1. Introduction to Microservices

Microservices is an architectural style that structures an application as a collection of small, autonomous services, modeled around a business domain. Each service is self-contained, independently deployable, and can be developed, deployed, and scaled independently.

Key Principles of Microservices:

- **Single Responsibility:** Each microservice is designed to perform a single business function. In our case, we have an "Account" service and a "Loan" service.
- **Independently Deployable:** You can make a change to a single service and deploy it without needing to redeploy the entire application.
- **Decentralized Governance:** Each team can choose the best technology stack for their specific microservice.
- **Decentralized Data Management:** Each microservice has its own private database to ensure loose coupling.
- **Resilience:** Failure in one microservice does not cascade to other services, making the overall application more resilient.

2. Our Application: Account and Loan Services

Based on the provided diagram, we will create two microservices:

- **Account Service:** This service will manage different types of accounts:
  - Savings
  - Current
  - Stock Trading
  - Fixed Deposits
  - Recurring Deposits
- **Loan Service:** This service will handle various loan types:
  - Personal
  - Car
  - Gold
  - Two Wheeler

3. Project Structure (for each microservice)

We will use Spring Boot to create our microservices. Here's a typical project structure for each service:

```
src
└── main
└── java
    └── com
        └── example
            └── {servicename}
                ├── {Servicename}Application.java  // Main Spring Boot Application
                ├── controller
                │   └── {Servicename}Controller.java // REST API endpoints
                └── model
                    └── {Servicename}.java        // Data model
```

4. Communication Between Microservices

While these services are independent, they might need to communicate. For instance, before approving a loan, the Loan service might need to check the account status from the Account service. Common communication patterns include:

- **Synchronous Communication (REST APIs):** One service makes a direct REST API call to another and waits for a response. This is simple to implement but can lead to tight coupling.
- **Asynchronous Communication (Message Queues):** Services communicate by sending messages through a message broker like RabbitMQ or Kafka. This promotes loose coupling and improves resilience.

5. Service Discovery

In a microservices architecture, the number of services and their locations can change dynamically. A service discovery mechanism is needed for services to find and communicate with each other. Tools like Eureka or Consul are commonly used for this purpose.

6. API Gateway

An API Gateway acts as a single entry point for all client requests. It can handle tasks like routing, authentication, and logging, simplifying the client-side code and the microservices themselves.