EPAM Cloud & DevOps practice

AZURE DEVOPS SERVICES

Final task.

HOME TASK DESCRIPTION

You are implementing a complete Azure DevOps CI-CD solution for an Azure infrastructure using Terraform and ASP.NET core application according to the following MS tutorial.

You should design and implement a solution which must meet the following requirements:

1. You must use Azure DevOps repositories.
2. Configure repository branching according to the one of the GitFlow or Trunk based flow.
3. You must use Terraform azurerm backend.
4. Use the following 4 environments: DEV, QA, UAT, PROD. Each environment must have the same infrastructure configuration.
5. Each infrastructure stage deployments should be performed to your subscription in Azure to an appropriate resource group named: 'rg-{RESOURCE_NAME_PREFIX}-{LOCATION}-{ENV}' which is defined in a terraform solution.
   {RESOURCE_NAME_PREFIX} it's basically some unique identifier of your project, project name or whatever, just make sure that it's unique.
   {LOCATION} mustn't contain spaces, for example 'westus2'.
   Use lowercase for all Azure resource names to avoid possible issues during deployment.
6. Pipelines must be in YAML format.
7. Configure ADO Environments and use deployment jobs for pipelines each time you need approvals to be set.
8. Each Terraform apply stage must require approval. Approver should have an ability to check Plan before approving.
9. You must use only variable groups to configure variables. Create a group for each environment to use for appropriate build/deployment stages. Encrypt sensitive data.
10. Allow trainer to access your Azure DevOps project and Azure resources with contributor level.

You should meet the following recommended best practices:

1. Configure ADO branch policies for all repositories.
2. Repository should have clear folder structure.
3. Last code version merged to default branch (develop – Gitflow, main - trunk).
4. Use Pull requests only to merge code to long-living branches.
5. Tests must fail pipeline in case of unsuccessful result.
6. Use scripts as a file whenever possible.
7. Any pipeline deployment task must be approved.
8. Pipeline artifact contains required files only.
9. Pipeline steps/jobs/stages, variables, files/folders naming etc. must be clear and indicate the purpose.
10. Use Azure DevOps predefined variables in pipeline.
11. No hardcode allowed.

Please do the following tasks according to the **requirements** and recommended **best practices:**

NOTE: Each task only describes the minimum which should be performed but it is not limited by it.

TASK 1.1

Infrastructure CI-CD pipeline.

1. Create a new repository with clear logical folder structure for Terraform solution to deploy infrastructure and copy provided Terraform solution. Configure git policy:
    a. Restrict deletion of long-living branches (develop, master and release/* if gitflow).
    b. Restrict direct commits to long-living branches.
    c. Allow merge with pull request only.
    d. Allow merge with only success builds.
    e. Reset approval vote in case of new changes.
    f. Pull request must contain linked work item.
    g. Build validation should be set up.
2. Manually create resource group in your Azure subscription 'rg-adolearn-{unique_project_name}-tfstate', storage account and container to store Terraform state files. Put resource group, storage account and container names to appropriate ADO variables using dedicated variable group, you will use it for the current and future modules.
   Don't remove this storage account till the end of the ADO course.
3. Create a service connection to your Azure subscription.
4. Create a new CI-CD pipeline with name **ADOLab-IaC-CI-CD** which contains the following stages (see description of stages below):
    - Build
    - Terraform Plan and Apply stages for each environment that must be triggered after Build stage. (use "TerraformInstaller@0" task to install Terraform on agent and "TerraformTaskV2@2" to run commands):
        a. DEV and QA must be triggered after new artifacts are created.
        b. UAT must be triggered after successful deployments on DEV and QA.
        c. PROD must be triggered only after successful deployment on UAT stage.
        d. Each apply stage should have an approval from 1 person before deployment except PROD – it should have 2 approvals
    - The main idea of splitting environment deployment process to 2 stages is to provide an opportunity to check executed plan before approving an apply stage. Therefore, apply stage must be triggered only after plan and appropriate confirmation.
5. Build stage should contain:
    - step which can validate your commit message (Commit message should start with {Project-code}-{work-item-number} your comment.
    - step, which updates Build Number for the running build with new name Infra-CI-CD-{yyyy.MM.dd.rev}. Set variable with old build number value to be used for generating archive name below.
    - Add step to install specified version of Terraform tool on your agent. Provide version as a variable from a dedicated variable group. Thus, you will be able to switch to new a version for all the pipelines just by changing variable value. (You probably will previously need to install Terraform extension to your organization, use this one.
    - Unit test: step which can check formatting of Terraform solution.
    - Unit test: step which can validate your terraform solution. Don't forget to run terraform init previously (that's why you had to create storage account for tfstate).
    - Copy required artifacts into build artifact folder on the agent.
      Tip: your artifact should contain only those files, which are required for further CD process.

- Create zip archive of your artifacts with the name iac.terraform.{version}.zip.
- Upload created zip archive into Azure Pipelines/TFS.

6. Terraform Plan (Apply) stages should contain:
   - Step to download build artifacts.
   - Step to unpack artifacts from the archive.
   - Step to Installing specific version of Terraform on agent.
   - Step to terraform init.
   - Step to run terraform plan (apply).
7. Configure triggers for the pipeline:
   - Your pipeline should be triggered with any commit to feature/*, bugfix/*, hotfix/*, develop, master, release/* branches (in case of the Gitflow).
   - In case of long-living branches (release/*, main and develop if Gitflow) artifact (zip archive) should be created, otherwise, just run Build stage without artifact creation and Terraform Plan.
8. Do not forget that some of the values should be passed to Terraform. Look through the Terraform code to figure out which.
9. Test your solution by deploying on DEV preliminary. When everything works fine just copy and rename stages. If stages configured properly, you don't need to change steps inside, use scope specific variable groups to provide environment specific adjustments.
10. Test your solution and deploy an infrastructure for all environments.

TASK 1.2

1. Make the following changes to Terraform solution by merging PR from feature branch and check how your automated CI-CD process works.
   Add key-value pair to 'app_settings' of your web app:
   - Key name - "Version".
   - The value should be passed using new Terraform variable with default value enabled. The default value itself is up to you.
2. If you use Gitflow:
   - Create a "release/1.1" branch. Update CI-CD pipeline to pass new parameter "Version". Parameter should be equal the value of the current release version number. Do not hardcode the value in pipeline code.

   If you use Trunk based flow:
   - Tag last commit in main branch as 1.1.0. Update CI-CD pipeline to pass new parameter "Version". Parameter should be equal the value of the tag. Do not hardcode the value in pipeline code.
3. Check if infrastructure has been updated for all environments.

TASK 1 ACCEPTANCE CRITERIA

1. All general requirements and best practices have been implemented.
2. Pipeline triggers configured according to requirements.
3. Step to validate commit message, update build number, check terraform format, terraform validate works properly.
4. Tests fail pipeline in case of unsuccessful result.
5. Scripts passed to pipeline as a file (where required).
6. Variable groups are used. All sensitive data encrypted.

7. Steps to create artifact and Terraform apply stages run from long-living branches only.
8. Terraform apply stages require approval.
9. Runs history contains automatically triggered pipeline from feature and long-living branches according to task requirements.
10. Infrastructure has “Version” app_setting, that correctly processed and passed to Terraform solution according to chosen flow.
11. Last pipeline’s build is successful.

TASK 2.1

Application CI-CD pipeline.

1. Use all general approaches and best practices from the previous task.
2. This task additionally requires the following tools to be installed:
   - Visual Studio Code.
   - Dotnet Core SDK v 6.0.
3. Clone the following GitHub repository with .NET Core application:

   https://github.com/Azure-Samples/msdocs-app-service-sqldb-dotnetcore

   into your local repository.
4. Reset the version of code you are using to the following:

   git reset --hard 3655d08a7503ce5ff3951a74e420afc639a8b7a8.
5. Rename your local “master” branch to “main”.
6. Create a new Azure DevOps repository for application code.
7. Connect your local repository to the Azure DevOps repository.
8. Commit and Push updated code to the Azure DevOps repo. Make sure you are using meaningful comments in the commit messages.
9. Create a new CI-CD pipeline with name **ADOLab-App-CI-CD** which contains the following stages (see description of stages below):
   - Build
   - Deploy App on staging slot of prod environment that must be triggered after previous stage.
   - Swap slots on Prod environment.
10. Add a "ASPNETCORE_ENVIRONMENT"=”Production” variable into your Pipeline, in order to tell your app to use Azure SQL database.
11. Import "ASPNETCORE_ENVIRONMENT"=”Production” variable into your app_settings in Azure WebApp by updating Terraform IaC solution or using intermediate PowerShell step in your CI/CD pipeline.
12. Build stage should contain:
    - Use .NET Core sdk 6.0.x task, select version 6.0.x, set it before all others.
    - Steps to validate commit message and update build number from previous task.
    - Steps to build application. You can use ASP.NET Core template.
    - Step to prepare SQL migration script. Use the following command: ‘dotnet ef migrations script -p <project file to use>.csproj -o <output file>.sql -i’ or use a matching snippet.
    - Steps to publish required artifacts. Make sure that “.sql” database migration script is published separately from the application code. You will need to use it to update your Azure SQL database. Also, make sure your application code artifact does not contain database migration script.

13. Deploy App stage should contain:
    - Step to update database schema.
    - Step to deploy application to the Azure App services Staging slot using published artifacts.
14. Swap slots stage should contain:
    - Step to swap Azure App services slots.
15. Configure pipeline triggers the same as for task1. Deploy App stages must be triggered from long-living branches only (release/*, main and develop if Gitflow).
16. Test your solution and deploy the application to previously created infrastructure.

TASK 2.2

1. Make the following changes to application files by merging PR from feature branch and check how your automated CI-CD process works:

   Views\Todos\Index.cshtml
2. Deploy the application in “Staging” slot and do not swap slots. Check that you have an updated version of your application running in the Staging slot and old one in Production slot.
3. Swap slots and check results.

TASK 2 ACCEPTANCE CRITERIA

1. All general requirements and best practices have been implemented.
2. Pipeline triggers configured according to requirements.
3. Step to validate commit message, update build number.
4. Tests fail pipeline in case of unsuccessful result.
5. Variable groups are used. All sensitive data encrypted.
6. Steps to create artifact and deploy application stages run from long-living branches only.
7. Deployment stages require approval.
8. Runs history contains automatically triggered pipeline from feature and long-living branches according to task requirements.
9. Last pipeline’s build is successful.
10. Application up and running.
11. Send the links of both Production and “Staging” slot to the trainer.

USEFUL LINKS

Version control system and GIT:

https://docs.microsoft.com/en-us/devops/develop/git/what-is-version-control

https://docs.microsoft.com/en-us/devops/develop/git/what-is-git

Git commands:

https://git-scm.com/docs/git

https://medium.com/flawless-app-stories/useful-git-commands-for-everyday-use-e1a4de64037d

https://www.edureka.co/blog/git-commands-with-example/

Workflows
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

https://www.toptal.com/software/trunk-based-development-git-flow

Terraform:

https://www.terraform.io/docs/index.html

Terraform code validation:

https://www.terraform.io/docs/cli/commands/validate.html

https://www.terraform.io/docs/cli/commands/fmt.html

Azure DevOps variables:

https://docs.microsoft.com/en-us/azure/devops/pipelines/build/variables?view=azure-devops&tabs=yaml

https://github.com/Microsoft/azure-pipelines-tasks/blob/master/docs/authoring/commands.md

Semantic versioning:

https://semver.org/

Azure Pipelines:

https://docs.microsoft.com/en-us/azure/devops/pipelines/?view=azure-devops

https://docs.microsoft.com/en-us/dotnet/architecture/devops-for-aspnet-developers/cicd?view=aspnetcore-2.2

Azure DevOps documentation (sections: Deployments, Concepts, How-to Guides):

https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/key-pipelines-concepts?toc=/azure/devops/pipelines/toc.json&bc=/azure/devops/boards/pipelines/breadcrumb/toc.json&view=azure-devops

Microsoft Naming Convention best-practices:

https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/ready/azure-best-practices/resource-naming

https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/ready/azure-best-practices/resource-abbreviations

Generate SQL scripts with EF Core migrations:
https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli#generate-sql-scripts

Code-First vs Model-First vs Database-First: Pros and Cons:

https://www.ryadel.com/en/code-first-model-first-database-first-vs-comparison-orm-asp-net-core-entity-framework-ef-data/