Subscription Management System - Phase 2

This project implements a subscription management system for internet providers, adhering to a clean architecture and incorporating microservices for billing and active plan management.

Project Structure

The project is composed of three main services:

- **ServicoGestao (Main Service):** Handles client, plan, and subscription management.
- **ServicoFaturamento (Billing Service):** Manages payments and charges.
- **ServicoPlanosAtivos (Active Plans Service):** Maintains a cache of active subscriptions and responds to queries about subscription validity.

Technologies Used

- Node.js
- Express.js
- SQLite (for ServicoGestao database)
- Message Broker (in-memory for demonstration, can be extended to RabbitMQ/Kafka)
- Docker & Docker Compose

Setup and Running

Prerequisites

- Docker Desktop installed and running.
- Node.js and npm (optional, if you prefer to run services individually without Docker Compose).

Steps to Run with Docker Compose

1. **Clone the repository (if applicable) or create the project structure as described.**

2. **Navigate to the root directory of the project (where `docker-compose.yml` is located).**

3. **Build and start the services:**

```
docker-compose up --build
```

This command will:

- Build the Docker images for each service based on their `Dockerfile` (which will simply copy the code and install dependencies).
- Start all three services in detached mode.

4. **Verify services are running:**

   You should see output indicating that each service is listening on its respective port (3000, 3001, 3002).

Running Services Individually (without Docker Compose)

1. **Navigate to each service's directory (`servico-gestao`, `microservices/servico-faturamento`, `microservices/servico-planos-ativos`).**

2. **Install dependencies:**

   ```
   npm install
   ```

3. **Start each service:**

   ```
   npm run dev
   ```

   (This uses `nodemon` for automatic restarts on code changes. For production, `npm start` would be used.)

API Endpoints (via ServicoGestao - Port 3000)

The main `ServicoGestao` acts as an API Gateway for certain functionalities. The `postman-collection.json` provides a comprehensive set of requests.

ServicoGerenciamentoPlanos (ServicoGestao)

- **GET /gerenciaplanos/clients**: List all clients.
- **POST /gerenciaplanos/clients**: Create a new client.

```
{
    "name": "John Doe",
    "email": "john.doe@example.com"
}
```

- **PUT /gerenciaplanos/clients/:id**: Update a client.

```
{
    "name": "John D. Smith"
}
```

- **GET /gerenciaplanos/plans**: List all plans.
- **POST /gerenciaplanos/plans**: Create a new plan.

```
{
    "name": "Basic Internet",
    "description": "100 Mbps",
    "price": 50.00
}
```

- **PUT /gerenciaplanos/plans/:id/cost**: Update the cost of a plan.

```
{
    "newPrice": 55.00
}
```

- **POST /gerenciaplanos/subscriptions**: Create a new subscription.

```
{
    "codCli": 1,
    "codPlano": 1,
    "startDate": "2023-01-01"
}
```

- **GET /gerenciaplanos/subscriptions/client/:codCli**: List subscriptions for a specific client.
- **GET /gerenciaplanos/subscriptions/plan/:codPlano**: List subscribers for a specific plan.

ServicoFaturamento (Accessed via ServicoGestao - Port 3000)

- **POST /registrarpagamento**: Register a payment for a subscription. This request is handled by `ServicoGestao` which then forwards it to `ServicoFaturamento` (http://servico-faturamento:3001/payments).

```
{
    "dia": 25,
    "mes": 6,
    "ano": 2025,
    "codAss": 1,
    "valorPago": 49.99
}
```

ServicoPlanosAtivos (Accessed via ServicoGestao - Port 3000)

- **GET /planosativos/:codass**: Check if a specific subscription is active. This request is handled by `ServicoGestao` which then forwards it to `ServicoPlanosAtivos` (http://servico-planos-ativos:3002/active-plans/:codass).

Postman Collection

A Postman collection named `postman-collection.json` is provided in the root directory. You can import this file into Postman to easily test all the endpoints.

Conclusion and Development Insights

Challenges Encountered and Solutions

1. **Inter-service Communication:**
    - **Challenge:** Deciding on synchronous vs. asynchronous communication for different scenarios.
    - **Solution:** For `ServicoFaturamento` and `ServicoPlanosAtivos` direct queries, synchronous HTTP requests were chosen, with `ServicoGestao` acting as a proxy/gateway. For payment events (which need to update the cache in `ServicoPlanosAtivos` without blocking the main flow), an asynchronous message broker pattern was implemented. Although a full-fledged message broker like RabbitMQ was not set up due to complexity within the project scope, an in-memory `MessageBrokerService` was created to simulate this behavior, demonstrating the event-driven architecture.
2. **Maintaining Active Plan Cache:**
    - **Challenge:** The `ServicoPlanosAtivos` needs to maintain a fast, up-to-date cache of active subscriptions.

- **Solution:** Implemented a `SubscriptionCacheRepository` within `ServicoPlanosAtivos` that stores active subscriptions. This cache is updated when new subscriptions are created (via direct HTTP call from `ServicoGestao`) and, critically, when payments are registered (via the message broker). The `CheckSubscriptionUseCase` in `ServicoPlanosAtivos` directly queries this cache.

3. **Clean Architecture Adaptation for Microservices:**

    - **Challenge:** Applying Clean Architecture principles across multiple services, ensuring clear separation of concerns within each, and defining clear interfaces between them.
    - **Solution:** Each microservice (`ServicoFaturamento`, `ServicoPlanosAtivos`) was designed with its own Clean Architecture structure (Entities, Use Cases, Repositories, Controllers). The communication points (`MessageBrokerService`, HTTP requests) were treated as external interfaces, allowing each service to remain largely independent in its internal logic.

4. **Database Management (SQLite for `ServicoGestao`):**

    - **Challenge:** Ensuring the SQLite database for `ServicoGestao` is properly initialized and persisted (for local development).
    - **Solution:** The `Database.js` in `ServicoGestao` handles the creation of the `database.sqlite` file and necessary tables on startup if they don't exist. For Docker, a volume mount ensures the database file persists across container restarts during development.

References that Aided Development

- **Clean Architecture by Robert C. Martin (Uncle Bob):** The foundational principles for structuring the application layers.
- **Node.js & Express.js Documentation:** For building the RESTful APIs.
- **SQLite Documentation:** For basic database operations.
- **Pattern: API Gateway:** Understanding how a central service can route requests to multiple microservices.
- **Pattern: Publisher/Subscriber (Message Broker):** For implementing asynchronous communication between services.

This phase successfully integrates the core `ServicoGestao` with new microservices, demonstrating inter-service communication patterns and extending the system's capabilities as per the project requirements.