# Chapter 10: Deploying the release

### Professional Automated Deployments with the Release Hub

Now that you’ve properly packaged release candidates, you can use the Release Hub of Azure Pipelines to model your environment pipeline and configure deployments. You can define multiple deployment pipelines that use a single build as a source of release candidates. In this example, you’re targeting Azure PaaS services for the runtime environment of your application. As the builds are released to your deployment pipeline, you’ll see something similar to **Figure 25**.

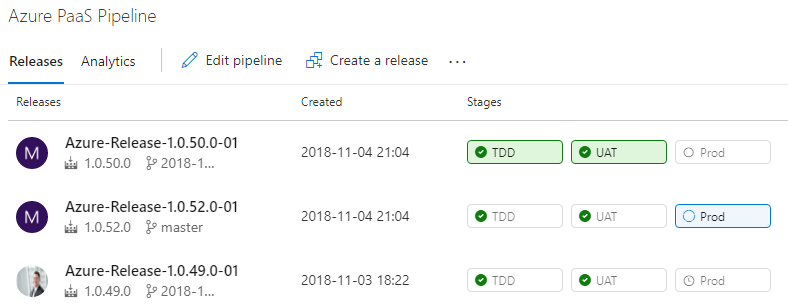


Figure 25: You can track each build as it’s deployed through your environments.

Earlier in this article, I discussed the three distinct types of environments in a DevOps pipeline. In your organization, you may need multiple instances of one or more of the environment types, but in the application here, you have one environment per type for demonstration purposes. **Figure 26** shows the environments configured in series.

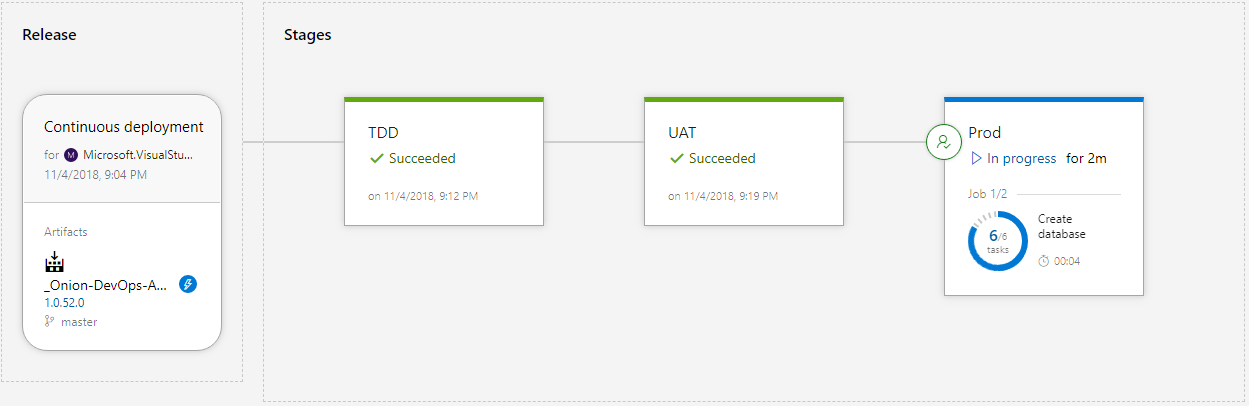


Figure 26: Each environment receives the same release candidate as it’s promoted from the environment that precedes it.

The source of the release configuration is a continuous integration build. The version number of the release inherits the build number. In most cases, you’ll configure your environments in series, starting with the TDD environment, then UAT, then Production. Your names may be different. The software is built and packaged exactly once, and the release candidate, in the form of NuGet packages, is deployed to each successive environment. Let’s see how to configure the Release Hub.

The source of the release configuration is a continuous integration build. The version number of the release inherits the build number

**Figure 27** shows how to enable builds to trigger a release.

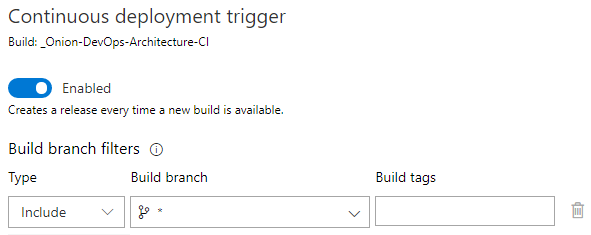


Figure 27: Configure the trigger in release artifacts to include builds from every branch.

It is important to configure the Build branch for every branch. If you don’t, your feature branch builds won’t trigger a release, and you won’t be able to use your full-system test suites to validate these builds before executing your pull request.

When you edit the deployment process for an environment, you’ll want to make sure that the steps are the same from environment to environment. The best way to do this is to structure the deployment steps like you would a PowerShell script. That is, you factor the steps into functions that are called Task groups. **Figure 28** shows the deployment process for your environments.

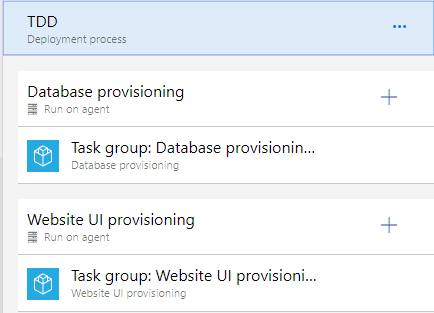
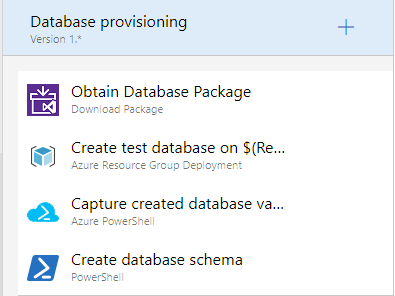


Figure 28: Each deployment process is essentially just a pointer to the Task groups that are relevant for the environment.

You use task groups so that you don’t violate the DRY principle (Don’t Repeat Yourself) when specifying what steps should happen per environment. Because you have three environments, you don’t want to copy and paste steps across environments.

When you look into the task groups for the deployment of each of your application components, you see the individual steps. **Figure 29** drills down into the individual steps needed in order to deploy the database.



**Figure 29:** Four steps create and deploy your SQL Server database.

These same steps can run on every environment because the behavior varies by the parameters that are passed in. For instance, in your TDD environment, you want to destroy the database and recreate it from scratch. In the UAT and Production environments, you want to preserve your data. A variable dictates which of these paths is taken per environment. **Figure 30** shows the full list of variables used by this release configuration.

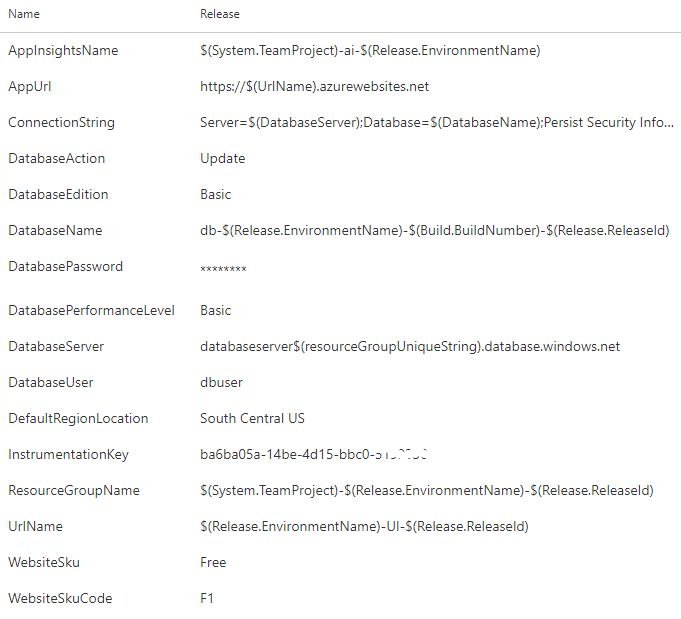


Figure 30: The release behavior varies by the variables that are configured.

Take a critical look at the values of the variables. This is true for the build variables, and it’s true for release variables as well. Some of the values are scalar values, but many of the values are templated patterns. Because you’re going to be creating many releases, you need values that are going to be resilient to the repetitive nature of DevOps. You also need a variable scheme that is going to be resilient to the inherent parallelism of deploying release candidates from multiple branches, all at the same time. Because of this dynamic, you can’t assume that only one release will be deploying at a time. Each of these values need to be unique, so you assemble the values based on properties of the release itself, the environment you are deploying to, and the component of the application being deployed. You’ll want to have a design session with your team to determine how to dynamically build your variables.

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