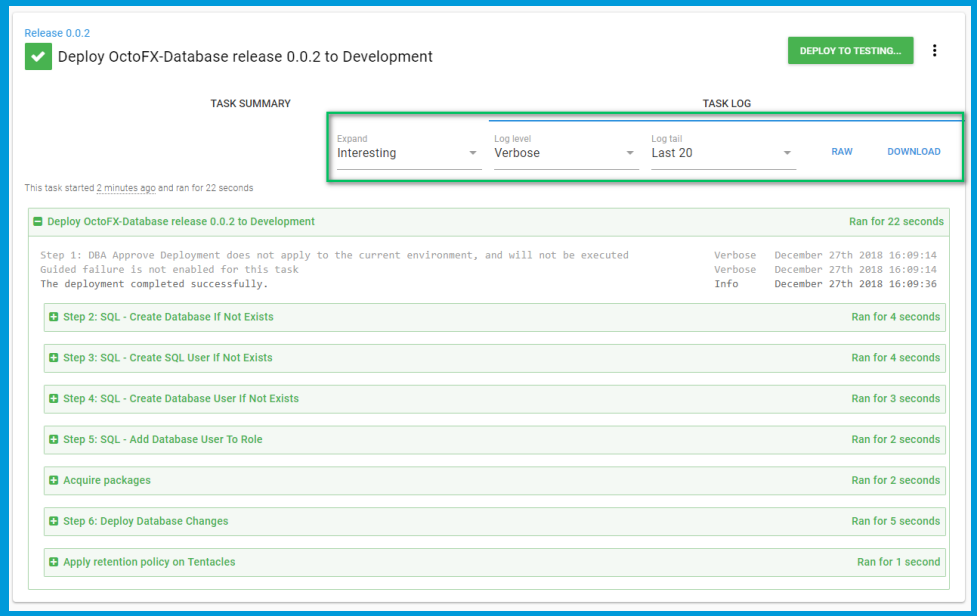
After a few adjustments to the variables, everything is now working!



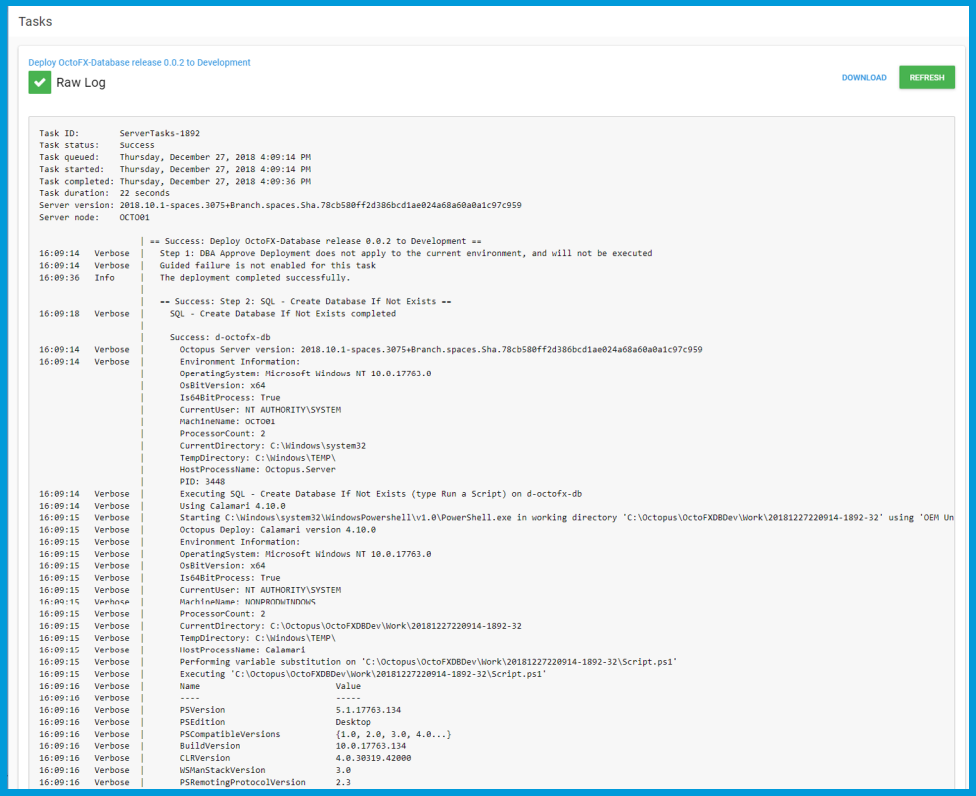
Now that we have a successful release we should take a moment to look at the task log and in particular the highlighted section below.



By changing the log level from info to verbose, you can get a lot more detail about each step. For successful deployments, this isn't very useful. It should be used for debugging purposes or when a failure occurs.



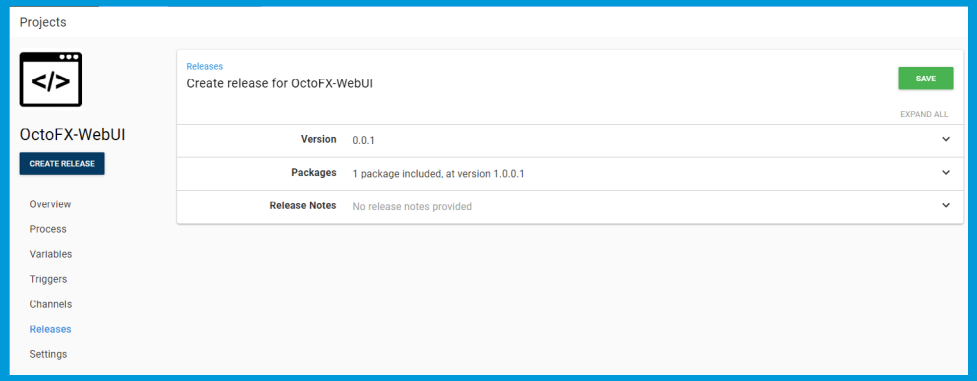
Clicking on the **Raw** button will allow you to view the log without any of the formatting. This view shows both the verbose and info log levels as well as any errors.



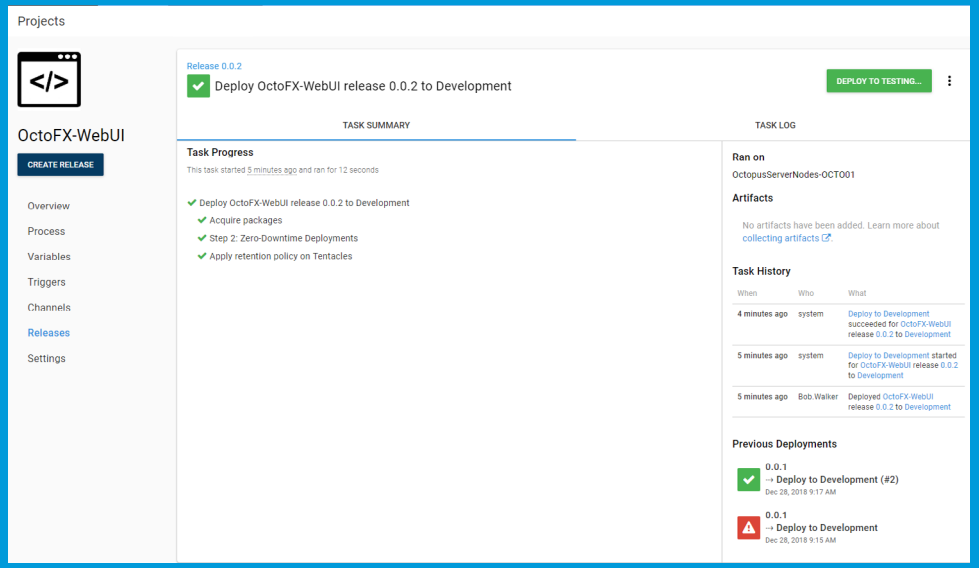
Finally, the **Download** button will download the logs so they can be zipped up and emailed to support.

First WebUI Project Deployment

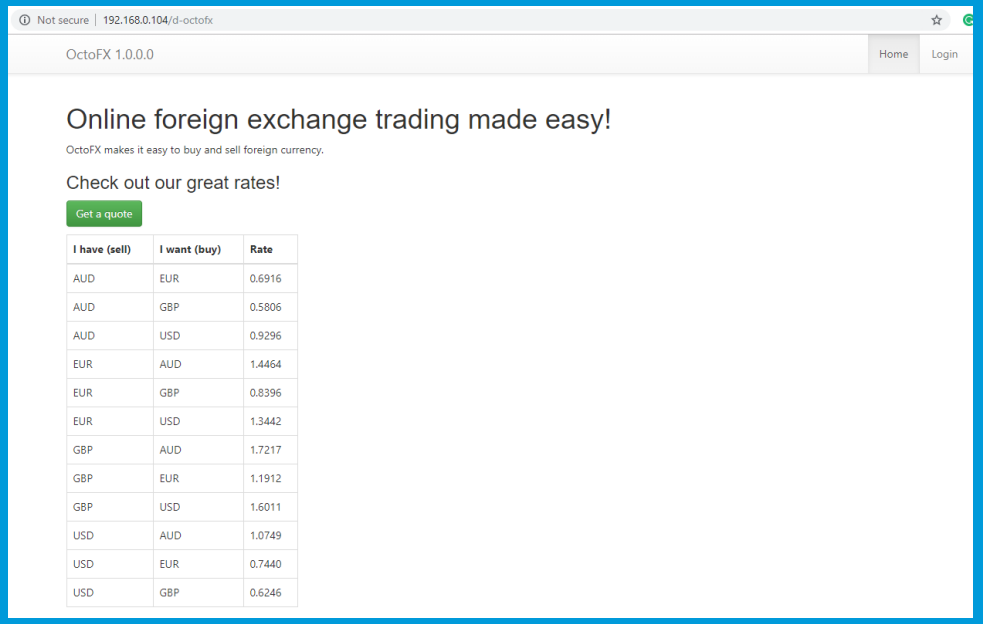
The database project has been deployed. Next up is the WebUI project. Just like with the database project, we are going to create a release for the WebUI project.



We will go ahead and deploy that to dev. Just like the database project it took a couple of times to get it to work.

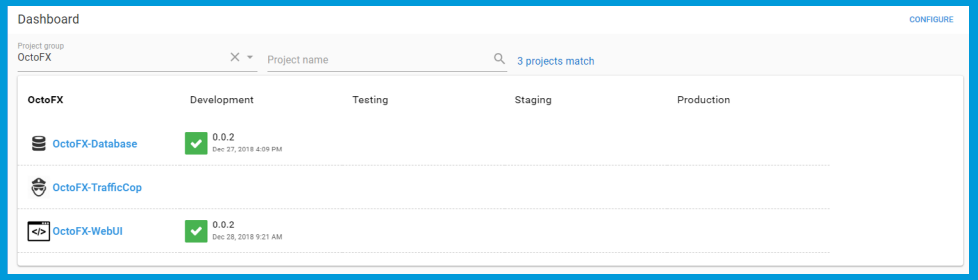


Now for the moment of truth, checking the website to make sure it deployed. Which we can see it did.



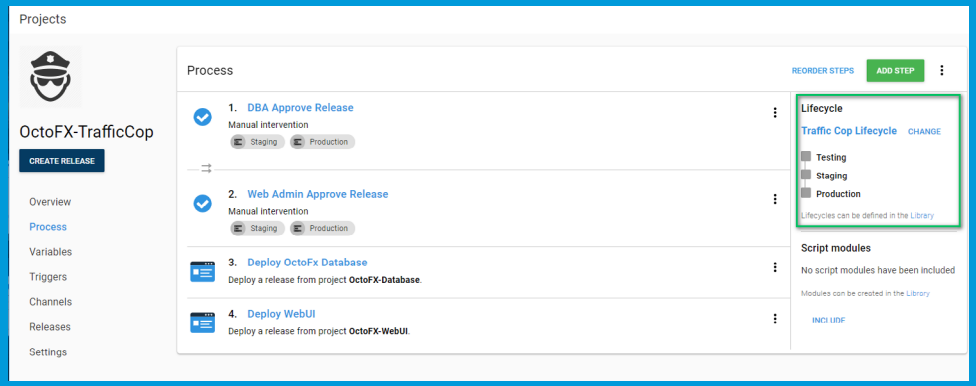
I have (sell)	I want (buy)	Rate
AUD	EUR	0.6916
AUD	GBP	0.5806
AUD	USD	0.9296
EUR	AUD	1.4464
EUR	GBP	0.8396
EUR	USD	1.3442
GBP	AUD	1.7217
GBP	EUR	1.1912
GBP	USD	1.6011
USD	AUD	1.0749
USD	EUR	0.7440
USD	GBP	0.6246

In looking at the dashboard, we can see that both projects were successfully deployed to the development environment.



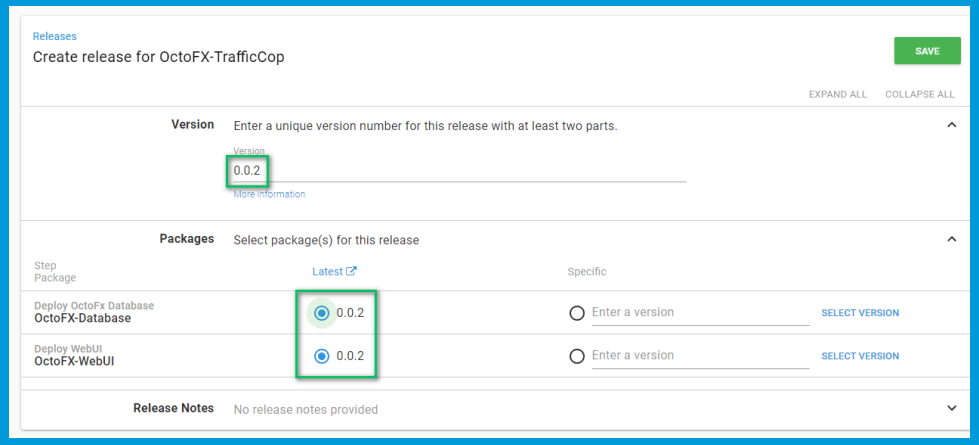
First Traffic Cop Deployment

If you recall from earlier, the Traffic Cop project doesn't deploy to the Development environment. Traffic Cop skips the development environment because we configured it use a unique lifecycle which skips Development and starts with Testing. The reason for this is because in most CI/CD scenarios the build server is the one creating the releases for the Database and WebUI project. The build server wouldn't create the Traffic Cop release because not all releases will be deploying both applications. There are several scenarios where only a database change is needed (sproc bug fix, new index, etc.) or only a code fix is required (bug fix, updating CSS, etc.).

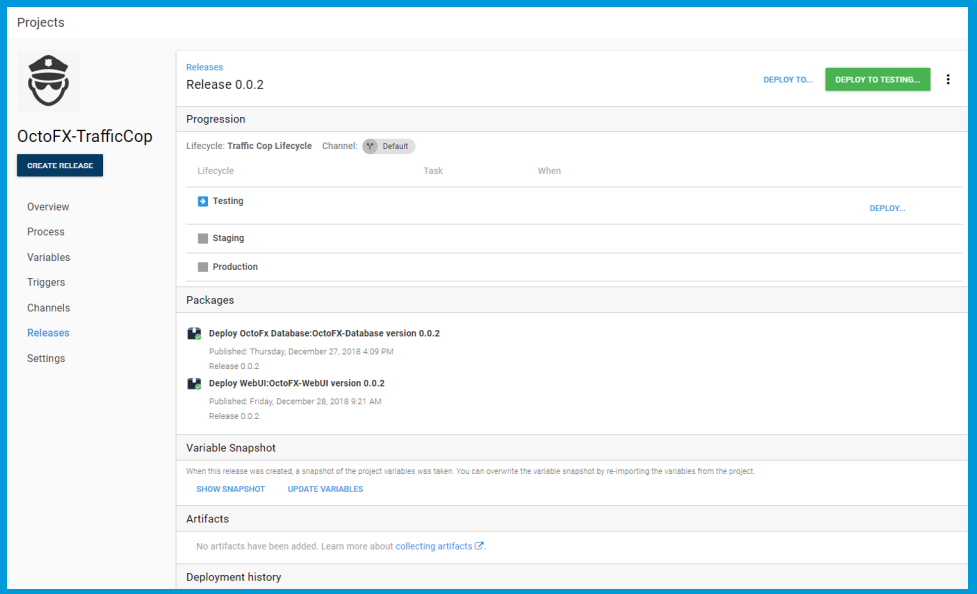


Because this is the first time this project is being deployed via Octopus Deploy, it makes sense to take advantage of the traffic cop project. Let's first start with creating the release.

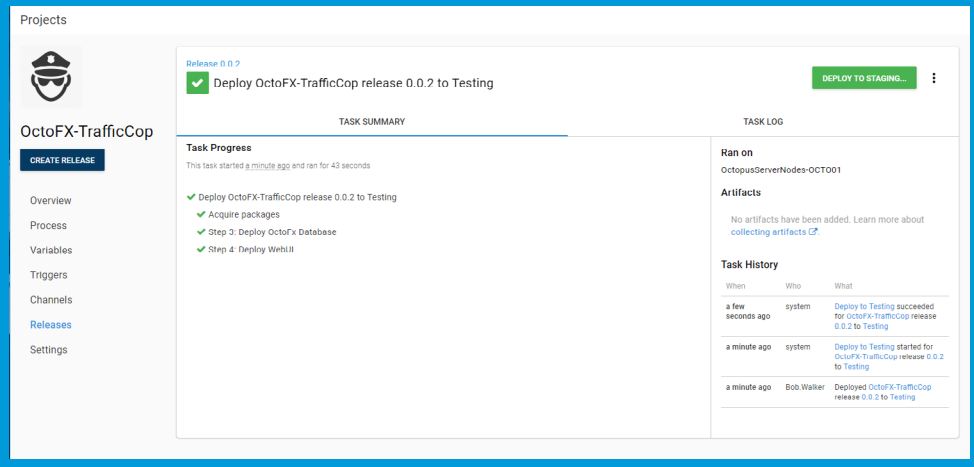
One thing you will notice the project will first try to create a release version of `0.0.1` but the projects it is deploying are currently on release `0.0.2`. We recommend keeping versioning the traffic cop release, so it matches the same, Major.Minor.Patch release. For example, if you have deployed `1.5.2.1` to the WebUI and `1.5.2.10` to the Database, then the first release for the Traffic Cop should be `1.5.2.1`, or `1.5.2-Release1`.



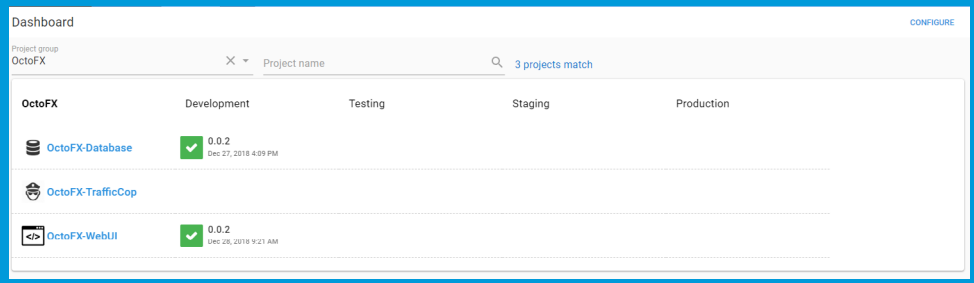
Because the lifecycle skips dev, the release will allow you to go directly to staging. Please note, the version of the projects being deployed is included in the snapshot. If you were to create a new WebUI release or a new Database release, then you would need to create a new TrafficCop release.



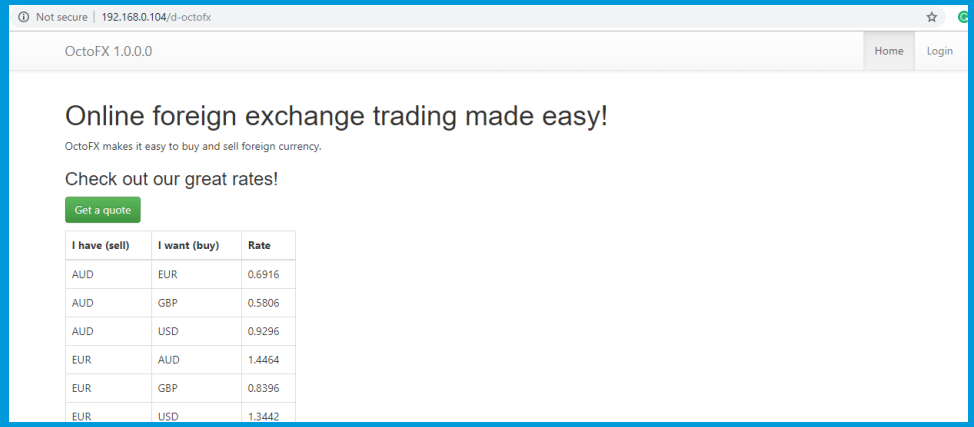
Unlike before, the first release of TrafficCop works because we sorted out any issues in Dev.



Taking a look at the dashboard shows successful results.



Even better, the website successfully loads.



Conclusion

We've finally done it. We finally have Octopus Deploy deploying code to a target. Even better the code successfully loads. The only downside is all the releases we did in this chapter were done through the UI. This is okay for testing purposes. However, we need to configure the build server to automatically create the WebUI and Database releases to add more automation into the CI/CD pipeline.

9

PACKAGING APPLICATIONS ON THE BUILD SERVER

Octopus Deploy will deploy a wide variety of package formats to deployment targets. This list includes, but not limited to NuGet packages, Tar packages, as well as Docker Images, Jar files and Zip files.

The build server, typically Jenkins, TeamCity, TFS/Azure DevOps, Bamboo, or AppVeyor, is most often responsible for packaging applications and pushing that package to Octopus Deploy. It is possible to package a folder without using a build server, but for this book, we are assuming the build server is the one who is packaging the applications.

The build server is responsible for packaging the application is because the build server monitors your source control for any changes. It builds and tests code once a change is detected. Once the build server is done it hands the package off to Octopus Deploy to handle the deployments.

Octopus Deploy is build server agnostic. It does not care about the origin of the packages. All it cares about is that it gets a package to deploy. We have written many plug-ins to support the more popular build servers. For the build servers where we don't have a plug-in, we have created a command line application called `octo.exe`. The plug-ins are wrappers for that command line application so that you will get the same functionality as the plug-ins.

Build Server Process

Our recommended build server process is:

1. Pull down changes
2. Build Code
3. Run Static Analysis
4. Run Unit Tests
5. Package application
6. Push package to Octopus Deploy
7. Create and deploy release in Octopus Deploy
8. Run integration tests

If any of the preceding steps fail then the build server will fail the build. When Octopus Deploy gets a package to deploy we know the code successfully passed analysis and unit tests.

The goal of that list above is to fail fast. If the code cannot be built, there is no reason to run the static analysis. If the static analysis fails, there is no need to run unit tests. This way the build server doesn't waste compute resources on a build.

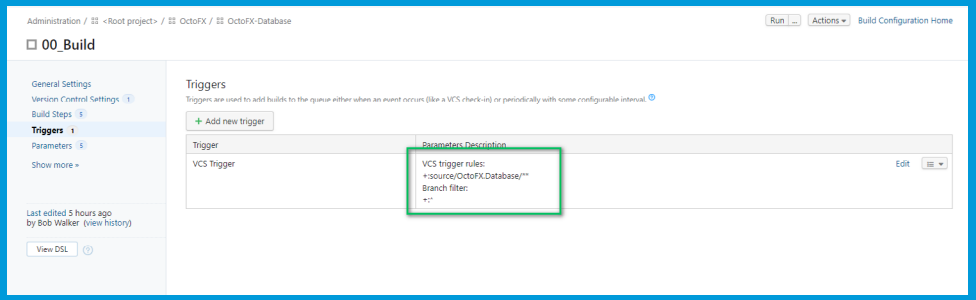
> Analysis and tests get exponentially more expensive as you move farther away from the build server. In terms of time, fixing a problem because analysis failure is much cheaper than fixing a problem in production. Unit tests should be self-contained, repeatable and fast while integration tests are more "soup to nuts" tests which require external components such as a database or file system. They tend to be much slower and more expensive to maintain. We've seen some projects with 10,000+ unit tests and a couple of hundred integration tests. The unit tests took a few minutes to run while the integration tests took 10+ minutes to run. If there is a problem in the code, it should fail as fast as possible.

One Source Control Repository or Multiple Source Control Repositories?

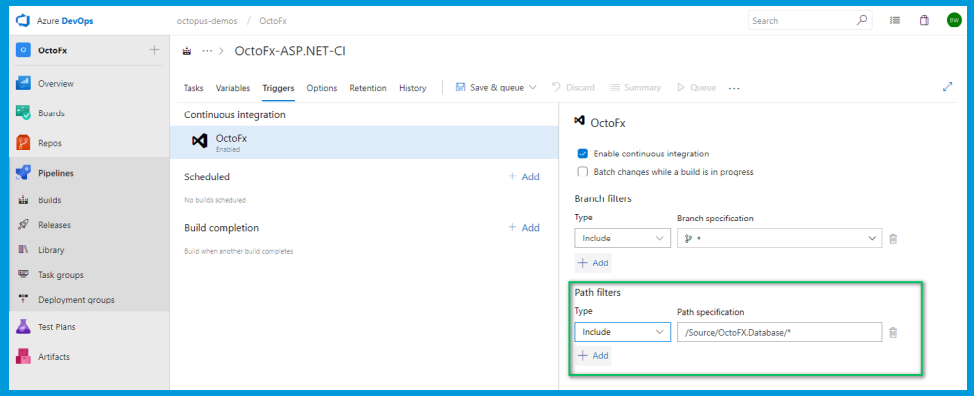
If you recall, the OctoFX project has two components, a database, and a UI. A question a team or even a company will struggle to answer is should all components be in the same source control repository or should there be multiple source control repositories? The answer to that depends on the build server chosen. Several build servers can be configured to watch specific folders. For example, OctoFx stores the database and WebUI in two folders.

- source/OctoFX.Database
- source/OctoFX.TradingWebsite

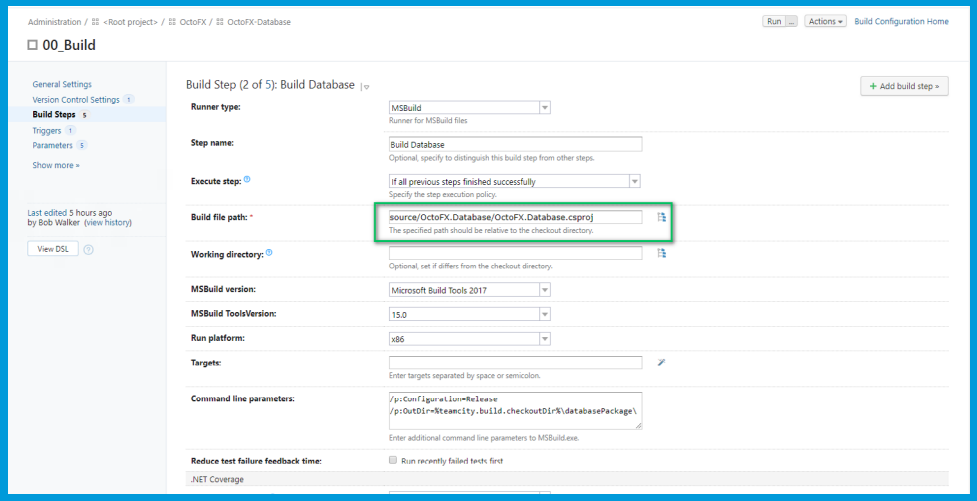
It is possible to configure a build server to have two builds, one to watch for changes to the DB folder and another to watch for changes to the src folder. In TeamCity trigger page allows you to create a trigger based on a watch folder.



In TFS/VSTS/Azure DevOps the trigger is located on the same page where you configure which branch to watch.



Both of these projects are in the same solution. The build step for each project only builds that project.



It does seem a bit extreme to separate the components into two builds. However, consider this. How often do you make a change to just the website? What if you make a change to the database, say add a new index? Should you have to wait for the entire application, both the front-end and database, to be built and tested because of a small index change?

> A good rule of thumb is a build shouldn't take any longer than it takes to get a cup of coffee from the break room.

Building and Packaging using Versioning Schemes

The Major, Minor, and Patch, of the version should be stored somewhere in the source control repository or a variable on the build server. You want to provide a source of truth for the version numbers.

> When compiling or building the code those values should be used to version the .dlls or .jar files. Versioning compiled items will allow you to see what version is on a specific server easily.

When packaging the application to ship to Octopus Deploy, you should also include a build number. For example, if you compiled `1.5.2` then you should include the build number, say 1000, in the package name `1.5.2.1000` or `1.5.2-Build1000` depending on your internal versioning guidelines.

> Including the build number with your package allows you to have multiple builds for the same version. Very rarely will a version have a single build which makes it to production.

Releases

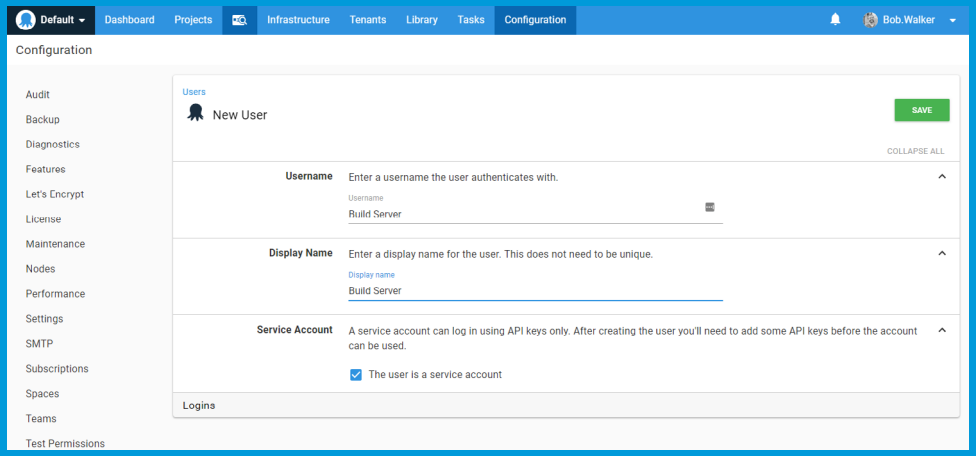
We recommend having the build server create and deploy a release to a Development environment. The build server should control when the code is deployed to that environment. Maybe your project needs to push two packages. Waiting until after both packages are pushed to create the release ensures Octopus will deploy those two packages at the same time.

All of our plug-ins allow you to wait for the deployment to Development to complete. It is recommended you enable that feature. This way the build will fail if the deployment to development fails. You can also run integration tests once the deployment is complete.

It is possible to configure the Octopus Deploy server to create a release when a package is pushed automatically. However, it only works if a particular set of conditions are met, such as using the internal NuGet feed, not using variables for package ids, and so on. As more and more people use your Octopus Deploy instance within your company, you will often find those conditions very constricting. Automatic release creation using Octopus Deploy should be treated as an exception rather than a rule.

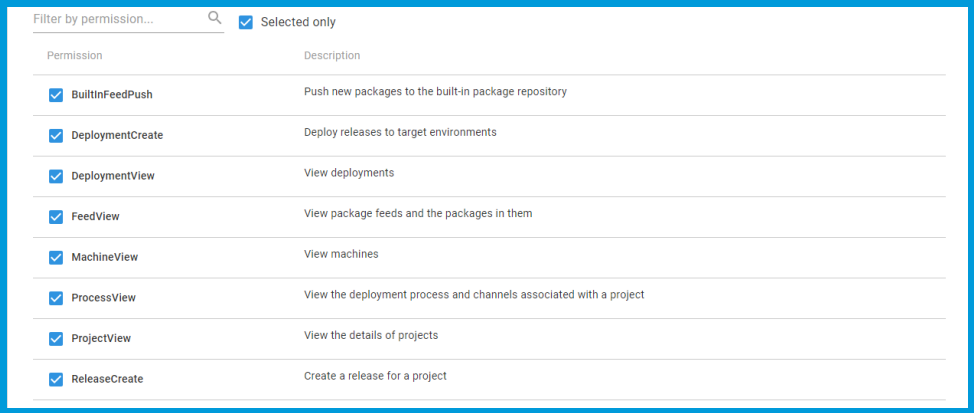
Build Server Permissions

Your build server will need to communicate with the Octopus Deploy server. To do that it will need to use an API Key of a user in Octopus Deploy. The user should be a service account and not a user account in Octopus Deploy. The service account property can only be set when a user is first created. You need to click the The user is a service account checkbox.

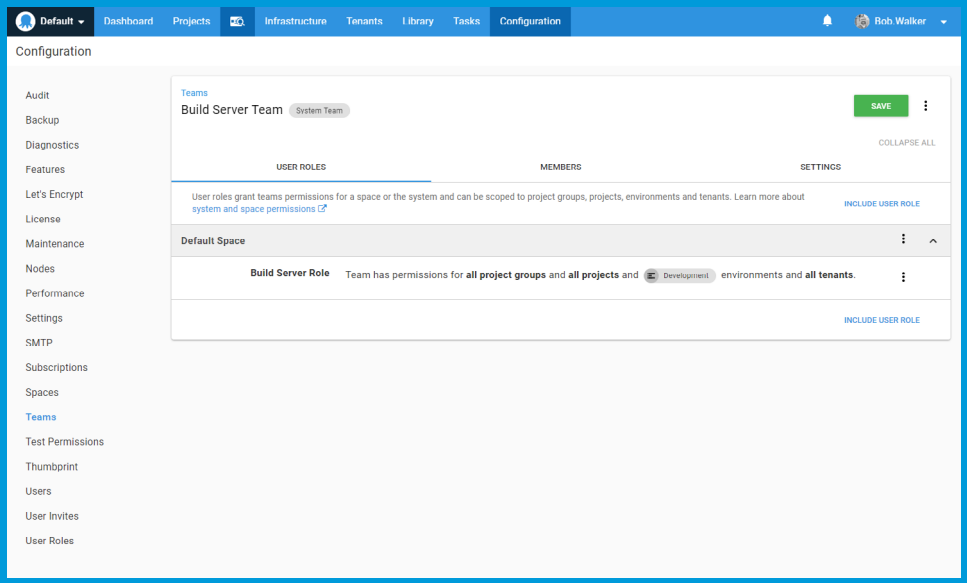


Service accounts don't have passwords. They only have API Keys. If an API Key is compromised, then all you need to do is delete it. Password resets are slightly harder. Also, service accounts are not tied to a specific user. Often we have seen a person configure the CI/CD pipeline using an API Key linked to their user account. When they leave the company, the typical clean-up process runs and disables the user account. Then all the builds start failing because they are unable to deploy.

It is also a good idea to limit what the service account can do. We recommend creating a custom role for your build server service account which we will call **`Build Server Role.`**



Create a new team and assign the user and the role to that team. You can also limit the deployment environments. That also helps lock down what that service account can do.



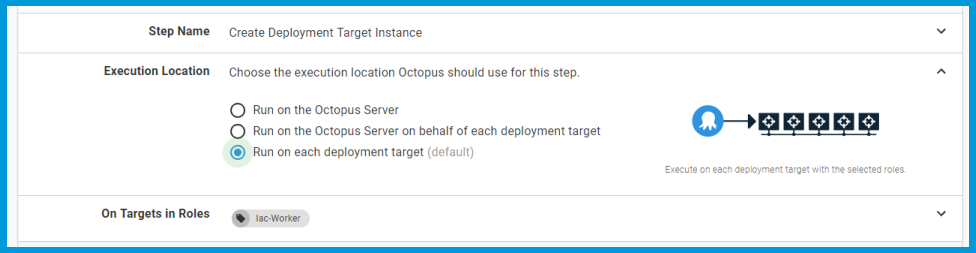
Conclusion

By integrating a build server into the mix, we now have an entire CI/CD pipeline. If you have any takeaway from this chapter, it is you should build your application once and then deploy those same build artifacts across all your servers. The code which is tested in QA should be the same code which is going to run in production.

10

OFFLOADING WORK ONTO WORKERS

When you add a **`Run a Script`** step, you are asked to define where the script will be run. If the script is to do some clean-up work after deploying to IIS, it makes sense to run it on a deployment target. That is pretty clear. However, what about other scenarios, say sending a notification to slack. It doesn't make a whole lot of sense to run that on a deployment target.



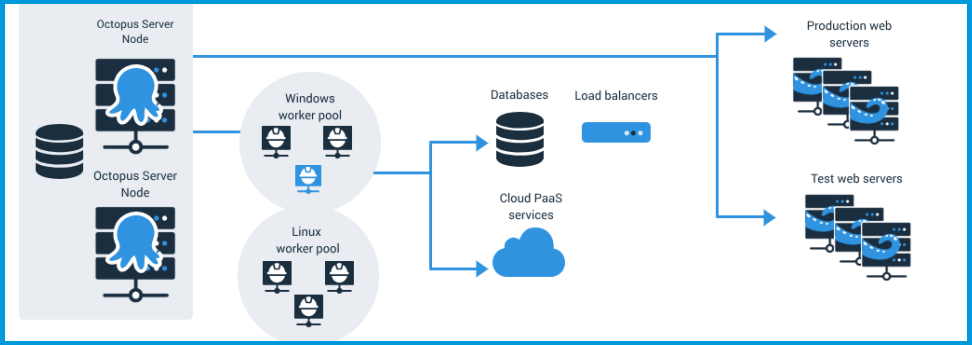
When Octopus Deploy was first created the number of times you would run a script on a server was limited. Send a slack notification, create a folder name based on some business logic, or notify NewRelic of a deployment. That work is easy for the server to do. It doesn't consume resources nor does it take much time.

With the addition of Platform as a Service options (PaaS), the Octopus Server is being asked to do quite a bit more. An Azure App Service isn't the same as a VM. It is not a server where you can download and extract a package. The deployments all occur over an API. Anytime you want to deploy an Azure ARM Template, or to a Kubernetes Cluster, you would run that directly on the Octopus Server.

It is entirely possible to have a deployment process run on the Octopus Deploy server. Running everything on the Octopus Deploy server is not a terrible thing when you have one or two projects. It doesn't scale all that well when you have several hundred or thousands of projects.

On top of that, these scripts are running on the Octopus Server. We like to think most people are good at heart and don't mean any ill-will. However, people make mistakes. A missing quote here. A missing quote there. Before you know it the server hosting Octopus Deploy crashes because of a lousy PowerShell script.

Those scenarios are why workers were created. They allow you to move that work from the Octopus Deploy server onto other machines. How it works is that you create a worker pool and put machines in that pool. When work needs to be done the machine is leased from the pool, the work is done on that machine, and the machine is placed back into the pool.



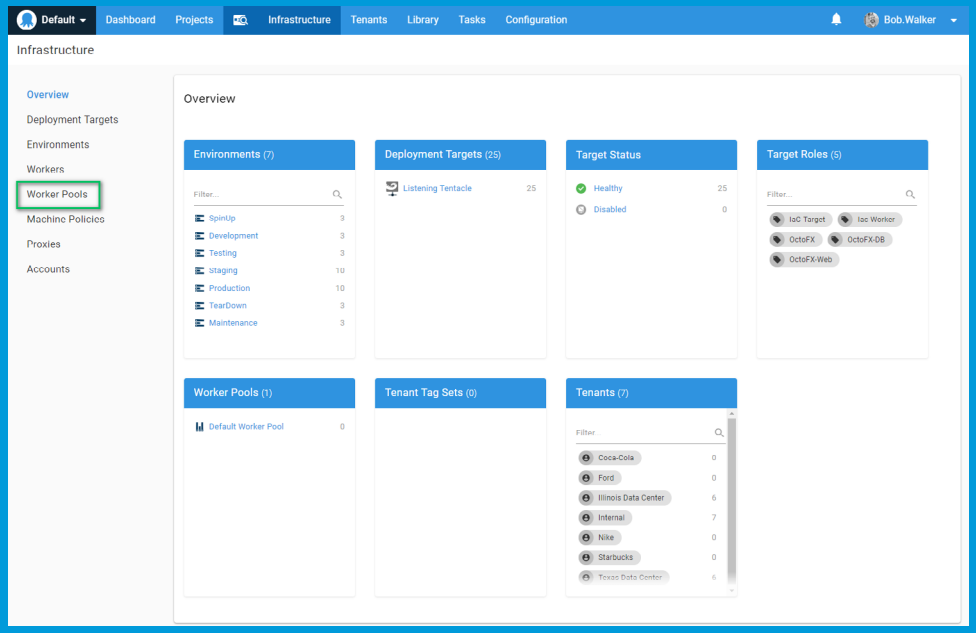
In this chapter, we will configure worker pools. While we are doing that we will provide recommendations to help secure your Octopus Server and scale your worker pools.

Worker Pools

In the diagram above there are two worker pools, Windows and Linux. That is a good start. However, we recommend creating a worker pool for the type of work being done. If you are doing Kubernetes deployments, Azure deployments, and database deployments, then we would recommend three worker pools, one for Kubernetes, another for Azure and another for database deployments.

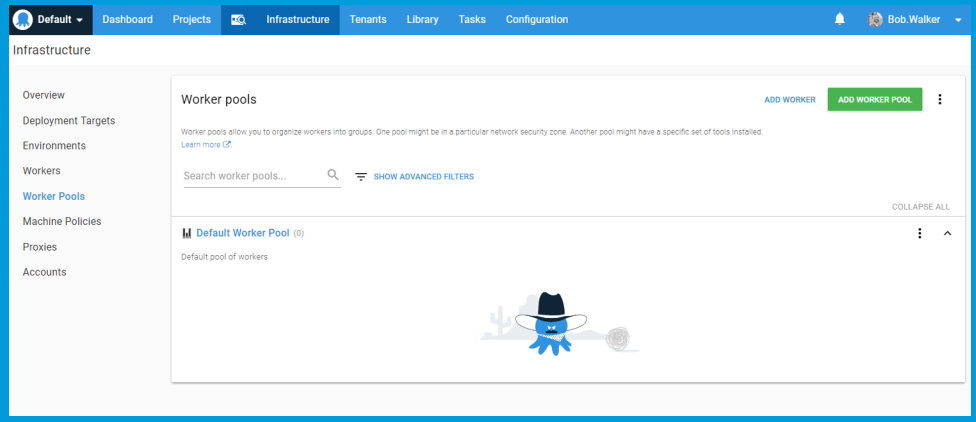
Breaking apart the worker pools by deployment type provides many benefits. The servers in each pool only have the be configured to handle the deployment type. For Kubernetes deployments this means just having to install KubeCtl. While the Azure deployments would only need the Azure CLI installed. Less software to install reduces the chance of any software conflicts. You can scale up your worker pools based on need. If you are deploying to hundreds of databases, then you could have a half dozen workers in the database worker pool. Meanwhile, if you are only deploying to a couple of Kubernetes clusters, then you might need one or two workers in the pool.

For this demo, we will be adding those three worker pools. To start, you will go to the infrastructure page and click on worker pools.

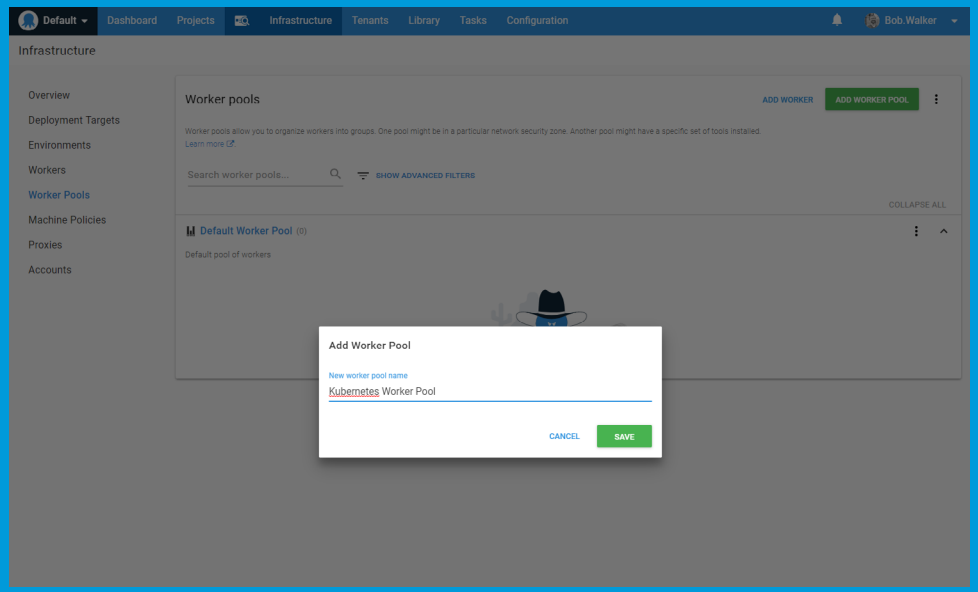


When you come to the worker pool page, you will notice there is already a worker pool, the Default Worker Pool. This worker pool is automatically created when you install Octopus Deploy. When there are no machines in the default worker pool, then the Octopus Server will handle the work.

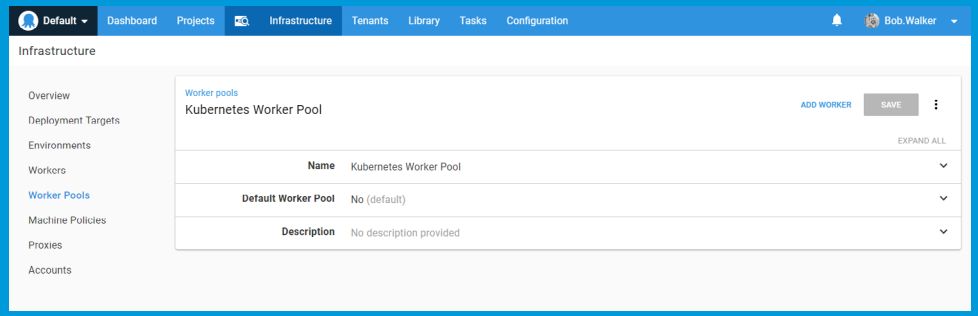
Leave the default worker pool alone when configuring workers and worker pools. Do not add any new machines to the default worker pool. Leaving the default pool as is will allow you to keep using Octopus Deploy as is while you create new workers and workers pools. That way you can phase in your rollout of workers rather than making big bang changes.



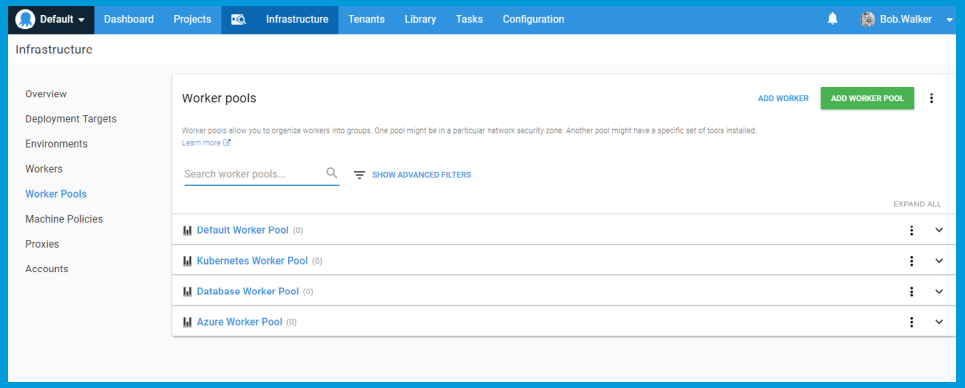
From here we will click on the **`Add Worker Pool`** button in the top right. You will be presented with a modal window where you can enter in the new worker pool name.



When you click save you will be sent to the worker pool form. On this form, you can add a description to the worker pool as well as make it the default worker pool. The worker pool we are adding is for **`Kubernetes`** deployments. It will not be the default worker pool.



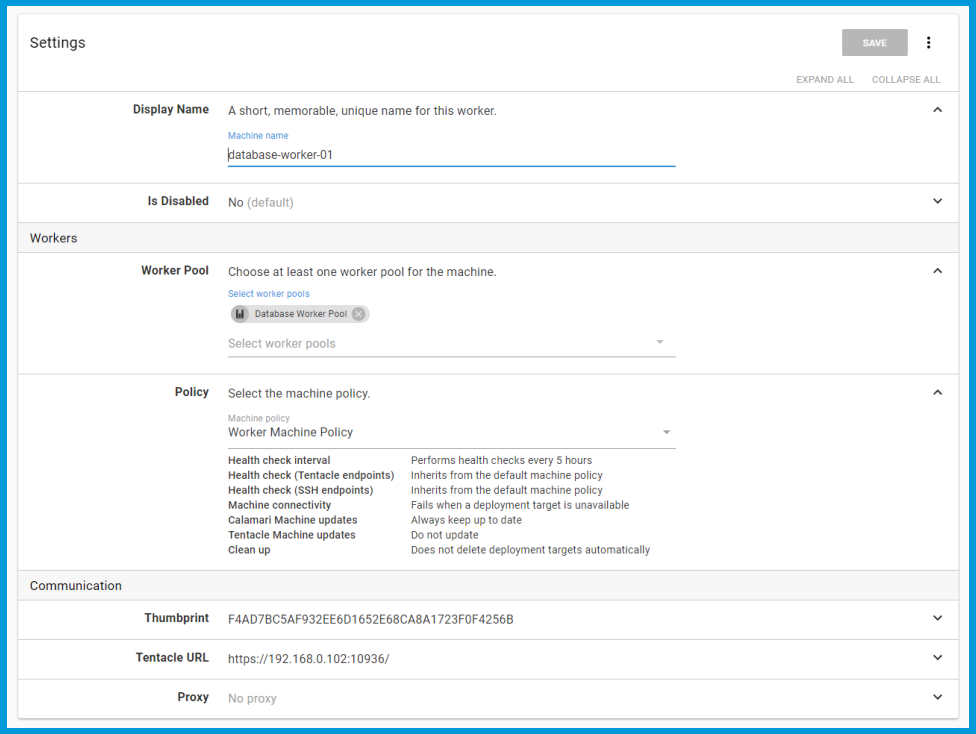
That is it! We have now configured the first worker pool. Go ahead and repeat that process two more times to add in the additional worker pools.



Workers

Adding workers to a worker pool is just like adding a deployment target to an environment. At the time of this writing, only three types of deployment targets are supported with worker pools, polling tentacles, listening tentacles and SSH targets. It is the same tentacle you have always installed on a VM. The primary difference is how it is registered with the Octopus Deploy server.

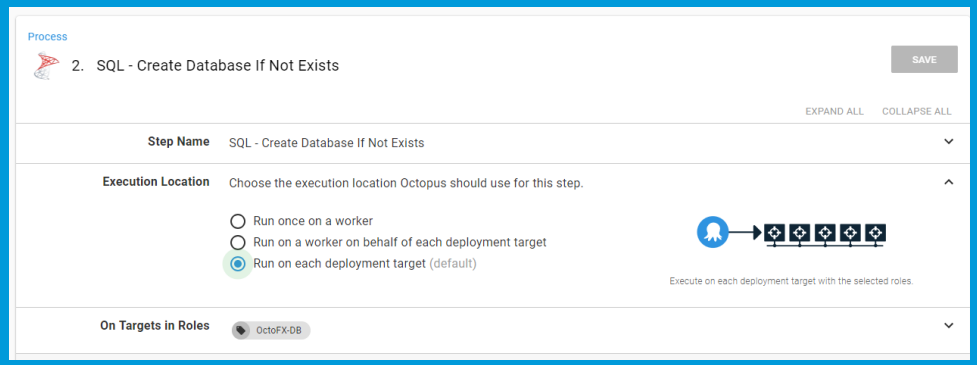
The form to add a worker is very similar adding a tentacle target. The difference in the form is instead of adding the tentacle to an environment and assigning it a role; you attach it to a worker pool.



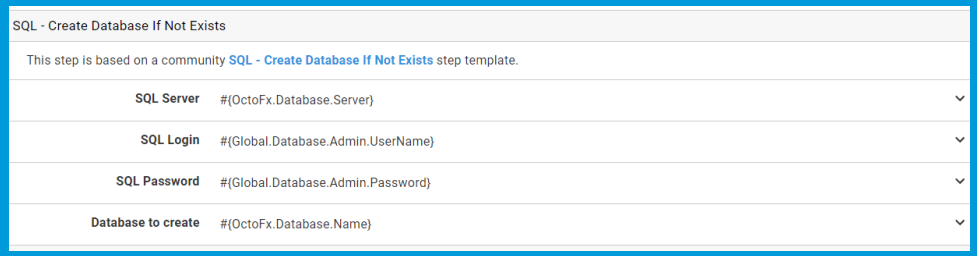
For naming convention, we recommend going with [workertype]-worker-[number]. For example, **`database-worker-01.`**

Changing Project Steps to Use Worker Pools

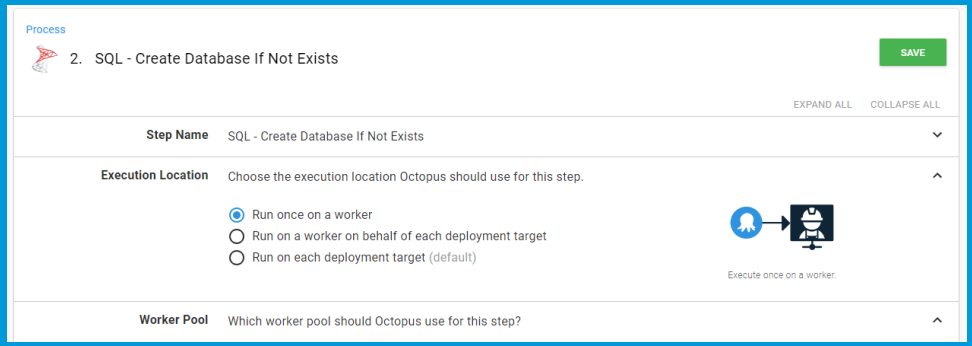
Now that we have added worker pools you will notice the execution location in the UI has changed. The text previously said “Run on the Octopus Server” now says “Run once on a worker.” As well “Run on the Octopus Server on behalf of the deployment target” now reads “Run on a worker on behalf of each deployment target.”



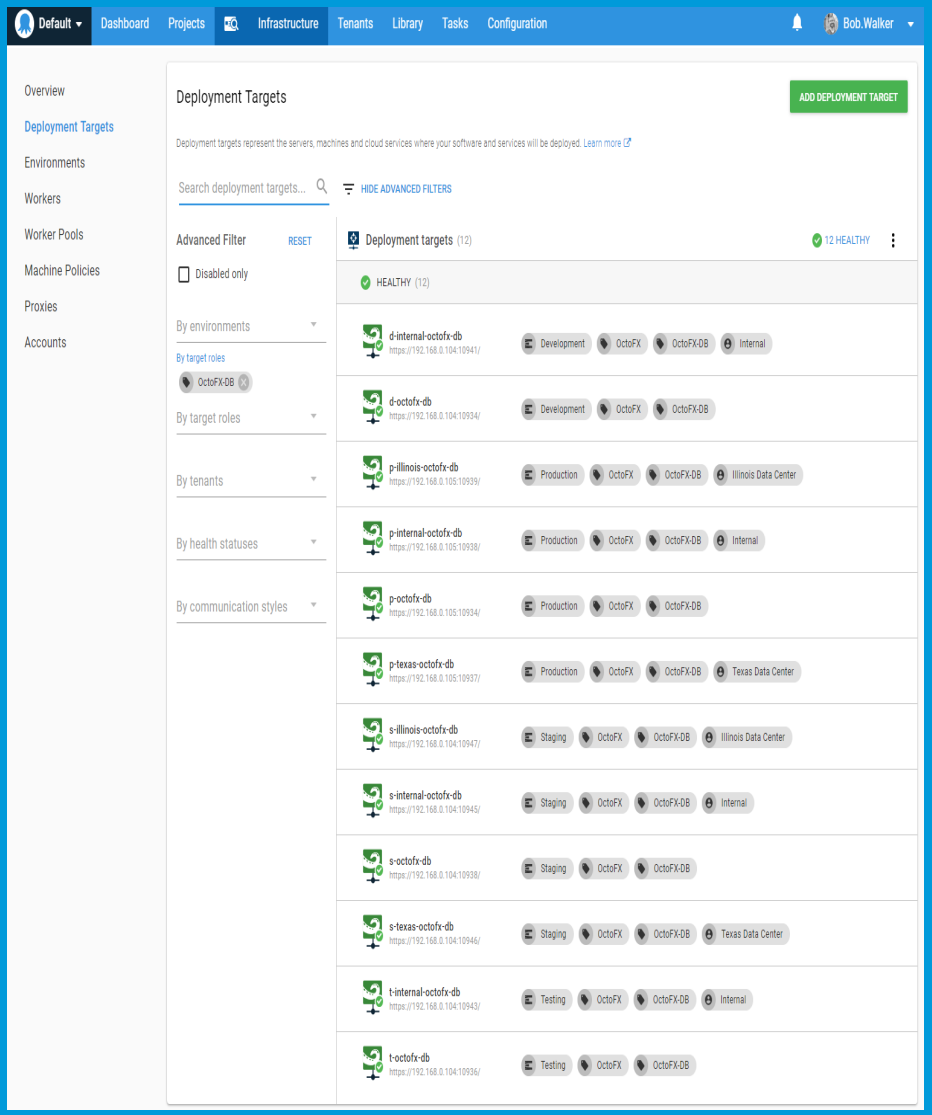
For this demo, we are going to be switching all the database steps over to run on the newly created database worker pool. The reason we are doing this is that the steps are using SQL Authentication, not Windows Authentication, to log into the SQL Server.



To change it, we change the execution location to be “Run once on a worker” and change the worker pool to be “Database Worker Pool.”

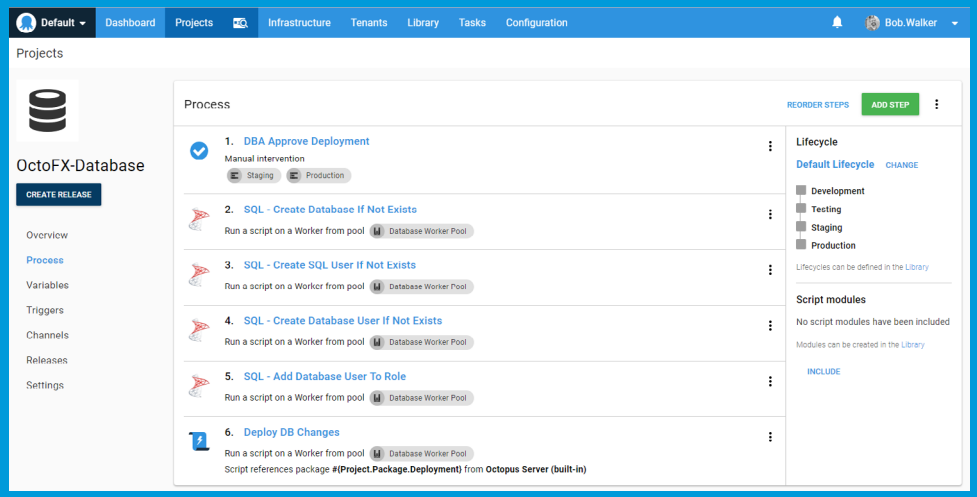


You might be asking yourself, why would I change the database steps over to running on a worker pool? There are several reasons for that. Right now, we have multiple deployment targets to handle my database deployments. When we look at our demo instance, we can see there are 12 targets just for database deployments.

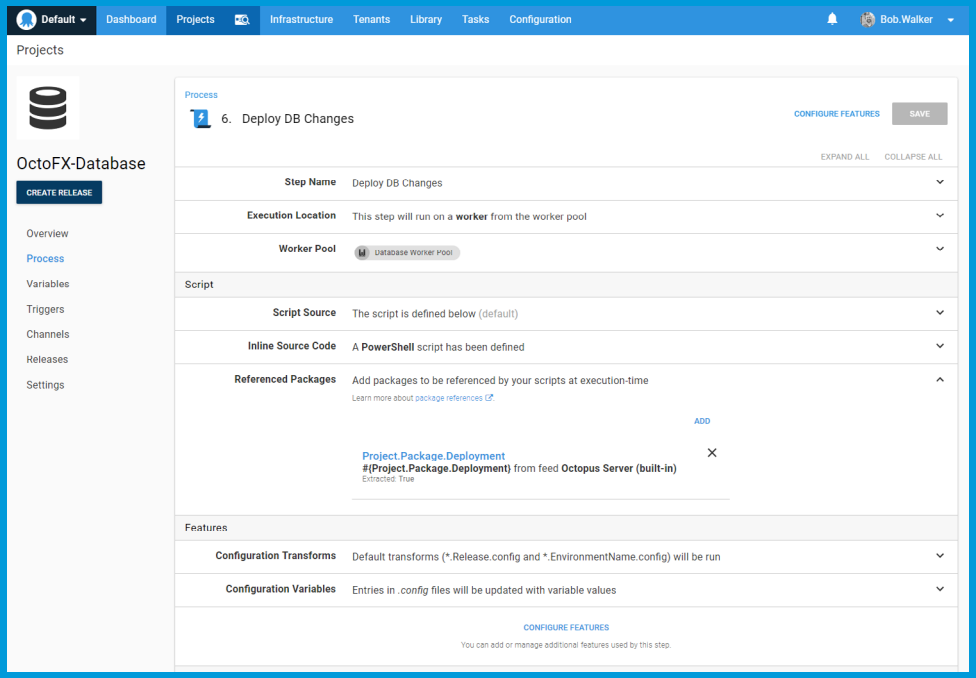


The license we are using for the test instance is limited to 250 machines. So that means those deployment targets consume almost 5% of the license. Besides, database deployments are not manipulating the server like an IIS or Windows Server deployment does. It needs to communicate with a SQL Server using port 1433. Because of that, multiple concurrent deployments can occur at once. Workers allow that.

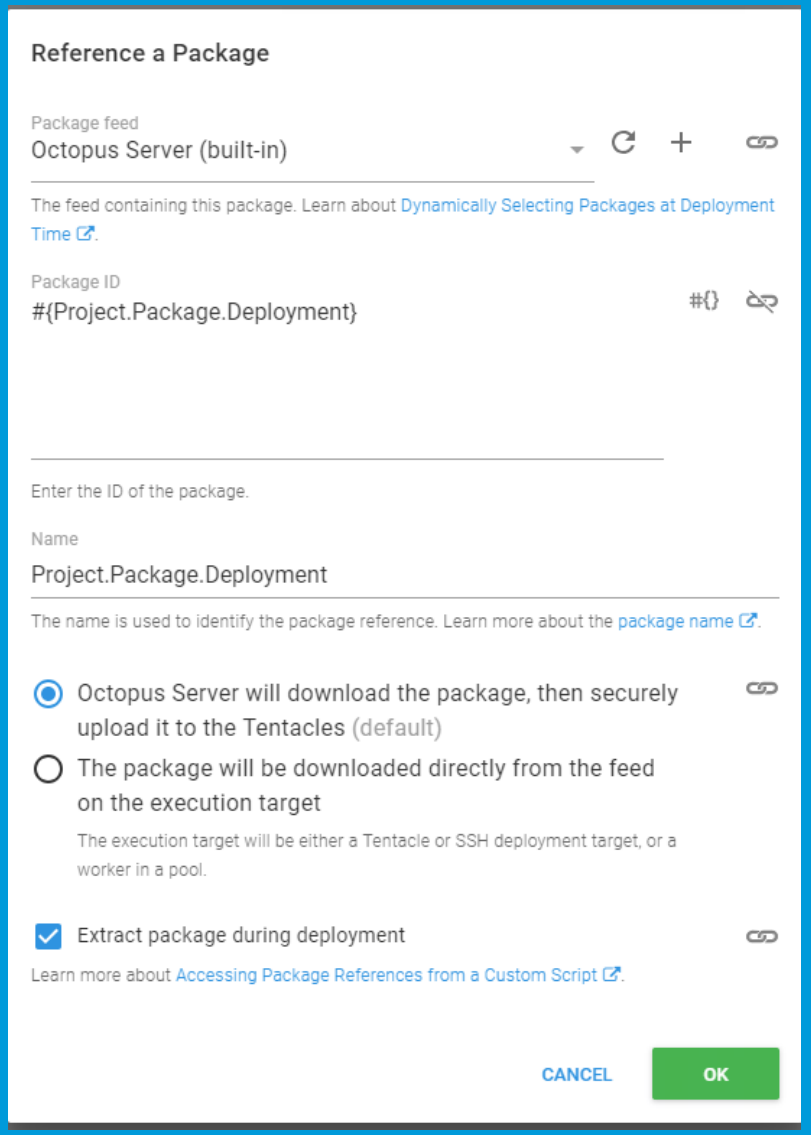
The updated process no longer uses deployment targets. It now uses workers.



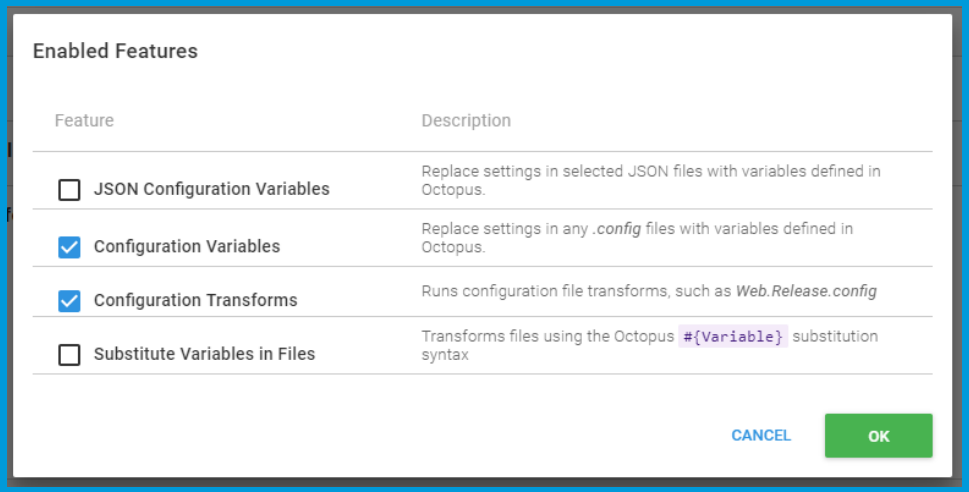
You will notice the last step is running a PowerShell script rather than the deploy a package step. A new feature added in 2018.8.0 was the ability to reference packages in the run a script step.



The reference a package feature allows you to specify a package from a feed. It also allows you to extract the package after downloading it. Typically users don't extract a package if all they want to do is copy it to a file share or something like that.



Because you can reference a package, we also added in the ability to do the configuration transforms.



You can reference the package's extracted path in the script. In this case, we are using DbUp, which means we can execute the console application in the package to do the upgrade.

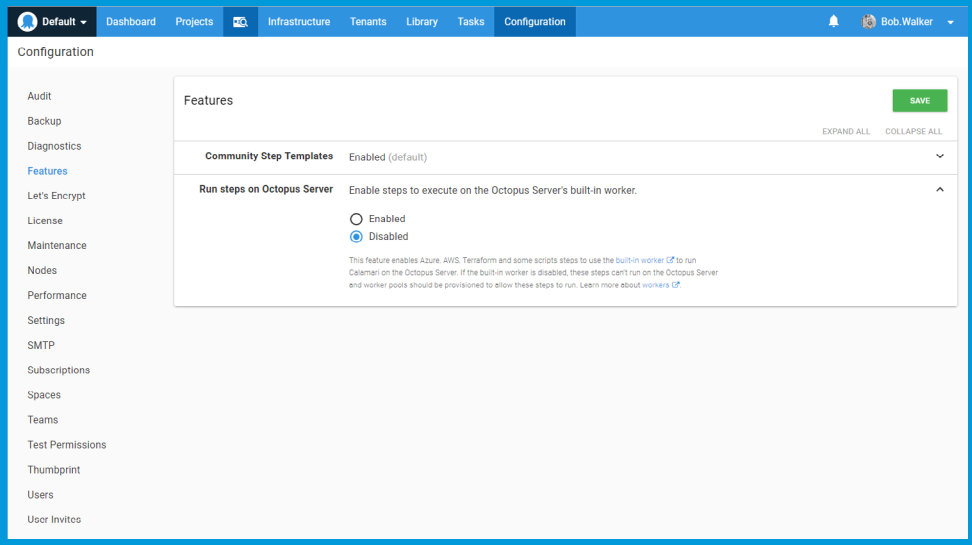
```
$packageExtract = $OctopusParameters["Octopus.Action.Package[Project.Package.Deployment].ExtractedPath"]
& "$packageExtract\OctoFX.Database.exe"

```

With this flexibility, it is now possible to move quite a bit of steps from deployment targets over to workers. The benefit is that they are not counted against your license. This also leaves deployment targets free to do actual deployments to IIS or Windows Services.

Disabling Executing Scripts on Octopus Server

Now that we have our worker pools configured and projects updated, we can now go in and turn off the default worker on the Octopus Server. This will prevent any script from running on the Octopus Server.

> Only do this after you have added worker pools and changed existing projects to use them. This could break many deployments.



Now all scripts have to run on an external worker.

Conclusion

Workers allow you to speed up and scale out your Octopus Deploy server with some minor configuration changes. Workers enable your Octopus Deploy server to focus on orchestration. Before workers, if you wanted to do Kubernetes deployments, you would need to install KubeCtl on the Octopus Server. KubeCtl installs required a restart of the service because KubeCtl adds a value into the PATH variable. By removing Kubernetes deployments from the Octopus Server, we have also eliminated the need to install additional software on the server. That should decrease the likelihood of any downtime.

11

LET'S TALK MULTI-TENANCY

Many of our customers deploy SaaS software. Typically, SaaS software has more than one customer. They don't want their customers to see other customers. They make their application multi-tenant. In our experience there are typically three ways an application is made multi-tenant.

The first approach is code based multi-tenancy. Each user is assigned to a tenant or a company. When a user logs in, they are shown that companies information. There is a single database and a single code base. Every table in the database has a column such as "CustomerId" column, and every query has "where CustomerId=[LoggedInCustomerId]" attached to it. Code based multi-tenancy is the easiest option to deploy and maintain. However, with code-based multi-tenancy the risk for cross tenant contamination is high. A missed where clause makes for a terrible day.

The second approach is a slight twist on the code based multi-tenancy approach. The twist is that each customer gets a separate database. When a user logs in a Customer Id or connection string is attached to their session. When the code needs to go to the database, the connection string is pulled from the database session. Compared to the first option, the database queries are more straightforward. However, there is still a risk of cross tenant contamination because everyone is using the same resources. Deploying the code is easy. Deploying the databases is not as easy as the first approach. Databases need to be deployed to before a code push. With a couple of databases, this isn't much of a concern. A few dozen or a couple of hundred databases? Then it becomes a concern.

The final approach is total isolation; each customer or tenant gets a web site or web servers and a separate database. Compared to the first two options, the code base much more straightforward. No need to worry about looking up a customer's id and maintaining the connection string somehow in memory. Also, the risk of cross tenant contamination is non-existent as each customer gets their own set of resources. The downside is deployments. The deployment process is the same, the only thing which is is the deployment targets and the connection string. Finally, this approach lends itself nicely to being a combination hosted or on-premise solution.

Multi-Tenancy In Octopus Deploy

Octopus Deploy version 3.4 introduced the Multi-Tenancy feature to help support the total isolation approach. The underlying concept of the feature is very similar to the underlying idea of Octopus Deploy. Consistency. The same process used to deploy to Customer A should be the same for Customer B. This is because the code is the same; only the destination is different.

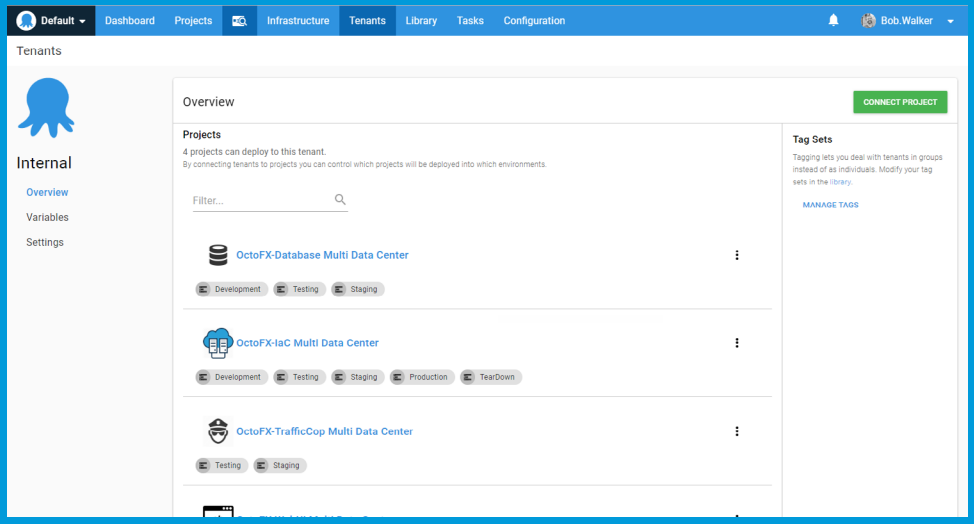
Another essential scenario we support with that model is different versions for each customer. It is very common to see a multi-tenant application deploying different versions to each customer. Perhaps a customer is in the middle of beta testing a new feature. That customer has to be on the latest and greatest or on a pre-release version. While another customer only wants the latest stable release. The total isolation approach makes it easier for this scenario, but with a good deployment strategy, that scenario became challenging to support.

Terms

There are many Terms we use with the Multi-Tenancy feature which are unique to Octopus Deploy.

Tenants

A tenant can be a customer, a feature branch, or a data center. Tenant is the top level object. We didn't want to limit ourselves to a specific one, so we picked tenant as the name.

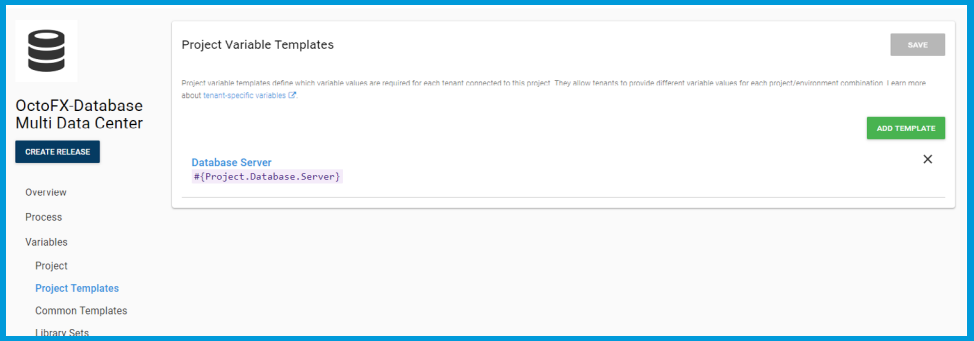


Tag Sets

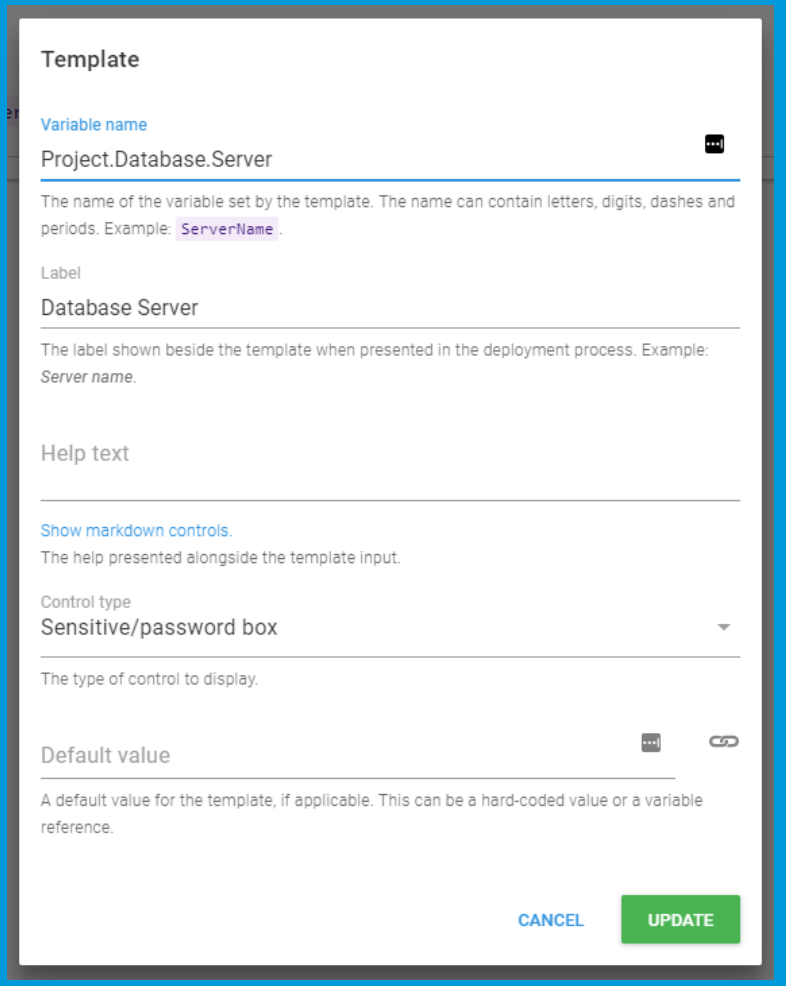
Tag sets allow you to group tenants together. You can deploy all tenants in a tag set as well as tie specific steps to only run on a tag set. We will cover tags set examples in a later chapter.

Project Template Variables

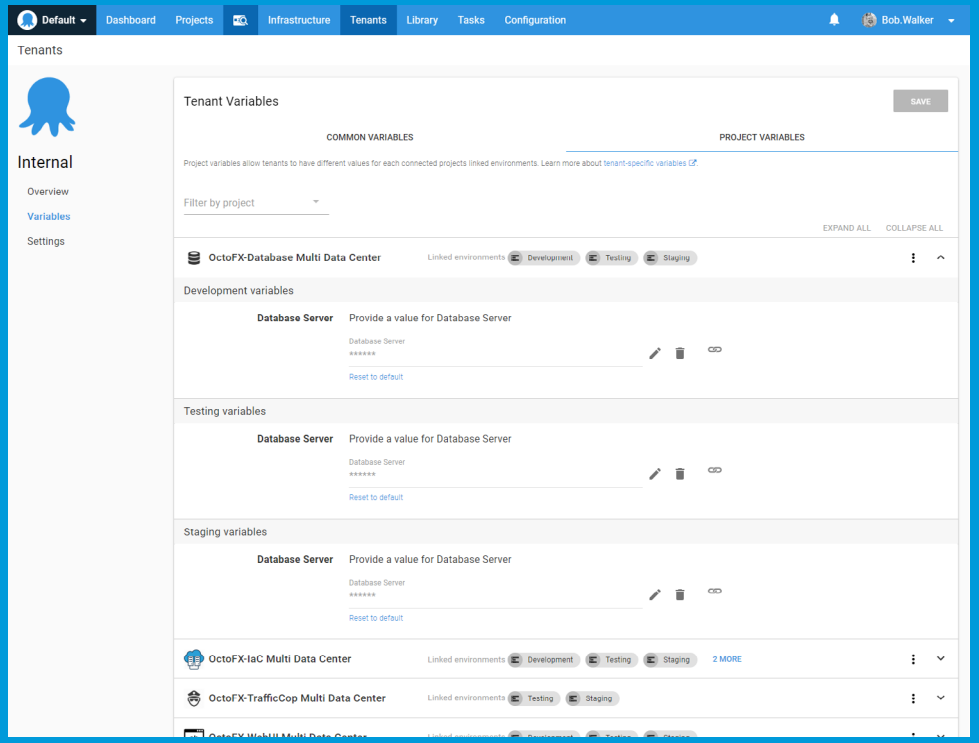
Project template variables allow you to specify a variable which a tenant can change. A perfect example would be a connection string or a database server. With project templates, you define them at the project level.



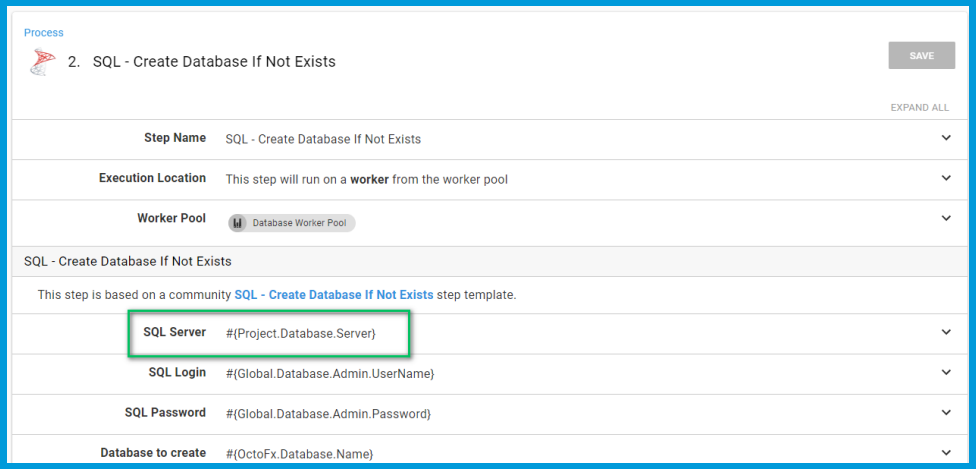
You can specify the variable type for the project template, just like regular variables. You can also provide a default value which the tenant can overwrite.



On the tenant variable screen, you can set those variables.

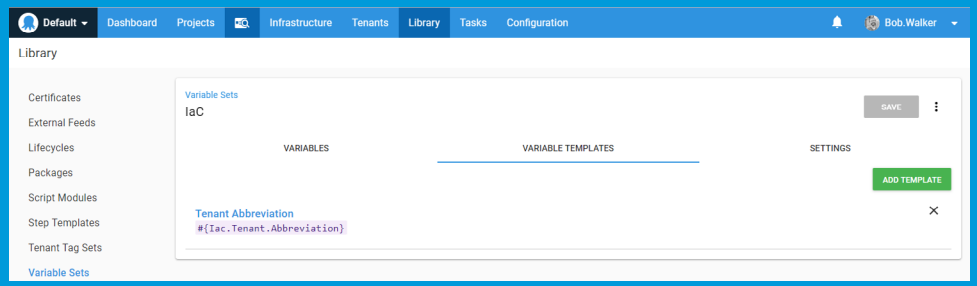


The nice thing about project template variables is that they are treated like any other variable.

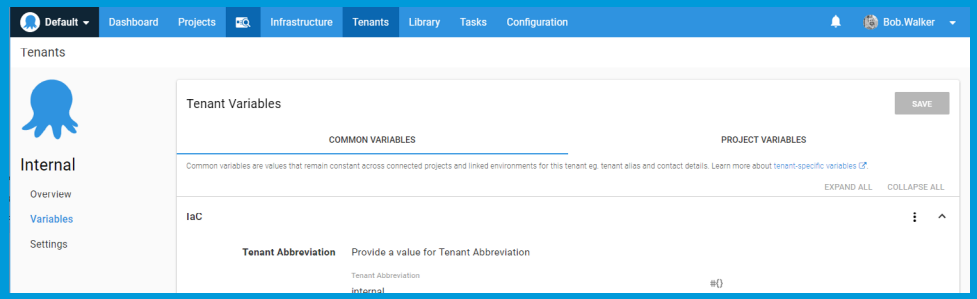


Library Set Variable Templates

Library set variable templates are similar to project template variables. The main difference between the two is that the library set variable templates can be used across multiple projects, and they are not scoped to environments. For example, we needed to define an abbreviation for the tenant to use when creating a target using IaC. We can configure a variable template for the library set.



Just like project template variables, those variables are defined at the tenant level.



Conclusion

In this chapter, we went over what is multi-tenancy. Also, we went through the common approaches taken to make an application multi-tenant. In the next few chapters, we are going to walk through some common uses of the multi-tenancy feature in Octopus Deploy.

12

DEPLOYING TO MULTIPLE CUSTOMERS

In the previous chapter, we walked through what is multi-tenancy as well as some common multi-tenant scenarios. In this chapter, we are going to take an existing set of projects and configure them to deploy to multiple customers.

We will be reusing the OctoFX project we have been using for this entire book. To make the example a little clearer, we have cloned that existing project. In a real-world scenario, you would not clone it but update it.

Scenarios

We want to be able to support the following scenarios for our Multi-Tenant applications.

1. Each of our customers has a separate web server and database. They get the same code. The deployment process should be the same. The only difference is the destination server and some variables such as the database server and user.

2. We should be able to choose when a customer gets a specific version of our code. Some of our customers want to be on the cutting edge, their tolerance for risk is high. They have no problem being the "guinea pig" for a new feature. Some of our other customers have a low-risk tolerance. They only want to be on the stable version.
We should be able to group our customers into different release rings,

3. such as Alpha, Beta and Stable. Different release rings allow us to release new versions of code to multiple customers at once.
Some of our customers have custom branding. During deployments,

4. we need to have the ability to run an additional step to add in the branding.

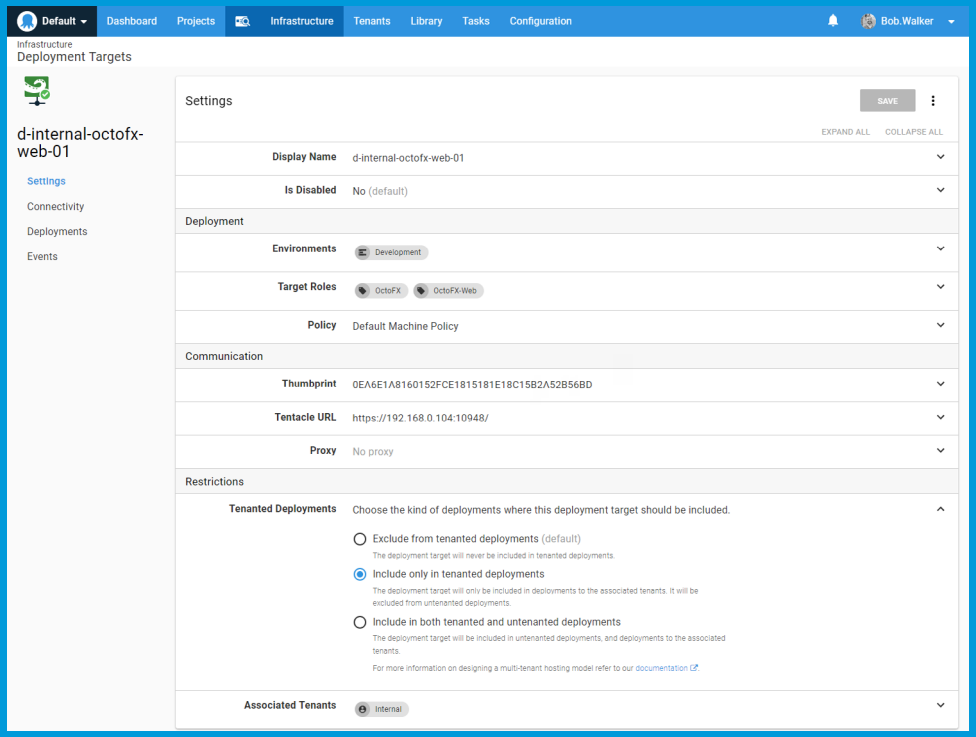
Target Configuration

Before jumping into the projects, we want first to configure our deployment targets. In a previous chapter, we offloaded all database deployments onto workers. We only need to configure the Web Servers for OctoFx. There are a couple of small items to point out when configuring a target for a specific tenant.

The name of the deployment target should include the name of the tenant. [EnvironmentPrefix]-[TenantName]-[AppName]-[Component]-[Number] is a an easy to remember naming convention. For example, **`d-ford-octofx-web-01`** if we wanted the machine to deploy the OctoFX website for Ford to Dev. In the event the target is used for multiple tenants, it is a good idea to group them using tenant tags and name the server after the tenant tag. [EnvironmentPrefix]-[TenantTag]-[AppName]-[Component]-[Number], or **`d-alphacustomers-octofx-web-01`**. As always, naming is easy to understand, difficult to master, as long as you have consensus around your naming conventions you should be fine.

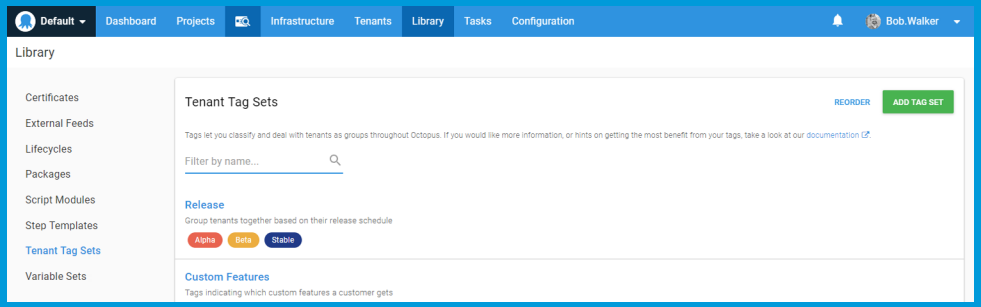
A good rule of thumb is to configure targets for tenanted deployments only. Allowing untenanted deployments for a target should be an exception. Typically we see our customers use a target for tenanted and untenanted deployments in lower environments when the number of test resources available to them is limited.

In a perfect world, each tenant would get a separate web server and database server. In the real world that is not very cost effective. Especially when it comes to hosting VMs on a cloud provider such as AWS or Azure. Don't be afraid to tie multiple tenants to the same machine. Octopus Deploy allows 1 to N number of tenants.

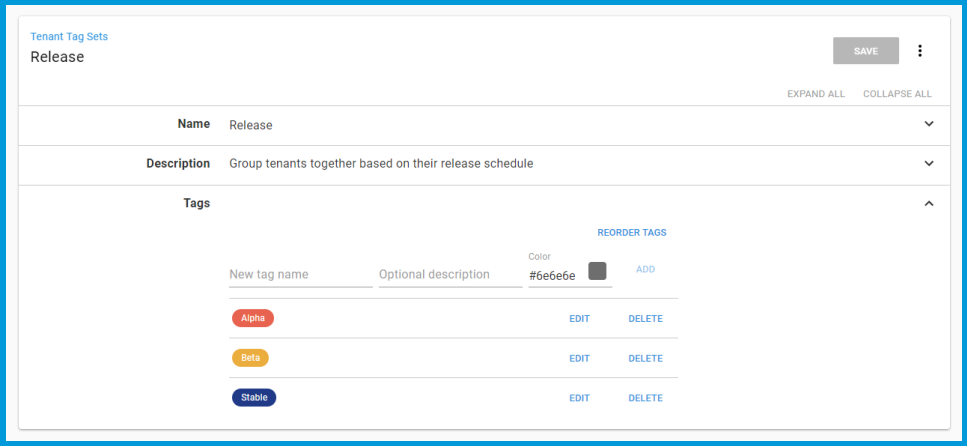


Tenant Tags

Tenant tags are a way to group tenants. To support our scenarios we are going to be creating two tenant tag sets, one called **`Release`** and the other called **`Custom Features`**. Go to the Library -> Tenant Tag Sets page to configure tenant tags.

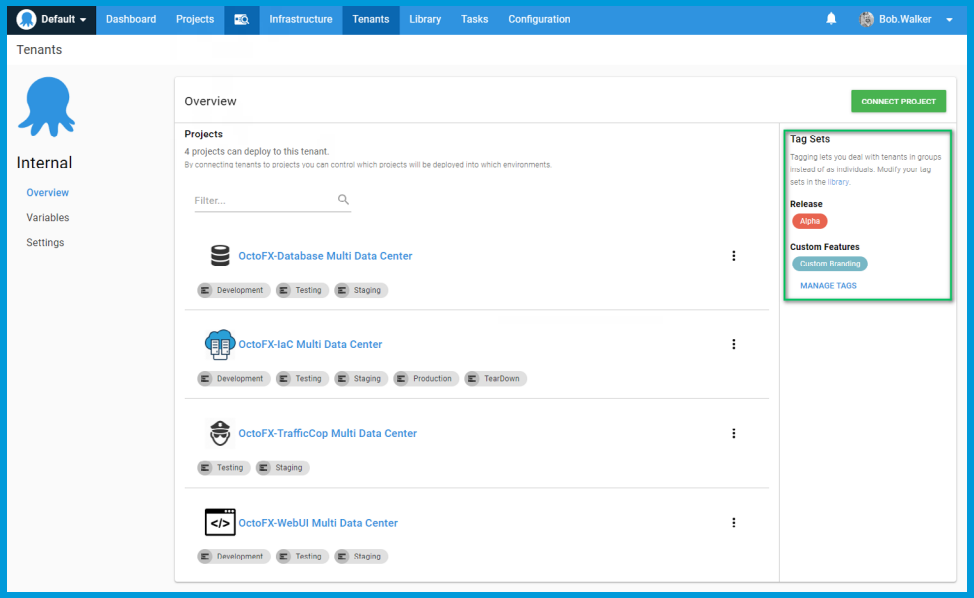


When creating the tenant tags, it is possible to assign each tag a color. We recommend doing this when it makes sense. The **`Release`** tenant tag is a great example. Red for Alpha customers as they are used to risk. Yellow for beta testers as they are okay with some limited risk. Blue for stable customers as they want no risk.

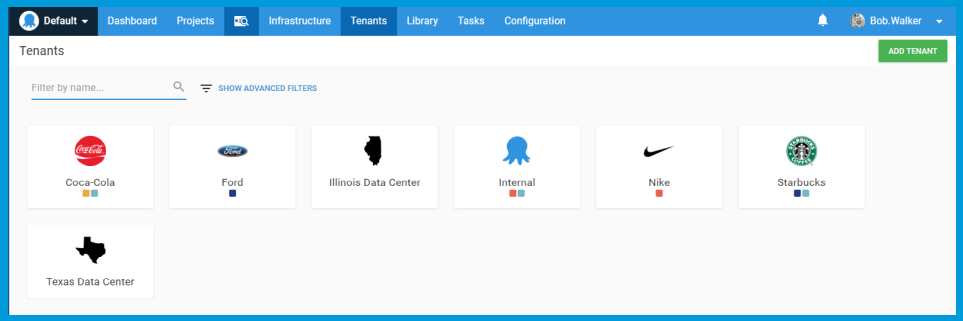


Now we can go back to each of our tenants and assign them the various tenant tags. Our tenant examples are:

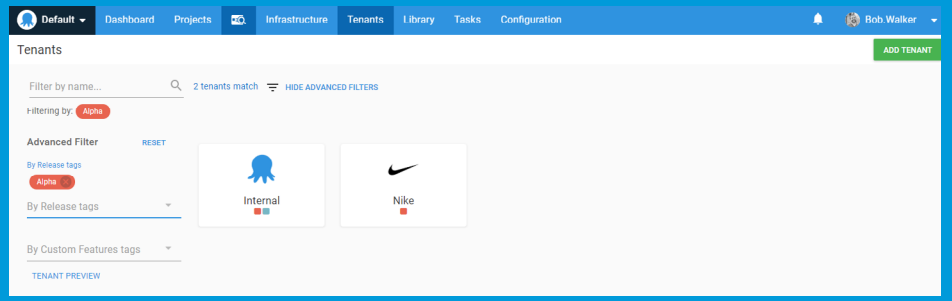
1. Internal - Alpha Release, Custom Branding Custom Features
2. Coca-Cola - Beta Release, Custom Branding Custom Features
3. Nike - Alpha Release, no custom features
4. Ford - Stable Release, no custom features
5. Starbucks - Stable Release, Custom Branding Custom Features



If you are using custom colors for your tenant tags, the tenant overview screen provides a quick overview of tenant tag assignment.



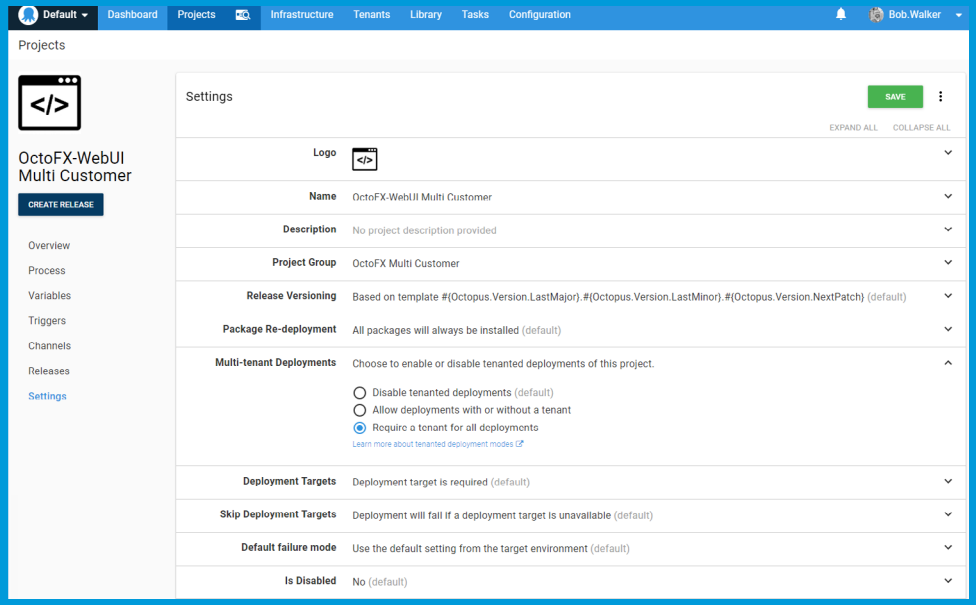
You can also apply filters based on Tenant Tags on the tenant screen.



Project Configuration

Now that we have that scaffolding out of the way, we can now move onto the project configuration. First, we need to go to each project we want to make multi-tenant and enable multi-tenant deployments. We select the option Require a tenant for all deployments. As we go through this configuration, we will make use of tenant-specific variables. Requiring a tenant for all deployments will ensure consistency across your deployments. Deployments without tenants and deployments with tenants only leads to confusion.

> Many times our customers want to enable allowing deployments with or without a tenant because they want to test internally first. Creating a set of internal tenants will accomplish the same goal while keeping your deployments consistent.



Next, we want to go to each tenant and link them to the projects. For this demo, we will be using the following scenarios for the customers.

1. Internal - This is our internal testing customer. They are in the alpha release group because they are always testing new features. Because they are an internal testing customer, they have access to all environments.

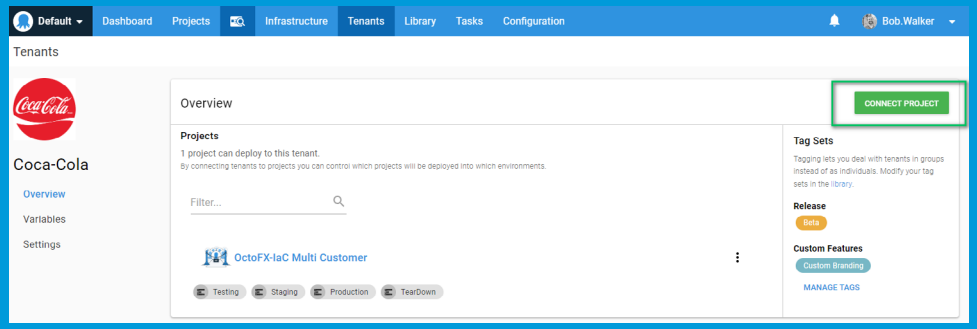
2. Coca-Cola - This is a customer who wants to try out new features, but they don't want to be on the bleeding edge. They are beta testers. Due to the increased risk of being a beta tester, they have machines in testing, staging, and production.

3. Nike - this is our customer who always wants to be on the cutting edge. They are comfortable with the risk. To help mitigate the risk, they have been given resources in the testing environment as well as staging and production.

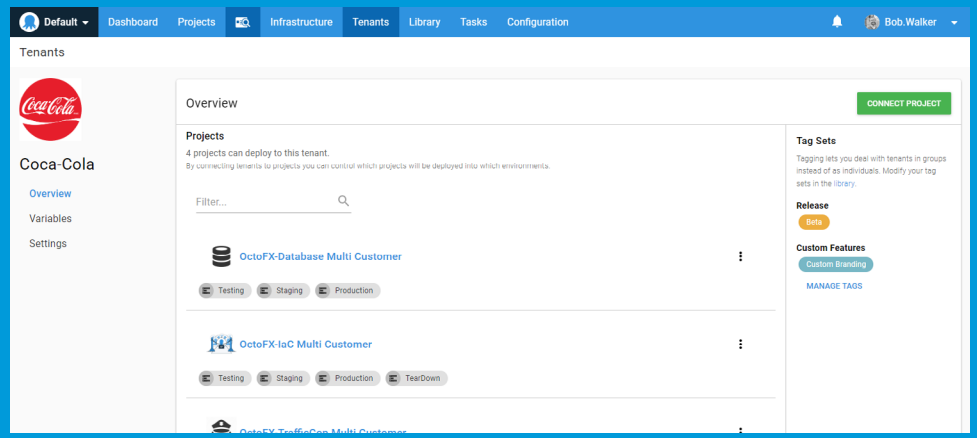
4. Ford - this is our most risk-averse customer. They only want the most stable releases. They want to test all releases before going to production. They have access to staging and production.

5. Starbucks - this customer doesn't have much tolerance for risk. Sometimes they want to test changes before going to production. Sometimes they don't. They have access to staging and production.

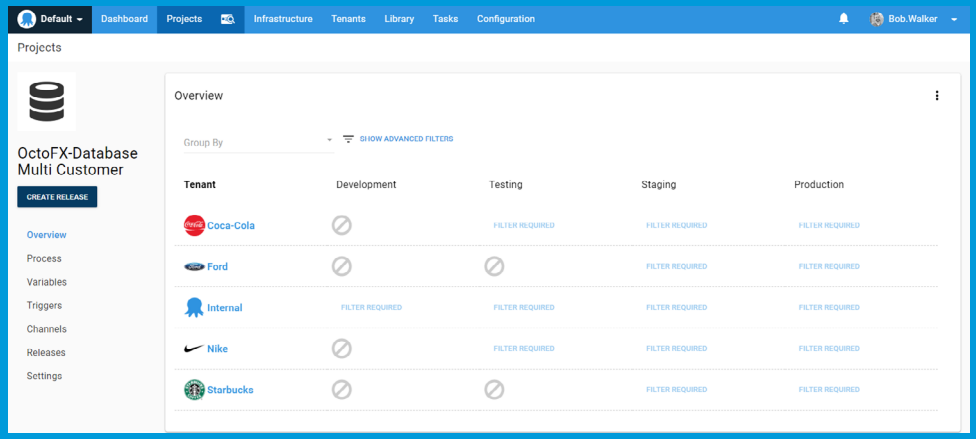
We link up these customers to their projects by going to the tenants screen and clicking the **`connect project`** button.



After connecting up Coca-Cola to our projects, the screen will look like something like this:

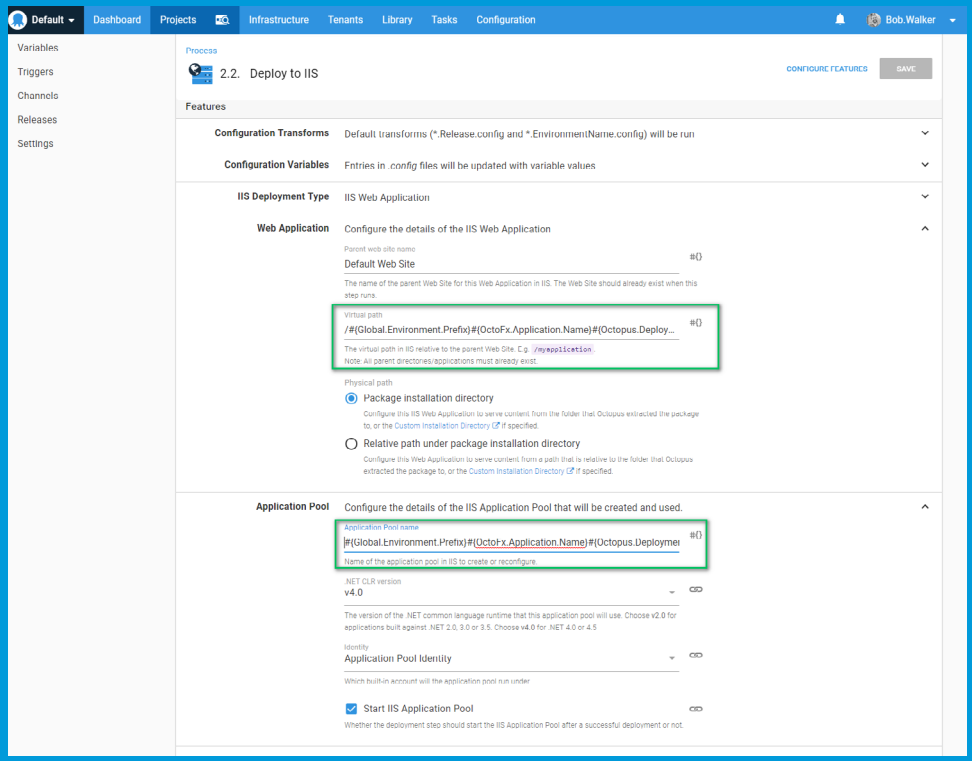


Now we need to repeat for the remaining four customers. Based on the scenarios described earlier, the project overview screen should look something like the one below. The text which says “filter required” indicates the project can deploy to that tenant for that environments.

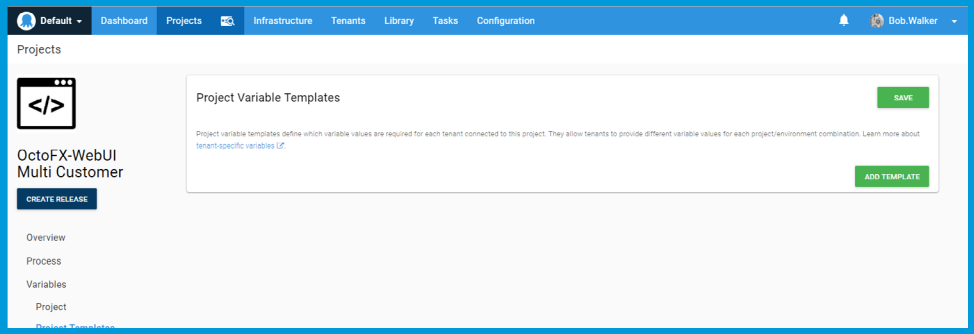


Project Variable Templates

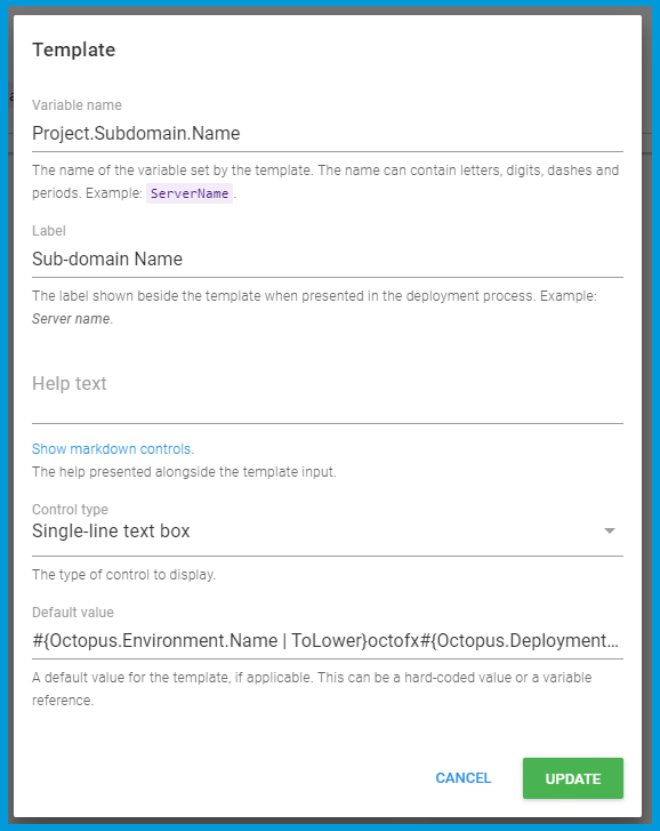
For a moment, let us jump over to the WebUI project for OctoFX. Diving into the step which deploys to IIS we can see that every tenant, at the moment, will be deploying to the same web application. That...is not ideal.



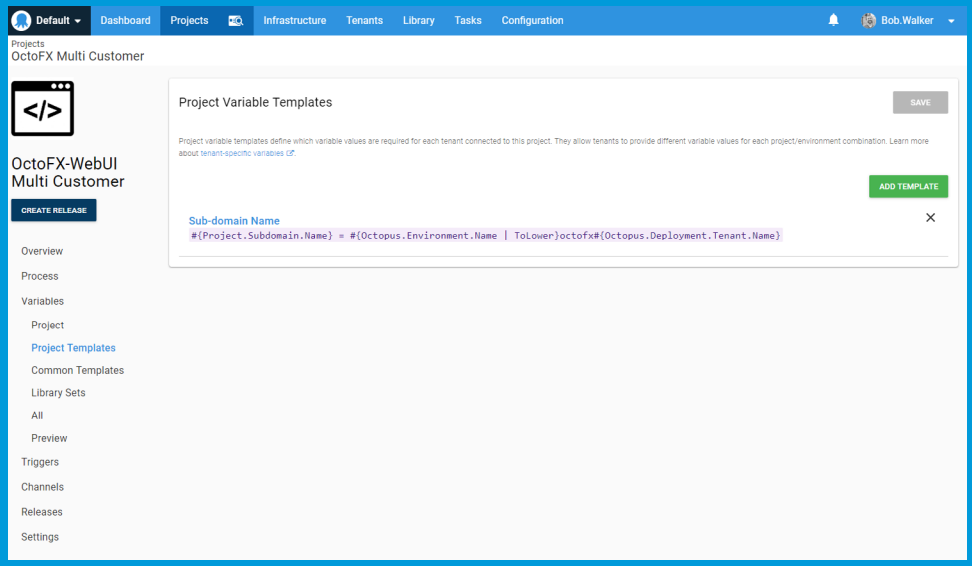
A far better solution is the customer gets their a custom sub-domain on their own set of servers. Majority of the time, we can set a default value, but in some cases, the tenant should have the ability to override that default value. Project Template variables are the feature added to support this scenario. You get to this screen by going to the variables section and selecting the project templates option on the left.



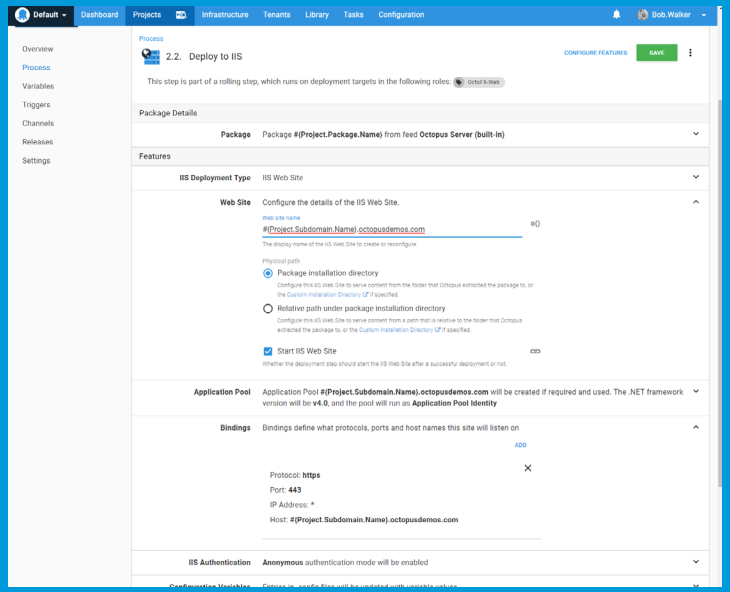
From here we are going to create a new variable to store the sub-domain name. We are going to use the tenant name and the application name as a default value for this variable.



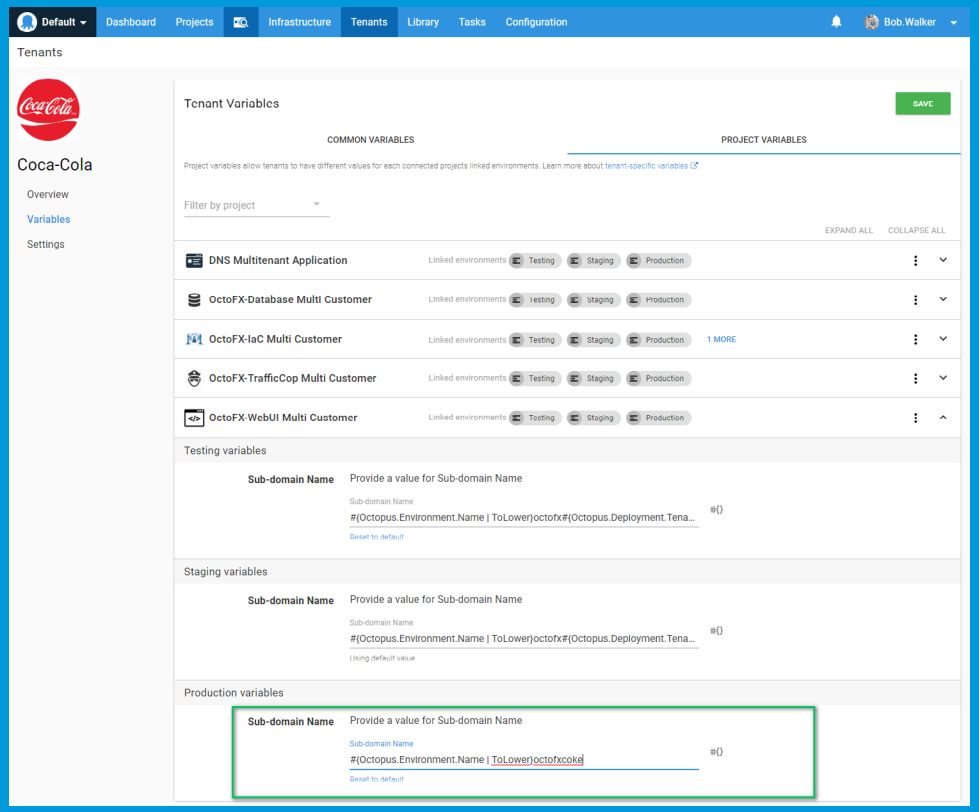
After saving the variable, we can see what it ends up looking like.



That variable can now be used by the Deploy to IIS step.



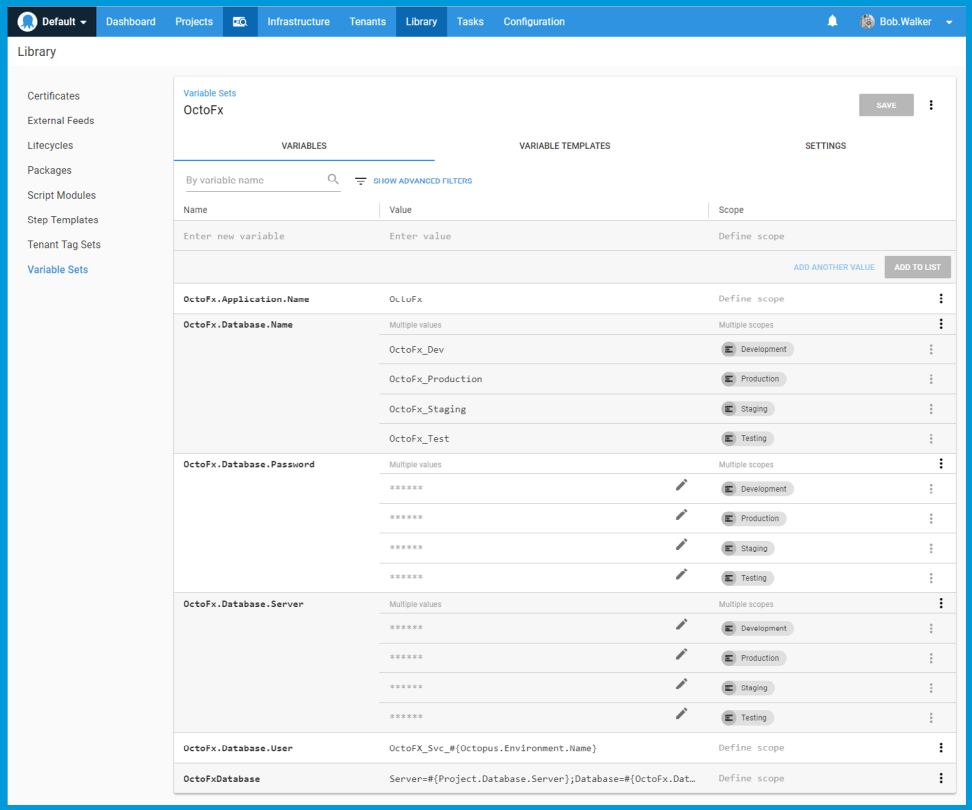
For the majority of the tenants, the default value is perfectly fine. However, in other cases, we want to overwrite that variable. For example, maybe in production, we want to call the domain name for Coca-Cola **`productionoctofxcoke.octopusdemos.com`** instead of **`productionoctofxcoca-cola.octopusdemos.com.`** By going back to tenant screen and changing the variables we can.



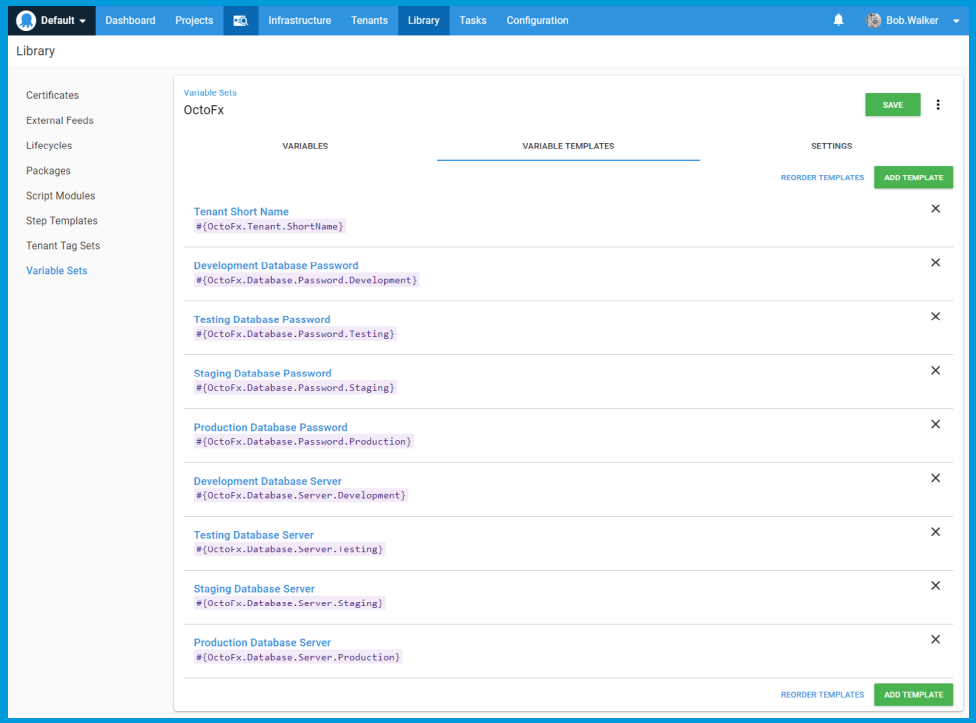
Tenant Specific Variables

Each one of these customers gets their own set of resources, including a SQL Server. If you recall, we have multiple projects, one for the database and the other for the web site. We want to share this information between the web project and the database project. We could go the project variable route, but that would mean duplicating data. The chances of remembering to change both are very slim.

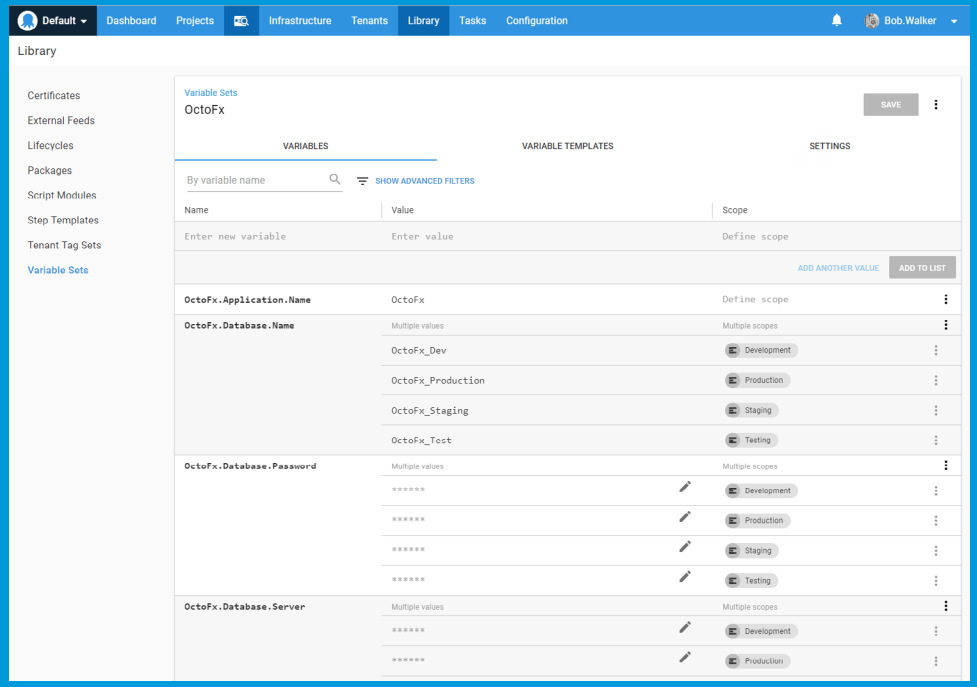
In an earlier chapter, we set up a variable set for the OctoFx application share values between the two projects. We are going to be making some changes to that to support multi-tenancy.



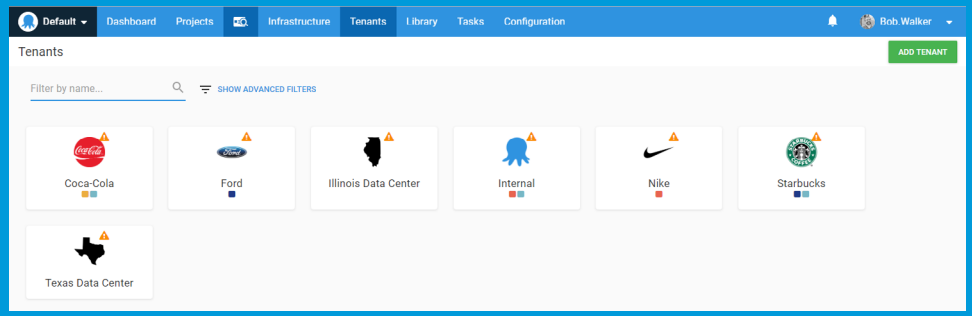
First, we are going to go to the variable template section on this screen and add in variables to store the tenant short name. That will be used to create the database name and the user. Then we will create a template for each environment.



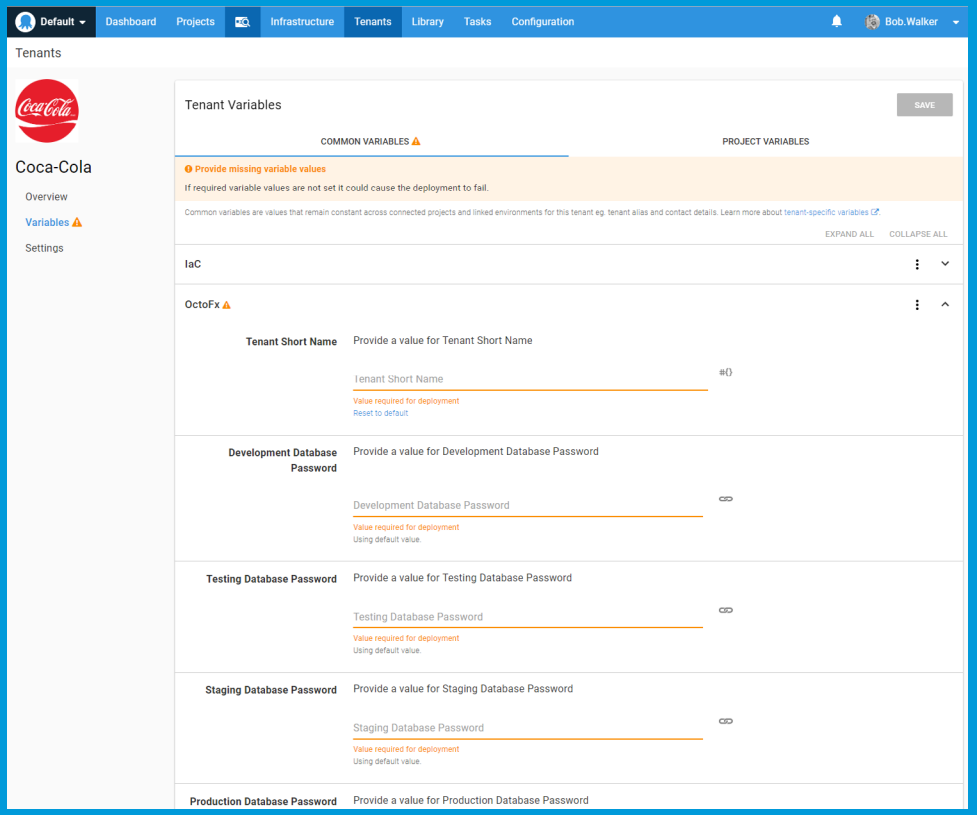
With those values, we can now update the variable set to use them.



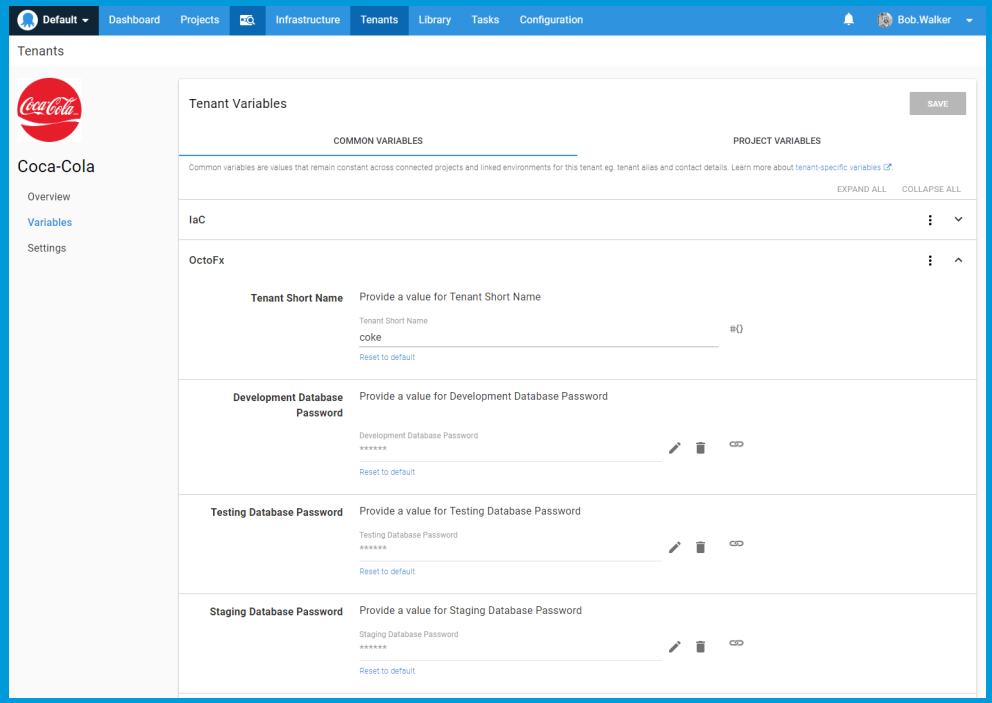
Because we didn't specify a default value for any of those variable templates we see orange icons for each tenant on the tenant screen.



Drilling into the tenant and we see which variables are missing.



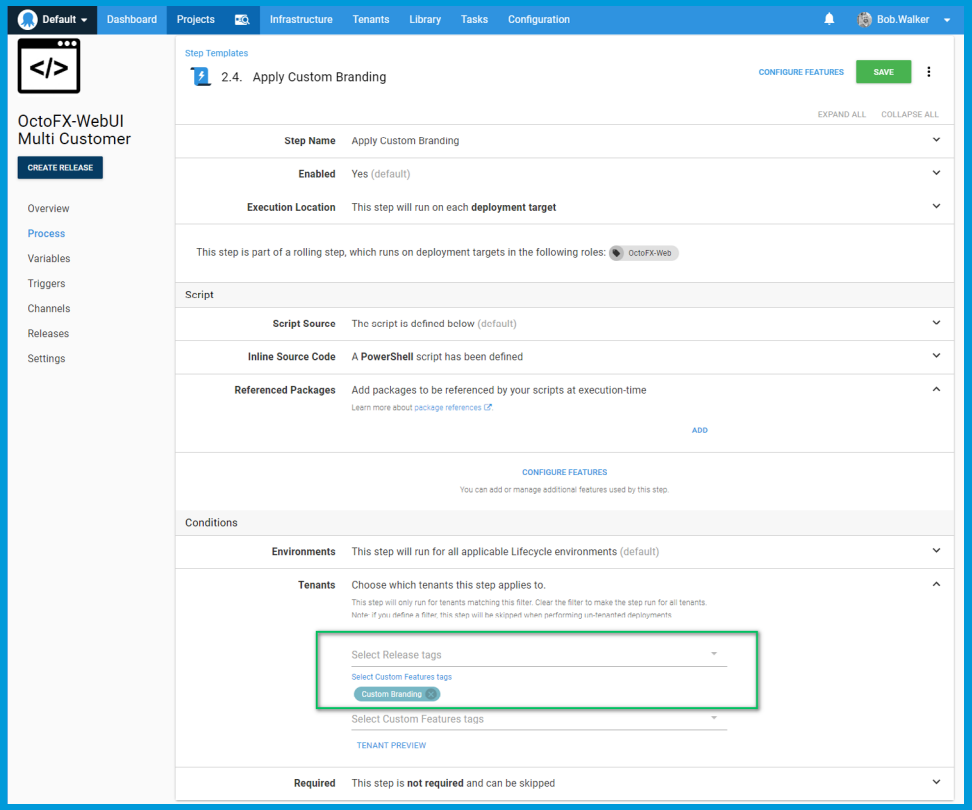
Now we need to go through and fill out the variables for each tenant.



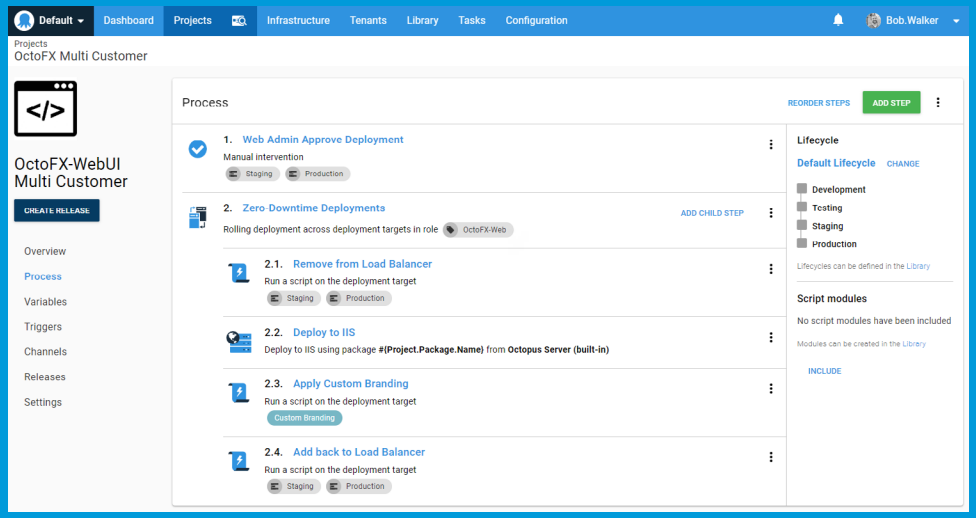
Adding variables to one or two tenants at a time isn't too terrible. However, this doesn't scale well when trying to work with 100s of tenants. It would be a good idea to automate the tenant setup using the API.

Binding Steps To Tenant Tags

At the beginning of the chapter, we created the tenant tag set to handle custom branding. Because we added tenant tags, you will now see a new condition in the UI. This condition allows you to filter which tenant tags will run this step.



The process will also show the step will only be run for tenants with that specific tenant tag.



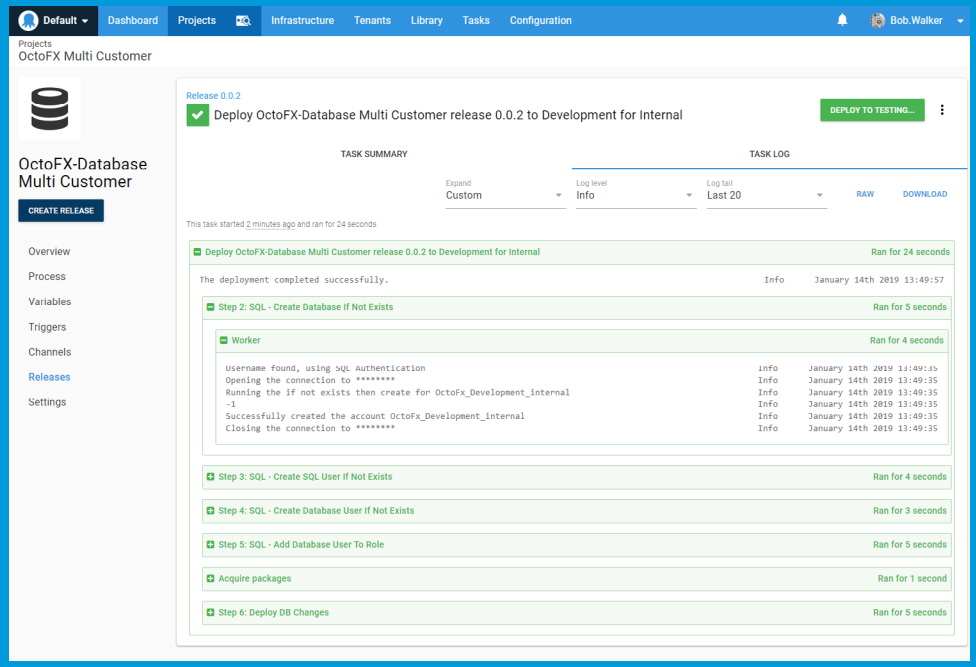
> Tenant tags are like roles, meaning it is treated as an OR for the runtime condition — not an AND. Specifying two tenant tags will run that step for all tenants with either tag, not bot tags.

Our recommendation with using this feature is only to use one tenant tag if at all possible. If necessary, create an additional tenant tag, or create a new tenant tag set.

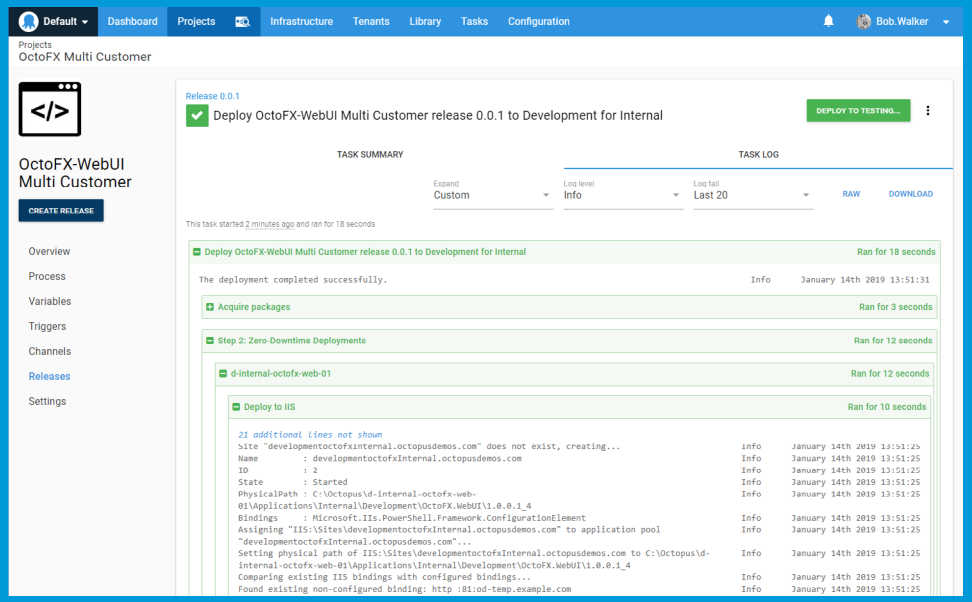
Creating the first releases

Everything required has been updated or created. It is time for the moment of truth, creating a release and deploying it to the development environment.

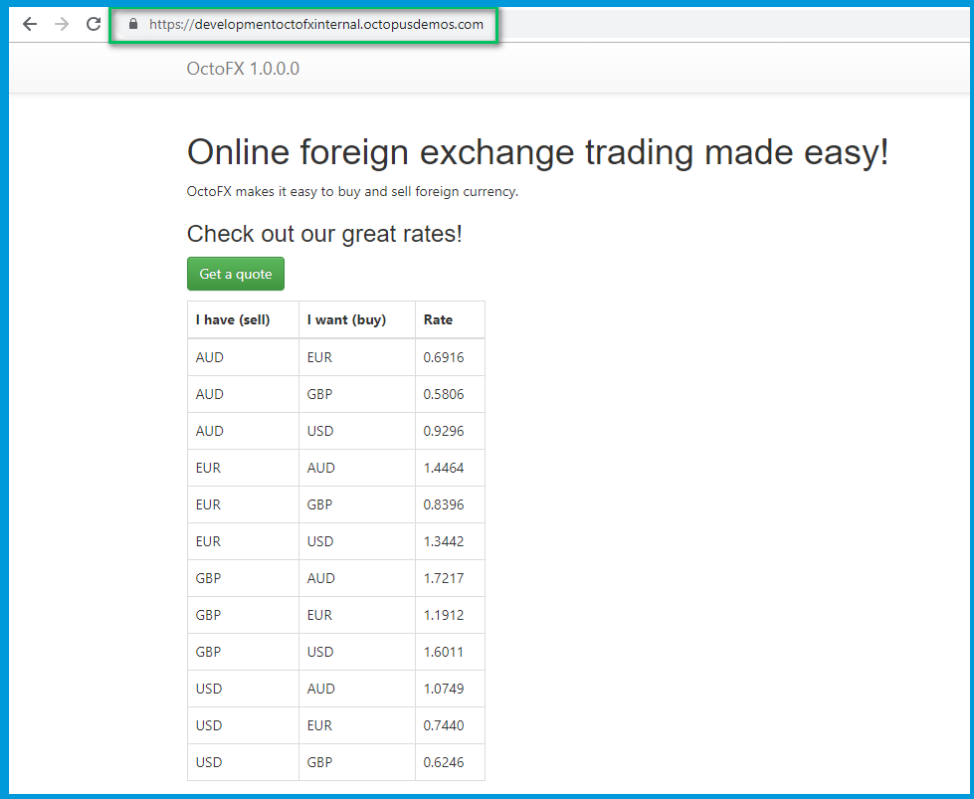
The database release was successful.



The website release was successful.

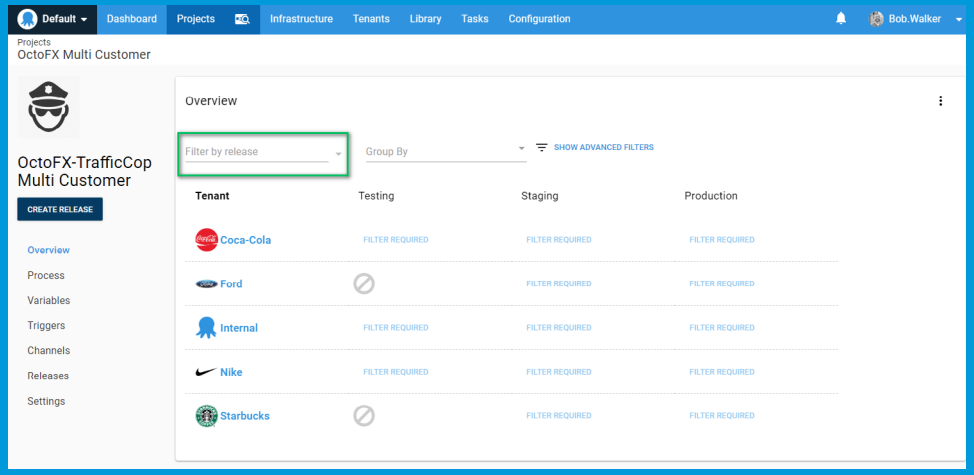


Also, if we go to the website we just deployed we can see that is successfully running.

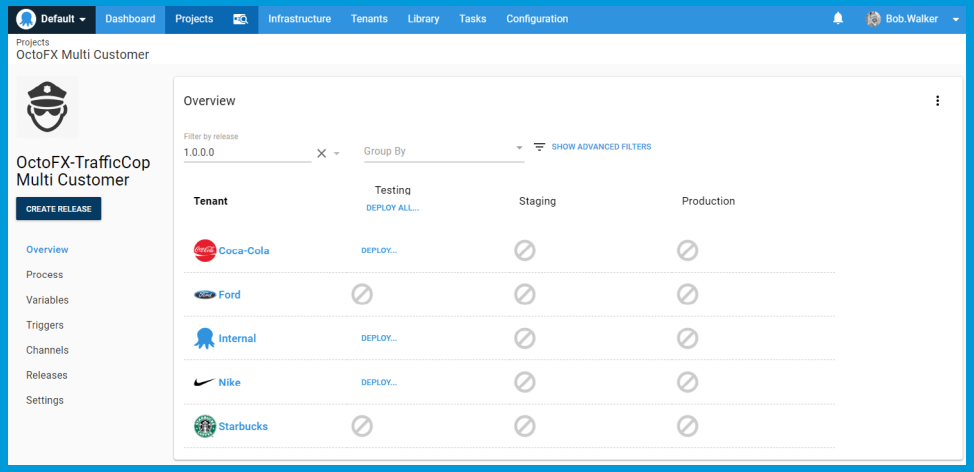


I have (sell)	I want (buy)	Rate
AUD	EUR	0.6916
AUD	GBP	0.5806
AUD	USD	0.9296
EUR	AUD	1.4464
EUR	GBP	0.8396
EUR	USD	1.3442
GBP	AUD	1.7217
GBP	EUR	1.1912
GBP	USD	1.6011
USD	AUD	1.0749
USD	EUR	0.7440
USD	GBP	0.6246

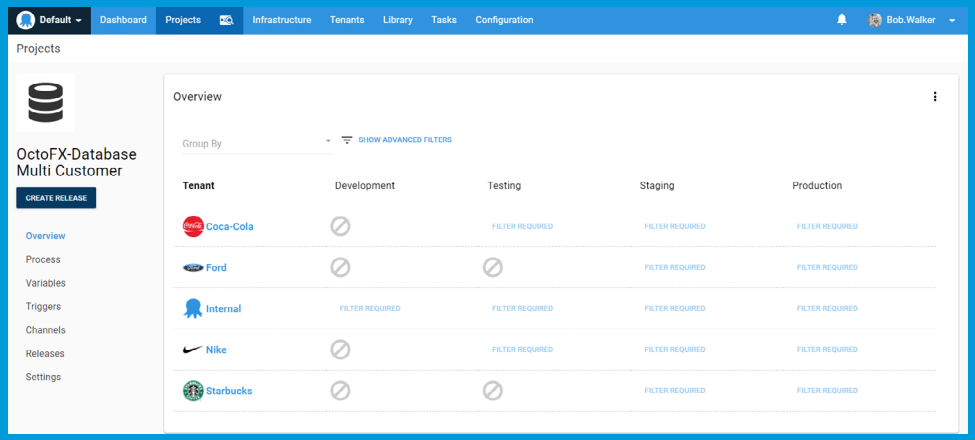
However, that was just to our internal testing website and just to development, which only has a single tenant. An exciting thing about tenants is you have to specify the version you want to deploy. For example, if we create a release for the traffic cop project we still don't see the deploy button. This is because we have to select a version from the drop-down.



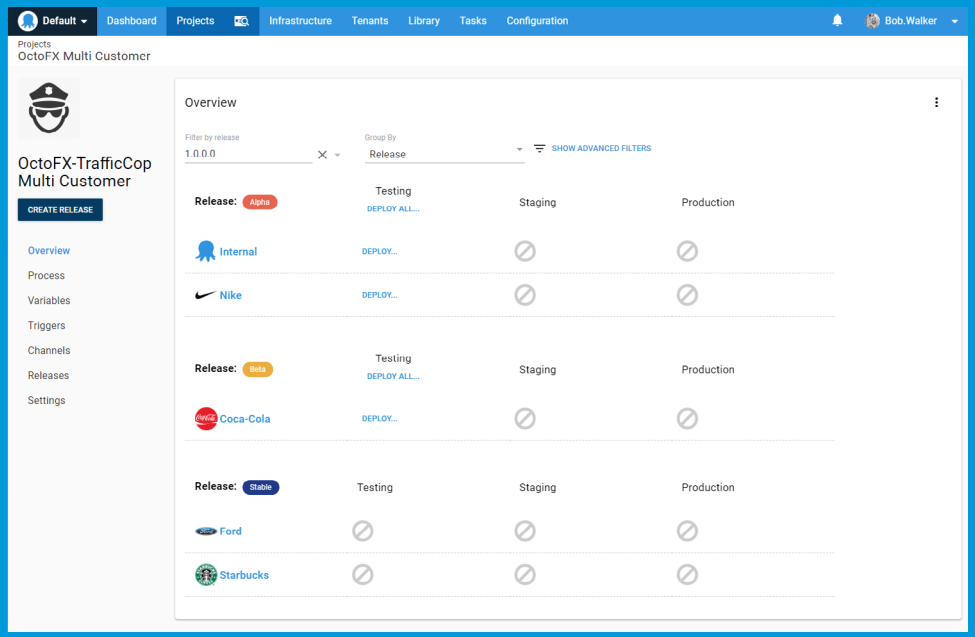
Now we see the deploy buttons for the tenants with a testing environment.



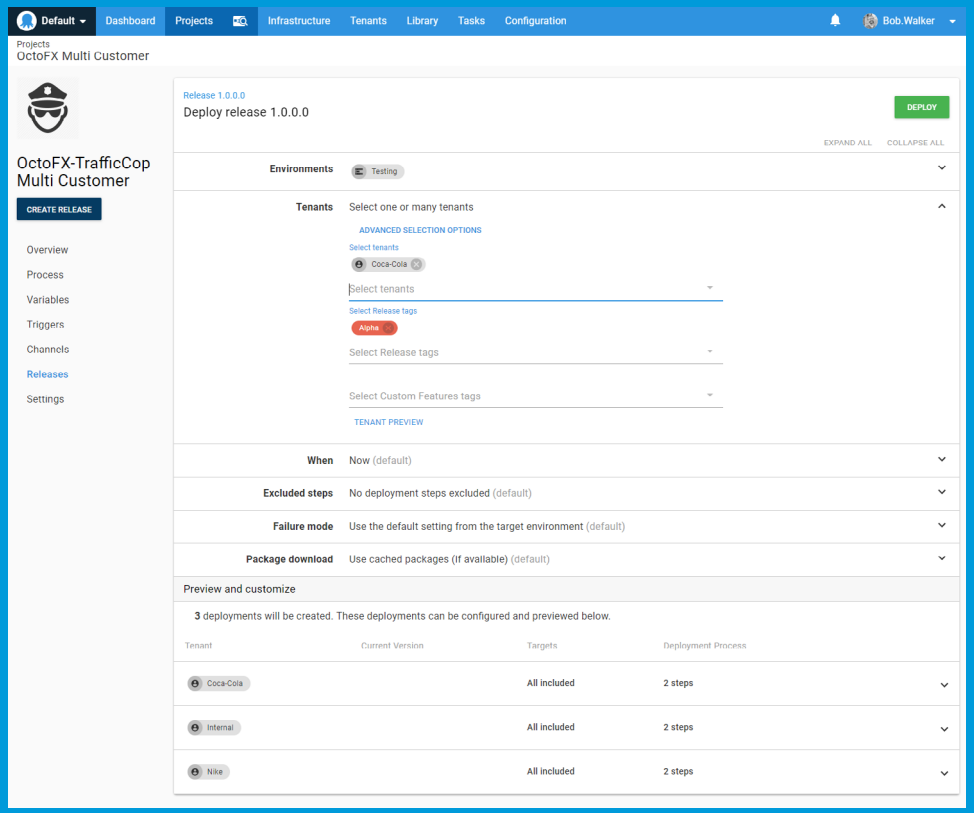
We can also group these tenants by the Release tenant tag we created earlier.

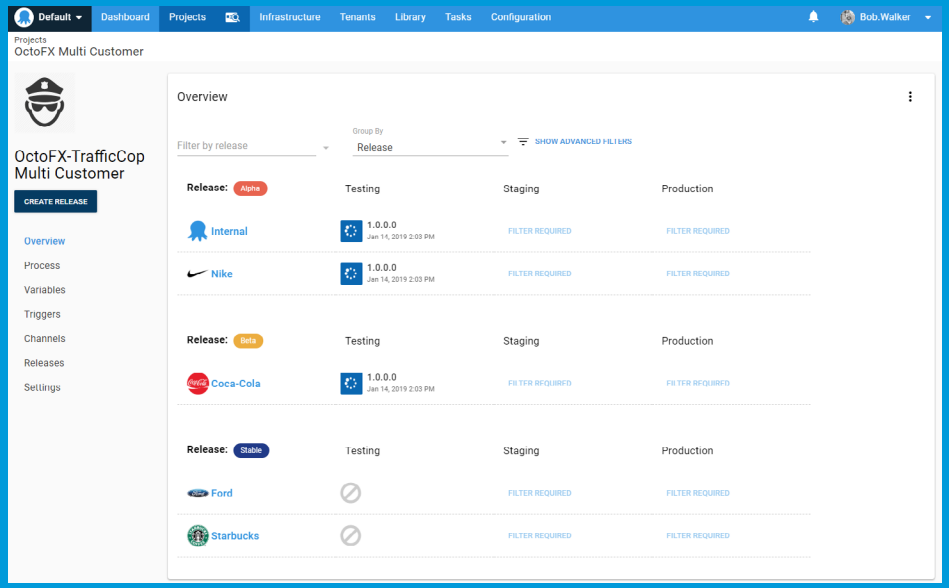


Once we do that the overview screen shifts again.



When you are ready to deploy a release, you can choose to deploy by tenant tag, by tenant name or both. At the bottom of the screen is a handy tenant preview to see which tenants will be deployed to.





It is important to point out that each tenant deployment is a unique task. This allows you to deploy to multiple tenants at the same time.

Conclusion

There were quite a number of changes we made to get this project ready for multi-tenancy. Some of the changes were a result of some shortcuts we took when we first set up the project in earlier chapters. For example, changing the website from being a web application under “default web site” to being a unique website with a sub-domain. Other changes were due to how multi-tenancy is set up, such as adding tenant tags. However, these changes were able to support all of these scenarios.

1. Each of our customers has a separate web server and database. They get the same code. The deployment process should be the same. The only difference is the destination server and some variables such as the database server and user.

2. We should be able to choose when a customer gets a specific version of our code. Some of our customers want to be on the cutting edge, their tolerance for risk is high. They have no problem being the “guinea pig” for a new feature. Some of our other customers have a low-risk tolerance. They only want to be on the stable version.

3. We should be able to group our customers into different release rings, such as Alpha, Beta and Stable. This allows us to release new versions of code to multiple customers at once.

4. Some of our customers have custom branding. During deployments, we need to have the ability to run an additional step to add in the branding.

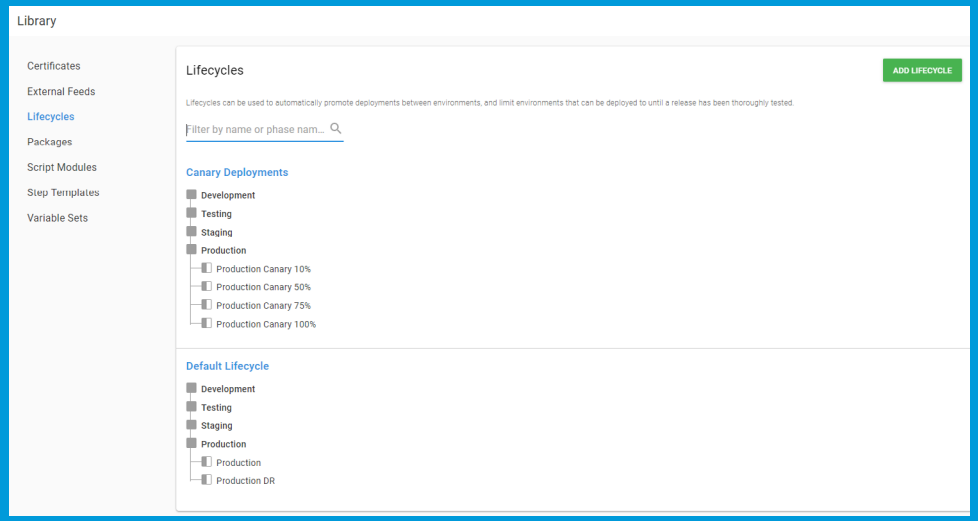
This only scratches the surface of what you can do with multi-tenancy. In the next chapter, we are going to walk through how to set up an application to be deployed across multiple data centers.

13

DEPLOYING TO MULTIPLE DATA CENTRES

In the age of cloud providers and disaster recovery, it is common to see scenarios where our users want to deploy to multiple data centers. In some cases, a customer could be running a live/live situation, where the code is running in all the data centers. In other cases we see customers wanting to deploy to the primary data center first and then deploy to a disaster recovery data center. Another typical scenario is a data center might be the "canary" where deployments occur first. After that 5% of all traffic is routed to that data center. Assuming everything looks good then the deployment is pushed to all remaining data centers.

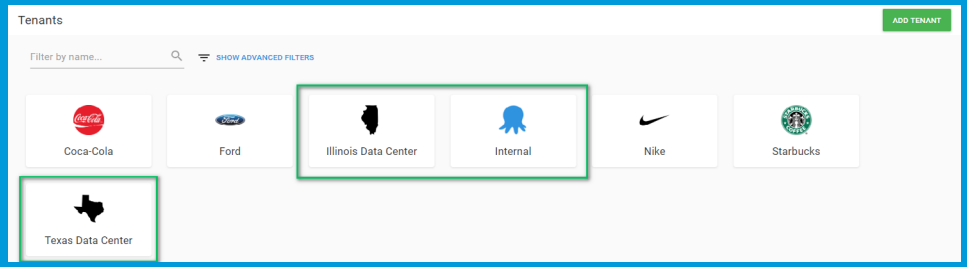
With some of our customers, we see all those scenarios on a single Octopus Server. All too often we see lots of "Production" environments and a lot of unique and custom lifecycles.



Scalability is a major concern with this approach. What happens if you want to add Production Canary 30% as an environment? You would first have to add the environment. Then you have a decision to make, do you add it to the Canary Deployments Lifecycle? That means any project gets that new environment. Is that good or bad (we don't know, but it is a question you would need to ask). What about any variables scoped to environments? How would you know which projects needed new variables added for Production Canary 30%?

> A good indicator something isn't quite right with your environments or lifecycles is when you have to add [Environment Name] [Qualifier], such as **`Production DR`** or **`Production Canary 30%.`**

Earlier in the book, we created three tenants, **`Internal, Illinois Data Center,`** and **`Texas Data Center.`** In this chapter, we will cover how to leverage those tenants to support a multi-data center deployment.



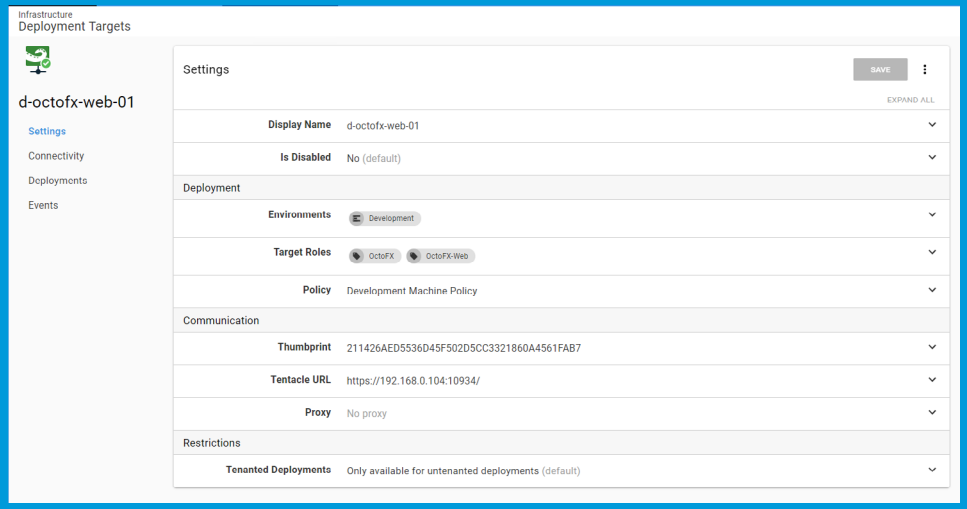
The use case for those tenants is as follows.

- Internal: These are servers hosted at our main headquarters. All development and testing deployments go to these servers.
- Texas Data Center: Our primary production data center for our external applications. That is a data center for a cloud provider, and we have VMs running.
- Illinois Data Center: Our disaster recovery data center. Our cloud provider has a fast connection between these two data centers. During the day the VMs are running, whereas on the nights and weekends they are turned off. In the event the Texas Data Center would go offline the load balancer would automatically redirect all traffic to Illinois.

When we do a deployment, we are first going to deploy to the internal data center for Dev, Test, and Staging. Assuming everything looks good in Staging on the internal data center, we would then push to Staging for Texas and then Illinois. For a production deployment we would deploy to Texas first, and the next day we would push to the Illinois data center.

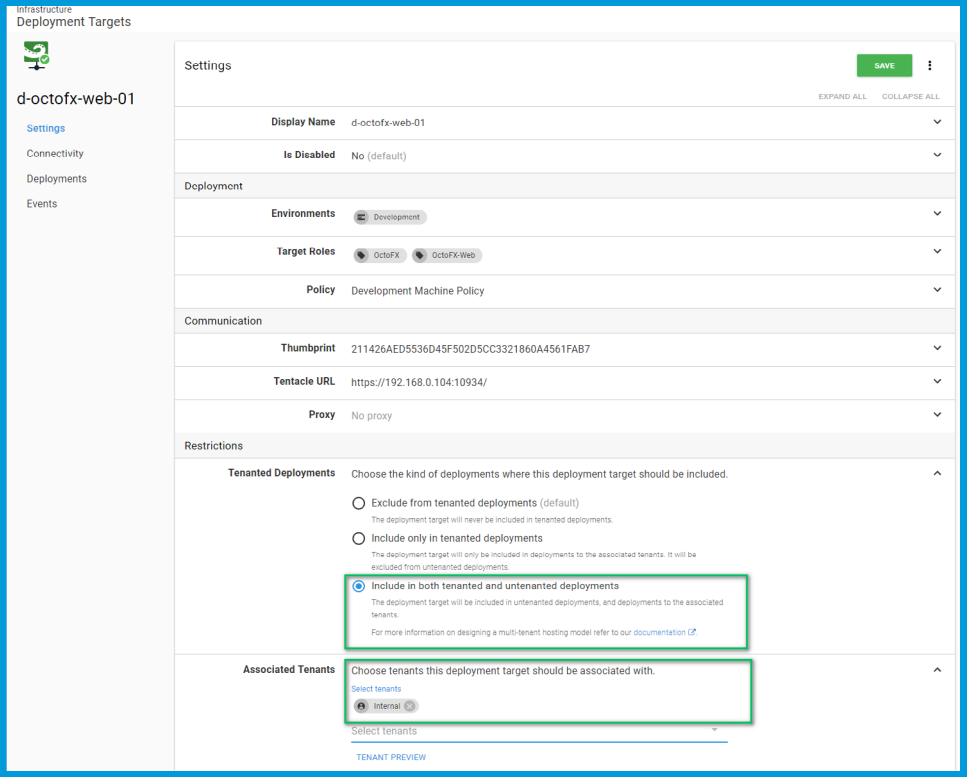
Infrastructure

First, we need to make some changes to our existing infrastructure. We have already had several servers located on-premise. The servers for Development, Testing, and Staging are shared for internal and external applications. We want to configure those for both tenanted and non-tenanted deployments.



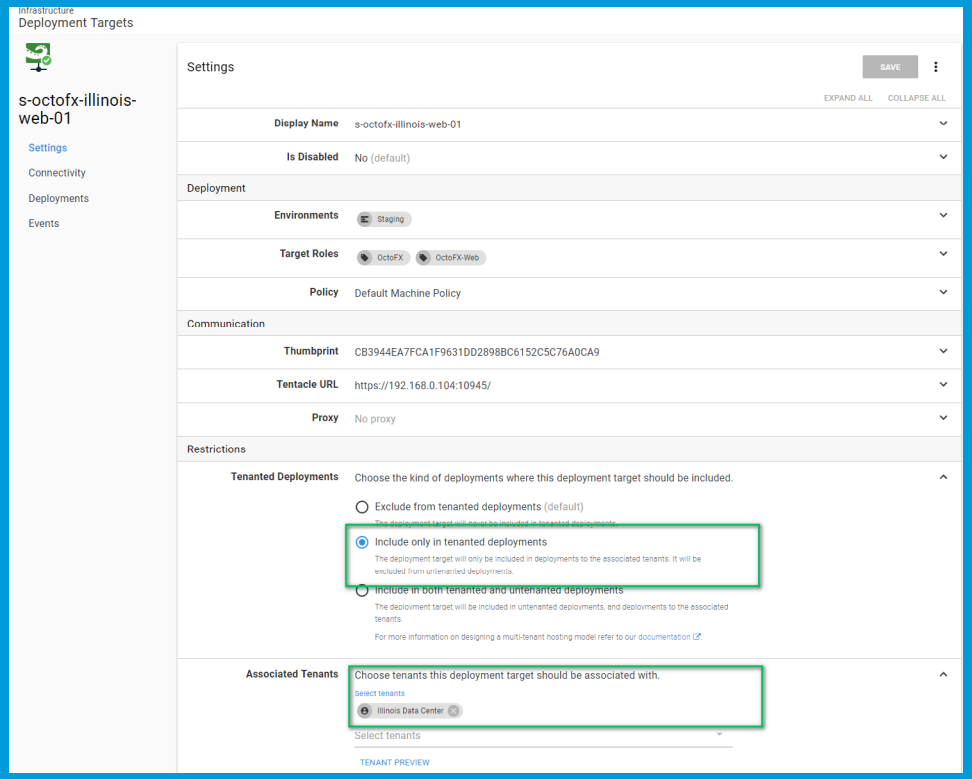
That configuration can be done by expanding out the tenant section and selecting **`Include in both tenanted and untenanted deployments`** and then selecting the internal tenant.

> This can get rather tedious if you have more than 20 targets to update. It would be a good idea to automate this using the API to speed this up.



For the machines in the Texas and Illinois Data Centers, we only want to use those for tenanted deployments. The tenant options for that machine will be **`Include only in tenanted deployments`** as well as the tenant for the machine. Please note, that for these machines the display name now includes the data center, for example, **`s-octofx-illinois-web-01.`**

This is somewhat contradictory to the recommendation about environments. Environments are used almost everywhere, lifecycles, variable scoping, step scoping, and so on. Machine display names are used much less; they are primarily used for logging during a deployment. While it is possible to scope a variable to them, that is rare.

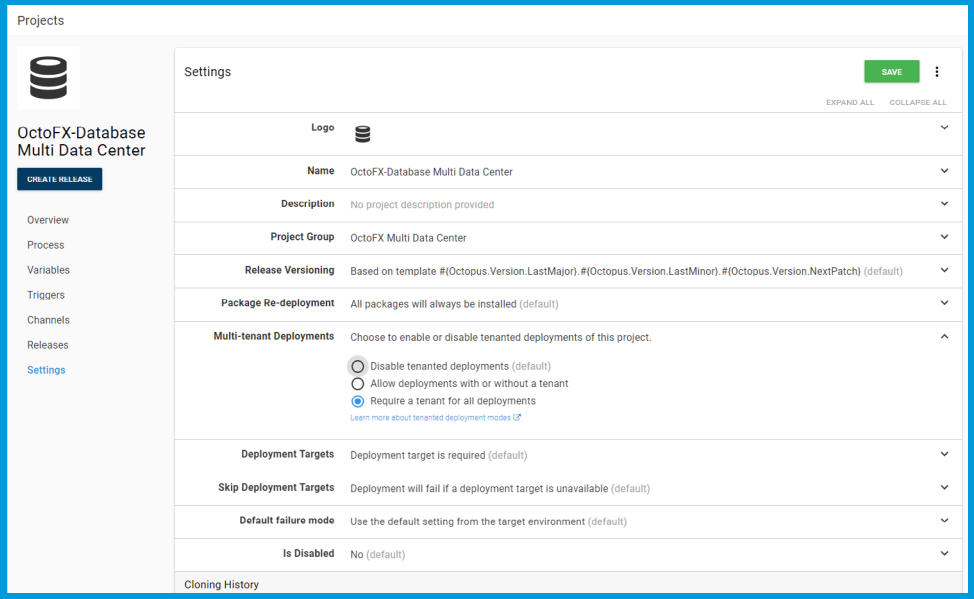


Projects

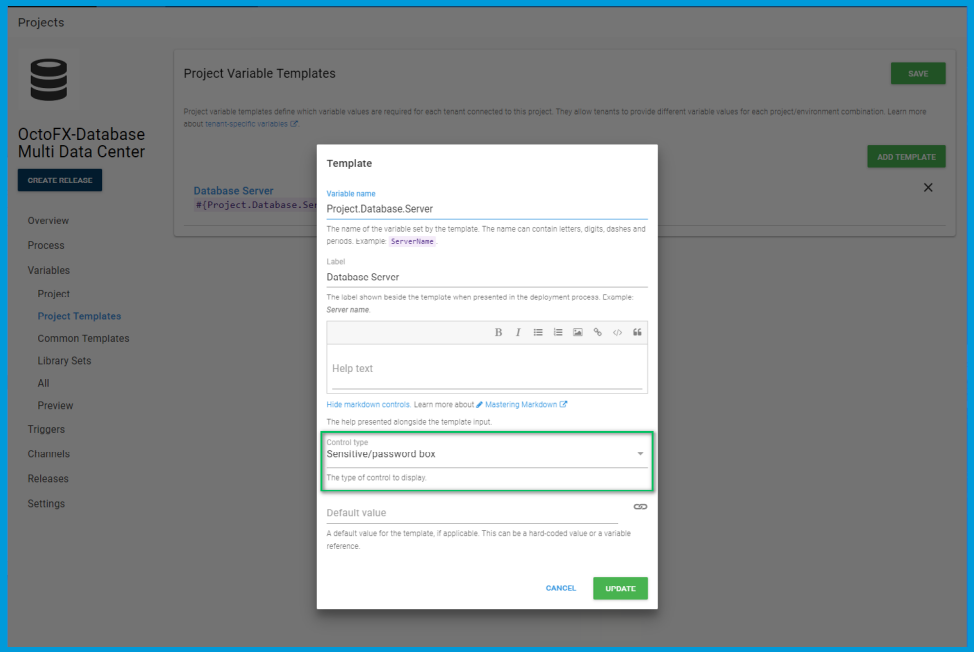
Next up is the project configuration. For this demo, we went ahead and cloned the existing projects and called them OctoFx-Database Multi Data Center, OctoFx-Traffic Cop Multi Data Center, and OctoFx-Web Multi Data Center. In the real world, you wouldn't have two projects, one tenanted and the other untenanted.

We need to configure the project to allow for multi-tenant deployments. Go to the project and then click on the settings link on the left. Expand out the multi-tenant deployment section and select Require a tenant for deployments.

While it is possible to have deployments with or without a tenant it something we recommend. The project should either require tenants or not allow tenants. Having a project with or without a tenant makes it very confusing to most users, which increases the chances of someone making a mistake.



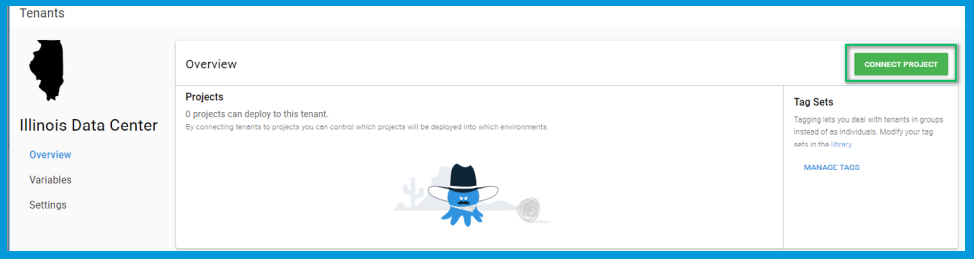
Let's take a quick step back and think about this specific project, which handles the database deployments. What would be different between the data centers? The database server would be different. However, other than that, everything else should be the same. To account for these differences, we are going to create a project template variable. The nice thing about the project template variables is that they also support the ability to mark the value as sensitive.



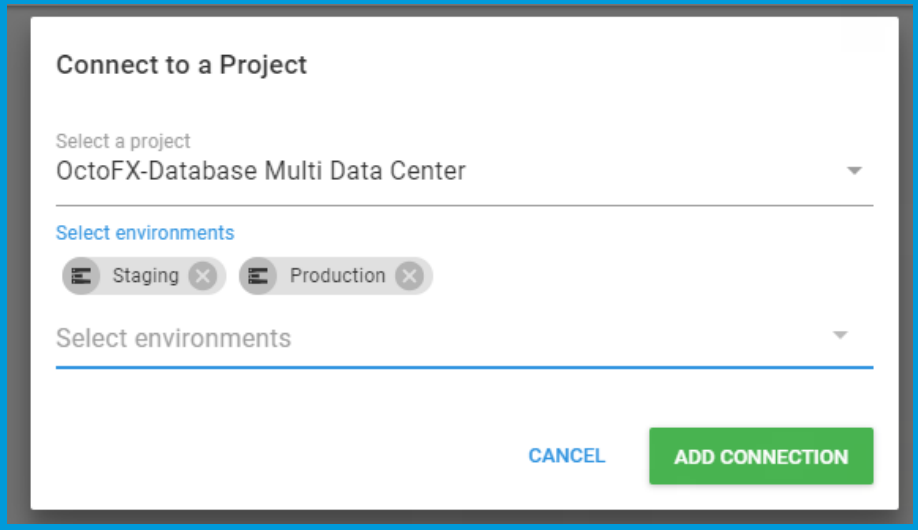
We will have to set up a couple of project template variables for the WebUI project, but that process is the same. Aside from that, that is all we need to configure on the project side.

Tenants

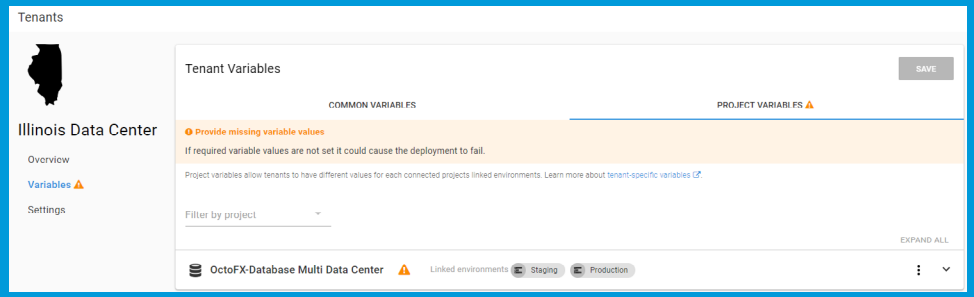
Finally, we need to link the data center tenants up to the projects. Go to the tenant's screen, selecting the tenant you wish to link up, and then clicking the **`Connect Projects`** button.



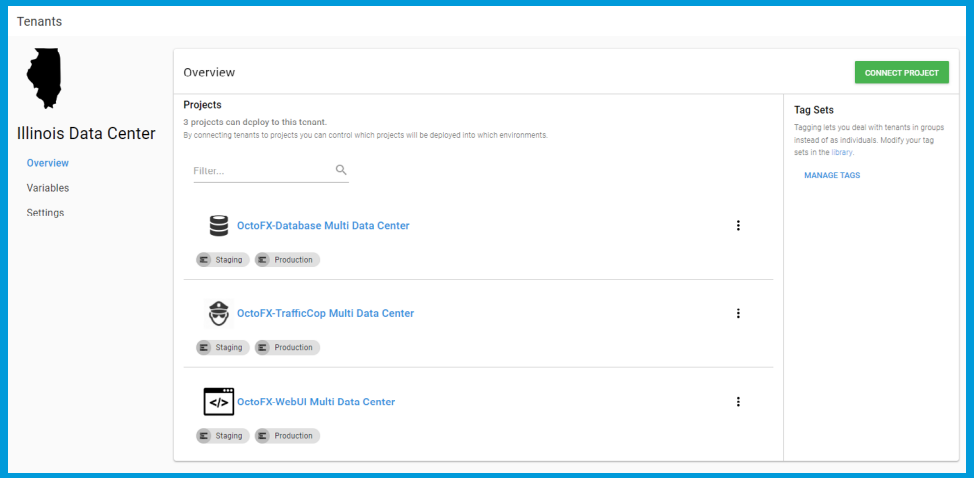
When you are connecting the project to the tenant, you can select the deployment environments for the tenant. If you recall from earlier, the Illinois Data Center will have two environments, **`Staging`** and **`Production`**.



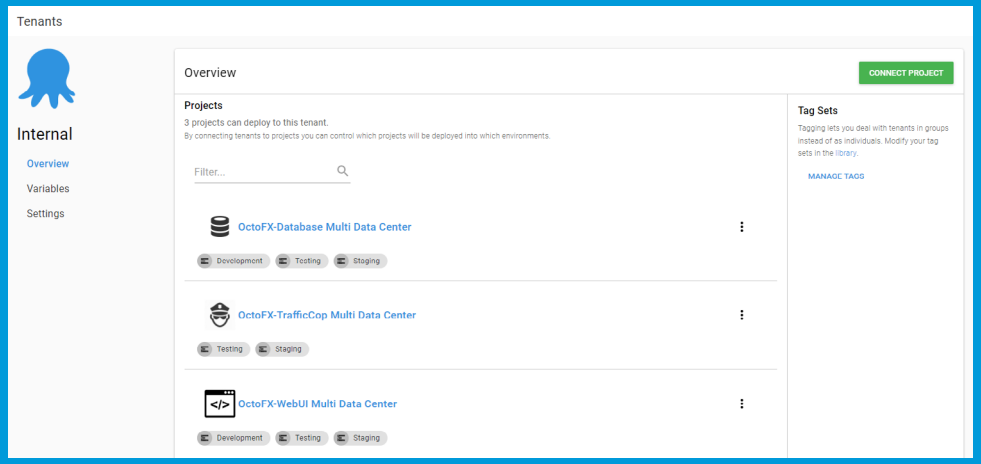
After connecting the project, the screen will then notify you of missing variables. We need to go to the variables section and fill out the database server variable.



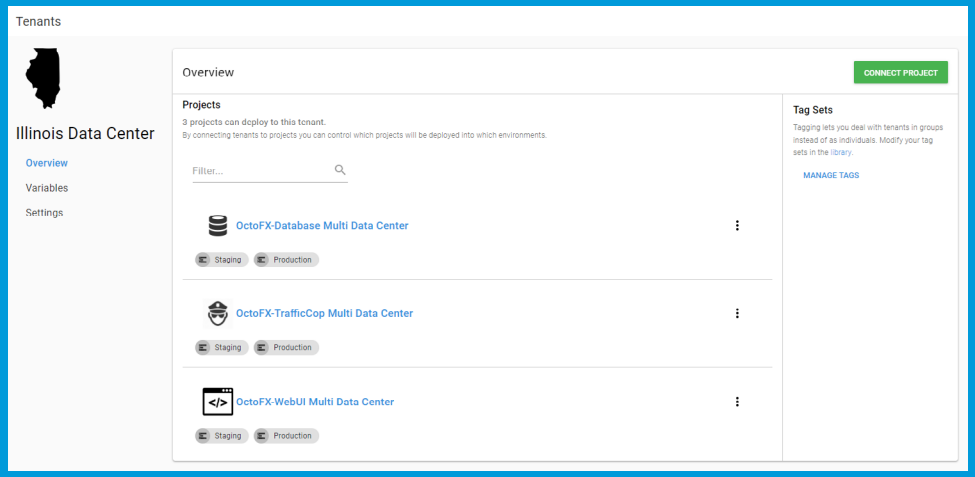
After linking the other projects, the result for the Illinois Data Center will look like this.



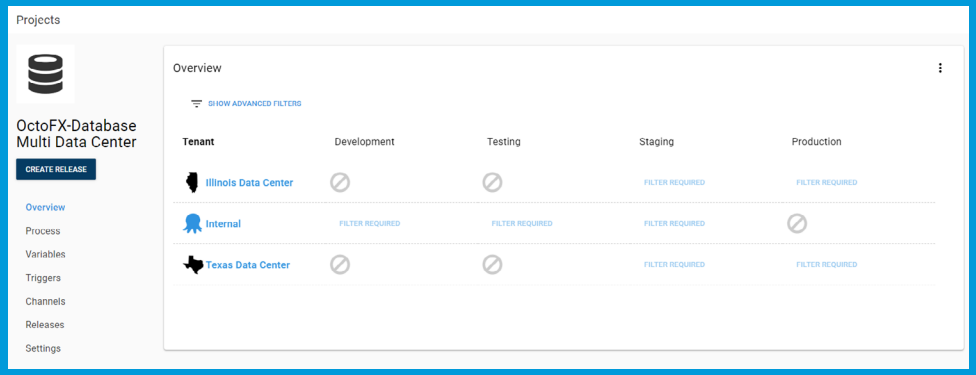
The internal tenant will look like this. Note that it configured to only deploy to **`Development`**, **`Testing`**, and **`Staging`**.



Finally, we can wrap it all up with the Texas Data Center. With regards to environments, it matches the Illinois Data Center.

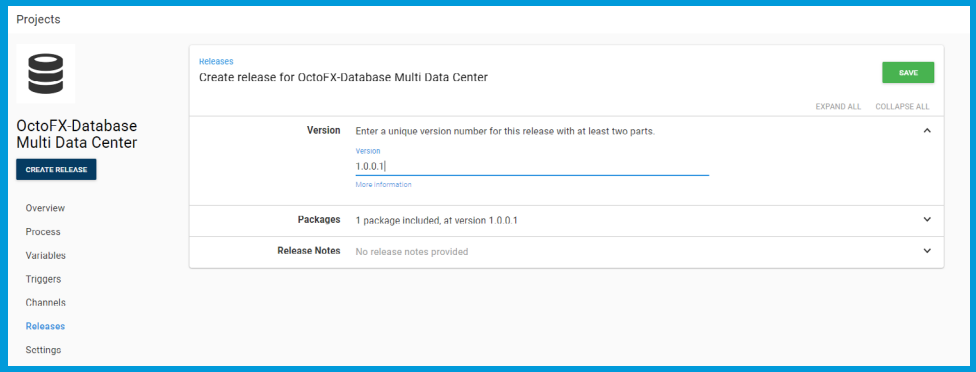


When we look at the database project's dashboard, we can see the view has changed from the typical untenanted deployment dashboard. Make note that you cannot deploy to the Texas or Illinois Data Centers for the **`Development`** or **`Testing`** environments.

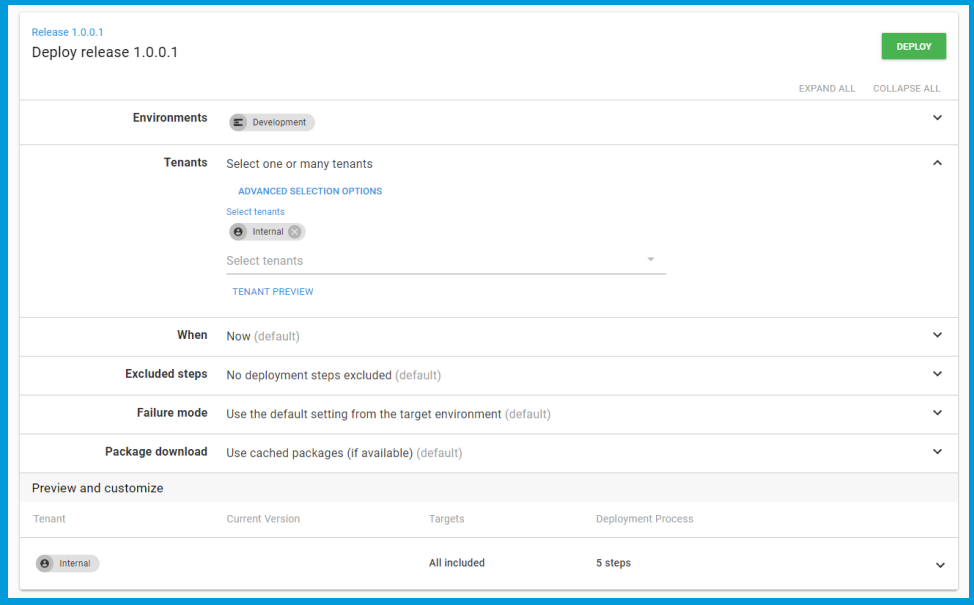


Deploying Code

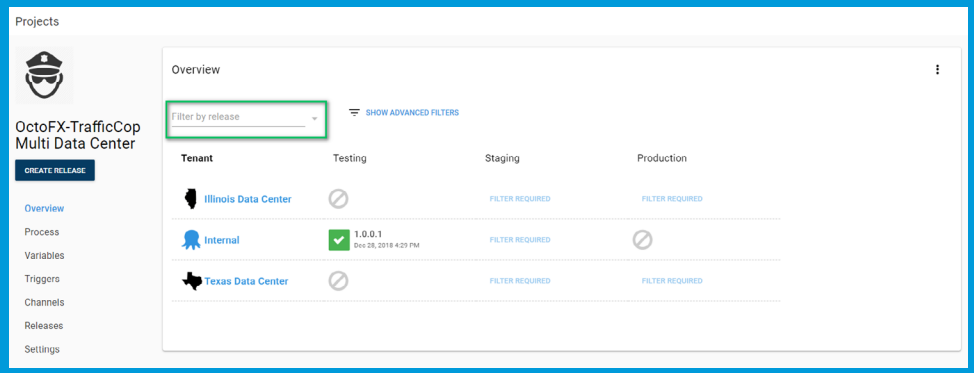
Creating a release is the same as before, you provide a version number and select the package you wish to deploy.



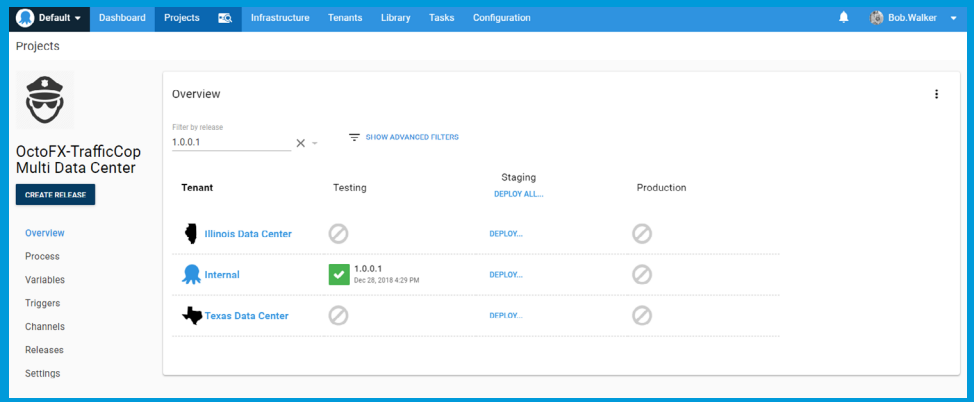
When you deploy to development, you can choose the tenants. The tenant is automatically pre-selected when there is a single tenant for that environment.



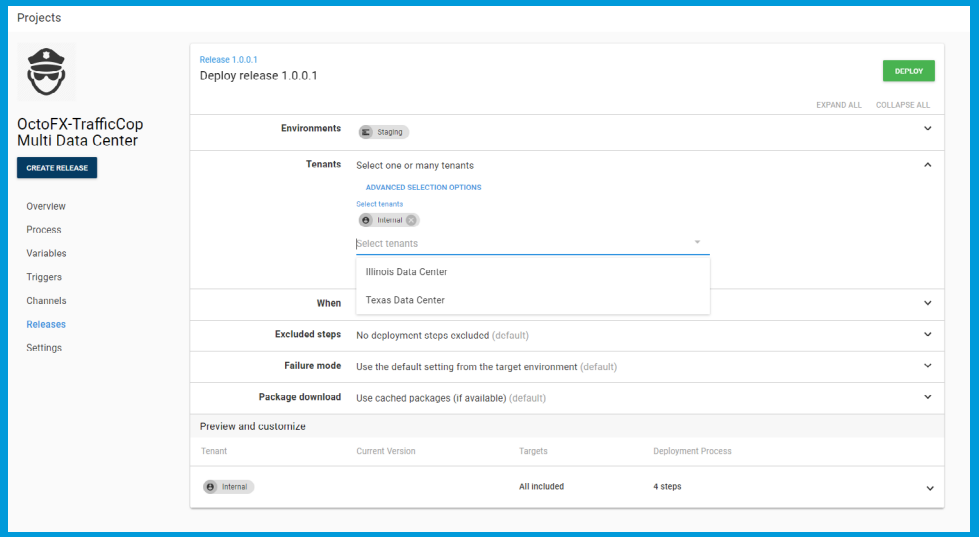
Let's get the WebUI, Database, and the Traffic Cop pushed up to the Testing environment. If you look closely at the main dashboard, you will notice a gray box has now appeared with 1/1. The numerator is how many tenants have been deployed with that release. The denominator is how many total tenants are available in that environment. What this is telling us is 1 out of 1 tenant in both **`Development`**, and **`Testing`** have the **`1.0.0.1`** release.



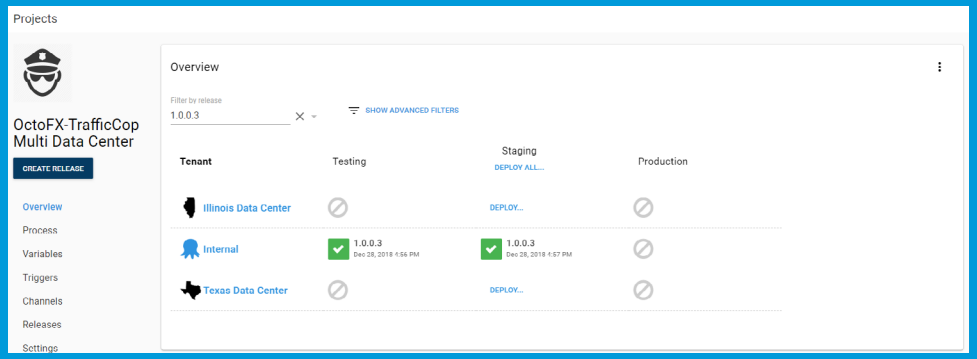
Once a release is selected, the Deploy buttons appear. Notice that each tenant has the Deploy button for the Staging environment. That allows us to deploy to **`Staging`** for the **`Internal`** data center first and then we can deploy to the other data centers.



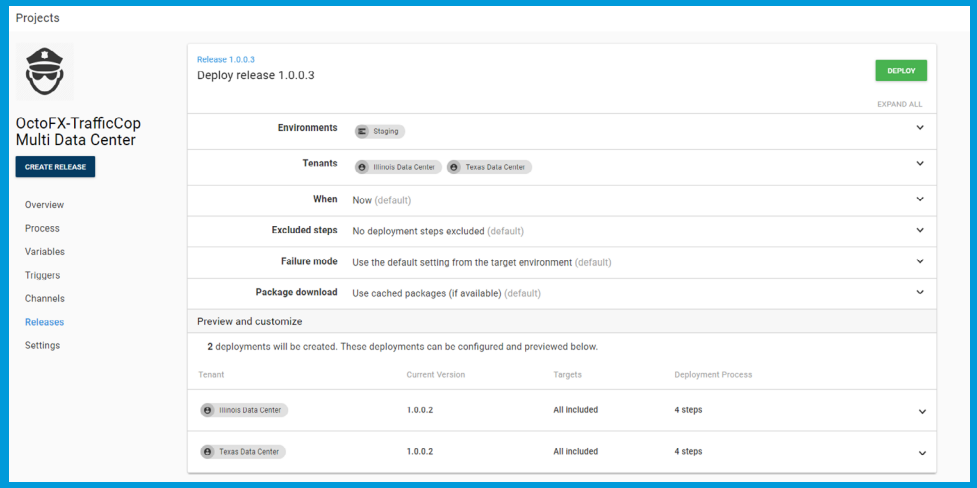
Clicking on the **`Deploy`** button allows us to choose the data centers for deployment. A multi-tenant deployment can push to 1 to N number of tenants at a time. Right now we are just going to pick the **`Internal`** data center.



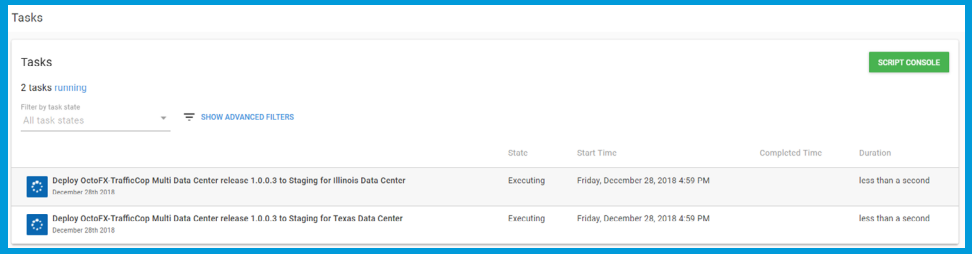
That deployment was successful.



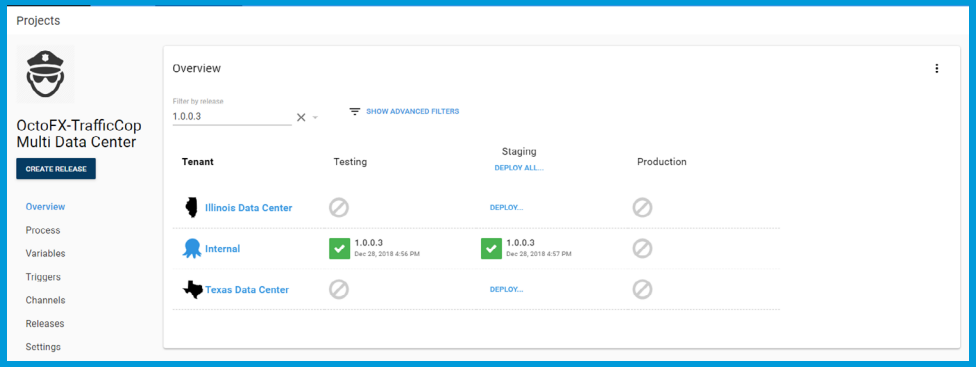
Now we can deploy to both Illinois and the Texas Data Center.



When clicking on the deploy button for both data centers, a navigation event will occur to a new screen. The screen shows the deployment for each tenant is treated as a separate task. That means deployments can run at the same time. Which makes the window for the data centers to be out of sync much smaller (assuming the deployment to both were successful)



The option to deploy to internal is not there when deploying to Production. Only deployments the Illinois and Texas Data Center are allowed.



Conclusion

In this chapter, we walked through how to configure a Multiple Data Center deployment using tenants. The primary advantage to using tenants is you can keep the same lifecycle as before (Development -> Testing -> Staging -> Production), but still have the flexibility to choose which data center to deploy to. It also provides the ability to know when specific variables are scoped to a data center which you must change. Finally, each deployment to each data center is treated as a unique task, allowing you to deploy to both data centers at the same time.

14

DEPLOYING FEATURE BRANCHES

One of the great things about GIT is how easy it is to create a branch. That ability has several benefits. Namely, it allows for experimentation. A developer can try out an idea in a branch and if it doesn't work they can delete the branch. No fuss. No muss. Or they can start work on a new feature in a feature branch. Once the feature is complete everything is merged into the main branch.

Eventually developers will need to get feedback on the feature branch. They want to check-in the code and deploy it so someone else can take a look at it. Often times the code is in an incomplete state. It shouldn't be merged into master or development. That could have some serious consequences. Incomplete code could get pushed out to test or heaven forbid, production before it is ready. QA doesn't want to test incomplete code.

A common technique is to hide the change behind a feature flag. But that doesn't scale all that well. Depending on the change, there could be a lot of if/then statements for the change. Someone has to go in and clean up that feature flag. Which causes more work.

In a perfect world, that feature branch would get its own set of resources. In our example application, OctoFX, this means a website and a database. Those resources would be automatically created when the user pushes the feature branch up to the server.

If you recall in our projects chapter we said a core rule when configuring your projects is Projects should be responsible for setting up what it needs to run. This means creating a website and database. In that chapter we created the database and web project with that rule in mind. The database project will create the database if it does not exist. The WebUI project will create a website if it doesn't exist. Resource creation is covered.

We need to make a few other tweaks to Octopus Deploy to help support this scenario.

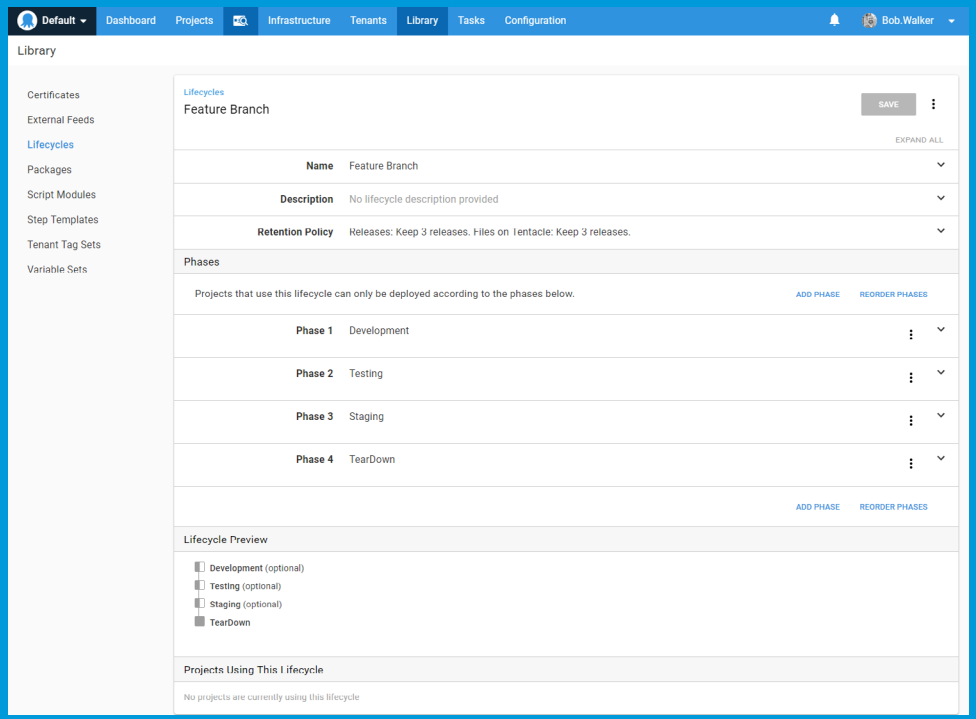
Lifecycle

First, we need to determine which environments we are going to be deploying to. In this book, we have been using four environments, Development, Testing, Staging, and Production. Our recommendation is to to be able to deploy to an environment where users can test out the changes. If you are working on a SaaS application, very often staging is a mirror of Production. The outside world has access to connect to it. Getting feedback from our external users means deploying all the way to Staging. No external users? Deploying to Development and Testing makes more sense.

Another item to consider, the resources being created for the feature branch, the database and the website, have a finite lifespan. We will need to destroy them once the feature branch has been merged into master. As a result of that, we should include a "TearDown" environment in our lifecycle.

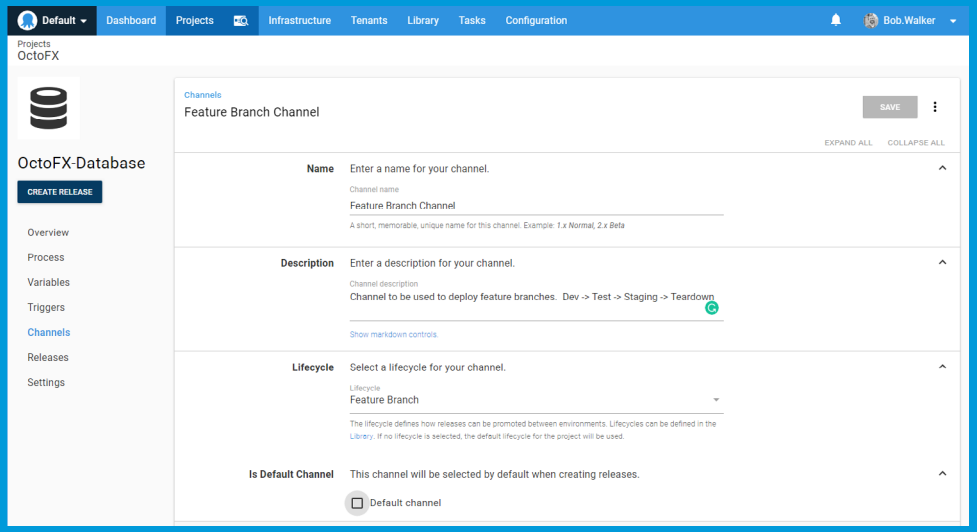
Each feature branch will be different. Some will go to development only. Others will go to Testing. And others might go all the way to Staging. We still need to deploy to TearDown to destroy the temporary website and database. Each phase in the lifecycle, Development, Testing, and Staging should be optional.

We will create a new lifecycle to support this scenario.



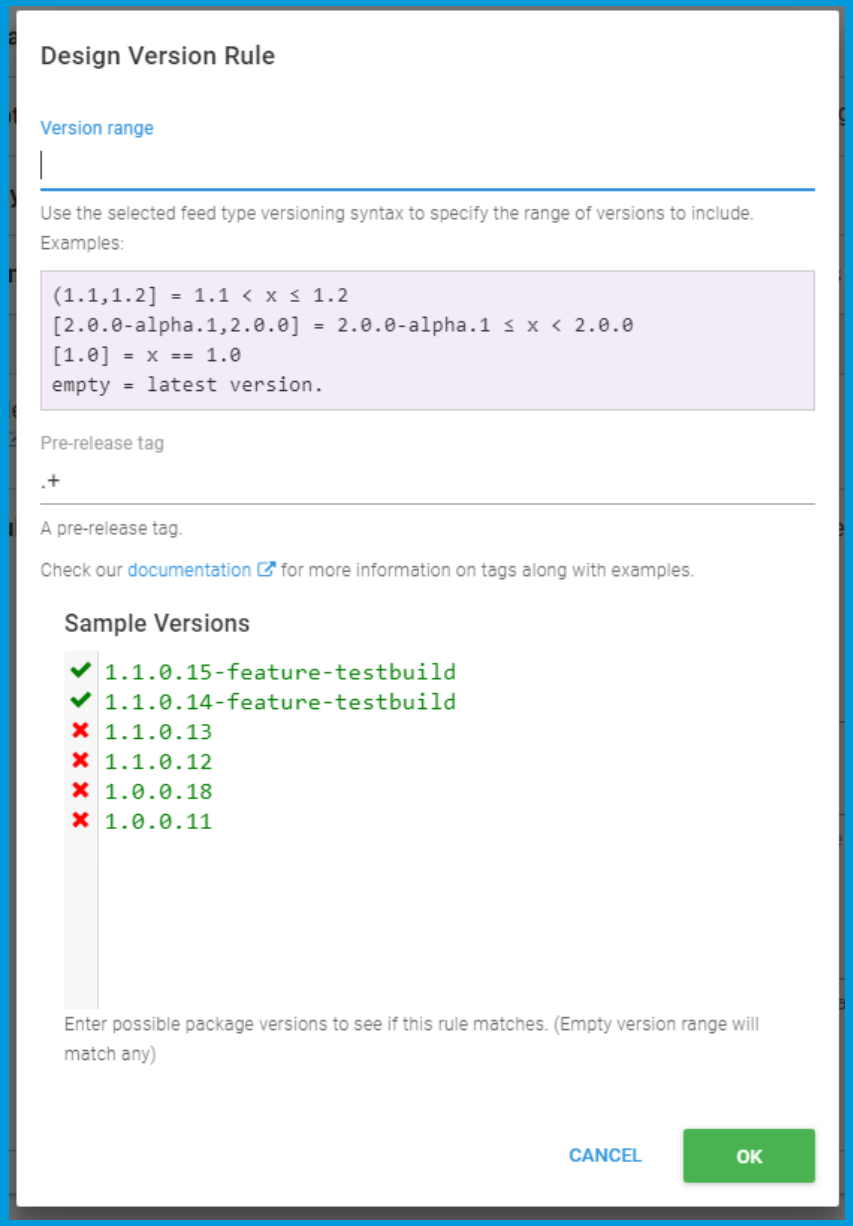
Channels and Versioning

Channels allow you to tie more than one lifecycle to a project. We will need to create a new channel to make use of this new lifecycle.

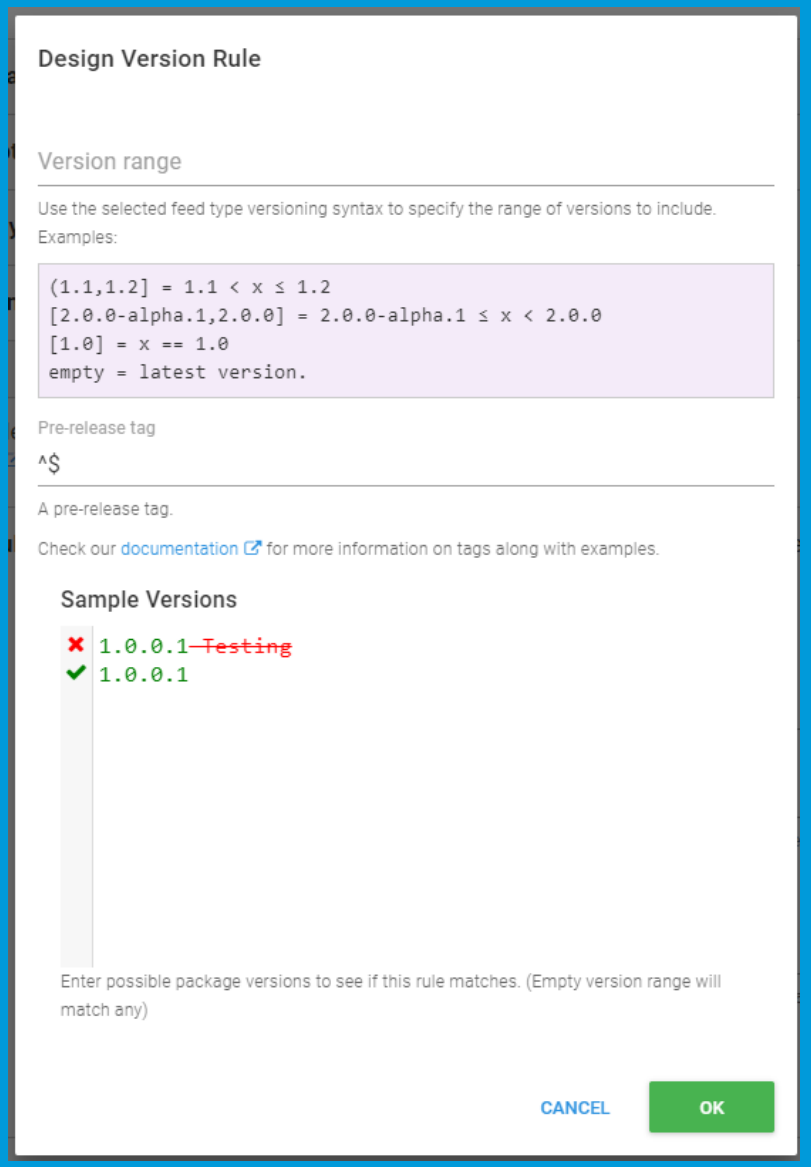


Octopus Deploy adopts a modified version of SemVer. We are not as strict as SemVer. Octopus Deploy implements a number of its requirements. This includes support for pre-release text after the version number (IE 2018.1.9-PreRelease or 1.0.0.0-PreRelease). We will be making use of that pre-release text. This will be for both the package version as well as the release number.

We will create a version rule for this channel where it only accepts packages with pre-release text.



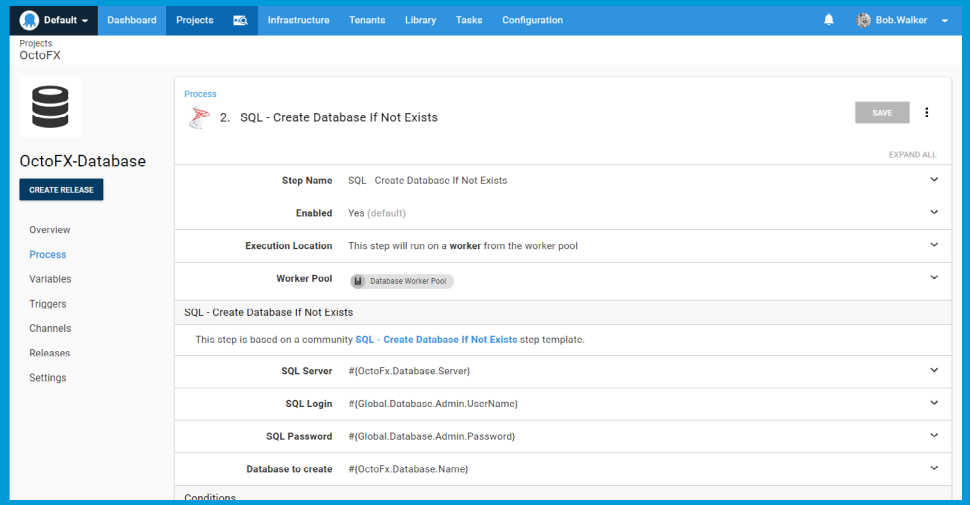
We need to go back to the default channel and tell it to accept any version without pre-release tags.



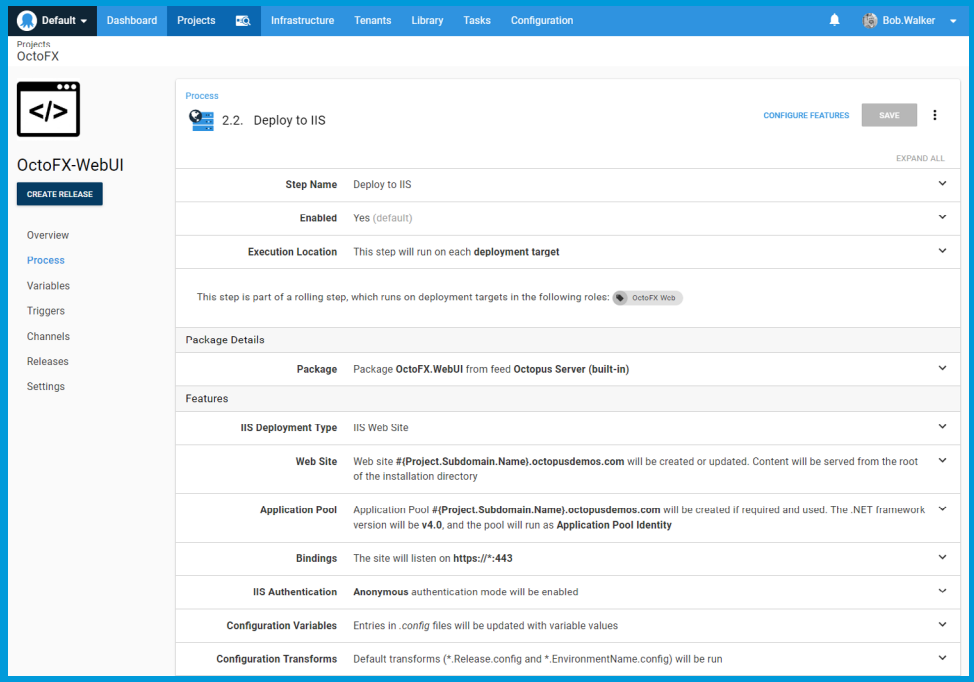
The only downside is you will need to repeat that for each of our projects we want to deploy feature branches.

Process Changes

If you have been following along so far with the book, you will note everything in the process is driven by variables. From the database deployments:



To the WebUI Deployments:



We recommended driving your process using variables for this very reason. We can change around those variables and not have to worry too much about our process. That being said, there might be a couple of new variables we will have to use.

The case against using tenants

There are two approaches to deploying feature branches. The first approach to consider is using tenants. Each feature branch becomes a tenant.

At first blush, this makes the most sense. The problem is, that doesn't scale all that well. Feature branches have a finite lifespan, a few days or a couple of weeks. You will be adding/removing tenants quite often. Yes, it is possible to automate that using the API, but that is brittle.

You also might have hundreds of projects on your Octopus Deploy server. You could have hundreds or even thousands of tenants for feature branches. With that many projects, there is a good chance for feature branch naming collision. Unless you come up with a naming standard where the feature branch has to include the name of the project.

In addition, you might already have regular customers as tenants. Feature branches as tenants would add to the list and make it harder to find a customer tenant.

The last issue is the tenant clean-up. If you happen to have a tenant name collision, when should the clean-up occur? How would you schedule something like that?

Using output variables

Rather than using tenants for feature branches, we will be using the release name to drive this. At the start of the process, we will have a PowerShell step to set an output variable based on the release name. The PowerShell script will only run on the channel we created for feature branch deployments.

15

CONFIGURING EMERGENCY BUG FIX DEPLOYMENTS

Hotfixes

It is not a great start to the day. Production is hosed. You need to get a hotfix out quickly before the start of business. What do you do? You could make those changes directly to the production database or files, but that's risky and time-consuming. And we already have this tremendous automated deployment pipeline. We just need to configure it for these scenarios. Luckily, we've gone part of the work earlier when we set up our lifecycles. Let's look at the next piece of the puzzle with channels.

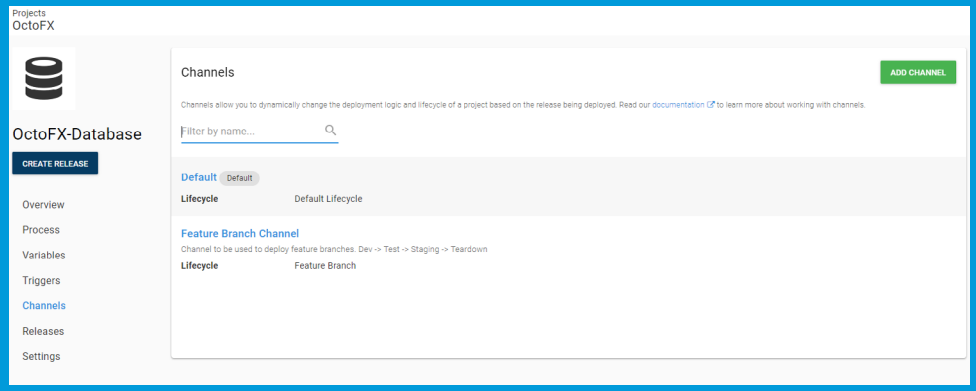
Channels

Channels let you configure your project releases to take different paths. The primary way this happens is that you can assign different lifecycles to each channel. You can limit what versions of packages are available for releases in each channel. You can also scope specific steps and variables to different channels, but for this chapter, we're going to focus on lifecycles and package version rules.

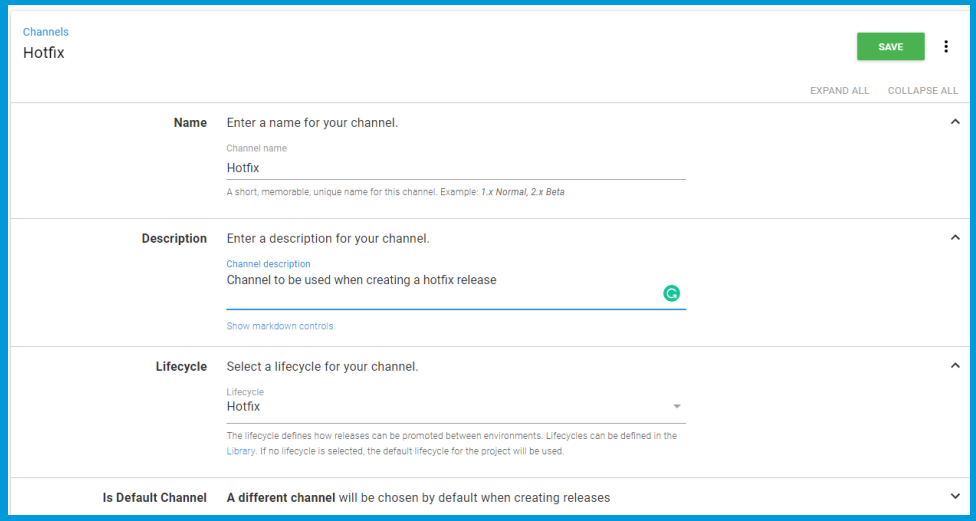
Creating the Hotfix Channel

Let's navigate over to our OctoFx-Database project and then click on Channels in the project menu.

You'll notice two channels on this page, **`Default`** and **`Feature Branch Channel.`** Every project has a default channel that uses the lifecycle defined on the Process page. The **`Feature Branch Channel`** was created in the previous chapter to support feature branch deployments.

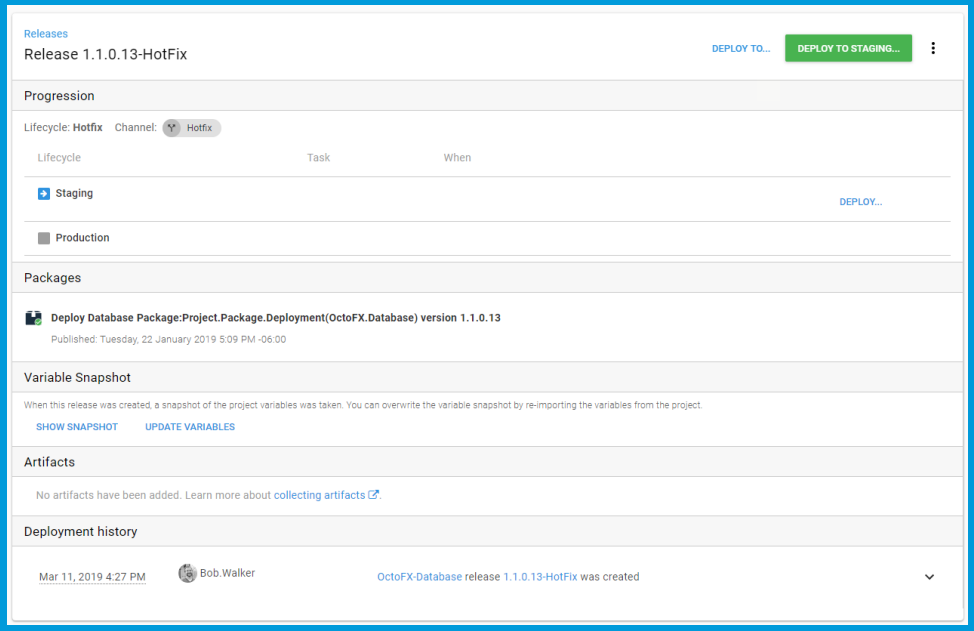


We need to add another channel to handle hotfix deployments. First, click the **`Add Channel`** button. We'll set the name to Hotfix and the Lifecycle to **`Hotfix.`**



Let's go ahead and save this and create a release in the channel to see what it looks like. Click on the **`Create Release`** button. Before you hit that save button, go ahead and change the Channel from Default to Hotfix.

Once you save the release, you'll notice that our release screen shows the Staging and Production environments only and the deploy button says "Deploy to Staging...". If we jump to the Overview page, you'll see that our releases are grouped by their channel and that the Hotfix channel only has Staging and Production environments. Very nice!

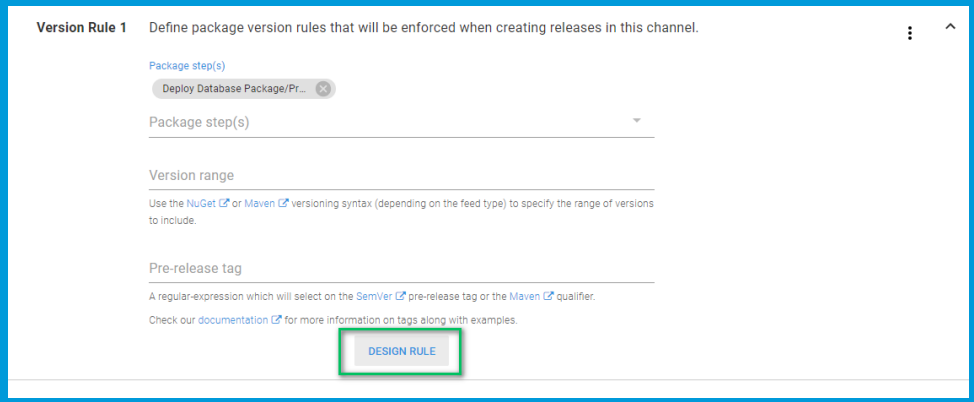


Adding the Package Version Rule

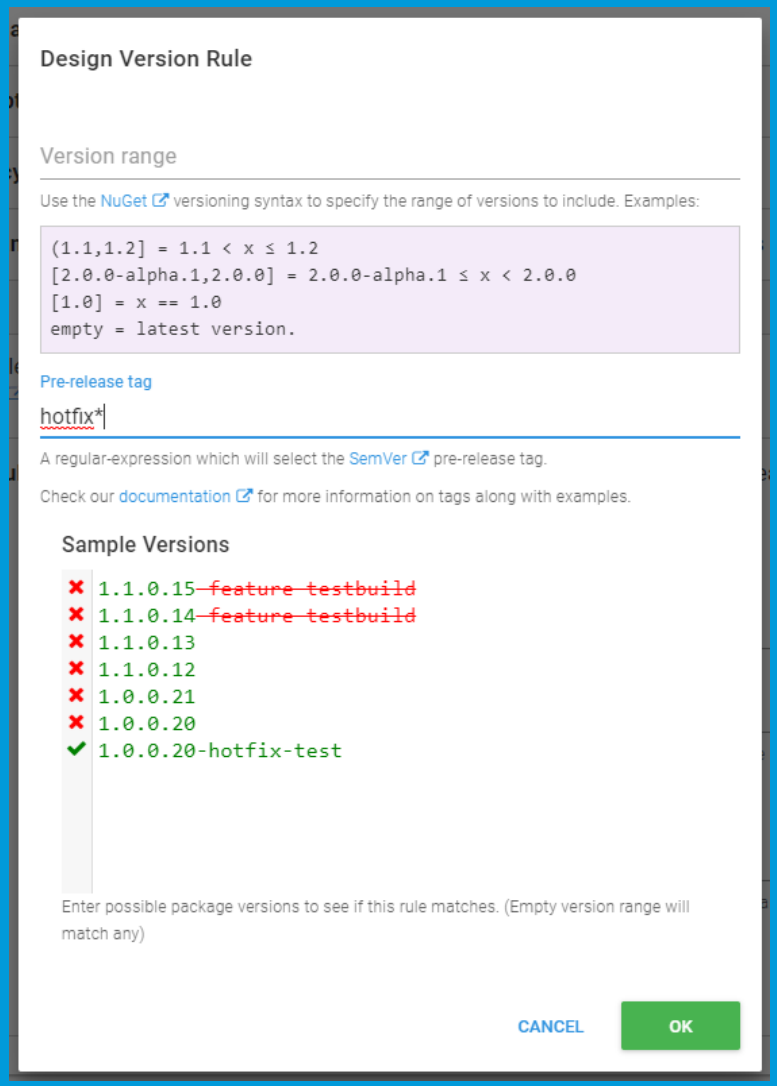
We highly recommend using either a different version or a pre-release tag to separate your standard releases from your hotfix releases. In this example, we're going to use a pre-release tag of "hotfix" to differentiate our packages. This means that our package will be named something like OctoFX.Database.1.0.40-hotfix.zip.

Usually, a hotfix branch is created from master, either the latest version or a tag that represents the last release, and merged back to master after the hotfix release is complete. This is so that you don't accidentally release new changes that weren't previously on production. We'll also have a separate build for hotfixes that can create the package with the hotfix tag. For now, you can download the latest version of your package, add -hotfix to the name, and upload that as a new version.

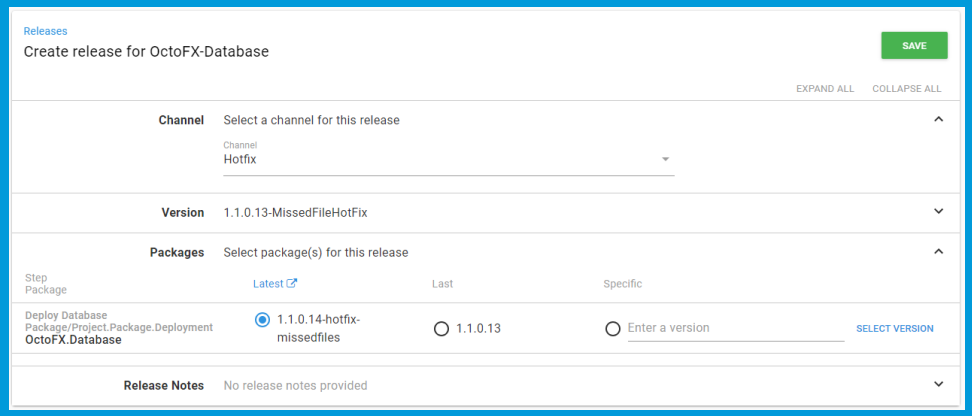
Now, let's go back and edit our Hotfix channel. In the Version Rules section, click "Add Version Rule." Choose "Deploy Database Changes" as the Package step. Click on the Design Rule button.



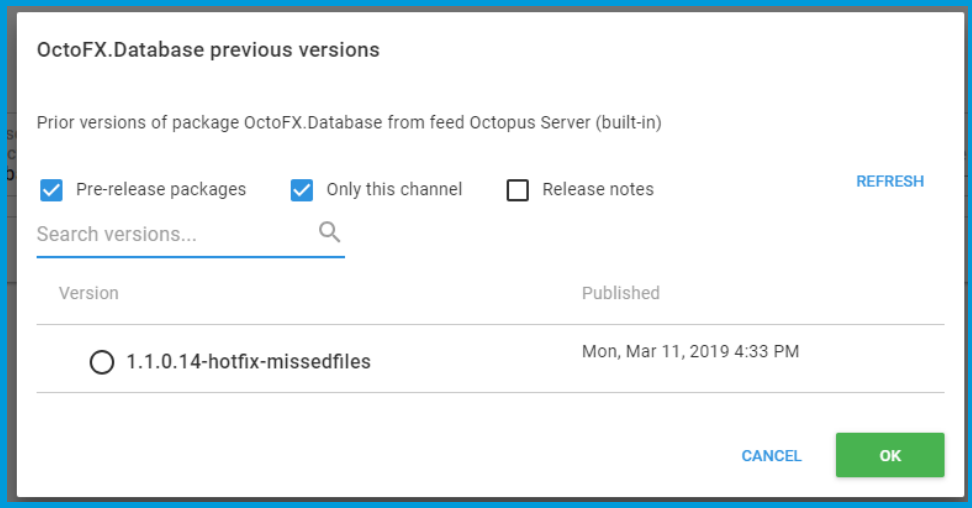
When the modal window loads, ignore Version range for now. The version range is handy when you are releasing a newer version of your software, say version 2 versus version 1. The version rule will limit packages with that version to their own channel with scoped steps and variables. Set the Pre-release tag to **`hotfix*`** and click Save.



Time to create another release. Expand the Channel and the Packages sections. When you choose the Hotfix channel, you should see the version of the package change to the one with the prerelease tag.



Clicking on the select version button will load a modal window with only packages versions which match that version rule.



You can override this, but by default, your standard packages can't be used for hotfix releases, and your hotfix packages can't be used for regular releases.

Conclusion

In this chapter, we covered how to set up a new channel for your project that uses a different lifecycle and different packages from your mainstream releases. Now you're set up to handle emergencies with deployments from Octopus!

16

STOPPING YOUR DEVELOPERS FROM DEPLOYING TO PRODUCTION

In our time at Octopus Deploy we have seen a lot of different team dynamics and security requirements. Some companies are okay with developers deploying to development, testing and staging. While other companies only allow their developers to deploy to development. QA deploys to testing, and the web admins deploy to staging and production.

Rather than try to create a one size fits all security model, Octopus Deploy took the approach of fine-grained security. We wanted to give you as much control as reasonably possible in defining your security module. Like everything else with Octopus Deploy, this can be a double-edged sword. In some cases, people attempt to use Octopus Deploy to solve a people problem rather than a technical problem.

It would be impossible for this book to cover every possible security scenario. The goal of this chapter is to give you a set of guidelines for you to adapt to your own Octopus Deploy configuration.

Scenarios

We recommend applying the "trust but verify" approach to security with Octopus Deploy. You either trust your team, or you don't. Do not try to come up with a security solution to limit a subset of people, for example, your "cowboy developers," from making mistakes. You are going to end up creating a maintenance nightmare with very fine-grained permissions.

Personas or roles we will be creating on our Octopus Deploy instance.

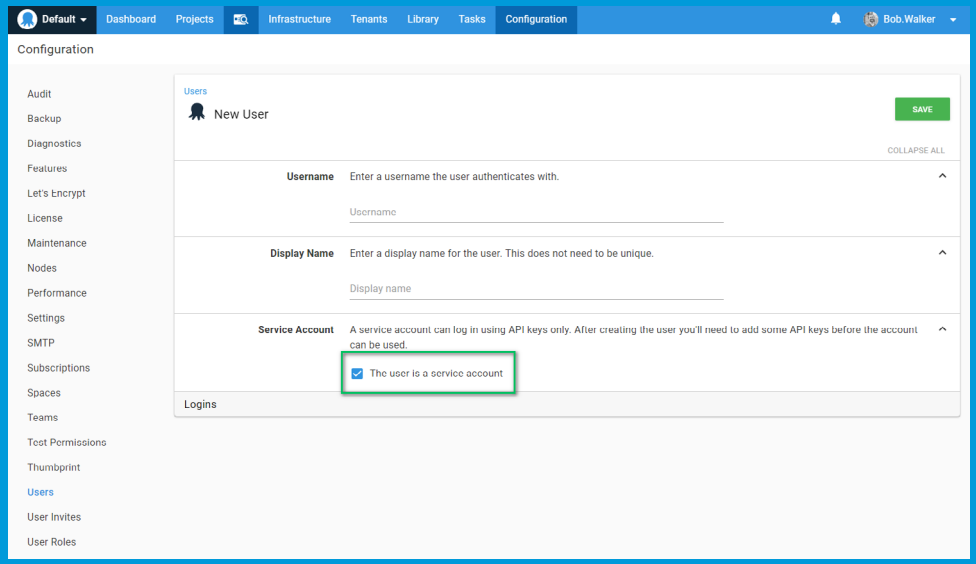
- **Octopus Administrator:** the group of in-house Octopus Deploy experts. They have unlimited rights in the system. They know with great power comes great responsibility.
- **Web Admins:** the group of people responsible for setting up new machines and ensuring everything is functioning correctly. Some of them are experts with Octopus Deploy while others are not. They are responsible for adding new deployment targets. They also trigger deployments to staging and production.
- **DBAs:** The group of people accountable for keeping all the databases up and running. For this demo, DBAs are assigned the manual intervention steps for database projects for staging and production.
- **Business Owners or project managers:** the group of people who are responsible for signing off on a release before it goes to production. They are assigned all non-DBA manual intervention steps for staging and production.
- Lead Developers or Technical Leads: The technical leader of the team. Typically has been with the company for several years and knows when something doesn't work they are the first ones triaging the issue. They have permissions to make changes to projects as well as deploy to Development, Testing, and Staging. They also have permissions to edit variable sets.
- Developers: They have permissions to change projects but cannot edit variable sets. Day to day interaction is focused on writing code and fixing bugs. When something goes wrong with a deployment, they typically will bring in the lead developer if they cannot figure out how to solve the issue in a few minutes.
- QA: QA test the application. They want control over their environment because their tests fail if they suddenly
- Build Servers: Service accounts used by build servers to push packages to Octopus Deploy and trigger deployments to development.
- Managers, VPs, and CTOs: They are not concerned with triggering builds. They want to be able to see the status of the deployments and if there are any significant problems.
- Auditors: These individuals do not trigger builds. Nor do they update projects or variables. They only want to see who deployed and approved a version of code to a specific environment on a specific date.

Service Accounts

Use service accounts for when you need an account for automation, such as build server integration or API interaction. Do not use regular user accounts for automation. Service accounts do not have passwords. They only have API Keys.

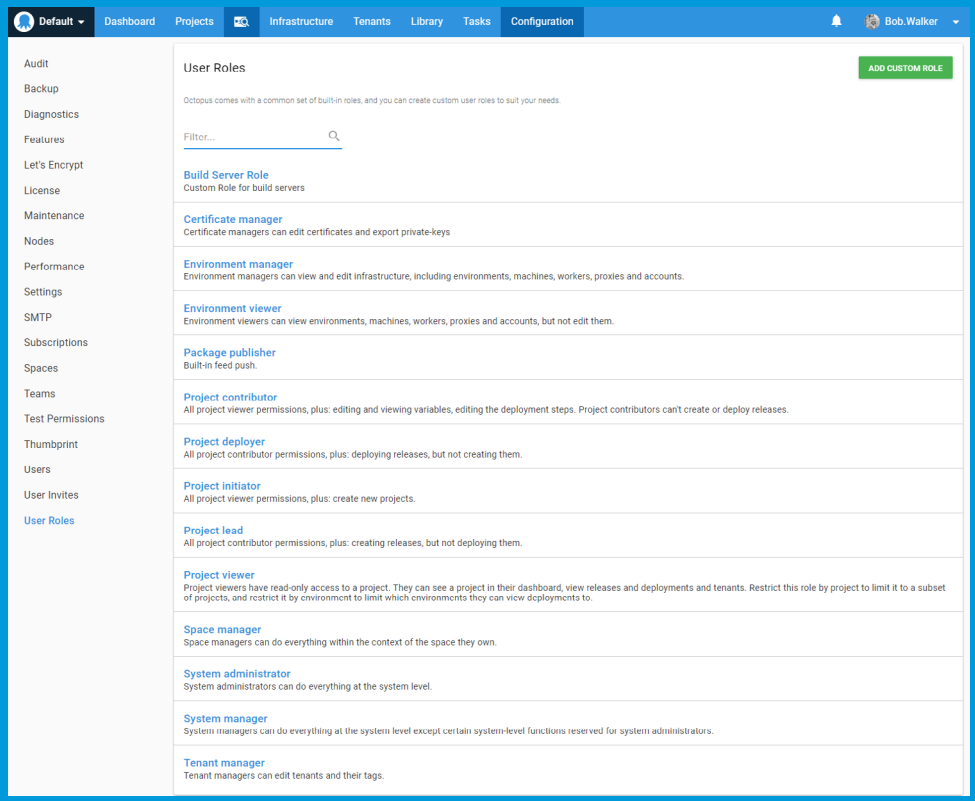
We recommend this for many reasons. Using a user account throws off auditing. All releases and deployments to development will appear as if it came from that user. If that user were ever to leave you would need to switch the keys over to someone else (typically when you create the service account).

Service accounts are set when you create a new user. Once a user is created they cannot be made into a service account. Once a service account is created they cannot be made into a user.



Built-In User Roles

Octopus Deploy comes with many built-in user roles to support these scenarios.



A team of people can be assigned to one or more roles. To support our scenarios, we will be assigning the following roles to the following teams.

- Everyone: Assign everyone project viewer role. By default, allow everyone to see projects and events but not edit them.
- Octopus Administrator: System administrator role
- Web Admins: Environment manager and project deployer
- DBAs: No additional rights needed (they have project viewer from

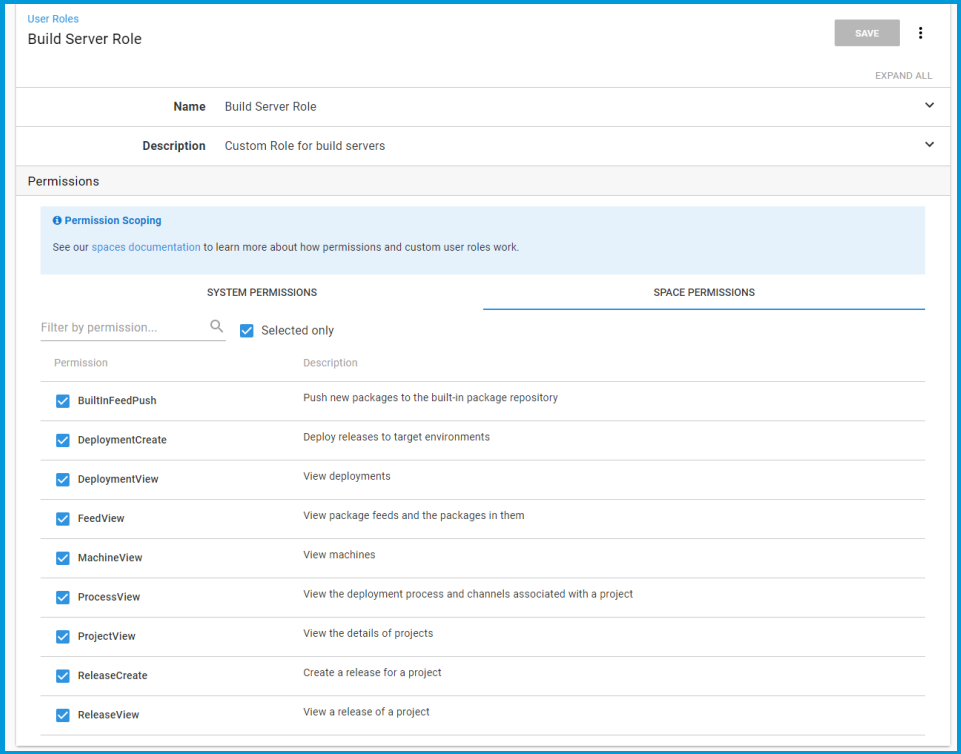
everyone)

- Business Owners or project managers: No other rights required (they have project viewer from everyone)
- Lead Developers or Technical Leads: Project lead for development, testing and staging environments, project contributor for production
- Developers: Project contributor role
- QA: Project deployer role restricted to the testing environment
- Build Servers: Custom role called “Build Server Role”
- Managers, VPs, and CTOs: No additional rights needed (they have project viewer from everyone)
- Auditors: No other rights required (they have project viewer from everyone)

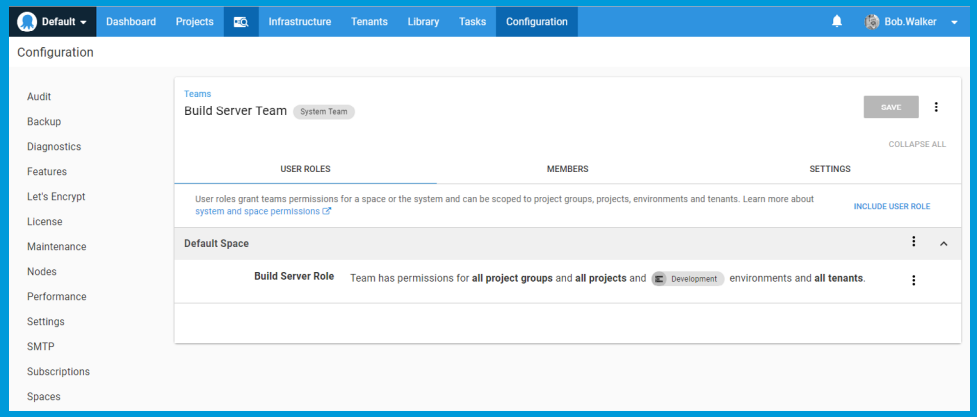
We typically don’t recommend creating your own custom role (except for the build server role below) as you could end up spending a lot of time trying to figure out why something isn’t showing up correctly.

Build Server Role

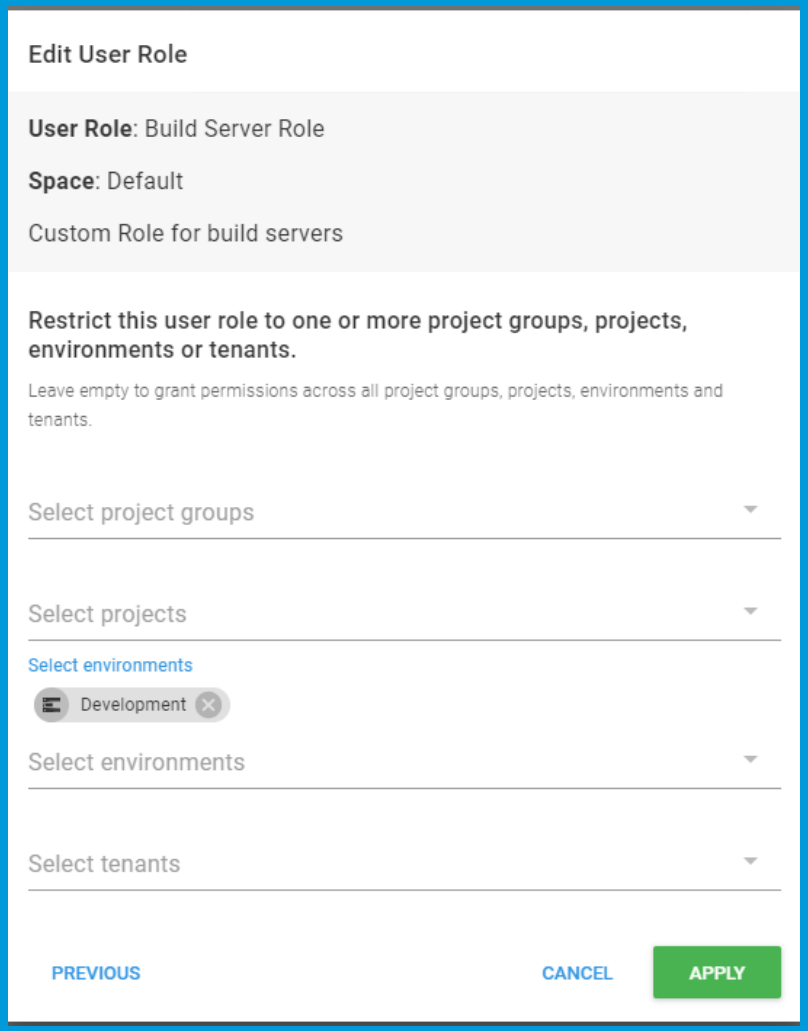
The build server role is a custom role which specific set of permissions. We want to limit the ability of this account only to publish packages and create releases. The view permissions are included, so the plug-in for the various build servers work.



When we create this team, we create it as a system team. We can restrict it to the development environment by setting the scope when assigning the role.

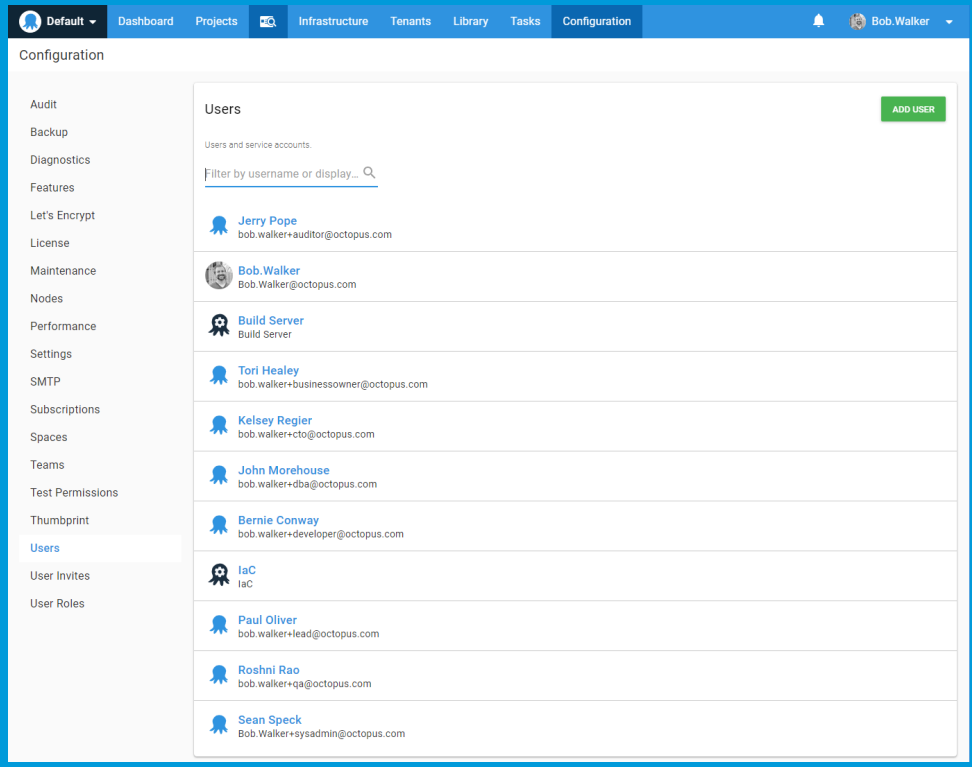


If you would like to edit that permission you can by clicking on the three . . . and making the changes on the modal window.



Users

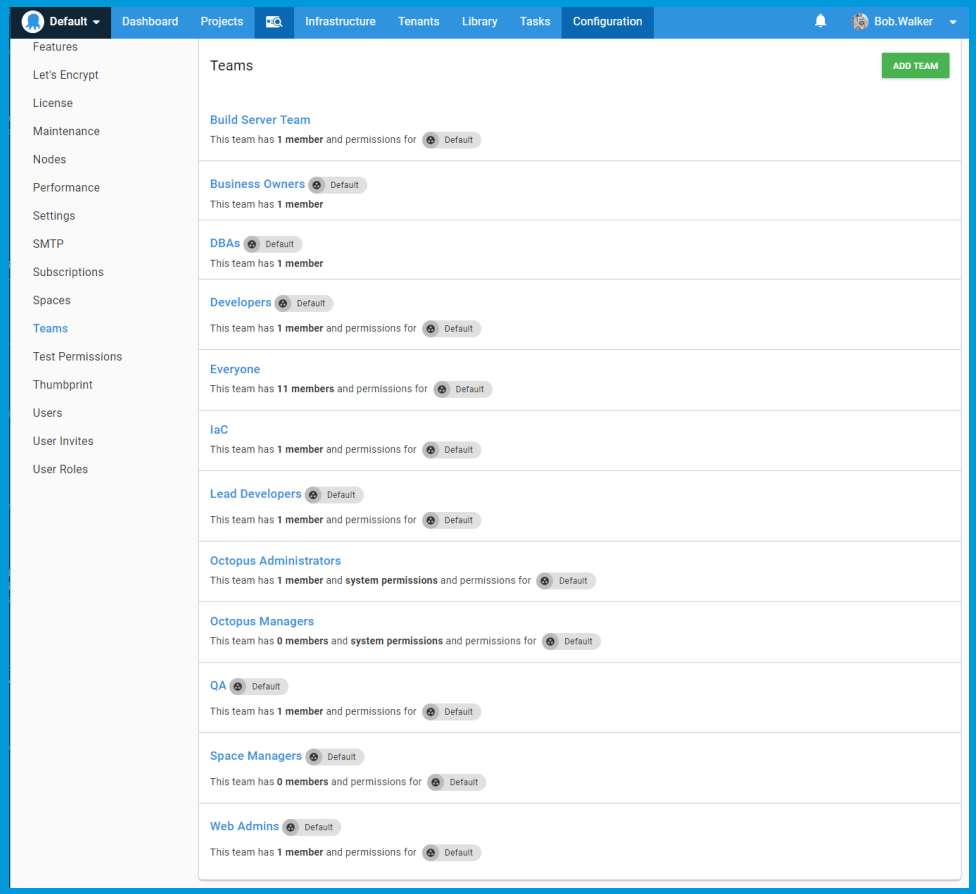
To help make things clearer for this demo, we have created many personas (based on real-life people) for each role.



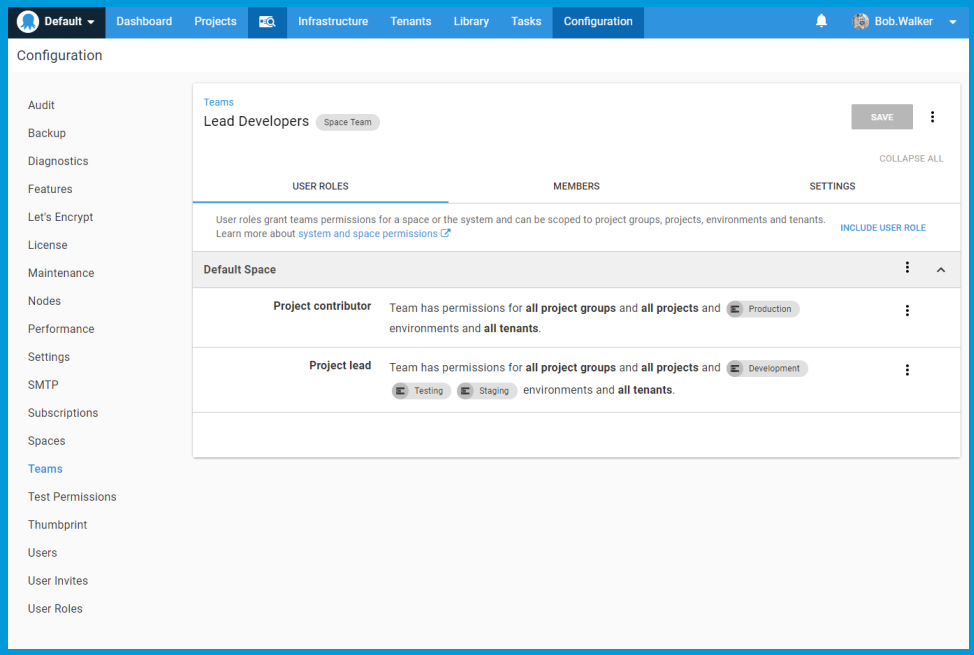
In the real world, you should set up external authentication, such as Active Directory, Okta, or Google Apps. External authentication will allow you to assign external teams to Octopus Deploy teams. When a new person joins your organization, they only need to be added to the external team to give them the necessary permissions in Octopus Deploy. Also, if the user's account is deactivated in the external provider, they can no longer log into Octopus Deploy.

Teams

Based on those scenarios and roles these are the teams created for this demo. You will notice that there is no Manager or Auditor team. That is because they already have project viewer rights. Besides, they are not going to be doing any approvals. There is no need to create a specific team for these scenarios.

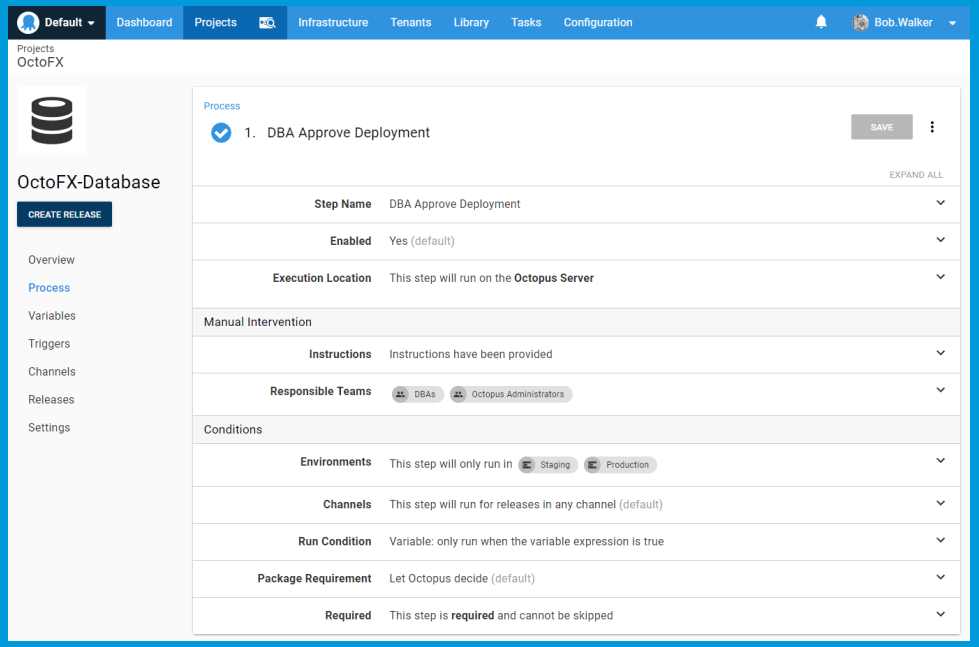


The lead developer team is a little unusual. It has rights to do quite a bit in development, testing and staging. However, it is limited in production. With the new team security structure introduced with the Spaces feature, you can now assign multiple roles to a single team.



Manual Interventions

With the above scenarios, we create a couple of teams to be used for approvals, DBAs, and Business Owners. The temptation is to limit manual interventions to only those teams. We recommend giving Octopus Administrators the ability to approve releases.



Octopus Administrators sometimes need to test new features. Having the ability to approve is critical. Alternatively, it is an emergency, and all the DBAs are out to a team lunch, and you need to get a deployment to production approved. The Octopus Admins should know what will happen when they click on **Approved**. They know it will be audited. They can tell and show anyone what was done after the emergency has passed.

Conclusion

Security in Octopus Deploy is very granular. Our recommendation is to go through each of the roles in your organization and match up appropriate permissions with them. At first, you will be making your best guess. Do not be afraid to iterate through your security until you get it dialed in. We do recommend granting the least permission as possible and then start opening it up.

SUBSCRIPTIONS

Who changed that project's deployment process? Who removed Bob from the administrator's group?

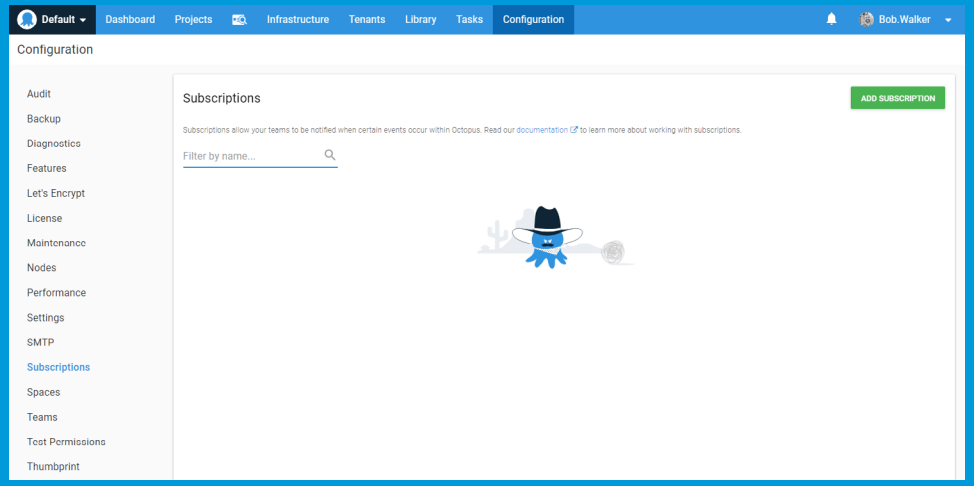
The Octopus audit log will tell you who did these actions which are useful after the fact, but wouldn't you rather know when specific changes happen?

If you answered "yes," then you're in luck! This is precisely what the Subscriptions feature is for.

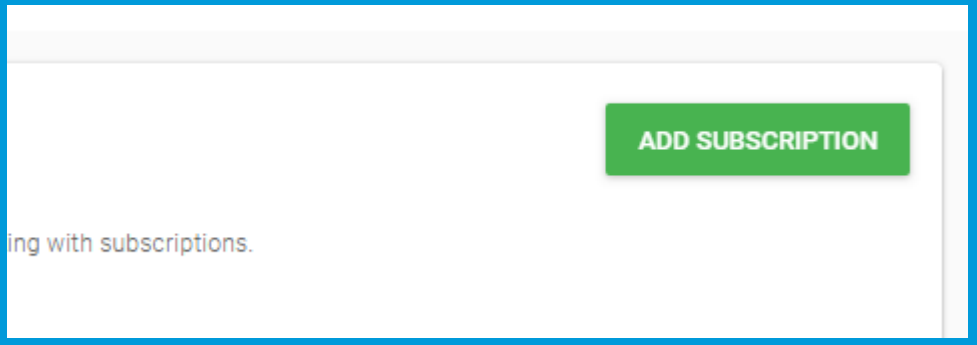
Let's create a subscription that will notify administrators whenever a production deployment starts. You can create an email step or Slack notification step in each project and scope it to the Production environment, but we can be notified about all production deployments with a single subscription.

Creating Subscriptions

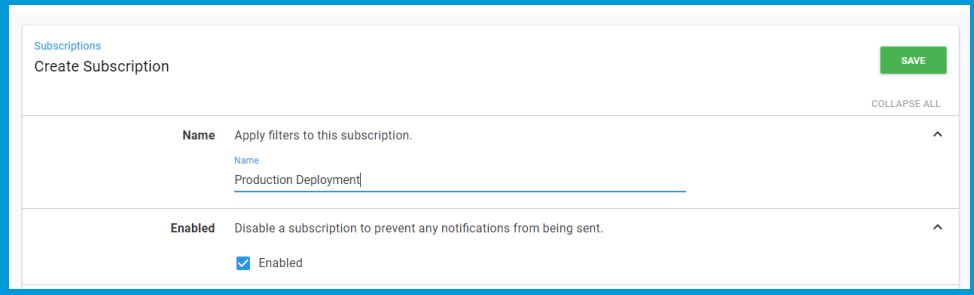
avigate over to Configuration > Subscriptions to see any existing subscriptions.



Click the Add Subscription button to create a new subscription.



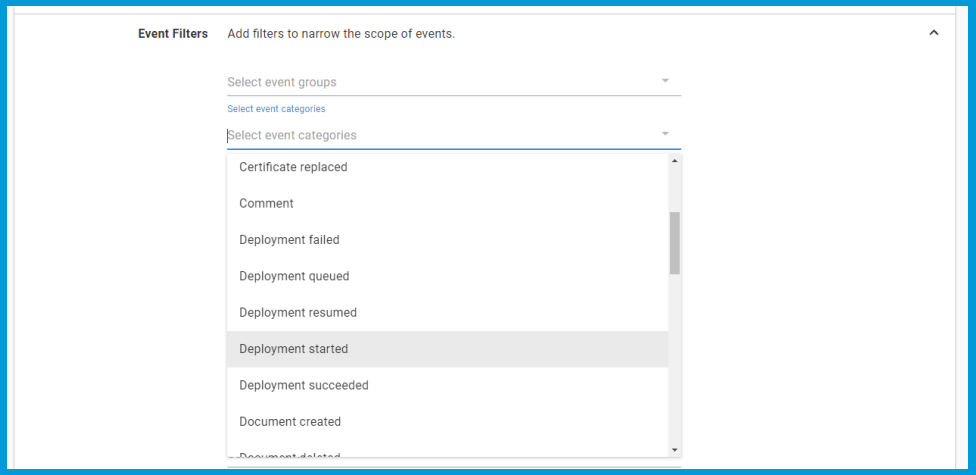
Let's give it a descriptive name like "Production Deployment." We'll leave it enabled since we want it to be active after we create it.



Event Filters

Now we need to create a filter that represents production deployments. The first drop down is the event group selection. Event groups are great if you want to filter to a known set of events. For example, **`machine critical-events`** includes **`machine cleanup failed`** and machine failed to be unavailable. We'll leave this blank since we're only looking for a single event.

We want to know when a production deployment starts, so click on **`select event categories`** and choose **`deployment started.`**

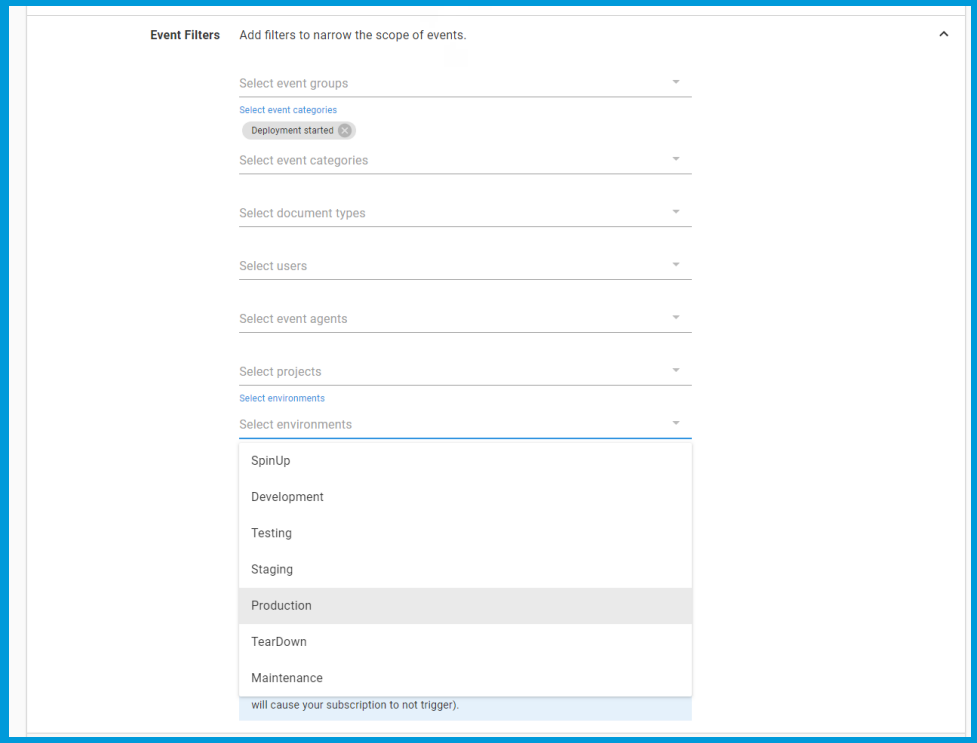


A document type is the type of record to watch. Examples are Project, Variable-Set, and Machine Policy. If you were to look in the database document types would almost match up with table names. Not exactly though, but close. For this particular subscription, we only care about deployments to production. No filter is needed here.

User filters allow you to filter certain actions for specific users. A useful example would be sending a notification if a service account user fails to log in. This filter isn't helpful for this particular subscription we are setting up. Leave that blank as well.

The project filter allows you to select specific projects. For this filter, maybe you only want to be notified when specific key projects go to production. This would be one of the first filters to look at when/if the subscription starts being too noisy.

The last piece for the filter is the environment. Click **`select environments`** and choose **`Production`**.

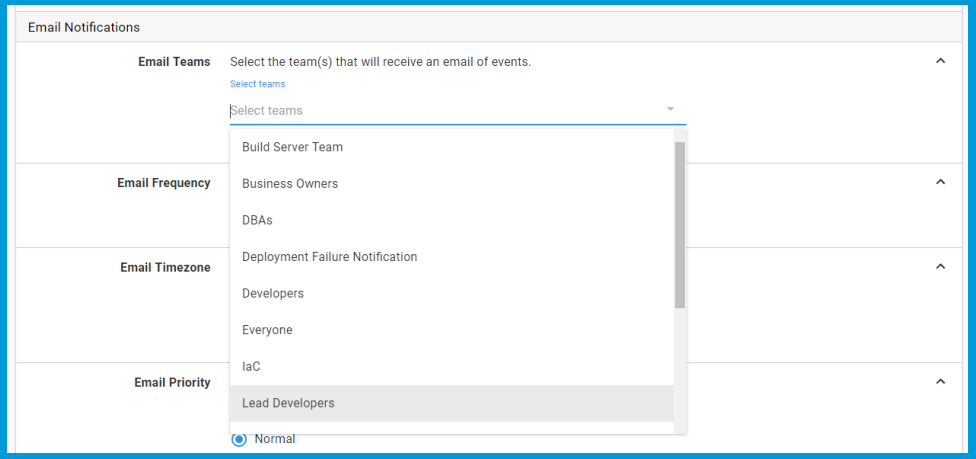


There are two options for how to deliver the notification, email or webhook.

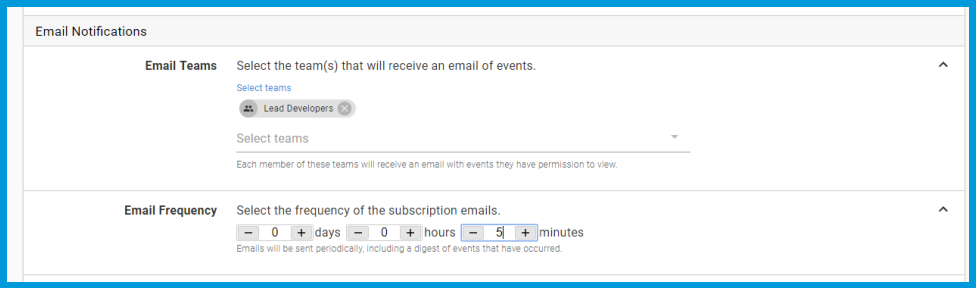
Email Notifications

We're going to set up this subscription with an email notification.

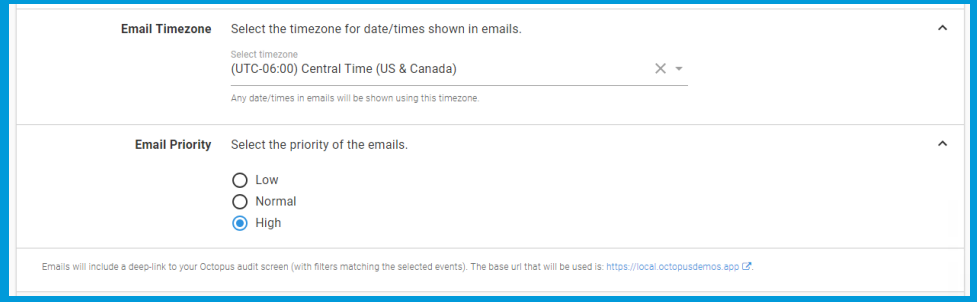
First, choose a team to receive the email. When the subscription fires, an email will be sent to each member of the team with the permissions to see that event.



Emails are sent at the frequency we choose. The default is one hour, so every hour this subscription will be evaluated, and a digest of all matching events will be sent out. One hour seems a bit long for monitoring production deployments, so let's turn that up to every five minutes.



Select your desired timezone and email priority. Because this is production, set the priority to high for this subscription.



Notice that small text at the bottom of the screenshot above. All the emails sent out will include a deep link to the audit screen. The deep link will allow the recipients to view the details of the deployment, such as who triggered it, when it was triggered, and other helpful information.

Webhook Notifications

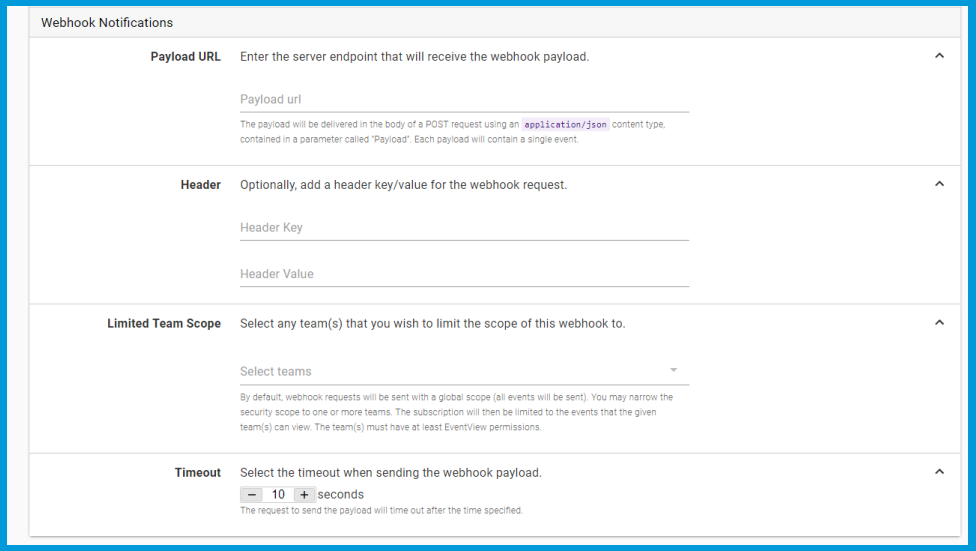
You can also configure the subscription to send the event payload to a webhook. We won't be configuring a webhook in this chapter, but we will cover the settings available.

The URL to an endpoint will accept the subscription payload. The payload will be delivered in the body of a POST request using an application/json content type, contained in a parameter called "Payload." Each payload will contain a single event.

You can also set an optional header to be sent with the request. Typically this is an authentication header for the application hosting the webhook.

By default, all events that match the filter will be sent to the webhook. You can limit this to events that match the permissions for one or more teams.

The last option is the timeout for the webhook request.



Conclusion

In this chapter, we introduced subscriptions and configured a subscription that will send us email notifications when releases are deployed to production.

SPACES.. WHAT ARE THEY GOOD FOR?

(almost everything!)

Spaces

Do you have multiple teams or departments using Octopus Deploy? Are you creating team specific environments or lifecycles to keep your teams as separate as possible? In the past, you might have also considered configuring a completely different Octopus server for your teams. Spaces might be the solution you didn't know you needed.

Spaces is a new way to organize your Octopus Server. They make it easy to group your projects, environments, and most other things into a space for each team. It's like moving teams from a large open plan office to private offices.

You might find Spaces useful if:

- You are continually scrolling through a long list of projects to find the one you are working on
- You have trouble finding the environments or tenants you're deploying to
- You see dropdowns with hundreds of items, and you struggle to know which option to pick

For Admins

It's not hard to create a good separation between teams in one server with some effort and configuration. It might lead to less than optimal practices like creating a new set of environments and lifecycles per team or product. Your teams will have nice views of Octopus; your Octopus Administrators will not be as lucky.

Octopus Administrators see everything in the Octopus server. They will see all of the projects, environments, tenants, and targets. That flexibility leads to highly populated pages that make it hard to find a specific item.

For Teams

Give Teams Their Own Space

Depending on the permissions you already have configured, your teams may not notice a change in their day to day usage of Octopus. Other configurations, where permissions haven't been configured, and each team has their projects and environments will see a big difference.

At the time of writing, all Octopus items are unique per space. The mantra of spaces is "hard walls." Hard walls mean that each Space has separate projects, environments, targets, tenants, templates, variable sets, and so on. In the future, some of these items may span across spaces, but for now, they are all unique per space and cannot be shared.

Put Team Leads in Control

Often, permissions are very locked down for teams that are sharing an Octopus server. Locking down makes sense, as you wouldn't want someone to accidentally remove the wrong machine from the production environment or change some variables used by another team. Most of the time, these restrictions are in place to keep one team from inadvertently breaking deployments for another team.

Locking down the Octopus Server can lead to bottlenecks. Teams may need to wait for their request for a new target or project to be added before they can continue their work.

When an Octopus Administrator creates a space, they will designate Space Managers, who can perform any administrative task required within the space.

Designated permissions are great because now you can put teams in control

of their own space. They will not have to wait to have changes or additions made, and you won't need to worry that they are affecting any other team spaces.

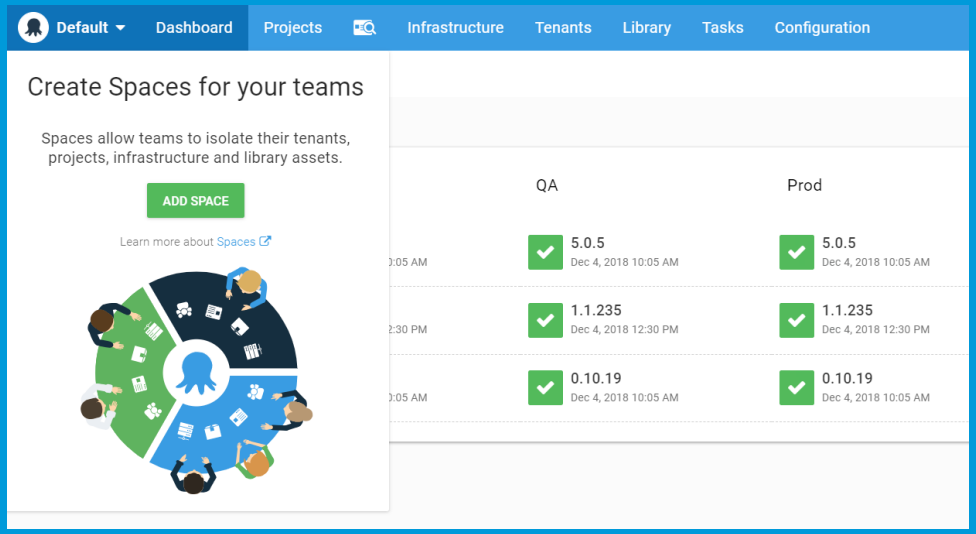
The Default Space

When you configure a new Octopus Server, it will come with a single space named "Default." Not only will it be named "Default," but it will be designated as the Default Space. The Default Space is important because if you are using any API requests that do not specify the Space Id, the server will assume that you are acting on the Default Space.

Everything we have done so far has been added to the Default Space. Let's look at how we can set up new spaces.

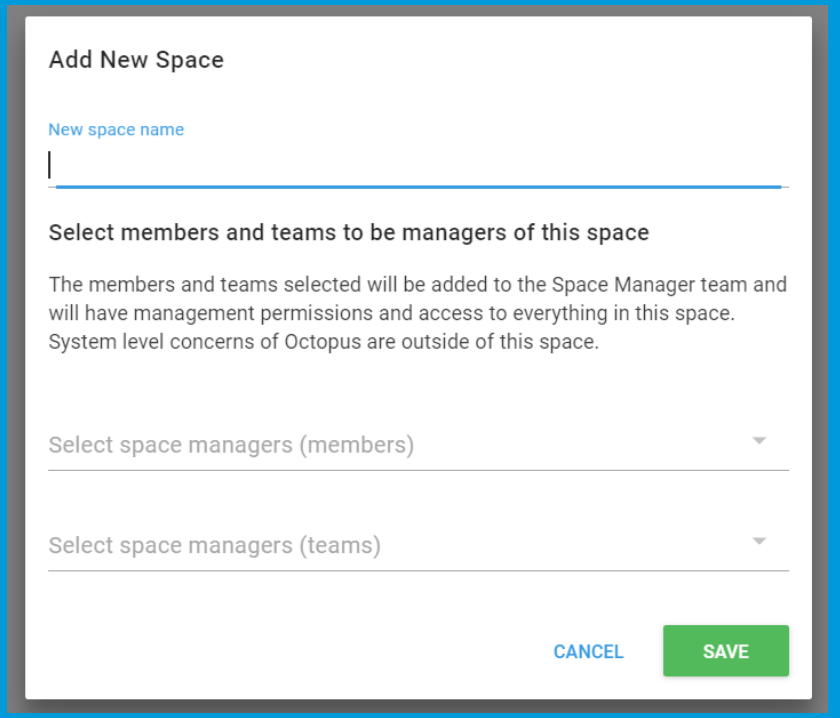
Switching Between Spaces

You can easily jump between spaces using the navigation bar. The leftmost item on the navigation bar displays the name of the current space. If you click on that name, you can choose to switch to one of your other spaces.



Creating Your First Space

Since we haven't created any other spaces yet, you will have the option to add a new space when you click on the space switcher. This button will take you to the Spaces page and from there you can click Add Space to bring up the Add Space Dialog.



Give the space a name and then choose the Space Managers. You can choose a combination of individual users or teams to be space managers. Not that if you do not choose yourself to be a manager, you won't be able to switch to the newly created space until one of the managers gives you access.

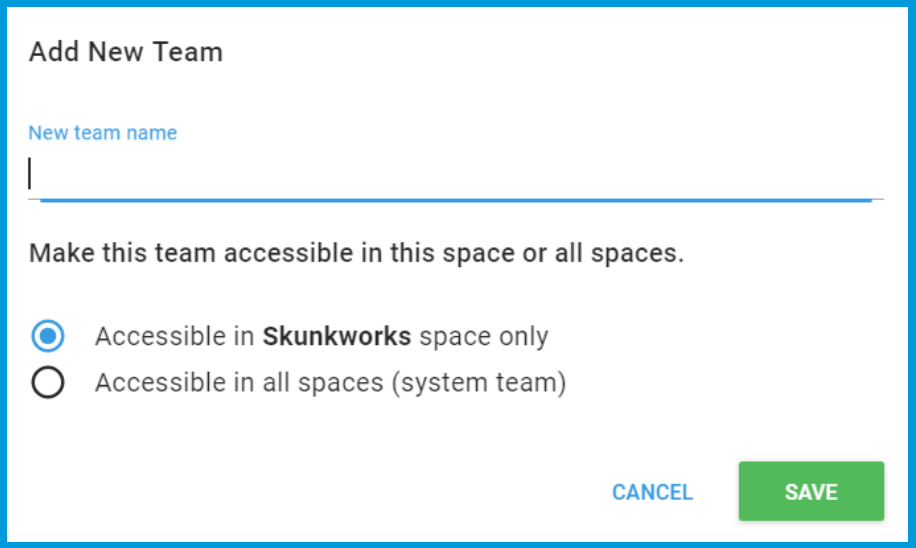
Once you've saved that space, you'll be taken to the space page. Here you

can change the name, managers, description, and also the logo. Yes! You can have a different logo for each space!

Space Teams and System Teams

First, can we start with just how fun it is to say Space Team?

Now that that is out of the way, let's give some users access to the space. Currently, only the space managers have access to the space. You will assign teams to the space to provide other users access.



The first type of team is a Space Team. Space teams exist only within the space it is created in and cannot be seen from other spaces. For example, you can create a Developers team within each space.

The second type of team is a System Team. System teams exist outside of spaces and can be given access to multiple spaces. Octopus Administrators is an example of a system team. You can also create your custom system teams. For example, you might build a QA team that has access to multiple spaces.

Conclusion

In this chapter, we introduced spaces, what they can do for your teams, and how to create and configure one.

19

SPINNING UP DEPLOYMENT TARGETS

and deploying them to automatically using Infrastructure as Code

Spinning up Deployment Targets and deploying to them automatically using Infrastructure as Code

The tentacle is an MSI you have to install on a VM. To get your POC going, you went the manual route. Download the MSI onto the VM, install it and configure it. Then you went back to the Octopus Deploy UI and registered the target with the Octopus Deploy server. For a few tentacles that works. Once you get above 25 or so machines, you realize that doesn't scale.

We recommend creating a process to automate the tentacle installation. If you are using Azure, you can leverage Azure Resource Manager Templates (ARM Templates). For AWS, you can leverage or CloudFormation to spin up new virtual machines. Both processes support running a PowerShell script to bootstrap them.

> CloudFormation templates allow you to include PowerShell in them. ARM templates require you to use the custom script extension. We recommend using Google to find the latest examples.

Meanwhile, if you are on-premise, you might be using hypervisor software such as Hyper-V or VMWare. Those have a robust API to script out spinning up a VM and bootstrapping them using a PowerShell script.

> We are not including samples on how to do this as each hypervisor is unique. If we tried to include scripts for every possible hypervisor and version, this chapter would end up being hundreds of pages long.

Regardless of the technology you are using, you will need a PowerShell script to Bootstrap the tentacle installation. Below is a sample script that we wrote for this book to bootstrap tentacles.

```
Param(
    [string]$octopusServerUrl,
    [string]$octopusApiKey,
    [string]$octopusServerThumbprint,
    [string]$instanceName,
    [string]$registrationName,
    [string]$environment,
    [string]$tenant,
    [string]$roles,
    [string]$machinePolicy
)

## This assumes you are installing on premise.
$IpAddresses = Get-NetIPAddress -AddressFamily IPv4

$ipAddress = “”
foreach ($address in $IpAddresses)
{
    $addressToCheck = $address.IPAddress

    # Change this to match against your own internal IP address range
    if ($addressToCheck -match “192.168.0.*”){
        $ipAddress = $addressToCheck
    }
}

Write-Host “Instance name: $instanceName”
Write-Host “Registration Name: $registrationName”
Write-Host “Environment: $environment”
Write-Host “Tenant: $tenant”
Write-Host “Roles: $roles”

Set-Location “${env:ProgramFiles}\Octopus Deploy\Tentacle”

$tentacleListenPort = 10933
Write-Host “Going to use port $tentacleListenPort”

Write-Output “Open port $tentacleListenPort on Windows Firewall”
& netsh.exe firewall add portopening TCP $tentacleListenPort “Octopus Ten-
tacle $instanceName”
if ($lastExitCode -ne 0) {
    throw “Installation failed when modifying firewall rules”
}

$tentacleHomeDirectory = “C:\Octopus\$instanceName”
$tentacleAppDirectory = “C:\Octopus\$instanceName\Applications”
$tentacleConfigFile = “C:\Octopus\$instanceName\Tentacle\Tentacle.config”

$rolesToRegister = $roles -split “,” | foreach { “--role `”$($_.
Trim())`”” }
$rolesToRegister = $rolesToRegister -join “ “
```

```
if ([string]::IsNullOrWhiteSpace($tenant) -eq $false){
    $tenantToRegister = “--tenant `”$tenant`””
}

& .\tentacle.exe create-instance --instance $instanceName --config $tenta-
cleConfigFile --console | Write-Output
if ($lastExitCode -ne 0) {
    throw “Installation failed on create-instance”
}
& .\tentacle.exe configure --instance $instanceName --home $tentacleHome-
Directory --console | Write-Output
if ($lastExitCode -ne 0) {
    throw “Installation failed on configure home directory”
}
& .\tentacle.exe configure --instance $instanceName --app $tentacleAppDi-
rectory --console | Write-Output
if ($lastExitCode -ne 0) {
    throw “Installation failed on configure app directory”
}
& .\tentacle.exe configure --instance $instanceName --port $tentacleLis-
tenPort --console | Write-Output
if ($lastExitCode -ne 0) {
    throw “Installation failed on configure port”
}
& .\tentacle.exe new-certificate --instance $instanceName --console |
Write-Output
if ($lastExitCode -ne 0) {
    throw “Installation failed on creating new certificate”
}
& .\tentacle.exe configure --instance $instanceName --trust $octopus-
ServerThumbprint --console | Write-Output
if ($lastExitCode -ne 0) {
    throw “Installation failed on configure trust with server”
}
& .\tentacle.exe service --instance $instanceName --install --start
--console | Write-Output
if ($lastExitCode -ne 0) {
    throw “Installation failed on service install”
}
$cmd = “& .\tentacle.exe register-with --instance `”$instanceName`”
--server $octopusServerUrl $rolesToRegister --environment `”$environ-
ment`” --name $registrationName $tenantToRegister --publicHostName $ip-
Address --apiKey $octopusApiKey --comms-style TentaclePassive --force
--console --policy=`”$machinePolicy`””
Write-Host $cmd
Invoke-Expression $cmd | Write-Host
if ($lastExitCode -ne 0) {
    throw “Installation failed on register-with”
}
```


> By automating the tentacle bootstrapping process, you also put yourself in a position to better handle increasing load. With automation, you can spin up a new server in a matter of minutes rather than hours or even days.

The bootstrap script can do so much more than install the tentacle. You can also leverage applications such as Chocolatey and built-in features such as DISM to install IIS, .NET Core, SQL Server Management Objects, and so on. Chocolatey is an application manager which allows you to install third-party applications in an automated fashion. DISM, or Deployment Image Servicing and Management, is built into Windows to allow you to enable or disable features. For example, if you wanted to automatically configure a VM to host a .NET core application this would be a script to do so.

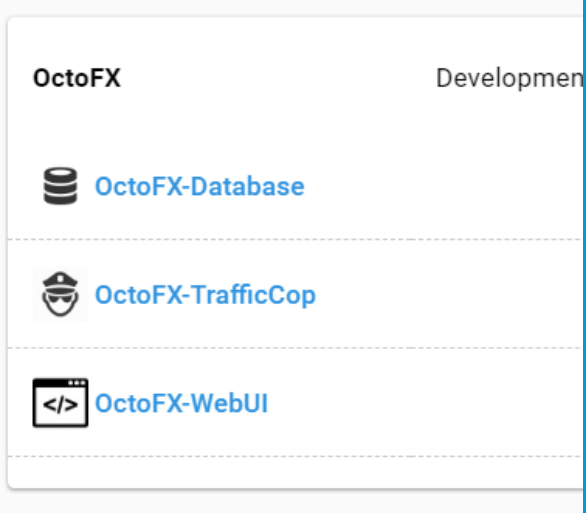
```
Write-Output “Installing ASP.NET 4.5”
Dism /Online /Enable-Feature /FeatureName:IIS-ASPNET45 /All | Write-Out-
put

Write-Output “Installing CertProvider”
Dism /Online /Enable-Feature /FeatureName:IIS-CertProvider /All |
Write-Output

Write-Output “Installing IIS Management”
Dism /Online /Enable-Feature /FeatureName:IIS-ManagementService /All |
Write-Output

Write-Output “Installing Chocolatey”
Set-ExecutionPolicy Bypass -Scope Process -Force; iex ((New-Object Sys-
tem.Net.WebClient).DownloadString(‘https://chocolatey.org/install.ps1’))

Write-Output “Installing .NET Core”
choco install dotnetcore-windowshosting -y
```

> For other deployment target types (Kubernetes, SSH, Azure Web Apps, etc), you should leverage the Octopus Deploy API to register the target. Octopus Deploy is an API first application. Everything you can do in the UI you can do in the API.

Project Triggers

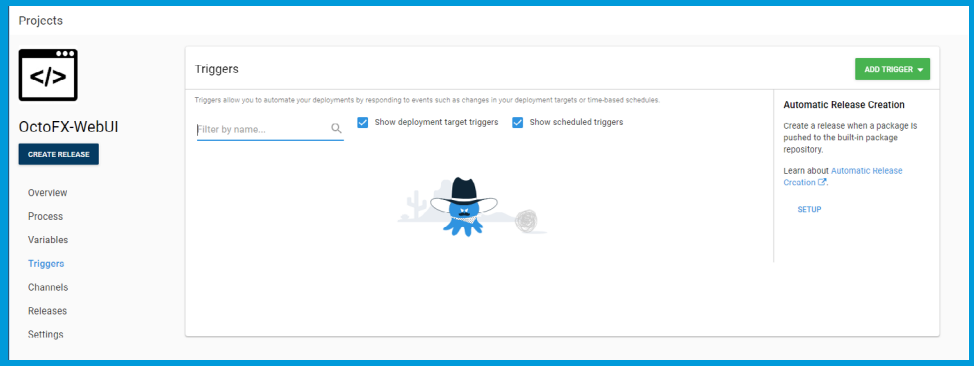
The creation of the deployment targets is now automated. The next step is to set up a project trigger which will see when a new machine is added for a specific role and then automatically deploy to that machine. This way the entire process is automated, from machine creation to deployment.

Before we get going on creating the triggers, let's take a quick step back and think about what this means. In the previous chapters, we set up three projects.

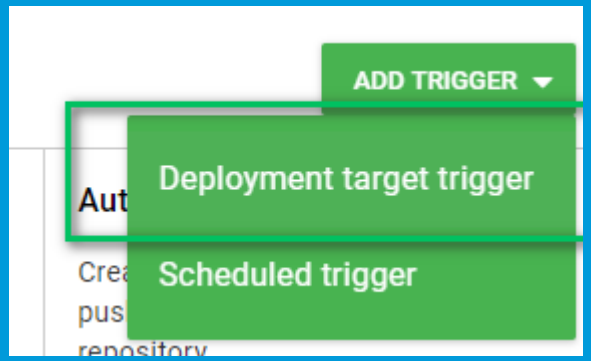


Of those three projects which one is most likely going to have a new machine added to it for scale? The OctoFX-WebUI project. If you add a new machine to a SQL Server cluster a DBA is required. There is quite a bit of work on the back-end for them to do. And you are not adding new machines willy-nilly to a SQL Cluster. But adding new machines into a web farm for the OctoFX-WebUI project is a lot more plausible. Maybe to handle some additional load. Maybe to replace an existing machine. OctoFX-WebUI is where we are going to add the trigger.

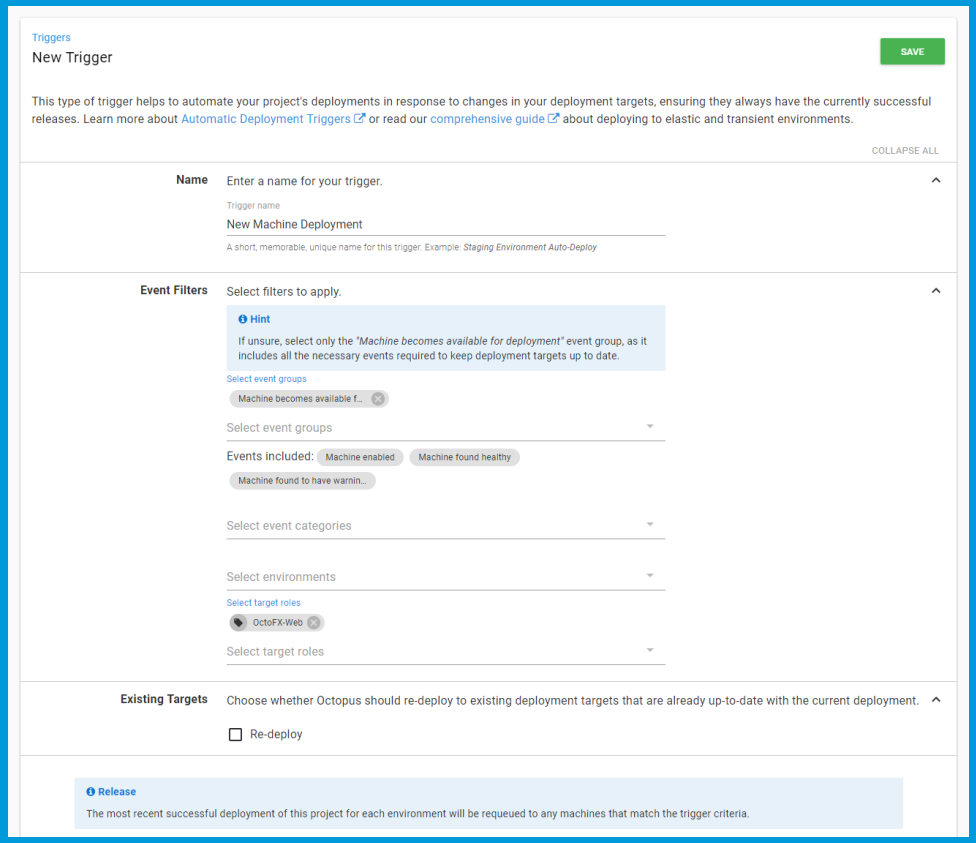
This is done by going to the project and selecting the triggers option on the left-hand menu.



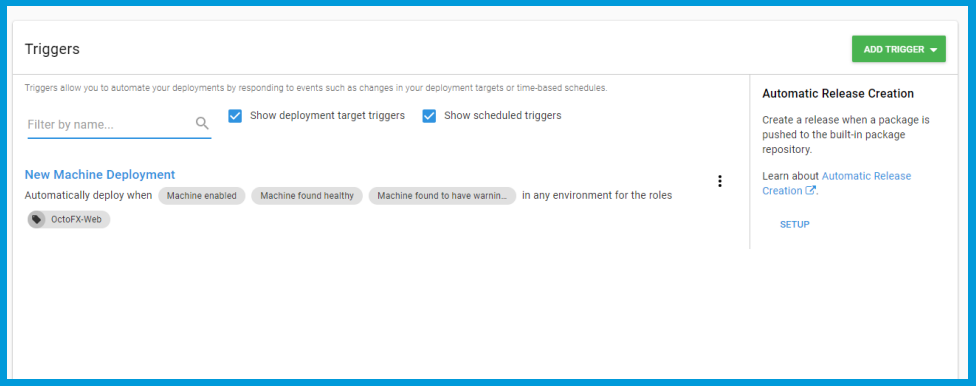
From here you are going to want to select the add triggers button in the top right corner of the screen and select deployment targets trigger.



On the next screen, you will want to the event, which should be "machine becomes available for deployment." Next select the machine role. This is also a great reason to have a specific role per project. It allows the trigger to monitor for machines it cares about.



And with a simple click of the save button, we have the trigger configured.



Conclusion

You can leverage technology to automatically spin up and down deployment targets. This can be done whether you are using a cloud provider such as AWS and Azure or if all your deployment targets are on-premise you running Hyper-V or VMWare. With deployment triggers, you can then tell Octopus Deploy to automatically deploy code when new machines come online. This allows you to scale up your application in a few minutes, and when you no longer need that extra horsepower, scale back down.

20

WHAT IS DATA CENTER, WHY DO I NEED IT

and when should I upgrade?

What is Data Center and why would I need it?

Everything we presented so far in this book can be done using the Standard Edition of Octopus Deploy. The standard license of Octopus Deploy provides quite a bit of functionality. You get unlimited users, projects, and workers. You get up to three unique instances of Octopus Deploy. You get 3 spaces per instance. There is a lot there. So...what exactly does Data Center provide?

Driving Force Behind Data Center

Octopus Deploy was originally designed to run on a single server. Our customers would scale up the resources dedicated to that server as they scaled up the number of projects, deployment targets, and users. But eventually, a breaking point was reached when it was impossible to scale up any further. There are only so many network connections and concurrent processes a computer can run.

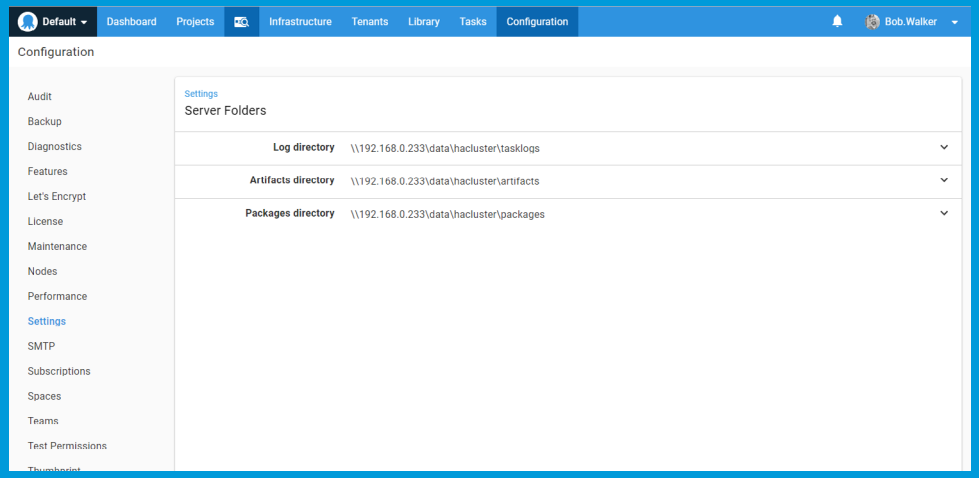
In addition, our customers had unwittingly created a massive single point of failure in their CD pipeline. If that server were to ever go down, for whatever reason (the most common cause was Windows Patches), that would bring all deployments to a grinding halt. When this only affected a team of 10-15 people it was an annoyance. When it affected an entire development shop of 100s of people this became a major problem.

Finally, a lot of our customers started treating their Octopus Servers like pets rather than chattel. The default configuration of Octopus Deploy didn't help much. All folders are stored on the C:\ drive. If that server were to ever crash then a lot of data would be lost. And if the master key was ever lost then getting it back up and running could take quite a while.

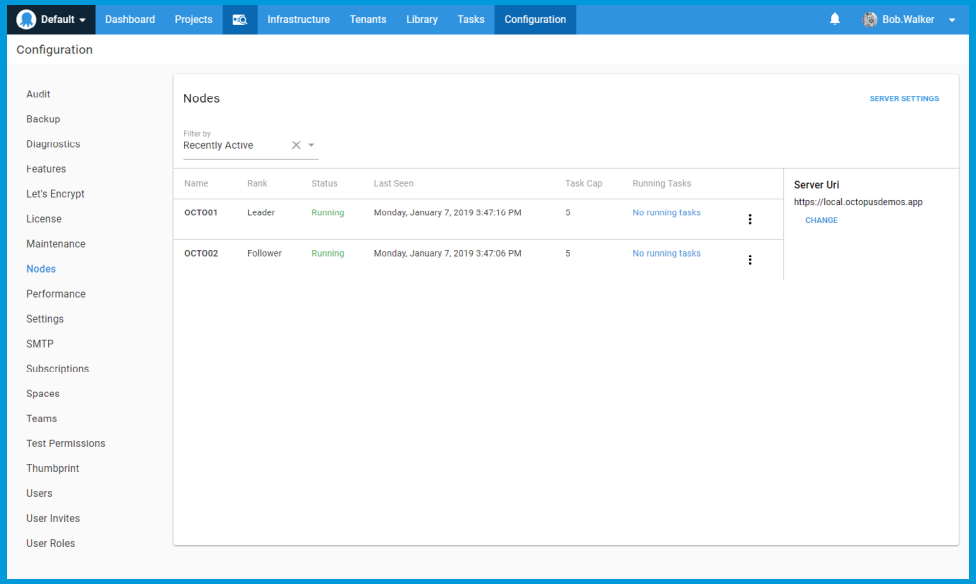
Data Center Overview

The standard edition of Octopus Deploy allowed you to have three (3) unique instances running. We determine uniqueness based on the database it is pointing to. With standard edition, you could only connect one instance to a database. Octopus Deploy Data Center allows you to connect several instances to the same database. The limit of three instances is gone. This allows you to create clusters of Octopus Deploy servers. The only kicker is the total number of targets, across all clusters must be below your license count. If you have a 1000 Machine Data Center license you can have 10 clusters with 100 targets each. Or, 1 cluster with 1000 targets.

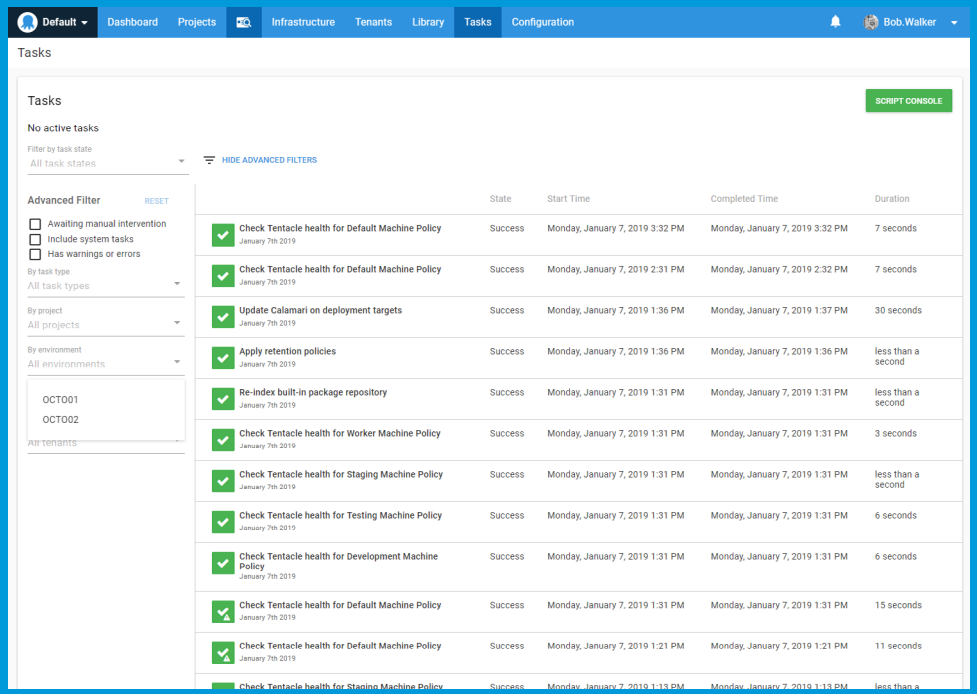
All the servers in the cluster use the same master key. They both point to the same database. All the task logs, artifacts, and packages must be on a file share which both servers can access. Typically the file share is on a NAS or on a DFS share.



The Octopus Servers use the database to coordinate tasks. Each server has its own task cap.



When a task is added into the queue to be processed one of the servers will pick it up. A task could be anything, a deployment, a health check, retention policy check, etc. When looking at the task log you can filter by a node to see which one is doing the work.



There are still some behind the scenes things only one server can do, such as processing subscriptions or scheduled tasks. This is why you see the label **`Leader`** and another label **`Follower`**. An Octopus Deploy Server cluster can only have one leader, but it can have many followers. If the leader is offline for an extended period of time, then one of the followers will become the leader.

Data Center Benefits

Data Center solves a number of the aforementioned issues. To begin with, you can have a cluster of multiple servers. This allows you to scale out the Octopus Deploy server horizontally as well as vertically. This reduces the IOPS required when using a single server to deploy to thousands of machines.

The Octopus Deploy UI is stateless. This means you can put a load balancer

in front of the Octopus Deploy cluster. This brings us to the next point. You no longer have a single point of failure. If a restart were to occur on one of the nodes, the other servers would keep on humming along and handle the deployments. The UI would continue to function as well (assuming it is behind a load balancer).

Because the important file directories are in a shared location, this makes it easier to spin up new Octopus Deploy Servers. When it comes time to upgrade the version of Windows, you can simply stand up a new Windows server and install Octopus Deploy on it. You would only have to point some configuration entries at the new shared file location and database.

Finally, the Data Center license is the top tier license. Of the functionality Octopus Deploy provides, it has the highest limits possible. Standard Edition can have 3 spaces while Data Center gets unlimited spaces. Standard edition gets 3 unique instances while Data Center gets unlimited instances. Standard edition only gets 1 node per database while Data Center gets unlimited nodes per database.

Conclusion

Octopus Standard Edition is great when you are getting started with Octopus Deploy. We've found that once customers hit 300 machines or so they move from beginners to advanced users. They also start running into the initial limits, such as the 5 concurrent task cap. Because they don't have the ability to scale horizontally they start to scale vertically. This will allow you to configure Octopus Deploy to scale much earlier than before.

Some good indicators you need Data Center are you have more than 100 machines, you need to have more than 5 concurrent tasks running, you are doing hundreds of deployments a day, or you worry about the single point of failure in the CD pipeline.

21

OCTOPUS PERFORMANCE & MAINTENANCE 101

(SQL Server Maintenance, Machine Policies, etc)

Octopus are generally hygienic creatures; they clean up after themselves. Octopus Deploy is not different. It does its best to clean up after itself. However, it still needs your help. In this chapter we will walk through some common tweaks you can make to Octopus Deploy to keep it running lean and mean.

Routine Maintenance

Octopus Deploy works best when regular maintenance is performed. Routine maintenance can help clear up the "cruft" left behind by old deployments.

Retention Policies

We cannot stress this enough. Please set your retention policies. Retention policies will clean old releases from your server and on the tentacles. The one exception to this is the Events table which records an audit trail of every significant event in your Octopus. We have an entire chapter in this book devoted to retention policies. Please read that.

Upgrade

It's critical to keep Octopus up to date. We are continually working to ensure that Octopus performs to a high standard. We will always recommend upgrading to the latest LTS version of Octopus Deploy. We have written an entire chapter in this book about upgrading Octopus Deploy. Please refer to that for more details.

SQL Server Maintenance

Starting with the first LTS release, 2018.10.x, any upgrade to Octopus Deploy will automatically rebuild fragmented indexes and regenerate stats. That helps. However, it is vital to rebuild fragmented indexes on a regular interval. Work with a DBA to set up a SQL job to rebuild indexes and regenerate stats every week.

Don't forget to back up the Octopus Deploy database. A good strategy is to do a full backup once a week, a differential backup once a day and a transaction log backup once every 10-15 minutes. If the SQL Server hosting the Octopus Deploy database were ever to crash you would only lose a few minutes worth of work.

Scaling Octopus

The Octopus Deploy server does quite a lot of work during deployments, mostly around package acquisition:

- Downloading packages from the package source (network-bound).
- Verifying package hashes (CPU-bound).
- Calculating deltas between packages for delta compression (I/O-bound and CPU-bound).
- Uploading packages to deployment targets (network-bound).
- Monitoring deployment targets for job status, and collecting logs.

At some point, your server hardware is going to limit how many of these things a single Octopus Server can do concurrently. There are only so many CPUs you can allocate and only so many network connections which can be opened. If a server overcommits itself and hits these limits, timeouts (network or SQL connections) will begin to occur, and deployments can begin to fail. Above all else, your deployments should be repeatable and reliable.

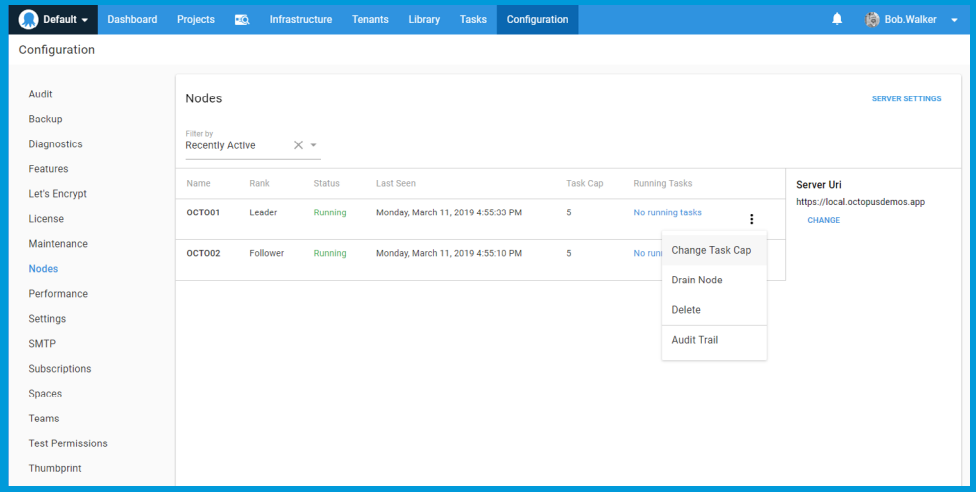
We offer three options for scaling your Octopus Server:

- Scale up by controlling the task cap and providing more server resources as required.
- Scale out using Octopus High Availability.
- Scale out using Workers.
- Scale out using Spaces.

Task Caps

An ideal situation would be an Octopus Server that's performing as many parallel deployments as it can while staying just under these limits. We tried several techniques to throttle Octopus Server automatically, but in practice, this kind of approach proved to be unreliable.

Instead, we decided to put this control into your hands, allowing you to control how many tasks each Octopus Server node will execute concurrently. This way, you can measure server metrics for your deployments, and then increase/decrease the task cap appropriately. Administrators can change the task cap in Configuration > Nodes.



The default task cap is set to 5 out of the box. Based on our load testing, this offered the best balance of throughput and stability for most scenarios. The highest we'd recommend you set the task cap to is 20. Anything more and you will start running into resource contention.

Leverage Workers

Workers are a new feature added to Octopus Deploy in version 2018.7.x. They allow you to offload several tasks typically done by the Octopus Deploy server onto a series of worker pools. Please see the chapter on workers for more details.

Octopus High Availability

You can scale out your Octopus Server by implementing a High Availability cluster. In addition to linearly increasing the performance of your cluster, you can perform certain kinds of maintenance on your Octopus Servers without incurring downtime. We have written a chapter about High Availability; please see that for more details.

Tips

Follow these tips to tune and maintain the performance of your Octopus:

1. Configure your Octopus Server and SQL Server on separate servers.
2. Avoid hosting your Octopus Server and its SQL Database on shared servers to prevent Octopus or the other applications from becoming "noisy neighbors".
3. Provide sufficient resources to your Octopus Server and SQL Server. Measure resource utilization during typical and/or heavy load, and decide whether you need to provision more resources.
    - We don't provide a one-size-fits-all set of specifications - your resource requirements will vary heavily depending on your specific scenario. The best way to determine what "sufficient resources"

means, is to measure then adjust until you are satisfied.

4. Configure short retention policies. Less history == faster Octopus.
5. Maintain your SQL Server.
   - See above.
   - Upgrade to the latest version of Octopus Server.
   - Quite often negative performance symptoms are caused by outdated statistics or other common SQL Server maintenance problems.
6. If you have saturated your current servers you may want to consider scaling up, by increasing the resources available to the Octopus and SQL Servers, or scaling out:
   - Consider Octopus High Availability if you are reaching saturation on your current infrastructure, or want to improve the uptime of your Octopus Server, especially across Operating System patches. Octopus High Availability is designed to scale linearly as you add nodes to your cluster.
   - Consider using Workers and worker pools if deployment load is affecting your server. See this blog post for a way to begin looking at workers for performance.
   - Consider separating your teams/projects into “spaces” using the upcoming Spaces feature.
7. Try not to do too much work in parallel, especially without thorough testing. Performing lots of deployment tasks in parallel can be a false economy more often than not:
   - You can configure how many tasks from the task queue will run at the same time on any given Octopus Server node by going to Configuration > Nodes. The default task cap is 5 (safe-by-default). You can increase this cap to push your Octopus to work harder.
   - Learn about tuning your deployment processes for performance.
8. Consider how you transfer your packages:
   - If network bandwidth is the limiting factor, consider using delta compression for package transfers.
   - If network bandwidth is not a limiting factor, consider using a custom package feed close to your deployment targets, and download the packages directly on the agent. This alleviates a lot of resource contention on the Octopus Server.
   - If Octopus Server CPU and disk IOPS is a limiting factor, avoid using delta compression for package transfers. Instead, consider downloading the packages directly on the agent. This alleviates a lot of resource contention on the Octopus Server.
9. Consider the size of your packages:
   - Larger packages require more network bandwidth to transfer to your deployment targets.
   - When using delta compression for package transfers, larger packages require more CPU and disk IOPS on the Octopus Server to calculate deltas - this is a tradeoff you can determine through testing.
10. Consider the size of your Task Logs:
    - Larger task logs put the entire Octopus pipeline under more pressure.
    - We recommend printing messages required to understand progress and deployment failures. The rest of the information should be streamed to a file, then published as a deployment artifact.
11. Prefer Listening Tentacles or SSH instead of Polling Tentacles wherever possible:
    - Listening Tentacles and SSH place the Octopus Server under less load.
    - We try to make Polling Tentacles as efficient as possible, but by their very nature, they can place the Octopus Server under high load just handling the incoming connections.
12. Reduce the frequency and complexity of automated health checks using machine policies.
13. Disable automatic indexing of the built-in package repository if not required.

Troubleshooting Common Issues

The best place to start troubleshooting your Octopus Server is to inspect the Octopus Server logs. Octopus writes details for common causes of performance problems:

1. Request took 5123ms: GET {correlation-id}: If HTTP requests are taking a long time to be fulfilled, you'll see a message like this. The timer is started when the request is first received, ending when the response is sent. Look for trends as to which requests are taking a long time. Look to see if the performance problem occurs, and goes away, regularly. This can indicate another process hogging resources periodically.
2. The dashboard or project overview is taking a long time to load: long retention policies usually cause this. Consider tightening up your retention policies to keep fewer releases. It can also be caused by the sheer number of projects you are using to model your deployments.
3. {Insert/Delete/Update/Reader} took 8123ms in transaction '{transaction-name}': If a particular database operation takes a long time you'll see a message like this. The timer is started when the operation starts, ending when the operation is completed (including any retries for transient failure recovery).
    - If you are seeing these operations take a long time it indicates your SQL Server is struggling under load, or your network connection from Octopus to SQL Server is saturated.
    - Check the maintenance plan for your SQL Server. See the tips above.
    - Test a straightforward query like SELECT * FROM OctopusServerNode. If this query is slow, it indicates a problem with your SQL Server.
    - Test a more complex query like SELECT * FROM Release ORDER BY Assembled DESC. If this query is slow, it indicates a problem with your SQL Server or the sheer number of Releases you are retaining.
    - Check the network throughput between the Octopus Server and SQL Server by trying a larger query like SELECT * FROM Events.
4. Task Logs are taking a long time to load, or your deployments are taking a long time: The size of your task logs might be to blame. See the tips above. Make sure the disks used by your Octopus Server have sufficient throughput/IOPS available for processing the demand required by your scenario. Task logs are written and read directly from disk.
5. If you are experiencing overly high CPU or memory usage during deployments which may be causing your deployments to become unreliable:
    - Try reducing your Task Cap back towards the default of 5 and then increase progressively until your server is reliable again.
    - Look for potential performance problems in your deployment processes, especially:
    - Consider how you transfer your packages.
    - Consider reducing the amount of parallelism in your deployments by reducing the number of steps you run in parallel or the number of machines you deploy to in parallel.

6. System.InvalidOperationException: Timeout expired. The timeout period elapsed prior to obtaining a connection from the pool. This may have occurred because all pooled connections were in use and max pool size was reached.: This error indicates two possible scenarios:
    - Your SQL Queries are taking a long time, exhausting the SQL Connection Pool. Investigate what might be making your SQL Queries take longer than they should and fix that if possible - see earlier troubleshooting points.
    - If your SQL Query performance is fine, and your SQL Server is running well below its capacity, perhaps your Octopus Server is under high load. This is perfectly normal in many situations at scale. If your SQL Server can handle more load from Octopus, you can increase the SQL Connection Pool size of your Octopus Server node(s). This will increase the number of active connections Octopus is allowed to open against your SQL Server at any point in time, effectively allowing your Octopus Server to handle more

concurrent requests. Try increasing the Max Pool Size in your SQL Connection String in the Octopus.Server.config file to something like 200 (the default is 100) and see how everything performs. Learn about Connection Strings and Max Pool Size.

- Octopus is leaking SQL Connections. This should be very rare but has happened in the past, and we fix every instance we find. We recommend upgrading to the latest version of Octopus and get help from us if the problem persists.

Don't be afraid to get in touch!

If none of these troubleshooting steps work, please get in contact with our support team and send along the following details (feel free to ignore points if they don't apply):

1. Are you running Octopus as an HA cluster or single node?
2. Is the SQL Database Server on the same machine as Octopus or a different machine?
3. Are you hosting any other applications on the same machine as Octopus or its SQL database?
4. What kind of server specs are you running for Octopus and SQL Server?
5. Approximately how many users do you have using Octopus?
6. Approximately how many projects and machines do you have?
7. Approximately how many deployments do you perform at the same time?
8. Do you notice any correlation between deployments of specific projects and the performance problem?
9. Do you notice any correlation between other Octopus Server tasks (like package retention policy processing) and the performance problem?
10. Does the Octopus Server ever become unresponsive and how frequently does it become unresponsive?
11. Does the Octopus Server recover after the performance degrades, or does it need to be manually restarted to recover?

In addition to answering those questions, please collect and attach the following diagnostics to your support request (probably the most important part):

1. Attach a screen recording showing the performance problem.
2. Attach any charts showing the Octopus Server performance (CPU/RAM) for normal deployment workloads both before/after the performance problem started.
3. Record and attach the performance problem occurring in your web browser (if applicable).
4. Attach the Octopus Server logs.
5. Attach the raw task logs for any tasks exhibiting the performance problem, or that may have been running at the same time as the performance problem.
6. If the performance problem is causing high CPU utilization on the Octopus Server, please record and attach a performance trace.
7. If the performance problem is causing high memory utilization on the Octopus Server, please record and attach a memory trace.

Please email Support@Octopus.com

Conclusion

Regular maintenance and keeping Octopus Deploy up to date will go a long way in ensuring your experience with Octopus Deploy is positive. If you do start to see a dip in performance please reach out to us. We will do our best to help out.

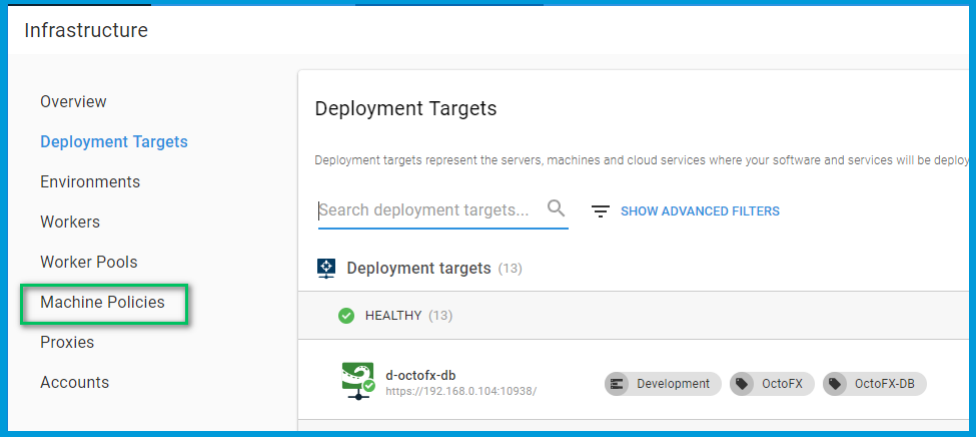
We want your experience with Octopus Deploy to be great.

22

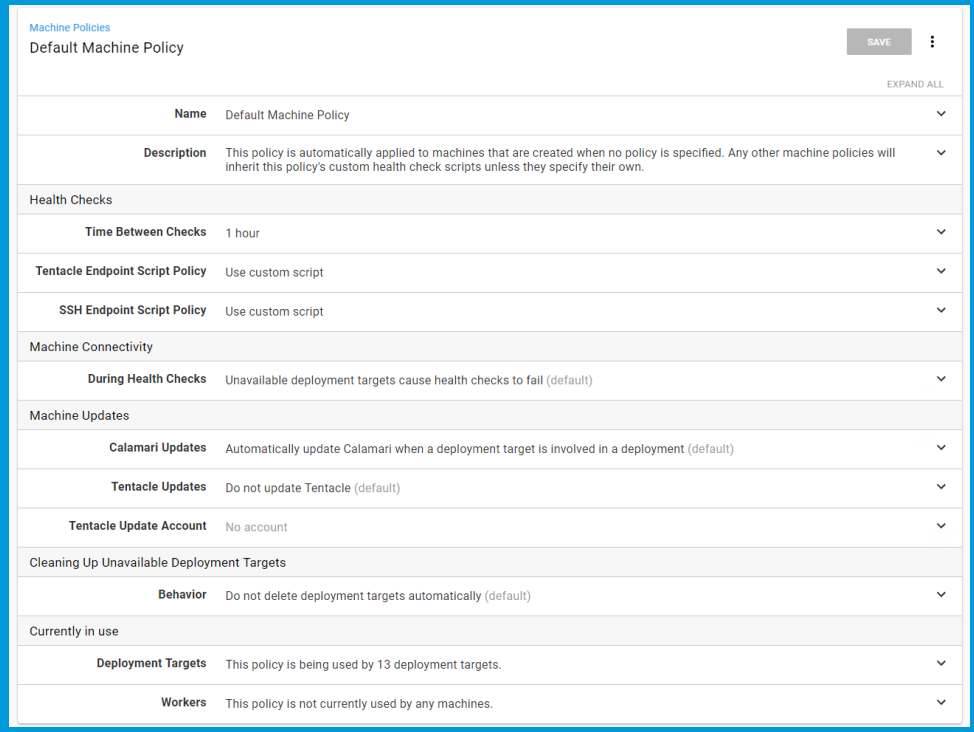
Making sure your server can

CONNECT TO DEPLOYMENT TARGETS USING MACHINE POLICIES

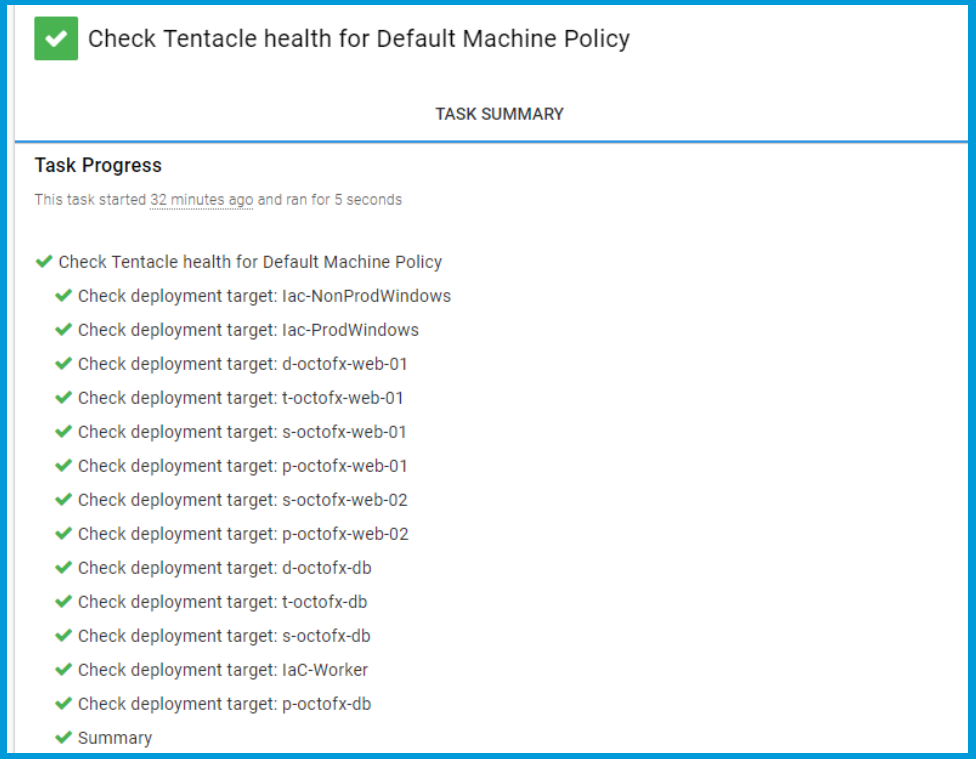
A topic that we have been dancing around for the past several chapters is machine policies. You probably noticed it on the left-hand menu and didn't think much of it.



The Octopus Deploy server needs to make sure that it can still connect to all the deployment targets. We call that a health check. It runs periodically on the Octopus Deploy server. It does this because many of our users want to know about problems with a machine before doing a deployment. If a system admin can fix a minor problem before it becomes a significant problem, then it is well worth the effort. It also does this as a sanity check to make sure that the server has not crashed or is about to run out of space. By default, health checks run every hour. If the Octopus Server cannot connect to the machine, it will fail the health check. All machines are added to the default machine policy when no policy is specified. That is configurable on the machine policy screen.



When a machine policy performs a health check, it queues a task on the Octopus Server. That is a blocking task for any deployments. The server needs to make sure the machines are there before it can deploy code.



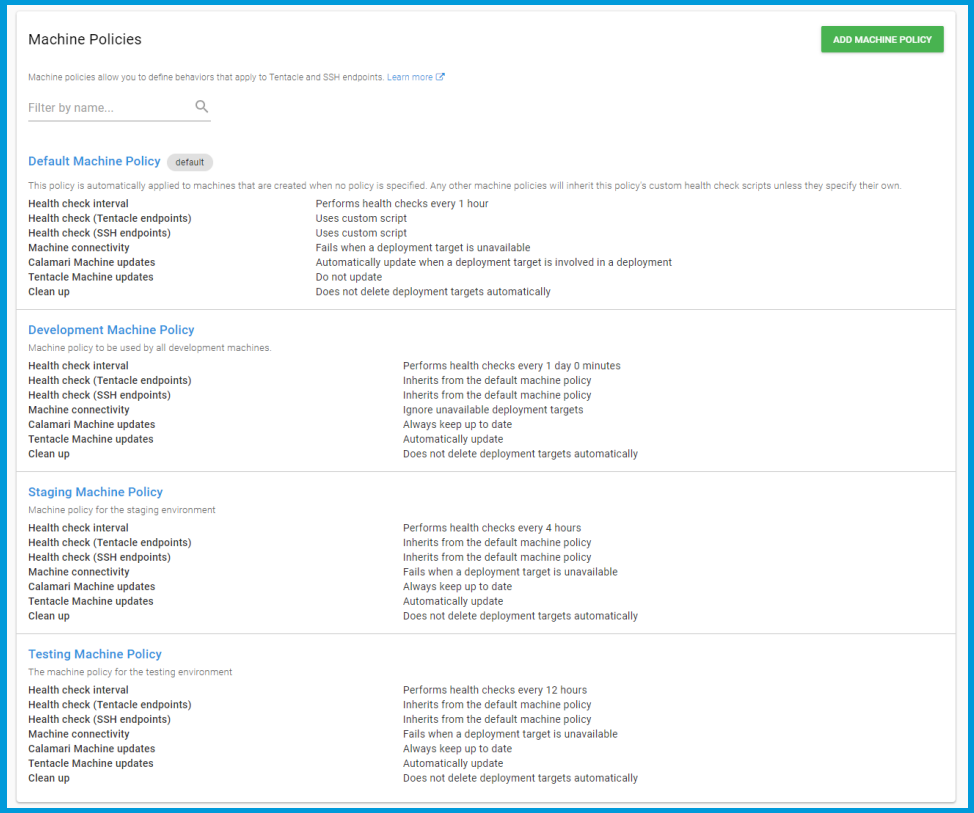
Using the default machine policy for less than a hundred machines is excellent. The health checks are quick. It starts to cause problems when you have thousands of machines. A health check could take hours to complete. It can only connect to 10 machines at a time to perform the health check; it can take a while to finish up.

You also might have a group of machines which are flaky and tend to go up and down. Perhaps they are test instances, or they're hosted on a client's site, and your connection is spotty. You expect the connection to fail once in a while. It doesn't make much sense to change default machine policy to account for one or two spotty machines. The other machines in the machine policy should be running. You want to know about those if they go down.

Our recommendation is to create several machine policies. Some possible strategies are:

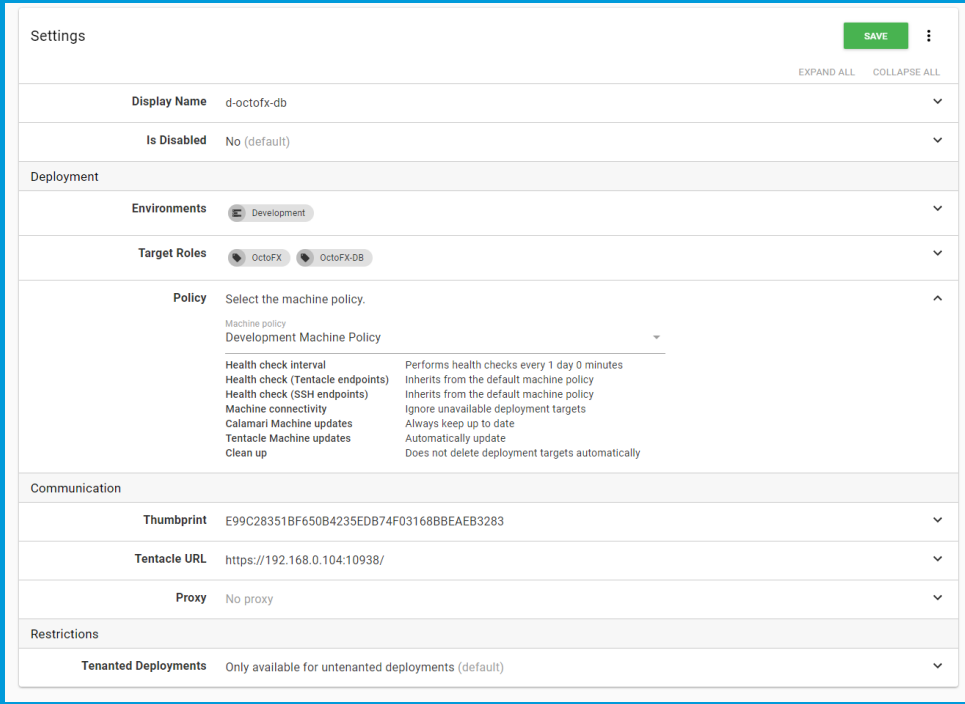
- Machine policy per environment
- Machine policy per data center
- Machine policy per tenant (if the tenant is hosting their servers and you deploy to them)
- Transient Machine Policy (for machines which tend to go offline randomly)

If you go the machine policy per environment, we recommend changing the times between health checks. Something substantial for dev, say once a day, while production stays at once an hour. Dev is deployed to several times per day while production deployments happen once a week. Dev machines tend to go up and down at random, change the configuration to not fail the health check.



Each machine policy health check is a unique task. That allows Octopus Deploy to multi-thread the health checks. If you are using Octopus Deploy's High Availability functionality (available with data center licenses), the health checks are spread across multiple nodes.

You change the machine policy for a specific deployment target by going to the deployment target screen.



Conclusion

In this chapter, we set up multiple machine policies to ensure we have a high signal to noise ratio if/when Octopus Deploy cannot connect to a tentacle. Production is checked the most because typically that has the least number of deployments. An environment such as development is getting deployed to hundreds of times a day, and if a machine were to go down most people would know about it, but it would have the smallest impact on the business.

KEEPING OCTOPUS DEPLOY UP TO DATE

Upgrading Octopus Deploy

You have your Octopus installation setup, and you have your deployments cranking away daily. You start to notice the bell icon at the top of the Octopus Deploy interface announcing new changes. Security starts asking questions about patching Octopus Deploy. Users want to see new features being published on Twitter and through the blog every month. It is time to define an upgrade strategy.

In this section, we will cover a few things that you will need to consider as part of your Octopus Deploy upgrade process.

> This section will not cover how to upgrade, it would be best if you looked at our website for the latest instructions.

What and How are the Octopus Deploy releases structured?

We use our version numbering scheme to help you understand the type of changes we have introduced between two versions of Octopus Deploy:

Major version change, breaking changes and new features.
Example: Octopus 3.x to Octopus 2018.x

Upgrading may require some manual intervention, we will provide a detailed upgrade guide. Downgrading will be very difficult. Pay close attention to the release notes. Setting up a test Octopus Deploy instance to try out the upgrade is recommended.

Major breaking changes examples: Octopus 1.x to Octopus 2.x we rewrote the HTTP API. Going from Octopus 2.x to Octopus 3.x we changed the Tentacle communication protocol and moved to SQL Server.

Minor version change, new features, the potential for minor breaking changes and database changes.
Example: Octopus 2018.1.x to Octopus 2018.2.x Upgrading should be easy, but rolling back will require restoring your database. The upgrade will most likely make changes to the database schema. There is a chance changes are made to the API. The goal is the changes will backward compatible wherever possible. Check our release notes for more details. Check for breaking changes as they are likely but less likely that in a Major version upgrade.

Minor breaking changes example: In Octopus 3.3 we made a change to how Sensitive Properties work in the API.

Patch version change, small bug fixes, and computational logic changes.
Example: Octopus 2018.8.1 to Octopus 2018.8.2

Patches should be safe to update, safe to roll back. There is more risk in not upgrading than upgrading. Database changes are possible but are rare. When database changes are made, it is to patch a critical bug, the change will be safe for any other patches of the same release.

Changes example: Octopus 2018.2.3 to any other patch of Octopus 2018.2.X (upgrade or downgrade). We may decide to make API changes, but any changes will be backward compatible.

Why stay up to date with Octopus Deploy?

There are several benefits to upgrading your Octopus Deploy instance.

- Whenever we get reports of performance problems we do our very best to isolate them and fix them as soon as possible. Except in rare cases, the

latest edition of Octopus Deploy will be faster than the version you have installed.
- Security flaws are treated as critical and we will fix them as soon as possible. The latest version of Octopus Deploy will have all the latest security patches.
- In 2018 a new feature was released roughly every six to eight weeks. This includes features such as Workers, Kubernetes Support, Azure Support, and Scheduled Triggers. Upgrading will give you access to new features.
- Any software is bound to have bugs in it. Octopus Deploy is no different. Each new minor version will include multiple bug fixes in it.

When should you upgrade

- This really will depend on the type of organization you are working in. Octopus has 2 types of releases and is categorized as:
- Long Term Support
- Fast Lane

Not every customer is the same, we will provide the facts so you and your organization can decide.

Long Term Support

Long Term Support versions of Octopus are shipped every three months and remain in support for six months. It contains a roll-up of all changes, including new features and bug fixes, released in the previous three months. This is the version we recommend to our customers. Once an LTS release is created we only ship security enhancements and fixes for show-stopping bugs. The rule of thumb is upgrading to the latest LTS release should be less risky than staying on the current LTS release.

Choose the slow lane releases with long-term support if this sounds like your scenario:

"We prefer stability over having the latest features." "We upgrade Octopus about every three months." "We evaluate Octopus in a test environment before upgrading our production installation."

Fast Lane

Features are released when they are ready, generally every four to six weeks, and ship bug fixes and minor enhancements into patches every few days.

You should choose the fast lane releases if this sounds like your scenario:

- "We need the latest and great features and fast turnaround on small incremental enhancements and bug fixes."
- "We want to engage closely with the Octopus team, so we can help them build the best automation tooling in the world!"
- "We have automated our install process. Upgrading to the latest version takes less than 10 minutes."

Changing from LTS to Fast Lane

It is possible to move from an LTS release to a Fast Lane release. Fast lane releases are almost always the highest version number you can download. To upgrade you will need to download and install the higher version number. Please be sure to check the release notes for any breaking changes.

To move from the Fast Lane to the Slow Lane will involve more effort. You will need to wait for the next LTS version to ship and then upgrade to the latest LTS version. For example, the first LTS release, 2018.10.x was created in late December 2018. In early 2019, the Spaces feature was introduced in 2019.1.x. Customers who upgraded from 2018.10.x LTS to 2019.1.x to get

spaces couldn't downgrade back to 2018.10.x LTS. They had to wait until the 2019.3.x LTS release.

Conclusion

We recommend coming up with an upgrade strategy as soon as possible. When we were in charge of our Octopus Deploy instances at other companies this meant being on the last patch release before a minor release. 2019.2.2 is the most recent release, we would download and install 2019.1.11. There is no guarantee 2019.1.11 was the most stable. 2019.2.1 could've fixed a major bug. It was always a guessing game.

LTS was designed to fix that guessing game. LTS is stable, only releases to fix show-stopping bugs and security fixes will be issued. For the majority of the people reading this document, we recommend LTS is their upgrade strategy.

Happy deployments!

Join the conversation over on our community Slack channel
https://octopus.com/slack


Octopus Deploy