Lab 1: Web Proxy

goFOOD - Online Food Delivery System 🍟🛵

Application Suitability

1. **Complexity Handling**: Online food delivery systems are inherently complex, involving various components such as order management, restaurant selection, delivery logistics, and customer interfaces. Microservices allow for the effective management of this complexity by breaking it down into smaller, independent services that can be developed and maintained separately.

2. **Scalability**: The demand for food delivery services can fluctuate significantly, especially during peak hours or special events. Distributed systems with microservices architecture enable elastic scalability, allowing individual services to scale up or down as needed. For example, during busy lunch hours, the order processing service can scale independently to handle increased orders.

3. **Data Isolation and Privacy**: Protecting user data and order information is crucial in the food delivery industry. Microservices with separate databases for each service ensure data isolation and privacy, reducing the risk of data breaches. This aligns with the need to maintain customer trust and adhere to data protection regulations.

4. **Fault Tolerance and Resilience**: Implementing features like task timeouts and health monitoring with alerts enhances fault tolerance and system resilience. Microservices architecture enables fine-grained control over error handling and recovery, reducing downtime and ensuring a reliable user experience.

5. **Continuous Deployment**: Microservices support continuous deployment and DevOps practices, allowing for rapid updates and feature releases without affecting the entire system. This is vital for staying competitive in the dynamic food delivery market.

Real-world examples of projects similar to goFOOD that employ microservices:

1. **Uber Eats** utilizes microservices for tasks like order processing, payment management, and restaurant operations. Their diverse technology stack ensures flexibility. They rely on microservices to meet high-demand scenarios and deliver a seamless food delivery experience.

2. **Glovo** employs microservices to handle various functions, including order fulfillment, courier tracking, and payment processing. They use horizontal scaling and load balancing for efficient deliveries, particularly during peak hours. Service boundaries are maintained to keep functional components independent.

3. **iFood** employs microservices for order placement, restaurant coordination, and user management. They use containerization and orchestration for efficient deployment. High availability is ensured through load balancing and redundancy within their microservices architecture.

In summary, goFOOD is highly relevant due to its complexity, scalability requirements, technological diversity, data privacy concerns, and the need for fault tolerance. Implementing this system through distributed microservices is necessary to effectively address these challenges and stay competitive.

Service Boundaries

1. **Food Ordering Service** 📱 :

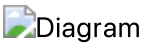
- This service handles everything related to placing and tracking food orders.
- Implement task timeouts for order requests to ensure timely responses.
- Maintain a database to store customer orders and delivery status.

1. **Restaurant Management Service:** 🍽 :

- The Restaurant Management Service is responsible for restaurant-specific functionalities.
- Expose APIs for restaurants to manage their menu items, receive and confirm orders, and update order status.
- Implement a status endpoint to check the service's health.
- Maintain a database to store restaurant information, menu items, and order notifications.

Here's a simplified interaction flow:

- Clients use the Food Ordering Service to browse menus, select items, and place orders.
- The Food Ordering Service communicates with the Restaurant Management Service to send orders to the relevant restaurants.
- Restaurants receive orders, confirm them, and prepare the food.
- The Food Ordering Service tracks order status, including preparation and delivery.
- Clients can monitor the status of their orders through the Food Ordering Service.

Here's a simplified system architecture diagram to illustrate the service boundaries for goFOOD:
 Diagram

- The "User Interface" remains the front-end for user interaction.
- The "Gateway" serves as the API Gateway responsible for routing requests.
- Both microservices, "Food Ordering Service" and "Restaurant Management Service," are present, encapsulating specific functionality.
- A cache is introduced to improve performance and reduce database load. Both microservices can interact with this cache to store frequently accessed data, such as menu items or restaurant details.
- Each microservice communicates with its respective database.

Technology Stack and Communication Patterns

Choosing the right technology stack and communication patterns is critical when implementing microservices in my goFOOD project.

FastAPI web framework is suitable for building RESTful APIs, which are commonly used in microservices architectures. For the gateway I will go with Elixir.

Communication Patterns:

- **RESTful APIs:** I'll use RESTful APIs for communication between the User Interface, Gateway, and Microservices. REST is well-suited for HTTP-based communication and is widely supported.

By choosing these technology stacks and communication patterns, I can ensure that my microservices are well-equipped to handle the specific requirements of goFOOD. This approach enables flexibility, scalability, and efficient inter-service communication.

Data Management

Food Ordering Service:

1. Place Order

- Endpoint: `/orders`
- Description: Allows placing an order.
- HTTP Method: POST
- Request:

```
{
  "customer_id": 123,
  "items": [
    {
      "item_id": 1,
      "quantity": 2
    },
    {
      "item_id": 3,
      "quantity": 1
    }
  ]
}
```

- Response:

```
{
  "order_id": 456,
  "message": "Order placed successfully"
}
```

2. Track Order Endpoint

- Endpoint: `/orders/{order_id}`
- Description: Allows users to track the status of their order.
- HTTP Method: GET
- Response:

```
{
  "order_id": 456,
  "status": "Processing",
  "items": [
    {
      "item_id": 1,
      "quantity": 2
    },
    {
      "item_id": 3,
      "quantity": 1
    }
```

```
    ]
}
```

3. Callback Endpoint for Receiving Order Completion Notification from Restaurant

- Endpoint: `/callback/orders/{order_id}/complete`
- Description: Notification from Restaurant.
- HTTP Method: POST
- Request:

```
{
  "order_id": 456,
  "status": "Complete",
  "message": "Order is ready for pickup"
}
```

- Response:

```
{
  "order_id": 456,
  "status": "Complete",
  "message": "Order is ready for pickup"
}
```

Restaurant Management Service:

1. Browse Restaurants

- Endpoint: `/restaurants`
- Description: Allows browsing restaurants.
- HTTP Method: GET
- Response:

```
[
  {
    "id": "1",
    "name": "Restaurant A",
  },
  {
    "id": "2",
    "name": "Restaurant B",
  }
]
```

2. Browse Menus

- Endpoint: `/restaurants/{restaurant_id}`

- Description: Allows browsing menus from a restaurant.
- HTTP Method: GET
- Response:

```
{
  "name": "Restaurant B",
  "menu": [
    {
      "id": "201",
      "name": "Dish 3",
      "price": 12.99
    }
  ]
}
```

3. Manage Menu Items

- Endpoint: `/restaurants/{restaurant_id}`
- Description: Allows restaurants to manage their menu items, including adding, updating menu items.
- HTTP Method: POST (to add), PUT (to update)
- Request (For add or update):

```
{
  "name": "New Dish",
  "description": "Delicious new dish",
  "price": 15.99
}
```

- Response:

```
{
  "item_id": "301",
  "name": "New Dish",
  "description": "Delicious new dish",
  "price": 15.99
}
```

4. Endpoint for Receiving Order Callbacks

- Endpoint: `/callback/restaurants/orders`
- Description: Allows restaurants to update the status of an order.
- HTTP Method: POST
- Request:

```
{
  "restaurant_id": 789,
```

```
  "order_id": 456,
  "status": "Received"
}
```

- Response:

```
{
  "restaurant_id": 789,
  "order_id": 456,
  "status": "Received"
}
```

Data Management Across Microservices:

- The "Food Ordering Service" and "Restaurant Management Service" each have their own databases to manage data independently.
- When an order is placed through the "Food Ordering Service," it communicates with the "Restaurant Management Service" to confirm the order and update its status.
- Data transfer between services is done via JSON over HTTP using the defined endpoints.
- Each service is responsible for maintaining its own data consistency and integrity within its database.

Deployment and Scaling

By implementing containerization with Docker, I ensure that my microservices can be deployed consistently and managed effectively. Docker containers encapsulate each microservice, including all its dependencies, ensuring that they run reliably across various environments. This approach simplifies the deployment process and enhances the scalability and resilience of my goFOOD - Online Food Delivery System, ultimately providing a robust and efficient platform for food ordering and restaurant management.