Université Catholique de Louvain

Microservices Documentation for Back-End deployment

Botton David
De Roover Filip

Table of Contents

November 2024

Contents

Abstract

This document provides a detailed description of the services implemented in the project, the technologies used, the deployment instructions, API details, and configuration setup for the first deliverable of the project, either employing Docker Swarm, Microsoft Azure or a bare metal machine. The entirety of the source code and the complete development pipeline is available as a GitHub repository at URL: https://github.com/nottoBD/LINFO2145-2024-2025/tree/main

1 Introduction

This project is based on a microservice architecture, utilizing Docker containers for various services including databases, API gateways, authentication, logging, and front-end. All services are interconnected and communicate over a Docker network called `scapp-net`.The following documentation is structured in several parts describing the different back-end services implemented with their associated technologies, the deployment script(s), azure configuration for deployment on VM's and logging items as demanded. The full API description for all services is detailed on the Swagger endpoint and as comments in the `api-gateway` service's source code.

2 List of Services

This section describes the various back-end services implemented and their characteristics.

2.1 API description

The full API description for all services is described below using Swagger UI, a service for interactive API documentation, and is available under the path `http://IP_ADDRESS:8000/api-docs/` once the API gateway service is deployed. See Figure 1: Swagger documentation. Also, find all documented endpoints in the folder src/routes/ of the `api-gateway` service.

2.2 API Gateway (`api-gateway`)

The API Gateway serves as the entry point for all client requests and forwards them to appropriate backend services. The API Gateway acts as the central entry point for routing HTTP requests to various microservices using HTTP verbs. It enforces role-based authentication and authorization, logs events, and provides API documentation using Swagger. API Gateway enforces RBAC by validating front-end tokens using an authentication middleware wired up to the `scapp-auth` service. From the token are derived session information and user role, multiple HTTP authentication headers are parsed here, in the API gateway, before being forwarded if allowed. It also includes a logger middleware, formating events for the `scapp-logs` service to save in its dedicated CouchDB instance.

Associated Technologies

- **Node.js** - Primary runtime environment for the API Gateway.
- **Express.js** - Framework used to build the server and manage routes.
- **http-proxy-middleware** - Middleware for proxying requests to back-end services.
- **dotenv** - For environment variables management.
- **JWT** - For handling authentication tokens.
- **Swagger** - For interactive API documentation.

Why we have chosen `http-proxy-middleware` over a Nginx server

For this project, the goal was to quickly set up a backend to allow the frontend team to test their user interface (UI) without waiting for the complete server architecture to be finalized. To achieve this, we chose http-proxy-middleware over more robust solutions like Nginx. This approach enabled us to deploy a proxy quickly and integrate routing logic directly within our Node.js/Express application, which is more practical for frequent and application-specific adjustments, such as role-based access control, without needing

to modify a static external configuration.By selecting http-proxy-middleware, we simplified the process by consolidating proxying, authentication, logging, and validation into the application itself, reducing the need to manage multiple configurations or external systems like Nginx. Additionally, this solution avoids adding an extra Nginx layer, especially in a context where the main goal is to provide a functional backend for the frontend. Finally, while http-proxy-middleware may not offer the same performance as Nginx for static content delivery, it is more than sufficient in this case, where the focus is on routing, authentication, and request handling, rather than serving static content. This fast and flexible approach aligns well with the current development needs, where the priority is enabling the frontend team to test their components without delay. Overall, http-proxy-middleware fits better with our development workflow and dynamic needs, making it a more maintainable choice than Nginx in this case.

Usage of `dotenv`, `JWT`, `Swagger UI`

The choice of `dotenv`, `JWT`, and `Swagger UI` reflects their suitability for key functionalities of the API Gateway. `dotenv` is used for managing environment variables, providing a simple yet effective way to securely configure sensitive information such as API keys, database credentials, and service endpoints. It integrates natively with the Node.js ecosystem and supports development and production environments without exposing secrets in the source code. `JWT` (JSON Web Tokens) is employed for authentication, offering a compact and stateless mechanism to verify user identity across services. Its efficiency in transmitting claims between parties in a self-contained format makes it ideal for the distributed nature of microservices architectures. Finally, `Swagger UI` facilitates interactive API documentation, enabling developers to explore, test, and understand endpoints directly from a web interface. It ensures that the documentation stays synchronized with the actual API implementation.

Ports: 8000 (mapped to 8000).

2.3 NoSQL Databases

The following services make usage CouchDB to store data for the application during user interactions as well as logging information. Each database operates in it's own Docker container and is configured with it's local `local.ini` file. CouchDB was selected as the primary database solution for this project due to its unique strengths in handling distributed, document-oriented data, which align closely with the requirements of a microservices-based architecture. CouchDB ensures data consistency across multiple instances and facilitates offline-first capabilities if required. Its schema-less, JSON-based document storage model is ideal for managing diverse datasets such as user profiles, products, carts, logs, and checkout information, as it allows for flexible schema evolution without downtime. Additionally, CouchDB's integrated HTTP API simplifies interactions between microservices, reducing the overhead of additional database connectors or drivers. It also provides efficient data retrieval mechanisms that could be exploited in future versions of the project (e.g, MapReduce for logs). Running each CouchDB instance in its own Docker container further enhances scalability, isolation, and ease of configuration. The setup ensures robust data storage among all the rest. Each database has a Fauxton interface endpoint with default credentials. These web interfaces improve the management and configuration of each couchdb instance, allowing a sharding setup, replication, or browsing JSON documents, executing queries.

Users Database (CouchDB - `users-db`)

Role: Stores user-related data and credentials for the application.
Technologies: Docker, CouchDB
Docker Setup:

```
docker compose build users-db
docker compose up -d users-db
```

API: The service interacts with other services such as `products-db`, `checkout-db`, etc.

Ports: 3010 (mapped to 5984).
Volumes:

- `users_db_data` for CouchDB data storage.
- `local.ini` for service configuration.

Products Database (CouchDB - `products-db`)

Role: Stores product data for the application.
Technologies: Docker, CouchDB
Docker Setup:

```
docker compose build products-db
docker compose up -d products-db
```

API: Interacts with `cart-db`, `users-db`.
Ports: 3020 (mapped to 5984).
Volumes:

- `products_db_data` for CouchDB data storage.
- `local.ini` for configuration.

Cart Database (CouchDB - `cart-db`)

Role: Stores shopping cart data for users.
Technologies: Docker, CouchDB
Docker Setup:

```
docker compose build cart-db
docker compose up -d cart-db
```

API: Interacts with `checkout-db`, `users-db`, `products-db`.
Ports: 3030 (mapped to 5984).
Volumes:

- `cart_db_data` for CouchDB data storage.
- `local.ini` for configuration.

Logs Database (CouchDB - `logs-db`)

Role: Stores logs for the system's activities.
Technologies: Docker, CouchDB
Docker Setup:

```
docker compose build logs-db
docker compose up -d logs-db
```

API: Logs data is aggregated from various services.
Ports: 3040 (mapped to 5984).
Volumes:

- `logs_db_data` for CouchDB data storage.

- `local.ini` for configuration.

Checkout Database (CouchDB - `checkout-db`)

Role: Stores checkout-related data, processing user orders.
Technologies: Docker, CouchDB
Docker Setup:

```
docker compose build checkout-db
docker compose up -d checkout-db
```

API: Interacts with `cart-db`, `users-db`.
Ports: 3050 (mapped to 5984).
Volumes:

- `checkout_db_data` for CouchDB data storage.
- `local.ini` for configuration.

2.4 Logging Service (`scapp-logs`)

The **Logging Service** is a centralized system for collecting, persisting, and managing logs across all microservices in the application. It operates as an Express.js server with API routes to handle log storage, querying, and deletion. Logs are persisted using CouchDB, chosen for its JSON-native document model and HTTP-based API, which align well with the structured nature of log data and simplify integration with other services. The service is containerized with Docker, ensuring consistent deployments and persistent storage through Docker volumes. A log retention policy is implemented to automatically delete logs older than a configurable period (default: 7 days), with the policy running every 24 hours. The service also features robust error handling, capturing application-level and process-level errors, including unhandled exceptions and promise rejections, and logging them for traceability.
The service supports CORS for cross-origin requests, enabling secure logging across different microservices, and includes a health check endpoint (/health) to track service status. It uses a custom logger for structured log output. CouchDB's replication features allow for easy scalability and fault tolerance.
CouchDB is chosen for its schema-less structure, support for RESTful operations, and efficient handling of JSON data. Express.js provides a lightweight framework for building the service, and Docker ensures isolated, consistent deployments. These technologies work together to offer a reliable and scalable solution for centralized logging management in a microservice setup.
Every backend service integrates a **structured logger** in its `daemon.js` file for capturing structure-specific logs. While these individual loggers manage these events, the application logging service (`scapp-logs`) takes responsibility for handling all user interactions, database operations, and events across the microservices. The application-related logs are stored in the `logs-db` CouchDB instance and available through HTTP requests and its Fauxton designated interface.
To assess `structure-related` logs:

```
docker logs scapp-logs
```

Docker Compose:

```
docker compose build scapp-logs
docker compose up -d scapp-logs
```

Port: 4700

2.5 Checkout service (`scapp-checkout`)

The **checkout service** is responsible for managing the checkout process, handling payment simulation, creating orders, and clearing users' carts once the purchase is completed. It also provides the user's purchase history. When a user initiates a checkout, the service first verifies that the request contains both a user ID and a valid authentication token. It then fetches the user's active cart and validates the items in it. The service calculates the total amount of the cart and proceeds with simulating a payment processing step . If the payment is successful, the service creates an order in the database, clearing the user's cart in the process. If payment fails, the service logs the error and returns a failure response. Additionally, the service provides an endpoint to fetch the user's purchase history, listing all completed orders. It ensures efficient querying through the creation of necessary database indexes and logs events like errors and successful order creation. A health check endpoint is also available to confirm the service's availability. The service uses Express.js for routing and handling HTTP requests, integrates with CouchDB for persistent data storage, and employs CORS to allow safe cross-origin requests.
To assess `structure-related` logs:

```
docker logs scapp-checkout
```

Docker Compose:

```
docker compose build scapp-checkout
docker compose up -d scapp-checkout
```

Port: 4600

2.6 Products service (`scapp-products`)

The **Products Service** is responsible for managing the lifecycle of products in an e-commerce system. It provides a variety of endpoints to create, update, fetch, and delete products. The service starts by setting up CORS for cross-origin requests and listens on a specific port. It uses Express.js for routing and handles all product-related operations, such as creating new products, updating existing ones, deleting products, and retrieving product details.

When creating or updating a product, the service ensures that all required fields are provided and performs necessary validations. It then stores the product in the database, logs the event, and returns the created or updated product in the response. For deleting a product, it checks the user's role to ensure only admins can perform this action and also notifies the cart service about the deletion using the Cart Notification Utility. This utility sends a request to the cart service to remove the deleted product from all active carts.

The service also provides an endpoint to fetch all products or a single product by ID, and it includes a health check endpoint to verify if the service is running. Throughout the service, logs are generated for each significant action (product creation, update, or deletion) to track events and errors for traceability. The service ensures a robust architecture with error handling, logging, and process management (for uncaught exceptions for example).
To assess `structure-related` logs:

```
docker logs scapp-products
```

Docker Compose:

```
docker compose build scapp-products
docker compose up -d scapp-products
```

Port: 5000

2.7 Authentification/User service (`scapp-auth`)

The **Authentification service** is an Express-based server designed to manage user-related operations in a backend system, such as user authentication, registration, and profile retrieval. It includes a variety of middleware for logging incoming requests, handling errors, and setting up cross-origin resource sharing (CORS). The server listens for incoming requests, logs events, and processes authentication via JWT tokens. Key API endpoints include /validate for validating the JWT token, /me for retrieving the authenticated user's profile, and /user for user registration, which involves validating input, creating a new user in the database, and generating a JWT token. The /user/authenticate endpoint handles user authentication by validating the provided username and password, fetching user details, and generating a JWT token. Error handling is robust, with specific log entries for different failure scenarios such as invalid tokens, unsuccessful authentication, or missing parameters. The server also includes catch-all error handling for undefined routes, logging detailed information about each request, including the user agent and IP address. The daemon listens on a configurable port, and includes event handling for uncaught exceptions and unhandled promise rejections, ensuring that critical issues are logged and managed properly. Additionally, logs are structured to include event levels (INFO, ERROR, CRITICAL), context, and detailed error information,
To assess `structure-related` logs:

```
docker logs scapp-auth
```

Docker Compose:

```
docker compose build scapp-auth
docker compose up -d scapp-auth
```

Port: 4000

2.8 Cart service (`scapp-cart`)

The **Cart service** built with Node.js and Express, is designed to manage user shopping carts for our e-commerce platform. It uses JWT (JSON Web Tokens) for authentication, ensuring only authorized users can access and modify their carts. The service features several key functionalities, including middleware for validating JWT tokens, which decode the token to extract user information and role, attaching them to the request object. Unauthorized requests or invalid tokens trigger a 401 response. The service supports CRUD operations: adding items to a cart (POST /cart), removing a specific product (DELETE /cart/:productId), updating item quantities (PATCH /cart/:productId), and clearing the cart (DELETE /cart). Admins can remove a product from all active carts (DELETE /cart/remove-product/:productId). A "Get Cart" endpoint (GET /cart) retrieves the current state of a user's cart. Items are validated for essential fields like product_id, name, price, quantity, and imageUrl. All actions are logged for both successes and failures, capturing detailed data such as user agent, IP address, session ID, and error details. The service includes CORS middleware, allowing cross-origin requests from a specified frontend origin , and a simple health check endpoint (GET /health) to verify service status. The server listens on a configurable port, loading environment variables from a .env file for flexible configuration.
To assess `structure-related` logs:

```
docker logs scapp-cart
```

Docker Compose:

```
docker compose build scapp-cart
docker compose up -d scapp-cart
```

Port: 4500

2.9 Azure Blob service (`scapp-blobs`)

The **Azure Blob service** is a RESTful API designed to handle image file uploads to Azure Blob Storage, process the images for optimization, and manage error handling and logging. It is built using Node.js with Express and utilizes Multer for handling file uploads. The service allows image files to be uploaded via a POST /upload route, where the image is first processed (resized and converted) using the Sharp library to optimize it for web usage. The processed image is then uploaded to Azure Blob Storage using the uploadImageToAzure function, which generates a unique filename using UUID to avoid collisions. The Azure Blob service also includes a health check endpoint (GET /health) to verify its operational status. All operations, including uploads and errors, are logged using a custom logEvent utility, providing detailed logs for success and failure cases. The image processing ensures that only supported formats (JPEG, PNG, WebP) are accepted, with various adjustments made depending on the format—such as resizing and applying compression to reduce file sizes for faster web loading. Temporary files are cleaned up after upload to maintain server resources. The service is configured with environment variables to specify storage connection strings and container names, and it listens on a port defined by the environment or a default port (4250). Error handling includes logging uncaught exceptions and unhandled promise rejections, ensuring stability for critical failures.
To assess `structure-related` logs:

```
docker logs scapp-blobs
```

Docker Compose:

```
docker compose build scapp-blobs
docker compose up -d scapp-blobs
```

Port: 4250

2.10 Front-end service (`scapp-frontend`)

The frontend provided to us communicates with backend APIs to fetch dynamic data, such as products, which are displayed in the user interface. We have added a refresh mechanism to it. If products do not load immediately, a refresh mechanism ensures their visibility. Developers can simulate a slow network (for example, by throttling to 2G via developer tools `F12` → `Network` → `Throttle` → `2G`) to observe the automatic retry system in action. This updates the UI without requiring a page reload. A description of it is here below

Docker Compose :

```
docker compose build scapp-frontend
docker compose up -d scapp-frontend
```

Port: 3000

3 Additional Functionalities

The project includes innovative features that exceed the requirements. These features are well implemented, useful, and detailed in this report. They showcase intercommunication between services and enhance user experience in our eCommerce platform.

3.1 Feature 1: Automatic Removal of Deleted Products from Active Carts

3.1.1 Technical Overview

The implemented functionality ensures that when a product is deleted from the products service, it is also removed from all active carts (active cart = before checking out the content of the cart). This in-

tercommunication between the `scapp-products` and `scapp-cart` services is achieved through a utility file, `cart-notify.js`, in the products service.

The API Gateway forwards product deletion requests to the `scapp-products` service. Upon receiving the request, `scapp-products` performs the deletion of the product by its `_id` in CouchDB and uses the `cart-notify.js` utility to notify the `scapp-cart` service. The cart service then triggers the removal of the product from all active carts, ensuring data consistency across the platform.

3.1.2 Logical Flow

The logical flow of this feature is as follows:

1. A DELETE request for a product is sent to the API Gateway.
2. The API Gateway forwards the request to the `scapp-products` service.
3. The `scapp-products` service deletes the product by its CouchDB `_id`.
4. After the product is deleted, the `cart-notify.js` utility is invoked to notify the `scapp-cart` service.
5. The `cart-notify.js` sends a DELETE request to the `scapp-cart` service's endpoint, `/cart/remove-product/{productId}`, with an admin token for authentication.
6. The `scapp-cart` service processes the request and removes the product from all active carts.
7. If the notification or deletion fails, the system logs an error for debugging.

3.1.3 Technical Implementation

The `cart-notify.js` file is a utility function implemented in the `scapp-products` service. For the complete implementation, see Figure 3, and Figure 4 .

3.1.4 Conclusion

This feature demonstrates robust intercommunication between microservices, ensuring consistency and reliability in our eCommerce platform. By enabling automatic synchronization between the products and cart services, we have enhanced the system's usability and maintainability. This functionality is a clear example of how the project goes beyond basic requirements, adding value to the overall implementation.

3.2 Feature 2: Automatic Products Refresher for Slow Connections

3.2.1 Technical Overview

The implemented functionality addresses an issue where the frontend fails to load products due to slow client connections, resulting in a blank homepage. To enhance user experience, an automatic products refresher has been added to the `scapp-frontend` Docker service. This ensures that products are reloaded automatically when the initial load fails.

3.2.2 Logical Flow

The logical flow of this feature is as follows:

1. The client accesses the eCommerce homepage.
2. If the connection is slow (e.g., simulated using 2G network settings), the frontend may fail to load products.
3. The automatic products refresher detects the failure to load products.

4. The refresher triggers a retry mechanism to reload the products from the backend service.
5. Once the products are successfully loaded, they are displayed on the homepage.
6. If the products cannot be loaded after several attempts, an error message is displayed to the user.

3.2.3 Technical Implementation

The automatic products refresher is implemented within the `scapp-frontend` Docker service. It utilizes JavaScript to detect when the products array is empty after an initial load attempt. If empty, it automatically retries fetching the products from the backend API at regular intervals until the products are successfully retrieved or a maximum number of retries is reached. This functionality has been tested using browser developer tools by simulating slow network conditions (e.g., using DevTools with `F12 > Network > Throttling > 2G`).

3.2.4 Conclusion

This feature ensures that products are loaded and displayed even under suboptimal network conditions. It improves the reliability and accessibility of our eCommerce platform, and is an indicator of lowered connection performances and helps handling these more gracefully.

4 Justification of Technology Choices

- **Node.js**: We chose Node.js because it is fast and can handle many tasks simultaneously (I/O-heavy operations) without slowing down. This makes it ideal for building responsive backends.
- **Express.js**: Express.js is a lightweight framework that simplifies the creation of RESTful APIs. It's flexible and integrates well with Node.js, allowing us to quickly build and manage routes for our application.
- **CouchDB**: CouchDB was selected for its flexibility as a schema-less database. It allows us to handle rapidly changing data structures without the need for complex database schema migrations, making it ideal for fast-changing applications.
- **JWT (JSON Web Tokens)**: JWT was chosen for authentication because it is stateless, meaning we do not need to store session data. It is secure, scalable, and works well for managing user authentication across different systems.
- **Docker**: Docker simplifies packaging our application and running it in different environments (development, testing, production) without compatibility issues. It makes deployment and scaling easier by isolating the application into containers.
- **Docker Swarm**: Docker Swarm is Docker's native orchestration tool for managing and deploying a cluster of Docker nodes. It enables high availability, load balancing, and scaling of services across multiple hosts.
- **Docker Compose**: Docker Compose is a tool for defining and running multi-container applications using a simple YAML configuration file. It is now a Docker CLI plugin requiring Docker version 20.10 or later, replacing the standalone 'docker-compose' tool.
- **Azure Cloud**: Microsoft Azure provides tools such as Virtual Machines (VMs), Virtual Networks (VNets), Network Security Groups (NSGs), and Azure Blob Storage to support Docker Swarm deployments. These tools enable secure communication, centralized logging, and scalable, fault-tolerant infrastructure for microservices.

5 Logging Items

The Logging Service logs the following events:

- **Event Levels:** INFO, WARN, ERROR.
- **Details:** Context of the event, user ID, error stack traces, timestamps.

Items Logged by Each Microservice

The following are examples of items logged by each back-end service:

5.1 Authentication Service (`scapp-auth`)

- User login attempts
- User registering attempts
- Failed authentication events
- Daemon status

5.2 Products Service (`scapp-products`)

- CRUD products
- Daemon status

5.3 Cart Service (`scapp-cart`)

- Cart updates
- Daemon status

5.4 Checkout Service (`scapp-checkout`)

- Checkout cart
- Daemon status

5.5 Frontend Service (`scapp-frontend`)

- Inspecting a product's page
- Session state, role, and JWT token

See Figure 2: Logs Sample for the authentication service.

6 Deployment

6.1 Local Deployment

The whole infrastructure can be deployed locally using a deployment script for automation to deploy all services, running the following commands to build and start the containers:

```
git clone -b development https://github.com/nottoBD/LINFO2145-2024-2025.git
cd LINFO2145-2024-2025/project
docker compose up --build --remove-orphans -d
######
```

This script will automatically configure the necessary Docker volumes, networks, and environment variables. Ensure that your Docker daemon is running and configured to use the correct Docker Compose version.

6.2 Swarm Deployment

The whole infrastructure can be deployed using Docker Swarm and two Virtual Machines in a Manager-Worker model. Ensure your network interfaces use a bridged connection and that your machines resolve each other by IPv4 and Docker services. No Nginx server is needed in the context of the project as long as machines have their netplan and firewall configured.

The Docker overlay network creation and all data volumes are handled in docker-compose.yml, only prebuilt images from a public Docker registry are used (https://hub.docker.com/repository/docker/nottobd/linfo2145/tags).
Running the following commands will build and start the containers over a Docker Swarm:

- Initialize the Swarm manager:

  ```
  docker swarm init --advertise-addr <Manager_IP>
  ```

- Add worker nodes using the join token:

  ```
  docker swarm join --token <SWARM_TOKEN> <Manager_IP>:2377
  ```

Once Swarm is initialized and machines manage to resolve each other, run the following commands:

```
git clone https://github.com/nottoBD/LINFO2145-2024-2025.git
cd LINFO2145-2024-2025/project
docker stack deploy -c docker-compose.yml scapp
######
```

Open your browser on your local machine and access the front-end service running on the manager VM.

6.3 Required Configuration on Microsoft Azure

To run a Docker Swarm cluster on Microsoft Azure, the following configuration and resources are required:

6.3.1 Virtual Machines (VMs)

- **VM Sizes:**
  - Minimum: `Standard B1ms` (1 vCPU, 4 GB RAM) for master and worker nodes.
- **Operating System:**
  - Ubuntu 20.04 LTS or higher.
- **Storage:**
  - At least 30 GB OS disk per VM.

6.3.2 Networking Configuration

- **Azure Virtual Network (VNet):**
  - Create a VNet for internal communication between Swarm nodes.
  - Subnet allocation: At least a `/24` subnet for Swarm nodes (e.g., `10.0.0.0/24`).
- **Public IP:** Assign static public IPs for manager nodes if external access is needed.

6.3.3 Security Group Rules

Inbound Rules:

Port Range	Protocol	Source	Purpose
2376	TCP	Any	Secure Docker daemon connection (TLS).
2377	TCP	VNet	Docker Swarm manager communication.
7946	TCP/UDP	VNet	Swarm node-to-node communication.
4789	UDP	VNet	Overlay network traffic.
8000	TCP	Any	API gateway service.
3000	TCP	Any	Front-end service.
5984	TCP	VNet	CouchDB communication.
3010, 3020, 3030, 3040, 3050	TCP	Any	Fauxton interfaces for CouchDB instances.

Table 1: Inbound Security Group Rules

Outbound Rules:

Port Range	Protocol	Destination	Purpose
443	TCP	Internet	Pull Docker images from registries.
Any	Any	VNet	Internal communication between nodes.

Table 2: Outbound Security Group Rules

6.3.4 Azure Templates

The following is an Azure Resource Manager (ARM) template to automate cluster creation, including a public Azure Blob Storage configuration:

Azure Resource Manager (ARM) templates allow for fully automated deployment of Azure resources using a declarative JSON syntax. This ensures consistency, reduces manual errors, and enables easy scaling or modification for multiple environments.

In the provided template:

- **Virtual Network (VNet):** A VNet is created to ensure secure communication between Swarm nodes. This network isolates internal traffic, providing a private environment for Docker Swarm operations.
- **Virtual Machines (VMs):** The template provisions VMs with the minimum required specifications (`Standard B1ms`) and ensures the operating system (Ubuntu 20.04 LTS) is pre-installed. The template also handles OS disk configuration and public IP assignment for manager nodes.
- **Network Security Groups (NSG):** Security rules are automatically applied:
  - Inbound rules allow communication for Docker Swarm and application services (e.g., 3000, 8000, etc.).

  - Outbound rules permit secure traffic to pull Docker images and internal communication between services.

- **Azure Blob Storage:** The template includes a public Azure Blob Storage account configured to:
  - Store logs and other application data persistently.
  - Use the "Hot" access tier for optimized performance with frequently accessed data.
  - Allow scalability and fault tolerance through Azure replication features.

  This storage can be integrated with the logging service (`scapp-logs`) and mounted into Docker containers using Azure Files or volume drivers like Docker `cloudstor`.

- **Scalability and Reusability:** The template is parameterized, allowing flexibility for different environments. For example, you can scale the number of VMs or adjust configurations without changing the core template logic.

The ARM template defines all the necessary resources to deploy the Docker Swarm cluster with minimal effort while ensuring compliance with the required Azure configurations.

For a detailed visual representation, refer to Figure 5, which illustrates the structure and logic of the ARM template used in this project.

7 Figures Table

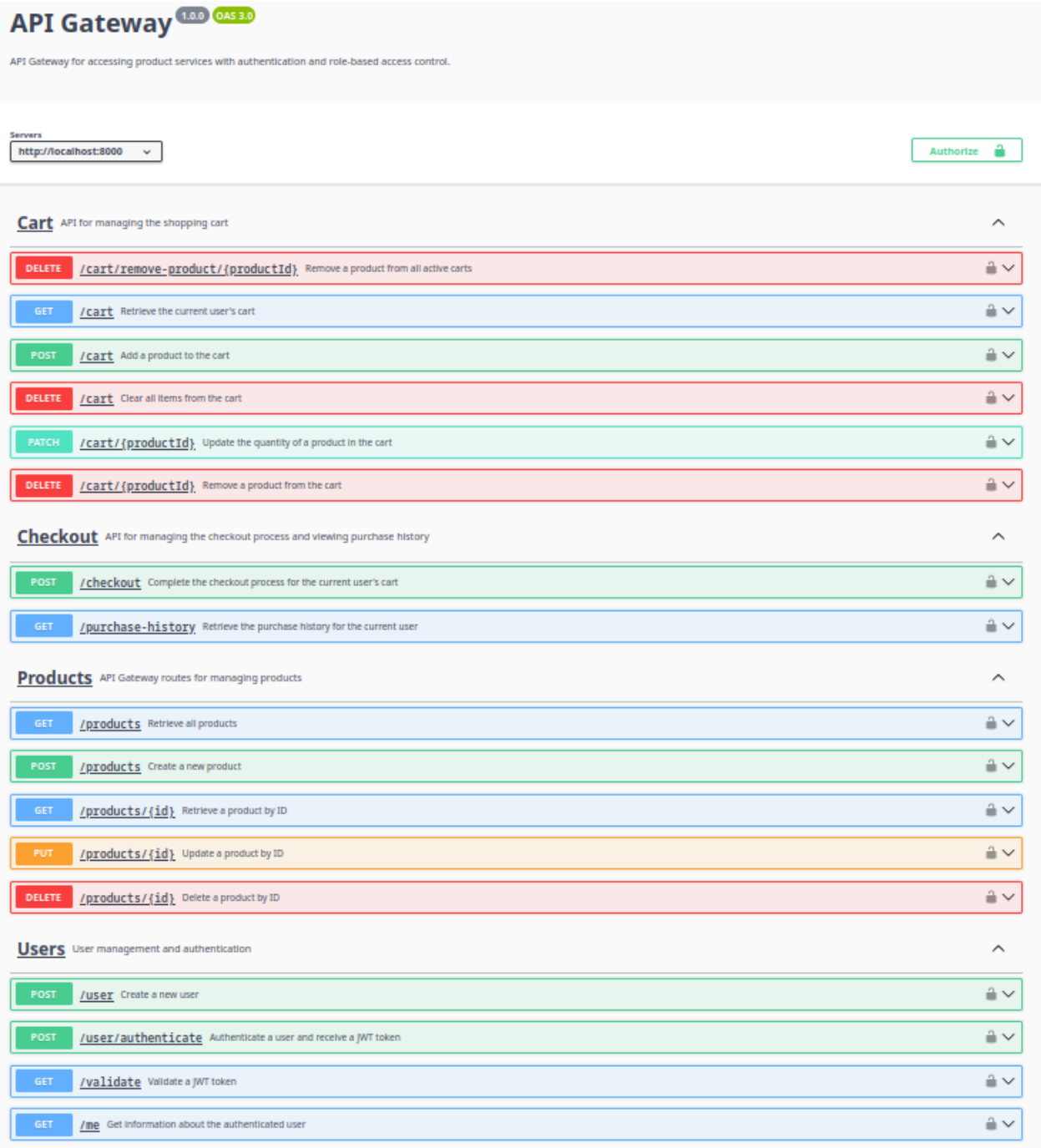

Figure 1: **Swagger API Documentation**

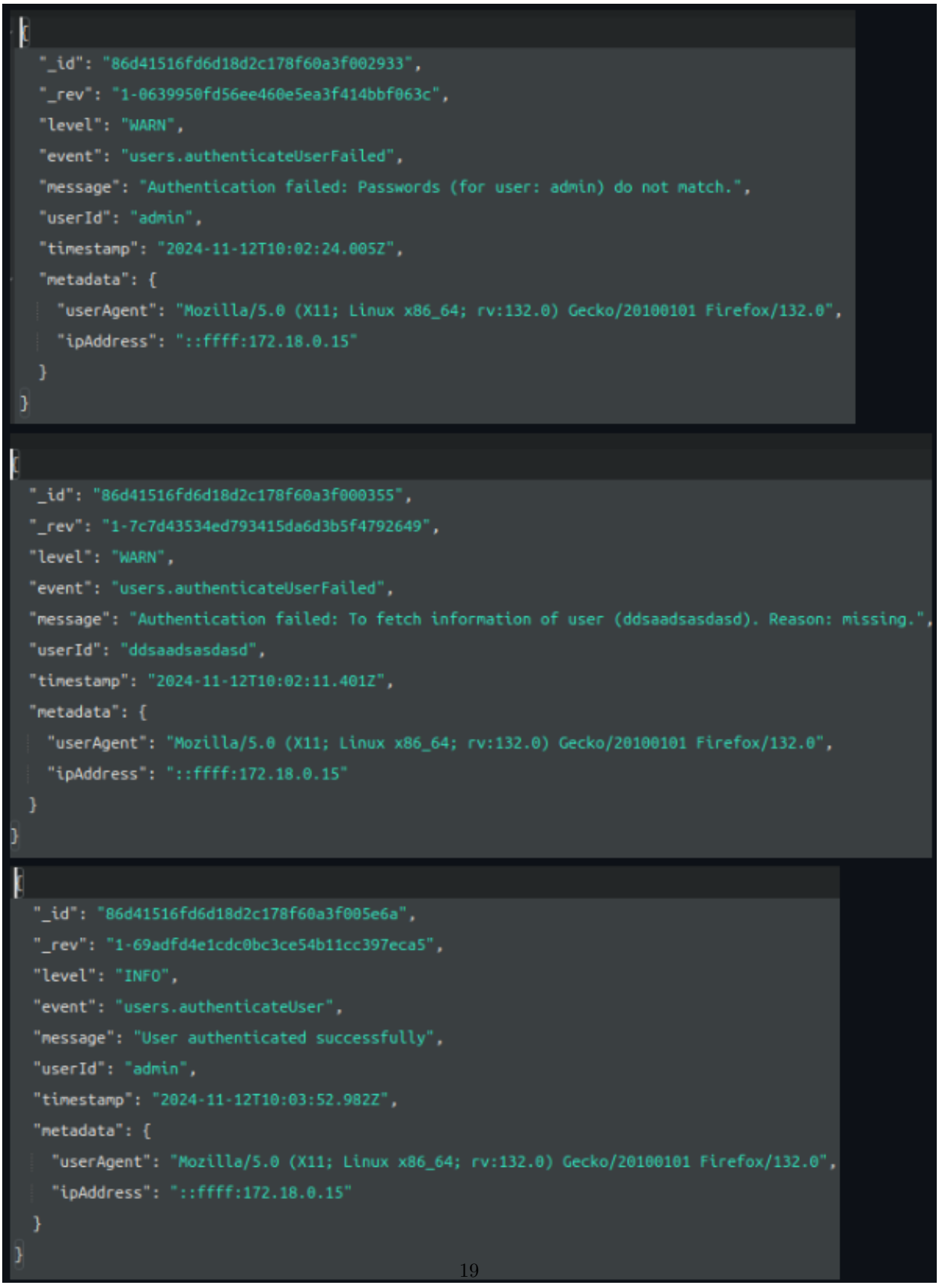

Figure 2: **Logs Sample of authentication service**

```
/*
 * Cart Notification Utility
 * This file provides a utility function to notify the cart service about product deletions.
 * It is used to ensure that products removed from the products service are also removed from all active carts.
 * Variables:
 *  - axios: HTTP client for making requests to the cart service.
 */

const axios = require('axios');

/*
 * notifyProductDeletion - Notifies the cart service about a product deletion.
 * Steps:
 *  - Validates the presence of an admin token.
 *  - Sends a DELETE request to the cart service to remove the product from all carts.
 * Logs an error if the request fails.
 * Parameters:
 *  - productId: ID of the product to be removed from all carts.
 *  - adminToken: Authorization token for authenticating with the cart service.
 */
async function notifyProductDeletion(productId, adminToken) {
    if (!adminToken) {
        console.error('Admin token is undefined. Cannot proceed with notifying the cart service.');
        return;
    }

    try {
        const response = await axios.delete(`http://scapp-cart:4500/cart/remove-product/${productId}`, {
            headers: {
                Authorization: adminToken,
            },
        });
        console.log(`Product ${productId} deletion notification sent to cart service. Response:`, response.data);
    } catch (error) {
        console.error('Failed to notify cart service of product deletion:', {
            message: error.message,
            response: error.response ? error.response.data : 'No response from cart service',
        });
    }
}

module.exports = { notifyProductDeletion };
```

Figure 3: **Implementation of cart-notify.js in scapp-products**

```
/*
 * removeProductFromActiveCarts - Removes a product from all active carts.
 * This is triggered when a product is deleted from the product catalog.
 * Parameters:
 *  - productId: The ID of the product to be removed from all active carts.
 * Returns a message indicating the result of the operation.
 */
async function removeProductFromActiveCarts(productId) {
    try {
        console.log(`Attempting to remove product ${productId} from all active
            carts.`);

        const carts = await cartDb.find({ selector: { status: 'active' } });

        for (const cart of carts.docs) {
            const initialLength = cart.items.length;
            cart.items = cart.items.filter(item => item.product_id !== productId);

            if (cart.items.length !== initialLength) {
                cart.totalPrice = cart.items.reduce((total, item) => total + item.
                    price * item.quantity, 0);
                const updatedCart = await cartDb.insert({ ...cart, _rev: cart._rev
                    });
                console.log(`Product ${productId} removed from cart ${cart._id}.
                    Updated cart revision: ${updatedCart.rev}`);
            } else {
                console.log(`Product ${productId} was not found in cart ${cart._id
                    }.`);
            }
        }

        return { message: `Product ${productId} removed from all active carts.` };
    } catch (error) {
        console.error('Failed to remove product from carts:', error.stack || error
            .message);
        throw new Error('Failed to remove product from carts: ' + error.message);
    }
}
```

Figure 4: **Implementation of crud-cart.js in scapp-cart**

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/
     deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "location": {
      "type": "string",
      "defaultValue": "[resourceGroup().location]",
      "metadata": {
        "description": "Location for all resources."
      }
    },
    "vnetName": {
      "type": "string",
      "defaultValue": "SwarmVNet",
      "metadata": {
        "description": "Name of the Virtual Network."
      }
    },
    "subnetName": {
      "type": "string",
      "defaultValue": "SwarmSubnet",
      "metadata": {
        "description": "Name of the Subnet."
      }
    },
    "addressPrefix": {
      "type": "string",
      "defaultValue": "10.0.0.0/16",
      "metadata": {
        "description": "Address space for the VNet."
      }
    },
    "subnetPrefix": {
      "type": "string",
      "defaultValue": "10.0.0.0/24",
      "metadata": {
        "description": "Subnet address range."
      }
    },
    "vmSize": {
      "type": "string",
      "defaultValue": "Standard_B1ms",
      "metadata": {
        "description": "VM size for the cluster nodes."
      }
    },
    "adminUsername": {
      "type": "string",
      "defaultValue": "azureuser",
      "metadata": {
        "description": "Admin username for VM access."
      }
    },
    "adminPassword": {
      "type": "securestring",
      "metadata": {
        "description": "Admin password for VM access."
      }
    },
    "storageAccountName": {
      "type": "string",
      "defaultValue": "scapplogsblob",
      "metadata": {
        "description": "Name of the Azure Blob Storage account."
```

```
    {
      "type": "Microsoft.Compute/virtualMachines",
      "apiVersion": "2020-12-01",
      "name": "SwarmManager",
      "location": "[parameters('location')]",
      "properties": {
        "hardwareProfile": {
          "vmSize": "[parameters('vmSize')]"
        },
        "osProfile": {
          "computerName": "SwarmManager",
          "adminUsername": "[parameters('adminUsername')]",
          "adminPassword": "[parameters('adminPassword')]"
        },
        "networkProfile": {
          "networkInterfaces": [
            {
              "id": "[resourceId('Microsoft.Network/networkInterfaces', '
                  SwarmManagerNIC')]"
            }
          ]
        },
        "storageProfile": {
          "imageReference": {
            "publisher": "Canonical",
            "offer": "UbuntuServer",
            "sku": "20_04-lts-gen2",
            "version": "latest"
          },
          "osDisk": {
            "createOption": "FromImage"
          }
        }
      }
    },
    {
      "type": "Microsoft.Storage/storageAccounts",
      "apiVersion": "2021-02-01",
      "name": "[parameters('storageAccountName')]",
      "location": "[parameters('location')]",
      "sku": {
        "name": "Standard_LRS"
      },
      "kind": "StorageV2",
      "properties": {
        "accessTier": "Hot"
      }
    }
  ]
}
```

Figure 6: **ARM Template for Automating Docker Swarm Cluster Deployment (Page 2)**

```
    {
      "type": "Microsoft.Network/networkSecurityGroups",
      "apiVersion": "2020-06-01",
      "name": "SwarmNSG",
      "location": "[parameters('location')]",
      "properties": {
        "securityRules": [
          {
            "name": "AllowSwarmManager",
            "properties": {
              "priority": 100,
              "protocol": "Tcp",
              "sourcePortRange": "*",
              "destinationPortRange": "2377",
              "sourceAddressPrefix": "VirtualNetwork",
              "destinationAddressPrefix": "*",
              "access": "Allow",
              "direction": "Inbound"
            }
          },
          {
            "name": "AllowOverlayNetwork",
            "properties": {
              "priority": 200,
              "protocol": "Udp",
              "sourcePortRange": "*",
              "destinationPortRange": "4789",
              "sourceAddressPrefix": "VirtualNetwork",
              "destinationAddressPrefix": "*",
              "access": "Allow",
              "direction": "Inbound"
            }
          },
          {
            "name": "AllowGateway",
            "properties": {
              "priority": 300,
              "protocol": "Tcp",
              "sourcePortRange": "*",
              "destinationPortRange": "8000",
              "sourceAddressPrefix": "*",
              "destinationAddressPrefix": "*",
              "access": "Allow",
              "direction": "Inbound"
            }
          },
          {
            "name": "DenyAllInbound",
            "properties": {
              "priority": 4096,
              "protocol": "*",
              "sourcePortRange": "*",
              "destinationPortRange": "*",
              "sourceAddressPrefix": "*",
              "destinationAddressPrefix": "*",
              "access": "Deny",
              "direction": "Inbound"
            }
          }
        ]
      }
    }
```


Figure 7: **ARM Template for Automating Docker Swarm Cluster Deployment (Page 3)**