# Azure Environment Governance with GitHub and Azure Blueprints

## Script 1.t For a Technical Developer Audience

See Script 1.B for Non-Technical Decision Makers for an alternative version



## Introduction

Today we want to talk about Governance. More specifically, we want to talk about how you can manage code deployments into multiple Azure Environments using both GitHub and Azure Blueprints. This is really a combined Development Team and Operational Team workflow, as the Dev Team will be responsible for developing and testing the code, which includes testing it in an environment similar to the production environment. The Ops Team then needs to test this package in an environment that is extremely similar to production and of course move the deployment package into production when it is ready and has been approved.

|  |  |  |
| --- | --- | --- |
| Scene Demo Steps | Talking Points | Content Notes |
|  | We are going to demonstrate how to do this using a combination of GitHub Actions and Azure Blueprints.  Specifically, we are going to:   * Demonstrate a best practices Governance Model for Azure Deployments * How to use GitHub actions to deploy CI/CD into Azure * Detail best practices as they apply to Dev/Test/Pre-Prod/Prod environments. It important to note that this is not prescriptive guidance for all CI/CD into Azure as there are many different permutations and requirements and one size does not fit all. However, we will detail the general best practices that you can modify into your specific environment. * Show how to use GitHub CI/CD in combination with Azure Blueprints to create well managed environments in Azure | The artifacts for this demo can be found in:  <http://github.com/onemtc/governance-script-sandbox-corp-templates>  <http://github.com/onemtc/governance-script-sandbox> |
|  | Why are we spending the time to discuss this? Good cloud/Azure governance is about keeping yourself from shooting yourself in the foot. This is normally organized around three principals:   * Policies - what folks are allowed to do * Permissions -- what specific folks can do these things * Costs -- how much we can be allowed to spend on these tasks   We're not really going to cover Costs in this demonstration, however, you can set up policies which would limit the types of services and skus (such as VM sizes) that folks can deploy, so this ends up having an indirect effect on costs.  We have spoken with a lot of customers, and at the start of their cloud journey, many do not have a handle on how to provide good cloud governance, as well as what tooling Microsoft provides to aid this effort.  We know this and have created an avalanche of documentation and frameworks in order to explain how to do governance in Azure. Due to the quantity of information, sometimes the important details get lost. So, we'd like to clarify how you would concretely implement part of the governance story.  Additionally, traditional enterprise governance has been the responsibility of IT departments. Now, we want to use traditional developer tooling in order manage this part of the enterprise, and these tools are more unfamiliar to the Ops side of the house. |  |
|  | Let's talk about environments in Azure. When we talk about environments on-prem, we usually think about servers and firewalls. In Azure however, environments are just a group of resources that work together. They could be network segments or firewall protected, but they don't have to be. The actual separation required will be different depending on your product structure and goals.  We usually recommend a standard model of 4 environments:   * Dev * Test * Pre-Prod * Production   We can draw these and think of them an individual resource groups in Azure, but the environments can span as many resource groups as you need. In fact, there are usually infrastructure resource groups that are connected to at least prod and pre-prod, which contain core networking and potentially databases which are used by multiple products.  Let's think about these environments for a moment. They will be different, of course, but let's discuss how they will be different.  Prod will contain items that are not intended to be torn down. While it is true in Azure that you can just use pipelines to build new infrastructure as code, you are not going to be taking down your core networking between releases. For one reason, it would not allow you to upgrade your application and keep it running, and for a second reason, it would be a huge pain to your ops team. They should be able to rebuild your environment if it should go down for some reason, but they should not take it down under normal circumstances. Thirdly, you normally have production data in database, which does not go away for the life of the product.  Pre-prod should be extremely close to prod. In a perfect world it would be a mirror of prod, as this is where you want to do your final operational testing before moving software into production, so all policies that could be restrictive in prod should be in place in pre-prod so that they can be tested. Generally, your databases in pre-prod will be identical in structure to that in prod, but may contain simulated data. You could very well share your database between prod and pre-prod depending on your application.  Prod and Pre-prod are owned by the Ops teams. Generally, the development team will not need to, or want to mess with these environments.  On the other side, you have Dev. The Dev environment is like the Wild West. You don’t really know how application pieces got there, as the Devs may not know what they need for the product when they start developing it. I'm going to get to policies in a minute, but in general, you want the Devs to own this environment. More specifically, you want the allowed policy structure of this environment to allow the Devs to do the work they need, which protecting the concerns of the rest of the organization. We like to think of this as minimal, but important guardrails. From an Ops standpoint, the Devs should be able to create, destroy, or rebuild services in this environment as they need.  In between you have the Test environment. The Test environment should be co-owned by both the Ops Team and the Dev Team. It should also be an environment that is regularly torn down and rebuilt, preferably through pipeline automation. It's important to note that when we say "test" here we are actually describing two separate tasks:   * Testing that the application runs outside of the uncontrolled Dev Environment. * Testing that the application runs with a set of policies that mimic prod/pre-prod. We say mimic, as it's really "whatever is reasonable", as you may not be able to make the environment duplicate prod and pre-prod, but you should be able to come reasonably close depending on things like network configuration and data access.   One of the important things to note in test is that during testing, both teams should be able to turn policies on and off, in order to facilitate testing. This does not mean that it should become wide open, and there should be a set of organizational enterprise policies which are always on everywhere, but also there should be a set of production policies which can be gradually introduced into a new version, in order to quickly get the version working outside of Dev.  I realize when I say this, it may give you some pause, but we are going to talk about policies in more detail in a moment. |  |
|  | When we think about deployment, we need to realize how the environments differ, which we've discussed. Let's look at how code should get into each environment.  As I mentioned earlier, we're not going to worry about how code gets into the Dev environment. The Devs will create it. Maybe manually, maybe through some kind of automated pipeline, maybe a combination -- we don't need to worry about this.  When we deploy from Dev to Test, we want a repeatable way to lay down both code and infrastructure resources. Here, the best practice is to use infrastructure as code in some kind of automated pipeline, which in our case, would be GitHub actions. The infrastructure would be defined in azure using an Azure Resource Manager Template, which is a JSON description of resources in Azure. If you are not familiar with ARM Templates, it's a native way to do IaC deployments into Azure. Some folks will want to use TerraForm instead, and that's just fine. You can manage the TerraForm files in a similar way to how I am going to describe managing the ARM Templates.  The key is that the code and infrastructure should be deployed to test only through the pipeline. This allows you to a) wipe and reset between tests and b) ensure that once code is ready to go, that there is an appropriate deployment pipeline set up. During initial testing, the Dev's are going to create the ARM templates for the first time, and as part of the testing process, the ARM templates will be reviewed by the Ops Security Team. The reason for the security review is to judge whether the ARM template is "good enough" to be used in production deployments. If it is not, the Devs and Ops folks can work together to make it good enough.  This is a key part of the workflow, as once the ARM template has been blessed by everyone, it should be stored in a separate corporate repository, so in addition to using the template on this product, you can use it on any product where you need that resource. ARM Templates can be written for a single resource, or a whole environment or anything in between, so it's up to your organization to determine the granularity of the template which you are storing. Since the stored templates have already been reviewed, this will bring down the amount of work needed for everyone over time, as reusable ARM Templates should not need to be reviewed again. In this example we are going to put the ARM Templates in a separate GitHub repository. There is some preview functionality which will eventually allow you to store these templates in Azure directly, but really, where you put them is not that important. The important piece is that over time, you will build up this repository of security-reviewed ARM templates that can be used across the Enterprise.  So, now we can deploy to Test; code and resources through IaC, but we also need to deploy to Pre-Prod and Prod. We will treat these the same since they should be nearly identical. As a best practice we do not want to use IaC for this. Remember, we have a lot of things in Prod which we do not want to regularly destroy, A better way to do deployments into these environments is though Azure Blueprints.  Azure Blueprints Are a combination of:   * Azure Policies * Azure RBAC * Azure Arm Templates   All combined together in order to lay down an environment. So, it's:   * What you can do * Who can do it * What resources are available for use   Additionally, once the blueprint is in place it can counter drift that can occur in environments over time. One of it's great features is that when a blueprint is published and deployed, it can be set to not allow any other resources to be put into the same resource groups unless they come through an updated, published version. This will stop administrators from just adding potentially forgotten pieces, which is always great at the time, and not so great a year later when no one remembers how that resource was created.  It is important to note that although we are going to use Azure Blueprints for resource deployment, your actual code is still going to be deployed through your pipeline, GitHub actions. ARM Templates do not define code that goes into resources. |  |
|  | I said that I would get back to policies.  When I speak about Azure Policies, I'm not just discussing the theoretical, but an actual implementation which can be put against a Resource Group or Management Group in Azure. This restricts what can be done in those environments -- by anyone -- this is not a permission substitute.  Initially, when we were thinking of this type of environment, we thought that Prod would have a lot of policies, and Dev would have less. We don't actually think that is true. We think that they all have policies of different types.  Dev policies are the guardrails to protect the enterprise. These may be things like:   * No public IPs allowed * Only use a VM up to a certain size   These are policies which you would put in place to protect your environment, your IP, and to control your costs. These policies would also reasonably be expected to be maintained in the test environment. They would be the policies which cannot be shut off.  In Production, however, you don’t want or need those policies. First of all, you may need public IP addresses, since this is the production environment. Secondly, since you are going to be deploying all of your resources through Azure Blueprints, you do not really need a policy to determine the VM size, as it can be baked into one of the available blueprints. You could have the policies in place, but you should not ever need to use them, since all of your resources have been security-reviewed and predefined.  Instead you will have different production policies, such as:   * Disk encryption should be applied on Virtual Machines * CORS should not allow every resource to access your function app   There are many, many built in policies in Azure and there is a robust descriptive syntax so you can define your own custom policies. |  |
|  | Let's take a look at this in action. Here I have my project repo, where I am going to deploy an app service and some code for a web site to Azure. The code is .Net core, but the actual application is completely unimportant for this demonstration. We are more concerned with how we get the code deployed.  I have three actions defined in GitHub. Deploy to test, deploy to pre-prod, and deploy to prod. The first one deploys the code and resources into test. The other two just deploy the code into an environment that is expected to already be there. |  |
|  | If we take a look at the top of the deploy to test action -- actually all of the actions are written the same way, we see that we have four variable definitions. These variables describe the location of an Azure storage account where we are using Azure Table Storage to hold the variables we really want to use in our pipeline. This is advantageous as we can set up a single table for each product, and then use the partition key to store variables for each action. You would not want to put the production variables in the action itself, as then you would need a different copy of each action for each environment, and you would end up with versions of versions of actions, and a change in one may not propagate to all the different permutations. Eventually, this would go bad.  We chose using Azure Table Storage for this because a) it's very inexpensive, b) it's quick and easy to set up, and c) if you use the Microsoft Azure Storage Explorer, you can read and edit the variables in a GUI interface directly from your desktop; for Windows, Mac and Linux  Important safety tip: This variable scheme only works for non-secret variables. If you have a secret which you are using in your action, like a connection string or a key of any type, then you should put that secret into an Azure Key Vault instead. We're not going to show that specifically in this demo, but the two can work together, you can have secret secrets in the key vault and non-secret secrets in Table Storage. | Azure storage explorer can be downloaded from: <https://azure.microsoft.com/en-us/features/storage-explorer/> |
|  | Going back to the Action. Underneath the variables, you will see that we actually checkout the code in the repo and the code in the Corporate Template Repo. As I mentioned earlier, your ARM Templates should be in and repo separate from your product code so that they can be reused across your organization.  Below that we build the code. It's a standard .Net Core Build, but I will make note that the publish step has a specific output directory. This is important as we are going to need to find the built code later in this action in order to do the deployment.  In all of the workflows in this demonstration, we rebuild the code each time, but in any type of real world scenario you'll want to store your outputs so that you can use the built code for future deployments. This discussion is confounded with a discussion of project branching and releases, and we're just not going to discuss these topics here.  Once we have built the code, we are going to log into Azure using credentials that have been stored in our GitHub secrets repository. In this case, the account logging in is a service principal with full access to the environment so we can access everything we need, including the table storage variables.  We use the Azure CLI to read the variables from table storage, and we grab the storage key directly from the storage account, since we are logged in with an authorized service principal. This is why we are not using the Kye Vault in this example -- we don’t actually store the key anywhere, we just ask Azure to give it to us, since we are logged in.  Once we have variables, we deploy the resources using the ARM template -- actually we are using two ARM templates here, again through the CLI. One thing to note about the CLI. The CLI is in the baseline images that GitHub uses, so you do not need to use a separate action to call into the CLI. There is a CLI action, but that was created for you to use older versions of the CLI, and does not allow the return of variables. Really, there is no case where you would not want to use the latest version of the CLI.  After we've deployed everything, we call out to a GitHub action to deploy the web code to an Azure AppService. Note that the package where it finds the code, is the directory where we published out build output.  We can run this action from the GitHub web page. Note that I don't have any triggers on my actions, but that is just so I can demo this. You could add any triggers you need. |  |
|  | While that is deploying, let's create a blueprint in production.  We're going to leave the GitHub web page and go to the Azure Portal.  I have already created a resource group for pre-prod and production. Now we're going to create an Azure Blueprint for production and deploy it.  There are two parts to using an Azure Blueprint; creating it, and then publishing it is the first part, and then assigning it is the second part.  To create the Azure Blueprint, we are going to click on Create Blueprint from the Azure Portal. We will start with a blank blueprint, and give it a name. We need to specify a location for the blueprint, and I will choose my subscription.  Then we need to add things to the blueprint:   * A resource group * ARM Templates, we will use the same two ARM Templates that we used for the Test IaC deployment. We've already downloaded the templates onto my local machine, so that I can use them in a blueprint * An RBAC configuration, so we can specify who has access to this environment. We will make them a contributor, but we will not actually select the person until assignment time * Finally, we will select an Azure policy. We will use the allowed location policy to restrict where we can deploy resources in this resource group.   As I go through this, there are a couple of things to note. The first is, that although you could deploy Azure Blueprints through GitHub actions, you do not really want to do that. Or Rather, if you were to do that, you would need to make sure that the blueprints were separate from all of the code. This is because it can take up to 30 minutes for an Azure Policy to take effect and up to 5 minutes for an RBAC role to take effect, so if you deploy Azure Blueprints through an automated pipeline and you don't wait the requisite amount of time, you could deploy code incorrectly before this pieces take effect, which would definitely be bad.  The second, is as I do this, I am going to name the AppServicePlan ARM Template as a\_appserviceplan and the AppService ARM Template as b\_AppService. This is because by default Azure will sequence the ARM Templates alphabetically and in order for this to work, we need the AppServicePlan to be deployed first. If we were doing this through the CLI or some other method, there are more elegant ways to specify the ordering, but the portal is currently limited to alphabetical only.  When we are done, we will save this as a draft, and then we will publish it with a version number.  Once that is done, we will assign the blueprint. Here we will choose all the parameters for our blueprints. There are quite a bit of parameters for the APRM templates, but this is really because these are the parameters which were coded into the Templates themselves. The assignment process will surface them, so if you create ARM Templates with less parameterization, then you would have less to enter here.  On the assignment page, there are two interesting items to note. One is that you can select the type of managed identity you want to use in order to give permissions to the Blueprint process to make the deployment. With a system identity you don’t have to do anything else, but in a locked down environment, you can also use a previously created user identity.  The other thing on this page is the lock assignment. I'm just going to run this demo as "Don't Lock", so the blueprint will run and deploy the resources, but after that, as the owner I can go in and change them if I want. If I specified "Do Not Delete" than I could go in after and add or modify items, but I could not delete any of the blueprint pieces. To do that I would have to unassign the blueprint or assign a new blueprint version which did not have the particular resource. Finally, as I mentioned earlier, "Read Only" will not allow any modifications to the blueprint resources, including the resource group, unless a new blueprint version is assigned.  So now we've assigned the blueprint and our production environment is being deployed.  For Pre-prod, I have two choices, I can create a new blueprint and go through the whole process again or I can just assign the existing blueprint a second time. If they are going to be the same, I do not need to actually create a second blueprint.  For timing concerns, however, I already deployed the pre-prod environment before this demo, so we'll just pretend that I did a second assignment. | Note, when assigning the ARM template, do not choose yourself into an RBAC role where you already have permissions. Blueprints do not like it, and you can get an error. I always just choose someone else for the demo, as I am already an owner on the resource groups, so I already have access. |
|  | While that is running, we can go back to GitHub and take a look at the Deploy to Pre-prod action. You'll see that it is almost the same as the Deploy to Test action except that we do not run the ARM Template deployments, and so we need less variables. The variables that we do need we still read in from table storage. Of course, since we are not running the resource deployment, we do not need to log in to the corporate repo.  We can now run this, and it will deploy.  We could then wait until the blueprint is finished and deploy that as well.  And that is an example of our best practices governance for deploying Azure environments with GitHub and Azure Blueprints. Any questions….. |  |