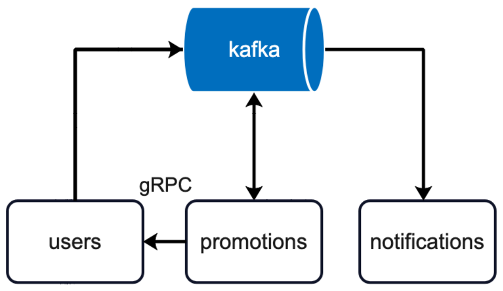
Casino Loyalty Reward System

Microservices Architecture Overview

The system consists of three microservices—**Users**, **Promotions**, and **Notifications**—that communicate via **Kafka** and **gRPC**.

- **Users Service**: Manages users and transactions.
- **Promotions Service**: Assigns and tracks promotions.
- **Notifications Service**: Handles notifications and real-time WebSocket updates.

Communication Flow



Kafka is used in this case instead of HTTP or gRPC to decouple microservices, allowing them to operate independently without direct dependencies. Unlike RabbitMQ, Kafka follows a pub/sub model, enabling future microservices to consume the same events without modifying existing services. Compared to Redis, Kafka provides better data persistence, reliability, and more advanced configuration options for handling message retention and delivery guarantees. Additionally, Kafka scales efficiently with high throughput, making it a more robust choice for long-term event-driven architectures

Scenario 1: Player Registration

When a **player registers**, the **Users** service sends a message to the Kafka topic **players**:

```
{
  "topic": "player",
  "eventType": "PLAYER_REGISTERED",
  "user_id": 12345
}
```

- The **Promotions** service listens for this topic and checks if `eventType` is `PLAYER_REGISTERED`.
- If true, a **welcome promotion** (**active and in currently available**) is assigned to the user and stored in the database table `player_promotions`:

```
{
  "user_id": 12345,
  "promotion_id": 67890,
  "claimed": false
}
```

- A message is then sent to the **notifications** topic:

```
{
  "topic": "notifications",
  "user_id": 12345,
  "eventType": "PROMOTION",
  "content": {
        "id": "5bcc490a-7645-420a-b8f8-16dcaeafe32c",
        "title": "Welcome Bonus",
        "description": "Get a welcome bonus on registration",
        "isActive": true,
        "type": "WELCOME_BONUS",
        "amount": "20.00",
        "startDate": "2025-01-01",
        "endDate": "2025-06-10"
      }
}
```

- The **Notifications** service listens for this event, sends a **real-time WebSocket notification** to a particular user, and stores the notification in the `notifications` table with additional `read` field:

```
{
  "user_id": "fbace2aa-4faa-4423-b41f-b319ff02c6bc",
  "notifications": [
    {
      "type": "promotion",
      "content": {
        "id": "5bcc490a-7645-420a-b8f8-16dcaeafe32c",
        "title": "Welcome Bonus",
        "description": "Get a welcome bonus on registration",
        "isActive": true,
        "type": "WELCOME_BONUS",
        "amount": "20.00",
        "startDate": "2025-01-01",
        "endDate": "2025-06-10"
      },
      "read": false
    }
  ]
}
```

- If a user was **offline**, they can later retrieve **unread notifications**.

Scenario 2: Admin-Assigned Promotion

When an **admin or staff member** assigns a active and in currently available promotion to **player**:

1. The **Users & Promotion** association is stored:

```
{
  "user_id": 12345,
  "promotion_id": 67890,
  "claimed": false
}
```

2. A notification message is sent for each user:

```json
{
  "topic": "notifications",
  "user_id": 12345,
  "eventType": "PROMOTION",
  "content": {
        "id": "5bcc490a-7645-420a-b8f8-16dcaeafe32c",
        "title": "Welcome Bonus",
        "description": "Get a welcome bonus on registration",
        "isActive": true,
        "type": "WELCOME_BONUS",
        "amount": "20.00",
        "startDate": "2025-01-01",
        "endDate": "2025-06-10"
      }
}
```

3. The **Notifications** service processes it just like in Scenario 1.

Scenario 3: Claiming a Promotion

When a **user claims** an available and active promotion:

- The `claimed` status in the **Promotions** database is updated to `true`:

```json
{
  "user_id": 12345,
  "promotion_id": 67890,
  "claimed": true
}
```

- The **Users** service is notified via **gRPC** with the transaction details:

```json
{
  "userId": 12345,
  "amount": 100,
  "transactionType": "CREDIT",
  "description": "Claimed 'Welcome Bonus' promotion of type 'Deposit Match'",
  "additionalData": {
    "promotionId": 67890
  }
}

```

- If this request fails, **the entire transaction is rolled back**.
- Otherwise, the **Users** service stores the transaction in `transactions` table like the audit log:

```json
{
  "transactions": [
    {
      "id": "tx_001",
      "user_id": 12345,
      "amount": 100,
      "transactionType": "CREDIT",
      "description": "Claimed 'Welcome Bonus' promotion",
      "additionalData": {
          "promotionId": 67890
     }
    }
  ]
}
```

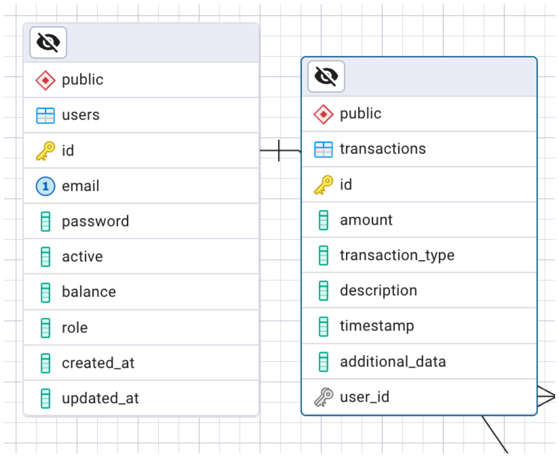
- The **user's balance** is updated accordingly in `users` table
-

gRPC gRPC is used instead of regular HTTP for communication between promotio due to its efficiency and lower latency. Kafka is not used in this case because the Promotions service depends on the Users service to know about transactions and determine whether to roll back, making direct communication through gRPC more suitable for this use case.

Databases per Microservice

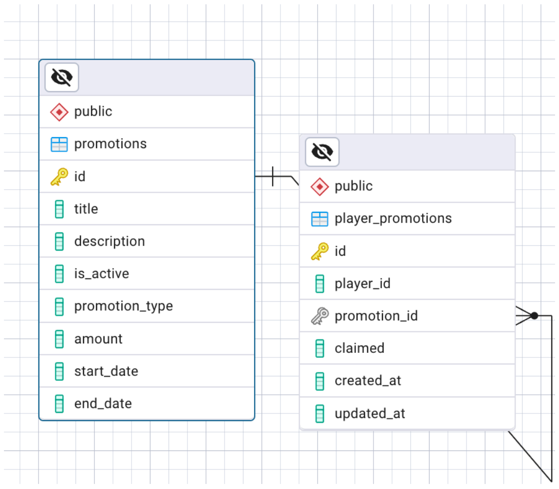
Users Service

- **PostgreSQL** is used because of relational data, especially for handling with **money** transactions securely.
- **Redis** is utilized to store **refresh tokens**. Redis is an **in-memory database** that **scales easily** and provides benefits like **TTL (Time-To-Live)** for automatic expiration. Redis uses master for writing and slaves for reading to reduce load.



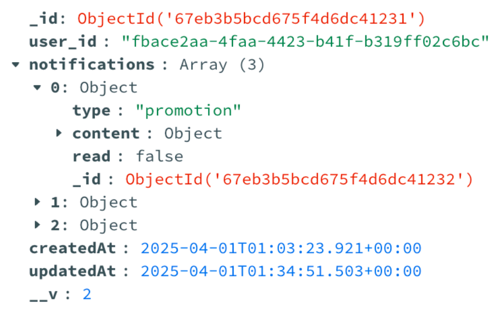
Promotions Service

- **PostgreSQL** is used due to the need for **relational data management**.



Notifications Service

- **MongoDB** is chosen because it allows for **easier scaling** and **efficient retrieval of notifications** for a specific player. MongoDB is a good choice for this schema because the `content` field can change dynamically depending on the type of notification.



```
    _id: ObjectId('67eb3b5bcd675f4d6dc41231')
    user_id : "fbace2aa-4faa-4423-b41f-b319ff02c6bc"
  ▾ notifications : Array (3)
    ▾ 0: Object
        type : "promotion"
      ▸ content : Object
        read : false
        _id : ObjectId('67eb3b5bcd675f4d6dc41232')
    ▸ 1: Object
    ▸ 2: Object
    createdAt : 2025-04-01T01:03:23.921+00:00
    updatedAt : 2025-04-01T01:34:51.503+00:00
    __v : 2
```

Scaling and Service Architecture:

The system is designed for scalability with multiple instances of essential services, including `users`, `promotions`, `notifications`, `Kafka`, `Redis`, and `MongoDB`.

1. **Users Service**:
   The `users` service is deployed with **3 replicas**. This scaling configuration allows the system to handle increased load by running multiple instances of the service, balancing the traffic among them. This improves both performance and fault tolerance.
2. **Promotions Service**:
   Similarly, the `promotions` service is scaled with **3 replicas**. Each instance of the service can independently process requests, ensuring better scalability and redundancy.
3. **Notifications Service**:
   The `notifications` service is also deployed with **3 replicas**, enabling it to handle a large number of simultaneous notification requests effectively.
4. **Kafka Cluster**:
   Kafka is configured with **3 brokers** (`kafka-0`, `kafka-1`, `kafka-2`), ensuring reliable message distribution and fault tolerance. Each broker is part of a distributed cluster, which handles high-throughput messaging across services.
5. **MongoDB Replica Set**:
   The MongoDB setup uses a **replica set** with a **master-slave configuration**, ensuring data availability and failover capabilities for the `notifications` service.
6. **Redis Cluster**:
   Redis is set up with a **master-slave configuration** (with `redis-master` and `redis-slave-1`), ensuring high availability and data replication for caching and real-time data.

Load balancing

Nginx: Nginx acts as a reverse proxy and load balancer, directing traffic to the correct service based on the URL. The Nginx configuration includes the following features:

- Load balancing between the `users`, `promotions`, and `notifications` services, distributing incoming traffic evenly across multiple instances of services, ensuring better scalability and reliability by preventing any single instance from being overloaded.
- WebSocket proxying for the `/notifications/ws` endpoint to support real-time communication.
- A health check endpoint to monitor the system's health.
- Nginx provides security headers and handles proxying requests efficiently across services.

Dockerization

This Dockerization ensures that the application runs consistently across different environments, facilitates scaling with multiple instances, and supports container orchestration using Docker Compose or Kubernetes.

API Routes Documentation

Users Microservice

- **POST** `/api/v1/auth/register`
  Registers a new player. After registration, a message is sent to the Kafka topic.
- **POST** `/api/v1/auth/register-staff`
  Registers a new staff member. Only an admin can perform this action.
- **POST** `/api/v1/auth/login`
  Authenticates a user and generates an `accessToken` and `refreshToken`. The `refreshToken` is stored in Redis with a TTL.
- **POST** `/api/v1/auth/logout`
  Deletes the `refreshToken` from Redis and logs out the user.
- **POST** `/api/v1/auth/refresh`
  Issues a new `accessToken` and `refreshToken`, while the old refresh token is removed from the database. (rotation refreshTokena)
- **gRPC** `addTransaction`
  Used for internal microservice communication to add a transaction and update the user's balance.

Promotions Microservice

- **POST** `/api/v1/promotions`
  Creates a new promotion. Some promotions with specific promotion types have to be unique for the current period and active. Only `staff` and `admin` can create promotions.
- **GET** `/api/v1/promotions`
  Retrieves all promotions with filtering and pagination. Only `staff` and `admin` can access this route.
- **GET** `/api/v1/promotions/players/{player_id}`
  Retrieves all promotions available to a specific player, with filtering and pagination.
- **POST** `/api/v1/promotions/claim`
  Allows a player to claim an available and active promotion. After claiming, a request is sent to the Users microservice to record the transaction and update the player's balance if it fails, the transaction will roll back.
- **POST** `/api/v1/promotions/{promotion_id}/assign`
  Assigns an active and available promotion to players. Only `staff` and `admin` can perform this action. After assigning, an event is sent to Kafka for each assigned player.

Notifications Microservice

- **GET** `/api/v1/notifications`
  Retrieves all notifications for a player.

Middleware & Security

- **Authentication Middleware**
  - Verifies authentication using either `accessToken` or `refreshToken` (refresh and logout route), ensuring their validity.
  - Checks Redis to confirm the validity of stored tokens.
- **Authorization Middleware**
  - Ensures the user has the required role (`player`, `staff`, `admin`).
  - Validates whether the user is active before allowing access.
- **Data Sanitization**
  - All incoming requests go through sanitization to prevent injection attacks.

- **API Versioning**
  - Versioning is supported to maintain backward compatibility (`/api/v1/...`).

Everything follows defined response structures and security best practices.

All endpoints are documented with Swagger:

- Users API Docs (http://localhost/users/api-docs/)
- Promotions API Docs (http://localhost/promotions/api-docs/#/)

Security Measures Implemented

Input Validation and Sanitization

- **Used validation libraries** such as `class-validator` in TypeScript to ensure incoming data conforms to expected formats.

Middleware for Security Enforcement

- Used **helmet.js** to set secure HTTP headers:

```
app.use(helmet());
```

- Disabled unnecessary headers like `x-powered-by` to prevent exposing Express as the framework:

```
app.disable("x-powered-by");
```

Database Security and ORM Usage

- Used an **ORM that supports parameterized queries**, such as TypeORM to prevent SQL injection.

Reverse Proxy Security with Nginx

- Correctly forward client IP addresses to applications:

```
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto $scheme;
```

Scaling and Failover Protection

- Deploy services using **multiple replicas** to avoid single points of failure
- Use **Redis replication** to maintain high availability of caching layers.
- Implement **Kafka for event-driven architecture** to handle failures and ensure message durability.
- Utilize **MongoDB replica sets** for database redundancy and failover.
- **Use circuit breakers and retries** to handle transient failures in microservices communication.

Internal-Only Transaction Services

- Transaction-related services and endpoints handling sensitive financial operations are **not exposed to external networks**.
- **Log all transactions securely** and monitor for anomalies.

Authentication and Authorization

- Use **Role-Based Access Control (RBAC)** to limit user permissions.
- **Rotate refresh tokens** periodically to minimize the risk of token theft.

- Store refresh tokens in **Redis** to provide an extra layer of security, instead of relying solely on token expiration time.
- **Use HTTP-only, Secure cookies** for storing refreshToken, reducing exposure to XSS attacks.

Custom Application Error Codes:

- **Custom Application Error Codes**: Returning detailed custom error codes (like `AppErrorCode`) improves **security** by providing more precise error information that can help diagnose issues without exposing sensitive internal details.

Performance Optimizations in the Project

1. Nginx Configuration

- **Worker Processes and Connections:** The number of worker processes and connections is optimized with the following configuration:

  This ensures efficient handling of concurrent connections by dynamically adjusting the number of worker processes based on the available CPU cores and setting a high limit for concurrent connections.
- **WebSocket Performance:** For WebSocket connections, which are essential for real-time features like notifications, we ensure that the connection stays open for longer periods.

2. Asynchronous Operations:

- The project leverages asynchronous programming to optimize resource usage and performance:
  - **Using `Promise.all` and `Promise.allSettled`** for parallel asynchronous operations, reducing the time for multiple independent tasks to complete by allowing them to run concurrently.

3. Scaling and Load Balancing:

- **Load Balancing:** The project uses Nginx to distribute traffic across multiple instances of services, ensuring that no single server is overloaded. For instance:
- **Horizontal Scaling:** Multiple replicas of services are deployed using Docker Compose, providing redundancy and load balancing across containers, such as for the `users`, `promotions`, and `notifications` services.

4. Connection Pooling for PostgreSQL and MongoDB:

Connection pooling is used to manage and reuse database connections efficiently, reducing the overhead of repeatedly opening and closing connections. It maintains a pool of active connections that can be reused, improving application performance by reducing latency and resource usage.

5. Graceful Shutdown:

- The application ensures a smooth shutdown process to handle termination signals gracefully.

  This prevents incomplete transactions or server errors during shutdown, ensuring that ongoing operations are completed before the service is stopped.

6. Database Migrations (vs. Auto Create):

- **Migrations over Auto Create:** Instead of automatically creating database schemas on production, the project uses migration tools.

  This approach ensures controlled schema changes in production, minimizing the risk of schema inconsistencies or data loss.

7. Production Logging Configuration:

- **Logging Adjustments for Production:** The project uses a minimal logging configuration in production, ensuring that log files remain readable and performant:
  - **Pretty Logger** is disabled in production to avoid unnecessary output.
  - Query logging is only enabled in development, keeping production logs clean and less resource-intensive.

8. Redis:

- **Read/Write Optimization:** Redis is configured to use separate instances for reading and writing, ensuring that read operations do not interfere with write operations, improving Redis performance:

9. GRPC over HTTP for Speed:

- The project uses **gRPC** instead of HTTP for internal service communication, significantly improving the speed and efficiency of communication between services:

  gRPC offers better performance, lower latency, and more efficient message serialization compared to traditional HTTP.

10. **Message Queue Optimizations:**

- Fine-tune Kafka and message queues for optimal throughput and lower latency in communication between services.

11. **Health check API in Nginx**:

- A health check endpoint in Nginx (`/health`) improves performance by ensuring that services are monitored and responsive. This helps prevent unnecessary load on unhealthy services, allowing for proactive management.

12. **Health Check in Docker Containers**

- Docker's container health check ensures that each container is properly functioning, contributing to performance by managing dependencies between containers. Containers dependent on others can be configured to wait for the necessary services to be healthy before starting.

Debugging Setup

In the Docker Compose configuration, the services `users`, `promotions`, and `notifications` are set up to run in debug mode using the `node --inspect` command. This allows for debugging of Node.js applications by exposing the debugger on specified ports.