

Relatório Projeto – Cloud Azure

Elaborado por:

Rute Vanessa Freire Marques

Professor Responsável:

Miguel Costa

Coordenador do Curso:

Nuno Garcia

Lisboa

Março de 2025

Índice

Introdução.....	3
Exercícios e Explicação	4
Exercise 1 - Architecture Exercise.....	4
Exercise 1 - SQL Exercise	5
Exercise 2.	6
Exercise 3. Python	7
Exercise 3. Azure	10
Exercise 3. Docker.....	11
Exercise 4. - Docker.....	13
Exercise 5. - Flask.....	13
Exercise 6. - Flask	15
Exercise 7. - Docker and Flask.....	16
Exercise 8. - Azure SQL	17
Exercise 9. - Files.....	18
Exercise 10 - ACR - Container Registry.....	19
Exercise 11. - Azure Functions	21
Exercise 12. - Azure Blob Storage	22
Exercise 13. - Azure Queues.....	25
Exercise 14. App Functions Queue Trigger	26
Exercise 15. - Azure Tables.....	28
Exercise 16. - Azure Data Factory	28
Exercise 17. - Azure Blob Storage	32
Exercise 18. Parquet Files - Data Factory	33
Exercise 19. - Homework - Queue Triggers	35
Exercise 20. - Event Hub	36
Exercise 21. - Stream Analytics	37
Exercise 22. Homework - Kafka	38
Exercise 23. Homework - Cosmos DB.....	40
Exercise 24 - Log Analytics	41
Arquitetura EM AZURE – Use Case	42
Vantagens arquitetura Event Hub Vs arquitetura Queue	44
Explicar detalhada da Arquitetura escolhida.....	46
Conclusão.....	54

Introdução

Este relatório foi desenvolvido como parte da unidade curricular do Módulo Azure, integrada no programa UPskill, na FCUL - Faculdade de Ciências da Universidade de Lisboa, sob a orientação do professor Miguel Costa.

A unidade curricular foi crucial para a introdução aos principais conceitos de computação em nuvem, contando com uma componente teórica que cobriu todos os tópicos da certificação AZ-900 - Azure Fundamentals. Além disso, incluiu uma forte vertente prática, abordando diversos temas dessa certificação e ensinando o uso de ferramentas essenciais para trabalhar tanto em ambientes de nuvem quanto on-premises. Aprendemos a operar com bancos de dados utilizando plataformas como pgAdmin (Postgres) e DBeaver, explorando formas de aceder e executar diversas consultas nessas bases de dados.

Aprofundamos o domínio da linguagem Python, implementando processos de ETL, APIs, App Functions e Queue Triggers, entre outros códigos desenvolvidos. Também avançamos no entendimento de conceitos de virtualização, abordando o que são, como funcionam e para que servem as VMs (máquinas virtuais). Compreendemos o motivo pelo qual o Docker é amplamente utilizado em comparação com as VMs, destacando-se por sua rápida inicialização e menor ocupação de espaço em disco. Além disso, exploramos ferramentas serverless como as Azure Functions, que funcionam a partir de Triggers e geram custos apenas enquanto estão em execução. A escolha entre VMs, containers ou Azure Functions depende das necessidades específicas do ambiente de trabalho, do uso frequente ou eventual, e do nível de poder computacional exigido.

Estudamos conceitos como Git e GitHub, criando repositórios e armazenando documentos. Também aprendemos a utilizar o Postman para testar APIs e o Flask que é um microframework em Python usado para criar aplicações web de forma simples e flexível.

Exploramos a utilização de message brokers, como Kafka, filas (Queues) e outras tecnologias semelhantes. Também nos familiarizamos com várias ferramentas essenciais para o desempenho no ambiente de Cloud Azure, como Virtual Machines, Container Instances, Azure Functions, Queue Triggers, Storage Account, Event Hub, Stream Analytics, Cosmos DB, Log Analytics, entre outras.

Além disso, a componente teórica cobriu extensivamente os temas da certificação AZ-900 - Azure Fundamentals da Microsoft.

Os exercícios aqui descritos foram organizados de forma cronológica da sua execução durante estes quatro meses, permitindo rastrear o progresso no desenvolvimento do projeto. Iniciámos com a criação local das bases de dados (RAW e REPORT), seguido pelo processo de ETL e o uso de ferramentas mais avançadas do Azure.

No final do relatório, apresentamos a Arquitetura Azure escolhida, incluindo as vantagens, explicações sobre sua implementação e uma breve conclusão.

Exercícios e Explicação

Exercise 1 - Architecture Exercise

Company A is an e-commerce company and has these goals:

1.1. Create Daily Aggregations:

a. Top purchased products by day;

b. Number of reviews by product;

1.2. Return All User Ranked Products in Real Time;

1.3. Return Real Time User Recommendations;

1.4. Create Search Products service.

1.a) Esta query vai retorna os **produtos mais vendidos por dia**, onde consta a quantidade total de vendas de cada produto. Como é o primeiro exercício vou explicar alguns conceitos e comandos:

1º SELECT: Este comando tem como objetivo selecionar os dados necessários para a pesquisa, tal como o nome do produto, o total de vendas e a data (a qual é convertida a partir do sales_ts).

2º JOIN: Faz a ligação entre as tabelas raw.sales e raw.products usando product_id, garantindo assim que cada venda esteja associada ao respetivo produto.

3º GROUP BY: Agrupa as vendas primeiro por data e depois pelo nome do produto, e assim calcular as vendas por produto em cada dia.

4º ORDER BY: Ordena os resultados por dia e, dentro de cada dia, ordena os produtos do mais vendido para o menos vendido.

1.b) Esta query conta o número de reviews por produto e agrupa por dia, fornece o nome do produto, e ordena os resultados de forma cronológica. O comando INNER JOIN é similar ao comando JOIN.

2. Esta query devolve todos os produtos avaliados por utilizadores em tempo real, agrupando as avaliações por produto e ordena média de classificação (rating) de formar descendente, o que significa que as melhores classificações estão no topo da lista.

3. Esta query serve para recomendações em tempo real, baseia-se nos produtos com maior média de avaliação, mas filtrando por produtos com pelo menos um número mínimo de avaliações, no meu caso escolhi um mínimo de 6 reviews.

4. Esta query implementa uma funcionalidade de pesquisa por produtos pelo nome, permitindo procurar produtos com base em palavras-chave, onde substitui [keyword] pela palavra-chave que se deseja pesquisar.

→ **Github:** Ver na pasta Exercise_1, o documento “1Querys_Architecture Exercise”.

Exercise 1 - SQL Exercise

1.1. Create two databases:

a. raw;

b. report.

1.2. Create tables on both databases;

1.3. Insert data into raw tables.

Para criar as duas bases de dados foi usada a aplicação pgAdmin, sendo a linguagem usada a PostgreSQL, cada base de dados está em um Schema diferentes, para tal tive de criar dois Schemas. Posteriormente também foram criadas as Bases de dados no Azure e para nos conectarmos a essa Data Base usamos o DBeaver.

Foi criada a base de dados RAW e a base de dados REPORT.

A base de dados RAW tem as seguintes tabelas:

- products;
- reviews;
- sales;
- users.

A base de dados Report tem as seguintes tabelas:

- product_reviews;
- product_sales;
- ReportBestRatedProductsDataset;
- ReportAverageSpentPurchases
- ReportSoldQuantityByProduct
- ReportTopSpenders
- ReportTopUsersAverageReviews

→ **Github:** Ver na pasta Exercise_1 o documento “2Create_tables_Inserts_SQL Exercise” para verificar como foram criadas as duas Bases de Dados (raw, report), as suas tabelas e exemplos de possíveis Inserts nas diversas tabelas da Base de dados Raw.

Exercise 2.

Create the tables and queries needed for this use case (adjust the existing if needed):

2.1. Which users spent more money on purchases;

2.2. Considering that for each sale we will also have the quantity of products sold, calculate the sold quantity by product;

2.3. What was the average spent on purchases;

2.4. Which top 3 users had the higher average reviews;

2.5. Which products had the highest average reviews.

1. Esta query retorna os cinco usuários que mais gastaram em compras (em DBeaver usamos o comando TOP e em PgAdmin usamos o comando LIMIT), calcula o total gasto por usuário através da soma dos preços dos produtos adquiridos e ordena o resultado de forma decrescente pelo total gasto, o que quer dizer que em primeiro lugar na lista iremos ver os usuários que mais gastarem.

2. Esta query começa por contar quantas unidades de cada produto foram vendidas, agrupando os resultados pelo nome do produto e ordenando pela quantidade vendida, da maior quantidade para a menor.

3. Esta query calcula a média dos gastos por compra, onde primeiro, calcula o total gasto por cada usuário em cada venda e, em seguida, calcula a média de todos esses valores.

4. Esta query mostra-nos os três usuários que avaliaram os produtos com as maiores médias de avaliações (ratings). Agrupando pelos nomes de usuários e ordenando a media de avaliaram os produtos de forma decrescente.

5. Esta query retorna os três produtos com a maior média de avaliações. Calcula a média de rating de cada produto e organiza o resultado em ordem decrescente, ou seja, do produto com maior avaliação para o produto com menor avaliação.

→ **Github:** Ver na pasta Exercise_2 os documentos “DBeaver_Querys” e “PgAdmin_Querys”.

Exercise 3. Python

3.1. Adjust the pipeline to generate random results for SQL exercise 2

3.2. Log steps into logger file

1. Entre as linhas 1 e 5, o código importa módulos e funções essenciais para a execução das diferentes tarefas realizadas no programa. Esses módulos desempenham papéis fundamentais, como operações em bases de dados, geração de logs, manipulação de dados e criação de resultados aleatórios. A combinação desses componentes é crucial para o funcionamento eficiente do código.

Nas linhas 18 a 35, temos a função `connect_to_azure_sql()`, cuja finalidade é estabelecer uma conexão com uma base de dados do Azure SQL Server, utilizando configurações obtidas através da função `get_db_config()`. Quando a conexão é bem-sucedida, uma mensagem de sucesso é registrada no log. Em caso de falha, a exceção é tratada, uma mensagem de erro é registrada no log, e a função retorna `None`, indicando que não foi possível estabelecer a conexão.

Nas linhas 38 a 55, encontra-se a função `get_raw_sales(connection)`, que é responsável por recuperar os dados de vendas da base de dados RAW. Primeiramente, a função verifica se a conexão fornecida é válida. Em seguida, executa uma consulta SQL que retorna as informações sobre os produtos mais vendidos por dia. Caso a consulta falhe ou a conexão não seja válida, o erro é registrado no log e a função devolve `None`.

Nas linhas 79 a 106, é apresentada a função `get_raw_reviews(connection)`, que tem como objetivo obter os dados de reviews da base de dados RAW. Assim como na função anterior, verifica-se a validade da conexão antes de executar a consulta SQL. A consulta retorna o número de reviews por produto e por dia. Se houver falhas na execução ou problemas na conexão, o erro é registrado no log e a função retorna `None`.

Nas linhas 58 a 76, encontramos a função `insert_sales(connection, sales_data)`, que recebe dois parâmetros: uma conexão com a base de dados e os dados de vendas (resultantes da função `get_raw_sales(connection)`). Essa função insere as informações de vendas na base de dados Report, especificamente na tabela `product_sales`. O processo inclui a verificação da conexão, a limpeza da tabela com o comando `TRUNCATE` e a inserção de cada linha dos dados de vendas. Em caso de sucesso, as alterações são confirmadas, e uma mensagem de conclusão é registrada no log. Caso ocorra um erro, ele é capturado e registrado no log.

Entre as linhas 109 e 135, temos a função `insert_reviews(connection, reviews_data)`, que funciona de maneira semelhante à anterior, mas lida com os dados de reviews (gerados pela função `get_raw_reviews(connection)`). Essa função insere informações sobre os reviews na tabela `report.product_reviews` da base de dados Report. O processo inclui verificar a conexão, limpar os dados existentes na tabela utilizando `TRUNCATE`, e inserir novas informações, como data, nome do produto e número total de reviews. Assim como nas outras funções, qualquer erro encontrado é capturado e registrado no log.

Entre as linhas 138 e 155, encontra-se a função `get_products(connection)`, que é responsável por buscar os nomes dos produtos presentes na base de dados RAW.

Primeiramente, a função verifica se a conexão fornecida é válida; caso contrário, registra um erro no log e retorna None. Em caso de conexão válida, executa uma consulta SQL que seleciona e organiza, em ordem alfabética, os nomes dos produtos na tabela `raw.products`. Os resultados são processados em uma lista e registrados no log. Caso ocorra algum erro, este é capturado e registrado no log, e a função retorna uma lista vazia.

Nas linhas 158 a 176, temos a função `get_product_id_list(connection, products_names)`, que tem como objetivo recuperar os IDs dos produtos da tabela `raw.products`. Inicialmente, verifica se a conexão fornecida é válida; caso contrário, registra um erro no log e retorna None. Em caso de sucesso, executa uma consulta SQL que busca os IDs dos produtos. Os resultados são armazenados em uma lista denominada `products_list`, e cada ID é registrado no log. Após finalizar a operação, a lista de IDs é retornada. Caso ocorra algum erro durante o processo, o problema é registrado no log e a função retorna None.

Nas linhas 180 a 193, está a função `lista_de_datas(data_inicial_str, data_final_str)`, que gera uma lista de datas dentro de um intervalo especificado. Primeiro, converte as strings das datas de entrada em objetos `datetime`. Em seguida, utiliza um `while` para adicionar cada data ao intervalo, no formato 'YYYY-MM-DD', a uma lista chamada `lista_datas`. As datas são incrementadas dia a dia com o uso de `timedelta(days=1)`. Por fim, a lista gerada é registrada no log, em nível `DEBUG`, e retornada.

Entre as linhas 196 e 214, encontra-se a função `get_users_id_list(connection)`, que seleciona os IDs dos utilizadores da tabela `raw.users`. Inicialmente, verifica se a conexão fornecida é válida; caso contrário, registra um erro no log e retorna None. Caso seja válida, realiza uma consulta SQL para buscar os IDs, armazenando-os em uma lista chamada `user_list`, e registra cada ID no log para fins de monitorização. Após encerrar o cursor, a função retorna a lista de IDs. Se ocorrer algum erro, este é capturado e registrado no log, e a função retorna None.

Entre as linhas 217 e 252, está a função `gerar_sales_aleatorias(connection, data_inicial_str, data_final_str)`, que cria dados de vendas aleatórios para um intervalo de datas. Primeiramente, obtém os nomes dos produtos por meio da função `get_products` e os IDs dos utilizadores com `get_users_id_list`. Caso não encontre produtos ou utilizadores, registra um aviso no log e retorna uma lista vazia. Para cada dia do intervalo, gera um número aleatório de vendas e seleciona aleatoriamente um produto e um utilizador. O ID do produto é obtido por meio da função `get_product_id_list`, e um timestamp aleatório é criado para cada venda. Os dados gerados são armazenados como tuplas (`product_id`, `sales_ts`, `user_id`) e registrados no log, em nível `DEBUG`. Caso ocorra algum erro, o problema é registrado no log e a função retorna uma lista vazia.

Nas linhas 256 a 274, encontra-se a função `insert_sales_data(connection, sales_data)`, que insere dados de vendas na tabela `raw.sales`. Inicialmente, verifica se existem dados a serem inseridos; caso contrário, registra um aviso no log e interrompe a execução. Se houver dados, limpa previamente a tabela com o comando `TRUNCATE`, garantindo que está vazia antes de novas inserções. Em seguida, insere cada venda na tabela utilizando uma consulta SQL. Após completar o processo, confirma as alterações com `connection.commit()` e registra o número de vendas inseridas. Caso ocorra algum erro, este é registrado no log e as alterações são revertidas com `connection.rollback()`.

Entre as linhas 277 e 281, temos a variável `REVIEW_TEXTS`, que contém uma lista de textos predefinidos para reviews (avaliações). Estes textos podem ser utilizados para simular dados em aplicações.

Nas linhas 285 a 327, está a função `gerar_reviews_aleatorias(data_inicial_str, data_final_str, connection)`, que cria reviews aleatórios relacionados a produtos e utilizadores, dentro de um intervalo de datas. Primeiramente, obtém os nomes dos produtos por meio de `get_products` e os IDs dos utilizadores com `get_users_id_list`. Caso não existam dados, regista um aviso no log e retorna uma lista vazia. Para cada dia do intervalo, cria um número aleatório de reviews (entre 1 e 5), seleciona um produto e um utilizador aleatoriamente e obtém o ID do produto com `get_product_id_list`. Cada review inclui uma classificação (rating) de 1 a 5, um texto selecionado da lista `REVIEW_TEXTS`, uma data/hora aleatória e um ID de utilizador. Os dados gerados são registrados no log, em nível `DEBUG`, e retornados como tuplas no formato `(product_id, rating, review_text, review_date, user_id)`. Em caso de erro, o problema é registrado no log e a função retorna uma lista vazia.

Entre as linhas 331 e 354, temos a função `insert_reviews_data(connection, reviews_data)`, que insere reviews na tabela `raw.reviews`. Primeiramente, verifica a validade da conexão; caso não seja válida, regista um erro no log e retorna `None`. Em caso de conexão válida, limpa a tabela com `TRUNCATE` e insere cada review na tabela por meio de uma consulta SQL. Após concluir as inserções, confirma as alterações com `connection.commit()`, regista o número de reviews inseridas e encerra a conexão. Em caso de erro, o problema é capturado, registrado no log, e as alterações são revertidas com `connection.rollback()`.

Nas linhas 357 a 372, encontra-se a expressão `if __name__ == '__main__':`, uma estrutura especial em Python que define o ponto de entrada principal de um programa. Essa construção garante que determinados blocos de código sejam executados apenas quando o ficheiro é executado diretamente, e não quando é importado como módulo em outros programas.

2. Um `Logger File` é um ficheiro utilizado para guardar registos de atividades (logs) de sistemas e aplicações. Este tipo de ficheiro é fundamental para manter um histórico das operações realizadas, registando eventos relevantes, mensagens de erro, estados do sistema, variáveis e informações sobre a execução do código. Esses dados são extremamente úteis para monitorizar e diagnosticar o desempenho e comportamento das aplicações. Além disso, os logs podem ser configurados para diferentes níveis de detalhe, como informações gerais (`info`), avisos (`warning`), erros (`error`) e falhas críticas (`critical`), o que contribui para uma gestão eficiente e segura do sistema.

No ficheiro `main.py`, na linha 3, é utilizada a instrução `“import logging”` para carregar o módulo `logging` do Python. Este módulo permite implementar registos de eventos e mensagens dentro da aplicação. Entre as linhas 7 e 16, o código configura o logger para registar as mensagens num ficheiro, definindo como essas mensagens serão formatadas e guardadas. Por exemplo, no ficheiro `app.log`, são registados detalhes como a data, o nome do logger e o nível da mensagem. Esta configuração ajuda a monitorizar a aplicação e facilita o diagnóstico de problemas.

Se o ficheiro app.log for apagado, ao executar novamente o programa principal (main.py), o ficheiro será recriado automaticamente e continuará a armazenar os logs da aplicação. Além disso, o código substitui todos os comandos print por registos com o logger, o que faz com que as mensagens, em vez de serem exibidas na consola, sejam armazenadas diretamente no ficheiro app.log. Cada uma das mensagens no código foi ajustada para incluir o respetivo nível de detalhe, permitindo uma organização clara e eficiente dos registos.

→ **Github:** Ver na pasta Exercise_3\1_Python_ETL os documentos “main.py” e “app.log”

Exercise 3. Azure

Check in the portal how to launch and answer these questions:

3.1. VM's - Can it be multi region? What would be the solution?

Sim, as máquinas virtuais (VMs) podem ser configuradas para funcionar em múltiplas regiões no Azure, mas a mesma máquina virtual não pode estar em duas regiões ao mesmo tempo, mas posso ter servidores nos EUA e outros servidores na Europa. Uma opção seria usar o **Azure Site Recovery** para replicar e recuperar os dados em caso de desastres nas regiões, mas para além disso, podem ser configurados **Load Balancers Globais** para distribuir o tráfego entre as VMs em diferentes regiões, e assim podemos garantir uma alta disponibilidade e resiliência.

3.2. What is the difference between Azure SQL and a Postgres DB?

O Azure SQL é um serviço que é gerido com base no Microsoft SQL Server, este oferece alta compatibilidade com T-SQL, é possível escalar automaticamente e tem uma integração nativa com muitos serviços fornecidos pelo Azure. O Azure SQL é um serviço ideal para quem já usa o ecossistema Microsoft.

O PostgreSQL é uma base de dados open-source, esta pode ser gerida através do Portal do Azure usando o serviço PostgreSQL do Azure Database. O PostgreSQL suporta extensões e múltiplos padrões de dados, oferecendo assim uma maior personalização para aplicações específicas.

3.3. What is the purpose of Azure Cosmos DB?

O Azure Cosmos DB é um serviço do Azure bastante usado, é um banco de dados distribuído e é facilmente escalável, este foi desenvolvido para aplicações que necessitam de precisão de baixíssima latência. Com o Azure Cosmos DB podemos realizar replicação global de dados, uma vez que suporta diversos modelos de dados, como documentos e grafos, é ideal para sistemas que requerem consistência configurável e alta disponibilidade.

3.4. What is the goal of Azure Data Factory?

O Azure Data Factory é usado essencialmente para a criação de pipelines de integração e transformação de dados, uma vez que ele permite mover dados de diferentes fontes, processá-los e prepará-los para análises. É amplamente aplicado em operações de ETL (extrair, transformar e carregar).

3.5. What is the goal of Azure DataBricks?

O Azure Databricks é uma ferramenta de Azure que foi desenvolvida para o processamento de grandes volumes de dados, como a criação de modelos de machine learning e de análises em tempo real, este é baseado no Apache Spark, o que é ideal para projetos de Big Data e de Inteligência Artificial, além disso ainda proporciona uma integração com outros serviços Azure.

Exercise 3. Docker

Create a Docker file to:

3.1. Execute a python file that prints to the console “Hello World”;

3.2. Connects to the local db and executes a query and shows the result to the console.

3.1. Podemos ver na pasta dois ficheiros o “Dockerfile” e “app.py” na pasta Exercise_3\2_hello_docker\app_hello_world.

O Dockerfile cria um container para executar uma aplicação em Python. Primeiro, utiliza a imagem base python:3.9-slim, que é uma versão leve do Python 3.9. Define o diretório de trabalho como /app, onde os comandos a seguir serão executados. O arquivo app.py é copiado para este diretório no container. Depois, instala o SQLite e a biblioteca Python pycopg2-binary para interagir com bases de dados PostgreSQL. O comando final especifica que o container deve executar python app.py para iniciar o aplicativo ao ser executado. Este Dockerfile garante um ambiente configurado para executar o aplicativo de forma leve e portátil.

O ficheiro app.py, é um ficheiro bastante simples, que importa a biblioteca pycopg2, que é a biblioteca que é usada para conectar e interagir com bases de dados PostgreSQL. Depois exibe uma mensagem no terminal: O comando print("Hello World") imprime o texto "Hello World" no terminal, servindo como uma verificação básica de funcionamento do programa.

Para criar a imagem em docker, executamos na linha de comandos o seguinte comando: “docker build -t hello-world-python-app .” e para executar o container #docker run --name hello-world-container hello-world-python-app”.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\Estudos\Programação\UpSkill\05_Modulo_
_docker\app_hello_world> python -u "c:\Estud
cao\Exercicios\Exercise_3\2_hello_docker\app
Hello World
```

Figura 1 – Hello World – Docker

→ **Github:** Ver na pasta \Exercise_3\2_hello_docker\app_hello_world os documentos “Dockerfile” e “app.py”

3.2. O código do ficheiro **Dockerfile** é similar ao que vimos no exercício anterior, o que altera é somente o nome dado à imagem Docker e ao container.

O ficheiro **app.py** conecta-se à base de dados PostgreSQL local para executar uma consulta e apresentar os resultados relacionados a reviews de produtos. Primeiramente, importa a biblioteca **psycopg2** para gerir a conexão com a base de dados. Em seguida, estabelece a conexão utilizando credenciais específicas (como host, base de dados, utilizador, palavra-passe e porta). A consulta `query_reviews` seleciona o nome dos produtos e conta o número de reviews associados, agrupando os dados pelo nome do produto e ordenando-os alfabeticamente. Os resultados da consulta são extraídos e apresentados no terminal. Se ocorrer algum erro durante o processo, este é tratado e exibido. Por fim, a conexão com a base de dados é fechada para garantir a boa gestão de recursos.

Este código tanto corre no terminal do Visual studio como diretamente no container do Docker, temos aqui uma imagem de o código a correr no Docker.

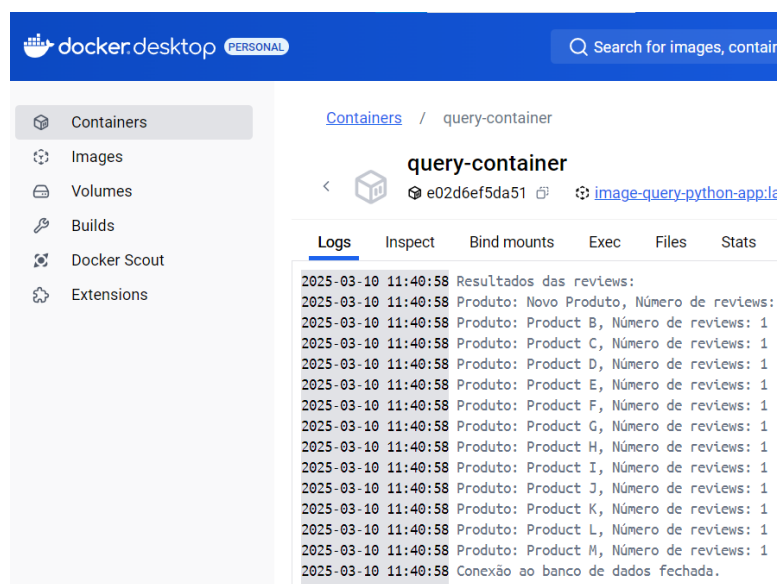


Figura 2 – Gerador de dados - Docker

→ **Github:** Ver na pasta \Exercicios\Exercise_3\2_hello_docker\app_hello_world+reviews os documentos “Dockerfile” e “app.py”.

Exercise 4. - Docker

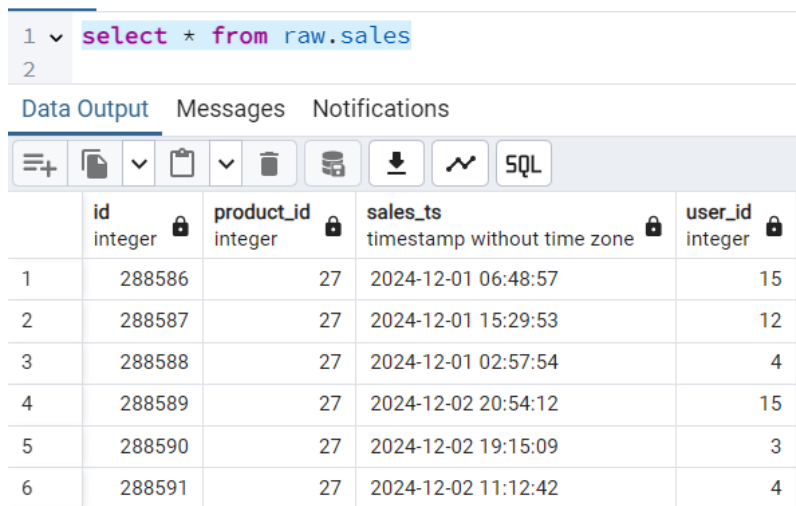
4.1. Deploy and run the python data processor that inserts data into the raw database;

4.2. Run with a connection to a local database or on a container with a postgres db.

Este código estabelece uma conexão com uma base de dados PostgreSQL utilizando as configurações fornecidas pelo módulo config para obter as credenciais. Após a conexão, obtém informações como nomes de produtos e IDs de utilizadores presentes na base de dados. Através dessas informações, o código gera dados de vendas dentro de um intervalo de datas especificado, associando produtos e utilizadores de forma aleatória. As vendas geradas incluem timestamps criados dinamicamente. Após gerar esses dados, a aplicação insere as vendas na tabela raw.sales, garantindo que a tabela seja limpa antes da nova inserção. O código também implementa tratamento de erros para assegurar a robustez do sistema, capturando e registrando falhas durante a execução.

Relativamente ao código Docker é realizado, é similar ao exercício anterior.

Os dados são inseridos na base de dados com sucesso:



The screenshot shows a database management tool interface. At the top, a SQL query is entered: `select * from raw.sales`. Below the query, there are tabs for 'Data Output', 'Messages', and 'Notifications'. The 'Data Output' tab is active, displaying a table with 6 rows of data. The table has columns: 'id' (integer), 'product_id' (integer), 'sales_ts' (timestamp without time zone), and 'user_id' (integer). Each column header has a lock icon. The data rows are as follows:

	id integer	product_id integer	sales_ts timestamp without time zone	user_id integer
1	288586	27	2024-12-01 06:48:57	15
2	288587	27	2024-12-01 15:29:53	12
3	288588	27	2024-12-01 02:57:54	4
4	288589	27	2024-12-02 20:54:12	15
5	288590	27	2024-12-02 19:15:09	3
6	288591	27	2024-12-02 11:12:42	4

Figura 3 – Base de Dados Raw

➔ **Github:** Ver na pasta \Exercicios\Exercise_4\ os documentos os documentos “Dockerfile” e “app.py”.

Exercise 5. - Flask

Create and API with Flask , with a GET request returning Hello World.

Primeiro passo foi instalar o Flask no terminal ou prompt de comando “pip install flask”, depois de realizar o download de todos os pacotes necessários, o código correu sem problemas.

O código cria uma aplicação web simples utilizando o Flask, um microframework em Python. Inicialmente, importa-se o módulo Flask para construir a aplicação. A variável

app representa a aplicação Flask. A função helloworld é criada e associada à rota / com o decorador @app.route("/"), ou seja, esta função será executada quando a URL raiz da aplicação for acessada.

Ao executar o código, o servidor web Flask será iniciado localmente na porta 5000 com o modo de depuração ativado (debug=True), o que facilita a identificação de erros durante o desenvolvimento. Quando se acede ao endereço `http://127.0.0.1:5000` no navegador, a função helloworld retorna a mensagem "Hello World!", que será exibida no navegador.

Agora passando para a execução do código:

```
* Serving Flask app 'tempCodeRunnerFile'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 457-112-094
```

Figura 4 – Hello World – Flask

Num Browser colocamos o URL “`http://127.0.0.1:5000`” e como esperado apareceu a mensagem “Hello World!”.

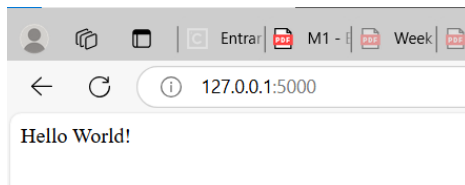


Figura 5 – Hello World Browser – Flask

→ **Github:** Ver na pasta \Exercicios\Exercise_5\ o ficheiro “hello_world.py”.

Exercise 6. – Flask

Create an API with Flask, with a POST, that will receive a JSON and will insert into the DB.

Podemos verificar na pasta o ficheiro flask_api.py, onde podemos ver código que define uma API Flask com um endpoint do tipo POST (/data) que aceita um JSON contendo os campos product, sales_ts e user_id. Existe uma conexão à base de dados de Azure SQL Server db-04 usando pyodbc e insere esses dados na tabela raw.sales. De referir que a conexão à base de dados é gerida pela função connect_to_azure_sql, onde se caso ocorra um erro na conexão, na validação dos dados ou na inserção, o código trata as exceções e retorna mensagens de erro apropriadas com códigos HTTP correspondentes. Além disso, todas as conexões são fechadas adequadamente após o uso. A API está configurada para rodar localmente no modo de debug.

Para testar a API, utilizámos o Postman. Primeiro, executámos o arquivo flask_api.py localmente no VSCode. Em seguida, no Postman, inserimos o URL com a rota correspondente, neste caso http://localhost:5000/data, seleccionámos o método HTTP **POST**, e inserimos o código JSON desejado no corpo da requisição. Depois, clicámos em "Send". Se tudo funcionar corretamente, o Postman retornará a mensagem **"message": "Dados inseridos com sucesso"**, confirmando que os dados foram adicionados com sucesso à base de dados. Essa inserção pode ser verificada executando um comando SQL como SELECT * FROM raw.sales, cujo resultado é mostrado na imagem abaixo.

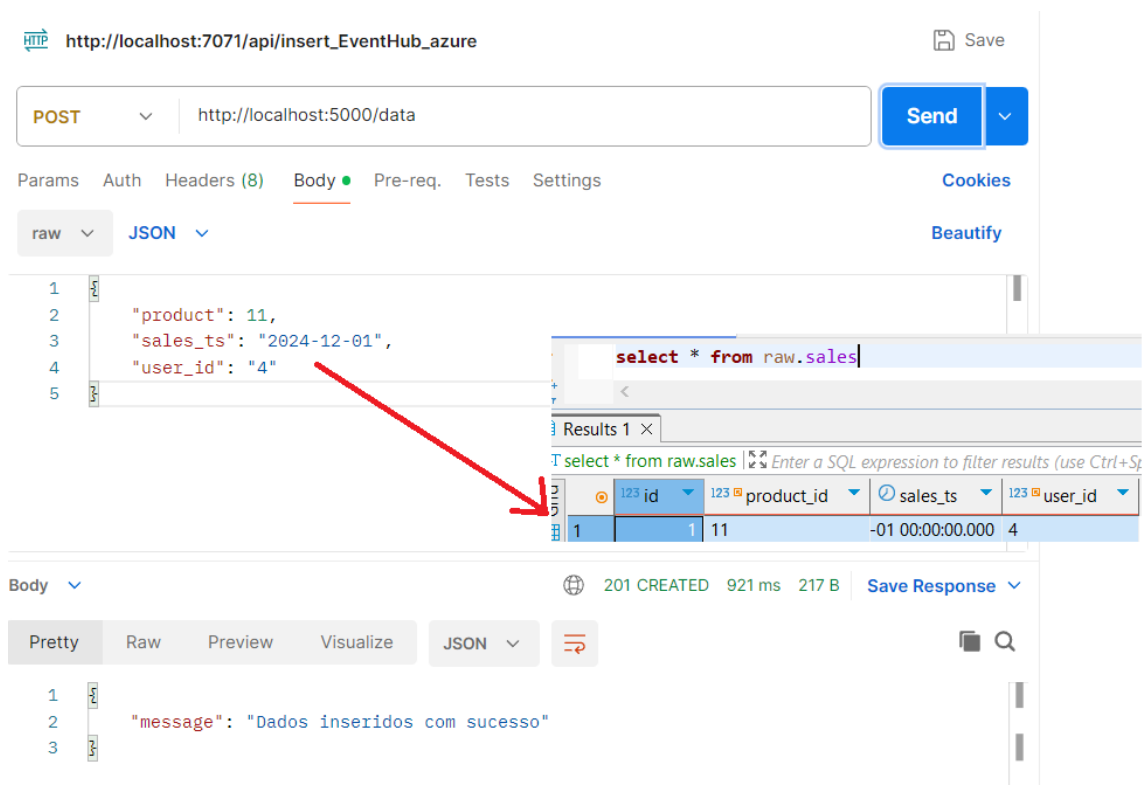


Figura 6 – Insert BD – Flask

→ **Github:** Ver na pasta \Exercicios\Exercise_6\ o ficheiro “flask_api.py”.

Exercise 7. - Docker and Flask

Change the Docker container to do a POST request to the FlaskAPI and will insert the data into the raw db.

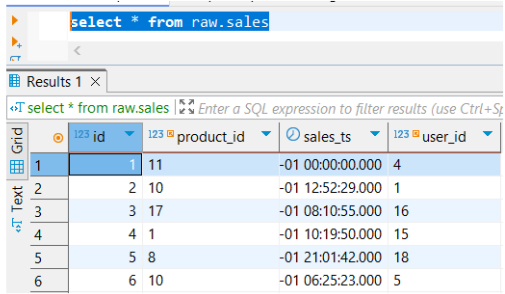
Este exercício é similar ao anterior no que diz respeito à Flask API, porém tive de realizar algumas pequenas alterações, como realizar a **configuração do Flask para aceitar conexões externas, para tal tive de** alterar a linha `app.run()` para aceitar conexões de qualquer host (embora também pudesse ter especificado um IP específico), onde ficou:

```
“
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)
“
```

Em relação ao gerador de dados, ou seja, o Docker (`app.py`), inicialmente foi necessário garantir que o JSON enviado estava em conformidade com o formato esperado pela API Flask. Além disso, foi preciso ajustar o URL para o qual o Docker envia as informações geradas, que, no caso, ficou definido como `url = "http://localhost:5000/data"` (atualizado para a API Flask local). No restante, o código do `app.py` permanece semelhante ao apresentado anteriormente.

Para testar este código, primeiramente foi necessário executar a FlaskAPI, depois de inicializada a API, inicializei o docker e a API começou a receber os dados do docker e a inseri-los na tabela `raw.sales` como podemos verificar na imagem a baixo.

```
PS C:\Estudos\Programação\UpSkill\05_Modulo_Azure\Projeto_Avaliacao\Exercicios\Exercise_7\1.FlaskAPI> python -u "c:\Estudos\Programação\UpSkill\05_Azure\Projeto_Avaliacao\Exercicios\Exercise_7\1.FlaskAPI\flask_api.py"
* Serving Flask app 'flask_api'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://192.168.1.230:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 457-112-094
Conexão bem-sucedida ao Azure SQL Server.
127.0.0.1 - - [15/Mar/2025 15:25:25] "POST /data HTTP/1.1" 201 -
Conexão bem-sucedida ao Azure SQL Server.
127.0.0.1 - - [15/Mar/2025 15:25:29] "POST /data HTTP/1.1" 201 -
Conexão bem-sucedida ao Azure SQL Server.
127.0.0.1 - - [15/Mar/2025 15:25:32] "POST /data HTTP/1.1" 201 -
Conexão bem-sucedida ao Azure SQL Server.
127.0.0.1 - - [15/Mar/2025 15:25:36] "POST /data HTTP/1.1" 201 -
Conexão bem-sucedida ao Azure SQL Server.
127.0.0.1 - - [15/Mar/2025 15:25:39] "POST /data HTTP/1.1" 201 -
```



	id	product_id	sales_ts	user_id
1	1	11	-01 00:00:00.000	4
2	2	10	-01 12:52:29.000	1
3	3	17	-01 08:10:55.000	16
4	4	1	-01 10:19:50.000	15
5	5	8	-01 21:01:42.000	18
6	6	10	-01 06:25:23.000	5

Figura 7 – Insert BD – Flask & Docker

→ **Github:** Ver na pasta `\Exercicios\Exercise_7`".

Exercise 8. - Azure SQL

8.1. Create and Azure Serverless DB with the previous tables;

8.2. Change the Flask API to insert into the new Azure DB.

A criação de uma Base de dados em Azure Serverless é bastante simples, para tal basta que no portal do Azure, procuremos na barra de pesquisa por “**SQL databases**” e criarmos uma nova de acordo com as configurações que achamos mais adequadas, escolhendo a subscrição, o grupo de recursos, o nome da base de dados, se ainda não tivermos criado um servidor temos de criar um e assim por diante, de seguida realizamos a conexão da Base de dados do Azure ao DBeaver, com podemos ver na imagem a baixo, e procedeu-se à criação das tabelas necessárias, de acordo com as tabelas que já tínhamos no Pgadmin.

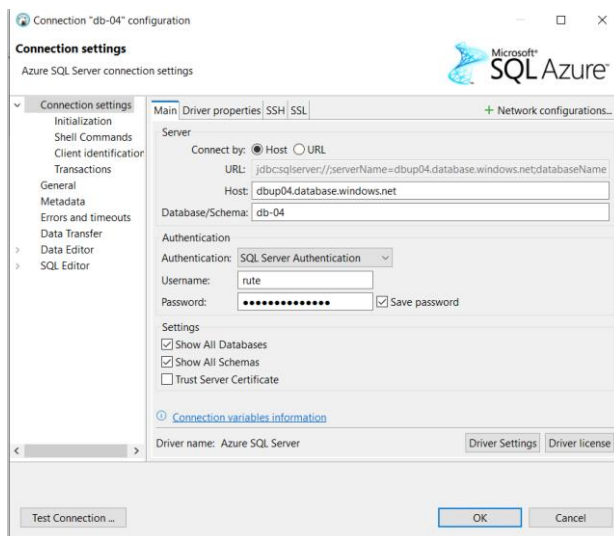


Figura 8 – Configuração - DBeaver

Para adaptar a Flask API e permitir a inserção de dados em uma base de dados local no PgAdmin ou em uma base de dados no Azure, basta modificar a função de connection (a função que retorna a connection) e atualizar as credenciais corretamente, garantindo que estejam de acordo com a base de dados à qual desejamos nos conectar.

→ **Github:** Ver na pasta \Exercicios\Exercise_8”.

Exercise 9. - Files

9.1. Change the Flask API to insert into the db and create files with json on in the same endpoint;

9.2. Create two endpoints in the Flask API, one to insert into the db and another to create files.

9.1 Para a realização deste exercício foi necessário realizar o import da biblioteca os e a inserção do código que consta nas linhas 34 a 42 onde este trecho de código cria um arquivo JSON contendo os dados recebidos, onde gera um nome único para o arquivo com base nos valores de product_id, user_id e sales_ts, substituindo caracteres que não podem ser usados em nomes de arquivos. O código garante que o diretório json_files exista e, em seguida, escreve os dados no arquivo JSON utilizando a codificação UTF-8.

Como anteriormente foi utilizada a ferramenta Postman para testar a Flask API, verificou-se que os dados foram inseridos com sucesso, tanto na base de dados (BD) quanto na pasta json_files.

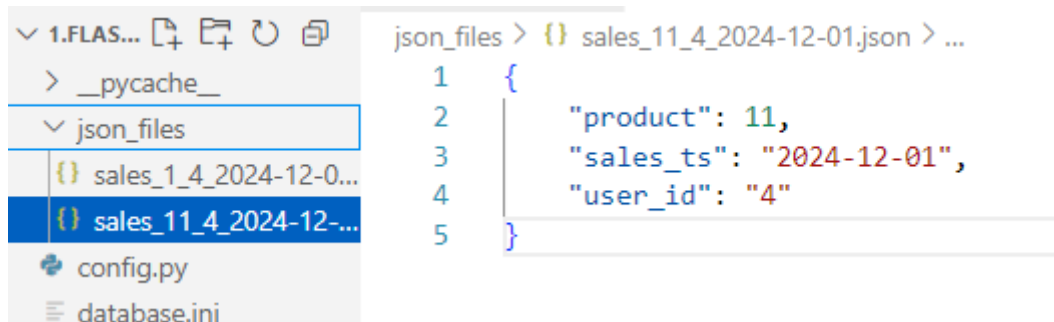


Figura 9 – DB & File - Flask

→ **Github:** Ver na pasta \Exercicios\Exercise_9\9.1.FlaskAPI”.

9.2. Para a realização deste exercício foi necessário realizar algumas pequenas alterações ao código anterior. Onde se começou por criar **dois endpoints separados** o /insert_db e outro /create_file, onde o endpoint /insert_db insere os dados na base de dados e o endpoint /create_file criar arquivos JSON a partir dos dados recebidos. É importante a separação das funções por endpoint, uma vez que assim cada endpoint tem a suas **responsabilidades, onde** cada endpoint realiza uma única tarefa (inserir na base de dados ou criar arquivos), promovendo um código mais organizado e fácil de manter. Posteriormente também ajustei as mensagens que cada endpoint, para assim fornecerem uma **resposta ajustada e assim indicar** o sucesso ou falha de cada endpoint, oferecendo um feedback claro ao utilizador. Em ambos os endpoints verifica-se se os campos obrigatórios (product, sales_ts, user_id) estão presentes no JSON antes de executar suas funções.

Como mencionado anteriormente, utilizamos o Postman para testar a API, obtendo os resultados esperados em ambos os endpoints, conforme é possível verificar na imagem abaixo.

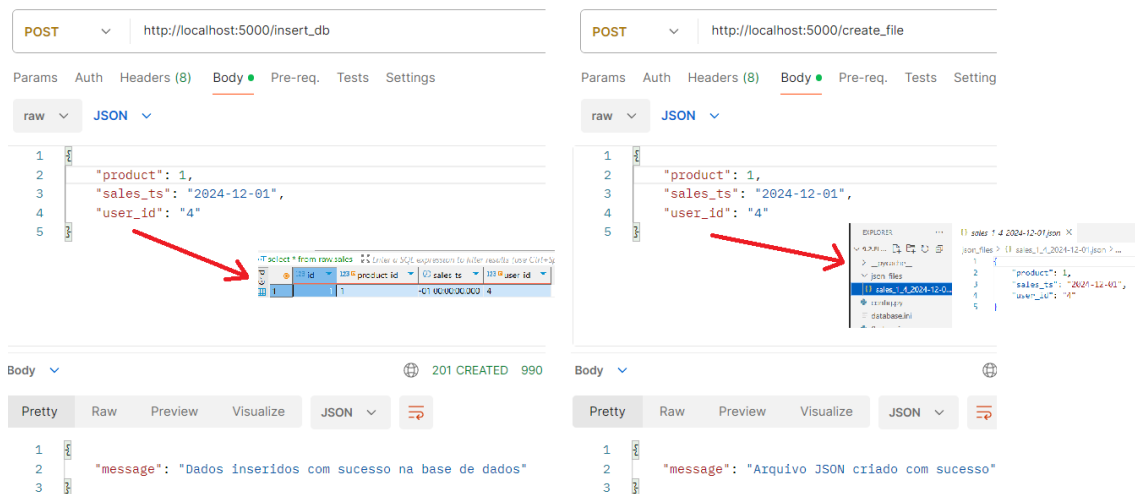


Figura 9 – DB & File two endpoints - Flask

→ **Github:** Ver na pasta \Exercicios\Exercise_9\9.2.FlaskAPI”.

Exercise 10 - ACR - Container Registry

10.1. Deploy the Docker local image into the Azure Container registry;

10.2 Run a Container instance on Azure to do two POST requests to the Flask API.

10.1. Para uma melhor compreensão deste exercício, é importante primeiro explicar o que são a Azure Container Registry (ACR) e a Azure Container Instance (ACI). A Azure Container Registry (ACR) é uma ferramenta destinada ao armazenamento e gerenciamento privado de imagens de contêiner. Funciona como um repositório que permite guardar, versionar e compartilhar essas imagens com diferentes ambientes, como Kubernetes ou Azure Container Instances. Já a Azure Container Instance (ACI) possibilita a execução de contêineres de maneira simples e ágil, sem a necessidade de gerenciar servidores ou infraestrutura. Essa solução é especialmente adequada para cenários como tarefas pontuais, aplicativos baseados em eventos ou processamento de dados.

O primeiro passo consiste em aceder ao Portal do Azure e criar uma Azure Container Registry. Em seguida, é necessário adicionar pelo menos uma imagem a essa Container Registry. Para isso, utilizamos a linha de comando no PowerShell começamos por fazer login no Azure Portal com o seguinte comando:

az login --tenant 6648d2ab-84ad-4a19-8ae1-c37cf174a849.

Neste momento, devemos inserir as nossas credenciais de acesso ao Portal do Azure, aceitar os termos e condições, e selecionar a conta desejada.

```
PS C:\Users\rutev> az login --tenant 6648d2ab-84ad-4a19-8ae1-c37cf174a849
Select the account you want to log in with. For more information on login with Azure CLI, see https://go.microsoft.com/fwlink/?linkid=2271136

Retrieving subscriptions for the selection...

[Tenant and subscription selection]

No. Subscription name Subscription ID Tenant
-----
[1] * FCUL-Azure 5fd70e4d-ae4e-42cf-8f1e-717d676ceedd 6648d2ab-84ad-4a19-8ae1-c37cf174a849
[2] FCUL-UPSKILL dabe8fcf-18f4-4bd6-9832-556535bb778d 6648d2ab-84ad-4a19-8ae1-c37cf174a849

The default is marked with an *; the default tenant is '6648d2ab-84ad-4a19-8ae1-c37cf174a849' and subscription is 'FCUL-Azure' (5fd70e4d-ae4e-42cf-8f1e-717d676ceedd).

Select a subscription and tenant (Type a number or Enter for no changes): 1

Tenant: 6648d2ab-84ad-4a19-8ae1-c37cf174a849
Subscription: FCUL-Azure (5fd70e4d-ae4e-42cf-8f1e-717d676ceedd)

[Announcements]
With the new Azure CLI login experience, you can select the subscription you want to use more easily. Learn more about it and its configuration at https://go.microsoft.com/fwlink/?linkid=2271236
If you encounter any problem, please open an issue at https://aka.ms/azclibug

[Warning] The login output has been updated. Please be aware that it no longer displays the full list of available subscriptions by default.
```

Figura 10 – Login Azure

az acr login --name container04 → Em seguida, realizamos o login no container registry para o qual queremos enviar a imagem. Na imagem apresentada, é possível confirmar que o login foi efetuado com sucesso.

docker images → Este comando permite visualizar as imagens locais disponíveis no Docker.

```
PS C:\Users\rutev> az acr login --name container04
Login Succeeded
PS C:\Users\rutev> docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
container04.azurecr.io/image_eventhub latest ebad02859117 13 hours ago 574MB
image_eventhub latest ebad02859117 13 hours ago 574MB
```

Figura 11 – Docker images

docker push container04.azurecr.io/image_eventhub:latest → Este comando é utilizado para enviar a imagem criada localmente para a Azure Container Registry, garantindo que ela fique armazenada de forma centralizada e pronta para ser utilizada em diferentes ambientes.

```
.dados> docker push container04.azurecr.io/image_eventhub:latest
The push refers to repository [container04.azurecr.io/image_eventhub]
c382c3cb70fd: Pushing [=====>] 4.194MB/4.778MB
7cf63256a31a: Pushing [=====] 3.146MB/28.22MB
82a59f028fa3: Pushed
1cce91ca30f0: Pushed
7c3c8cf8b5c0: Pushed
a64b7cccb523: Pushing [=] 3.146MB/92.65MB
fb0009da06dd: Pushed
768545f3e542: Pushed
09f8c21e9f8e: Pushing [=====] 7.34MB/14.94MB
```

Figura 12 – Push container04 to Azure

az acr repository list --name container04 --output table → Este comando exibe uma lista das imagens armazenadas na Azure Container Registry de forma organizada, utilizando o formato de tabela para facilitar a visualização.

```
PS C:\Users\rutev> az acr repository list --name container04 --output table
Result
-----
image_eventhub
my-python-app
my-python-app1
```

Figura 13 – Azure images

Após confirmarmos que a nossa imagem está armazenada na Azure Container Registry, retornamos ao Portal do Azure para criar a nossa Container Instance. É nela que o nosso código será executado. O processo de criação é muito semelhante ao de outros recursos no Azure, com a diferença principal na etapa **Image Source**, onde selecionamos a Azure Container Registry previamente criada e escolhemos a imagem que desejamos utilizar.

Image source * ⓘ ☐ Quickstart images ☒ Azure Container Registry ☐ Other registry

Run with Azure Spot discount ⓘ ☐
 ⓘ Spot containers are not available in the selected region. [Learn more](#) ⓘ

Registry * ⓘ
 ⓘ If you do not see your Azure Container Registry, ensure you have been assigned the Reader Role for the Azure Container Registry or select an Azure Container Registry in a different subscription. [Learn more](#) ⓘ

Image * ⓘ
 ⓘ

Image tag * ⓘ
 ⓘ

OS type * ☒ Linux ☐ Windows

Size * ⓘ
 1 vcpu, 1.5 GiB memory, 0 gpus
 [Change size](#)

Figura 14 – Container Instance

10.2 Agora é necessário garantir que a nossa Flask API esteja em execução. Em seguida, podemos ativar a nossa Container Instance no Azure. O processo é semelhante ao abordado no exercício anterior, onde a Container Instance será responsável por gerar os dados (anteriormente era o docker localmente), enquanto a Flask API os receberá e direcionará para os seus respectivos endpoints (Base de Dados e Ficheiro).

Exercise 11. - Azure Functions

Create an App function with a GET request, that return a Hello World

O código do ficheiro `function_app.py` implementa uma função HTTP utilizando o Azure Functions, que é uma solução de computação serverless fornecida pela Microsoft. Ele define um endpoint HTTP chamado `http_trigger1`, configurado para operar com autenticação no nível "ANONYMOUS", permitindo que qualquer cliente envie requisições sem necessidade de autenticação.

A função processa requisições HTTP e busca por um parâmetro chamado `name`. Este parâmetro pode ser fornecido diretamente na query string da URL ou como parte do corpo da requisição em formato JSON. Caso o parâmetro seja encontrado, a função retorna uma mensagem personalizada no formato "Hello, [name]". Se o parâmetro não for enviado, retorna uma mensagem padrão: "Hello, World".

Adicionalmente, o código utiliza o módulo `logging` para registar mensagens informativas sobre o processamento da requisição, o que é útil para monitoramento e depuração. A arquitetura do Azure Functions permite que este código seja escalado

automaticamente com base na demanda, tornando-o ideal para aplicações que exigem alta flexibilidade e baixa manutenção.

Podemos ainda ver na pasta diversos ficheiros (`__pycache__`/, `function_app.py`, `host.json`, `local.settings.json`, `ReadMe.txt` e `requirements.txt`), esses ficheiros juntos formam a base de um projeto estruturado e configurado para execução local e possivelmente em um ambiente de produção, como o Azure Functions e assim a app function pode correr corretamente.

Na linha de comando executamos o comando “func start”, é nos fornecido um URL, e colocamos esse URL no Brower e conseguimos ver com sucesso a mensagem “Hello, World.”

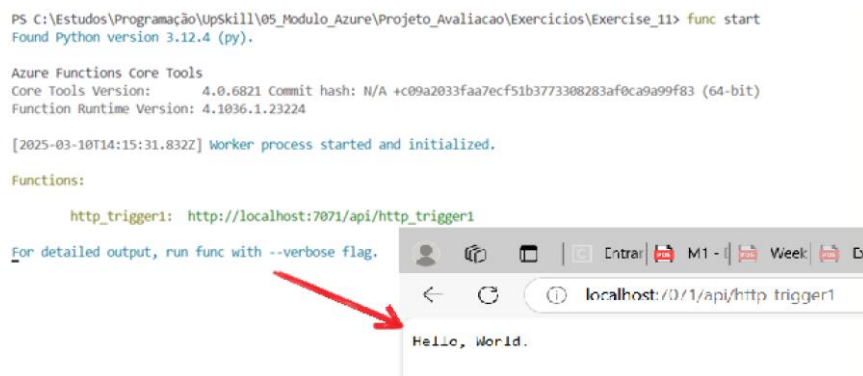


Figura 15 – Hello World - Azure Functions

→ **Github:** Ver na pasta \Exercicios\Exercise_11\.

Exercise 12. - Azure Blob Storage

12.1. Create two Endpoints in the App Function to receive POST Requests:

- One endpoint inserts into the DB;**
- Another endpoint uploads files to the Blob Storage.**

12.2. Change the Docker to send the requests to the new endpoints.

1. O código do ficheiro `function_app.py` implementa uma aplicação baseada no Azure Functions, com dois endpoints, uma para inserir dados na Base de Dados do Azure SQL e o outro endpoints permite o armazenamento de dados em arquivos no Azure Blob Storage.

Primeiramente, para iniciar a aplicação o código utiliza o módulo `azure.functions` para criar uma aplicação serverless com autenticação configurada para ANONYMOUS, o que significa que qualquer cliente pode aceder os endpoints sem necessidade de autenticação. De seguida passamos para a configuração do Azure Storage, onde se define uma string de conexão (`STORAGE_CONNECTION_STRING`) para interagir com o Azure Blob Storage, especificando as credenciais da conta e também garante que a pasta local

chamada "Files" exista para armazenar temporariamente os arquivos JSON antes de enviá-los para o Azure.

O endpoint para inserir dados na Base de dados, definimos a rota: `insert_db_azure`, sendo que este endpoint recebe dados de vendas via requisições POST (em JSON) e insere essas informações em uma tabela chamada `raw.sales` na Azure SQL Database. Para tal estabelece uma conexão com o banco de dados utilizando o driver ODBC, verifica e valida o corpo da requisição (`product`, `sales_ts`, e `user_id`), caso todos os dados sejam válidos, insere-os no banco utilizando uma query SQL parametrizada (`INSERT INTO raw.sales`) e por fim retorna uma resposta indicando sucesso ou falha.

Um exemplo de JSON Recebido:

```
{  
  "product": 11,  
  "sales_ts": "2024-12-01",  
  "user_id": "4"  
}
```

O endpoint para armazenamento no Azure Blob Storage, tem a rota: `insert_blobfile_azure`, este endpoint recebe um JSON com dados de vendas, escreve esses dados em arquivos JSON locais e os faz upload para o Azure Blob Storage. Primeiramente recebe os dados no corpo da requisição em formato JSON, gera um UUID (identificador único) para nomear cada arquivo gerado, garantindo nomes exclusivos, de seguida escreve o conteúdo do JSON recebido em um arquivo local (`Files_<UUID>.json`), faz o upload do arquivo para o Azure Blob Storage, dentro do container `container04`.

Podemos verificar que cada um dos endpoints inclui tratamento de erros usando `try-except` para capturar problemas durante o processamento (como falhas na conexão com o banco de dados ou no upload para o Azure). Os erros são registados nos logs, o que ajuda na depuração e no monitoramento.

Na linha de comandos, utilizamos o comando `func start` para iniciar a execução do código, o que disponibiliza os dois endpoints definidos na aplicação. Em seguida, podemos utilizar uma ferramenta como o Postman para enviar requisições do tipo POST e testar o funcionamento da API, garantindo que os endpoints estão a responder corretamente e a realizar as operações esperadas. O Postman é uma ferramenta amplamente usada para testar APIs, e envia somente uma requisição do tipo POST, pelo que depois de testada usei o Docker para enviar os dados das Sales, o que foi um sucesso.

```

PS C:\Estudos\Programação\UpSkill\05_Modulo_Azure\Projeto_Avaliacao\Exercicios\Exercise_1
pp_funtion_azurite_Insert_BlobFile&BD_AZURE> func start
Found Python version 3.11.0 (python).

Azure Functions Core Tools
Core Tools Version:          4.0.6821 Commit hash: N/A +c09a2033faa7ecf51b3773308283af0ca9a9
(64-bit)
Function Runtime Version: 4.1036.1.23224

[2025-03-10T15:07:15.557Z] Worker process started and initialized.

Functions:

    insert_blobfile_azure: [POST] http://localhost:7071/api/insert_blobfile_azure

    insert_db_azure: [POST] http://localhost:7071/api/insert_db_azure

For detailed output, run func with --verbose flag.

```

Figura 16 – App Function

Resultado do endpoint insert_db_azure, que insere na base de dados, podemos ver que foi um sucesso:

SELECT * FROM raw.SALES | Enter a SQL expression to filter results (use Ctrl+S)

123 id	123 product_id	sales_ts	123 user_id
1	11	-01 00:00:00.000	4

Figura 17 – BD App Function

Podemos ver o resultado do endpoint insert_blobfile_azure, que insere no container container04 o ficheiro:

Home > storageaccountt04 | Containers >

container04 Container

Search Upload Change access level Refresh Delete Change tier Acquire lease Break lease

Overview Diagnose and solve problems Access Control (IAM) Settings

Authentication method: Access key (Switch to Microsoft Entra user account)
Location: container04

Search blobs by prefix (case-sensitive) Show deleted blobs

Add filter

Name	Modified	Access tier	Archive status	Blob ty
Files_2d54aea8-6942-47ee-ab43-228987...	3/10/2025, 3:03:04 PM	Hot (Inferred)		Block b

Figura 18 - Blob Storage - Azure Functions

→ **Github:** Ver na pasta \Exercicios\Exercise_12\
 \1.app_funtion_azurite_Insert_BlobFile&BD_AZURE

2. Relativamente aos ficheiros Dockerfile e app.py, a sua estrutura e funcionalidade são muito semelhantes ao que foi abordado anteriormente. Contudo, há uma modificação importante na variável URL, que deve ser ajustada conforme o ambiente ou destino onde o serviço App Function estará a ser executado. Essa configuração é essencial para garantir o acesso ao serviço na máquina host, permitindo a comunicação com a base de dados a partir do container Docker. Esta alteração demonstra a flexibilidade necessária para adaptar o serviço às diferentes infraestruturas onde será implementado.

→ **Github:** Ver na pasta \Exercicios\Exercise_12\ 2.Docker_ETL_API_Flask

Exercise 13. - Azure Queues

13.1. Create two Azure Storage Queues:

- One will receive the data to be inserted in the db the other will receive the data to be inserted into files.

13.2. Change the API to only have one endpoint that will send the data to the two created Azure Storage Queues.

1. Para criar duas Azure Storage Queues, deve-se começar por selecionar a conta de armazenamento apropriada, que neste caso é a storageaccountt04. Em seguida, acede-se à opção **Queues** disponível no menu lateral da conta de armazenamento. Após isso, procede-se à criação de cada fila, atribuindo nomes específicos, como bdqueue-sales para a primeira fila e filequeue-sales para a segunda. As filas criadas serão exibidas na página de Queues, juntamente com os respetivos URLs de acesso, e estarão prontas para receber mensagens da aplicação.

2. No ficheiro function_app.py podemos verificar o código que cria uma **API serverless** utilizando o Azure Functions e permite o envio de dados para duas filas (queues) no **Azure Queue Storage**. A API é projetada para receber informações de vendas (como product, sales_ts e user_id) via requisições POST e distribui esses dados para duas Queues diferentes, chamadas bdqueue-sales e filequeue-sales.

Cada Queue destina-se a diferentes processos (uma para banco de dados e outra para arquivo). Valida os dados recebidos, adiciona um identificador único e envia a mensagem para as filas configuradas, garantindo que estejam criadas caso ainda não existam. O código faz logs de sucesso e tratamento de erros para monitorar e depurar o processo. É uma solução escalável para distribuição de dados em sistemas desacoplados.

```
PS C:\Estudos\Programação\UpSkill\05_Modulo_Azure\Projeto_Avaliacao\Exercicios\Exercise_13> func start
Found Python version 3.11.0 (py).

Azure Functions Core Tools
Core Tools Version:      4.0.6821 Commit hash: N/A +c09a2033faa7ecf51b3773308283af0ca9a99f83 (64-bit)
Function Runtime Version: 4.1036.1.23224

[2025-03-10T16:17:30.235Z] Worker process started and initialized.

Functions:

    insert_into_queues_azure: [POST] http://localhost:7071/api/insert_into_queues_azure

For detailed output, run func with --verbose flag.
```

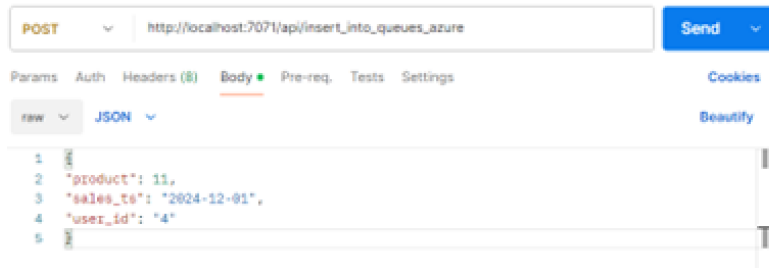


Figura 19 - Azure Queues

Utilizando o comando Func Start, iniciamos a execução da função no Azure Functions, tornando os endpoints definidos na aplicação acessíveis. Após enviar as mensagens através do Postman no formato adequado, conseguimos verificar que essas mensagens foram enviadas com sucesso para as duas Queues configuradas no Azure Storage Account, chamadas bdqueuesales e filequeuesales. Este fluxo assegura que os dados estão devidamente distribuídos para ambas as Queues para posterior processamento.

→ **Github:** Ver na pasta \Exercicios\Exercise_13\

Exercise 14. App Functions Queue Trigger

Create two App Functions Queue Trigger:

a. One will consume the data from the db queue and insert into the db;

b. The other will consume the data from the files queue and insert into files and upload to the Blob Storage.

a. O código no ficheiro function_app.py implementa uma Queue Trigger no Azure Functions, que é ativada sempre que uma mensagem é adicionada à fila denominada bdqueuesales no Azure Queue Storage. Após ser acionada, a função decodifica a mensagem da fila, convertendo os dados JSON recebidos. Em seguida, utiliza o driver ODBC para estabelecer uma conexão com um banco de dados SQL hospedado no Azure,

onde valida se os campos obrigatórios (product, sales_ts e user_id) estão presentes. Após validação, os dados decodificados são inseridos na tabela raw.sales dentro do esquema RAW. O processo inclui também logs detalhados para monitorar e rastrear possíveis falhas, seja na conexão com o banco de dados ou na inserção das informações. O objetivo principal deste código é garantir o processamento automatizado e confiável das mensagens da fila, armazenando as informações na base de dados de forma eficiente e escalável.

→ **Github:** Ver na pasta \Exercicios\Exercise_14\
|1.app_funtion_azurite_Insert_QUEUEs_BD

b. Este código implementa uma Queue Trigger no Azure Functions que é acionada sempre que uma mensagem é inserida na fila chamada filequeue-sales no Azure Queue Storage. Assim que o Trigger é ativado, o código processa a mensagem da fila(queue), que contém dados em formato JSON, e realiza o upload desses dados para o Azure Blob Storage no container chamado container04. O objetivo é transformar os dados recebidos em arquivos JSON e armazená-los de forma organizada na nuvem. A função começa por decodificar a mensagem recebida da fila e convertê-la de texto para um dicionário Python utilizando o módulo json. Para identificar cada arquivo de forma única, é gerado um identificador (UUID). A mensagem decodificada é então gravada em um arquivo JSON temporário, cujo nome é formatado com o UUID gerado. Depois de criado o arquivo JSON, a função realiza o upload do arquivo para o container especificado no Azure Blob Storage. Para isso, utiliza a conexão definida pela string de conexão do Azure Storage. Caso o processo de gravação ou upload falhe, o código inclui tratamento de erros para registrar o problema através de logs, garantindo que possíveis falhas sejam monitoradas. O código combina a capacidade de filas para comunicação assíncrona e o armazenamento em nuvem para persistência e acessibilidade dos dados. É projetado para ser uma solução escalável e confiável, ideal para armazenar e processar mensagens que necessitam de persistência no formato de arquivos no Azure.

→ **Github:** Ver na pasta \Exercicios\Exercise_14\
|2.app_funtion_azurite_Insert_QUEUEs_FILE

Exercise 15. - Azure Tables

Create one Azure Storage Table:

- Create a schema to insert logs into this table. The schema must have at least the timestamp and a message column
- Change the API code to send logging info to this table
- Create additional columns in the table to have more detailed info

O ficheiro `function_app.py` tem código que cria uma API baseada no Azure Functions para registar entradas de log no Azure Table Storage. Ele é configurado para aceitar requisições POST, onde recebe dados de log em formato JSON. O código usa a string de conexão do Azure Storage para se conectar à tabela `ApplicationLogs` e garante que a tabela exista antes de inserir os dados. A função principal é acedida pela rota `insert_log`, processa a mensagem recebida, valida o campo obrigatório `message` e gera um timestamp (se não fornecido). Em seguida, compila um objeto de log com um identificador único (RowKey) e campos adicionais (como `product`, `sales_ts` e `user_id`) provenientes da requisição. Este objeto é então inserido na tabela utilizando o cliente da Tabela do Azure.

O código também inclui tratamento de erros e logs para monitorizar a operação e facilitar a depuração. O principal objetivo do código é capturar, estruturar e armazenar entradas de log.

→ **Github:** Ver na pasta `\Exercicios\Exercise_14\`

Exercise 16. - Azure Data Factory

Create a new Azure Data Factory:

- a. Read data from the raw db with the queries from the exercise 1;
- b. Insert the data into the report db.

a. O Azure Data Factory é uma ferramenta poderosa para orquestrar e automatizar fluxos de trabalho de dados em escala empresarial.

Para criar um Azure Data Factory no Portal do Azure temos de procurar pelo Data Factory e criar um novo, primeiramente temos de escolher uma assinatura do Azure, grupo de recurso, inserir um nome único para o Data Factory, escolher uma região onde o Data Factory irá ser hospedado, escolher a versão, e escolher um endpoint.

Depois de criarmos o Azure Data Factory, no nosso caso é o `04datafactory`, clicamos em “launch studio” e podemos começar a criar pipelines de dados, configurar conjuntos de dados, integrar serviços e monitorar suas atividades.

b. Primeiramente, é necessário criar as cinco tabelas na base de dados db-04 no Azure, especificamente no esquema Report, com todos os campos necessários. As tabelas a serem criadas são: ReportAverageSpentPurchases, ReportBestRatedProductsDataset, ReportSoldQuantityByProduct, ReportTopSpenders e ReportTopUsersAverageReviews, conforme ilustrado na imagem."

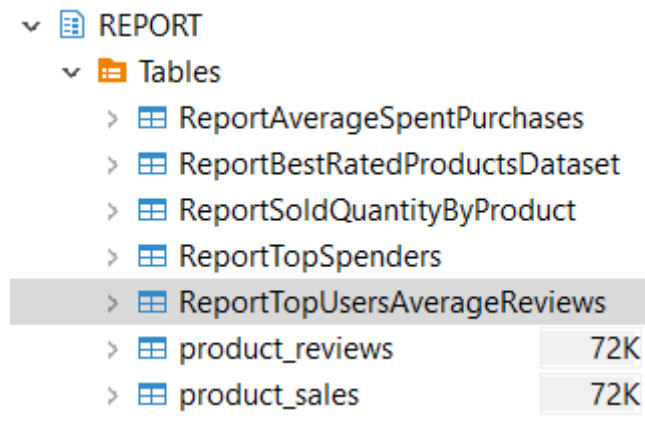


Figura 20 – REPORT DBs

Em seguida, é necessário estabelecer a ligação e configurar as cinco tabelas no Azure Data Factory, juntamente com a tabela **Raw.sales**, e testar a conexão. Para isso, no separador *Datasets*, foram criadas as seis conexões necessárias.

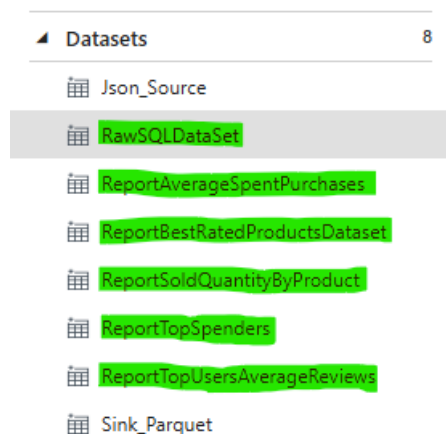


Figura 21 – BDs - Azure Data Factory

Após configurar e testar a ligação às seis tabelas, acedemos ao separador Pipelines e criamos um pipeline com o nome ETL_AnalyticalQueries. Depois de criado, abrimos o pipeline e, no separador Move and Transform, adicionamos cinco atividades do tipo Copy Data. Renomeamos cada uma delas para nomes mais apropriados: CopyTopSpenders, CopyBestRatedProducts, CopySoldQuantityByProduct, CopyAverageSpentPurchases e CopyTopUsersAverageReviews.

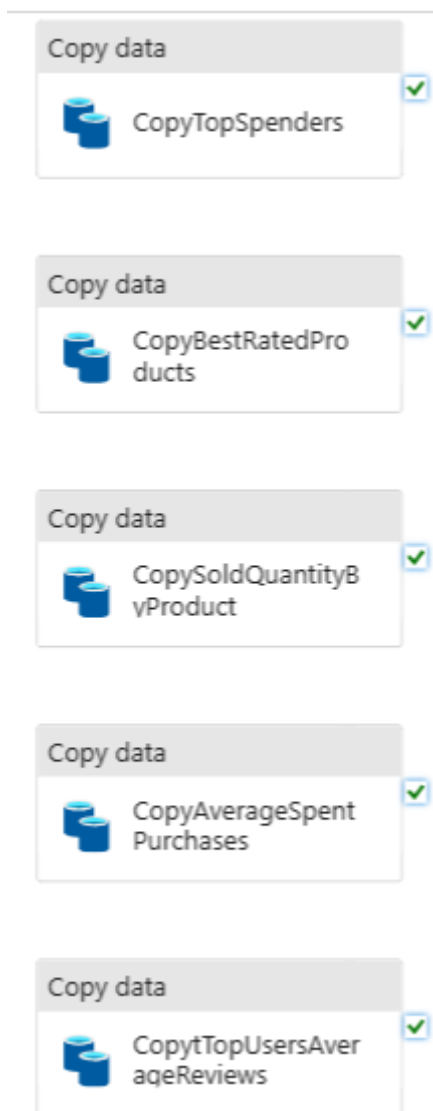


Figura 22 – Pipelines - Azure Data Factory

Passo agora a exemplificar a configuração do **CopyBestRatedProducts**, sendo o processo semelhante para as outras atividades, alterando apenas a *query* e a tabela de destino da cópia.

No separador General, é necessário atribuir um nome à atividade. Neste caso, optei por CopyBestRatedProducts.

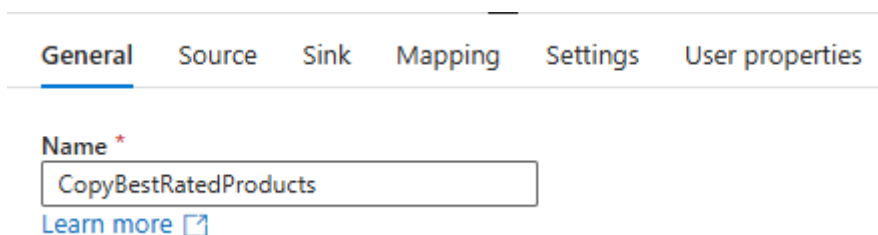


Figura 23 – CopyBestRatedProducts - Azure Data Factory

No separador Source, devemos selecionar o dataset de origem dos dados, definir o tipo de query que desejamos executar—neste caso, uma query—e introduzir o código correspondente. Os restantes parâmetros podem ser mantidos com as definições padrão.

General **Source** Sink Mapping Settings User properties

Source dataset *

RawSQLDataSet

Open + New Preview data Learn more

Use query

☐ Table ☒ Query ☐ Stored procedure

Query *

SELECT p.name AS product_name,
AVG(r.rating) AS avaliacao_media
FROM raw.reviews r

Edit

Query timeout (minutes) ⓘ

120

Isolation level ⓘ

Select...

Partition option ⓘ

☒ None ☐ Physical partitions of table

Figura 24 – Source - Azure Data Factory

No separador Sink, é necessário selecionar o dataset de destino dos dados. A base de dados varia conforme a atividade escolhida, uma vez que cada atividade possui uma tabela de destino distinta. Os restantes parâmetros podem ser mantidos com as definições padrão.

General Source **Sink** Mapping Settings User properties

Sink dataset *

ReportBestRatedProductsDataset

Open + New Learn more

Write behavior

☒ Insert ☐ Upsert ☐ Stored procedure

Bulk insert table lock ⓘ

☐ Yes ☒ No

Table option

☒ Use existing ☐ Auto create table ⓘ

Figura 25 – Sink - Azure Data Factory

Para o nosso caso de estudo, não há necessidade de realizar alterações nos restantes separadores. Assim, avançamos diretamente para a execução da atividade. Para isso, selecionamos a atividade desejada e clicamos no botão **“Add Trigger – Trigger Now”**, que é responsável por iniciar o processo. Após a ativação, aguardamos um momento e confirmamos clicando em **OK**. No canto superior esquerdo, será exibido o progresso da execução da pipeline, como ilustrado na imagem abaixo.

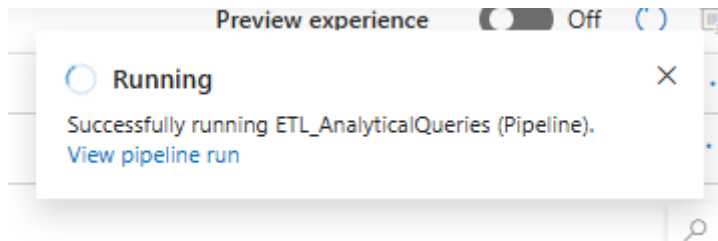


Figura 26 – Running - Azure Data Factory

Após executarmos o comando `SELECT * FROM REPORT.ReportBestRatedProductsDataset`, confirmamos que os dados foram transferidos corretamente da consulta realizada na base de dados **Raw** para a tabela **REPORT.ReportBestRatedProductsDataset**.

Exercise 17. - Azure Blob Storage

Create two Endpoints in the App Function to receive POST Requests:

- a. One endpoint inserts into the DB;*
- b. Another endpoint uploads files to the Blob Storage.*

O enunciado deste exercício é igual ao do exercício 12.

Exercise 18. Parquet Files - Data Factory

Create a pipeline to transform data from Azure Blobs json files into parquet and upload to another container and run it on Azure Data Factory.

O formato **Parquet** é uma tecnologia amplamente utilizada para armazenamento de dados, especialmente em ambientes de grande escala. Ele organiza os dados em colunas, em vez de linhas, otimizando o desempenho para tarefas analíticas e reduzindo o uso de espaço por meio de compressão eficiente. Essa característica permite o acesso rápido às informações relevantes sem a necessidade de ler o conjunto de dados completo.

Para resolver este exercício, o primeiro passo foi criar um novo container chamado "parquet" na Storage account "storageaccountt04", destinado a armazenar os dados provenientes do container "container04", que contém arquivos em formato JSON.

Em seguida, no separador Datasets, foram adicionadas duas novas bases de dados: Json_Source e Sink_Parquet.

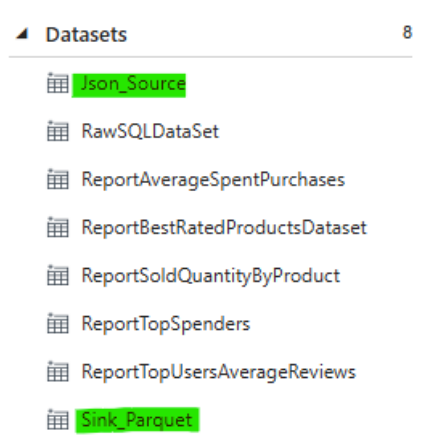


Figura 27 – Datasets - Azure Data Factory

Relativamente ao Json_Source, a imagem abaixo ilustra como foi realizada a sua configuração. Esta fonte de dados, em formato JSON, utiliza o serviço vinculado AzureBlobStorage, que está direcionado para o diretório container04. O arquivo não está comprimido e utiliza a codificação padrão UTF-8. Essas configurações são essenciais para garantir o processamento eficiente de arquivos JSON no pipeline.

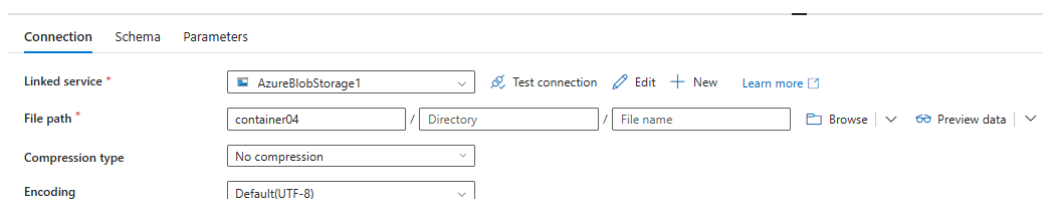


Figura 28 – JSON_Source - Azure Data Factory

Relativamente ao **Sink_Parquet**, a imagem abaixo demonstra como foi configurado. Esta estrutura recebe os dados processados e os armazena no **Azure Blob Storage**, utilizando o diretório especificado como **parquet**. O tipo de compressão selecionado é **snappy**, o que melhora a eficiência de armazenamento e leitura dos dados. Essas definições garantem que os dados sejam escritos corretamente no formato **Parquet** dentro do container designado.



Figura 29 – Parquet - Azure Data Factory

seguida, foi criada a pipeline denominada “JSON_to_Parquet_Pipeline” no separador Pipelines. Foram configurados três separadores principais, enquanto os restantes foram mantidos com as definições padrão.

Separador General: Definiu-se o nome da pipeline como JSON_to_Parquet_Pipeline.

Separador Source: Especificou-se a base de dados que forneceria os dados, neste caso, a Json_Source, indicando que os dados seriam obtidos do diretório container04.

Figura 30 – JSON_Source_Parquet - Azure Data Factory

Separador Sink: Configurou-se o local onde os dados seriam armazenados.

Figura 31 – Sink_Parquet - Azure Data Factory

Após a conclusão destas etapas, podemos confirmar o correto funcionamento da pipeline no Azure Data Factory. Para realizar o teste, clicamos na pipeline, selecionamos Add Trigger – Trigger Now e, em seguida, clicamos em OK. Dessa forma, verificamos que os arquivos presentes no container container04 foram copiados com sucesso para o container parquet.

→ Pode verificar a realização do exercício no Portal do Azure.

Exercise 19. - Homework - Queue Triggers

Create two consumers for the same Azure Storage Queue and check if both can read messages from the queue.

Este exercício é similar a outros exercícios realizados anteriormente, pelo que só irei abordar os pontos mais importantes do exercício.

Podemos verificar que o código da function_app.py possui duas Queue Triggers no Azure Functions. Ambas processam mensagens da fila filequeue-sales, decodificam os dados em JSON, geram arquivos localmente e enviam-nos para o Azure Blob Storage. A primeira função carrega os dados no container container04, enquanto a segunda está configurada para um segundo container. Ele automatiza a movimentação e armazenamento de dados de forma eficiente.

Respondendo à questão inicial, pode verificar que, sim, é possível ter dois consumidores a processar mensagens da mesma fila no Azure Queue Storage. No entanto, estes consumidores não irão processar as mesmas mensagens, uma vez que as mensagens da Queue só podem ser lidas uma única vez.

Quando um consumidor lê uma mensagem, essa mensagem é marcada como "invisível" para os outros consumidores por um determinado período de tempo, onde durante esse tempo, o consumidor deve processar a mensagem e depois de processada deve eliminá-la da fila. Caso o processamento da mensagem falhar ou não for concluído, a mensagem será visível novamente para outros consumidores após o término do timeout.

Posso concluir que ambos os consumidores podem ler mensagens, porém cada mensagem será apenas atribuída a um consumidor cada vez, e assim garante que não existe duplicação de processamento e assim podemos permitir a escalabilidade e a distribuição eficiente da carga de trabalho entre múltiplos consumidores.

→ **Github:** Ver na pasta \Exercicios\Exercise_19\

Exercise 20. - Event Hub

20.1. Create an Event Hub endpoint and sent the Sales data to the new hub from the api;

20.2. Change the Docker that creates events and sends to the api, to send 10000 events each time with a interval of 1 second for each 10 events.

20.1. O Event Hub do Azure é um serviço de processamento de dados em tempo real, voltado para capturar, armazenar e distribuir grandes volumes de eventos provenientes de diferentes fontes. Ele é amplamente utilizado para aplicações como telemetria, registo de logs e processamento de dados de forma contínua, conectando sistemas e promovendo análises rápidas e eficientes.

Criar um Event Hub no Azure é um processo simples e segue um fluxo semelhante ao de outros recursos na plataforma. O nome do nosso Event Hub, neste caso, é **04sales-events_teste**. Na aba “**Data Explorer**”, é possível visualizar, em tempo real, os dados que estão sendo recebidos pelo Event Hub, bem como os dados armazenados anteriormente.

Para enviar dados da API para o Event Hub, o processo também é semelhante ao de outras ferramentas do Azure. É necessário configurar a **connection string** e o nome do Event Hub na API. Em seguida, utilizamos uma **Azure Function**, que recebe os dados via HTTP, realiza a validação das informações e envia eventos devidamente formatados para o Azure Event Hub.

→ **Github:** Ver na pasta \Exercicios\Exercise_20\ \20.1\ \2.app_funtion_azurite_Insert_EventHub

20.2. Neste exercício foi necessário alterar a função de gerar_sales_aleatorias(connection, data_inicial_str, data_final_str), onde esta passou a cria eventos e envia para a API, para enviar 10.000 eventos de cada vez com um intervalo de 1 segundo por cada 10 eventos.

→ **Github:** Ver na pasta \Exercicios\Exercise_2020.2\ \1.Docker_ETL_API_Flask_Gerador_de_dados o ficheiro “app.py”.

Exercise 21. - Stream Analytics

21.1. Create an Azure Stream Analytics (ASA) Job that consumes data from the Event Hub and counts the number of sales by user id, on each minute, and outputs the result to parquet files;

21.2. Create another Stream Analytics that aggregates 100 rows and creates a parquet file with 100 rows.

O Azure Stream Analytics é uma ferramenta de processamento de dados em tempo real, desenvolvida para analisar e transformar informações provenientes de diversas fontes, como Event Hubs, IoT Hubs e Blob Storage. Este serviço permite criar consultas personalizadas, identificar padrões, detectar anomalias e extrair insights de maneira rápida e integrada ao ecossistema Azure, com alta escalabilidade.

Para este exercício, foi criada a Stream Analytics **04StreamAnalytics**, que possui como entrada (input) o **eventhub-sales-04**, configurado para receber dados do Event Hub criado previamente. Como saída (output), foi configurado o **stream-event-hub-04**, que grava dados em um container chamado "Parquet", localizado em uma Storage Account. Durante a configuração do output, foram definidos parâmetros como a conta de armazenamento (Storage Account), o container, o formato do arquivo (neste caso, **Parquet**) e o intervalo de execução da tarefa, configurado para um minuto. Para visualizar arquivos no formato Parquet, é possível usar ferramentas como o **parquetreader.com**.

Após configurar o input e o output, foi necessário aceder à aba **“Query”** e inserir a seguinte consulta SQL:

```
SELECT *  
  
INTO  
  
[stream-event-hub-04]  
  
FROM  
  
[eventhub-sales-04];
```

Este código recupera todos os dados da entrada denominada **[eventhub-sales-04]**, que se refere a um Event Hub, e direciona-os para a saída designada como **[stream-event-hub-04]**, que corresponde a um Blob Storage. Após inserir a query, salvamos e testamos. Caso tudo funcione corretamente, podemos iniciar o Job utilizando a opção **“Start Job”**. Com o Job ativo, os dados recebidos pelo Event Hub serão processados pelo Stream Analytics e, em seguida, enviados para um Blob Storage que armazena os arquivos no formato Parquet. Abaixo, podemos visualizar exemplos de arquivos gerados pelo Stream Analytics.

Name	Modified	Access tier	Archive status	Blob type	Size	Lease state
711096815_03a5195a0e49f8532f1b67c0e742_1.parquet	3/15/2025, 4:10:24 AM	Hot (inferred)		Block blob	3.12 KiB	Available
711096815_05a80b779e8b4d3f80bce05c34b83f6c_1.parquet	3/15/2025, 3:42:04 AM	Hot (inferred)		Block blob	1.97 KiB	Available
711096815_18405023a3854e44a4d6d6b7ade974f_1.parquet	3/15/2025, 3:32:48 AM	Hot (inferred)		Block blob	2.08 KiB	Available
711096815_1e5686d39de4b51a027e509aceabf9_1.parquet	3/15/2025, 4:41:11 AM	Hot (inferred)		Block blob	5.47 KiB	Available
711096815_2002d62394b4c15ac3cb7919fce7ade_1.parquet	3/15/2025, 4:44:17 AM	Hot (inferred)		Block blob	5.85 KiB	Available
711096815_2b0f6d1949fa299cc616c9eebb429_1.parquet	3/15/2025, 4:43:15 AM	Hot (inferred)		Block blob	5.6 KiB	Available
711096815_3b02d038d204668d516c5a11653b0d0_1.parquet	3/15/2025, 3:27:33 AM	Hot (inferred)		Block blob	1.97 KiB	Available
711096815_3b57152632a42f1a8cac7da8519ac0_1.parquet	3/15/2025, 4:33:00 AM	Hot (inferred)		Block blob	2.48 KiB	Available
711096815_52c25513ba3c42f4c3d6f0ec94ead_1.parquet	3/15/2025, 4:45:17 AM	Hot (inferred)		Block blob	5.72 KiB	Available
711096815_57d1917a71de4f6f8f9baf12686415d_1.parquet	3/15/2025, 3:51:48 AM	Hot (inferred)		Block blob	2.31 KiB	Available
711096815_5b634690b78043e7918e6af9f84b1f_1.parquet	3/15/2025, 4:53:28 AM	Hot (inferred)		Block blob	6.34 KiB	Available
711096815_69f05cb41d75f388eedcb37b0640dc_1.parquet	3/15/2025, 4:09:22 AM	Hot (inferred)		Block blob	2.61 KiB	Available
711096815_742f614b3d6a035914c5802f4973531_1.parquet	3/15/2025, 3:38:59 AM	Hot (inferred)		Block blob	1.97 KiB	Available
711096815_80fb096f0e0a4c8f5d3e873669d0b0_1.parquet	3/15/2025, 4:42:12 AM	Hot (inferred)		Block blob	5.6 KiB	Available
711096815_81b7577a48b964a3ac14218f10cdecde_1.parquet	3/15/2025, 4:34:01 AM	Hot (inferred)		Block blob	3.99 KiB	Available
711096815_83ba0a4391ae4070a34f68f4020429531_1.parquet	3/15/2025, 4:51:26 AM	Hot (inferred)		Block blob	6.34 KiB	Available
711096815_8a9e686cd54302bc57f323b12441a1_1.parquet	3/15/2025, 4:30:56 AM	Hot (inferred)		Block blob	1.97 KiB	Available
711096815_8c2ed800df94c999aed491375c6077_1.parquet	3/15/2025, 4:48:21 AM	Hot (inferred)		Block blob	6.34 KiB	Available

Figura 32 – Parquet - Stream Analytics

Exercise 22. Homework - Kafka

Create two Kafka consumers to replace the App functions Queue Trigger.

O Kafka é uma plataforma de streaming de dados distribuída desenvolvida pela Apache. Ele é utilizado para publicar, consumir, armazenar e processar grandes volumes de dados em tempo real. Projetado para alta performance e escalabilidade, o Kafka é amplamente empregado em sistemas que exigem troca contínua de informações, como monitoramento, sistemas financeiros e análises em tempo real. Ele organiza os dados em "tópicos", que funcionam como canais para os quais produtores enviam mensagens e consumidores as leem.

O primeiro passo para a realização do exercício é executar o arquivo docker-compose.yaml, que configura um broker Kafka utilizando a imagem mais recente. Esse código define portas, variáveis de ambiente e parâmetros essenciais para um ambiente standalone (usando KRaft), especificando identificadores, listeners, replicação mínima, papéis e o diretório de logs.

No segundo passo, iniciam-se ambos os dois consumidores. É importante notar que cada consumidor pertence a um consumer group diferente, pois, caso estivessem no mesmo consumer group, apenas um deles consumiria as mensagens. Os dois consumidores diferem na seguinte forma: o grupo b-group grava as mensagens no arquivo file1, enquanto o grupo a-group escreve no arquivo file2. Ambos consomem mensagens de um tópico Kafka e as salvam em arquivos comprimidos no formato Gzip, criando um novo arquivo a cada 100 mensagens. Essa configuração permite observar o funcionamento distinto de ambos os consumidores e verificar que eles leem as mesmas mensagens (podemos verificar que geraram exatamente os mesmos ficheiros), algo que não ocorre em exercícios utilizando Queues.

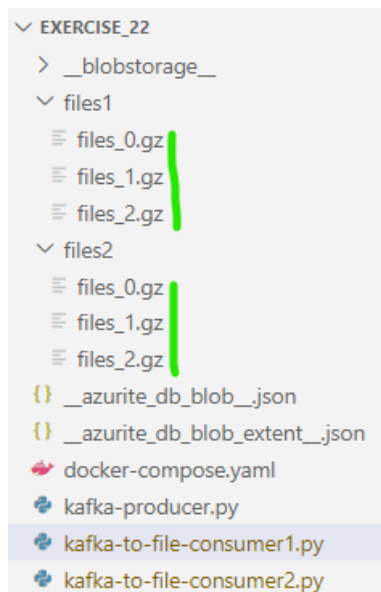


Figura 33 – Files - Kafka

O terceiro passo consiste em produzir os dados, ou seja, executar o produtor. Este desempenha duas funções principais: primeiro, verifica se o tópico test-topic existe e, caso não exista, cria-o com duas partições e um fator de replicação de 1. A segunda função é gerar mensagens, onde são criados 10.000 eventos JSON contendo timestamp, user_id, nome de produto aleatório e classificação. Essas mensagens são enviadas para o tópico test-topic através de um KafkaProducer.

Por fim, é importante referir que foram necessários quatro terminais em execução simultânea, um para cada um dos arquivos.

→ **Github:** Ver na pasta \Exercicios\Exercise_22\ Ver ficheiros “docker-compose.yaml”, “kafka-producer”, “kafka-to-file-consumer1” e “kafka-to-file-consumer2”.

Exercise 23. Homework - Cosmos DB

Create a Cosmos DB table to insert number of sales by user.

O **Cosmos DB** do Azure é um banco de dados distribuído globalmente, projetado para oferecer alta disponibilidade e baixa latência. Ele suporta diferentes modelos de dados, como documentos, chaves-valor, grafos e colunas, sendo ideal para aplicações que exigem escalabilidade e performance em tempo real, integrando-se facilmente ao ecossistema Azure.

Para que o Cosmos DB possa receber dados, foi necessário criar um output direcionado para ele na Stream Analytics. As configurações seguem uma lógica semelhante às realizadas anteriormente.

A criação de uma conta no Cosmos DB é bastante similar ao processo de criação de outros recursos no Azure. Para este caso específico, optamos pela configuração **DB for NoSQL**. Após a criação do Cosmos DB denominado **“04cosmos”**, foi necessário configurar um **container**. Na imagem abaixo, é possível visualizar as definições utilizadas para este container.

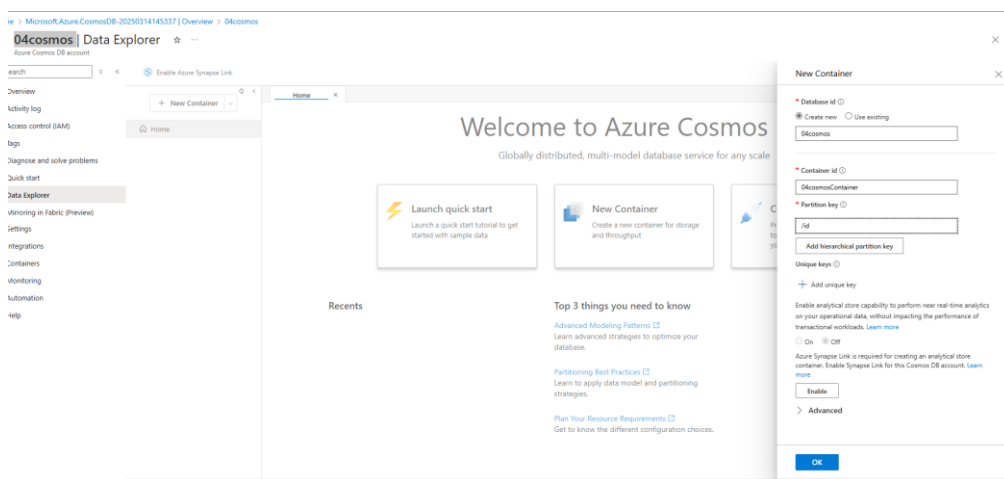


Figura 34 – Container - Cosmos DB

Após a Stream Analytics processar os dados recebidos do Event Hub e encaminhá-los para o Cosmos DB 04cosmos, é possível visualizar alguns dos eventos registrados a partir do container como se pode verificar na imagem abaixo.

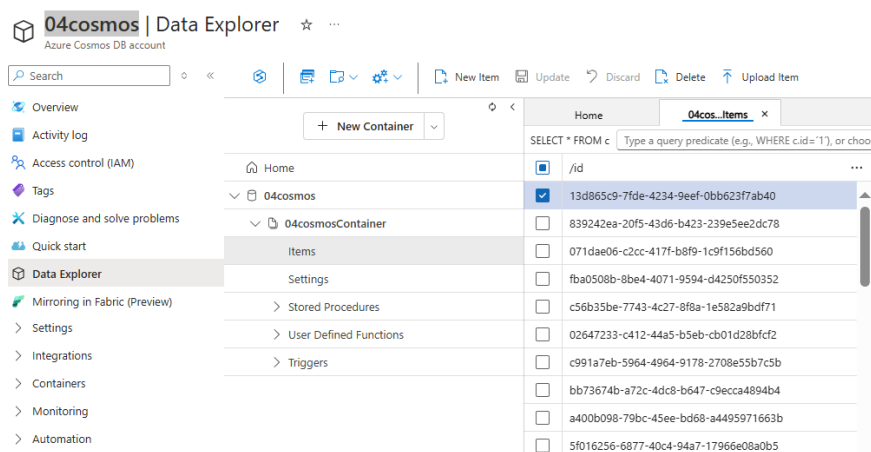


Figura 35 – Registos - Cosmos DB

Exercise 24 - Log Analytics

O Log Analytics do Azure é uma ferramenta utilizada para coletar, analisar e monitorar dados de desempenho e diagnósticos de diferentes recursos e serviços no Azure. Ele permite centralizar logs e métricas em um único local, facilitando a identificação de problemas, análise de tendências e criação de alertas personalizados para manter a eficiência e a segurança das aplicações.

O primeiro passo é acessar o Portal do Azure e criar um Log Analytics Workspace, que, neste caso, foi denominado de 04LogAnalytics. Dentro do workspace 04LogAnalytics, navegamos até a aba “Diagnostic Setting”, onde é possível adicionar um diagnóstico conforme a necessidade. Na imagem abaixo, é possível visualizar essa configuração.

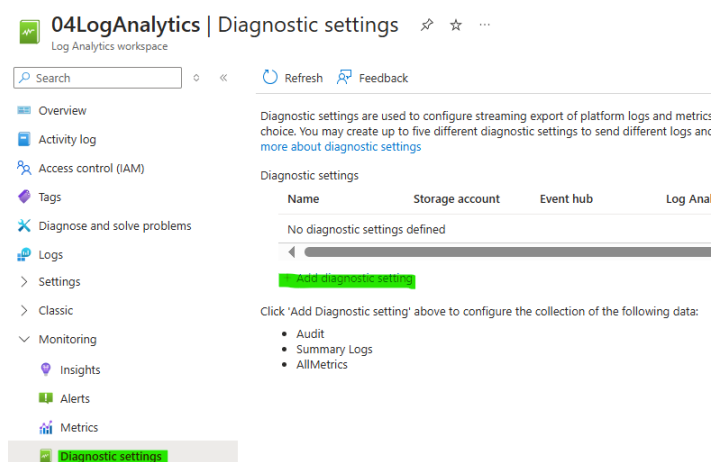


Figura 36 – Diagnostic setting - Log Analytics

É possível personalizar as configurações do diagnóstico conforme nossas preferências, ajustando o nível de detalhamento necessário. No nosso caso específico, optamos por selecionar todas as opções disponíveis para obter o máximo de informações. Na imagem abaixo, estão apresentadas as configurações aplicadas.

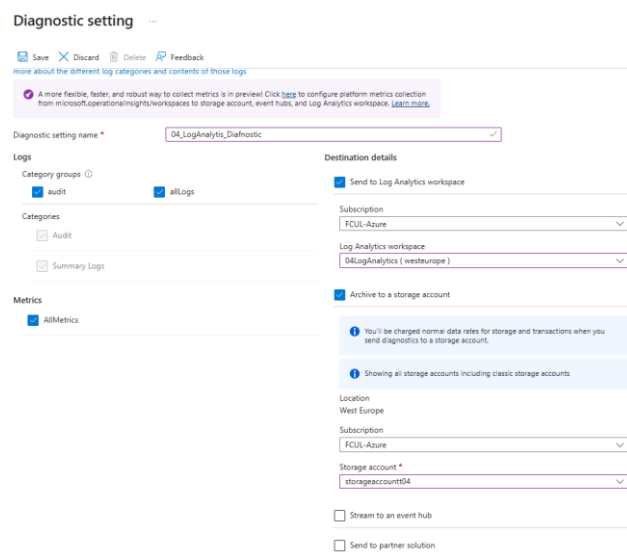


Figura 37 – Diagnostic setting config - Log Analytics

Arquitetura em Azure – Use Case

Após a conclusão de todos os exercícios, o professor Miguel Costa propôs nos um desafio: Criar nossa própria arquitetura no Azure e analisar o fluxo dos dados dentro dela.

A arquitetura que desenvolvi é apresentada na figura abaixo:

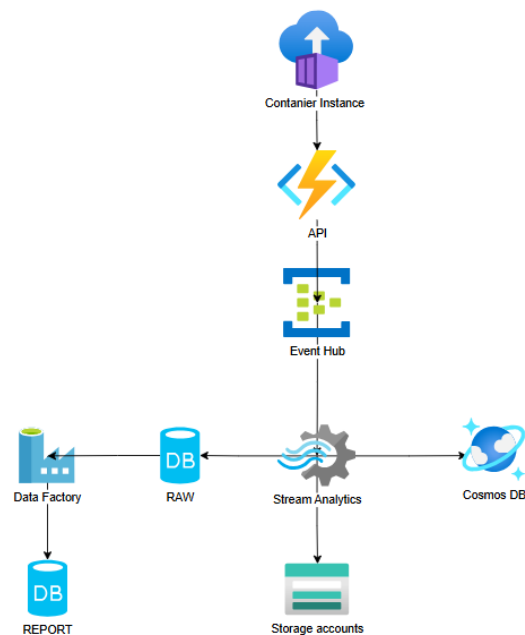


Figura 38 – Arquitetura Azure

A arquitetura representa um sistema desenvolvido no Azure, integrando diferentes serviços para oferecer uma solução completa de processamento e armazenamento de dados. Vamos realizar uma está a explicação detalhada de cada componente e do fluxo de dados.

Relativamente aos **Componentes** da arquitetura e as suas **Funções**:

- **Container Instance:** Este componente serve como o ponto inicial do sistema, funcionando como um gerador de dados. Ele simula um website de uma empresa que oferece seus produtos para venda online, permitindo que os clientes realizem suas compras. A instância executa uma aplicação containerizada, que processa e organiza os dados gerados pelas interações dos clientes e, em seguida, envia essas informações para a API para continuidade no fluxo do sistema;
- **API:** Funciona como uma interface intermediária que recebe os dados gerados pela Container Instance e os encaminha para o próximo serviço, neste caso, o Event Hub. A API garante que os dados estejam formatados corretamente para o envio;
- **Event Hub:** É um serviço de ingestão de dados em tempo real, ideal para lidar com grandes volumes de eventos. Ele recebe os dados enviados pela API e funciona

como um ponto de coleta para posterior processamento em outras partes do sistema;

- **Stream Analytics:** Realiza o processamento de dados em tempo real. A partir dos dados recebidos do Event Hub, este serviço aplica análises e transformações para preparar os dados para os destinos finais;
- **Cosmos DB:** Um banco de dados escalável e distribuído, utilizado para armazenar dados estruturados que precisam ser acedidos rapidamente. Os dados processados pelo Stream Analytics são armazenados aqui para aplicações que requerem baixa latência;
- **Storage Account:** Serve como repositório de armazenamento flexível, usado para guardar dados em diferentes formatos. No contexto desta arquitetura, pode ser utilizado para arquivar dados brutos ou processados;
- **RAW (Banco de Dados Bruto):** É um local específico para armazenar dados brutos antes de serem processados. A partir daqui o Data Factory pode aceder aos dados para transformações adicionais;
- **Data Factory:** É responsável por orquestrar e automatizar os processos de movimentação e transformação de dados. Ele conecta os dados armazenados no RAW ao destino final para relatórios;
- **REPORT (Banco de Dados para Relatórios):** É o destino final para os dados já transformados. Serve para armazenar informações que serão utilizadas em análises, relatórios ou visualizações.

Relativamente ao **Fluxo dos dados:**

- **Geração de Dados:** O processo começa na Container Instance, onde os dados são gerados e/ou pré-processados para posteriormente ser enviado para a API;
- **Ingestão pelo Event Hub:** A API encaminha os dados para o Event Hub, e este age como um ponto de entrada e distribuição para outros serviços;
- **Processamento em Tempo Real:** O Stream Analytics recebe os dados do Event Hub, aplica análises e transforma as informações;
- **Armazenamento:** Após o processamento no Stream Analytics, os dados são encaminhados para diferentes destinos:
 - **Cosmos DB:** Para armazenamento estruturado;
 - **Storage Account:** Para armazenar dados brutos ou processados;
 - **RAW DB:** Para arquivar os dados brutos, que serão processados posteriormente.
- **Transformação e Relatórios:** O Data Factory acessa os dados armazenados no RAW, realiza transformações e os transfere para o REPORT, onde ficam prontos para visualizações e análises detalhadas.

Relativamente à importância da arquitetura apresentada, esta arquitetura destaca como os serviços do Azure podem ser integrados para fornecer uma solução robusta e escalável. Ela permite processar dados em tempo real, armazená-los em diferentes formatos e utilizá-los para análises e relatórios. A flexibilidade e a integração de ferramentas como Event Hub, Cosmos DB e Stream Analytics tornam-na ideal para aplicações que exigem alta disponibilidade e processamento eficiente de grandes volumes de informações.

Vantagens arquitetura Event Hub Vs arquitetura Queue

Ao escolher uma arquitetura, é essencial realizar uma análise cuidadosa. Assim, vamos explorar algumas das vantagens de optar por uma ou por outra arquitetura. A comparar entre as diferentes arquiteturas é a melhor abordagem para identificar a solução mais adequada às nossas necessidades.

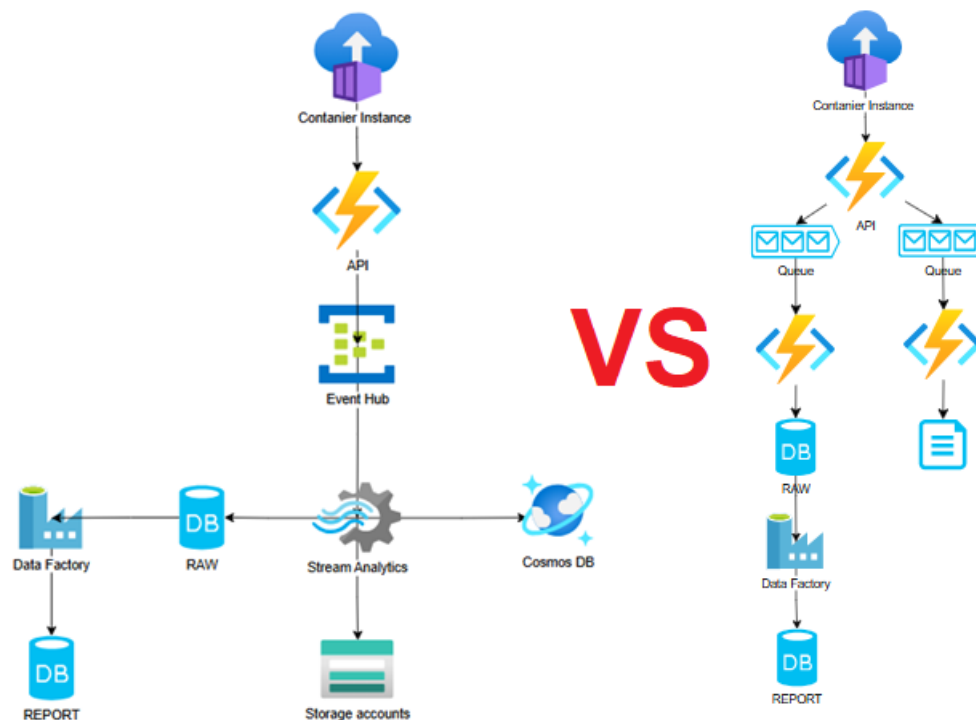


Figura 39 – Arquitetura Event Hub vs Arquitetura Queues Azure

As duas arquiteturas apresentadas possuem características específicas que oferecem vantagens dependendo do cenário.

Vantagens da Arquitetura escolhida (Container Instances, APIs, Event Hub, Stream Analytics, etc.)

- **Processamento em Tempo Real:** Esta arquitetura utiliza o Event Hub e Stream Analytics, permitindo o processamento em tempo real dos dados, o que é ideal para aplicações que exigem respostas instantâneas;
- **Análise Avançada:** Com o Stream Analytics, é possível aplicar transformações e realizar análises complexas nos dados antes de enviá-los para os destinos finais;
- **Armazenamento Distribuído:** Inclui o Cosmos DB, que é uma solução robusta para armazenar dados estruturados com alta disponibilidade global;
- **Flexibilidade de Destinos:** Permite o direcionamento dos dados para diferentes serviços, como Cosmos DB, Storage Accounts e bancos de dados RAW, atendendo a múltiplas finalidades.
- **Escalabilidade Global:** Adequada para cenários que demandam replicação de dados e baixa latência em uma escala global.

- **Vantagens da segunda Arquitetura (Docker Gerador de Dados, Queues, Functions, Data Factory, etc.)**
- **Desacoplamento de Componentes:** O uso de filas (Queues) desacopla os diferentes serviços, permitindo que cada parte do sistema funcione de forma independente. Isso facilita manutenção, atualizações e resolução de problemas;
- **Resiliência:** As filas armazenam mensagens temporariamente, garantindo que os dados não sejam perdidos mesmo em caso de falhas nos serviços subsequentes.
- **Simplicidade de Integração:** A arquitetura utiliza componentes bem conhecidos e de fácil integração, como filas e funções, reduzindo a complexidade do design;
- **Processamento Diferenciado:** Permite dividir os dados em diferentes fluxos (por exemplo, enviar para RAW BD ou Files), atendendo a necessidades distintas com eficiência.

Quando usar cada Arquitetura

- **Primeira Arquitetura:** Ideal para cenários que exigem processamento em tempo real, análises avançadas e armazenamento estruturado distribuído.
- **Segunda Arquitetura:** Mais adequada para sistemas que necessitam de resiliência, automação e desacoplamento entre os serviços, com custos otimizados.

Ambas as arquiteturas são poderosas e podem ser escolhidas com base nos requisitos específicos do projeto. Optamos por uma arquitetura baseada no **Event Hub** em vez das filas tradicionais (Queues), devido à sua capacidade de realizar streaming em larga escala, oferecendo baixa latência e maior eficiência no processamento de dados em tempo real.

Explicação detalhada da Arquitetura escolhida

→ **Github:** Ver na pasta \Exercicios\Projeto

Docker Instance → Explicação do Código da pasta

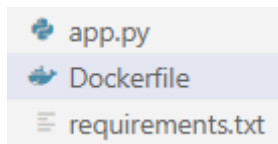


Figura 40 – pasta Docker

Explicação do código do ficheiro: app.py

1º Criar imagem Docker e Container → Gerador de dados

O código Python da Docker Instance conecta-se a um banco de dados SQL do Azure e realiza operações para gerar dados de vendas aleatórias e para posteriormente enviá-los para uma API.

Primeiramente, o código configura o logging, que é usado para registar mensagens de erro, sucesso ou informações úteis durante a execução do programa. Isso ajuda no monitoramento e na depuração.

Conexão à Base de Dados Azure

A função `connect_to_azure_sql` estabelece uma conexão com uma instância do SQL Server no Azure utilizando a biblioteca `pyodbc`. Caso a conexão seja bem-sucedida, ela é retornada para uso em outras funções. Em caso de erro, uma mensagem de erro é registrada e a função retorna `None`.

Captação dos Dados do Azure

Existem três funções principais que retiraram informações da Base de Dados do Azure:

- **`get_products()`:** Retira os nomes dos produtos da tabela `raw.products`;
- **`get_users_id_list()`:** Recupera uma lista de IDs de usuários da tabela `raw.users`;
- **`get_product_id_list()`:** Coleta uma lista de IDs de produtos da tabela `raw.products`.

Cada função verifica se a conexão é válida antes de executar consultas SQL e retorna os resultados como listas. Caso ocorram erros, mensagens são registradas nos logs.

Criar Vendas Aleatórias

A função principal, `gerar_sales_aleatorias()`, é responsável por gerar dados de vendas entre duas datas fornecidas.

Primeiro conecta-se ao banco de dados e obtém listas de produtos e IDs de usuários, de seguida gera para cada dia no intervalo, um número aleatório de vendas (entre 1 e 100).

Para cada venda, seleciona aleatoriamente um ID de produto e um ID de usuário, além de gerar um timestamp aleatório.

Cria uma carga de dados no formato JSON, contendo o ID de venda aleatório, o ID do produto, o timestamp e o ID do usuário.

De seguida envia essa carga para uma API usando o método HTTP POST. A URL da API é configurada no código, e o cabeçalho da requisição especifica que os dados estão no formato JSON.

Aguarda um segundo antes de processar a próxima venda, para evitar sobrecarga. Passa para o próximo dia no intervalo até que todos os dias tenham sido processados. A função regista mensagens de sucesso ou erro ao enviar os dados para a API.

Main

O bloco de código `if __name__ == '__main__'` executa o programa que tenta estabelecer uma conexão com o banco de dados. Caso a conexão seja bem-sucedida, chama a função `gerar_sales_aleatorias()` para gerar e enviar os dados de vendas. Caso contrário, regista um erro indicando que a conexão falhou.

O objetivo deste programa é criar um container instancie que automatiza a geração de vendas simuladas aleatoriamente e o envio para uma API, sendo útil para testes em sistemas que processam dados em tempo real. Ele demonstra boas práticas como o uso de logging, modularidade com funções separadas e tratamento de erros, o que o torna robusto e fácil de manter. Além disso, é adaptável, permitindo mudanças na URL da API ou nos dados gerados conforme necessário.

Explicação do código do ficheiro: Dockerfile

O código é um arquivo Dockerfile usado para criar uma imagem Docker personalizada destinada a executar um aplicativo Python. Ele segue uma série de etapas para preparar o ambiente necessário e configurar o container para executar a aplicação de forma eficiente.

FROM python:3.9-slim: É utilizada a imagem oficial do Python na versão 3.9-slim, que é uma versão reduzida e otimizada da imagem Python. Essa escolha minimiza o tamanho da imagem, garantindo que apenas os componentes essenciais sejam incluídos.

WORKDIR /app: Define-se o diretório de trabalho dentro do container, onde todos os arquivos da aplicação serão armazenados. Isso simplifica as operações subsequentes no ambiente do container.

COPY . /app: Copia todos os arquivos do projeto, localizados no diretório atual do host, para o diretório definido no container (/app).

RUN apt-get update && apt-get install -y \ (...): Este trecho de código realiza a instalação de dependências do sistema. Após a instalação, os caches de pacotes são limpos para reduzir o tamanho final da imagem.

RUN pip install --no-cache-dir -r requirements.txt: Instala as dependências necessárias para a aplicação Python, listadas no arquivo requirements.txt. A flag **--no-cache-dir** evita que o cache do pip seja armazenado, ajudando a manter a imagem menor.

CMD ["python", "app.py"]: É o comando para executar o aplicativo, onde define o comando padrão que será executado quando o container for iniciado. Neste caso, o script app.py será executado usando o interpretador Python.

Posteriormente podemos realizar a construção e execução da Imagem com o comando **“docker build -t image_eventhub .”** e de seguida executar a imagem em um container com **“docker run -d --name container_eventhub -p 8000:8000 image_eventhub”**.

Este Dockerfile automatiza a criação de um ambiente isolado para a aplicação Python, garantindo que todas as dependências necessárias estejam presentes. Ele é especialmente útil para implantar a aplicação em diferentes ambientes com consistência e praticidade, permitindo testes, desenvolvimento e produção em uma infraestrutura containerizada.

App Function → Explicação do Código da função `function_app.py`

Este código implementa uma aplicação baseada em Azure Functions que processa requisições HTTP do tipo POST e envia eventos para o Azure Event Hub, um serviço de ingestão de dados em tempo real.

O código começa por realizar a importação de bibliotecas e dos módulos essenciais, de seguida inicializa a aplicação.

`app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)` →

Este comando define a Azure Function com um nível de autenticação anônimo, permitindo que qualquer pessoa consiga aceder à função sem a necessidade de credenciais.

De seguida realiza-se a conexão com o Event Hub, que é estabelecida usando uma connection string, este é o Event Hub que receberá os eventos.

Passando para a definição da Função HTTP, esta é a função principal, `insert_EventHub_azure`, é registada para ser acionada quando uma requisição HTTP do tipo POST for enviada para a rota `insert_EventHub_azure`. Dentro dela, são realizadas diversas operações, tal como o processamento da requisição, a criação do evento, o envio ao Event Hub e o tratamento de erros.

O bloco “finally:” fecha o cliente produtor do Event Hub para liberta os recursos.

Podemos verificar que este código configura uma Azure Function para receber requisições HTTP do tipo POST, processar os dados enviados, e enviá-los como eventos para um Event Hub do Azure.

Explicação do código do ficheiro: `requirements.txt`

Este código utiliza o arquivo `requirements.txt` para gerenciar dependências de bibliotecas essenciais ao funcionamento de um projeto Python, permitindo que o ambiente de desenvolvimento seja replicado de forma eficiente. O arquivo inclui as seguintes bibliotecas:

- **psycopg2:** Esta biblioteca é usada para estabelecer conexões e executar operações em bancos de dados PostgreSQL. Ela oferece uma interface robusta para interagir com o banco, possibilitando execução de consultas, inserções e atualizações diretamente a partir do código Python;
- **pyodbc:** Esta biblioteca permite a conexão com bancos de dados que suportam ODBC (Open Database Connectivity), como o SQL Server. Ela é frequentemente utilizada em projetos que envolvem a interação com bancos de dados de diferentes tipos, proporcionando flexibilidade na integração;
- **requests:** É uma biblioteca altamente popular para realizar chamadas HTTP, seja para APIs RESTful ou outros serviços web. Ela facilita a execução de requisições

como GET e POST, envio de dados JSON e recebimento de respostas, sendo fundamental para integração com serviços externos.

O arquivo requirements.txt é utilizado em conjunto com o comando **pip install -r requirements.txt**, que instala automaticamente todas as dependências listadas. Isso garante que o projeto possa ser configurado de forma padronizada em diferentes máquinas, promovendo a consistência entre os ambientes de desenvolvimento, teste e produção.

Criação e conexão ao Container registry e Container instance em Azure

O processo começa com a criação da imagem chamada “image_eventhub” e do container denominado “container_eventhub” localmente no Docker. Este é o primeiro passo para preparar o ambiente.

No segundo passo, é necessário aceder ao Portal do Azure para criar uma Container Registry. Neste caso, foi criada uma registry com o nome “Container04”.

az login --tenant 6648d2ab-84ad-4a19-8ae1-c37cf174a849 → Conectar à conta do Azure a partir da máquina local usando o PowerShell, executando o comando. Esse comando permite autenticação com a modalidade Single Sign-On (SSO).

De seguida tivemos de seleccionar uma subscrição no Azure para continuar com as configurações.

az acr login --name container04 → Este comando foi usado para conectar à Container Registry chamada “Container04”. Após a execução bem-sucedida do comando, a mensagem “Login Succeeded” confirmou que a conexão foi estabelecida corretamente.

docker tag sha256:9ba43dd2feab0cf9612edc7bdc758ff9f64ff782fe5da1bf0b28c3b21888f3f7 container04.azurecr.io/image_eventhub:latest → Este comando cria uma nova tag para a imagem Docker existente.

docker push container04.azurecr.io/image_eventhub:latest → Com este comando envia-mos a imagem Docker com a nova tag para o Azure Container Registry (ACR), permitindo que a imagem fosse armazenada de forma centralizada e estivesse pronta para implantação em outros ambientes.

```

REPOSITORY          TAG          IMAGE ID      CREATED      SIZE
image_eventhub      latest      9ba43dd2feab 31 hours ago 574MB
image_queue_insert_bd latest      0b7a39727cbc 32 hours ago 574MB
PS C:\Estudos\Programação\UpSkill\05_Modulo_Azure\Projeto_Avaliacao\Exercicios\Exercise_20\20.1\1.Docker_ETL_API_Flask_Gerador_de
_dados> docker push container04.azurecr.io/image_eventhub
Using default tag: latest
The push refers to repository [container04.azurecr.io/image_eventhub]
tag does not exist: container04.azurecr.io/image_eventhub:latest
PS C:\Estudos\Programação\UpSkill\05_Modulo_Azure\Projeto_Avaliacao\Exercicios\Exercise_20\20.1\1.Docker_ETL_API_Flask_Gerador_de
_dados> docker tag sha256:9ba43dd2feab0cf9612edc7bdc758ff9f64ff782fe5da1bf0b28c3b21888f3f7 container04.azurecr.io/image_eventhub:
latest
PS C:\Estudos\Programação\UpSkill\05_Modulo_Azure\Projeto_Avaliacao\Exercicios\Exercise_20\20.1\1.Docker_ETL_API_Flask_Gerador_de
_dados> docker push container04.azurecr.io/image_eventhub:latest
The push refers to repository [container04.azurecr.io/image_eventhub]
c382c3cb70fd: Pushing [=====] 4.194MB/4.778MB
7cf63256a31a: Pushing [=====] 3.146MB/28.22MB
82a59f028fa3: Pushed
1cce91ca30f0: Pushed
7c3c8cf8b5c0: Pushed
a64b7cccb523: Pushing [=] 3.146MB/92.65MB
fb0009da06dd: Pushed
768545f3e542: Pushed
09f8c21e9f8e: Pushing [=====] 7.34MB/14.94MB

```

Figura 41 – Container registry

az acr repository list --name container04 --output table → Com este comando podemos verificar as imagens que se encontram no container04 do Azure:

```

_dados> az acr repository list --name container04 --output table
Result
-----
image_eventhub
my-python-app
my-python-app1
PS C:\Estudos\Programação\UpSkill\05_Modulo_Azure\Projeto_Avaliacao\
_dados>

```

Figura 42 – Imagens do Container registry do Azure

Após a criação da Container Registry (repositório de imagens) no Azure, foi configurada uma Container Instance utilizando como base a imagem **imagem_eventhub** armazenada na Container Registry. A configuração foi realizada para garantir que a instância estivesse pronta para executar o aplicativo a partir da imagem definida.

Microsoft Azure Search resources, services, and docs (G+/I) Copilot

Home > Container instances >

Create container instance ...

Project details
Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *

Resource group * [Create new](#)

Container details

Container name * ✓

Region *

Availability zones (Preview)

SKU

Image source * ☒ Quickstart images ☒ Azure Container Registry ☐ Other registry

Run with Azure Spot discount ☐

Registry *
! If you do not see your Azure Container Registry, ensure you have been assigned the Reader Role for the Azure Container Registry or select an Azure Container Registry in a different subscription. [Learn more](#) >

Image *

Image tag *

OS type * ☒ Linux ☐ Windows

Figura 43 – Criação da Container Instance

Azure

De seguida foi criada a App Function chamada “04eventhubfunction” no Azure.

Inicializou-se o serviço da Container Instance “04eventhubcontainer”, o que resultou no acionamento automático de um trigger para a App Function “04eventhubfunction” e a partir desse momento, esta começou a gerar eventos e a enviá-los para o Event Hub denominado “04sales-events_teste”. Este Event Hub, por sua vez, direciona os eventos recebidos para a Stream Analytics chamada “04StreamAnalytics”.

A Stream Analytics recebe os dados do Event Hub como um input e de seguida processa os eventos e os encaminha com sucesso para três outputs configurados anteriormente: o **CosmosDB**, uma **Base de Dados** alojada no Azure para a tabela Raw.salestream e para a **Storage Account** “storageaccounttt04”.

Todo o fluxo foi implementado com êxito, garantindo a integração completa dos componentes.

Podemos verificar a chegada dos novos eventos ao **Event Hub**:

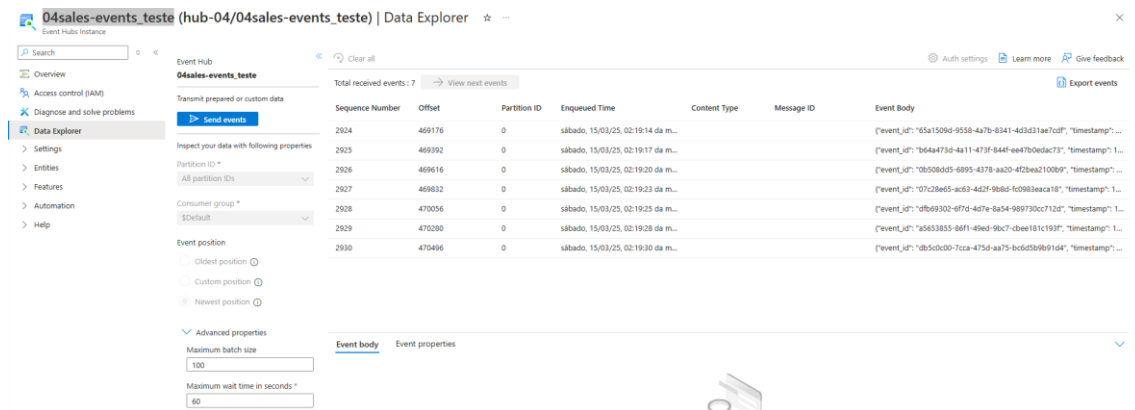


Figura 44 – Novos eventos - Event Hub

Podemos verificar que os dados foram inseridos com sucesso na **CosmosDB**:

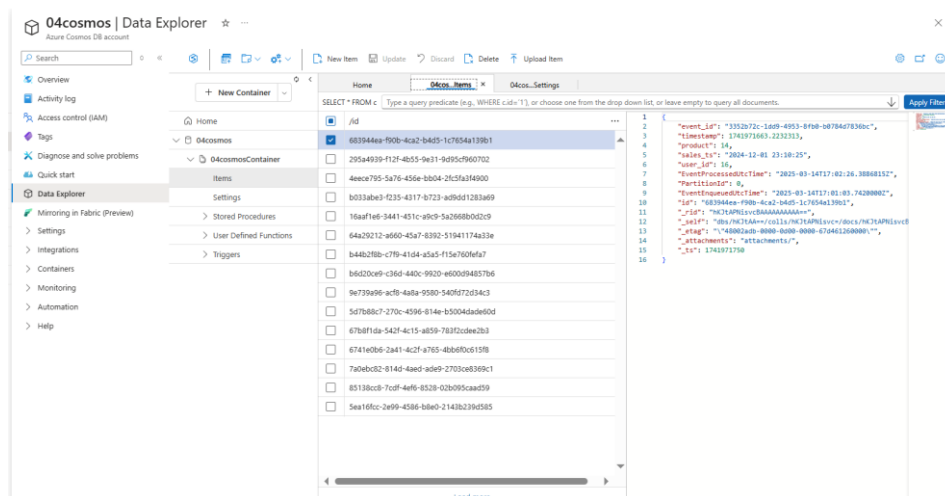
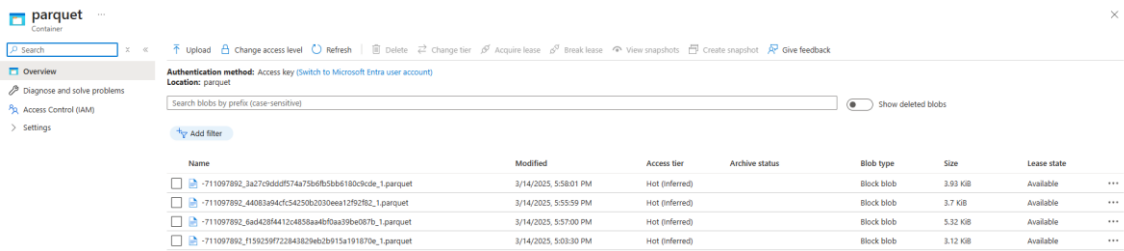


Figura 45 – CosmosDB

Podemos verificar que os dados foram inseridos com sucesso na Storage account:

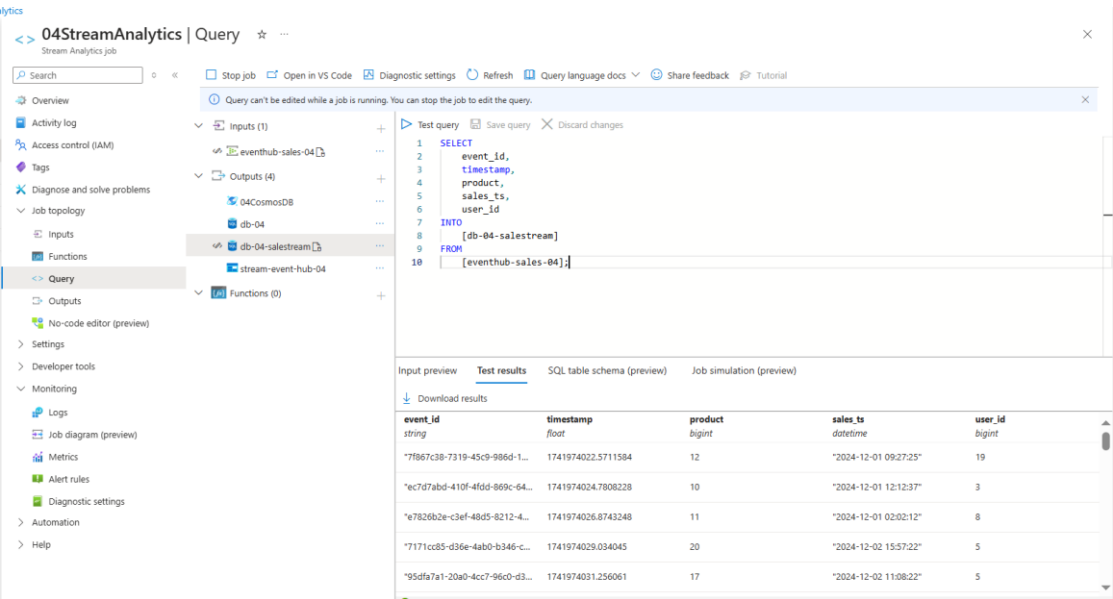


Name	Modified	Access tier	Archive status	Blob type	Size	Lease state
711097892_3a27c6dd574a75bdfb5b6d190c9cde_1.parquet	3/14/2025, 5:58:01 PM	Hot (inferred)		Block blob	3.93 KiB	Available
711097892_44063a94c54250b2030eeaa12f9282_1.parquet	3/14/2025, 5:55:59 PM	Hot (inferred)		Block blob	3.7 KiB	Available
711097892_6a428f4412c4838aa4f0a39be087b_1.parquet	3/14/2025, 5:57:00 PM	Hot (inferred)		Block blob	5.32 KiB	Available
711097892_f15925972384329eb32815a191870e_1.parquet	3/14/2025, 5:03:30 PM	Hot (inferred)		Block blob	3.12 KiB	Available

Figura 46 – Storage accounts

Nota: Os dados armazenados na Storage Account estão no formato Parquet. Para visualizar as informações contidas nesses arquivos, basta descarregar o ficheiro pretendido do portal e pesquisar na internet por ferramentas como "parquetreader", que permitem abrir e consultar as linhas do arquivo Parquet de forma simples e eficiente.

Para inserir os dados na **Base de Dados** na tabela Raw. Salestream temos de ter esta query:



```
SELECT
  event_id,
  timestamp,
  product,
  sales_ts,
  user_id
INTO
  [db-04-salestream]
FROM
  [eventhub-sales-04];
```

Figura 47 – Query DB - Stream Analytics

Podemos verificar que os dados foram inseridos com sucesso na Base de Dados:

	event_id	timestamp	product	sales_ts	user_id
29	cd2edc93-491b-48b3-9727-6f436867004b	1,741,977,566.8894522	13	2024-12-09 15:09:05.000	19
30	d11a0f8d-b4cf-4258-be5e-fb4d01487cfe	1,741,977,569.0699933	20	2024-12-09 12:21:26.000	12
31	777beb8b-051a-4c25-bf2b-c245dc6aac11	1,741,977,571.4465332	20	2024-12-09 15:14:51.000	1
32	47972279-39c6-4dba-80d1-966d858d0588	1,741,977,573.5235524	5	2024-12-09 08:55:15.000	23
33	cb65223f-ee90-478a-90cf-5852394edddde	1,741,977,575.5227146	7	2024-12-09 23:23:00.000	11
34	7942b6a9-4e11-417d-a4b2-90148b499c2f	1,741,977,577.4899354	20	2024-12-09 04:01:21.000	17
35	593d5e07-cf5e-47fa-af65-b80f8257adc7	1,741,977,579.4892719	12	2024-12-09 12:06:43.000	6
36	1629ed45-5e53-41ed-82a0-fbacd8434b70	1,741,977,581.592315	20	2024-12-09 09:00:51.000	2
37	dedfd30-fa6e-462a-987f-d994a6328dcc	1,741,977,583.566574	8	2024-12-10 18:02:20.000	15

Figura 48 – Base de dados

Após a receção dos dados na Base de Dados Raw, podemos prosseguir para o processamento no Data Factory. Este processo já foi detalhadamente abordado nos exercícios 16 e 18. Para evitar duplicação de informação, basta referir que o fluxo seguido é semelhante ao descrito nesses exercícios.

→ Além disso, é importante assistir aos dois vídeos que mostram o fluxo de dados em execução conforme o esperado, garantindo que todo o processo esteja funcionando corretamente.

Conclusão

Este módulo de Azure foi fundamental para uma compreensão mais profunda sobre o funcionamento, os conceitos e a aplicação de soluções em cloud computing, com um foco especial no Azure, que foi a plataforma mais explorada durante o curso. Aprendemos que um bom profissional de Cloud precisa ir além do simples uso do portal do provedor de serviços. É essencial saber trabalhar com diferentes plataformas, como Azure, AWS, Google Cloud e Oracle, além de ter a capacidade de se adaptar rapidamente a novas ferramentas e tecnologias, que estão em constante evolução. Estar disposto a enfrentar novos desafios é uma qualidade indispensável.

Este módulo teve um carácter bastante prático e dinâmico, como pode ser observado nos 24 exercícios apresentados e detalhadamente explicados neste relatório. No entanto, também incluiu uma componente teórica valiosa, cobrindo os tópicos da certificação AZ-900 - Azure Fundamentals da Microsoft. O próximo passo natural será realizar com sucesso essa certificação.

Ao longo do módulo, tivemos contato com diversas aplicações e funcionalidades abordadas pela certificação, bem como outras ferramentas que, embora não estejam diretamente incluídas no exame AZ-900, são essenciais para nossa formação profissional em Cloud.

Concluir este curso marca o início da minha trajetória no mundo da Cloud Computing, representando um passo importante para o desenvolvimento da minha carreira nesta área.