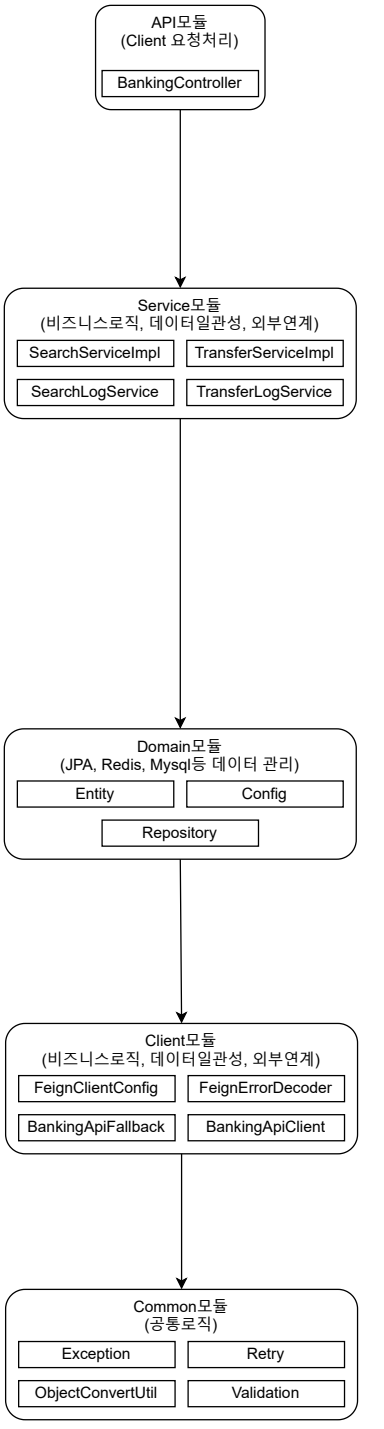

1. 비동기 프로세싱으로 네트워크 대기 시간 최적화

- `Mono.fromCallable`과 `Schedulers.boundedElastic()`를 사용하여 **비동기 방식**으로 작업을 처리하고, **I/O 블로킹 작업**을 별도의 스레드 풀에서 실행
-네트워크 호출이나 외부 시스템과의 통신이 느려도 애플리케이션의 다른 요청 처리에 영향을 최소화.

2. 다양한 예외 처리
- **특정 예외**(`IllegalArgumentException`, `ServiceUnavailableException`)에 대해 상세한 로그를 남기고 적절한 HTTP 응답 코드를 반환.
- 알 수 없는 예외 상황에 대비해 `onErrorResume`으로 **일괄 처리**.

3.대용량트래픽대비(미구현)
- Webclient Connection Pooling설정 : ConnectionProvider(커넥션풀), HttpClient(서버연결, 응답시간)

1. 네트워크 및 시스템 장애를 고려한 예외처리
- Retry : `RetryHandler`를 통해 네트워크 불안정이나 일시적인 장애 상황에서 외부 API 호출을 재시도
- FeignException에 대한 구체적 처리 : `FeignException.GatewayTimeout`과 일반 `FeignException`을 구분하여 각각의 상황에 맞는 메시지를 로깅 및 처리
- 에러 로그 관리 : `TransferLogService`와 `SearchLogService`를 활용해 예외 발생 시 로그를 기록하며, 로그에는 `txID`와 상세 메시지가 포함
- (미구현) : 조회(잔액, 거래결과)API에서는 Ecache 및 @CachePut, @Cacheable을 사용해 DB중단시에도 inmemory활용해 정보 제공가능하도록 구현

2. 계좌 이체의 일관성을 유지하기 위한 설계
- 트랜잭션 상태 관리 : `TransferEntity`를 통해 각 이체 상태(`PENDING`, `COMPLETED`, `FAILED`)를 명확히 기록하여, 장애가 발생해도 상태를 확인, 복구가능
- 출금, 입금과 롤백처리 : 출금이 실패하면 즉시 트랜잭션을 종료하고, 입금이 실패하면 출금을 롤백
- 롤백로직 : 롤백이 실패할 경우에도 `RollbackFailureStateRepository`에 상태를 기록하여, 외부 시스템에서 복구할 수 있도록 정보를 제공
- (미구현) : @TransactionalEventListener를 통한 출금 커밋 완료 후 입금 처리

3. 장애 복구 및 상태 추적
- 로그 및 상태 관리 : 각 트랜잭션 로그를 MDC활용해 txID포함하여 기록
- Idempotency : txID를 통해 중복된 요청 처리 방지

4. 확장성과 유지보수성
- 클라이언트 매핑 : 은행별 API 클라이언트를 `Map<String, BankingApiClient>`로 관리하며, 새로운 은행을 쉽게 추가할 수 있도록 확장성을 제공
- 모듈화 : `SearchService`, `TransferService`, `LogService` 등으로 책임을 분리하여 코드의 가독성과 유지보수성을 향상
- 재사용 가능 유틸 : `ObjectConvertUtil`, `RetryHandler` 등을 활용하여 중복 코드를 최소화하고, 코드의 재사용성향상

1. 조회 성능 향상을 위한 Entity 설계
- orderedUUID : 시간 기반 UUID로 기존 UUID 대비 조회 및 파티셔닝/샤딩 성능 향상 가능
- @PrePersist로 id를 체크해 데이터 업데이트 시 불필요한 조회 방지
- (미구현) R2DBC를 활용한 비동기 DB 조회

2. 롤백 실패 관리로 장애 대응 강화
- Redis를 이용한 롤백 실패 기록 : `RollbackFailureStateRepository`를 통해 롤백 실패 상태를 Redis에 저장합니다.
- (미구현) Kafka등 Event Listener를 Redis에 저장된 롤백실패기록들을 재처리 가능

3. **데이터일관성을 위한 Lock(미구현)**
- ReddisonReactiveClient을 활용한 비동기 Redis락 구현
- Mysql for update를 활용한 쓰기Lock
- JPA 비관적 Lock을 활용한 Phantom Read등 방지

1. 네트워크 불안정성 및 장애 상황에 대한 대응
- Resilience4j기반 Circuit Breaker 적용 : API 호출 실패 시 추가적 실패 방지, 장애 발생시 재시도 제한하여 시스템 안정성 유지
- Fallback처리 : 예외가 발생해도 기본동작 보장하며, 호출실패로 인해 전체 시스템이 중단되지 않도록 보호

2. 데이터 일관성 및 서비스 신뢰성 유지
- 은행별 클라이언트 캐싱 : 이미 생성된 클라이언트가 있다면 재사용, 불필요한 생성 방지하여 성능 최적화
- URL기반 동적처리 : 은행코드에 따라 URL 동적 매핑 및 잘못된 은행코드의 경우 예외처리
- 은행코드, 네트워크 타임아웃, 외부API 문제 등 다양한 상황에 대한 명확한 예외구분 처리, 예외상황별로 커스텀 메시지 제공

3. 장애 복구 및 확장성을 고려한 설계
- Fallback 기본값 제공 : 잔액조회실패, 출금보상처리 실패 등 기본값 제공을 통해 실패 이유 명확하게 도출
- CircuitBreaker와 Fallback 결합 : 네트워크 불안정성, API 장시간 미응답 두가지 장애에도 즉시 대응가능

1. 다양한 Exception 처리
- 계좌 미존재, 은행 미존재, 컨버젼, Feign, Feign Timeout, 거래결과 미존재 등 다양한 Exception 처리

2. Retry, CircuitBreaker Config
- Retry : 최대 3번, 500밀리초 간격 재시도
- CircuitBreaker : 스프링Actuator와 연동가능하게 설정, 10개요청 대상으로 성공/실패 50%기준, 에러나면 30초동안 OPEN 상태 유지

3. Validation
- BankCode, PhoneNumber, Email등 DTO에서 사용하는 컬럼 Valid처리

클라이언트에게 제공할 API 변경 내역

1. /transfer : 출금 -> 입금 및 rollback 처리가 필요하여 따로 API 개발
2. /balance : 조회성으로서 계좌번호로만 조회가능하도록 API 개발
   - request : accountNumber
   - response : amount, isSuccess, message(실패시 사용)
3. /transactions/{txID} : 조회성 및 GetMapping으로 조회 가능하도록 API 개발
   - response : isSuccess, message(실패시 사용)

2,3번의 경우는 구현하지 못한 내용 6번관련하여 Ecache, Redis 활용해 장애 대응 및 속도 최적화하고 싶어 추가 개발

사용한 오픈소스

1. webflux : 비동기/논블로킹 처리로 대용량 트래픽 대비하기 위해 사용
2. resilience4j-circuitbreaker : 장애 발생시 전파 방지 위해 사용
3. resilience4j-retry : 처리 재시도 및 실패 시 해당 트랜잭션을 redis에 저장하여 차후 재시도 하기 위해 사용
4. f4b6a3:uuid-creator : Time-based UUID 생성기로서 조회성능향상 및 차후 샤딩/파티셔닝 향상을 위해 사용
5. mapstruct : Jackson에서 구현하지 못하는 복잡한 DTO Mapping을 위해 사용 및 컴파일타임에 에러 체크하기 위해 사용

구현하지 못한 내용

1. 클라이언트 API요청을 Redis클러스터에 기록, Kafka를 통해 순서대로 Service모듈에 전달
2. Webclient Connection Pooling설정을 통한 대용량 트래픽 대비 ConnectionProvider(커넥션풀), HttpClient(서버연결, 응답시간) 등
3. 출금, 입금 처리 등 연관된 처리에서 @TransactionalEventListener를 활용한 Commit 완료 후 다음 단계 처리
4. R2DBC를 활용한 비동기 DB 처리
5. 실패한 롤백처리에 대해 Redis클러스터에 기록, 향후 Kafka로 전달하여 배치등으로 재처리 로직
6. 뱅킹API의 내역 중 조회 관련(잔액, 거래결과) 실행시 Ecache 및 Redis클러스터에 저장
   - DB 장애 발생 시 Ecache -> Redis순으로 조회가능하도록
   - Ecache, Redis에 데이터가 있다면 조회 속도 최적화