.NET Microservices – Azure DevOps and AKS

Section 7: Sync Microservice Communication - Notes

Microservice Communication Patterns

In microservice architecture, services are typically deployed independently and communicate with each other to accomplish business goals. One common way for microservices to interact is through **HTTP-based communication**. Communication between microservices can generally be categorized into two main patterns:

1. **Synchronous Communication (Sync)**
2. **Asynchronous Communication (Async)**

Both patterns have their own advantages, challenges, and specific scenarios where they are more suited.

Synchronous Communication (Sync)

Definition:
Synchronous communication means that the client service sends a request to the server service and waits for the response before proceeding further. The process is "blocking" in nature, i.e., the client service is halted until a response is received.

The most common protocol for synchronous communication in microservices is **HTTP** with **REST APIs**, although other protocols like **gRPC** are also used.

Advantages of Synchronous Communication

1. **Simple and Intuitive**:
   Synchronous communication is easy to understand and implement. It's very similar to how traditional applications communicate using HTTP/REST calls.
2. **Immediate Feedback**:
   Clients receive an immediate response, allowing them to make decisions based on real-time data.
3. **Consistency**:
   Since a response is required immediately, there is often a higher level of consistency in the state of the system because the client waits for a completed operation before continuing.
4. **Suitable for Request-Response Workflows**:
   If the interaction involves asking for data or requesting the completion of an operation with immediate feedback (e.g., retrieving user details, processing a payment), synchronous communication is ideal.

Challenges of Synchronous Communication

1. **Blocking Nature**:
   The client is blocked while waiting for a response, which can lead to poor performance if the server is slow or unresponsive.
2. **Tight Coupling**:
   Synchronous communication creates tighter coupling between services because one service's availability and response time directly affects the calling service.
3. **Scalability Issues**:
   If the number of calls grows, synchronous communication can cause bottlenecks. If the dependent service has a slowdown, it can ripple through the entire system.
4. **Timeouts and Reliability**:
   Network failures or timeouts can become problematic. A failure in one microservice can cascade into the failure of other services if they depend on a synchronous response.

Scenarios Where Synchronous Communication is Best

- **Real-Time Data Requests**:
  When you need an immediate response to proceed with further actions. For example, when you need to retrieve details from an inventory service before placing an order.
- **Client-Facing Operations**:
  When the client expects a response in real time, such as web or mobile applications that require data from a microservice (e.g., showing the current balance in a banking app).
- **Read Operations**:
  For operations that primarily involve retrieving data without complex or long-running processes.

Asynchronous Communication (Async)

Definition:
Asynchronous communication is non-blocking. The client service sends a request and does not wait for a response. The server service processes the request independently, and the client can continue its own workflow. Responses are usually sent via callbacks, message queues, or event streams.

Common implementations of asynchronous communication include:

- **Message Brokers** like RabbitMQ, Kafka, or Azure Service Bus
- **Event-Driven Architectures** using Event Sourcing

Advantages of Asynchronous Communication

1. **Decoupling**:
   Asynchronous communication allows services to be more loosely coupled. Since services do not have to wait for one another, they can be more independent.

2. **Scalability**:
   Asynchronous systems can handle a large number of requests without creating bottlenecks. The decoupled nature allows each service to scale independently.

3. **Non-Blocking**:
   The client is not blocked while waiting for a response, making the system more efficient, especially when dealing with long-running tasks (e.g., order fulfillment, background processing).

4. **Resilience and Fault Tolerance**:
   Even if one microservice is temporarily unavailable, requests can be queued and processed later. This ensures the system remains operational even in the face of partial failures.

Challenges of Asynchronous Communication

1. **Complexity**:
   Handling asynchronous workflows can be more complex than synchronous communication. You have to manage message queues, callbacks, and retries, making error handling more difficult.

2. **Eventual Consistency**:
   Asynchronous systems are typically **eventually consistent**. This means that while data will be consistent over time, there may be temporary discrepancies across services, which could lead to issues in highly transactional systems.

3. **Latency**:
   Since the response is not immediate, there can be delays in processing. This may not be acceptable in scenarios where real-time feedback is required.

4. **Monitoring and Debugging**:
   It can be harder to monitor and trace asynchronous calls, especially in distributed systems, where messages may be processed at different times and locations.

Scenarios Where Asynchronous Communication is Best

- **Long-Running Processes**:
  For workflows that require a lot of processing time (e.g., batch jobs, order fulfillment, or report generation), asynchronous communication allows services to handle tasks in the background.
- **Event-Driven Systems**:
  Systems built on an event-driven architecture where actions are triggered by certain events (e.g., sending an email after an order is placed) are best suited for asynchronous communication.
- **High Availability and Scalability**:
  When services need to handle a high volume of requests, asynchronous patterns can scale more efficiently. For example, in a system where hundreds of thousands of requests are placed, asynchronous message queues can handle the load without causing delays.

Comparison: Synchronous vs Asynchronous Microservice Communication

1. Definition

- **Synchronous Communication**: This method requires the services to wait for a response from the other service before continuing with their own processing. Typically, HTTP/REST and gRPC are used for synchronous communication.
- **Asynchronous Communication**: In this method, services do not wait for a response from the other service. They continue processing and handle responses or notifications when they are available. This often uses message queues, such as RabbitMQ or Kafka.

2. Communication Flow

- **Synchronous Communication**: The client sends a request to the server and waits for the server to process the request and send back a response. The client is blocked until the response is received.
- **Asynchronous Communication**: The client sends a request to the server and proceeds without waiting for a response. The server processes the request and sends a notification or response back to the client when it is ready, which can be handled through callbacks or message queues.

3. Performance

- **Synchronous Communication**: Performance can be affected by the response time of the server. If the server is slow, the client's performance and throughput are impacted as it has to wait.
- **Asynchronous Communication**: Generally improves performance and throughput, as the client does not have to wait for the server's response and can handle multiple requests or perform other tasks in parallel.

4. Complexity

- **Synchronous Communication**: Easier to implement and understand because it follows a straightforward request-response model.
- **Asynchronous Communication**: More complex to implement due to the need for additional components such as message brokers, and managing the state of the communication and handling retries or failures.

5. Scalability

- **Synchronous Communication**: Scalability can be challenging as the system needs to handle a large number of simultaneous requests and responses. Scaling often involves increasing server capacity or implementing load balancing.
- **Asynchronous Communication**: More scalable as it decouples request processing from response handling. It allows the system to handle high loads more effectively and can process messages at different rates without affecting the client's performance.

6. Error Handling

- **Synchronous Communication**: Errors are typically handled immediately by sending an error response to the client. This can be straightforward but may impact the user experience if not managed properly.
- **Asynchronous Communication**: Error handling can be more complex, involving mechanisms for retries, error queues, or compensating actions. However, it can provide a more resilient system that can recover from transient issues.

7. Use Cases

- **Synchronous Communication**: Suitable for scenarios where real-time responses are critical, such as user interactions, financial transactions, or data retrieval operations where immediate feedback is required.
- **Asynchronous Communication**: Ideal for scenarios where tasks can be processed independently and do not require immediate responses, such as background jobs, notifications, or event-driven architectures.

Synchronous communication is straightforward and suitable for real-time interactions but can suffer from performance bottlenecks. Asynchronous communication, while more complex, offers better performance and scalability, making it suitable for high-load and decoupled systems.

Key Points for Interview Preparation

1. **Synchronous Communication**:
    - Blocks the client until a response is received.
    - Ideal for real-time, request-response workflows.

    - Simpler but leads to tight coupling between services.
    - Introduces challenges with scalability and fault tolerance.
2. **Asynchronous Communication**:
    - Non-blocking, allowing services to work independently.
    - Suitable for long-running tasks and event-driven architectures.
    - More complex to implement but highly scalable and fault-tolerant.
    - Introduces the concept of eventual consistency, which can be a challenge in transactional systems.
3. **Decoupling and Scalability**:
    - Async communication allows for more decoupling between services, which is essential for scaling large systems.
4. **Eventual Consistency**:
    - Be prepared to discuss how to handle eventual consistency and its implications on the business logic.
5. **Trade-offs**:
    - Understanding when to use sync vs async is crucial. In an interview, emphasize the trade-offs between simplicity and flexibility, performance, and consistency.

Docker Compose Launch Profiles in Visual Studio

When developing microservices, especially with Docker, Visual Studio provides built-in support for managing and running multiple services using Docker Compose. A **Docker Compose launch profile** helps you configure how your services are built, run, and tested during development.

This section covers the following key aspects:

- Overview of Docker Compose Launch Profiles in Visual Studio
- Step-by-step guide to creating and using launch profiles
- Configuration options in launchSettings.json
- Best practices for using Docker Compose in Visual Studio
- Key points for interview preparation

1. Overview of Docker Compose Launch Profiles

A **launch profile** in Visual Studio allows developers to specify how a microservice should be launched for debugging and testing purposes. When working with multiple microservices, using Docker Compose helps spin up and coordinate multiple containers seamlessly. Visual Studio integrates with Docker Compose to provide a streamlined way to:

- Build, run, and debug multiple microservices together.
- Set up environment variables, ports, and volumes for individual services.
- Simulate a production-like environment locally.

The **launchSettings.json** file defines how Visual Studio should handle the project during the debugging phase. This file includes important configurations like the launch profile (which defines the environment, startup URL, and more), as well as settings for each microservice that Docker Compose will manage.

2. Creating and Using Docker Compose Launch Profiles

Step 1: Add Docker Support to Your Project

1. **Right-click on your solution** in Visual Studio and select **"Add" > "Container Orchestrator Support"**.
2. Choose **Docker Compose** as your orchestration tool. Visual Studio will automatically generate a docker-compose.yml file and integrate Docker support into your project.

Step 2: Create a Launch Profile

1. After adding Docker Compose support, Visual Studio will generate a **docker-compose launch profile** inside the launchSettings.json file.
2. You can view and modify this file by navigating to the **Properties** folder of the main project.

Step 3: Configure the Launch Profile

In the launchSettings.json, you'll see a "docker-compose" profile that looks something like this:

```
{
  "profiles": {
    "docker-compose": {
      "commandName": "DockerCompose",
      "serviceName": "webapi",
      "dockerComposeFilePath": "docker-compose.yml",
```

```
    "launchBrowser": true,
    "launchUrl": "{Scheme}://{ServiceHost}:{ServicePort}",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
   }
  }
}
```

- **commandName**: Defines how Visual Studio will launch the project. For Docker Compose, it should always be "DockerCompose".
- **serviceName**: This is the name of the service specified in the docker-compose.yml file. Visual Studio will run this service when you start the project.
- **dockerComposeFilePath**: Specifies the path to the docker-compose.yml file that defines how services are built and run.
- **launchBrowser**: If set to true, Visual Studio will automatically launch a web browser when the application starts.
- **launchUrl**: Specifies the URL that the browser should navigate to when the application launches.
- **environmentVariables**: Allows you to set environment variables for the application (such as ASPNETCORE_ENVIRONMENT).

Step 4: Customize the Docker Compose File

You can configure your services in the docker-compose.yml file, which defines how Docker should build and run your microservices. Here's a basic example:

```
version: '3.8'
services:
  products-microservice:
    image: harshamicroservices/ecommerce-products-microservice:v1.0
    environment:
      - MYSQL_HOST=mysql-host
      - MYSQL_PASSWORD=admin
    ports:
      - "8080:8080"
```

```
  users-microservice:
    image: harshamicroservices/ecommerce-users-microservice:v1.0
    environment:
      - POSTGRES_HOST=postgres-host
      - POSTGRES_PASSWORD=admin
    ports:
      - "9090:9090"
```

Visual Studio will use this Docker Compose file to coordinate the two microservices (products-microservice and users-microservice). The docker-compose launch profile ensures that both services are up and running when you start the solution in Visual Studio.

3. Configuration Options in launchSettings.json

When working with Docker Compose in Visual Studio, the launchSettings.json file plays a critical role in defining the development environment. Let's dive into some important properties:

- **commandName**:
  The commandName for Docker Compose should always be "DockerCompose", which tells Visual Studio to use Docker Compose as the execution engine.
- **serviceName**:
  Defines which service from the docker-compose.yml file should be the primary service for debugging. For example, "serviceName": "products-microservice" means that the products-microservice service will be launched and debugged.
- **launchBrowser**:
  This boolean flag determines whether Visual Studio should automatically launch the default browser after starting the service. Set this to false if you don't want the browser to open automatically.
- **launchUrl**:
  Defines the URL that Visual Studio should open when the application is launched. Placeholders like {Scheme}, {ServiceHost}, and {ServicePort} allow for dynamic URL generation.
- **environmentVariables**:
  Set environment variables for your service here. This is especially useful for configuring the environment (e.g., Development, Production) or any other custom environment variables your service requires (like database credentials).

4. Best Practices for Using Docker Compose in Visual Studio

1. **Use Separate Compose Files for Development and Production**:
   Keep separate docker-compose.yml files for development and production environments. For instance, use docker-compose.override.yml for development settings such as mounting local volumes for code hot-reloading, and reserve docker-compose.yml for production-like configurations.
2. **Leverage Debugging with Docker Compose**:
   Visual Studio integrates with Docker to provide debugging capabilities for multiple services. This means you can set breakpoints, inspect variables, and step through code across microservices.
3. **Maintain Environment-Specific Settings**:
   Use environment variables to manage environment-specific settings, like database connection strings and API keys. These can be specified in the launchSettings.json or passed directly through the docker-compose.yml file.
4. **Optimize for Rebuilds**:
   Make sure to structure your Dockerfile in a way that minimizes rebuild times. For example, place frequently changing files like source code lower in the file and install dependencies first to leverage Docker's caching mechanism.

Key Points for Interview Preparation

1. **Launch Profiles**:
   - A launch profile defines how Visual Studio runs your application (including environment variables, ports, and URLs).
   - Docker Compose launch profiles are used when multiple microservices need to be run together.
2. **launchSettings.json**:
   - Contains key configurations for running the service in Visual Studio. It defines the commandName, serviceName, launchBrowser, and environment variables.
   - The "DockerCompose" command triggers the execution of services defined in docker-compose.yml.
3. **Visual Studio Integration**:
   - Visual Studio provides tight integration with Docker Compose, allowing you to manage and debug multiple services seamlessly.
4. **Environment Variables**:
   - Environment variables defined in launchSettings.json or the docker-compose.yml file allow you to configure microservices for different environments (e.g., Dev, QA, Production).

5. **Service Coordination**:
    - Docker Compose enables you to run, test, and debug multiple microservices together, simulating real-world production environments locally.

Docker Compose in Visual Studio - Code Explanation

```
version: '3.4'


services:
  ordersmicroservice.api:
    image: ordersmicroserviceapi
    build:
      context: .
      dockerfile: OrdersMicroservice.API/Dockerfile
    environment:
     - MONGODB_HOST=mongodb-container
     - MONGODB_PORT=27017
     - MONGODB_DATABASE=OrdersDatabase
     - UsersMicroserviceName=users-microservice
     - UsersMicroservicePort=9090
     - ProductsMicroserviceName=products-microservice
     - ProductsMicroservicePort=8080
    ports:
     - "7000:8080"
    networks:
     - orders-mongodb-network
     - ecommerce-network
    depends_on:
     - mongodb-container
```

```
mongodb-container:
 image: mongo:latest
 ports:
  - "27017:27017"
 volumes:
  - ../mongodb-init:/docker-entrypoint-initdb.d
 networks:
  - orders-mongodb-network


products-microservice:
 image: products-microservice:latest
 environment:
  - ASPNETCORE_HTTP_PORTS=8080
  - ASPNETCORE_ENVIRONMENT=Development
  - MYSQL_HOST=mysql-container
  - MYSQL_PORT=3306
  - MYSQL_DATABASE=ecommerceproductsdatabase
  - MYSQL_USER=root
  - MYSQL_PASSWORD=admin
 ports:
  - "6000:8080"
 networks:
  - products-mysql-network
  - ecommerce-network
 depends_on:
  - mysql-container


mysql-container:
```

```
 image: mysql:8.0
 environment:
  - MYSQL_ROOT_PASSWORD=admin
 ports:
  - "3307:3306"
 volumes:
  - ../mysql-init:/docker-entrypoint-initdb.d
 networks:
  - products-mysql-network



users-microservice:
 image: users-microservice:latest
 environment:
  - ASPNETCORE_HTTP_PORTS=8080
  - ASPNETCORE_HTTP_ENVIRONMENT=Development
  - POSTGRES_HOST=postgres-container
  - POSTGRES_PORT=5432
  - POSTGRES_DATABASE=eCommerceUsers
  - POSTGRES_USER=postgres
  - POSTGRES_PASSWORD=admin
 ports:
  - "5000:9090"
 networks:
  - users-postgres-network
  - ecommerce-network
 depends_on:
  - postgres-container



postgres-container:
```

```
  image: postgres:13
  environment:
   - POSTGRES_USER=postgres
   - POSTGRES_PASSWORD=admin
   - POSTGRES_DB=eCommerceUsers
  ports:
   - "5433:5432"
  volumes:
   - ../postgres-init:/docker-entrypoint-initdb.d
  networks:
   - users-postgres-network



networks:
 orders-mongodb-network:
  driver: bridge
 products-mysql-network:
  driver: bridge
 users-postgres-network:
  driver: bridge
 ecommerce-network:
  driver: bridge
```

1. Version of Docker Compose

version: '3.4'

- **Explanation**: This specifies the version of the Docker Compose file format. Version 3.4 is used to define services, networks, and volumes. It is compatible with the latest features, making it a good choice for modern applications like microservices.

2. Services Definition

This section defines the microservices and their configurations.

Orders Microservice

```
services:
  ordersmicroservice.api:
    image: ordersmicroserviceapi
    build:
      context: .
      dockerfile: OrdersMicroservice.API/Dockerfile
```

- **Explanation**:
    - services: This block defines each service (or container) that will be launched.
    - ordersmicroservice.api: The name of the service for the Orders Microservice.
    - image: The Docker image that will be used for the service. This can either point to an existing image (e.g., from Docker Hub) or be built from the source code.
    - build: Specifies the build context and Dockerfile.
        - context: The current directory (.) is used as the build context. This means the source files for the Docker image are located in the current directory.
        - dockerfile: This points to the path of the Dockerfile that will be used to build the image (OrdersMicroservice.API/Dockerfile).

Environment Variables

```
  environment:
    - MONGODB_HOST=mongodb-container
    - MONGODB_PORT=27017
    - MONGODB_DATABASE=OrdersDatabase
    - UsersMicroserviceName=users-microservice
    - UsersMicroservicePort=9090
    - ProductsMicroserviceName=products-microservice
    - ProductsMicroservicePort=8080
```

- **Explanation**: The environment block sets environment variables for the service, which are passed to the application running inside the container.
    - MONGODB_HOST: The hostname for the MongoDB service. This is the name of the service (mongodb-container), which allows the Orders Microservice to connect to MongoDB.

- MONGODB_PORT: The port MongoDB will use.
- MONGODB_DATABASE: The name of the database to connect to.
- UsersMicroserviceName, UsersMicroservicePort: The name and port for the Users Microservice.
- ProductsMicroserviceName, ProductsMicroservicePort: The name and port for the Products Microservice.
- These variables allow the Orders Microservice to communicate with the MongoDB database and other microservices.

Ports Configuration

```
ports:
  - "7000:8080"
```

- **Explanation**:
  - This exposes port 8080 (the internal port of the container where the application runs) to port 7000 on the host machine. It allows external traffic to access the Orders Microservice via http://localhost:7000.

Networks and Dependencies

```
networks:
  - orders-mongodb-network
  - ecommerce-network
depends_on:
  - mongodb-container
```

- **Explanation**:
  - networks: Specifies the networks the service is connected to. Here, the Orders Microservice is connected to two networks:
    - orders-mongodb-network: To communicate with the MongoDB container.
    - ecommerce-network: To communicate with other services in the eCommerce system.
  - depends_on: This indicates that the Orders Microservice depends on the mongodb-container. Docker Compose ensures that the MongoDB container is up and running before the Orders Microservice starts.

3. MongoDB Container

```
mongodb-container:
  image: mongo:latest
  ports:
   - "27017:27017"
  volumes:
   - ../mongodb-init:/docker-entrypoint-initdb.d
  networks:
   - orders-mongodb-network
```

- **Explanation**:
  - image: This pulls the latest MongoDB image from Docker Hub (mongo:latest).
  - ports: Maps MongoDB's internal port (27017) to the host machine's port (27017).
  - volumes: This mounts a local directory (../mongodb-init) to the MongoDB container at /docker-entrypoint-initdb.d. This is often used to initialize MongoDB with predefined scripts or data.
  - networks: Connects the MongoDB container to the orders-mongodb-network to allow communication between it and the Orders Microservice.

4. Products Microservice

```
products-microservice:
  image: products-microservice:latest
  environment:
   - ASPNETCORE_HTTP_PORTS=8080
   - ASPNETCORE_ENVIRONMENT=Development
   - MYSQL_HOST=mysql-container
   - MYSQL_PORT=3306
   - MYSQL_DATABASE=ecommerceproductsdatabase
   - MYSQL_USER=root
   - MYSQL_PASSWORD=admin
  ports:
```

- "6000:8080"

networks:

- products-mysql-network

- ecommerce-network

depends_on:

- mysql-container

- **Explanation**:
    - image: Specifies the Docker image for the Products Microservice.
    - environment: Defines environment variables such as the HTTP ports, environment (Development), and MySQL connection settings (MYSQL_HOST, MYSQL_PORT, etc.).
    - ports: Exposes port 8080 (internal) to port 6000 on the host.
    - networks: Connects this service to products-mysql-network and ecommerce-network.
    - depends_on: Ensures that the MySQL container is up and running before starting the Products Microservice.

5. MySQL Container

mysql-container:

image: mysql:8.0

environment:

- MYSQL_ROOT_PASSWORD=admin

ports:

- "3307:3306"

volumes:

- ../mysql-init:/docker-entrypoint-initdb.d

networks:

- products-mysql-network

- **Explanation**:
    - image: Pulls the MySQL 8.0 image from Docker Hub.
    - environment: Sets the root password for MySQL (MYSQL_ROOT_PASSWORD=admin).
    - ports: Maps port 3306 (MySQL's default port) to 3307 on the host.

- volumes: Mounts a local directory (../mysql-init) to initialize the database.
- networks: Connects the MySQL container to products-mysql-network.

6. Users Microservice

```
users-microservice:
  image: users-microservice:latest
  environment:
   - ASPNETCORE_HTTP_PORTS=8080
   - ASPNETCORE_HTTP_ENVIRONMENT=Development
   - POSTGRES_HOST=postgres-container
   - POSTGRES_PORT=5432
   - POSTGRES_DATABASE=eCommerceUsers
   - POSTGRES_USER=postgres
   - POSTGRES_PASSWORD=admin
  ports:
   - "5000:9090"
  networks:
   - users-postgres-network
   - ecommerce-network
  depends_on:
   - postgres-container
```

- **Explanation**:
  - image: Specifies the Docker image for the Users Microservice.
  - environment: Defines environment variables for ASP.NET Core and PostgreSQL connection details.
  - ports: Exposes port 9090 (internal) to port 5000 on the host.
  - networks: Connects to users-postgres-network and ecommerce-network.
  - depends_on: Ensures that the PostgreSQL container starts before the Users Microservice.

7. PostgreSQL Container

```
postgres-container:
  image: postgres:13
  environment:
   - POSTGRES_USER=postgres
   - POSTGRES_PASSWORD=admin
   - POSTGRES_DB=eCommerceUsers
  ports:
   - "5433:5432"
  volumes:
   - ../postgres-init:/docker-entrypoint-initdb.d
  networks:
   - users-postgres-network
```

- **Explanation**:
  - image: Uses the PostgreSQL 13 image from Docker Hub.
  - environment: Defines PostgreSQL user, password, and database.
  - ports: Maps PostgreSQL's port 5432 to port 5433 on the host.
  - volumes: Mounts initialization scripts for the database from a local directory.
  - networks: Connects to users-postgres-network.

8. Networks Definition

```
networks:
 orders-mongodb-network:
  driver: bridge
 products-mysql-network:
  driver: bridge
 users-postgres-network:
  driver: bridge
 ecommerce-network:
  driver: bridge
```

- **Explanation**:

- This section defines the networks used by the services.
- driver: bridge: Uses the bridge network driver, which allows containers to communicate with each other as if they are on the same network. This ensures services in different microservices can communicate efficiently.

Summary

This Docker Compose file defines a multi-container environment for an **eCommerce system** consisting of:

- Orders Microservice (with MongoDB).
- Products Microservice (with MySQL).
- Users Microservice (with PostgreSQL).