매치 매이킹 (match making) 시스템

개요

* 매칭 시스템은 사용자 간의 매칭을 자동화하고 관리하는 시스템입니다.

* 실시간 웹소켓을 통해 사용자와 상호작용하며, Redis를 사용하여 빠르고 효율적인 데이터 처리를 수행합니다.

* 주요 기능으로는 사용자 인증, 매칭, 재시도 메커니즘 등이 포함됩니다.

* **소스 코드** - https://github.com/injaeshin/nestjs-match-making

기술 스택

백엔드: NestJS

- Restful API와 WebSocket을 통해 핵심 로직을 처리하여 높은 확장성과 유지보수성을 제공합니다.

데이터베이스: Redis

- 사용자 세션, 매칭 큐 관리 등 빠른 데이터 접근이 필요한 작업에 사용됩니다.

실시간 통신: Socket.IO

- 실시간 매칭 상태 업데이트 및 알림 전송을 처리합니다.

테스트: Jest

- 단위 및 통합 테스트로 코드 안정성과 주요 로직 검증을 수행합니다.

코드 스타일: ESLint, Prettier

- 일관된 코드 스타일과 품질 개선을 위해 린팅 및 포맷팅을 적용합니다.

외부 패키지

- **generic-pool**: 사용자 객체 관리를 위한 풀링 메커니즘을 지원합니다.
- **bull**: 매칭 요청 큐를 통해 순차적으로 매칭을 관리합니다.
- **ioredis**: Redis 데이터베이스와의 상호작용을 지원합니다.
- **yaml**: 설정 데이터를 YAML 형식으로 관리하여 가독성을 높입니다.

기능

사용자 인증

* 사용자는 Restful API로 로그인하며, 이름을 기준으로 신규 여부를 확인합니다.

* 신규 사용자는 자동 등록되며, 1~1000 사이의 무작위 점수를 부여받습니다.

* 로그인 성공 시 토큰이 발급되며, 웹소켓 연결과 매칭 시스템은 토큰 인증 사용자만 이용할 수 있습니다.

매칭 시스템

* 실시간 웹소켓을 통해 매칭 요청을 처리하고 대기열에서 상대를 탐색합니다.

* 적합한 사용자가 없을 경우 재탐색하고, 대기 중인 사용자가 많으면 무작위로 상대를 선정합니다.

* 매칭 성공 시 즉시 결과를 양측 사용자에게 반환합니다.

재시도 메커니즘

* 매칭 실패 시 최대 3회 재시도를 자동으로 진행합니다.

* 모든 시도가 실패하면 봇 매칭으로 전환해 사용자 점수 ±50 범위 내의 봇과 매칭합니다.

* 봇 매칭은 최대 3회 재시도하며, 최종 결과는 실시간으로 사용자에게 전달됩니다.

기능 상세

로그인

* 이름으로 로그인하며, 신규 사용자는 자동 등록과 함께 무작위 점수(1~1000점)를 부여받습니다.

* 로그인 시 서버는 접속 토큰과 사용자 정보를 반환합니다.

매칭 요청 및 검증

- 사용자는 웹소켓을 통해 매칭을 요청하고, 서버는 토큰을 검증하여 사용자를 매칭 대기열에 등록합니다.

매칭 프로세스

- 대기열에서 상대 사용자를 무작위로 선택해 매칭을 시도합니다.
- 매칭 성공 시 결과를 양측에 전달하며, 실패 시 3초 대기 후 재탐색을 최대 3회 반복합니다.
- 3회 실패 시 봇 매칭으로 전환됩니다.
- 사용자 점수 ±50 범위에서 적합한 봇을 탐색하고, 실패 시 범위를 ±50씩 확장해 최대 3회 재시도합니다.

매칭 결과 반환

- 매칭 성공 시 상대방 혹은 봇 정보를 포함한 결과를 사용자에게 반환하며, 최종 실패 시 "매칭 실패" 상태를 반환합니다.

구조 및 시퀀스

디렉토리 구조

Auth

사용자 접속 및 인증 관리를 위한 로직을 포함하여, 로그인 및 토큰 검증을 처리합니다.

Common / Config

공통 유틸리티와 환경 설정 관련 코드로, 여러 모듈에서 재사용 가능한 코드와 기능을 포함합니다.

Data

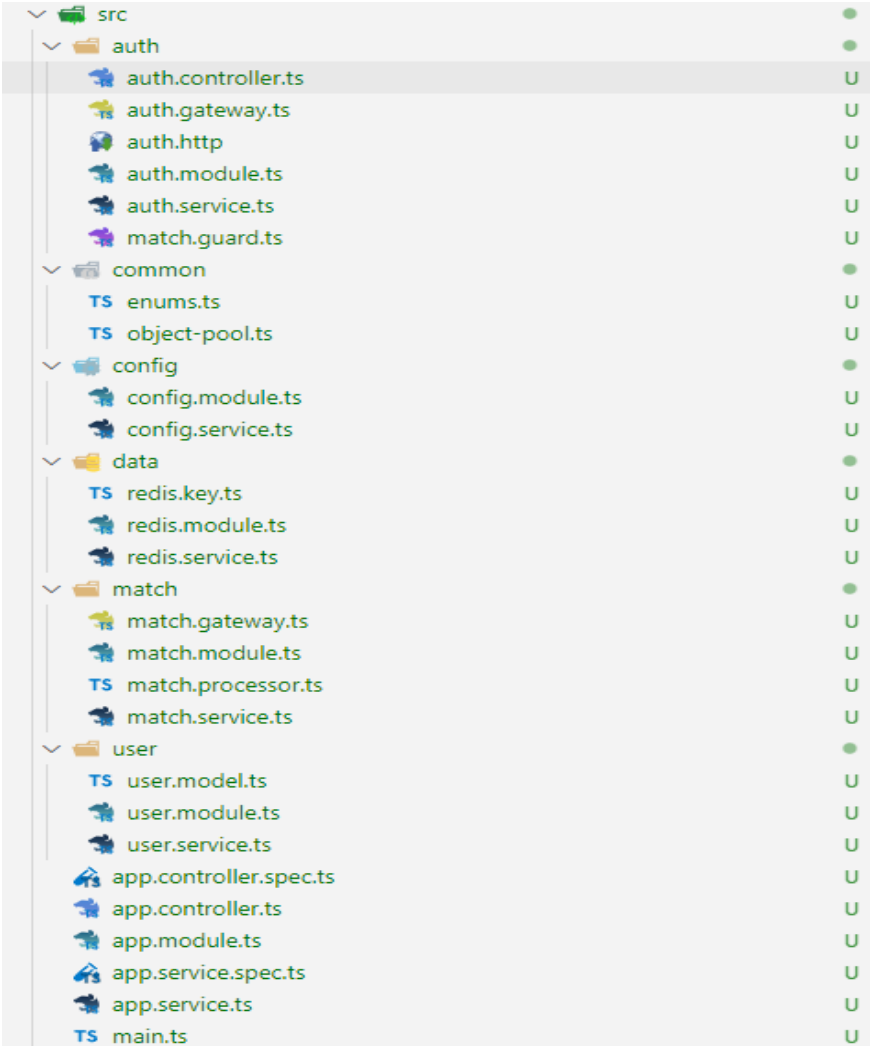
데이터베이스와 관련된 코드를 모아 관리하며, 데이터베이스 모델과 연결 설정이 포함됩니다.

Match

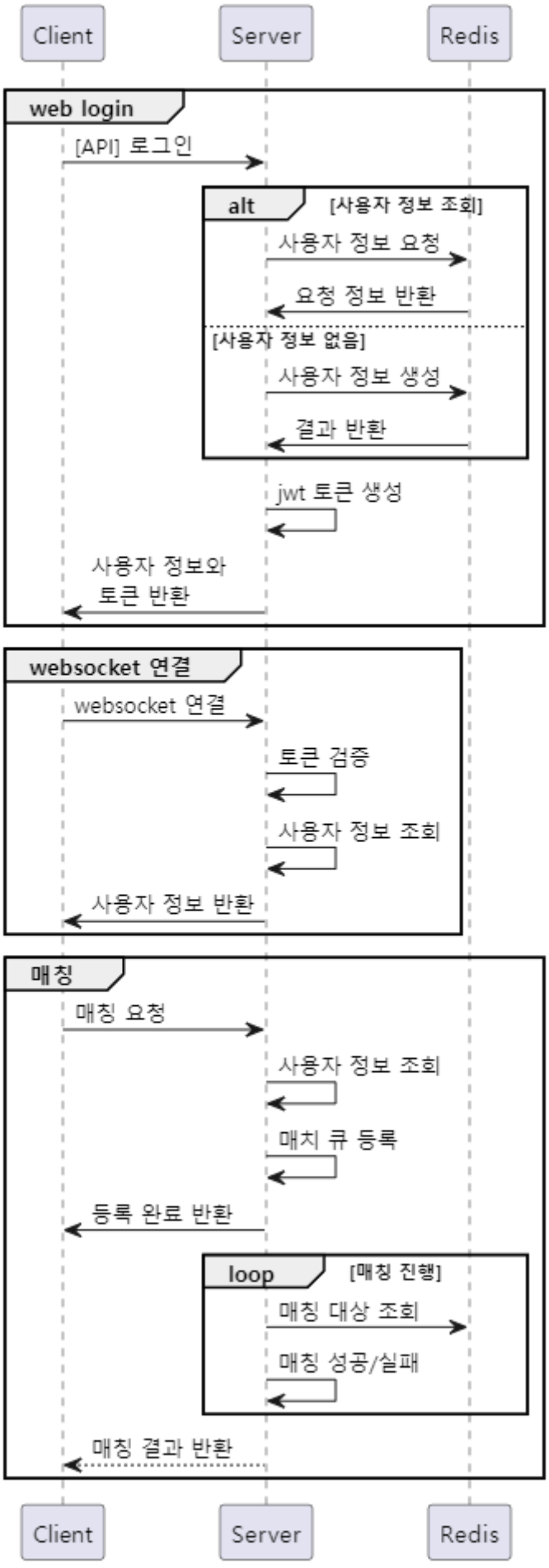
유저 간 또는 봇과의 매칭 로직을 처리하여 매칭의 전체 흐름을 관리합니다.

User

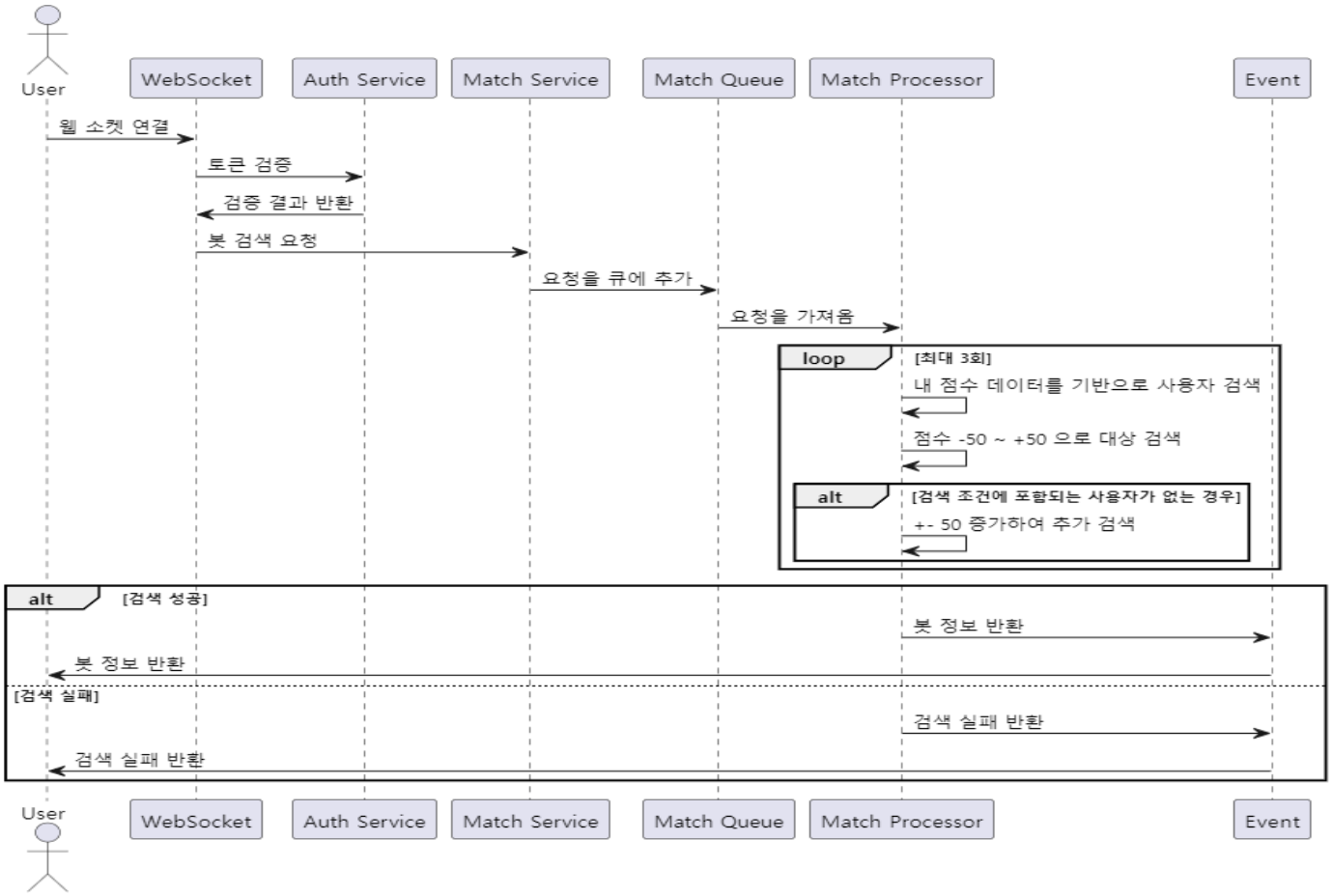
사용자 정보 로직을 관리하여, 사용자 이름, 점수 등과 같은 기능을 포함합니다.



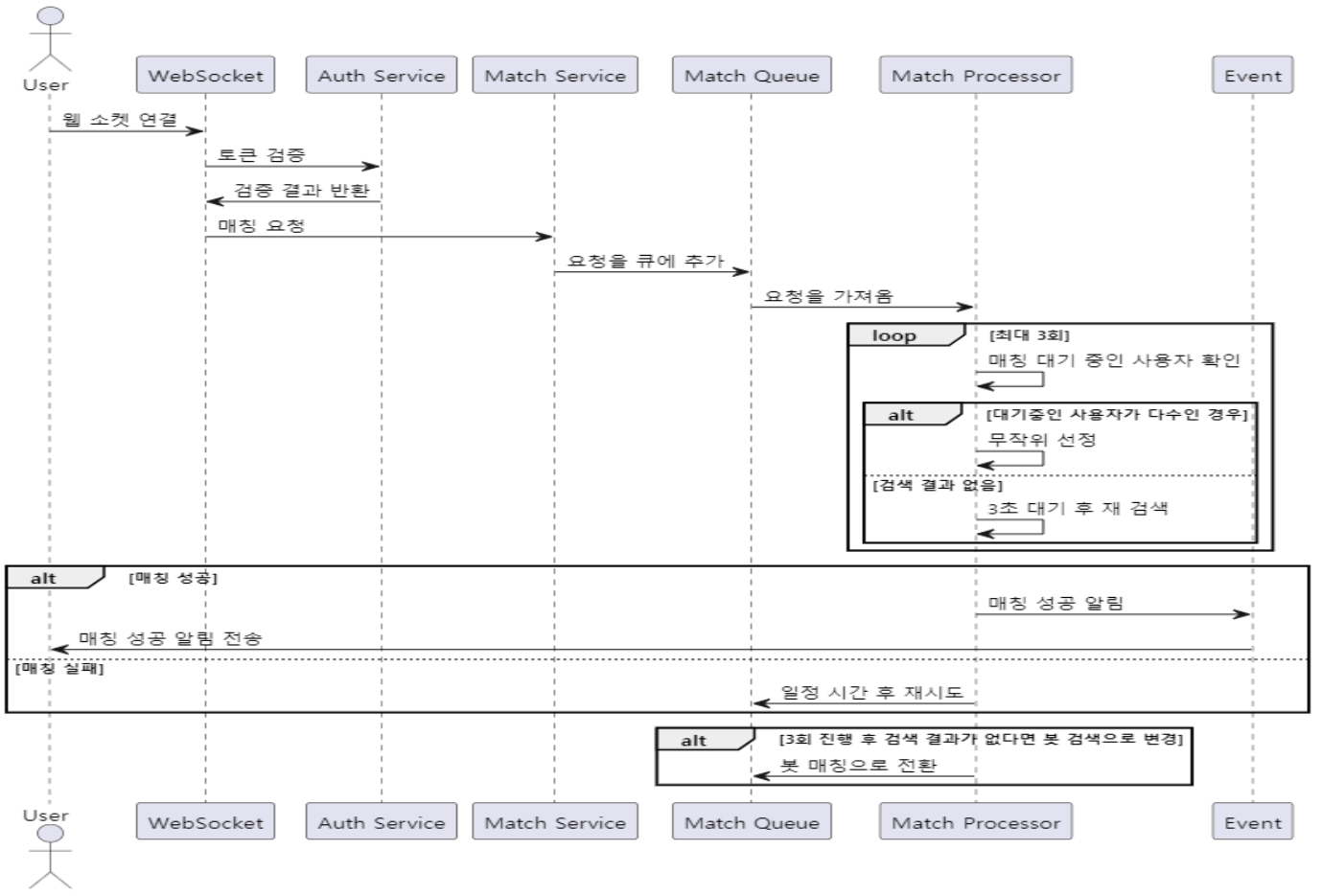
앱 기능 시퀀스


Client
Server
Redis
web login
[API] 로그인
alt
[사용자 정보 조회]
사용자 정보 요청
요청 정보 반환
[사용자 정보 없음]
사용자 정보 생성
결과 반환
jwt 토큰 생성
사용자 정보와
토큰 반환
websocket 연결
websocket 연결
토큰 검증
사용자 정보 조회
사용자 정보 반환
매칭
매칭 요청
사용자 정보 조회
매치 큐 등록
등록 완료 반환
loop
[매칭 진행]
매칭 대상 조회
매칭 성공/실패
매칭 결과 반환
Client
Server
Redis

유저 매칭 큐 시퀀스



봇 매칭 큐 시퀀스



서버 확장 예상 문제와 해결 방안

예상 문제

1. **부하 증가**
   사용자 수 증가로 인해 매칭 요청과 실시간 통신 등에서 서버 부하가 증가할 수 있습니다.

2. **데이터 일관성**
   여러 서버 인스턴스에서 데이터 일관성 유지가 어려워지며, Redis와 같은 인메모리 데이터베이스에서 동기화 문제가 발생할 수 있습니다.

3. **실시간 통신 지연**
   웹소켓 연결 증가로 인해 연결 관리와 메시지 전달에서 지연이 발생할 수 있습니다.

해결 방안

1. **부하 분산**

   로드 밸런서
   로드 밸런서를 사용하여 트래픽을 여러 서버 인스턴스로 분산합니다. 이를 통해 각 서버의 부하를 줄이고 시스템의 전체 성능을 향상시킬 수 있습니다.

   오토 스케일링
   클라우드 환경에서 오토 스케일링을 설정하여 트래픽 증가에 따라 서버 인스턴스를 자동으로 추가하거나 제거합니다. 이는 급격한 트래픽 변화에 유연하게 대응할 수 있도록 합니다.

2. **데이터 일관성 유지**

   Redis 클러스터
   Redis 클러스터를 사용하여 데이터를 분산 저장하고 복제를 통해 데이터 일관성을 유지합니다. 클러스터링을 통해 여러 인스턴스에서 데이터에 접근할 때 동기화 문제를 최소화합니다.

   트랜잭션
   Redis의 트랜잭션 기능을 활용하여 데이터의 원자성을 보장하고, 일관성을 유지합니다.

3. **실시간 통신 최적화**

   웹소켓 클러스터링
   웹소켓 연결을 여러 서버 인스턴스로 분산시키기 위해 웹소켓 클러스터링을 사용합니다. 예를 들어, Socket.IO의 Redis Adapter를 통해 여러 서버 간 메시지 전달을 최적화할 수 있습니다.

   메시지 큐
   메시지 큐를 사용하여 실시간 메시지를 효율적으로 관리하고, 지연을 최소화하여 사용자 간의 즉각적인 상호작용을 보장합니다.