**Building HigherEdu App Microservices**

#### **Introduction**

This document outlines the steps and considerations required to migrate an existing monolithic application to a microservices architecture. The current monolithic application includes the following components:

* **Services**: Student, Course, Enrollment, Department, Grading
* **Layers**: Service, Controller, Repository, Model, DTO

Each service is tightly coupled within the monolithic architecture. The goal is to decouple these services into independent, deployable microservices.

#### **Problem Statement**

In today's fast-paced technology landscape, the increasingly complex nature of software systems makes it difficult for monolithic applications to handle changes and scale effectively. Monolithic architectures typically involve a single, large codebase, where all the components and functionalities are tightly coupled. This leads to several issues:

1. **Scalability Challenges**: Scaling a monolithic application often necessitates scaling the entire application, even if the demand is only for specific services.
2. **Development Bottlenecks**: A large, tightly coupled codebase makes it difficult for development teams to work independently, leading to bottlenecks and longer release cycles.
3. **Maintenance and Upgrades**: Updating or refactoring a single component can necessitate extensive testing and deployment, impacting the entire application.
4. **Limited Technology Stack Flexibility**: All components must use the same technology stack, limiting innovation and ability to leverage best-of-breed technologies.

#### **Preparation**

**Project Understanding and Analysis**

* **Identify Modules**: Recognize and understand the current modules (Student, Course, Enrollment, Department, Grading).
* **Decouple Dependencies**: Comprehend dependencies between modules and data flow.
* **Database Structure**: Examine the current database structure and shared data.

#### **Solution**

The transition to a microservices architecture addresses these core challenges by breaking down a monolithic application into smaller, independent services. Each microservice represents a specific business capability and can be developed, deployed, and scaled independently.

Key aspects of the solution include:

1. **Service Decomposition**: Identifying and segregating individual business functionalities into discrete services.
2. **Independent Deployment**: Each service is independently deployable, often managed through containerization strategies using tools like Docker and Kubernetes.
3. **API Gateway**: Introducing an API Gateway to manage and route requests to respective microservices.
4. **Data Management**: Decoupling data storage, ensuring each microservice manages its own database schema.
5. **Inter-Service Communication**: Employing lightweight communication methods such as REST, HTTP/HTTPS, and messaging queues.
6. **Security and Monitoring**: Ensuring robust mechanisms for security, monitoring, and resilience.

#### **Migration Plan**

**Define Microservice Boundaries**

* **Service Decomposition**: Break down the monolithic application into smaller, manageable microservices:
  + Student Service
  + Course Service
  + Enrollment Service
  + Department Service
  + Grading Service

**Establish Communication Strategies**

1. **Service Registry and Discovery**: Incorporate Eureka Server as a service registry to manage and maintain a list of available microservices. Each microservice should be configured to register itself with the Eureka Server upon startup. This allows the API Gateway and other services to discover and communicate with the necessary microservices dynamically.

**Eureka Server Setup**:

* Add the Eureka Server dependency in your Spring Boot application.

A screen shot of a computer

Description automatically generated

* Annotate your main class with **@EnableEurekaServer** to enable the Eureka Server functionality.
* Configure the **application.properties or application.yml** to set up the necessary Eureka Server configurations.

A computer screen shot of white text

Description automatically generated

* Example **of application.properties or application.yml** for the Eureka Server:

A black screen with white text

Description automatically generated

**Service Registration with Eureka**:

* Add the **Eureka Client** dependency to each microservice.

A screen shot of a computer code

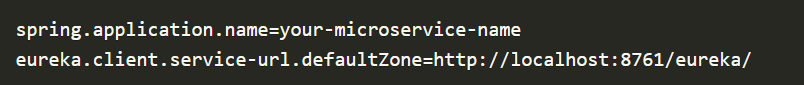
Description automatically generated

* Annotate the main class of each microservice with **@EnableEurekaClient** to enable Eureka Client functionality.
* Configure **the application.properties or application.yml** to point to the Eureka Server.

A computer code on a black background

Description automatically generated

* Example **of application.properties or application.yml** for a microservice:



1. **Inter-Service Communication**: Use REST/HTTP for communication between microservices.
   * **REST/HTTP**: Utilize RESTful APIs for synchronous communication between microservices. Each service can discover the required service endpoint via the Eureka Server.
   * **Optional - Messaging Queues**: If your application requires asynchronous communication, you can implement messaging queues using tools such as RabbitMQ or Apache Kafka. Messaging queues can help handle tasks that do not require immediate responses and can aid in decoupling services.

**Database Decomposition**

* **Database Per Service**: Each microservice should have its own database to ensure data independence.
* **Data Synchronization**: Implement data synchronization mechanisms as needed.

#### **Implementation Steps**

**Create Project Structure** Each microservice follows a similar structure for better organization and clarity.

**Example: Grading Service**

grading-service/

├── src/

│ ├── main/

│ │ ├── java/

│ │ │ ├── com/

│ │ │ │ ├── example/

│ │ │ │ │ ├── student/

│ │ │ │ │ │ ├── controller/

│ │ │ │ │ │ ├── dto/

│ │ │ │ │ │ ├── model/

│ │ │ │ │ │ ├── repository/

│ │ │ │ │ │ ├── service/

│ ├── resources/

│ │ ├── application.properties

├── pom.xml

Similarly, we have the structure for:

* Course Service
* Enrollment Service
* Department Service
* Student Service

**Extracting Services** Each service has its own Models, Repositories, and Controllers.

**Models**

* Extract the model representing the data structure.

**Repositories**

* Extract the repository interface for database interactions.

**Services**

* Extract the business logic.

**Controllers**

* Extract the controller logic for handling HTTP requests.

#### Configuration and Deployment

1. **Spring Boot Configuration**:
   * Create an application.properties file specific to each microservice with database and server port configurations.

***Grading Service***  
The Grading Service Microservice is designed to manage and process student grades in an educational context. This service handles the operations related to maintaining the gradebook, which includes student records, various types of assessments, and the individual grades received by students. Below, we outline the primary components and functionality of the Grading Service.

**Basic Concepts:**

1. **RosterRow:** Represents a student.
2. **RosterColumn:** Represents a type of grade or assessment.
3. **RosterResult:** Represents an individual result or grade that a student receives in a specific column.
4. **Roster:** The collection of all students (rows), grade types (columns), and individual grades (results) - it is the complete gradebook

#### **How It Benefits Eagle Apps**

Embracing a microservices architecture offers several significant benefits specifically for Eagle Apps:

1. **Improved Scalability**: Each microservice can be scaled independently based on its demand, leading to optimized resource utilization and performance.
2. **Enhanced Agility**: Smaller, autonomous teams can develop, deploy, and maintain individual services, improving overall productivity and enabling more frequent releases.
3. **Resilience and Fault Isolation**: Failures in one microservice do not impact the entire application, improving overall system reliability and availability.
4. **Technology Diversity**: Enables the use of diverse technologies best suited for each service, fostering innovation and adaptability.
5. **Simplified Maintenance**: Isolated codebases lead to easier maintenance, lowering the risk of unintended side effects during updates or enhancements.
6. **Focused Business Logic**: Services are built around business capabilities ensuring clear alignment with business objectives and streamlined operations.

By breaking down the monolith into microservices, Eagle Apps can achieve significant operational efficiencies, better quality software, and an improved ability to respond rapidly to market demands.