07 - Introduction to Microservices

Table of Contents

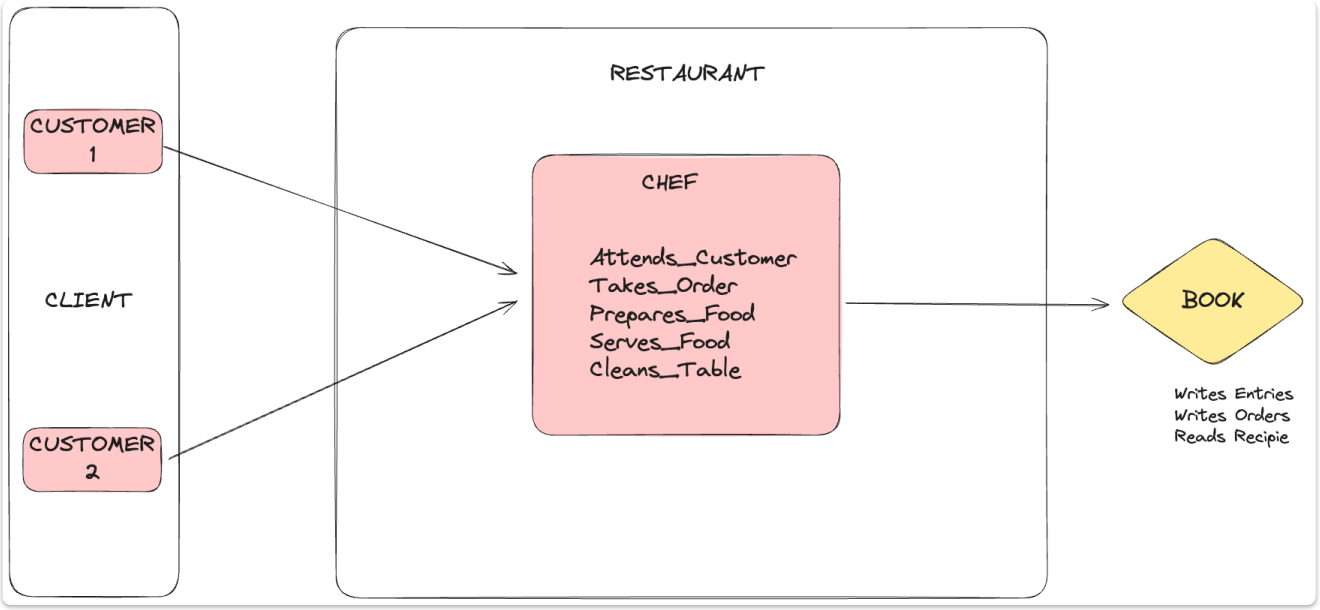
Agenda

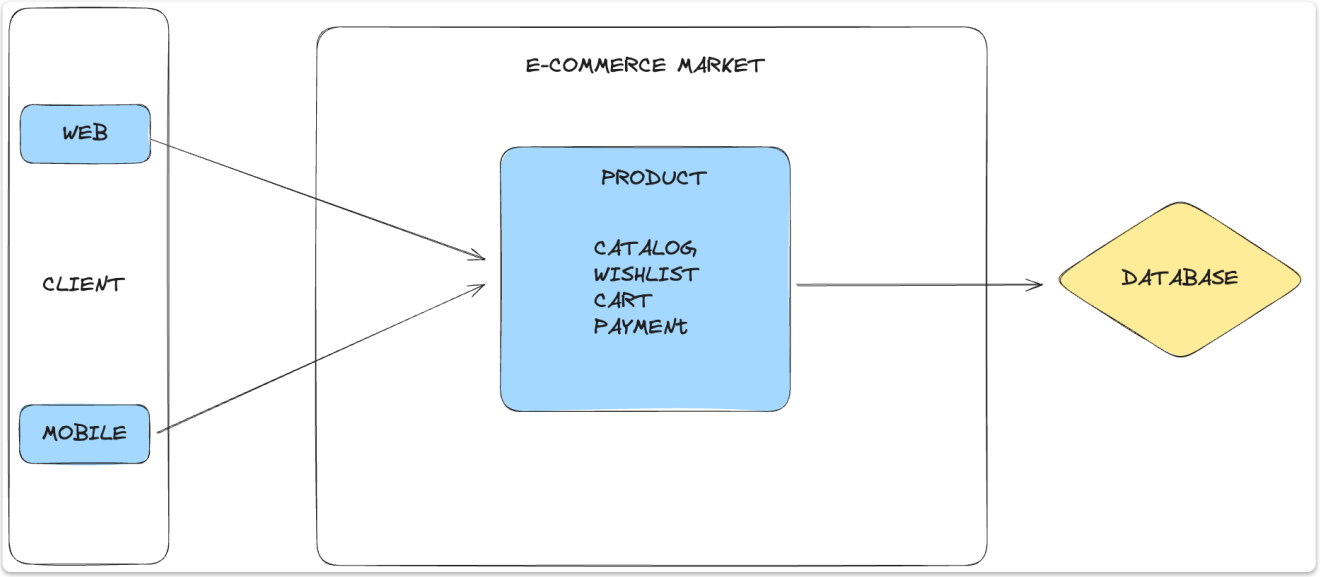
- Monolithic Architecture
- Microservices Architecture
- Overview of Microservices Architecture
- Benefits and challenges of microservices
- Setting Up Microservices
- Creating different server ports for services
- Working with the database for each service
- Inter-Service Communication
  - Communicating between services using HTTP requests
  - Using OpenFeign for inter-service communication

Monolithic Architecture

Traditional Monolithic Architecture:

- Single Codebase: A single point of failure exists in the entire application. If a bug or error occurs in any part of the code, it can bring down the entire application.





Disadvantages

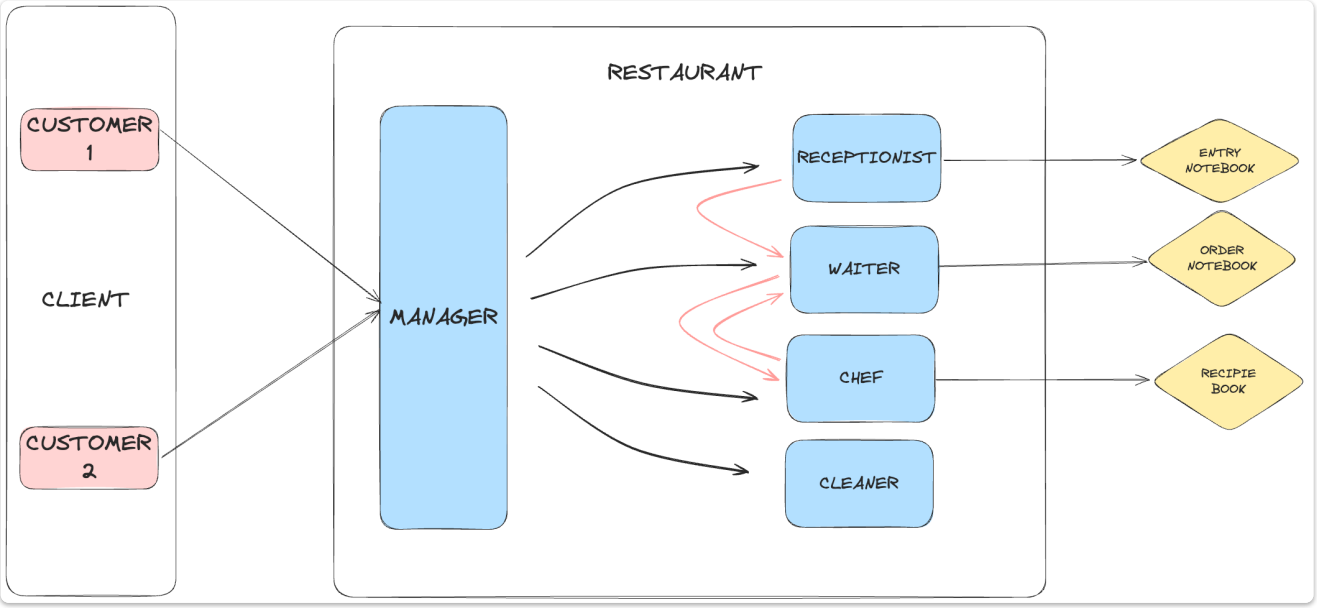
- Slow speed of development
- Performance issues
- The cost of infrastructure
- Lack of flexibility
- Problems with deployment: Even a small change requires the redeployment of the whole monolith.

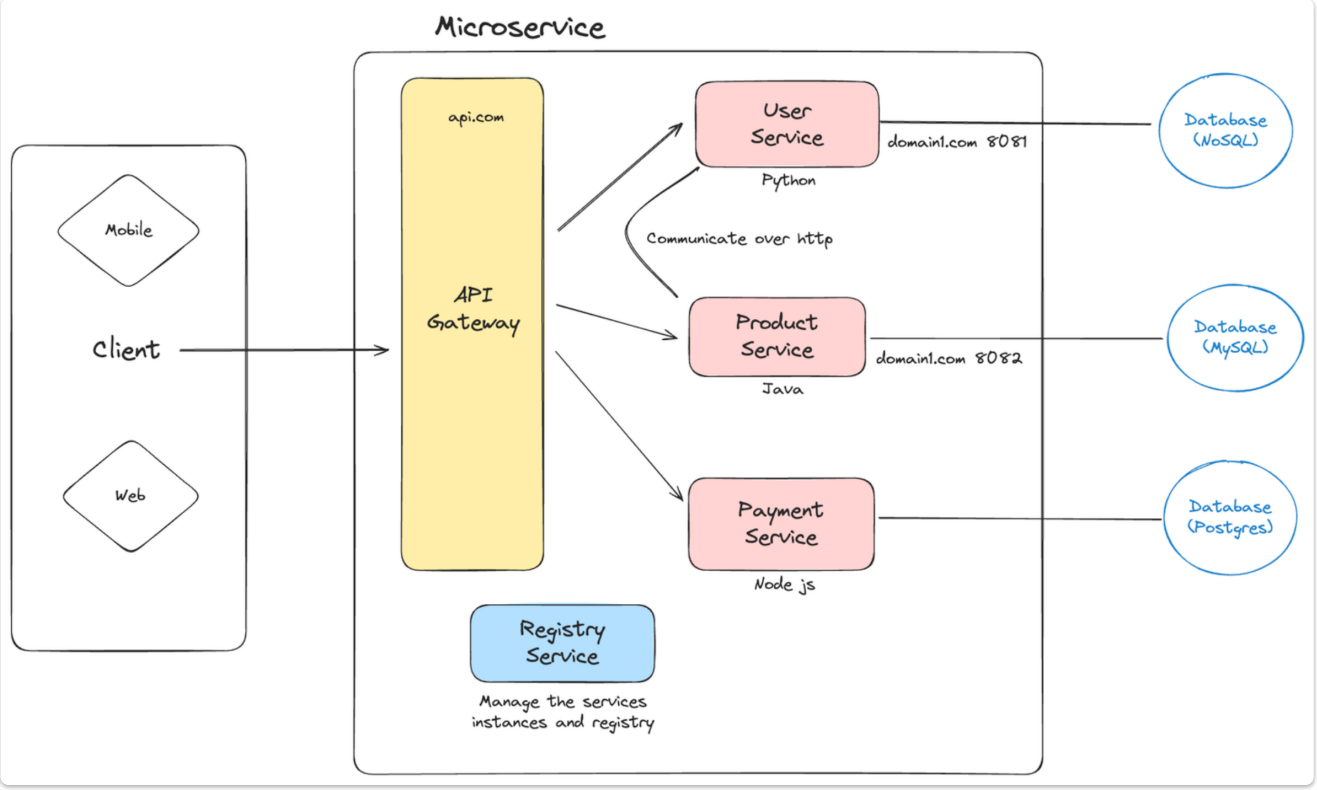
Microservices Architecture

Software Design Pattern where software is composed of small independent services that communicate over well-defined APIs.

- Improved Fault Isolation: Each microservice is a separate unit. If one service encounters an issue, it's less likely to affect other services, making the overall system more resilient.

- Microservices allow each service to be independently scaled to meet demand for the application feature it supports.





Microservices offer scalability, maintainability, and faster development, but introduce distributed system complexity to manage as single points of failure.

Difference between Monolithic and Microservice Architecture

Aspect	Monolithic	Microservice
Architecture	Single unified codebase	Collection of small, independent services

Aspect	Monolithic	Microservice
Development	Easier to develop initially	More complex development and setup
Deployment	Single deployment unit	Independent deployment of services
Scalability	Scales as a whole	Scales independently by service
Fault Isolation	Faults affect the whole application	Faults isolated to individual services
Technology Stack	Usually a single technology stack	Can use different technologies per service
Performance	Direct calls within the same process	Inter-service communication adds overhead
Maintenance	Easier maintenance initially	Can become difficult as the codebase grows
Team Structure	Typically one large team	Small, independent teams per service
Data Management	Single database	Decentralized data management, per service database
Inter-Process Communication	Not needed, internal calls	Required, often via REST, messaging, or gRPC
DevOps	Simplified DevOps pipeline	Complex DevOps with CI/CD for each service
Updates	Single, coordinated update	Independent, frequent updates possible
Testing	Easier to perform end-to-end testing	Requires testing of interactions between services
Resilience	Less resilient to failures	More resilient due to isolated services

Hands-On

1. Create project `UserService` and `ProductService` from spring initialiser and add dependencies:
    1. Spring Web
    2. Spring Data JPA
    3. MySQL driver
    4. Lombok
2. Add configuration in `application.properties` for both services `UserService` and `ProductService`

```
server.port=PORT_NUMBER

#Database setup MySQL
spring.datasource.url=jdbc:mysql://localhost:3306/DATABASE_NAME
spring.datasource.username=USERNAME
spring.datasource.password=PASSWORD
spring.jpa.hibernate.ddl-auto=update
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.show-sql=true
```

3. In the `entities` directory of each service, create their entities:

In `UserService`

```
@Entity
@Data
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int userId;

    @Column(nullable = false, unique = true)
    private String username;

}
```

In `ProductService`

```
@Entity
@Data
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int productId;
    private String productName;

}
```

4. Create `UserService` and `ProductService` interface and their implementations under `services>impl` directory:

In `UserService`
Create interface `UserService`

```
public interface UserService {
    User add(User user);
```

```
    List<User> getAll();

    User getOne(int userId);
}
```

Implement in the `impl` directory inside `service` directory

```
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserRepository userRepository;
    @Autowired
    private ProductClient productClient;

    @Override
    public User add(User user) {
        return userRepository.save(user);
    }

    @Override
    public List<User> getAll() {
        return userRepository.findAll();
    }

    @Override
    public User getOne(int userId) {
        return userRepository.findById(userId).orElseThrow(() -> new
RuntimeException("User not found"));
    }
}
```

In ProductService

```
public interface ProductService {
    Product add(Product product);

    List<Product> getAll();

    Product getOne(int id);
}
```

Add implementation of the interface

```
@Service
public class ProductServiceImpl implements ProductService {
```

```
    @Autowired
    private ProductRepository productRepository;

    @Override
    public Product add(Product product) {
        return productRepository.save(product);
    }

    @Override
    public List<Product> getAll() {
        return productRepository.findAll();
    }

    @Override
    public Product getOne(int id) {
        return productRepository.findById(id).orElseThrow(() -> new
RuntimeException("Product not found"));
    }
}
```

5. Create controllers for each microservice

`UserController`

```
@RestController
@RequestMapping("/user")
public class UserController {

    @PostMapping
    public User create(@RequestBody)

    @GetMapping
    public List<User> getAll()

    @GetMapping("/{userId}")
    public User getOne(@PathVariable )
}
```

`ProductController`

```
@RestController
@RequestMapping("/products")
public class ProductController {

    @PostMapping
    public Product create(@RequestBody )

    @GetMapping
```

```
    public List<Product> getAll()

    @GetMapping("/{productId}")
    public Product getOne(@PathVariable )

    @GetMapping("/user/{userId}")
    public List<Product> getProductsofUser(@PathVariable )

}
```

6. Add repositories and extend them to the JPA Repository

Now we will be able to add data into each of our microservice
Further, when a user selects (or adds a product to the cart) we need to keep a record of that
So we will add in the Product table the userId of the user to whom that product will belong to

7. In the Product microservice make the below changes/additions

In the `Product` entity, add userId

```
public class Product {
        // existing code...
    private int userId;
}
```

In `ProductService` interface

```
public interface ProductService {
        //existing code ..
    List<Product> getProductsOfUser(int id);
}
```

In `ProductServiceImpl`

```
public class ProductServiceImpl implements ProductService {
        //existing code..
    @Override
    public List<Product> getProductsOfUser(int userId) {
        return productRepository.findByUserId(userId);
    }
}
```

In `ProductRepository`

```
public interface ProductRepository extends JpaRepository<Product, Integer>
{
    List<Product> findByUserId(int userId);
}
```

In `ProductController`

```
public class ProductController {
        //existing code...
    @GetMapping("/user/{userId}")
    public List<Product> getProductsofUser(@PathVariable)
}
```

Now we have a record of which product belongs to which user
But still, to get details, we have to call the product microservice and get the data
We need when the User microservice is called we get details of products that belong to a particular user
And we need to hit User microservice for that
User microservice will now need to communicate with Product microservice for this functionality

OpenFeign for Inter-service communication

1. Add Spring Cloud OpenFeign Dependency to the `UserService`

Add Dependency Management for Spring Cloud

```
<dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>${spring-cloud.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
```

Add dependency of OpenFeign

```
  <dependency>
      <groupId>org.springframework.cloud</groupId>
```

```
      <artifactId>spring-cloud-starter-openfeign</artifactId>
    </dependency>
```

In the properties add spring-cloud-version

```
<properties>
    <java.version>21</java.version>
    <spring-cloud.version>2023.0.3</spring-cloud.version>
  </properties>
```

2. Create Product entity in the User microservice

```
@Data
public class Product {

    private int productId;
    private String productName;
    private int userId;
}
```

Keep everything same as the `Product` entity in the Product microservice but do not add annotations that are used for database management

3. In the `UserService` microservice create the `ProductClient` interface

```
@FeignClient(url = "http://localhost:8082", value = "Product-Client")
public interface ProductClient {

    @GetMapping("/products/user/{userId}")
    List<Product> getProductsOfUser(@PathVariable int userId);
}
```

4. Enable feignClient in the main `@EnableFeignClients`
5. In `UserServiceImpl`, add the `productClient` and change the `getAll()` and `getOne()` methods to get products with user.

```
@Service
public class UserServiceImpl implements UserService {
        //existing code...
    @Autowired
    private ProductClient productClient;

        @Override
        public List<User> getAll() {
```

```
            List<User> users = userRepository.findAll();
            List<User> newUserList = new ArrayList<>();

            for (User user : users) {

user.setProducts(productClient.getProductsOfUser(user.getUserId()));
                newUserList.add(user);
            }

            return newUserList;
        }

    @Override
    public User getOne(int userId) {
        User user = userRepository.findById(userId).orElseThrow(() -> new
RuntimeException("User not found"));

user.setProducts(productClient.getProductsOfUser(user.getUserId()));
        return user;
    }
}
```