Getting started - New Blueprint

Overview (Context)

This is a step by step guide for getting started to developing blueprints.

Pre-requisites

- Developer Guide

Setup guide

Setup boilerplate

1. Create an EMPTY blueprint repo Azure DevOps blueprint organization following the naming conventions. **Make sure to de-select the** `include README.md` checkbox. Follow the steps here to create a new repo in Azure DevOps.
2. Clone the created blueprint repo.

```
git clone git@ssh.dev.azure.com:v3/BHP-Tech/Landing-Zone-Azure-Blueprints/<repo-name>
```

3. Clone the blueprint seed repo

```
git clone git@ssh.dev.azure.com:v3/BHP-Tech/Landing-Zone-Azure-Blueprints/_git/azr-blueprint-seed
```

4. Push the seed repo code to master of your empty repo.

```
rsync -r --exclude '.git' azr-blueprint-seed/ azr-<blueprint-name>
cd azr-<blueprint-name>
git add .
git commit -m "Initial commit +semver: none"
git push origin master
```

5. You now have your boilerplate ready. Follow the steps below to complete your blueprint. You will need to customize the following sections before raising a PR :
   - `Terraform files for resources`

    - Test cases
    - input.tfvars
    - tests/dependencies.tf
    - tests/vars/*.tfvars
    - ci-pipeline.yml
    - cd-pipeline.yml
    - outputs.tf
    - README.md

Boilerplate structure

.azuredevops folder

- This contains the Pull Request template which has placeholders for:
- Pull request change impact (major/minor/patch) to be used for Semantic Versioning using Git tags.
- Common content to be automatically used for Git tag description and CHANGELOG.md.
- README.md
    - Standard file used for giving a summary of the repo.
    - Follows the format :
        - Introduction
        - Dependencies
        - Input parameters
        - Output variables
        - Testing (Manual and Automated)
        - CI Pipeline
- git-version.yml
    - Seed version used for tagging. Starts at 0.1.0. After that the CI Pipeline depends on tag versions.
    - One exception : To move from 0.x.x series to 1.x.x, you will have to change this file to be 1.0.0.
    - Read more about Versioning Strategy in SemVer and GitVersion.
- .gitignore
    - Provides common list of files to ignore.
- CHANGELOG.md
    - Change log with information on updates to the repo.
    - Content in this file is auto generated via the CI pipeline.
    - Users can still add modify it if needed.
- ci-pipeline.yml
    - Continuous Integration pipeline file.
    - Depends on pipeline templates.
- cd-pipeline.yml
    - Continuous deployment pipeline file.
    - Depends on pipeline templates.
    - Deploys resources taking env specific values from variable groups.
- input.tfvars
    - Contains values for the variables to be used while deploying the resources.
    - Certain values are injected from the variable groups while running the pipeline.
- **Terraform files**
    - meta.tf
        - Contains minimum Terraform version supported, complex data transformations and data sources.
    - outputs.tf
        - Contains module output.
    - variables.tf
        - Contains input variables.
- **Testing folder**
    - models
        - Contains structure of go files
    - dependencies.tf
        - Contains any resources which need to be created for integration testing.
        - Contains pipeline variables to which will be replaced by the pipeline during execution.
    - vars
        - Contains files with values of input variables.
        - Contains pipeline variables to which will be replaced by the pipeline during execution.
    - test-provider.tf
        - Provider file for running Terraform scripts.
        - For Pipeline testing, its moved outside automatically.
    - Gopkg.toml

    - Contains list of Golang dependencies.
    - Read more about the TOML format here
- `Gopkg.lock`
    - Auto generated lock file created while installing Golang dependencies.
- `unit_test.go`
    - Golang file used for unit testing.
- `integration_test.go`
    - Golang file used for integration testing.

Development

1. Create a file for the blueprint with the name `<blueprint-usecase-name>.tf`. Create separate files for each logical unit in the blueprint.
2. Refer the usage section in readme of individual components for using components as modules.
3. Chain the different modules by passing the outputs from one module to another.
4. Update the `variables.tf` and `output.tf` file for the blueprint.
5. Add the dependent resources in the file: `tests/dependencies.tf`. These would be required for testing.
6. Update the test values for the variables in the blueprint: `tests/test-input.tfvars`
7. Test the code manually. Refer to manual testing section in the Readme file.
8. Validate the creation of all the components in the blueprint on the portal.
9. Destroy all the components post manual testing.
10. Update the relevant sections of the readme.

Setup unit test cases

1. Update the path of the models folder in the file: `tests/unit_test.go`
2. Run the unit test cases. Refer to the Testing section in Readme.
3. The unit test case would run **validate and plan**
4. Fix if any errors or warnings from the validate output
5. Copy the json output from the plan command
6. Convert the json output to go struct. Use an online tool like JSON-to-Go.
7. Update the models folder with the structs.
8. Format the json output and use the values for specifying asserts on the outputs from the plan command.
9. Run unit the test cases and validate that the test pass.
10. Create functions and add inputs for different testing scenarios in the folder `./tests/vars`

Setup integration test cases

1. Go to the file: `tests/integration_test.go`
2. Update the test input file path
3. Add assertions for the outputs from the apply command
4. Run the integration test cases which will run **apply and destroy** commands

Setup CI pipeline

1. Push code to a branch (eg : <developer-initials>/<type-of-work>/<ticket-number>-<title> ie uk/feat/1234-add-documentation).
2. The CI pipeline has been tailored to work for almost all uses. Refer to the Blueprints pipeline repo in case any custom changes are required.
3. To understand about pipeline variables and dependency injection refer the blueprint pipeline Readme.
4. For setting up the pipeline using `ci-pipeline.yml`:
    - Go to Pipeline section in the Azure DevOps
    - Select the following: New Pipeline => Azure DevOps git => Select your repo => Existing azure pipeline => Select your branch => Select your ci-pipeline.yml
    - Refer the document for getting started with creating a pipeline
5. Run the pipeline for executing the security scans and running the unit and integration test cases.

Setup CD pipeline in Terraform Cloud

1. Ensure the Terraform Cloud organization is linked to the Azure DevOps(ADO) organization (Details on linking can be seen here.)
2. Create a workspace in the Terraform Cloud following the convention : <repo-name>-<environment-if-any> (eg: azr-caas-web-spoke-dev).
3. Ensure the workspace trigger is set to master branch.
4. Add variables to the above workspace.
    a. The environment variables need to be added while ensuring their values are hidden for security reasons. They are :
        i. ARM_CLIENT_ID
        ii. ARM_CLIENT_SECRET
        iii. ARM_SUBSCRIPTION_ID

iv. ARM_TENANT_ID

b. The Terraform variables to be added can be obtained from the tests/vars/test-input.tfvars file.

5. Make sure that the service principal used has the desired permissions to create resources in the specified subscription.
6. You can run the CD pipeline manually now by hitting the ‘Queue plan’ button.
7. Alternatively the CD pipeline is executed whenever a PR is merged in ADO.

Bonus : Setup CD pipeline in Azure DevOps

1. Add the actual deployment values for the variables in the `./input.tfvars` updating the values from the parent blueprints which have been specified as dependencies.
2. Identify the variables that will change based on env like the resource group.
3. Create a variable group with the name `<repo-name>-<env>` and add all the env specific variables in the variable group. Follow the documentation here for getting started with creating a variable group.
4. The blueprints pipeline repo has pipeline templates for the common use cases. Based on the use case, select of the template or create a custom template.
5. The CD pipeline template inherits variable groups mentioned here. Moreover, the spoke template have env specific shared the variable groups: `<env-spoke-vars>`
6. Make sure that the service connection has the desired permissions to create resources in the specified resource groups.
7. Create a CD pipeline:
   - Go to Pipeline section in Azure DevOps
   - Select the following: New Pipeline => Azure DevOps git => select your repo => Existing azure pipeline => select your branch => select your ci-pipeline.yml
   - Refer the document for the steps to create pipeline
8. Run the pipeline to deploy the resources.
9. Update the branch policy to trigger the CD pipeline on master merge. Follow the document here to update the build validation in the branch policies.