**Assignment 1 : Introduction to Microservices**

The software should follow the architectural design

principles. It should be cohesive, loosely

coupled, agile, flexible,and distributed. Provide a design for the above

mentioned.

Requirement (refer to the Problem Statement) with the following functions:

1. Manage students' onboarding

2. Manage students' fees

3. Fees payment to be initiated internally by student's management service

4. Mention the details of inter-process communication

5. Keep high availability into account in your design

**Solution:**

To create a cohesive microservices architecture for managing student onboarding, fees, and payment, following best practices such as high availability, loose coupling, flexibility, and scalability, here's how the design can be structured. Below, I will break down the solution in terms of services, communication, database-per-service pattern, and high availability considerations.

**Microservices Architecture Design**

**Overview of the Services**

The system can be broken into several microservices, each having a clear responsibility to manage students' data, fees, and payments.

1. **Student Management Service**
   * **Responsibilities**: Manage student onboarding details, such as personal data, academic details, enrollment, etc.
   * **Endpoints**:
     + POST /students - Onboard a new student.
     + GET /students/{studentId} - Get student details.
     + PUT /students/{studentId} - Update student information.
   * **Database**: Each service will have its own database, and this service will maintain a "Student" database.
2. **Fees Management Service**
   * **Responsibilities**: Manage the fee structure, calculate fees for a student, and track pending fees.
   * **Endpoints**:
     + GET /fees/{studentId} - Retrieve the fee details for a specific student.
     + POST /fees/{studentId} - Add or update fee details for a student.
   * **Database**: The database will contain information related to fee details, including charges, deadlines, etc.
3. **Payment Management Service**
   * **Responsibilities**: Handle the payment process, including initiating and confirming payments.
   * **Endpoints**:
     + POST /payments/{studentId} - Initiate a payment for a specific student.
     + GET /payments/{paymentId} - Retrieve payment status for a specific payment.
   * **Database**: Contains transaction details, payment history, and statuses.
4. **Notification Service**
   * **Responsibilities**: Send notifications (email/SMS) when actions are completed (e.g., student onboarded, fees updated, payment completed).
   * **Endpoints**:
     + POST /notifications - Send a notification.
   * **Database**: Stores sent notifications for tracking purposes.

**Inter-Service Communication**

The communication between microservices should be efficient and reliable. Based on the architecture, there are two primary forms of communication:

1. **Synchronous Communication (REST APIs)**:
   * Each microservice exposes REST APIs for the other services to interact with.
   * Example: When the Student Management Service creates a student, the Fees Management Service can fetch the fees based on the student ID.
2. **Asynchronous Communication (Message Queues/Events)**:
   * Services like Payment Management can emit events (e.g., “PaymentInitiated” or “PaymentCompleted”) via a message broker like Kafka or RabbitMQ.
   * For instance, after initiating a payment, the Payment Management Service will publish a "PaymentInitiated" event to inform other services that the payment has been initiated.
3. **Event-Driven Architecture**:
   * We can use event-driven architecture to handle complex workflows asynchronously, where services react to events. For example:
     + **Student Management Service** triggers a StudentCreated event once a student is onboarded.
     + **Payment Management Service** listens to this event and checks if there is any fee payment due for the student.

**Database per Service Pattern**

Each microservice maintains its own database to ensure autonomy and scalability:

* **Student Management Service Database**: This database contains student-specific information (name, academic details, etc.).
* **Fees Management Service Database**: Stores fee-related data, such as fee structure, due dates, and amounts.
* **Payment Management Service Database**: Maintains records of payments, transaction statuses, and logs.

**Benefits of Database per Service Pattern**:

* Loose coupling between services.
* Each service is independent and can scale or evolve independently.
* Optimized database design for each service's needs.

**High Availability Design**

To ensure high availability, the following steps should be taken:

1. **Service Redundancy**:
   * Deploy each service in multiple instances across different availability zones to prevent a single point of failure.
2. **Load Balancing**:
   * Use a load balancer (e.g., AWS ELB, NGINX, or HAProxy) to distribute traffic evenly across multiple instances of each service.
3. **Auto-Scaling**:
   * Implement auto-scaling policies (e.g., AWS Auto-Scaling Groups, Kubernetes Horizontal Pod Autoscaler) to handle varying traffic loads.
4. **Database Replication**:
   * Use database replication to ensure high availability of the database. For example, use master-slave replication or sharded databases for the microservices’ databases.
5. **Circuit Breakers**:
   * Implement circuit breakers (e.g., using libraries like Hystrix) to prevent cascading failures in the system if one of the services goes down.
6. **Disaster Recovery**:
   * Maintain backup copies of the databases and configuration to allow for quick recovery in case of a catastrophic failure.

**Agility and Flexibility Considerations**

* **Loose Coupling**:
  + By following the microservices pattern, each service is loosely coupled and can evolve independently. Changes in one service won’t affect others.
* **API Gateway**:
  + An API Gateway (e.g., Kong, NGINX, or AWS API Gateway) can be used to simplify API consumption for clients by providing a single entry point. It also helps in handling cross-cutting concerns like authentication, rate limiting, and logging.
* **Continuous Integration/Continuous Deployment (CI/CD)**:
  + Use a CI/CD pipeline for quick deployment of microservices, allowing each service to be developed and deployed independently. Jenkins, GitLab CI, and GitHub Actions can be used to automate testing and deployment.

**Security Considerations**

* **Authentication & Authorization**:
  + Use OAuth2/JWT tokens for service-to-service authentication. An Identity Provider (e.g., Keycloak or Auth0) can handle user authentication and authorization.
* **Data Encryption**:
  + Encrypt sensitive data both in transit (TLS) and at rest (database encryption).
* **Audit Logs**:
  + Maintain audit logs for each service to track important events, such as student onboarding and payment initiation.

**Conclusion**

The design outlined above is a modular and flexible microservices architecture that adheres to key principles such as loose coupling, flexibility, agility, and high availability. Each service is independent, has its own database, and communicates asynchronously through events and synchronous REST APIs, ensuring that the system can scale and evolve without disrupting other parts of the application.