Documentation MixIT #1

Team members:

Hebe Kamer  
Senna van der Klaauw  
Lucas Quakkelaar  
Jung Kim  
Stef Bijen  
Lara Doorenspleet  
Jasper Houweling  
Noah Ruiters

Table of Contents

[1 Introduction 3](#_Toc157072051)

[1.1 Flowchart 3](#_Toc157072052)

[1.2 Testing locally 5](#_Toc157072053)

[2 Front-end 7](#_Toc157072054)

[2.1 Bootstrap 7](#_Toc157072055)

[2.2 Google maps link 9](#_Toc157072056)

[3 Database 11](#_Toc157072057)

[4 API 14](#_Toc157072058)

[4.1 Overview 14](#_Toc157072059)

[4.2 Setup 15](#_Toc157072060)

[5 Terraform 17](#_Toc157072061)

[5.1 Introduction to Terraform 17](#_Toc157072062)

[5.2 Environment Setup 18](#_Toc157072063)

[5.3 Code Explanation 18](#_Toc157072064)

[6 Key vault 27](#_Toc157072065)

[6.1 Overview 27](#_Toc157072066)

[6.2 Purpose of the Key Vault 27](#_Toc157072067)

[6.3 Test locally 29](#_Toc157072068)

[6.4 CI/CD integration 29](#_Toc157072069)

[7 CI/CD 30](#_Toc157072070)

[8 Add function 34](#_Toc157072071)

[8.1 Azure AI 34](#_Toc157072072)

[8.2 Infobip Base URL 35](#_Toc157072073)

[8.3 Differences 35](#_Toc157072074)

[8 Downloads 37](#_Toc157072075)

# 1 Introduction

Welcome to our innovative project with MixIT, where we have developed a Proof of Concept to enhance the efficiency and effectiveness of healthcare processes. MixIT, standing for Medical Information eXchange and Integration Technology, plays a crucial role in streamlining communication and information exchange within the medical sector. In this document we will explain our proof of concept in detail.

**Proof of Concept Scenarios:**

Our Proof of Concept focuses on addressing several critical scenarios to enhance day-to-day operations in healthcare:

1. *Summary of the Patient:*

Employees now have the ability to quickly obtain a summary of any patient. This includes not only basic information such as name, address, and route but also details from a separate system. Additionally, the last recorded policy is displayed along with a standard medication list.

1. *Changes in Patient Scheduling:*

Employees are immediately notified of changes in patient scheduling. They gain insights into appointments that are canceled, new appointments added, and special notes applicable to each patient, such as the required equipment.

1. *Scheduling Appointments:*

The system allows reporting remarkable situations during a visit to a physician promptly. Employees can automatically schedule an appointment. Moreover, the link to the medical record is automatically included with the appointment.

1. *Standard 'Bed Cleaning' Protocol:*

For patients requiring regular bed changing, a standard 'Bed Cleaning' protocol is added to every appointment. This ensures that crucial tasks are consistently performed.

This Proof of Concept is a step toward a more integrated, responsive, and streamlined healthcare environment, utilizing technology to enhance patient care and alleviate the burden on healthcare professionals. We are excited about the possibilities this project presents for the future of healthcare.

## 1.1 Flowchart

In our solution, the architecture is meticulously designed to ensure seamless communication and interaction among various components, both within the Azure Resource Group and beyond, as displayed in the following picture.

A diagram of software development

Description automatically generated with medium confidence

Within the Azure Resource Group:

1. Webapp:

* Our web application serves as the user interface, connecting users to critical healthcare information.
* *Connections:* It seamlessly interacts with the Key Vault for secure storage of sensitive information and interfaces with both the Apotheek API and Patient API.

1. Apotheek API:

* This API is our bridge to an external Apotheek Database, bringing in valuable medication information.
* *Connections:* It establishes a connection with the external Apotheek Database and collaborates with the webapp to deliver essential data.

1. Patient API:

* The Patient API is responsible for retrieving patient information from our local Patient Database.
* *Connections:* It communicates with the local Patient Database and shares data with the webapp.

1. Patient Database:

* Our local repository housing detailed patient information.
* *Connections:* Accessed and queried by the Patient API to provide real-time patient data.

1. Key Vault:

* The fortified vault securing our application's secrets and keys.
* *Connections:* Linked with both the webapp and APIs, ensuring sensitive information is stored securely.

Beyond the Azure Resource Group:

1. Apotheek Database:

* An external database holding crucial medication details.
* *Connections*: Accessed by the Apotheek API in our resource group, facilitating the extraction of medication information for a comprehensive healthcare picture.

1. Outlook Calendar via Microsoft Graph API:

* We've integrated an Outlook Calendar, enriching our user experience.
* *Connections*: Empowered by the Microsoft Graph API, our webapp seamlessly manages appointments, offering features like creating/editing appointments and fetching appointments within a specific timeframe.

1. Maps via HTTP GET Request:

* Enhancing user convenience, we've integrated Maps functionality.
* *Connections*: Activated through an HTTP GET request from the webapp, allowing users to navigate to specific addresses with ease.

1. App Registration with Enterprise Tenant:

* Our app registration, coupled with an Enterprise Tenant, enhances user capabilities through the Microsoft Graph API.
* *Connections*: By granting logged-in users Calendar ReadWrite-rights, our webapp can seamlessly integrate with Outlook Calendar for comprehensive appointment management.

## 1.2 Testing locally

When you modify the code, it is a lot of work to test everything via the pipeline, so we recommend that you test everything locally for functionality. To run our application locally, you'll need to ensure that all the necessary Python packages and their versions are correctly set up. We've streamlined this process by encapsulating the dependencies in a ‘**requirements.txt**’file. Here's how you can get everything up and running:

1. Fetching the ‘requirements.txt’ file:

The file can be found in GitHub repository (**mixit-1/flaskApp.Folder/Flask.App**)

1. Installing the required packages:

The required packages can be installed by opening a terminal window in the directory where the ‘requirement.txt’ file is located. By executing the pip install -r requirements.txt command all the required packages will be installed.

The application can be run in two ways:

1. Webapp Only:

If you only want to run the webapp, to check on modifications you’ve made for example, a terminal window needs to be opened in the directory where the app.py is located. In this instance that directory is **mixit-1/flaskApp.Folder/Flask.App**. To run the app.py file the command **flask run** will have to be run.

1. Full application with API’s:

To test the application locally with the two API’s the following things will have to be done:

* The API’s will have to start. This can be achieved by navigating to the correct directory for the API’s and starting them using the **uvicorn main:app --reload** command.
* For the Apotheek API you’ll have to navigate to the **mixit-1/flaskApp.Folder/Mixit-apotheek.Api** directory. This API runs on port 4000. The command that needs to be used to start the API is: **uvicorn main:app --reload --port 4000**
* For the Patient API you’ll have to navigate to the **mixit-1/flaskApp.Folder/Mixit-patient.Api** directory. This API runs on port 8000. The command that needs to be used to start the API is: **uvicorn main:app --reload --port 8000**
* Now the webapp can be started using the **flask run** command.

# 2 Front-end

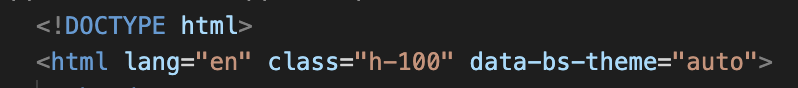
## 2.1 Bootstrap

Our application leverages the power of Bootstrap for a sleek and responsive user interface. The core of our design lies in the ‘**base.html**’ file, acting as the foundation for every other HTML file in our project.

The file encapsulates the overall layout of our application. Key components are:

1. **DOCTYPE Declaration:**

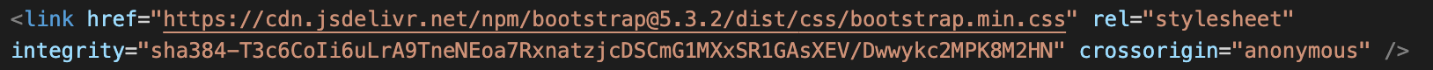
The DOCTYPE declaration is an essential and standard declaration at the beginning of an HTML document. In our case, it's declared as:



* The **lang="en***"* attribute specifies that the content is in English, aiding accessibility tools.
* The **class="h-100"** ensures that the HTML element takes up the full height of the viewport, contributing to a responsive design.
* The **data-bs-theme="auto"** attribute introduces Bootstrap's automatic dark mode switching feature, enhancing user experience.

1. **Bootstrap CSS CDN link:**

The Bootstrap CSS CDN (Content Delivery Network) link is crucial for styling and responsiveness. It's included with the following code:



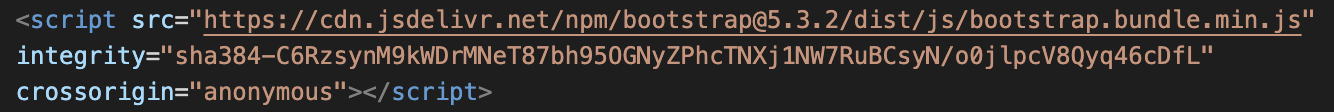
* **<link>:** This HTML tag is used to link an external resource, in this case, the Bootstrap CSS file.
* **href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css":** Specifies the URL of the Bootstrap CSS file on the CDN.
* **Integrity:** Provides a Subresource Integrity (SRI) hash to ensure the integrity of the file, preventing unauthorized alterations.
* **crossorigin="anonymous":** Specifies that the resource is loaded anonymously, preventing the transmission of user credentials.

1. **Navigation Bar:**

The navigation bar is a fundamental component for user navigation and branding. The navigation bar is defined by the **<nav>**element. It employs the Bootstrap class navbar for the basic styling and layout of the navigation bar. The style attribute sets the background color to a soothing shade, enhancing the application's aesthetic. Key elements include:

* Branding: The Mix-IT logo and application title.
* Responsive Navigation: A collapsible menu for smaller screens.
* Navigation Links: Hyperlinks to different pages of the application, dynamically updating based on user authentication and configuration.
* Logout Button: A styled button with a Mixit logo, offering a consistent and visually appealing logout option.

1. **Bootstrap JavaScript CDN Link:**

****

* This script tag links to the minified Bootstrap JavaScript bundle hosted on the CDN.
* The JavaScript bundle includes Bootstrap's functionalities like modals, tooltips, and dropdowns, adding interactive elements to our application.
* Similar to the CSS link, it includes an integrity check to ensure the authenticity of the file.

1. **Content Blocks:**

The **{% block content %}** and related blocks act as placeholders. When extending **base.html**, developers can populate these blocks with unique content for each page.

1. **Dynamic Title and Head Block:**

The **{% block title %}** and **{% block head %}** sections allow dynamic generation of HTML title tags based on the specific content of each page. This ensures a descriptive and context-aware title for each webpage.

1. **Footer:**

The footer is thoughtfully designed with Bootstrap classes. It includes:

* A Mix-IT logo.
* Copyright information with the year and team name.
* Positioned at the bottom of the page, regardless of content length, for a polished appearance.

1. **JavaScripts and Axios Integration:**

The **base.html** file incorporates essential JavaScript libraries and Axios for asynchronous HTTP requests to our Apotheek and Patient APIs. API URLs are dynamically set based on configuration.

1. **jQuery Inclusion:**

jQuery, a fast and feature-rich JavaScript library, is included via CDN for DOM manipulation and event handling. It complements Bootstrap's JavaScript functionalities.

1. **Custom Javascript:**

Additional custom JavaScript logic can be inserted within the **{% block js %}** section.

In summary, our ‘**base.html**’ file serves as the architectural blueprint, seamlessly integrating Bootstrap, JavaScript, and Axios to provide a cohesive and visually appealing user experience.

**Inclusion in HTML files:**

In our web application, we maintain a unified and consistent layout across multiple pages through a mechanism known as template inheritance. This is facilitated by the **{% extends 'base.html' %}** statement placed at the beginning of each HTML file.

1. Template Inheritance:

* The **{% extends 'base.html' %}** statement signifies template inheritance in the context of Flask, the web framework we are utilizing.
* Template inheritance allows for the creation of a base template (in this case, **base.html**) that serves as a blueprint for the overall structure and layout of the application.
* Other HTML files, representing individual pages, can then extend or inherit from this base template. This inheritance mechanism ensures a consistent look and feel across all pages.

1. Defining Content Blocks:

* Within **base.html**, certain areas are marked as content blocks using **{% block content %},** **{% block title %}**, and **{% block head %}** statements.
* When an HTML file extends base.html, these content blocks act as placeholders that can be filled with unique content specific to that page.
* For example, the **{% block content %}** section in **base.html** defines where the main content of the page should appear.

1. Customization per Page:

* By extending **base.html**, each HTML file can focus solely on the unique content and features of that particular page without the need to replicate the entire HTML structure.
* Developers can override the content blocks defined in base.html with content specific to the current page. This modular approach simplifies development and maintenance.

A black background with white text

Description automatically generated

## 2.2 Google maps link

In our patient\_page.html, we have implemented a JavaScript function that dynamically creates a Google Maps link based on the patient's address information. This functionality enhances user experience by allowing easy navigation to the patient's location. This is the JavaScript that is being used:

A screen shot of a computer code

Description automatically generated

* **Event Listener**: The code utilizes the DOMContentLoaded event listener, ensuring that the JavaScript runs only when the HTML document has fully loaded.
* **Retrieving Address Information**: The script fetches the street name, house number, and postcode from specific HTML elements on the page.
* **Generating Google Maps Link**: Using the retrieved address information, a Google Maps link is constructed. The link includes the street name, house number, and postcode in the query parameters.
* **Setting Link in HTML**: The dynamically created Google Maps link is then assigned to the href attribute of an anchor element with the id "googleMapsLink" in the patient\_page.html.

**HTML Integration:** In the patient\_page.html file, a paragraph element displays the patient's route information, and a link with the id "googleMapsLink" is provided to open the location in Google Maps.

****

**How It Works:**

* When the patient\_page.html loads, the JavaScript function executes.
* It extracts the street name, house number, and postcode from designated HTML elements.
* Using this information, it constructs a Google Maps link.
* The link is dynamically set to the "googleMapsLink" anchor element's **href** attribute.
* Users can click on "Open in Google Maps" to navigate to the patient's location on Google Maps.

# 3 Database

Whitin our Mixit project we use databases, to save data within the application. We use two types of databases, a database for storing the data for the application (Patiant datase) this database is within our design hosted by Mixit and stores the personal (caretakers) of the application as well as the working doctors. There is also a separate database with the medicine information connected with the patients. This is split due to within the real-life scenario the data would also come from two different places.

**Azure Database**

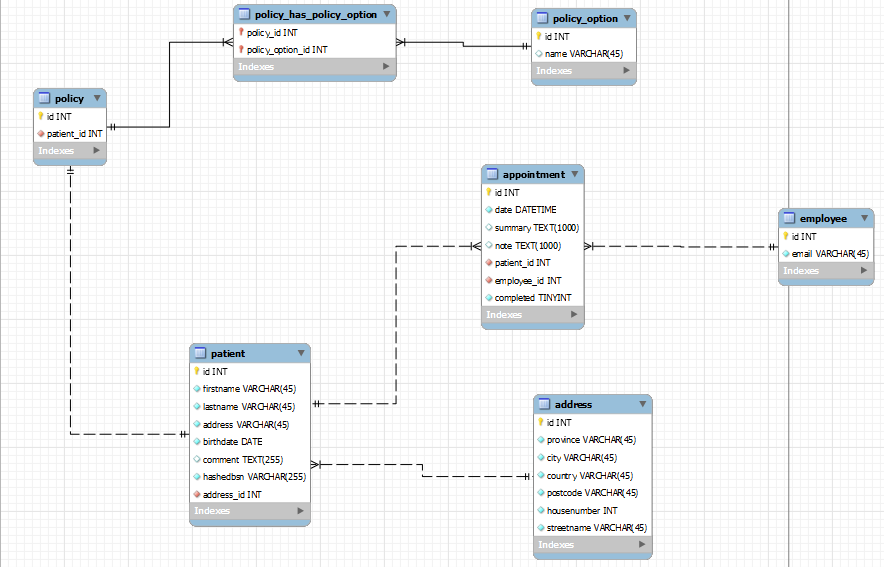
Within the terraform, we also added the deployment of a database in azure, this creates two database servers and two databases. We created this for our application to be true cloud native. When running the CI/CD it automatically creates two databases which can be used to run the application.

**Oege**

For saving costs, during our testing phase we used Oege to function as a database. Oege is an HVA owned server which enables students to host databases for HBO ICT students. To use Oege you first need to sign up to create a database environment. This can be done [here.](https://oege.ie.hva.nl/registratie/#services)

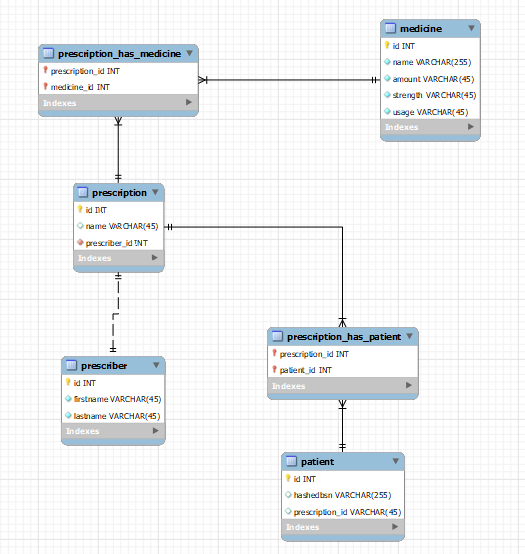
**Patient database**

The following picture shows how we created the patient database, this shows the relationship between the tables and which keys are primary and foreign keys.



**Pharmacy database**

The following picture shows how we created the Pharmacy database, this shows the relationship between the tables and which keys are primary and foreign keys.



**How to deploy**

The deployment consist of two different stages, because we are using an ORM, the tables within the databases are being created by the selective API, to connect the API to the database please follow the following steps:

Make sure your OEGE or azure database credentials are stored in:  
**/flaskApp.Folder/Mixit-patient.Api/cred.json**

As well as in: **/flaskApp.Folder/Mixit-apotheek.Api /cred.json**

Where you have to change the fields: database name, username and password accordingly.

Secondly change the Host in: **/flaskApp.Folder/Mixit-patient.Api/database.py**



If it's oege it should be: oege.ie.hva.nl if it's azure, it should be patient-server.database.windows.net or apotheek-server.database.windows.net. now when running the API’s the tables are created within the chosen database.

To add the mock data to the database you can import our mock data, we created a SQL script which after the tables are created you can import the data. these are located at:   
**flaskApp.Folder/Database\_Insert/ apotheek.sql a**nd **flaskApp.Folder/Database\_Insert/ patient.sql**

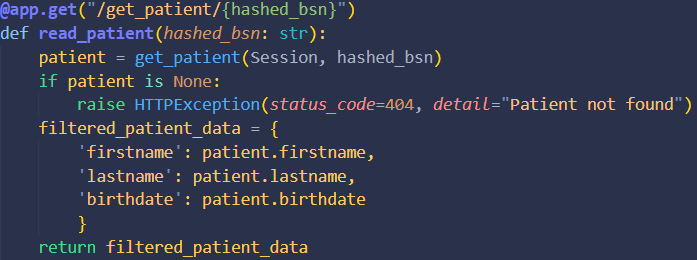
# 4 API

This part of the documentation provides information about the API. Currently, there are two APIs available. The first API connects to a database containing patient information such as names, addresses, contacts, etc. The second API acts as a placeholder for a pharmacy, where the information would typically be requested.

## 4.1 Overview

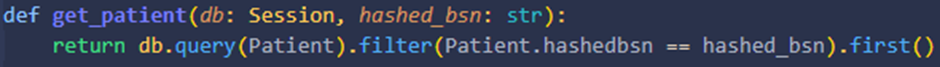
Both the APIs consist of four python files. The modules that are being used for the APIs are FastAPI and SQLalchemy.

* **Main.py:** In the file main.py, FastAPI is initialized, and the endpoints are defined. These endpoints provide information from the database stored in the cloud or write information to the database. The endpoints deliver information based on the provided data. Additionally, error handling is implemented in main.py. Within the endpoints themselves, functions in other Python files are called, responsible for retrieving or writing information to the database. Below is an example of one of the endpoints:



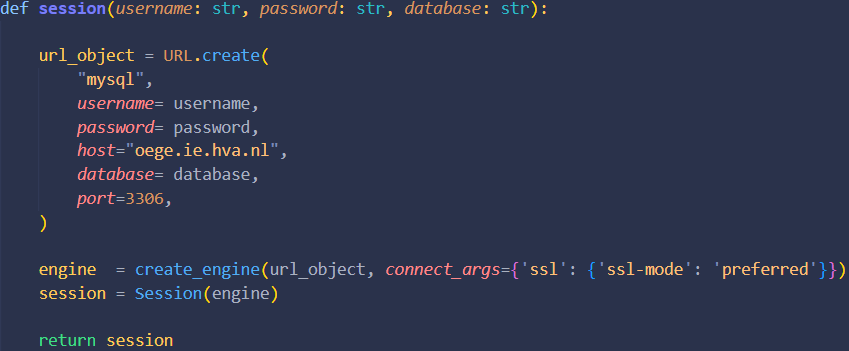
*Endpoint get\_patient*

* **Crud.py:** In the file crud.py, the SqlAlchemy ORM is applied. It includes seven functions that retrieve information from the database or write to it. These functions receive variables from main.py, allowing for the retrieval of specific information or writing to the database. The functions return the requested information or confirm the writing to the database. Below is an example of one of the functions:



*Function get\_patient*

* **Database.py:** In the file database.py, a connection to the database is established. This is done by creating a URL based on the database backend, username, password, host, database, and port. With this URL, an engine is created, which in turn initializes a session. This session is then passed to main.py. Additionally, credentials are retrieved from a JSON file using another function. Below is the connection:



*Function session*

* **Models.py:** In the file models.py, the models are included that are used for the database. This is essential for the ORM to function. The classes represent all tables in the database and can also be used to create the database. Below is an example of one of the classes:

A screen shot of a computer code

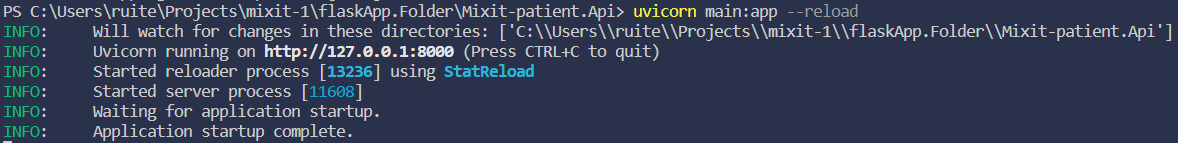
Description automatically generated

*Class patient*

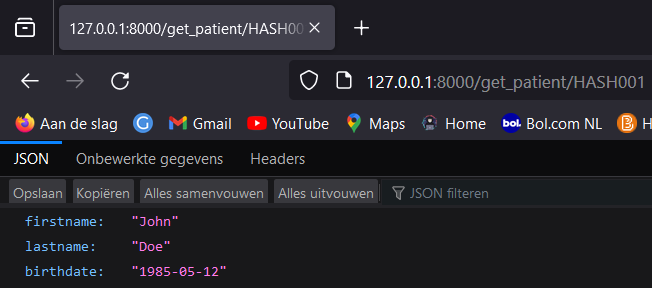
## 4.2 Setup

To run the API on your local machine, start by installing all the required modules listed in the **requirements.txt** file. Use pip to install the modules. Once installed, navigate to the appropriate directory where the Python files are located. Run the API locally using the command: `**uvicorn main:app --reload**`.

Expected output after running the command:



Expected output from API:



# 5 Terraform

## 5.1 Introduction to Terraform

**What is Terraform**

Terraform is an open-source infrastructure as code software tool created by HashiCorp.

It enables users to define and provision data center infrastructure using a declarative configuration language.

Terraform automates the deployment and management of various cloud and on-premises resources efficiently and consistently.

**Advantages of Terraform**

Multi-Cloud Support: Terraform supports multiple cloud providers, enabling seamless integration and management in multi-cloud and hybrid cloud strategies.

Scalability and Flexibility: Terraform's modular design allows easy replication and expansion of infrastructure, enhancing flexibility for evolving project needs.

Enhanced Collaboration and Version Control:Terraform integrates with version control systems, promoting immutable infrastructure for predictable and consistent deployment changes.

**Why Adopt Terraform**

Infrastructure Automation and Standardization: Using Terraform, complex medical system infrastructures can be automated and standardized. This simplifies the deployment and management of various systems and services required in healthcare processes, and helps maintain consistency.

Rapid Scalability and Flexible Management: The MixIT project includes various features such as patient summary information, schedule change notifications, and standard 'bed cleaning' protocols. Terraform allows for these services to be scaled quickly and flexibly, and to respond promptly to changes.

Enhanced Security and Compliance: In the healthcare sector, the security and privacy of patient data are extremely important. Terraform is an essential tool for managing infrastructure more securely and for meeting compliance requirements.

## 5.2 Environment Setup

Necessary Tools and Software

**⚙️ Terraform Installation:** It's essential to have the latest version of Terraform installed. Terraform is a crucial tool for infrastructure configuration and management. You can find download and installation instructions on the official website.

**⚙️ Azure CLI or PowerShell:** To manage Azure resources, you'll need Azure CLI or Azure PowerShell. These tools are used for interacting with Azure resources and are essential for authentication and configuration tasks.

## 5.3 Code Explanation

**Overview of the Entire File Structure**

├── main.tf  
├── outputs.tf  
├── providers.tf  
└── variables.tf

**Role of Each Files**

**🗂 main.tf**

This file is a code that uses Terraform to configure various resources in the Azure cloud environment. Key components include web applications, SQL servers, and key vaults, which are intended to support cloud-based services and databases.

**🗂 outputs.tf**

This file defines output values for Azure resources using Terraform. The primary outputs include the name of the resource group, the name of the key vault, and the unique ID of the key vault. This allows for sharing important information about the specified Azure resources with users or other Terraform modules.

**🗂 providers.tf**

This file sets up providers using Terraform to manage Azure cloud services ('azurerm') and random resources ('random'). The code specifies a particular Azure subscription ID, configuring Terraform's access to these services.

**🗂 variables.tf**

This file is a Terraform configuration that defines various variables related to Azure Key Vault. It includes user-defined variables such as GitHub authentication token, resource group location, key vault name, and others. These variables are used for configuring and managing security-related resources in the Azure cloud environment, including key vault creation, key management, and service identity settings.

**Detailed Explanation of Code by File**

**1. main.tf**

✅ This file is a Terraform configuration file designed for building and managing web application-related resources in the Azure cloud environment.

✅ It primarily defines and configures resources related to Azure Resource Groups, Service Plans, Linux-based Web Apps, SQL Servers and Databases, as well as Azure Key Vaults.

**Basic Infrastructure Setup**

This section explains the settings of Azure Resource Group (**azurerm\_resource\_group**), Service Plan (**azurerm\_service\_plan**), Linux-based Web App (**azurerm\_linux\_web\_app**), as well as SQL Server (**azurerm\_mssql\_server**) and Database (**azurerm\_mssql\_database**) resources.

**resource "azurerm\_resource\_group" "example" {**

**name = "mixit-740144f4"**

**location = "westeurope"**

**}**

**resource "azurerm\_service\_plan" "example" {**

**name = "app-serviceplan-mixit"**

**resource\_group\_name = azurerm\_resource\_group.example.name**

**location = azurerm\_resource\_group.example.location**

**os\_type = "Linux"**

**sku\_name = "S1"**

**}**

**resource "azurerm\_linux\_web\_app" "example" {**

**name = "website-740144f4"**

**resource\_group\_name = azurerm\_resource\_group.example.name**

**location = azurerm\_service\_plan.example.location**

**service\_plan\_id = azurerm\_service\_plan.example.id**

**https\_only = true**

**site\_config {**

**application\_stack {**

**python\_version = "3.11"**

**}**

**}**

**}**

**Security and Configuration Management**

In this section, the use of Azure Key Vault (**azurerm\_key\_vault**) for securely managing critical configuration and secret information of applications and databases (such as database names, usernames, passwords, etc.) is explained.

**resource "azurerm\_key\_vault" "vault" {**

**name = local.generated\_name**

**resource\_group\_name = azurerm\_resource\_group.example.name**

**location = azurerm\_resource\_group.example.location**

**tenant\_id = data.azurerm\_client\_config.current.tenant\_id**

**sku\_name = var.sku\_name**

**soft\_delete\_retention\_days = 7**

**access\_policy {**

**tenant\_id = data.azurerm\_client\_config.current.tenant\_id**

**object\_id = local.current\_user\_id**

**key\_permissions = var.key\_permissions**

**secret\_permissions = var.secret\_permissions**

**}**

**depends\_on = [**

**azurerm\_resource\_group.example**

**]**

**Resource Naming and Password Generation**

This part describes the use of **random\_pet** and **random\_string** resources for generating resource names, and **random\_password** resource for creating secure passwords.

**resource "random\_pet" "rg\_name" {**

**prefix = var.resource\_group\_name\_prefix**

**}**

**resource "random\_string" "azurerm\_key\_vault\_name" {**

**length = 13**

**lower = true**

**numeric = false**

**special = false**

**upper = false**

**}**

**resource "random\_password" "sql\_admin\_password" {**

**length = 20**

**special = true**

**min\_numeric = 1**

**min\_upper = 1**

**min\_lower = 1**

**min\_special = 1**

**}**

**2. outputs.tf**

✅ This code is used in Terraform to define output variables.

✅ Output variables are used to return information about resources created after Terraform provisions infrastructure.

✅ For example, after creating an Azure Key Vault, there might be situations where you need to know its name or ID. With output variables, you can easily access this information, eliminating the need to manually log into the Azure portal to find it.

Defines the name of the **Azure Resource Group** as an output variable.

**output "resource\_group\_name" {**  
 **value = azurerm\_resource\_group.rg.name**  
**}**

Defines the name of the **Azure Key Vault** as an output variable.

**output "azurerm\_key\_vault\_name" {**  
 **value = azurerm\_key\_vault.vault.name**  
**}**

Defines the **unique identifier (ID) of the Azure Key Vault** as an output variable.

**output "azurerm\_key\_vault\_id" {**  
 **value = azurerm\_key\_vault.vault.id**  
**}**

**3. providers.tf**

✅ This code provides the basic Terraform configuration for deploying and managing resources on Azure.

✅ Specifically, it focuses on the configuration and version management of the **azurerm** provider, which is essential for using Azure services.

In Terraform, **providers** play a crucial role. Providers are plugins that enable Terraform to communicate with various cloud providers or services (such as AWS, Azure, Google Cloud, Oracle Cloud, etc.) for managing infrastructure.

Below code is part of the Terraform configuration, used for declaring and configuring essential providers for managing infrastructure with Azure cloud services.

**terraform {**  
 **required\_providers {**  
 **azurerm = {**  
 **source = "hashicorp/azurerm"**  
 **version = "~>3.0"**  
 **}**  
 **random = {**  
 **source = "hashicorp/random"**  
 **version = "~>3.0"**  
 **}**  
 **}**  
**}**

**provider "azurerm" {**  
 **features {}**  
  
 **subscription\_id = "subscription id"**  
**}**

**4. variables.tf**

✅ This file contains variable settings for configuring and managing resources related to Azure Key Vault using Terraform.

✅ These variable settings define various parameter values for the creation and management of Azure resources.

**Resource Identification and Location**

**variable "resource\_group\_location" {**

**type = string**

**description = "Location for all resources."**

**default = "westeurope"**

**}**

**variable "resource\_group\_name\_prefix" {**

**type = string**

**description = "Prefix of the resource group name that's combined with a random ID so the name is unique in your Azure subscription."**

**default = "KeyVaultResourceGroup"**

**}**

**variable "vault\_name" {**

**type = string**

**description = "The name of the key vault to be created. The value will be randomly generated if blank."**

**default = "MixItKeyVaultName"**

**}**

**Resource Configuration and Specifications**

this includes variables such as u\_name, key\_type, key\_size, and key\_ops.

These variables are crucial for specifying the technical details and configurations of the resources, such as the type and size of the Key Vault key, and the SKU for the vault.

|  |
| --- |
| **variable "key\_name" {**  **type = string**  **description = "The name of the key to be created. The value will be randomly generated if blank."**  **default = "MixITKeyName"**  **}** |
| **variable "sku\_name" {**  **type = string**  **description = "The SKU of the vault to be created."**  **default = "standard"**  **validation {**  **condition = contains(["standard", "premium"], var.sku\_name)**  **error\_message = "The sku\_name must be one of the following: standard, premium."**  **}**  **}** |

**Permissions and Security**

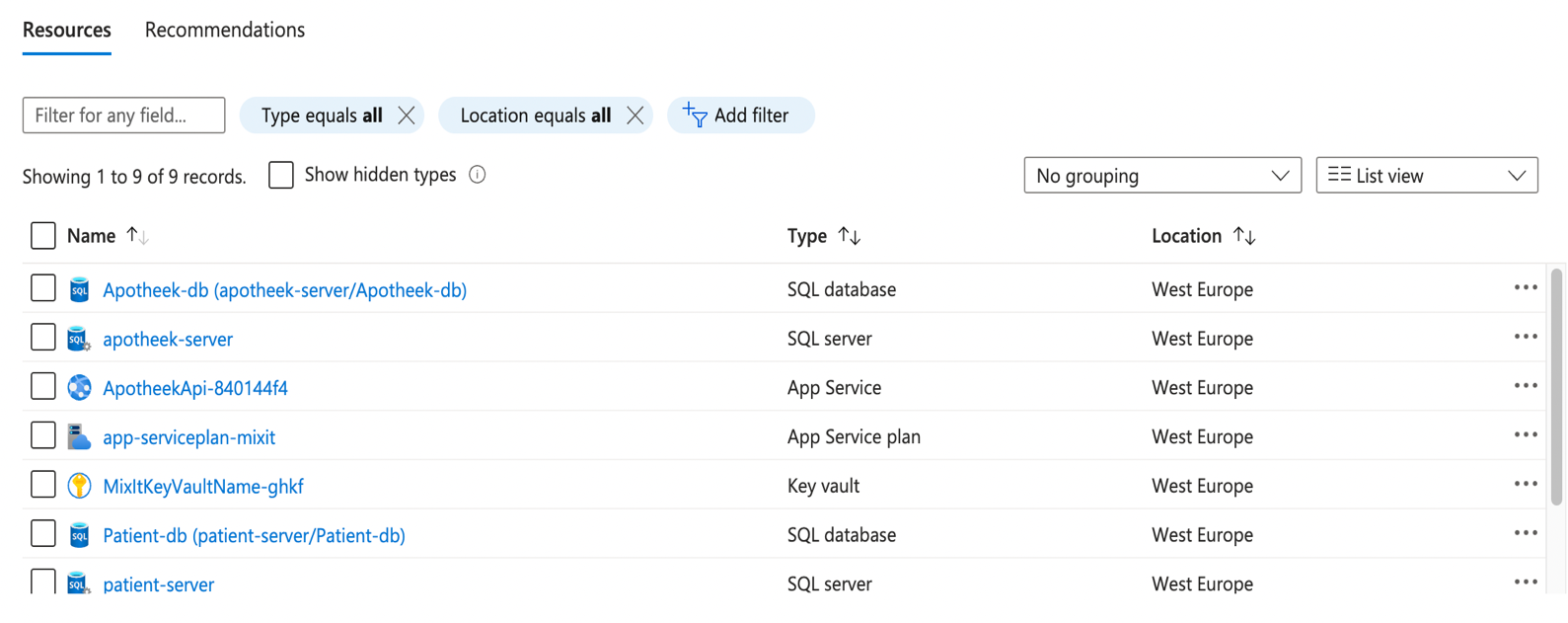
Variables like **key\_permissions**, **secret\_permissions**, and **msi\_id** are centered around security and access management.

They define the level of access and permissions for the Key Vault keys and secrets, and manage service identities, ensuring secure and controlled access to the resources.

|  |
| --- |
| **variable "key\_permissions" {**  **type = list(string)**  **description = "List of key permissions."**  **default = ["List", "Create", "Delete", "Get", "Purge", "Recover", "Update", "GetRotationPolicy", "SetRotationPolicy"]**  **}** |
| **variable "secret\_permissions" {**  **type = list(string)**  **description = "List of secret permissions."**  **default = ["Set", "Get", "List", "Delete"]**  **}**  **variable "msi\_id" {**  **type = string**  **description = "The Managed Service Identity ID. If this value isn't null (the default), 'data.azurerm\_client\_config.current.object\_id' will be set to this value."**  **default = null**  **}** |

5.4 Expected Outputs

* Apotheek-db (apotheek-server/Apotheek-db)
* apotheek-server
* ApotheekApi-840144f4
* app-serviceplan-mixit
* MixItKeyVaultName-ghkf
* Patient-db (patient-server/Patient-db)
* patient-server
* PatientApi-940144f4
* website-740144f4



# 6 Key vault

Welcome to our Terraform code documentation, where we describe the implementation of an Azure Key Vault. The Azure Key Vault acts as a secure and central storage point for sensitive information, such as passwords and keys, within our application infrastructure.

## 6.1 Overview

Our Key Vault, called "MixItKeyVaultName-xxxx", is created in the resource group "mixit-740144f4" in the region "West Europe." It is configured for disk encryption and template deployments. The SKU type is set to "standard" to meet our security needs. Every time you set up the CI/CD pipeline the code ensures that the correct name is set. Four random letters are also added in place of "xxxx" so that when everything is deleted and the pipeline is restarted, there are no overlapping names due to the soft delete functionality.

Afbeelding met tekst, schermopname, Lettertype

Automatisch gegenereerde beschrijving

The Terraform code provided in this section is dedicated to the creation and configuration of an Azure Key Vault within the Azure cloud. The Key Vault serves as a secure repository for sensitive information, such as database credentials, required for the deployment and operation of web applications.

## 6.2 Purpose of the Key Vault

The primary purpose of the Azure Key Vault in this project is to securely store and manage sensitive information, including database usernames and passwords. This centralized and secure storage ensures that critical credentials are not exposed directly in the Terraform code or configuration files, contributing to a more robust and secure deployment process.

1. **Secure storage of secrets**

* In the main.tf file, the Azure Key Vault is used to securely store sensitive information, such as database usernames and passwords. This ensures that critical credentials are not exposed directly in the Terraform code or configuration files:  
  Afbeelding met tekst, schermopname, Lettertype

  Automatisch gegenereerde beschrijving

In the picture above, the ApotheekApiDatabaseName secret is stored securely in the Azure Key Vault, and it can be later referenced in the application configuration.

1. **Enhanced security for cryptographic keys**

* The Azure Key Vault is utilized for managing cryptographic keys used for encryption and decryption. This enhances security by centralizing key management:  
  Afbeelding met tekst, schermopname, Lettertype

  Automatisch gegenereerde beschrijving  
  This section of code creates a cryptographic key within the Azure Key Vault, specifying rotation policies and other key-related attributes.

1. **Dynamic key and secret generation**

* The naming conventions for the Azure Key Vault and cryptographic keys are dynamically generated to ensure uniqueness and improve security:  
  Afbeelding met tekst, schermopname, Lettertype

  Automatisch gegenereerde beschrijving  
  In this example, the vault\_name variable and a random suffix contribute to the dynamically generated name for the Azure Key Vault.

1. **Flexible configuration**

* Variables defined in variables.tf allow for flexible and customizable configurations, accommodating various deployment scenarios:  
  Afbeelding met tekst, schermopname, Lettertype

  Automatisch gegenereerde beschrijving  
  Here, the sku\_name variable enables users to choose between standard and premium SKUs for the Azure Key Vault, providing flexibility in deployment configurations.

## 6.3 Test locally

When you modify the code, it is a lot of work to test everything via the pipeline, so we recommend that you test everything locally for functionality. This can be done by executing the line code below:

* Initialization of Terraform

terraform init -upgrade

* Create an implementation plan

terraform plan -out main.tfplan

* Execute implementation plan

terraform apply main.tfplan

* Delete everything

terraform plan -destroy -out main.destroy.tfplan

terraform apply main.destroy.tfplan

## 6.4 CI/CD integration

The overall execution of the code and the creation of the Azure Key Vault are integrated into a Continuous Integration/Continuous Deployment (CI/CD) pipeline. This automation ensures that the infrastructure is set up efficiently and securely, reducing manual intervention and enhancing the reliability of the deployment process.

By referencing the Azure Key Vault in these different sections of code, the project ensures a secure, dynamic, and flexible approach to managing sensitive information within the Azure cloud. This not only strengthens the security posture but also contributes to efficient and controlled management of secrets in the application infrastructure.

# 7 CI/CD

CI/CD stands for continues integration and deployment. This means that it ensures efficiency by providing a continuous flow for developers working on the product. It accomplishes this by continuously testing whether everyone’s new code is compatible with the entire source code.

7.1 Gitlab

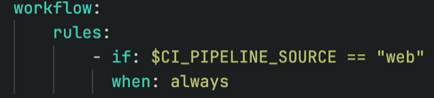
Our team utilizes Gitlab for our Git repository; this was required by the HVA. Naturally, we used Gitlab’s integrated CI/CD functionality to deploy and test our code. Gitlab’s CI/CD repositories are managed by a single file named: **{.gitlab-ci.yml}**

As the file extension suggests, this is a YAML file (Yet Another Markup Language). Gitlab CI/CD is a fairly flexible system that allows you to quickly stitch together more *“Stages”* and create more stages later; you can even specify the conditions under which those extra jobs and stages run. Gitlab CI/CD uses *“Runners”,* which are agents that listen to Gitlab on virtual machines with Docker pre-installed. Because of the Docker capabilities, they can run anything that is conceivable in containers.

Our team makes extensive use of that functionality to work with Terraform and Azure-Cli, both of which aid with the deployment of our cloud solution. Another feature we used extensively is the ability to hide secret parameters in the **{.gitlab-ci.yml}** file. Which are filled automatically in by authorized maintainers. This enhances the security of our CI/CD pipeline, which would otherwise contain confidential information.

7.2 Pipeline

Our pipeline is configured so that it will only activate if done manually. While this is not the typical method of setting up a pipeline, we had our reasons to do it this way. **CI\_PIPELINE\_SOURCE == web** is our means of checking for a manual pipeline trigger.

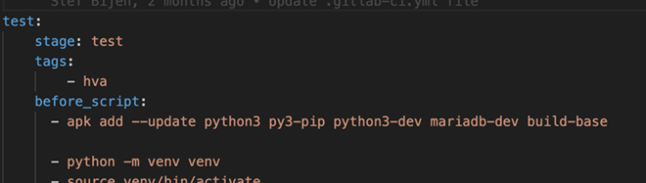


We have three stages:

1. Test
2. Zip
3. TerraAndAzCli

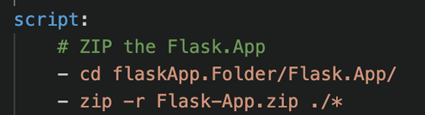
**7.2.1** **Stage – Test**

This stage spins up a running which will then receive a copy of our source code. After that, it will install all of the required dependencies to run our application and launch Pytest, a Python testing framework. Pytest runs test code to check our application’s primary features for errors and irregularities. If it identifies even one anomaly, it will terminate the entire pipeline. The test stage used the image defined in the template in our CI/CD file before moving on to the next stages, as the Terraform image is sufficient, and no other images are defined in the stage.



**7.2.2** **Stage – Zip**

This stage, as the name implies, zips our individual applications so that they may be transferred down the pipeline and deployed to the Azure cloud. This is accomplished by identifying them as artifacts that other stages can pick up. The zip stage is the simplest of them all, but it is still important because Azure needs the webapp to be delivered in zip format.

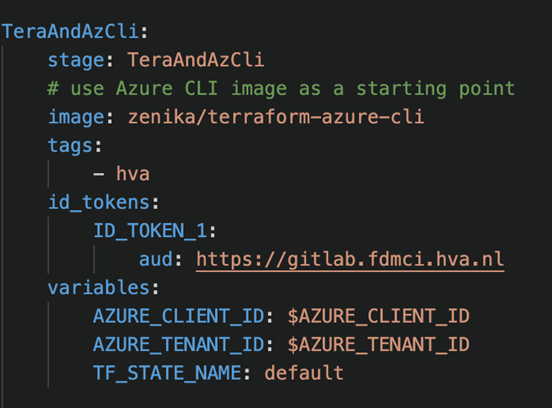


**7.2.3** **Stage – TerraAndAzCli**

The final stage entails preparing our cloud infrastructure and deploying it. For this, an open-source Docker container is employed, which includes Terraform and Azure-Cli, making it ideal for our needs. We also receive some secrets, which we use to obtain adequate authorization for our Azure cloud deployment. These variables are only entered in certain branches; hence it is important to run the pipeline from the correct branch. The working branches are:

1. Main
2. Terraform Implementation

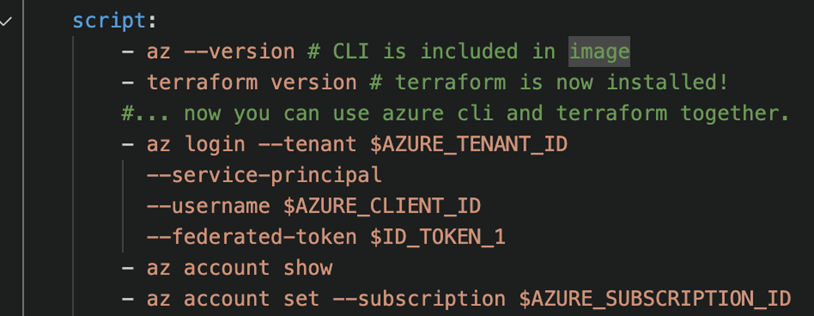
In these branches, you can execute our pipeline and obtain adequate permissions to work on Azure.



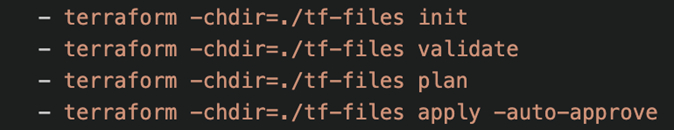
Next, we need to ensure that Azure-Cli and Terraform are running properly by requesting the version installed from both. Afterward, we attempt to login to Azure cloud by providing the secrets we obtained before; we require the following to have sufficient access:

* Tenant ID
* Client ID
* Federated token

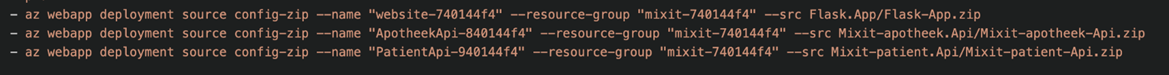
After logging in, we set our subscription to the value of a global CI/CD variable.



We still need to prepare our Azure cloud infrastructure using Terraform, which includes initializing, validating, planning, and applying our configuration. These should all proceed smoothly, but we need to apply the argument -auto- approve to run the pipeline without manual user input. -auto- approve prevents Terraform from asking if we are sure we want to perform certain actions.



The final piece of the puzzle is the deployment of our software to our designates Azure webapps we created using Terraform. The code below makes use of a webapp deployment which is provided with the name, resource group, and src of our software. This source of our software is where we left it when we finished the zip stage.

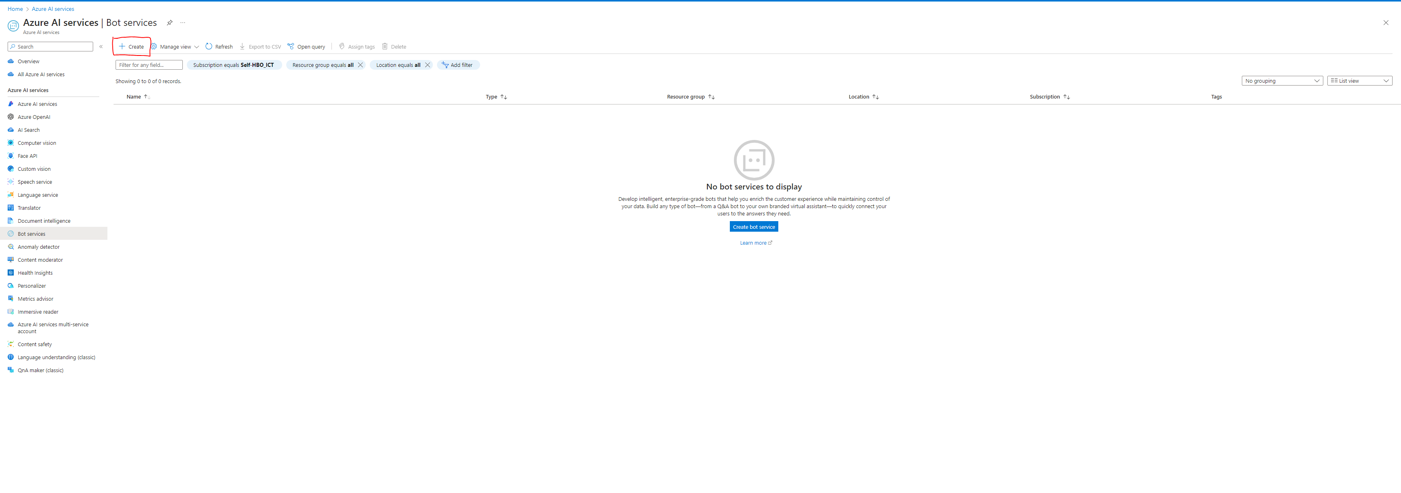


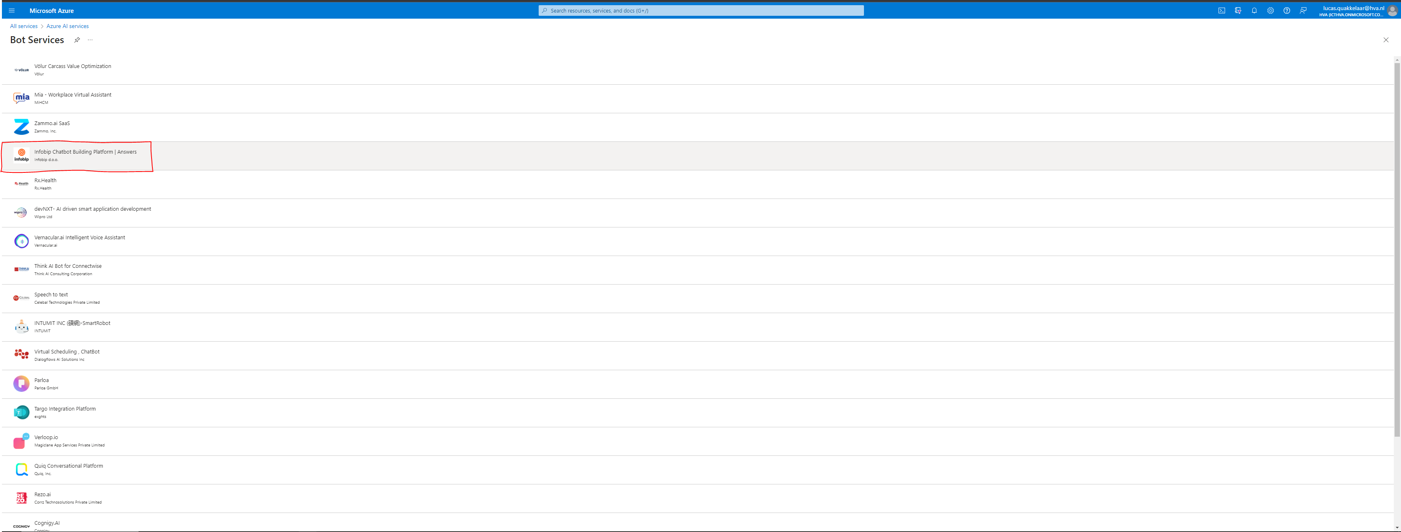
# 8 Add function

This part of the documentation will include a plan on how to add a function to the current application. We will be taking a chat bot function as an example, and we will be looking at multiple ways of implementing this functionality.

## 8.1 Azure AI

The first way of implementing a chatbot is through the Azure AI services, here you can choose a platform that you want to build your bot from, take for example the ‘Infobip Chatbot’. This way of making a chatbot is especially useful when you have your whole application built on azure. This way you can use Function apps to ask questions and get the answers displayed in your application.





Another benefit of using Azure AI is that you can use service buses to make sure every call to the API is made and nothing is lost because of bad calls or network issues. Using a service bus essentially means that you first put a request in a waiting line, and it executes it whenever it is possible.

Infobip is a chatbot system that makes you set up your own customizable chatbot, this means that you first must register on their website and make your chatbot, here you can easily add possible questions and the answers to those questions to them.

## 8.2 Infobip Base URL

A different way of implementing a chatbot to your application is through an api call to the api that’s built into Infobip, this is called Base URL. This way you first must authenticate yourself, after the authentication you can make api calls from the questions and get the answers as a response.

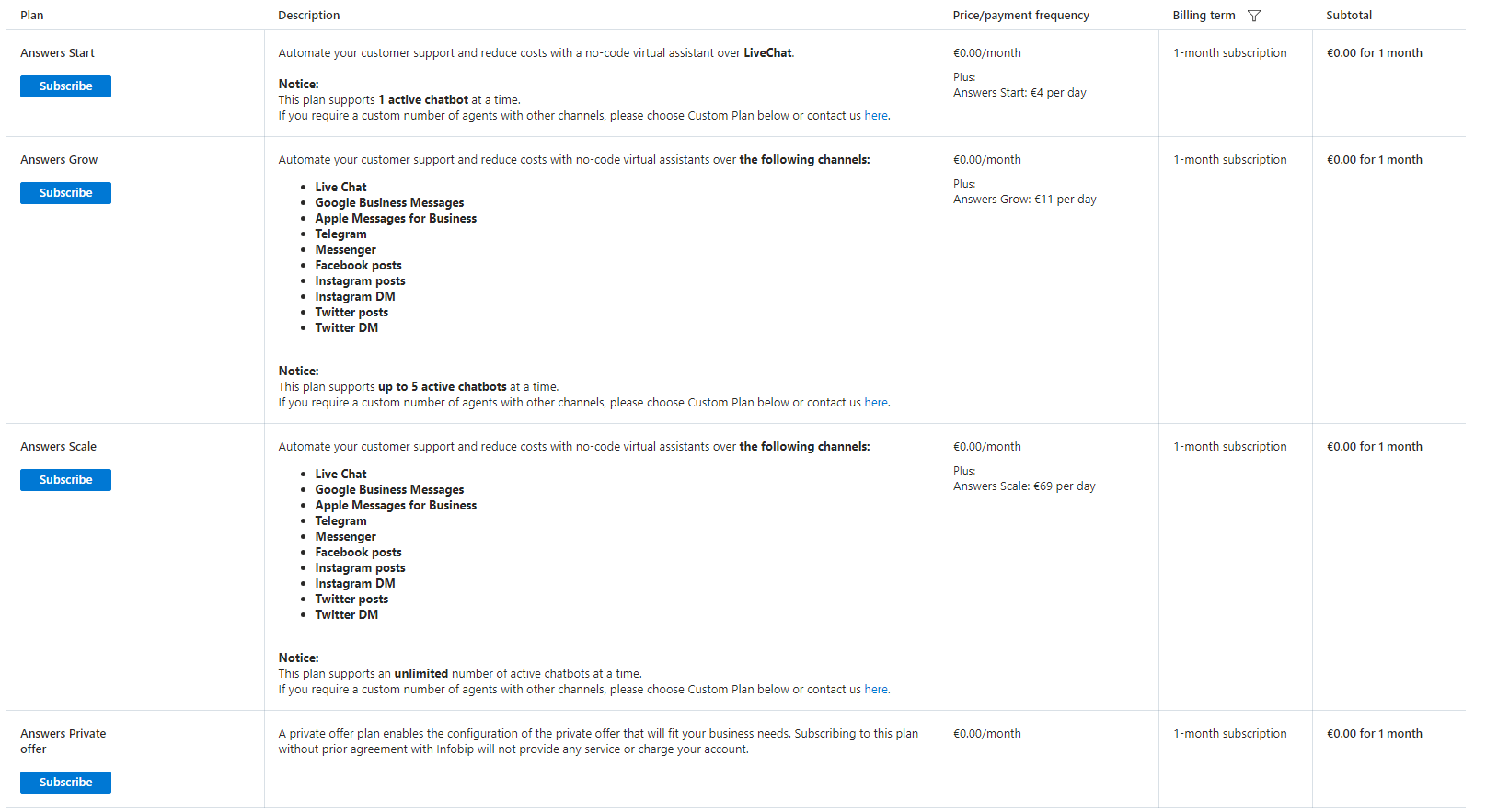
For more information about the authentication process and api calls to infobip see: <https://www.infobip.com/docs/essentials/api-authentication>

Using the api call it isn’t integrated with the azure resources and it’s harder to use somethig like a function app or service bus with it. The plus side to using the api call is that it’s also possible to use the application off the cloud. So when your organisation is still contemplating on using the cloud or only using it partly it could be beneficial to use the api call instead of the Azure AI way.

## 8.3 Differences

The main difference between the Azure AI and the Base URL is the fact that the chatbot is on the azure cloud or not. When your whole application is running on azure resources it’s easier to connect everything because of the easy integration of azure. The downside to using everything on azure is that you become dependent on the azure cloud and that could become a problem when you’d like to use your own servers or switch to a different cloud, for example AWS (Amazon web services).

Another difference is the pricing, when you’re using the Azure AI version of Infobip it can cost from €4 a day for the cheapest program where you can only have one active chatbot at a time to €69 a day for unlimited chatbots and integration with multiple following channels.



# 8 Downloads

Git bash (Windows Only!): <https://git-scm.com/download/win>