Devops Assignment 2 - Yousef Amirghofran (Briefbot)

In the first part of the assignment, I built a great project called Briefbot that allowed users to submit links via the web page or a browser extension and an AI-enabled workflow would then extract information from the link, summarize it, and provide a podcast about the item using external API services.

In this assignment, I improved the code quality and pursued DevOps practices to deploy the project on Azure.

The report will focus on explaining each part of this process in better detail with examples.

Refactoring Code Smells

After learning about SOLID principles and code smells, I reviewed the code in my project to see whether I could identify any code smells and improve the quality of the code. And I found a lot of places where I could improve the code:

Bloated Handler (SRP Violation)

```
type Handler struct {
    userService    services.UserService
    itemService    services.ItemService
    digestService  services.DigestService
    podcastService services.PodcastService
    sseManager     *services.SSEManager
}
```

I had created a handler struct in my project that would receive each service (userService, itemService, digestService, podcastService, and sseManager) as an input. This made the handler depend on 5 different services and handle the routing for multiple domains (users, items, podcasts, and digests) which violated the Single Responsibility Principle as the struct had too many reasons to change. Any change in any of those services could potentially propagate the change to the handler struct.

To fix this problem, I refactored the handler struct and broke it down into more focused structs where each handler dealt with its own domain. This made the code much easier to maintain and scale later on.

```go
type UserHandler struct {
    userService services.UserService
}
type ItemHandler struct {
    itemService services.ItemService
    sseManager  *services.SSEManager
}
type PodcastHandler struct {
    podcastService services.PodcastService
    sseManager     *services.SSEManager
}
type DigestHandler struct {
    digestService services.DigestService
}
```

Hardcoded Worker Service (OCP Violation)

I had written a `processBatch` function in the worker service for the workers to process the jobs of creating a podcast for the items. However, I had hardcoded the processing of the podcasts from the podcasts service, tightly coupling the service that created the jobs with the worker service. This made it so that any time I wanted to add a new type of job or service to be processed, I had to edit the `processBatch` function which broke the Open-Closed principle where the function should be open to extension but closed for modification

```go
func (s *workerService) processBatch() error {
    // Process items first
    if err := s.processItemBatch(); err != nil {
        return fmt.Errorf("failed to process item batch: %w", err)
    }

    // Process podcasts if enabled
    if s.enablePodcasts && s.podcastService != nil {
        if err := s.processPodcastBatch(); err != nil {
            return fmt.Errorf("failed to process podcast batch: %w", err)
        }
    }
    return nil
}
```

To fix this, I decided to use the strategy pattern in the `processBatch` function and used Dependency Injection to pass a list of the processors to the worker service, decoupling the services that create the jobs from the worker service. Now, I don't have to edit the function when I add a processor, it automatically handles it when I pass in the processor when initiating the service.

```go
type JobProcessor interface {
    Process(ctx context.Context, batchSize int32) error
}
type workerService struct {
    jobQueueService JobQueueService
    processors      []JobProcessor
    // ... other fields
}
func (s *workerService) processBatch() error {
    for _, processor := range s.processors {
        if err := processor.Process(s.ctx, s.batchSize); err != nil {
            return err
        }
    }
    return nil
}
```

These two examples serve to represent the larger number of refactorings of similar nature that I did to improve the code quality. Overall, these fixes significantly improved the quality of my code and made it more maintainable and extensible for the future.

Testing

I followed the testing conventions in Go in this project, writing tests in `_test.go` files and using the rich go testing library.

I started with writing unit tests that verified functionality of small isolated functions. These tests shape the majority of the tests because they are easier and faster to write and run (which is important later when we run them in the CI/CD pipeline with every pull request).

At this stage, I also wrote mock tests. For that, I created a centralized test utilities package where I defined the fixtures, mock interfaces, and helper test functions for writing mock tests. For example, the `MockQuerier` interface mocked real database operations and the `TestDataBuilder` function used the Factory method to create the test data for certain mock tests. Since my project uses a lot of complex external APIs, I was limited in the number of services I could mock. That's why my code coverage wasn't as high as others at ~70%.

However, I also wrote integration tests to test the actual behavior of some of the services I was using such as Cloudflare R2 for object storage and FFMPEG for audio stitching. There were still some limitations to test the Fal service since it's a very complicated API.

Over the different parts of the project (metrics, handlers, middleware, and services), I reached a test coverage of ~85% with all tests passing. Test reports are included in the repo at `backend/test-results.txt` and `coverage.html`.

Containerization

To prepare for the deployment of this project, I containerized the project using Docker. First, I created multi-stage Dockerfile builds for the backend (Go API) and frontend (React Tanstack Start) components of the project to enable creating docker images for each of them. Then, I created a `docker-compose.yml` file at the root of the project to orchestrate all the interconnected services including the backend, frontend, PostgreSQL database, and the monitoring infrastructure (Prometheus and Grafana images and configurations).

This not only allowed me to build and run the project much faster with a single `docker-compose up` script, but it enabled me to configure the deployment pipeline as well and automate the deployment as well.

Monitoring

After a bit of research, I realized that the most common stack for monitoring is using Prometheus and Grafana where Prometheus acts as a time-series database that scrapes the project for metrics such as latency, error rate, and health status and Grafana is an easy way to declaratively create the graphical dashboards using YAML files to display the metrics in a visual dashboard.

To do this, I first added a simple `/health` endpoint to my backend API that returned success if the backend was up. Then, I created a `metrics.go` package to instrument my backend functions with metrics that would be exposed through the new `/metrics` endpoint. Additionally, I used the Prometheus Go client library to create a middleware to automatically log request metrics without changing the business logic. This made it so that Prometheus could scrape the metrics from my app. Then, I configured the Prometheus settings in `monitoring/prometheus.yml` to scrape metrics in 15-second intervals. Here are the main metrics I collected with Prometheus:

- HTTP: request counts, latency, request/response sizes, active requests

- Job Queue: num. enqueued tasks, num. processing tasks, num. completed tasks, num. failed tasks
- Workers: num. active workers, num. jobs processed per worker, worker error rates, batch durations
- Podcasts: counts, durations, audio requests, failure rate
- SSE: num. active connections, num. events sent by type

Under `monitoring/grafana/` , I set up the Grafana dashboard structure and determined a number of panels with PromQL queries to aggregate and visualize the data from Prometheus.

Finally, in the docker-compose file, I set up both the Prometheus and Grafana services with persistent volumes (`prometheus_data` and `grafana_data`) and configured the ports across the `briefbot-network` bridge network.

CI/CD Pipeline

I used Github Actions as my CI/CD pipeline solution because my code already resides in Github and the most natural way to perform Continuous Integration, and later Deployment, is in the same platform.

There are multiple stages in the CI/CD pipeline. First, it runs lint and formatting checks on the code. This is standard practice to make sure all code follows the projects' conventions and allows developers to focus on the logic of the code, rather than noisy formatting inconsistencies. After that, it builds them to verify they compile.

Next, since I use `SQLc` to auto-generate Go functions from SQL files, I use the `verify` function from `SQLc` to verify my queries are in sync with my database schema. This catches potential database migration errors before they can cause any problems so they can be fixed in the Pull Request itself.

Then, the pipeline runs the test suite. It both verifies that all tests pass and the overall test coverage is above 70%. If these conditions aren't true, it fails and the developer can see how to fix the issues. This also helps prevent pushing errors to production.

The previous stages run on any pull request. On any push to the `main` branch, after all the previous stages succeed, the pipeline builds the docker containers, as discussed previously, and publishes them to Docker Hub (the authentication tokens for Docker Hub are configured in Github Secrets). This allows our deployment provider to pull these containers every time they are updated and deploy them.

Deployment

Finally, everything is ready to deploy the project. I thought the best Azure service I could use was the App Service since it works with containers and natively integrates with monitoring products such as Azure Insights.

I created two Azure App Services, one for the frontend component of my app, and one for the backend. In both cases, I configured the container registry as Docker Hub and provided the name of the containers to the corresponding services (backend and frontend) with the `latest` tag. Every time my CI/CD pipeline publishes a new container, the App Services pull them and deploy them.

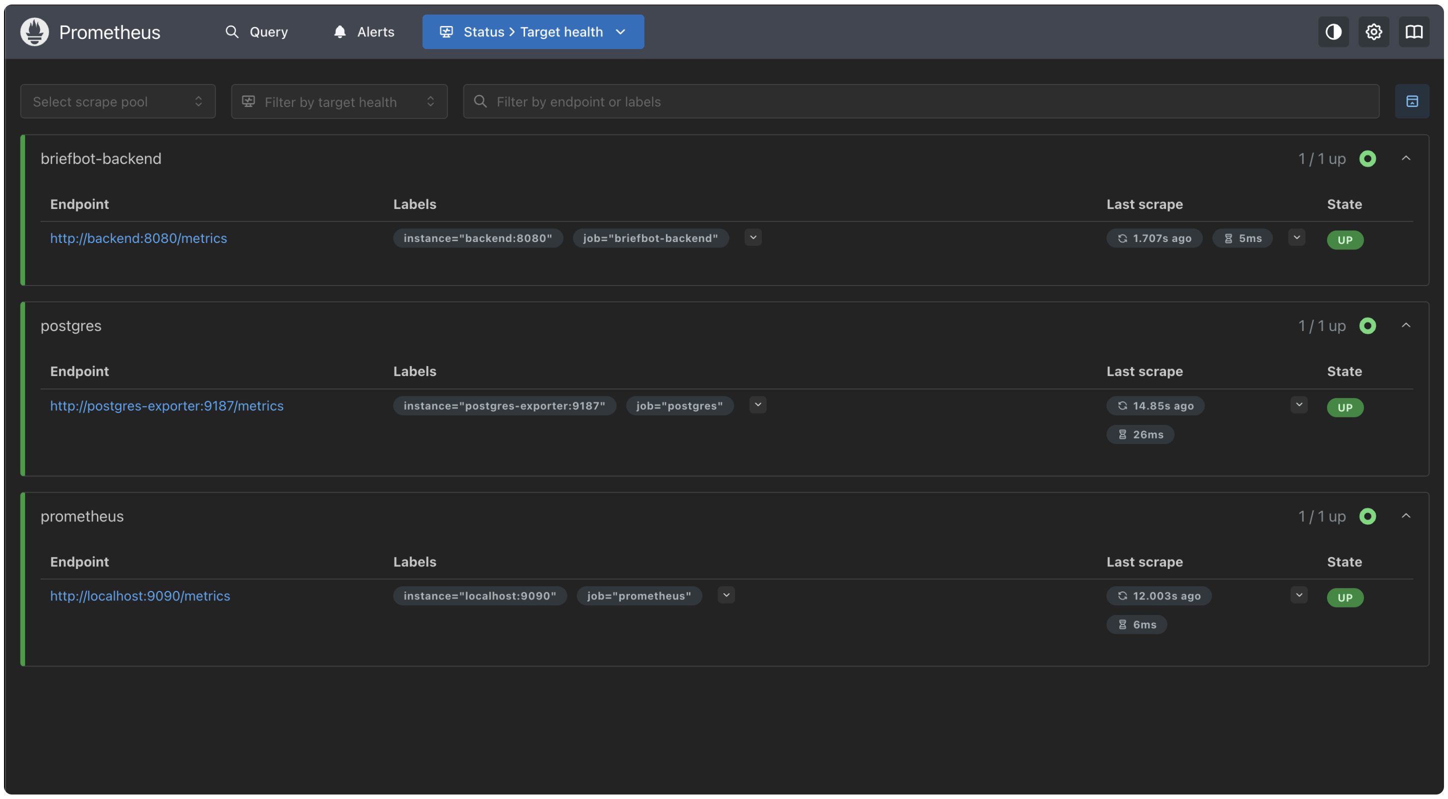
Finally, I created a Flexible PostgreSQL instance in Azure and use the URL for it as the `DATABASE_URL` environment variable in my backend service to connect the backend to the database.

Conclusion

In this assignment, I reviewed the project to refactor certain code smells to improve the quality of the code and make it compliant with SOLID principles, added unit and integration tests to ensure the robustness and reliability of the project, containerized the different components for easier deployment, added monitoring using Prometheus and Grafana, and created a CI/CD pipeline to automate deployment to the Azure cloud.

Appendix

Prometheus Dashboard



briefbot-backend 1 / 1 up

Endpoint	Labels	Last scrape	State
http://backend:8080/metrics	instance="backend:8080" job="briefbot-backend"	1.707s ago 5ms	UP

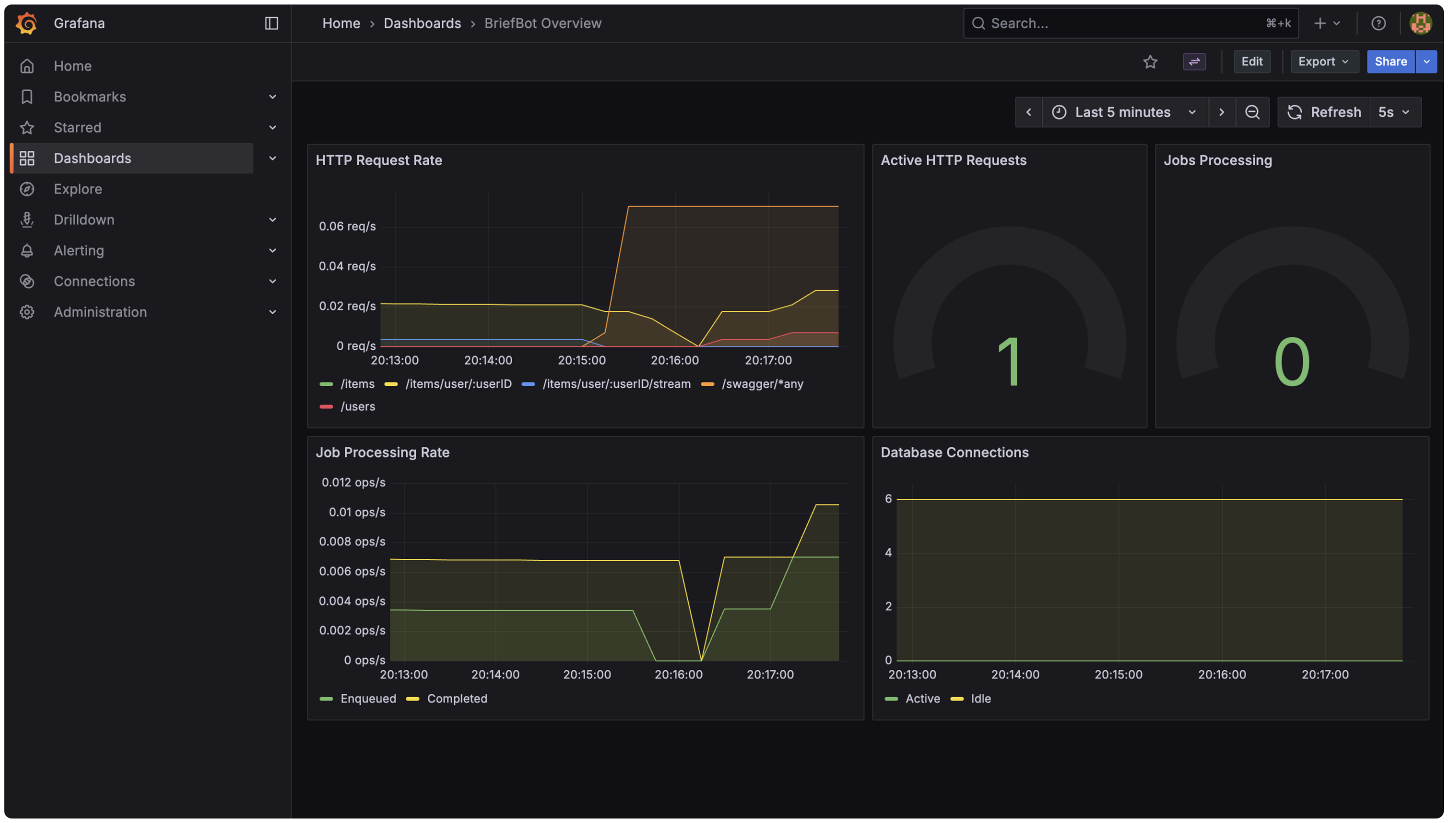
postgres 1 / 1 up

Endpoint	Labels	Last scrape	State
http://postgres-exporter:9187/metrics	instance="postgres-exporter:9187" job="postgres"	14.85s ago 26ms	UP

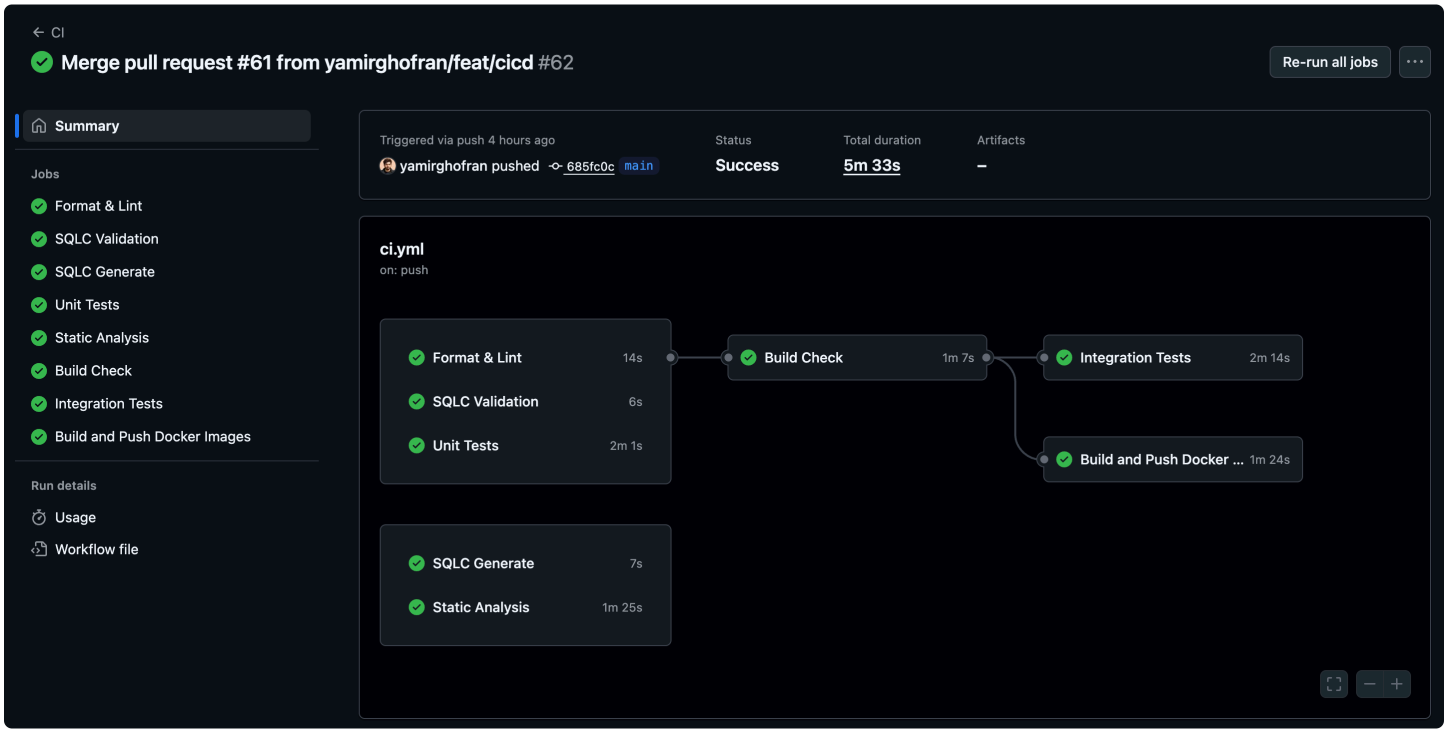
prometheus 1 / 1 up

Endpoint	Labels	Last scrape	State
http://localhost:9090/metrics	instance="localhost:9090" job="prometheus"	12.003s ago 6ms	UP

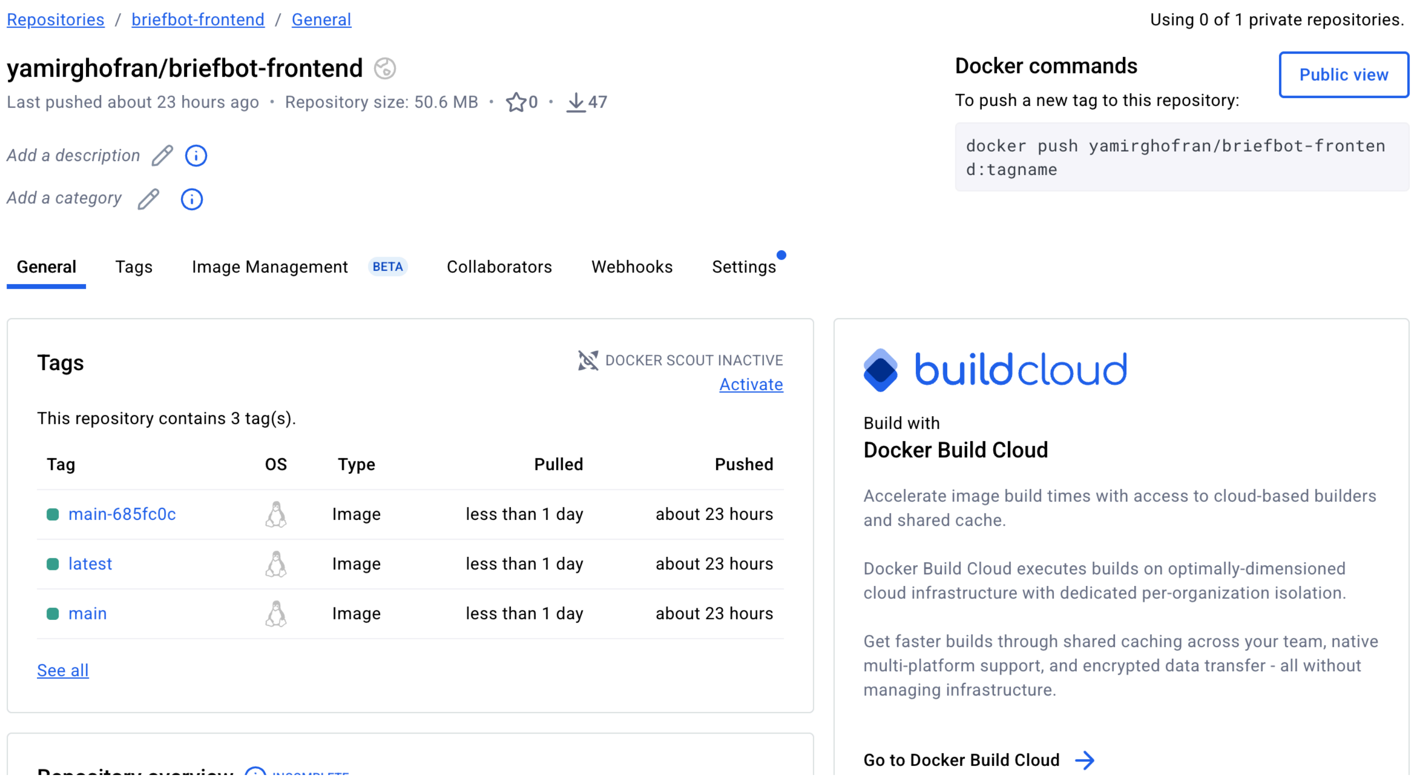
Grafana Dashboard



Github Actions CI/CD



Docker Hub Containers



Repositories / briefbot-frontend / General

Using 0 of 1 private repositories.

yamirghofran/briefbot-frontend

Last pushed about 23 hours ago • Repository size: 50.6 MB • ☆0 • ⤓47

Add a description

Add a category

Docker commands

To push a new tag to this repository:

```
docker push yamirghofran/briefbot-frontend:tagname
```

General · Tags · Image Management BETA · Collaborators · Webhooks · Settings

Tags

DOCKER SCOUT INACTIVE Activate

This repository contains 3 tag(s).

Tag	OS	Type	Pulled	Pushed
main-685fc0c		Image	less than 1 day	about 23 hours
latest		Image	less than 1 day	about 23 hours
main		Image	less than 1 day	about 23 hours

See all

buildcloud

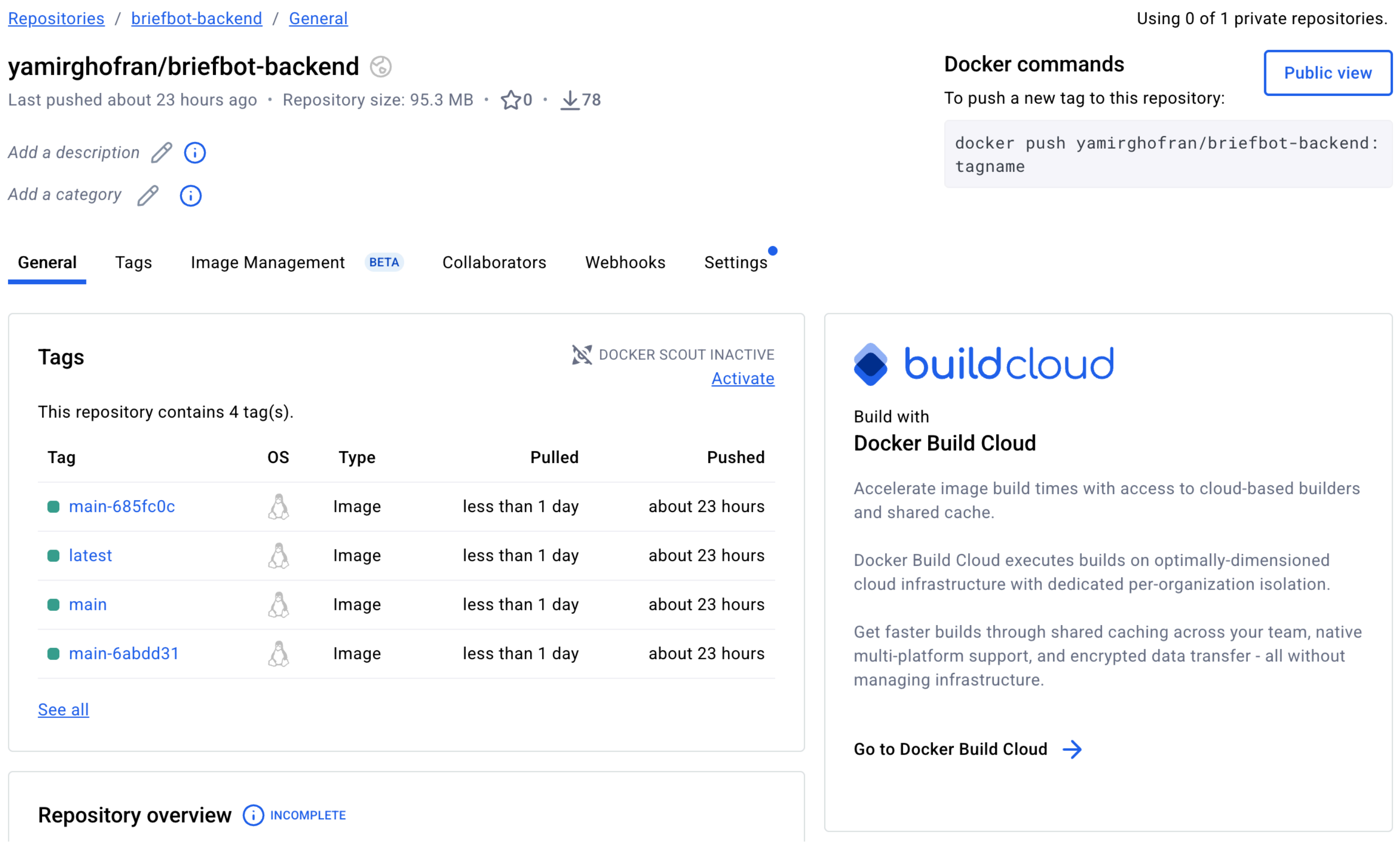
Build with
Docker Build Cloud

Accelerate image build times with access to cloud-based builders and shared cache.

Docker Build Cloud executes builds on optimally-dimensioned cloud infrastructure with dedicated per-organization isolation.

Get faster builds through shared caching across your team, native multi-platform support, and encrypted data transfer - all without managing infrastructure.

Go to Docker Build Cloud →



Repositories / briefbot-backend / General

Using 0 of 1 private repositories.

yamirghofran/briefbot-backend

Last pushed about 23 hours ago • Repository size: 95.3 MB • ☆0 • ⤓78

Add a description

Add a category

Docker commands

To push a new tag to this repository:

```
docker push yamirghofran/briefbot-backend:tagname
```

General · Tags · Image Management BETA · Collaborators · Webhooks · Settings

Tags

DOCKER SCOUT INACTIVE Activate

This repository contains 4 tag(s).

Tag	OS	Type	Pulled	Pushed
main-685fc0c		Image	less than 1 day	about 23 hours
latest		Image	less than 1 day	about 23 hours
main		Image	less than 1 day	about 23 hours
main-6abdd31		Image	less than 1 day	about 23 hours

See all

buildcloud

Build with
Docker Build Cloud

Accelerate image build times with access to cloud-based builders and shared cache.

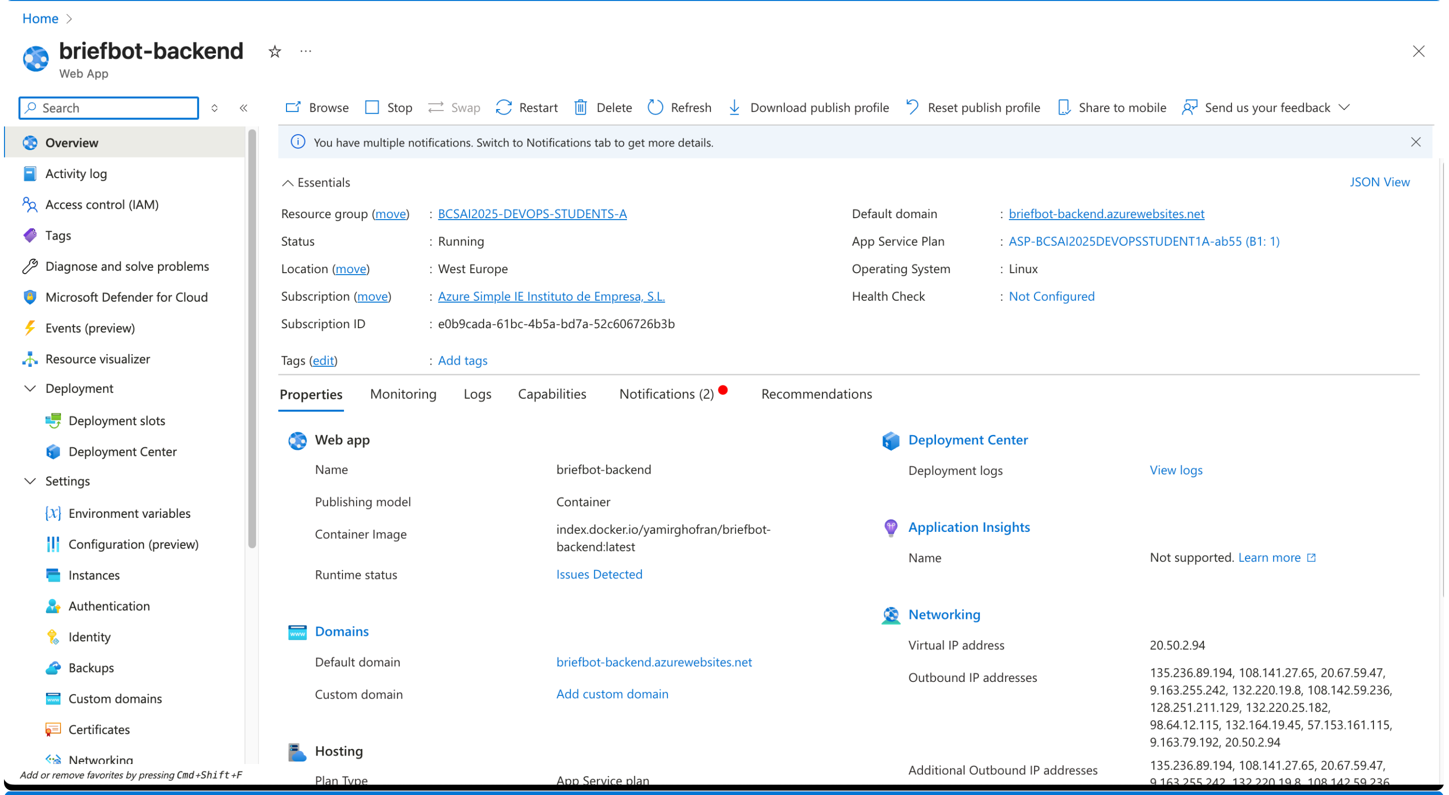
Docker Build Cloud executes builds on optimally-dimensioned cloud infrastructure with dedicated per-organization isolation.

Get faster builds through shared caching across your team, native multi-platform support, and encrypted data transfer - all without managing infrastructure.

Go to Docker Build Cloud →

Repository overview INCOMPLETE

Azure App Services



briefbot-backend
Web App

You have multiple notifications. Switch to Notifications tab to get more details.

Essentials

Resource group (move)	: BCSAI2025-DEVOPS-STUDENTS-A	Default domain	: briefbot-backend.azurewebsites.net
Status	: Running	App Service Plan	: ASP-BCSAI2025DEVOPSSTUDENT1A-ab55 (B1: 1)
Location (move)	: West Europe	Operating System	: Linux
Subscription (move)	: Azure Simple IE Instituto de Empresa, S.L.	Health Check	: Not Configured
Subscription ID	: e0b9cada-61bc-4b5a-bd7a-52c606726b3b		
Tags (edit)	: Add tags		

Web app

Name	briefbot-backend
Publishing model	Container
Container Image	index.docker.io/yamirghofran/briefbot-backend:latest
Runtime status	Issues Detected

Domains

Default domain	briefbot-backend.azurewebsites.net
Custom domain	Add custom domain

Hosting

Plan Type	App Service plan

Deployment Center

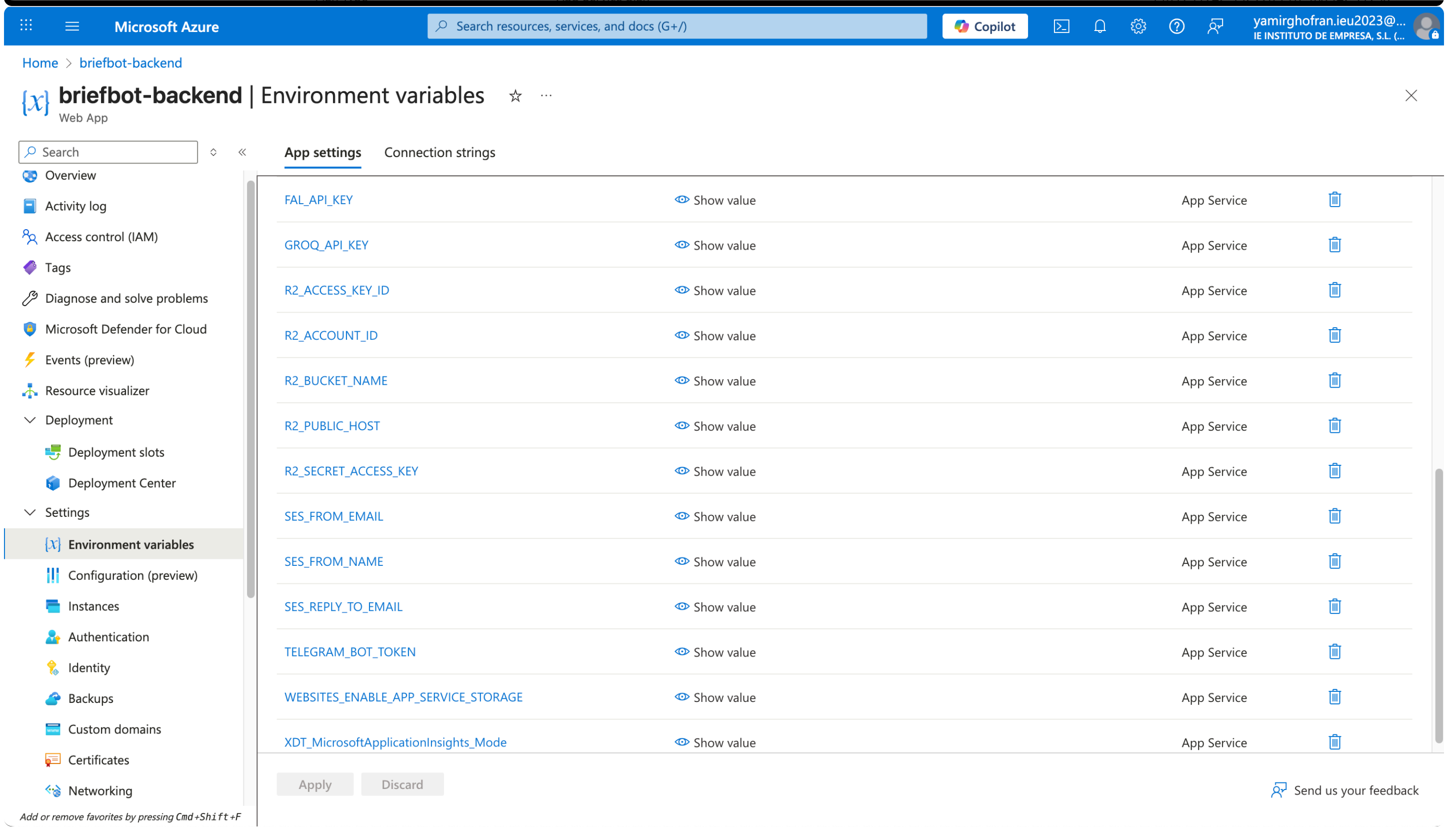
Deployment logs	View logs

Application Insights

Name	Not supported. Learn more

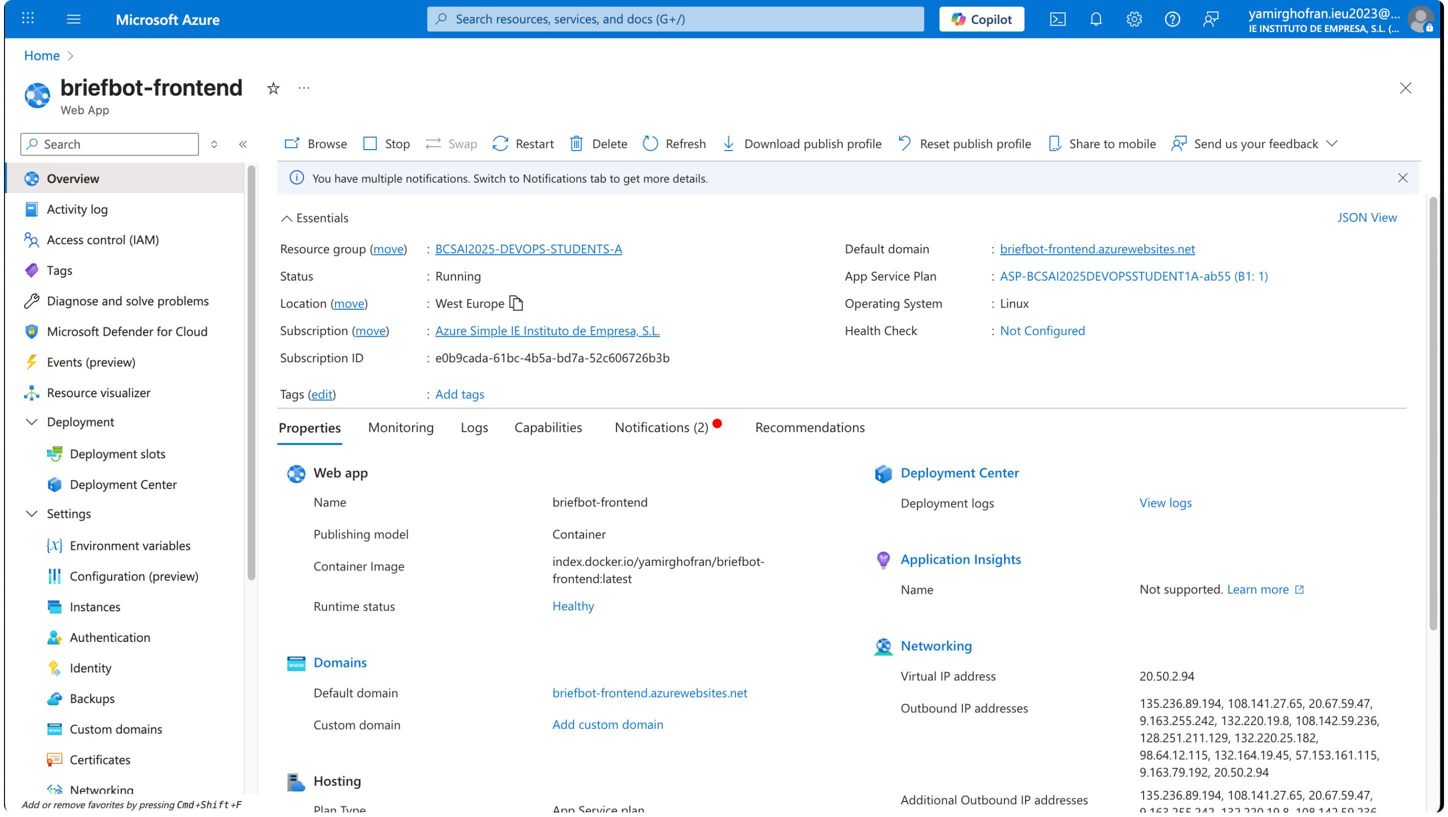
Networking

Virtual IP address	20.50.2.94
Outbound IP addresses	135.236.89.194, 108.141.27.65, 20.67.59.47, 9.163.255.242, 132.220.19.8, 108.142.59.236, 128.251.211.129, 132.220.25.182, 98.64.12.115, 132.164.19.45, 57.153.161.115, 9.163.79.192, 20.50.2.94
Additional Outbound IP addresses	135.236.89.194, 108.141.27.65, 20.67.59.47, 9.163.255.242, 132.220.19.8, 108.142.59.236



briefbot-backend | Environment variables
Web App

FAL_API_KEY	Show value	App Service
GROQ_API_KEY	Show value	App Service
R2_ACCESS_KEY_ID	Show value	App Service
R2_ACCOUNT_ID	Show value	App Service
R2_BUCKET_NAME	Show value	App Service
R2_PUBLIC_HOST	Show value	App Service
R2_SECRET_ACCESS_KEY	Show value	App Service
SES_FROM_EMAIL	Show value	App Service
SES_FROM_NAME	Show value	App Service
SES_REPLY_TO_EMAIL	Show value	App Service
TELEGRAM_BOT_TOKEN	Show value	App Service
WEBSITES_ENABLE_APP_SERVICE_STORAGE	Show value	App Service
XDT_MicrosoftApplicationInsights_Mode	Show value	App Service



briefbot-frontend
Web App

You have multiple notifications. Switch to Notifications tab to get more details.

Resource group (move)	: BCSAI2025-DEVOPS-STUDENTS-A	Default domain	: briefbot-frontend.azurewebsites.net
Status	: Running	App Service Plan	: ASP-BCSAI2025DEVOPSSTUDENT1A-ab55 (B1: 1)
Location (move)	: West Europe	Operating System	: Linux
Subscription (move)	: Azure Simple IE Instituto de Empresa, S.L.	Health Check	: Not Configured
Subscription ID	: e0b9cada-61bc-4b5a-bd7a-52c606726b3b		
Tags (edit)	: Add tags		

Web app

Name	briefbot-frontend
Publishing model	Container
Container Image	index.docker.io/yamirghofran/briefbot-frontend:latest
Runtime status	Healthy

Domains

Default domain	briefbot-frontend.azurewebsites.net
Custom domain	Add custom domain

Hosting

Plan Type	App Service plan

Deployment Center

Deployment logs	View logs

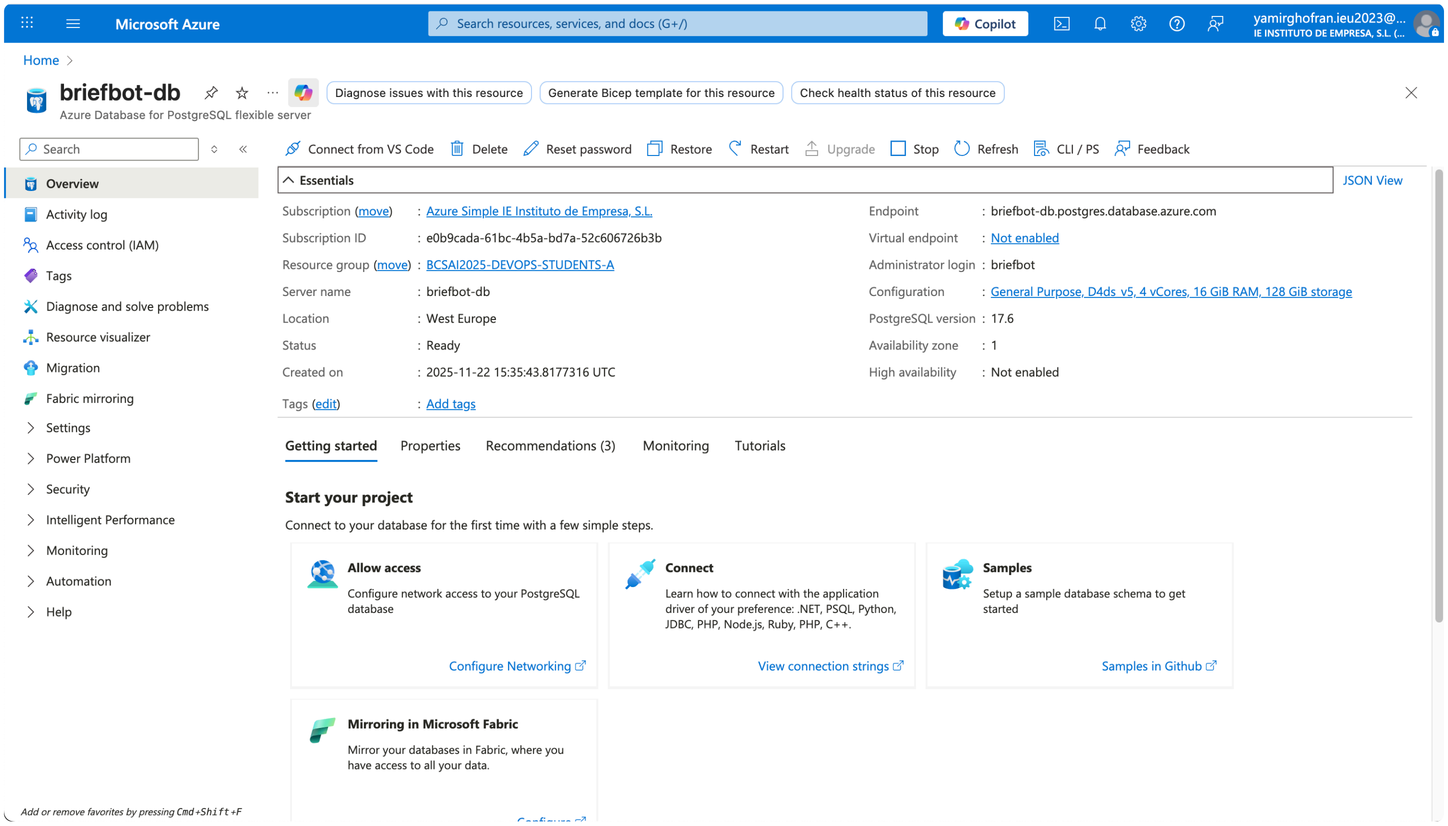
Application Insights

Name	Not supported. Learn more

Networking

Virtual IP address	20.50.2.94
Outbound IP addresses	135.236.89.194, 108.141.27.65, 20.67.59.47, 9.163.255.242, 132.220.19.8, 108.142.59.236, 128.251.211.129, 132.220.25.182, 98.64.12.115, 132.164.19.45, 57.153.161.115, 9.163.79.192, 20.50.2.94
Additional Outbound IP addresses	135.236.89.194, 108.141.27.65, 20.67.59.47, 9.163.255.242, 132.220.19.8, 108.142.59.236

Azure PostgreSQL



briefbot-db
Azure Database for PostgreSQL flexible server

Subscription (move)	: Azure Simple IE Instituto de Empresa, S.L.	Endpoint	: briefbot-db.postgres.database.azure.com
Subscription ID	: e0b9cada-61bc-4b5a-bd7a-52c606726b3b	Virtual endpoint	: Not enabled
Resource group (move)	: BCSAI2025-DEVOPS-STUDENTS-A	Administrator login	: briefbot
Server name	: briefbot-db	Configuration	: General Purpose, D4ds_v5, 4 vCores, 16 GiB RAM, 128 GiB storage
Location	: West Europe	PostgreSQL version	: 17.6
Status	: Ready	Availability zone	: 1
Created on	: 2025-11-22 15:35:43.8177316 UTC	High availability	: Not enabled
Tags (edit)	: Add tags		

Start your project

Connect to your database for the first time with a few simple steps.

Allow access
Configure network access to your PostgreSQL database
Configure Networking

Connect
Learn how to connect with the application driver of your preference: .NET, PSQL, Python, JDBC, PHP, Node.js, Ruby, PHP, C++.
View connection strings

Samples
Setup a sample database schema to get started
Samples in Github

Mirroring in Microsoft Fabric
Mirror your databases in Fabric, where you have access to all your data.
Configure