Tampere University of Applied Sciences

Creating Azure DevOps Pipelines for Web Application

Eetu Koskelainen

BACHELOR'S THESIS
April 2021

ICT Engineering
Software Engineering

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Software Engineering

KOSKELAINEN, EETU:
Creating Azure DevOps Pipelines for Web Application

Bachelor's thesis, 39 pages
April 2021

DevOps is a concept which combines practices from software development and IT operations. The DevOps automation in web development projects is becoming the de facto standard in the web development industry. Automation can solve problems related to repetitive and time-consuming tasks. It can reduce the possibility of human error, for example, in the application deployment phase.

This thesis provides directional instructions on creating automated pipelines for basic client-server web applications using Microsoft Azure cloud computing products and Pulumi infrastructure as code method-based cloud resource management tool. The instructions focus on the technical perspective of setting up pipelines for the client, server, and infrastructure sections of the application. Instructions and solutions introduced in the thesis are based on development experiences during the implementation phase and literature review of technical documentation, articles, and publications.

The pipelines were successfully created for each part of the application. Pipeline development was not completely straightforward due to constraints of the Azure DevOps platform. Even though fully dynamical pipeline architecture was not achieved, pipelines were developed to support all necessary features like multiple environments, stages and parametrization.

Keywords: DevOps, web application, Azure, pipelines, automation

CONTENTS

ABBREVIATIONS AND TERMS

Bash script	A set of terminal commands executed in order.
CD/CI	Continuous Delivery & Continuous Integration.
CLI	Command line tool.
Regression testing	Testing functionality of previously developed features.
Codebase	Source code of the application.
REST API	Representational state transfer application programming interface.
SQL	Structured Query Language. Used to operate relational database systems.
NPM	Javascript Package Manager. Node.js runtime environment package manager.
JSON	Javascript Object Notation. It is human-readable data format widely used in Web Applications.
Typescript	Superset of Javascript with static typing capabilities. Typescript is transcompiled to Javascript.

1 INTRODUCTION

Web development can be a wild west with very few predefined processes and guidelines. Sometimes development is done in a tight schedule, or the system has many manual tasks to be performed during the development cycle.

Manual procedures are easy to remember and do when performed daily. But for example, after six months of only maintenance, manual deployment without good documentation can be challenging to accomplish on the first try. Sometimes automation or environment configurations are lacking due to schedule or ignorance. There is a severe risk of an error when tasks are done manually, and that is what automation is trying to solve.

There are a lot of technical solutions and services to create automation pipelines. Same end results can be achieved with almost every solution. All of the technologies provide documentation how to use them individually, but comprehensive instructions are more rarely found.

Primary goal for this thesis is to provide directional instructions on using Azure products with Pulumi infrastructure as code tool to create automated pipelines for basic client-server Web application. Solutions introduced in this thesis are based on experiences from the implementation phase and literature review.

The document's first sections will present DevOps ideology, tools, technologies, and Azure Cloud resources used in this document. Latter parts focus heavily on the technical perspective and critical parts of the implementation. The reader is assumed to have a basic knowledge of modern software development tools and ideologies.

2 BACKGROUND

2.1 DevOps

DevOps is a set of practices meant to boost the software development cycle and reduce the time spent between developing new features and deploying them to production. DevOps term comes from the words Development and IT operations, and it means combining and automating the traditional responsibilities of both departments. (Loukides M. 2019)

In the past, development, testing, and deployment tasks were usually split between teams or units. Different units had various kinds of goals, and in the usual scenario, communications between the units were slow and inefficient. When the development lifecycle is slow and production deployments are made rarely, critical bugs and issues with the production deployments are often unavoidable. (Kim, Humble, Debois & Willis 2016, 15-28)

The core tool of DevOps is automation, which takes care of tasks that are repeated often. For example, necessary regression testing can be automated so that the software tester can focus on testing the new features.

DevOps practices usually include building automated pipelines that are executed after the developer changes the codebase. The pipelines can build, test, and deploy the code to the testing or development environment in minutes when traditional testing and deployment with manual processes would have taken days or at least several hours. If any phase of the automated pipeline fails, the developer gets immediate feedback on what task failed, and execution of the pipeline is stopped. (Kim, Humble, Debois & Willis 2016, 15-28)

An example development cycle of a software project is shown in Figure 1. The development cycle usually starts from the planning phase, where new features or tasks are planned and defined by all teams together. After a successful planning development team can start implementing planned features. The development team uses integration tools such as automated pipelines to build

software and run automatic tests after merging changes to the codebase to find possible regression or defects.

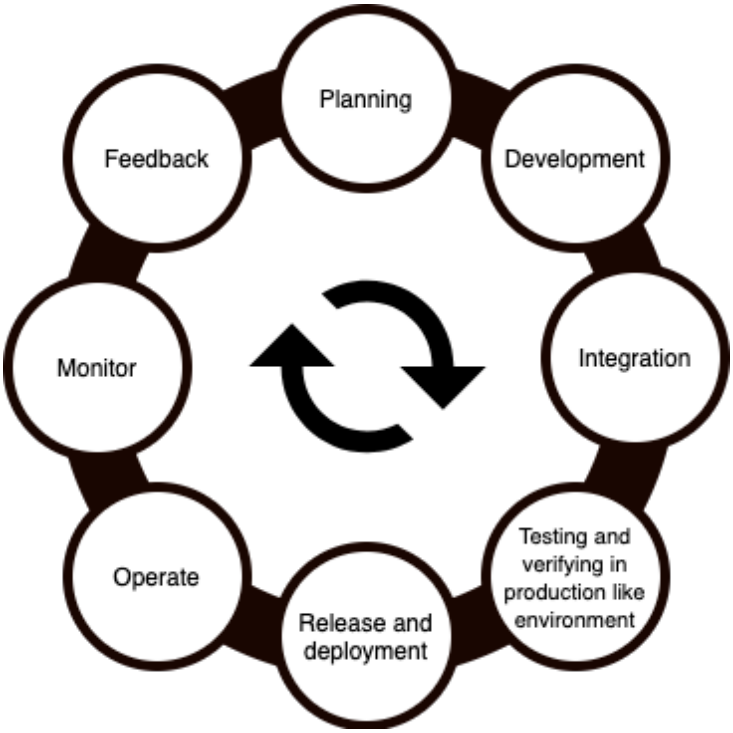

Figure 1. Example of the development cycle.

After the development phase, new features can be moved to a testing environment where testing of new features can be done by the team responsible for user testing. In the testing stage, issues or defects can be found, and the feature can be returned to development or even to the planning phase.

When features are successfully tested, new release and deployment can be made. In best case scenario any member of the development team should be able to do the release and deployment phase.

The last three phases are mostly executed simultaneously. Developed features are configured if needed, monitored for bugs or errors, and feedback of the features is gathered. After full DevOps cycle, developed features can be marked as completed.

For the development cycle to be efficient, each team or phase should have a designated environment to use. All environments should be a production like with similar data. Using separate environments for different use-cases has multiple benefits.

If multiple teams were using the same environment for testing and development, confusion or misunderstanding could happen. If the testing phase would be done in a development environment, failing database migration could stop testing work for hours or days.

An example of development flow between environments is shown in Figure 2. Environments and flow between them can vary very a lot depending on the product and teams developing it. Example shown in Figure 2 is from an average size project. The development environment is used mainly by developers when new features are developed. Testing environment is clients testing environment that has production-like data. A staging environment is used for final testing of the features that are going to be deployed to the production environment. The number of environments entirely depends on the development and release processes and project structure.

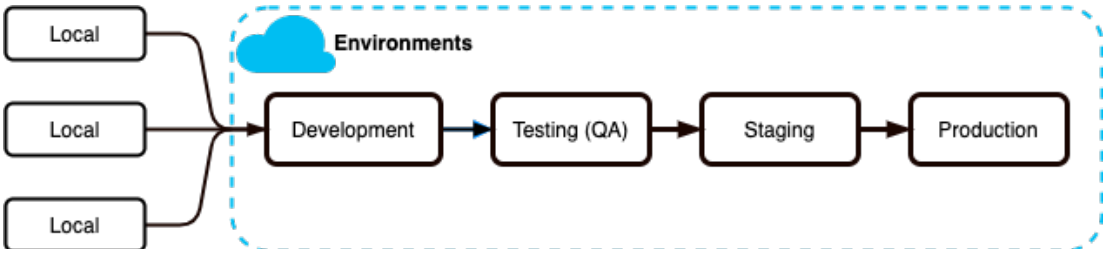

Figure 2. Multi-environment development flow example.

2.2 Cloud services

Cloud services enable the usage of shared computing resources remotely through the internet for an annual fee. Many different kinds of platforms offer disk space, databases, website hosting, remote servers, or all of them. Cloud services provide users with almost unlimited technical resources to build applications fast without seeing or thinking of any hardware. Cloud services enable applications to be scaled for big audiences faster than traditional server hosting.

When cloud services arrived at the market, the main selling point was cheaper storage and calculating power compared to the existing physical hardware. Nowadays, businesses mainly use cloud services because of the speed and ease of development. Most companies operating in the cloud have designated cloud strategies to achieve fast development and secure architecture without owning physical hardware or having a team to maintain it.
(Daniel Kirsch & Judith Hurwitz, 2020, Chapter 1)

In cloud computing, there are usually multiple different hosting models to choose the most suitable for the current need. Infrastructure as a Service (IaaS) is a self-service model where the customer rents virtualized servers or storage and pays for the time and resources used. Environment and software running on the server are usually entirely under the control of the user. (Daniel Kirsch & Judith Hurwitz, 2020, Chapter 8). IaaS model is usually the best option when high customization is needed, and the user is willing to maintain and update the environment routinely.

One abstraction layer to the IaaS is Platform as a Service (PaaS) model that offers middleware services for developers to deploy the applications to the cloud without maintaining the server environments or updates (Daniel Kirsch & Judith Hurwitz, 2020, Chapter 10). PaaS can offer different services from Web Application containers to Managed Database solutions. These services usually offer easy deployment options and configurable managed environments, making developing applications easy. The PaaS model's downside can be missing options for more customized applications or usually higher price points than the IaaS.

The highest point of abstraction is Software as a Service (SaaS) model, which usually offers complete products for end-users. An example of a SaaS service is the Azure DevOps platform, covered more in chapter 3.4. Software as a Service is a good option when the product meets the needs, and there is no need for higher customization. SaaS usually provides reliability and is a good option for business-critical use-cases.

The most significant and extensive Cloud Service Providers in 2021 are Amazon Web Services, Microsoft Azure, and Google Cloud. Each of the providers has a similar offering on essential services within IaaS, SaaS, and PaaS models. Amazon Web Service was the first major cloud computing provider in 2008, and it is known for the IaaS services and significant investments in the offering of databases, machine learning, and serverless resources. In comparison, the Microsoft Azure platform offers a grand scale of integrations between the Microsoft product family. The platform is widely used by companies relying heavily on other Microsoft products making service management more straightforward. Google Cloud offers a wide variety of SaaS products such as Google Drive, Sheets, Docs, and Meet, making its ecosystem intriguing for medium sized companies. Google Cloud also has a similar offering of IaaS and PaaS products as its competitors. (Dignan, 2021)

3 TOOLS AND TECHNOLOGIES

3.1 Version control system (Git)

In software development, there is often a team of developers working on the same project. All developers need to make changes to the same codebase and sometimes edit even the same files. For efficient working, there has to be a system that helps engineers avoid conflicts and retain the codebase history. Git version control system and GitHub platform are used in the example project introduced in chapter 4.

Distributed version control system such as Git tries to solve these problems by allowing each developer to work on their copy of the same codebase. Developers can make the changes to the code locally and merge the changes to the main codebase. The full history of changes can be tracked, and any version of the codebase can be recovered at any time. (Github.com, 2020)

3.2 Infrastructure as code (Pulumi)

Infrastructure as code (IaC) is a practice to define the application's infrastructure with code that can be stored in version control. The declarations code can be thought of as a list of resources needed for the application to be deployed. Most of the IaC tools provide a way to automatically deploy the infrastructure to the desired cloud without any manual configurations done on the cloud platform. (Stackify, 2019)

In the example project, the Pulumi tool is used to create the infrastructure declarations and deploy the desired infrastructure to the Microsoft Azure Cloud platform. Pulumi supports multiple programming languages and cloud providers.

3.3 Microsoft Azure

3.3.1 Key Vault

Azure Key vault is a service to store application secrets such as API tokens, passwords, or other secret information related to the application. It provides a centralized way to store secrets and fetch them when needed. Key vault will eliminate the need to save secret values to the applications code because the application can be granted access to fetch the wanted secret values from the designated key vault. (Azure Documentation, 2020)

3.3.2 App Service Plan and App Service

Azure App Service is a PaaS application platform to host modern Web and Mobile applications. It can run an application built with modern languages such as Node.js, Java, or Python. The developer can select the operating system of the App Service. App Service is a fully managed service, which means that application developers do not need to maintain or update the application's actual servers. (Kiriaty Y., 2017)

App Service Plan is needed to use App Services in Azure. App Service Plan defines resource limits and pricing point of the App Services. Multiple App Services can be included in the same App Service Plan. (Azure Documentation, 2020)

3.3.3 Managed SQL Server and Database

Azure SQL Database is a fully managed PaaS service that provides secure and administration free SQL Server and database to store application data. The database size can be easily scaled to meet the needs of the application.

Azure SQL Database is utilizing structured query language syntax which is commonly used on relational databases. Like all different SQL implementations, Azure SQL Database is based on the standard SQL syntax and has its own features. (Azure Documentation, 2021)

3.4 Azure DevOps

Azure DevOps is a SaaS product by Microsoft used to manage DevOps environments, environment variables, and automated pipelines. The platform also offers other features such as version control, agile tools, and test plans but these features are not covered in this document.

Azure DevOps offers tools to create automated pipelines to build, test and deploy applications. Automated pipelines can be created using YAML configuration language or with a user interface. Pipelines created with the user interface are called classic pipelines. Same functionalities can be achieved using either one, but pipelines configured with YAML are stored in version control and are that way more easily editable.

Applications are configured with environment variables. In Azure Pipelines, environment variables are stored in variable groups that can be accessed in the portal's Library section. Variables groups can be used in pipelines to configure the application. Variables can be imported from the Azure Key Vault or added manually from the portal.

Projects usually have multiple environments such as development, testing, and production. In Azure Pipelines, environments are used to track deployments made to the current environment and to create checks or approval rules that must be met before deployment can be made.

4 BUILDING EXAMPLE ENVIRONMENT AND PIPELINES

4.1 Prerequisites and goals

In this section, proof-of-concept implementation of DevOps environment for a small-scale web application project will be presented with directional instructions. Understanding all terms and conventions, readers should have the basic knowledge of the web applications, command-line usage, and tools used.

This proof-of-concept implementation's primary goal is to have automated pipelines that are triggered by version control changes. When the developer creates a new pull request or commits to the development or master branch, the build and test phases of pipelines will be run automatically. If the pipeline fails, the run will be stopped, and no further steps will be executed. If the pipeline succeeds, it requires approval from the user to deploy the current build to the desired environment.

4.2 Example app and environment

Pipelines or DevOps environments cannot be set without an application to build, test, and deploy. An example application used for this proof of concept will have a typical modern web application structure.

An example application is a straightforward counter which shows the current count and time of the latest increase. The application's user interface is shown in Picture 1. It has a button labeled “Increase” for the user to increase the counter number and "Fetch" labeled button to fetch the latest counter number and time of the action. Counter increase actions will be persisted in the database. As described in Figure 3, example application will consist of client web app, REST API server, key vault, and SQL database.

Picture 1. Web application example user interface.

Client web application is created with React which is a popular Javascript library to build user interfaces (React Documentation, 2021). Application sends and fetches data from the REST API Server. Server-side is Node.js application with REST API that will provide endpoints for clients to fetch and persist data to Azure SQL Database. Both App Services will be under the same Linux App Service Plan. All secrets, such as database connection string, are stored in the Azure Key Vault, where only the server will have access. Pulumi infrastructure as a code tool is used to manage the Azure components.

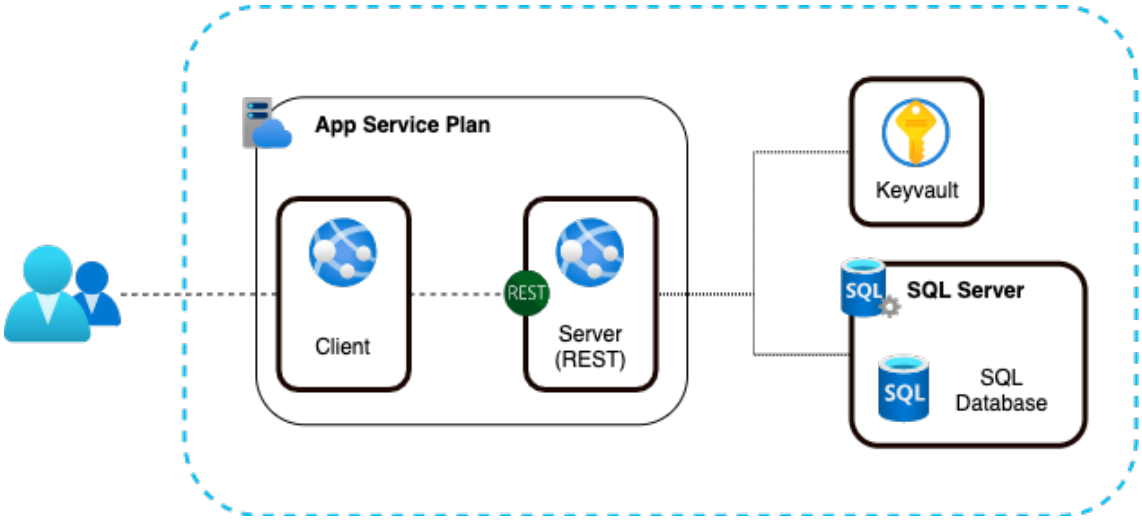

Figure 3. Architecture diagram of the example application.

The codebase of the whole system will be separated into three repositories in Github. Using separate repositories for client, server, and infrastructure code will be the best option for this proof of concept because each repository will have its own pipeline. Using a mono repository for all services can lead to more complex pipeline configurations.

4.3 Development environment and technical prerequisites

This proof-of-concept project is created using the OS X operating system on Macbook Pro, and all the command line tools or commands are run on OS X Terminal. All commands and tools used should have equivalents on Windows and Linux but are not presented in this document. The code editor used is Visual Studio Code, and the database tool is Azure Data Studio. Both of these tools are also available on Windows and Linux.

Before setting up the pipelines, services and tools must be taken into use. This example will assume that the executor has

- Microsoft Azure account with a trial period or pay-as-you-go subscription
- Pulumi account
- Github account
- Azure and Pulumi command-line tools installed
- two repositories with existing client and server apps and one empty repository for infrastructure

4.4 Manual configurations

Before project infrastructure can be implemented, specific manual steps must be done. That includes configuring services for use with pipelines and installing all the necessary development tools.

4.4.1 Creating Azure service principal account

Pulumi tool needs to access Azure to control and create resources. Azure service principal account must be created to grant Pulumi access to the desired Azure subscription. Pulumi could be configured with Azure account, but that is not recommended (Pulumi documentation, 2021).

Azure service principal accounts can be created via Azure CLI or Azure Portal. The Azure CLI command shown on the first line of Code 1 can be used to generate it.

```
az ad sp create-for-rbac --name serviceAccountName
{
  "appId": "appId-123456",
  "displayName": "serviceAccountName",
  "name": "http://serviceAccountName",
  "password": "password-123456",
  "tenant": "tenantId123456"
}
```

Code 1. Azure service principal creation command and output.

After running the command successfully, the service principal information should be printed to the terminal as shown in Code 1 starting from the second line. Azure Pipelines Github extension must be installed on all of the three repositories. Extension allows Azure pipelines to use the repositories as sources.

4.4.2 Creating new Pulumi project

In this example, Typescript is used as an infrastructure declaration language for Pulumi. Pulumi also supports other languages, but Typescript was chosen based on familiarity and good typing capabilities.

New Pulumi project can be created with the Pulumi CLI. Before the actual creation command, empty GitHub repository was cloned. All following code in this chapter is executed in the repository folder.

After cloning the repository, a new Pulumi project can be created with the CLI command shown in Code 2 below (Pulumi documentation, 2021).

```
pulumi new azure-typescript -n "azure-thesis-infrastructure" -d
"Example Pulumi project for thesis" -s "dev"
```

Code 2. Command for creating new Pulumi project.

In the command shown in Code 2, parameters are

- -n which is the name of the project
- -d is the description of the project
- -s is Pulumi stack name of the current project

After running the command, it will prompt the type of environment defaulted to public and the location selected for the Azure subscription. The location value of the Pulumi project and Azure subscription must be same.

When the command finishes execution new Pulumi project and stack have been created. New files and folders have been created, such as

- node_modules folder
- tsconfig.json
- package.json
- package-lock.json
- Pulumi.yaml

- index.ts

Most importantly, Pulumi.yaml file contains configurations regarding the project, and the index.ts file contains the infrastructure declarations. Other files and folders are related to Node.js environment and will not be covered in this document.

4.5 Setting up infrastructure

4.5.1 Environment

Before starting the development of the infrastructure with Pulumi, environment variables need to be set for Pulumi to access Azure subscription. Environment variables can be set with a .env file, for example.

Pulumi commands can be run locally without environment variables by logging into Azure and Pulumi command-line tools. However, using environment variables locally can help to make sure that the development environment is similar to the automated pipeline. Pipelines must use environment variables for accessing Azure and Pulumi.

Variables that need to be set to the .env file are

- Client id
- Client secret
- Tenant id
- Subscription id
- Pulumi access token

Three firstly mentioned can be obtained from the reply of the service principal account creation command shown in Code 1. Pulumi access token can be found from Pulumi website under account settings. Subscription id can be fetched with command shown in the first line of Code 3. The command should print out all of the subscriptions associated with the Azure account shown in Code 3.

```
az account list --output table
Name        CloudName    SubscriptionId                        State    IsDefault
----------  -----------  ------------------------------------  -------  -----------
Free Trial  AzureCloud   1234567-1234-1234-1234-1234567891011  Enabled  True
```

Code 3. Azure subscription id fetching command with example output.

Environment variables are set in the .env file. Inconveniently, the naming of the variables is not the same as in Azure. The mapping of the variables is shown in Code 4.

```
ARM_CLIENT_ID="appId"
ARM_CLIENT_SECRET="password"
ARM_TENANT_ID="tenant"
ARM_SUBSCRIPTION_ID="subscriptionId"
PULUMI_ACCESS_TOKEN="pulumiAccessToken"
```

Code 4. Example content of the .env file.

Variables can be set to current terminal session with the command shown in Code 5.

```
set -o allexport; source .env; set +o allexport
```

Code 5. Command to set variables listed in .env files to the current terminal session.

4.5.2 Infrastructure declaration as code

Infrastructure is declared with Typescript code in the index.ts file located in the root of the infrastructure repository. All resources of the project except some secrets in the Azure Keyvault are declared with code.

Firstly, needed Pulumi libraries are imported in the first three lines of Code 6. Pulumi library is needed for base functionalities, Azure library is needed to create and configure Azure resources, and the random library is used to create randomized tokens.

In the Code 6 prefix for the resources is defined by the first ten characters of the stack name. This prefix will ease identifying the environment of the resource (Microsoft documentation, 2020). In small-sized projects like this, naming conventions for resources are not critical. More significant infrastructures need naming conventions for the team to work with the resources efficiently. In this example, almost all resources are named with the same global name.

```
import * as pulumi from "@pulumi/pulumi";
import * as azure from "@pulumi/azure";
import * as random from "@pulumi/random";

// use first 10 characters of the stack name as prefix for resource names
e.g. "dev"
const prefix = pulumi.getStack().substring(0, 9);

// Global name for the services
const globalName = `${prefix}-azure-thesis`;

// Get azure config
const clientConfig = azure.core.getClientConfig({ async: true });
const tenantId = clientConfig.then((config) => config.tenantId);
const currentPrincipal = clientConfig.then((config) => config.objectId);
```

Code 6. First configurations in the infrastructure declaration file (index.ts).

The global name for the infrastructure is defined with the prefix and descriptive and unique name of the project. In the last section of Code 6, Tenant id and Object id are fetched from the used Azure subscription.

Before any actual resources can be declared, the resource group must be defined. The resource group will contain all resources of the environment. In the first section of Code 7, a resource group is created with the global name declared earlier, and the name is extracted from the object to be used in further declarations.

Variable globalName is used twice in Resource Group creation because resources have logical and physical names. The first argument in the ResourceGroup object is a logical name, and the name provided in the configuration object is physical. The logical name is used in the Pulumi abstraction layer, and the physical name is the Azure resource's real name. If physical name is not provided, Pulumi will automatically create it using the logical name and adding random sequence of numbers to the end to make it unique. This example name is already unique across the environments because of the environment identifier used in global name (Code 7).

In the second part of the Code 7, Key Vault is created. Standard Key Vault version is selected by setting skuName to “standard”. Access policy to the Key Vault for

the current Service Principal must be granted for it to be able to modify or delete the resource (Pulumi, 2020).

```
// Create an Azure Resource Group
const resourceGroup = new azure.core.ResourceGroup(globalName, {
  name: globalName,
  location: "NorthEurope",
});
const resourceGroupName = resourceGroup.name;

// Create a keyvalt for saving secrets
const keyVault = new azure.keyvault.KeyVault(globalName, {
  name: globalName,
  resourceGroupName,
  skuName: "standard",
  tenantId,
  accessPolicies: [
    {
      tenantId,
      objectId: currentPrincipal,
      secretPermissions: ["delete", "get", "list", "set"],
    },
  ],
});
```

Code 7. Declarations for the Resource Group and Key Vault.

SQL Server, Database, and admin user are declared in Code 8. The admin password is created using a random password generator. The SQL Server resource requires admin username and password as parameters. Free tier versions of the Server and database are created. Admin credentials for the SQL Server are stored to the Key Vault in the last section of Code 8.

```
// Create admin password with random generator
const adminPassword = new random.RandomPassword("password", {
  length: 24,
  special: true,
}).result;
const adminUsername = "sqladmin";

// Create SQL Server
const sqlServer = new azure.sql.SqlServer(globalName, {
  name: globalName,
  resourceGroupName,
  administratorLogin: adminUsername,
  administratorLoginPassword: adminPassword,
  version: "12.0",
});

// Create SQL Database for the server
const db = new azure.sql.Database(globalName, {
  name: globalName,
  resourceGroupName,
  serverName: sqlServer.name,
  edition: "Free",
});

// Add new secrets for sql server admin
const adminUsernameSecret = new azure.keyvault.Secret("sqlAdminUsername", {
  name: "sqlAdminUsername",
  keyVaultId: keyVault.id,
  value: adminUsername,
});
const adminPasswordSecret = new azure.keyvault.Secret("sqlAdminPassword", {
  name: "sqlAdminPassword",
  keyVaultId: keyVault.id,
  value: adminPassword,
});
```

Code 8. Resources created related to SQL Database.

App Service Plan for client and server App Services is declared in the first section of Code 9. It is created with the global name and Linux as an operating system. In the second section, client App Service is declared.

```
// App Service Plan for the app services
const appServicePlan = new azure.appservice.Plan(globalName, {
  name: globalName,
  resourceGroupName,
  kind: "Linux",
  reserved: true,
  location: "NorthEurope",
  sku: {
    tier: "Free",
    size: "F1",
  },
});
const appServicePlanId = appServicePlan.id;

const clientAppService = new azure.appservice.AppService(
  `${globalName}-client`,
  {
    name: `${globalName}-client`,
    appServicePlanId,
    resourceGroupName,
  }
);
```

Code 9. Declaration of the App Service Plan and Client App Service.

The final part of the infrastructure declaration file is shown in Code 10. Server App Service is declared similarly to the client App Service. The server application needs to use the Key Vault and have an Access Policy for it. It is created in the second section of Code 10. The server's identity must be set to “SystemAssigned” so that the Access Policy can be created. Azure creates an identity for the App Service automatically, and it can be referenced when creating the Access Policy to the Key Vault.

In the end of Code 10, an empty object is exported. Pulumi requires an object to be exported from the declaration file. The object could contain information related to the infrastructure, for example, resource names. Exported variables are not needed in this case, and an empty object is exported.

```
const serverAppService = new azure.appservice.AppService(
  `${globalName}-server`,
  {
    name: `${globalName}-server`,
    appServicePlanId,
    resourceGroupName,
    identity: {
      type: "SystemAssigned", // Must be on so that app service can have
system assigned access to keyvault etc.
    },
  }
);

// Access policy for server to access keyvault
const ServerKeyvaultAccessPolicy = new azure.keyvault.AccessPolicy(
  `${globalName}-server`,
  {
    keyVaultId: keyVault.id,
    objectId: serverAppService.identity.principalId,
    tenantId: serverAppService.identity.tenantId,
    secretPermissions: ["get"],
  }
);

export {};
```

Code 10. Declaration of the server App Service and Key Vault Access Policy for it.

4.6 Pipelines

In Azure Pipelines configuration, stages are the main sections of the pipelines. A stage is an unlimited group of jobs that are executed in the desired order. One job can include multiple tasks such as bash scripts, npm commands, or Azure extension executions. The advantage of Azure Pipelines is a great number of pipeline task extensions, for example, Azure Web App Deployment and good integration with the Azure environment.

Azure pipelines are often configured with .yaml files located in the root of the repository. Each of the pipelines must have its own configuration file, which defines the pipeline structure.

4.6.1 Infrastructure

The purpose of the infrastructure pipeline is to automatically preview infrastructure changes in the pull request and make the deployment automatic and reliable. The stages of the infrastructure pipeline are shown in Figure 4.

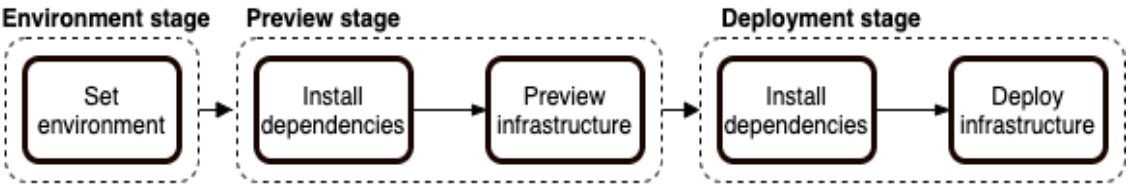

Figure 4. Stages of the application infrastructure pipeline.

The infrastructure pipeline is separated into three stages. The first executed stage is the environment stage, which sets the pipeline environment variables based on the branch name. Environment variables define the used Pulumi Stack and the Azure environment where the infrastructure is previewed and deployed.

The Preview stage is executed if the environment was set successfully. As a first job in the Preview stage, all necessary dependencies related to Pulumi and setup are installed. After a successful dependency installation, infrastructure is previewed. The infrastructure preview will compare the existing infrastructure to

the one defined in the current branch. The preview will show the changes that will be made to the infrastructure. If the infrastructure code has errors, the preview stage will fail, and pipeline execution will be stopped.

If the preview stage is executed without errors, the pipeline will prompt user permission to continue to the deployment stage. In the deployment stage, dependencies must be installed again because stages use different execution environments. After the dependencies are installed, infrastructure is deployed to the desired environment.

Pipeline for infrastructure repository is configured in azure-pipelines.yaml file located in the root directory. The first lines of the configuration file shown in Code 11 define when the pipeline execution should be triggered. In this example, develop branch is the main trigger, but all pull requests trigger the pipeline.

```
trigger:
  - develop

pr:
  branches:
    include:
      - "*"
```

Code 11. Pipeline trigger configuration.

The first stage of the pipeline is presented in Code 12. It includes a bash script that determines the environment to be used in the rest of the pipeline. Every pipeline has system variables that are related to the repository or the pipeline itself. Variables used to determine the environment are called Build.SourceBranchName and System.PullRequest.TargetBranch. First mentioned holds information of the source branch, which is the branch that triggered the current execution. If the pipeline is run for a pull request, the lastly mentioned system variable contains the target branch name.

```
stages:
  - stage: setEnvironment
    displayName: Set environment
    jobs:
      - job: setEnv
        displayName: Set environment
        steps:
          - script: |
              if [[ "$(Build.SourceBranchName)" == "develop" ||
"$(System.PullRequest.TargetBranch)" == "develop" ]]
              then
                echo "##vso[task.setvariable variable=STACK;isOutput=true]dev"
              fi
            name: setEnv
```

Code 12. First stage of the pipeline which sets environment variables.

Based on the branch name, ENVIRONMENT and STACK environment variables are set to desired environment values. In the bash script, pipeline variables can be set by echoing the command shown in Code 13.

```
##vso[task.setvariable variable=VARIABLE_NAME;isOutput=true]VARIABLE_VALUE.
```

Code 13. Command to set pipeline variable.

The second stage shown in Code 14 is used to make sure the infrastructure code does not have any errors and to preview the changes to be made before applying them to the current infrastructure. STACK variable set in the previous stage is used in the Pulumi@1 task. Variable group named common is added to provide the Azure and Pulumi access tokens.

```
- stage: preview
    displayName: Preview infra
    variables:
      - group: common
      - name: STACK
        value: $[ stageDependencies.setEnvironment.setEnv.outputs['setEnv.STACK'] ]
    jobs:
      - job: previewInfra
        displayName: Preview infra (pulumi)
        steps:
          - task: Npm@1
            displayName: Pulumi npm install
            inputs:
              command: install
          - task: Pulumi@1
            displayName: Pulumi infra preview
            inputs:
              azureSubscription: "Azure thesis resource manager"
              command: "preview"
              stack: $(STACK)
```

Code 14. Preview stage of the infrastructure pipeline.

The preview stage's first task is dependency installation, where all libraries are installed to the pipeline workspace. After successful installation, Pulumi Azure Extension is used to run the preview command with the stack defined by the environment variable.

The last stage of the pipeline is deployment where infrastructure changes are deployed to the real environment. In the example shown in Code 15 deployment stage is executed only if the source branch name equals “develop”. This conditional statement makes sure that changes from the pull requests will not be deployed to the real environment. Deployment depends on the success of the preview stage.

```
- stage: "deployDev"
    displayName: "Deploy to dev"
    dependsOn: preview
    condition: and(succeeded(),
eq(variables['Build.SourceBranchName'], 'develop'))
    jobs:
      - template: "azure-pipelines.deploy.yaml"
        parameters:
          commonVariablesGroup: "common"
          environment: "infra-dev"
          stack: "dev"
```

Code 15. Deployment stage of the infrastructure pipeline.

The deployment stage uses a template file, which can be thought of as a function in programming. Template executes steps based on the parameters it receives. The template is defined in a file named azure-pipelines.deploy.yaml. It receives common variables group, deployment environment, and stack name as parameters.

Deployment template file content is shown in Code 16. Template file parameters are passed to the deployment job, which will deploy the infrastructure. Tasks run in the job are almost identical to the preview stage apart from the Pulumi command that is up instead of preview. Command “up” will apply the changes to the current infrastructure.

```
parameters:
  - name: commonVariablesGroup
    type: string
  - name: environment
    type: string
  - name: stack
    type: string
jobs:
  - deployment: deployInfra
    displayName: "Deploy infra"
    variables:
      - group: ${{ parameters.commonVariablesGroup }}
    environment: ${{ parameters.environment }}
    strategy:
      runOnce:
        deploy:
          steps:
            - task: Npm@1
              displayName: Pulumi npm install
              inputs:
                command: install
            - task: Pulumi@1
              displayName: Pulumi up
              inputs:
                azureSubscription: "Azure thesis resource manager"
                command: "up"
                args: --yes
                stack: ${{ parameters.stack }}
```

Code 16. Infrastructure pipeline deployment template file.

Full source code for the infrastructure pipeline configuration can be found in GitHub repository azure-thesis-infra. https://github.com/EetuK/azure-thesis-infra.

4.6.2 Server

The server pipeline has two stages, which are the build and test stage and deployment stage, as shown in Figure 5. The build and test stage is started with installing dependencies needed for the build itself and testing. After dependency installation, automatic tests are executed. If tests are failed, the execution of the pipeline will stop.

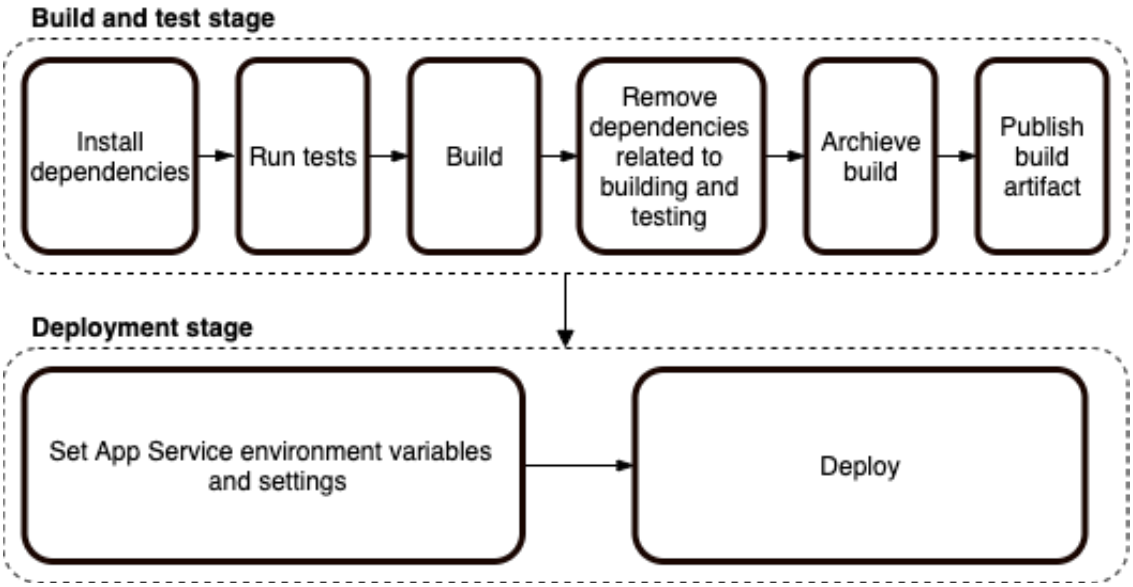

Figure 5. Stages of the server pipeline.

Build job will be started after successful test results. Build job builds the server's production version, and after a successful build, all dependencies related to the testing or building phases are removed. The build is archived in ZIP format, and it is published as a pipeline artifact. The deployment stage cannot access the resources created in the build and test stage, so the build artifact must be published to the pipeline's global space so that the deployment stage can access it.

In the deployment stage, environment variables and settings are configured to the app service where the server is deployed. After successful configuration, the build artifact is deployed to the app service, and execution of the pipeline is finished.

The server pipeline configuration file starts by defining pipeline triggers identically to the Code 11 explained in the Infrastructure chapter 4.6.1. In the current configuration, automatic execution of the pipeline is triggered when changes are made to the develop branch or pull request is created. The definition of the first stage is shown in Code 17. Stages are defined as list and are executed in the listed order by default.

```
stages:
  - stage: build
    displayName: Build
    jobs:
      - job: buildServer
        displayName: Build Server
        cancelTimeoutInMinutes: 15
        steps:
          - task: Npm@1
            displayName: "npm ci"
            inputs:
              command: "custom"
              customCommand: "ci"
          - task: Npm@1
            displayName: "npm run test"
            inputs:
              command: "custom"
              customCommand: "run test"
          - task: Npm@1
            displayName: "npm run build"
            inputs:
              command: "custom"
              customCommand: "run build"
          - task: Npm@1
            displayName: "npm prune production"
            inputs:
              command: "custom"
              customCommand: "prune --production"

          - task: ArchiveFiles@2
            inputs:
              rootFolderOrFile: "$(System.DefaultWorkingDirectory)"
              includeRootFolder: false
              archiveType: "zip"
              archiveFile: "$(Build.ArtifactStagingDirectory)/$(Build.BuildId).zip"
              replaceExistingArchive: true

          - task: PublishPipelineArtifact@1
            inputs:
              targetPath: "$(Build.ArtifactStagingDirectory)/$(Build.BuildId).zip"
              artifact: "server-$(Build.BuildId)"
              publishLocation: "pipeline"

                azureSubscription: "Azure thesis resource manager"
                command: "up"
                args: --yes
                stack: ${{ parameters.stack }}
```

Code 17. Build stage of the server pipeline.

The build stage includes one job, which includes all tasks needed. The three tasks install dependencies, run tests and build the production version of the server. The fourth task removes dependencies that are used for testing and are not needed for the actual server functionalities. After building server, files are published to the pipeline as a ZIP artifact. Timeout for the server building job is

configured to be 15 minutes for this project. Value of the timeout depends on the size of the project and how long the job takes in normal conditions.

The deployment stage is shown in Code 18. The first phase of the deployment stage uses Azure pipelines extension AzureAppServiceSetting@1, which alters App Service configurations. The extension needs Azure subscription name, app name, resource group name and desired app settings as parameters. App settings are set as JSON array of objects which include environment variable name and value. Environment variables are provided by variable group named server-dev.

```
- stage: deploy
    displayName: Deploy
    dependsOn: build
    jobs:
      - deployment: deployServer
        displayName: Deploy server
        environment: server-dev
        variables:
          - group: server-dev
        strategy:
          runOnce:
            deploy:
              steps:
                - task: AzureAppServiceSettings@1
                  displayName: Azure App Service Settings
                  inputs:
                    azureSubscription: "Azure thesis resource manager"
                    appName: "dev-azure-thesis-server"
                    resourceGroupName: "dev-azure-thesis"
                    appSettings: |
                      [
                        {
                          "name": "DB_USER",
                          "value": "$(DB-USER)",
                          "slotSetting": false
                        },
                        {
                          "name": "DB_PASSWORD",
                          "value": "$(DB-PASSWORD)",
                          "slotSetting": false
                        },
                        {
                          "name": "DB_SERVER",
                          "value": "$(DB-SERVER)",
                          "slotSetting": false
                        },
                        {
                          "name": "DB_NAME",
                          "value": "$(DB-NAME)",
                          "slotSetting": false
                        },
                        {
                          "name": "PORT",
                          "value": "8080",
                          "slotSettings": false
                        }
                      ]
                - task: AzureRmWebAppDeployment@4
                  inputs:
                    ConnectionType: "AzureRM"
                    azureSubscription: "Azure thesis resource manager"
                    appType: "webAppLinux"
                    WebAppName: "dev-azure-thesis-server"
                    packageForLinux: "$(Pipeline.Workspace)/server-
$(Build.BuildId)/*.zip"
                    RuntimeStack: "NODE|12-lts"
                    StartupCommand: "npm start"
```

Code 18. Deployment stage of the server pipeline.

The second task in the stage uses AzureRmWebAppDeployment@4 extension, which is used to deploy the build artifact to App Service. Inputs for the task are

- connectionType which defines the type of authorization used to connect to the web app. Value "AzureRM" means Azure Resource Manager service connection.
- azureSubscription is the name of the service connection to the Azure subscription created in the Azure Pipelines portal.
- appType which defines the type and operating system of the web app.
- WebAppName contains the exact name of the App Service resource in the Azure.
- packageForLinux which contains the path to the artifact to be deployed.
- RuntimeStack which defines the environment where the web app is executed.
- StartUpCommand which defines the command to start the web app.

The task deploys the ZIP artifact to the App Service, extracts it, and runs the startup command. Full source code for the server pipeline configuration can be found in GitHub repository azure-thesis-server. https://github.com/EetuK/azure-thesis-server.

4.6.3 Client

The client pipeline structure is almost exactly like the server pipeline, and all jobs with similar naming have the same functionality as the server pipeline in chapter 4.6.2. From Figure 6 can be seen that the building process differs from the server pipeline only by dependency removal job. The client build command will automatically include only necessary dependencies to the build.

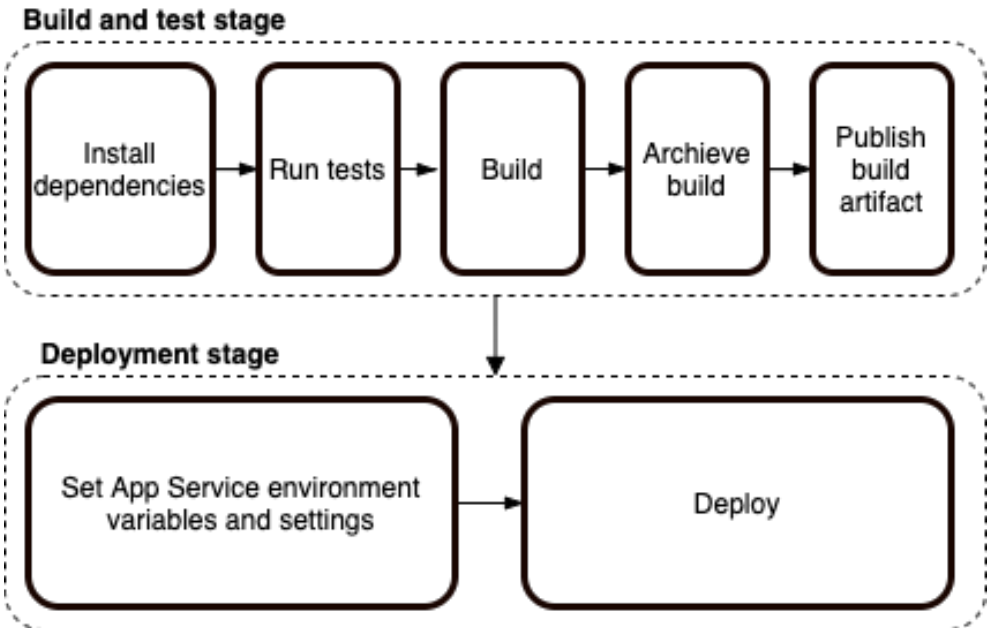

Figure 6. Stages of the client pipeline.

The client code will not be explained to avoid repetition because of the similarities between client and server pipeline setup. Full source code for the client pipeline configuration can be found in GitHub repository azure-thesis-client. https://github.com/EetuK/azure-thesis-client.

5 TESTING

Before the pipelines can be taken into use, functionality must be tested end-to-end. Testing the pipelines is performed by creating scenarios where pipeline should produce the expected outcome. All testing scenarios and results are shown is the Table 1.

The testing scenarios are the most usual circumstances developer will face when using automated pipelines as part of the daily work. All possible scenarios cannot be tested because of limited time and capabilities of the platform. The most important functionality with the pipelines is that execution will be stopped when any unexpected error occurs. All scenarios in Table 1 produced expected outcomes and were successfully passed.

Table 1. Pipelines testing scenarios and results.

Testing scenario	**Expected result**	**Pipeline**	**Passed**
Change (commit) is made to the develop branch.	Pipeline is triggered automatically. All stages are executed successfully.	Client	Yes
		Server	Yes
		Infrastructure	Yes
Pull Request is made with develop as target branch.	Pipeline is triggered automatically. Building stage is executed.	Client	Yes
		Server	Yes
Pull Request is made with develop as target branch.	Pipeline is triggered automatically. Set environment and preview stages are executed.	Infrastructure	Yes
Erroneous change (commit) or pull request is made that results as failure in the automatic tests or preview.	Pipeline execution is stopped after testing or preview phase raises error.	Client	Yes
		Server	Yes
		Infrastructure	Yes

6 CONCLUSIONS

This study aimed to create and document the creation process of DevOps environments and automated CI/CD pipelines for Web application with client and server structure -components with separate infrastructure repository. The goal was achieved, and directional documentation was successfully created. All three repositories have designated multi-environment pipelines that can be parameterized for each environment. Almost all of the often-repeated tasks were automatized, but most of the one-time tasks related to project creation must still be done manually because automation will not pay the effort.

However, the implementation process was not straight forward all the time. The documentation provided for YAML-pipelines was often causing confusion with the discrepancy. YAML configuration in Azure Pipelines is a fairly new feature, and it lacks some features that could be beneficial with more complicated pipelines.

Setting pipeline environment variables by the branch is possible already, but its implementation is inconvenient. Trying to implement a fully dynamic pipeline configured with multiple branches can be tricky and time-consuming. The easier way is to abstract, for example, the deployment to the template file and configure separate stages for each environment. More general good practices are needed because there are only separate guidelines available for each technology at the moment.

Due to the limited timeframe and resources used creating this document, there are some areas of improvement for the future. Documentation of the setup phase of the pipelines and repositories could be more extensive and precise. Azure Pipelines platform and its features could have also been introduced more widely to make documentation more comprehensive.

REFERENCES

Azure Documentation. 2020. Azure Key Vault Overview - Azure Key Vault. Accessed 24.1.2021. https://docs.microsoft.com/en-us/azure/key-vault/general/overview

Azure Documentation. 2020. App Service plans - Azure App Service. Accessed 24.1.2021. https://docs.microsoft.com/en-us/azure/app-service/overview-hosting-plans

Azure Documentation. 2020. Managed identities for Azure resources. Accessed 6.1.2021. https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/overview

Azure Documentation. 2020. Azure App Service Settings task - Azure Pipelines. Accessed 2.1.2021. https://docs.microsoft.com/en-us/azure/devops/pipelines/tasks/deploy/azure-app-service-settings?view=azure-devops

Azure Documentation. 2020. What is Azure SQL Database? Accessed 14.4.2021. https://docs.microsoft.com/en-us/azure/azure-sql/database/sql-database-paas-overview

Dignan, L. 2021 Top cloud providers in 2021: AWS, Microsoft Azure, and Google Cloud, hybrid, SaaS players. Accessed 1.2.2021. https://www.zdnet.com/article/the-top-cloud-providers-of-2021-aws-microsoft-azure-google-cloud-hybrid-saas/

Github.com. 2020. Git Handbook · GitHub Guides. Accessed 24.1.2021. https://guides.github.com/introduction/git-handbook/

Karhunen, J. 2020. Azure App Service - “503 Service Unavailable.” Accessed 2.1.2021. https://janik6n.net/azure-app-service-503-service-unavailable

Kim G., Humble J., Debois P. & Willis J. 2016. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. Portland: IT Revolution Press, LLC

Kirsch, D. & Hurwitz J., 2020. Cloud Computing for Dummies. John Wiley & Sons, Inc., Hoboken, New Jersey.

Loukies, M. 2019. What is DevOps? Accessed 24.1.2021. https://learning.oreilly.com/library/view/what-is-devops/9781449340346/ch01.html

Loukides, M. 2014. Revisiting What is DevOps. Accessed 24.1.2021.http://radar.oreilly.com/2014/06/revisiting-what-is-devops.html

Microsoft Documentation. 2020. Define your naming convention - Cloud Adoption Framework. Accessed 6.1.2021. https://docs.microsoft.com/en-

us/azure/cloud-adoption-framework/ready/azure-best-practices/resource-naming

Pulumi. 2021. Create a New Project | Azure. Accessed 2.1.2021. https://www.pulumi.com/docs/get-started/azure/create-project/

Pulumi. 2021. Azure Setup. Accessed 3.1.2021. https://www.pulumi.com/docs/intro/cloud-providers/azure/setup/.

Pulumi. 2019. Infrastructure as Code Resource Naming. Accessed 6.1.2021. https://www.pulumi.com/blog/infrastructure-as-code-resource-naming/

Pulumi. 2019. 7 Ways to Deal with Application Secrets in Azure. Accessed 6.1.2021. https://www.pulumi.com/blog/7-ways-to-deal-with-application-secrets-in-azure

React documentation. 2021. Getting started. Accessed 13.2.2021. https://reactjs.org/

Stackify. 2019. What Is Infrastructure as Code? How It Works, Best Practices, Tutorials. Accessed 24.1.2021. https://stackify.com/what-is-infrastructure-as-code-how-it-works-best-practices-tutorials/

Yochay, K. 2017. Azure - Inside the Azure App Service Architecture. Accessed 24.1.2021. https://docs.microsoft.com/en-us/archive/msdn-magazine/2017/february/azure-inside-the-azure-app-service-architecture