E-Commerce Docker

Overview

In this advanced assignment, you will develop a containerized e-commerce microservices platform using Docker. This project will test your ability to design, implement, and orchestrate a complex system of interconnected services. You'll be working with multiple APIs, a database, and an API gateway, all containerized and working together seamlessly.

Objective

Create a fully functional, containerized e-commerce platform consisting of an API Gateway, three supporting microservices, and a PostgreSQL database. This system should demonstrate your proficiency in Docker, microservices architecture, and backend development.

Components

1. Web Application
2. API Gateway
3. Product Catalog API
4. Order Management API
5. PostgreSQL Database

Template

This assignment comes with a template containing a single web application. The web application is a Next application which needs to be configured.

The .env.local file consists of the following and all values can be found within Auth0. In order to retrieve these values in Auth0, a regular web application must be created.

```
AUTH0_SECRET='openssl rand -hex 32'
AUTH0_BASE_URL=
AUTH0_ISSUER_BASE_URL=
AUTH0_CLIENT_ID=
AUTH0_CLIENT_SECRET=
AUTH0_AUDIENCE=<unique-identifier-of-api-gateway>
AUTH0_SCOPE=openid profile email offline_access
```

After the file above has been successfully populated the following steps must be taken:

1. Run `npm install` in the root of the web application
2. Run `npm run dev` in the root of the web application
3. This will run the application on port 3000 - therefore can be accessed through http://localhost:3000
4. The web application makes use of Auth0 and depending on how Auth0 is configured - it can allow multiple ways to login
5. The preferred way to login should be using Google authentication

Detailed Requirements

Authentication

- The initial authentication request to the API Gateway should make use of a Bearer token generated by the web project within the template. The web project uses authentication with Google and the token can be copied from the web project and used for subsequent requests to the API gateway.
- All interconnected API calls should make use of Machine-to-Machine (M2M) authentication (see preferred documentation below)
- All claims from the initial authenticated request to the API Gateway should be forwarded to subsequent requests for internal services. This can be done using triggers and actions in Auth0 (see preferred documentation below)
- Auth0 should be used as the IdP for M2M

Post-Login trigger

```
exports.onExecutePostLogin = async (event, api) => {
  api.accessToken.setCustomClaim("name", event.user.given_name);
  api.accessToken.setCustomClaim("email_address", event.user.email);
};
```

The code seen above is a claim forwarding custom action which can be used to attach custom claims to the access token. The access token is then retrieved by the API which can make use of these claims.

In order for the custom action to be applied, a Post-Login trigger must be created within Auth0 to enable the action.

M2M Client Credentials trigger

```
exports.onExecuteCredentialsExchange = async (event, api) => {
  api.accessToken.setCustomClaim("name", event.request.body['name']);
```

```
  api.accessToken.setCustomClaim("email_address",
event.request.body['email_address']);
};
```

The code seen above is a claim forwarding custom action which can be used to attach custom claims to the access token during M2M authentication.

In order for the custom action to be applied, a Credentials Exchange trigger must be created within Auth0 to enable the action.

API Gateway

- API Gateway should be a C# Web API using .NET
- All Product Catalog endpoints should be available
- All Order Management endpoints should be available
- A Dockerfile should be setup for the API Gateway

Product Catalog

- Product Catalog should be a C# Web API using .NET
- The Product Catalog should make use of the PostgreSQL database setup within the Docker network to either write or read from
- The following endpoints should be available:
    - GET /api/products - List products

```
[
  {
    "id": "number",
    "name": "string",
    "description": "string",
    "price": "number",
    "categoryId": "number"
  }
]
```

    - GET /api/products/{id} - Get a specific product

```
{
  "id": "number",
  "name": "string",
  "description": "string",
  "price": "number",
  "categoryId": "number"
```

```
}
```

- POST /api/products - Create a new product

```
{
  "name": "string",
  "description": "string",
  "price": "number",
  "categoryId": "number"
}
```

- PUT /api/products/{id} - Update a product

```
{
  "name": "string",
  "description": "string",
  "price": "number",
  "categoryId": "number"
}
```

- DELETE /api/products/{id} - Delete a product
- GET /api/categories - List categories

```
[
  {
    "id": "number",
    "name": "string",
    "description": "string"
  }
]
```

- POST /api/categories - Create a new category

```
{
  "name": "string",
  "description": "string"
}
```

- PUT /api/categories/{id} - Update a category

```
{
  "name": "string",
  "description": "string"
```

```
}
```

- DELETE /api/categories/{id} - Delete a category

Order Management

- Order Management should be a C# Web API using .NET
- The Order Management should make use of the PostgreSQL database setup within the Docker network to either write or read from
- The following endpoints should be available:
  - POST /api/orders - Create a new order

```
{
  "emailAddress": "string",
  "items": [
    {
      "productId": "number",
      "quantity": "number"
    }
  ],
  "shippingAddress": {
    "street": "string",
    "city": "string",
    "state": "string",
    "zipCode": "string",
    "country": "string"
  }
}
```

  - GET /api/orders/{id} - Get a specific order

```
{
  "id": "number",
  "emailAddress": "string",
  "items": [
    {
      "productId": "number",
      "productName": "string",
      "quantity": "number",
      "unitPrice": "number"
    }
  ],
```

```
  "totalAmount": "number",
  "status": "string",
  "shippingAddress": {
    "street": "string",
    "city": "string",
    "state": "string",
    "zipCode": "string",
    "country": "string"
  },
  "createdAt": "string (ISO 8601 date)",
  "updatedAt": "string (ISO 8601 date)"
}
```

- GET /api/orders - List orders

```
[
  {
    "id": "number",
    "emailAddress": "string",
    "totalAmount": "number",
    "status": "string",
    "createdAt": "string (ISO 8601 date)"
  }
]
```

- GET /api/orders/users/{username} - Lists orders for a specific user

```
[
  {
    "id": "number",
    "totalAmount": "number",
    "status": "string",
    "createdAt": "string (ISO 8601 date)"
  }
]
```

PostgreSQL

- A PostgreSQL database should be setup which all services make use of
- The entity models should be setup according to the input models, and or DTOs seen in the response above
- The database should make use of the default PostgreSQL Docker image (https://hub.docker.com/_/postgres)

Docker compose

- Create a docker-compose.yml file to orchestrate all services
- Configure networking between containers
- Set up volume for PostgreSQL data persistence

Documentation

M2M Authentication

- https://auth0.com/features/machine-to-machine
- https://auth0.com/blog/using-m2m-authorization/

Triggers and Actions

- https://auth0.com/docs/customize/actions/actions-overview
- https://auth0.com/docs/customize/actions/write-your-first-action
- https://auth0.com/docs/customize/actions/explore-triggers
- https://auth0.com/docs/customize/actions/explore-triggers/signup-and-login-triggers/login-trigger
- https://auth0.com/docs/customize/actions/use-cases