Redis

Redis

- Redis
  - ✓ 레디스와 커넥션을 생성하고 명령어를 실행하고 그 결과를 받는 자바 클라이언트 라이브러리는 다양한데 많이 사용하는 것은 Jedis, Lettuce, Redisson 등
  - ✓ 애플리케이션을 개발하면서 쓰기 동작보다 읽기 동작이 많은 데이터가 있다면 캐시(Cache) 도입을 고민
  - ✓ 일반적으로 캐시는 메모리에 데이터를 미리 적재하고 이를 빠르게 읽어 응답하는 구조
  - ✓ 읽기 동작이 많은 서비스에 캐시를 사용하면 서비스 응답 속도를 향상할 수 있으며. 시스템 리소스도 효율적으로 사용할 수 있음
  - ✓ 스프링 프레임워크는 다양한 저장소에 데이터를 캐시할 수 있는 기능을 제공
  - ✓ 저장소에 독립적이고 추상화된 캐시 메커니즘을 제공
  - ✓ AOP 기반의 애너테이션을 제공하므로 간편하게 캐시 기능을 운영 중인 애플리케이션에 도입할 수 있음

Redis

- Spring Data Redis
  - ✓ 스프링 부트 프로젝트에서는 spring-boot-starter-data-redis 스타터를 제공하는데 Spring Data Redis 프로젝트 라이브러리와 레디스를 연결하는 드라이버, 레디스를 사용하는데 필요한 라이브러리를 모두 포함하고 있음
  - ✓ 사용할 수 있는 드라이버로 Jedis, Lettuce 라이브러리를 기본 포함하며 이를 스프링 부트 프로젝트에서 쉽게 사용할 수 있는 클래스들도 제공
  - ✓ spring-boot-starter-data-redlis 의존성을 추가
  - ✓ 레디스 명령어 사용 방법
    - org.springframework.data.redis.core의 RedisTemplate 클래스를 사용하는 방법과. Spring Data 프로젝트에서 제공하는 CrudRepository를 확장한 RedisRepository를 사용
    - RedisRepository를 사용하는 방식도 내부에서는 RedisTemplate 스프링 빈에 의존하는데 레디스를 주 데이터 저장소로 사용한다면 RedisRepository 방식을 사용하는 것이 좋지만 데이터 저장소뿐만 아니라 다른 여러 가지 목적으로 활용하는 경우에는 RedisTemplate 클래스를 사용하는 것이 효율적

Redis

- Docker를 이용한 설치 및 접속
  - ✓ 이미지 다운로드
    - docker pull redis
  - ✓ 이미지 확인
    - docker images
  - ✓ 컨테이너 생성 및 실행
    - docker run --name myredis -d -p 6379:6379 redis
  - ✓ 접속
    - docker exec -it myredis /bin/bash
    - redis-cli -h 127.0.0.1

Redis

- Lettuce 라이브러리 와 커넥션 설정
  - ✓ 스프링 부트 애플리케이션은 Lettuce 라이브러리를 기본으로 사용
  - ✓ Lettuce는 내부에 Netty 프레임워크를 포함하고 있어 비동기 논블로킹으로 구현
  - ✓ 스프링 애플리케이션 과 레디스 사이에 커넥션 풀이 필요 없어서 하나의 커넥션을 맺고 사용하면 됨
  - ✓ 비동기 논블로킹이므로 스프링 부트 애플리케이션이 멀티 스레드를 사용하더라도 멀티 스레드에 안전한 프로그래밍이 가능
  - ✓ Lettuce 공식 문서에서도 하나의 커넥션을 만들고 만든 커넥션들을 공유해서 사용하는 방식으로 가이드
  - ✓ 레디스의 트랜잭션 기능을 사용한다면 커넥션 풀을 설정해서 사용하는 것을 권장

Redis

- ❖ Lettuce 라이브러리 와 커넥션 설정
  - ✓ RedisConnetionFactory
    - RestTemplate은 레디스에 명령어를 실행하는 기능을 제공하는 클래스
    - RestTemplate은 레디스와 애플리케이션 사이에 커넥션을 맺고 관리하는 기능을 org.springframework.data.redis.connection 패키지의 RedisConnectionFactory 구현체에 위임
    - RedisConnectionFactory 객체를 생성한 후 RedisTemplate의 setConnectionFactory() 메서드 인자로 설정하면 됨
    - Spring Data Redis에서는 RedisConnectionFactory 인터페이스를 구현한 구현체로 JedisConnectionFactory와 LettuceConnectionFactory 클래스를 기본 제공
    - 애플리케이션에서 사용할 레디스 드라이버 라이브러리에 따라 구현체를 선택
    - 레디스 서버는 아키텍처에 따라 레디스 서버 단독 구성, 레디스 센티넬 구성, 레디스 클러스터 구 성으로 구분할 수 있는데 사용하는 아키텍처에 따라 LettuceConnectionFactory 객체를 설
    - 스프링 프레임워크는 org.springframework.data.redis.connection 패키지의 RedisConfiguration 인터페이스를 제공하는데 레디스 설정 정보를 포함하는 인터페이스이며 아키텍처에 따른 구현체들을 제공
    - 적절한 구현체를 생성 및 설정한 후 LettuceConnectionFactory 생성자의 인자로 전달

Redis

- ❖ Lettuce 라이브러리 와 커넥션 설정
  - ✓ RedisConnetionFactory
    - ● RedisConfiguration 인터페이스가 제공하는 구현 클래스
      - ◆ RedisStandaloneConfiguration: 단독으로 구성된 레디스 서버에 커넥션을 맺을 때 사용
      - ◆ RedisStaticMasterReplicaConfiguration: 레디스 마스터 레플리카 구조로 구성된 레디스 서버에 커넥션을 맺을 때 사용
      - ◆ RedisSocketConfiguration: 유닉스 도메인 소켓을 이용하여 애플리케이션과 같은 로컬 호스트에 설치된 레디스 서버에 커넥션을 맺을 때 사용하는데 TCP 소켓을 사용하는 것보다 빠르며 메모리 사용량이 적은 장점이 있음
      - ◆ RedisSentinelConfiguration: 레디스 센티넬 구조로 구성된 레디스 센티넬 서버에 커넥션을 맺을 때 사용
      - ◆ RedisClusterConfiguration: 레디스 클러스터 구조로 구성된 레디스 서버에 커넥션을 맺을 때 사용

Redis

- ❖ Lettuce 라이브러리 와 커넥션 설정
  - ✓ RedisConnetionFactory
    - RedisStandaloneConfiguration 사용 설정

```
@Configuration
public class CacheConfig {
@Bean
public RedisConnectionFactory cacheRedisConnectionFactory() {
    RedisStandaloneConfiguration configuration = new
    RedisStandaloneConfiguration ("127.0.0.1", 6379); °
    configuration. setDatabase(0);
    configuration. setUsername("username");
    configuration. setPassword("password");
    final Socketoptions socketoptions = SocketOptions.builder()
    . connectimeout(Duration.ofSeconds(10)).build();
    final Clientoptions clientoptions = ClientOptions.builder()
    .socketOptions(socketOptions).build();
```

Redis

- ❖ Lettuce 라이브러리 와 커넥션 설정
  - ✓ RedisConnetionFactory
    - ● RedisStandaloneConfiguration 사용 설정

```
LettuceClientConfiguration lettuceClientConfiguration =
  LettuceClientConfiguration.builder()
  .clientoptions(clientoptions)
  . commandTimeout(Duration.ofSeconds(5))
  .shutdownTimeout(Duration.ZERO)
  .build();
return new LettuceConnectionFactory(configuration, lettuceClientConfiguration);
}
// 생략
}
```

Redis

- ❖ Lettuce 라이브러리 와 커넥션 설정
  - ✓ RedisConnetionFactory
    - ● RedisSentinelConfiguration 사용 설정

```
@Bean
public RedisConnectionFactory cacheRedisConnectionFactory() {
    RedisSentinelConfiguration configuration = new RedisSentinelConfiguration();
    configuration setMaster("REDIS_MASTER_NAME");
    configuration sentinel("127.0.0.1", 19999);
    configuration sentinel("127.0.0.1", 19998);
    configuration sentinel("127.0.0.1", 19997);
    configuration setPassword("password");

    return new LettuceConnectionFactory(configuration);
}
```

Redis

- ❖ Lettuce 라이브러리 와 커넥션 설정
  - ✓ RedisConnetionFactory
    - ● RedisClusterConfiguration 사용 설정

```
@Bean
public RedisConnectionFactory cacheRedisConnectionFactory() {
    RedisClusterConfiguration config = new RedisClusterConfiguration();
    config. setMaxRedirects(3);
    config.setClusterNodes(List.of(
       new RedisNode("127.0.0.1", 19999),
       new RedisNodeC 127.0,0.1", 19998),
       new RedisNode("127.0.0.1", 19997)
    ));
    return new LettuceConnectionFactory(config);
}
```

Redis

- Redis 사용 – property 의 의존성 사용
  - ✓ data-redis, springweb, lombok 라이브러리의 의존성을 설정한 프로젝트 생성

Redis

- Redis 커넥션 설정
  - ✓ Redis 설정 클래스

```
@Configuration
@EnableRedisRepositories
@RequiredArgsConstructor
public class RedisConfig {

    private final RedisProperties redisProperties;

    @Bean
    public RedisConnectionFactory redisConnectionFactory() {
        return new LettuceConnectionFactory(redisProperties.getHost(),
     redisProperties.getPort());
    }

}
```

Redis

- Redis 커넥션 설정
  - ✓ Redis 설정 클래스 생성 및 작성
    - RedisConnectionFactory 인터페이스를 통해 LettuceConnectionFactory를 생성하여 반환

Redis

- RedisTemplate
  - ✓ RedisTemplate은 제네릭 타입 K, V를 설정할 수 있는데 첫 번째 제네릭 타입은 레디스 키 에 해당하며 두 번째 제네릭 타입은 레디스 밸류에 해당
  - ✓ setKeySerializer, setValueSerializer 설정해주는 이유는 RedisTemplate를 사용할 때 Spring - Redis 간 데이터 직렬화, 역직렬화 시 사용하는 방식이 JDK 직렬화 방식이기 때문인데 동작에는 문제가 없지만 redis-cli을 통해 직접 데이터를 보려고 할 때 알아볼 수 없는 형태로 출력되기 때문에 적용한 설정
  - ✓ 직렬화 과정을 거치면 자바 객체가 byte[] 로 변환되어 레디스에 저장되고 역질렬화 거치면 레디스에 저장된 byte[] 가 자바 객체로 변환됨

Redis

- RedisTemplate
  - ✓ RedisConfig 클래스에 RestTemplate 클래스의 bean 생성 코드를 추가

```
@Bean(name="redisTemplate")
public RedisTemplate<String, String> redisTemplate(){
        RedisTemplate<String, String> redisTemplate = new RedisTemplate<>();
        redisTemplate.setConnectionFactory(basicCacheRedisConnectionFactory());
        redisTemplate.setKeySerializer(new StringRedisSerializer());
        redisTemplate.setValueSerializer(new StringRedisSerializer());
        return redisTemplate;
}
```

Redis

- Redis 사용 - CrudRepository
  - ✓ DTO 클래스 생성 – dto.Member

```
@Getter
@Setter
@RedisHash("member")
public class Member {

    @Id
    private String id;
    private String name;
    private int age;

    public Member(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

- Redis를 통해 저장할 Member Class
- @RedisHash 어노테이션을 통해 설정한 값을 Redis의 key 값 prefix로 사용
- @Id 어노테이션은 JPA와 동일한 역할을 수행하는데 member:{id} 의 위치에 자동으로 generate 값이 대입됨

Redis

- Redis 사용 - CrudRepository
  - ✓ Repository 인터페이스 생성 – persistence.MemberRedisRepository

```
public interface MemberRedisRepository extends CrudRepository<Member, String> {
}
```

Redis

- Redis 사용 - CrudRepository
  - ✓ Service 클래스 생성 – service.MemberService

```
@RequiredArgsConstructor
@Service
public class MemberRedisService{
    private final MemberRedisRepository memberRedisRepository;

    public void addMember(){
        Member member = new Member("아담", 45);
        memberRedisRepository.save(member);
    }

    public Member getMember(String id){
        Member member = null;
        Optional<Member> result =  memberRedisRepository.findById(id);
        if(result.isPresent()){
            member = result.get();
        }
        return member;
    }
}
```

Redis

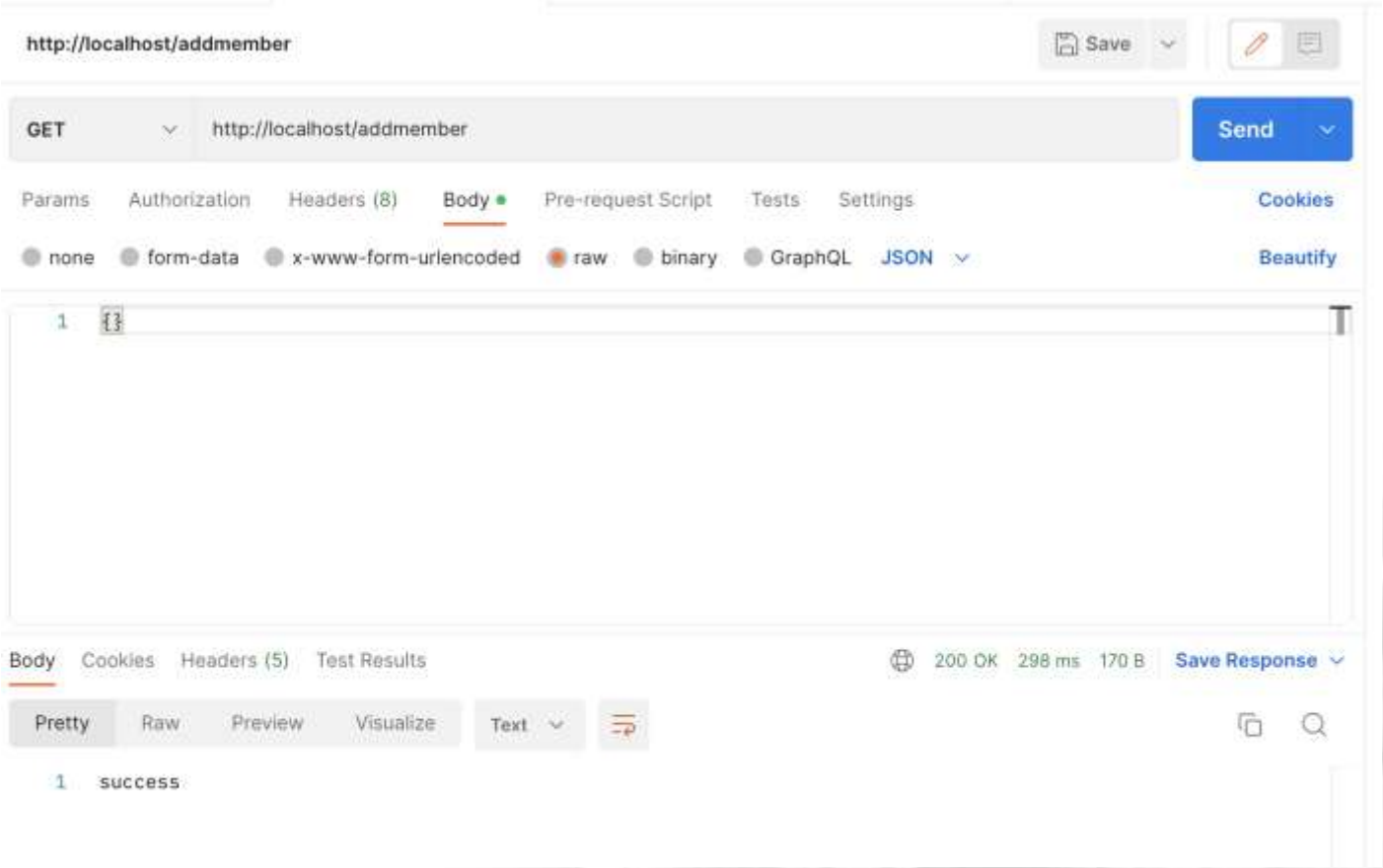
- Redis 사용 - CrudRepository
  - ✓ Controller 클래스에 처리 코드 추가

```
@RestController
@RequiredArgsConstructor
public class MemberRedisController {
    private final MemberRedisService memberRedisService;

    @PostMapping("/redis")
    public ResponseEntity<String> add(){
        memberRedisService.addMember();
        return new ResponseEntity<>("success", HttpStatus.OK);
    }
    @GetMapping("/redis/{id}")
    public ResponseEntity<Member> get(@PathVariable String id){
        Member member = memberRedisService.getMember(id);
        return new ResponseEntity<>(member, HttpStatus.OK);
    }
}
```

Redis

- Redis 사용 - CrudRepository
  - ✓ 확인



Redis

- Redis 사용 - CrudRepository
  - ✓ 확인

```
redis:6379> keys *
 1) "name"
 2) "key5"
 3) "member"
 4) "green"
 5) "key1"
 6) "member:0c0f8007-00f4-4a28-8a0e-02c06615e376"
 7) "key"
 8) "red"
 9) "a"
10) "yellow"
11) "key3"
12) "key2"
13) "b"
redis:6379> hgetall member:0c0f8007-00f4-4a28-8a0e-02c06615e376
1) "_class"
2) "com.adamsoft.jpa.dto.Member"
3) "age"
4) "99"
5) "id"
6) "0c0f8007-00f4-4a28-8a0e-02c06615e376"
7) "name"
8) "jan"
redis:6379>
```

Redis

- Redis 사용 - RedisTemplate
  - ✓ 문자열 사용
    - org.springframework.data.redis.core.ValueOperation<K, V> 인터페이스는 레디스의 문자열 자료 구조를 사용할 수 있는 기능을 제공
    - RedisTemplate의 opsForValue() 메서드를 사용하면 ValueOperations 객체를 획득할 수 있는데 ValueOperations의 제네릭 클래스 타입은 RedisTemplate의 제네릭 설정을 따름
    - 사용자가 RedisTemplate<HotelCacheKey, HotelCacheValue> 스프링 빈을 정의하면 opsForValue() 메서드가 리턴하는 클래스 타입은 ValueOperations<ㅗotelCacheKey, HotelCacheValue>가 됨

```
ValueOperations<HotelCacheKey, HotelCacheValue> hotelCacheOperation =
hotelCacheRedisTemplate.opsForValue();
```

Redis

- Redis 사용 - RedisTemplate
  - ✓ 문자열 사용
    - ValueOperations에서 제공하는 메서드
      - void set(K key, V value): key와 value를 저장
      - void set(K key, V value, long timeout, TimeUnit unit): key와 value를 저장할 때 동시에 유효 기간을 설정하는데 TimeUnit 값과 timeout 인자를 사용하여 유효 기간을 설정
      - Boolean setIfAbsent(K key, V value): key와 매칭되는 데이터가 없으면 value를 저장하고 메서드 실행에 성공하면 true를 실패하면 false를 리턴
      - Boolean setIfAbsent(K key, V value, long timeout, TimeUnit unit): setIfAbsent()와 기능이 같지만 데이터를 저장할 때는 timeout과 timeunit 값을 사용하여 유효 기간을 설정
      - Boolean setIfPresent(K key, V value): setIfAbsent()와 반대로 key와 매칭되는 데이터가 있으면 데이터를 덮어쓰고 메서드 실행에 성공하면 true를 실패하면 false를 리턴
      - Boolean setIfPresent(K key, V value, long timeout, TimeUnit unit): setIfPresent()와 기능이 같지만 데이터를 저장할 때는 timeout과 timeunit 값을 사용하여 유효 기간을 설정

Redis

- Redis 사용 - RedisTemplate
  - ✓ 문자열 사용
    - ValueOperations에서 제공하는 메서드
      - V getAndDelete(K key): key와 매칭되는 데이터를 조회하여 리턴하는 동시에 데이터를 삭제
      - V getAndExpire(K key, long timeout, TimeUnit unit): key와 매칭되는 데이터를 조회하면서 데이터의 유효 기간을 설정
      - V getAndSet(K key, V value): key와 매칭되는 이전 데이터를 조회하고 리턴gkrh 인자로 받은 value를 새로 저장
      - Long increment(K key): 저장된 문자열이 숫자라면 1을 증가
      - Double increment(K key, double delta): delta 인자만큼 증가시키고 Long 타입으로 리턴
      - Long decrement(K key): 저장된 문자열이 숫자라면 1을 감소
      - Long decrement(K key, long delta): delta 인자만큼 감소

Redis

- Redis 사용 - RedisTemplate
  - ✓ Controller 클래스에 추가하고 테스트

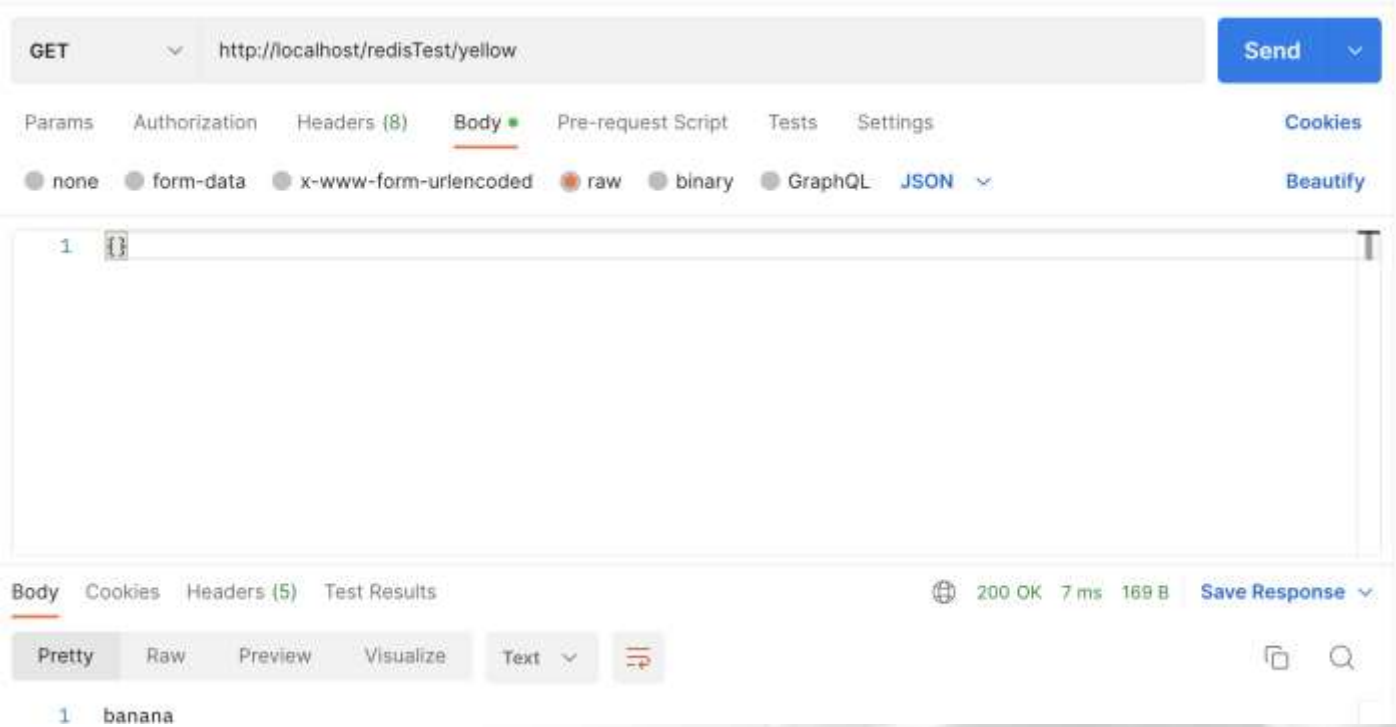
```
@Autowired
@Qualifier("redisTemplate")
private RedisTemplate<String, String> redisTemplate;

@PostMapping("/redisTest")
public ResponseEntity<?> addRedisKey() {
        System.out.println(redisTemplate);
        ValueOperations<String, String> vop = redisTemplate.opsForValue();
        vop.set("yellow", "banana");
        vop.set("red", "apple");
        vop.set("green", "watermelon");
        vop.set("blue", "블루베리", Duration.ofSeconds(10)); //유효 시간이 10초
        return new ResponseEntity<>(HttpStatus.CREATED);
}

@GetMapping("/redisTest/{key}")
public ResponseEntity<?> getRedisKey(@PathVariable String key) {
        ValueOperations<String, String> vop = redisTemplate.opsForValue();
        String value = vop.get(key);
        return new ResponseEntity<>(value, HttpStatus.OK);
}
```

Redis

- Redis 사용 - RedisTemplate
  - ✓ Controller 클래스를 만들고 테스트




```
redis:6379> get "yellow"
"banana"
```

Redis

❖ Redis 사용 - RedisTemplate

✓ 토큰 저장

```
String refreshToken = jwtTokenProvider.createRefreshToken();
ValueOperations<String, String> valueOperations = redisTemplate.opsForValue();
valueOperations.set(rRTKey + user.getIdx(), refreshToken);
log.info("redis RT : {}", valueOperations.get(rRTKey + user.getIdx()));
```

- Service 계층에서 사용하였고 RedisTemplate을 생성자 주입 방식으로 추가하여 사용
- 프로젝트에서 로그인 시 RefreshToken을 cache 서버인 Redis에 저장할 용도로 사용하였고, 'rRTKey'라는 값은 따로 정의하여 가져온 String 값

Redis

- Redis 사용 - RedisTemplate
  - ✓ 자료 구조에 따른 메서드
    - opsForValue　　Strings를 쉽게 Serialize / Deserialize
    - opsForList　　List를 쉽게 Serialize / Deserialize
    - opsForSet　　Set을 쉽게 Serialize / Deserialize
    - opsForZSet　　ZSet을 쉽게 Serialize / Deserialize
    - opsForHash　　Hash를 쉽게 Serialize / Deserialize

Redis

- 분산 락
  - ✓ MSA 환경에서 컴포넌트들은 고가용성을 확보하고자 스케일아웃 방식을 주로 사용
  - ✓ 컴포넌트마다 적어도 두 대 이상의 인스턴스가 필요한데 이때 여러 인스턴스가 동시에 공유 자원을 사용하는 일이 발생할 수 있음
  - ✓ 동시성 문제를 해결하지 않고 공유 자원을 작업하면 공유 자원의 무결성이 깨질 수 있기 때문에 개발자가 원하는 결과를 얻을 수 없음
  - ✓ MSA 환경에서는 분산 락을 사용하여 공유 자원의 원자성을 보장하도록 설계
  - ✓ 분산 락의 기본 개념은 원자성(atomic)을 제공하는 저장소를 사용하여 동시성 문제를 해결하는 것
  - ✓ 공유 자원에 작업하기 전에 저장소에 락을 생성하고 작업을 마치고 생성한 락을 제거
  - ✓ 락을 생성할 때 다른 인스턴스가 생성한 락이 있다면 공유 자원에 작업하지 않는데 그러면 작업 시간 동안 다른 인스턴스들은 공유 자원에 접근할 수 없으므로 공유 자원은 한 번에 한 인스턴스만 점유하여 처리할 수 있으므로 데이터가 무결성을 가질 수 있음
  - ✓ 싱글 스레드 방식의 빠 른 처리 속도를 제공하는 레디스가 락 저장소의 매우 적합한 대안이 될 수 있음
  - ✓ 레디스 라이브러리 중 Redisson은 분산 락을 처리할 수 있는 메서드를 제공하지만 Lettuce는 분산 락을 만들 수 있는 별도의 기능을 제공하지 않음

Redis

- ❖ 분산 락
  - ✓ 분산 락 실습
    - ● RedisTemplet을 생성해주는 RedisConfig 클래스에 추가

```
@Bean(name = "lockRedisTemplate")
public RedisTemplate<String, Long> lockRedisTemplate() {
        RedisTemplate<String, Long> lockRedisTemplate = new RedisTemplate<>();

    lockRedisTemplate.setConnectionFactory(basicCacheRedisConnectionFactory());
        lockRedisTemplate.setKeySerializer(new
    GenericToStringSerializer<>(Long.class));
        lockRedisTemplate.setValueSerializer(new
    GenericToStringSerializer<>(Long.class));
        return lockRedisTemplate;
}
```

Redis

- ❖ 분산 락
  - ✓ 분산 락 실습
    - RedisTemplet을 사용하는 LockAdapter 클래스 생성

```
@Component
@Slf4j
public class LockAdapter {

    @Autowired
    @Qualifier("lockRedisTemplate")
    private final RedisTemplate<String, Long> lockRedisTemplate;

    private ValueOperations<String, Long> lockOperation;

    public LockAdapter(RedisTemplate<String, Long> lockRedisTemplate) {
        this.lockRedisTemplate = lockRedisTemplate;
        this.lockOperation = lockRedisTemplate.opsForValue();
    }

    public Boolean holdLock(String hotelId, Long userId) {
        String lockKey = "key";
        return lockOperation.setIfAbsent(lockKey, userId, Duration.ofSeconds(10));
    }
```

Redis

- ❖ 분산 락
  - ✓ 분산 락 실습
    - RedisTemplet을 사용하는 LockAdapter 클래스 생성

```
    public Long checkLock(String hotelId) {
        String lockKey = "key";
        return lockOperation.get(lockKey);
    }

    public void clearLock(String hotelId) {
        lockRedisTemplate.delete("key");
    }
}
```

Redis

- ❖ 분산 락
  - ✓ 분산 락 실습
    - ● Test를 위한 LockAdapterTest 클래스를 생성하고 작성

```
@SpringBootTest
public class LockAdapterTest {

    private final Long firstUserId = 1L;
    private final Long secondUserId = 2L;
    private final Long thirdUserId = 3L;

    @Autowired
    private LockAdapter lockAdapter;
```

Redis

- ❖ 분산 락
  - ✓ 분산 락 실습
    - ● Test를 위한 LockAdapterTest 클래스를 생성하고 작성

```
@Test
@DisplayName("firstUserId가 락을 선점한다.")
public void testLock() {
    final String hotelId = "123123123L";

    Boolean isSuccess = lockAdapter.holdLock(hotelId, firstUserId);
    Assertions.assertTrue(isSuccess);
    System.out.println(isSuccess);

    isSuccess = lockAdapter.holdLock(hotelId, secondUserId);
    Assertions.assertFalse(isSuccess);
    System.out.println(isSuccess);

    isSuccess = lockAdapter.holdLock(hotelId, thirdUserId);
    Assertions.assertFalse(isSuccess);
    System.out.println(isSuccess);

    Long holderId = lockAdapter.checkLock(hotelId);
    Assertions.assertEquals(firstUserId, holderId);
}
```

Redis

- ❖ 분산 락
  - ✓ 분산 락 실습
    - ● Test를 위한 LockAdapterTest 클래스를 생성하고 작성

```
@Test
@DisplayName("3명이 동시에 락을 선점하지만 1명만 락을 잡는다.")
public void testConcurrentAccess() throws InterruptedException {

    final String hotelId = "9999999";
    lockAdapter.clearLock(hotelId);

    CyclicBarrier cyclicBarrier = new CyclicBarrier(3);
    new Thread(new Accessor(hotelId, firstUserId, cyclicBarrier)).start();
    new Thread(new Accessor(hotelId, secondUserId, cyclicBarrier)).start();
    new Thread(new Accessor(hotelId, thirdUserId, cyclicBarrier)).start();
    TimeUnit.SECONDS.sleep(1);

    Long holderId = lockAdapter.checkLock(hotelId);
    System.out.println(holderId);
    Assertions.assertTrue(List.of(firstUserId, secondUserId,
 thirdUserId).contains(holderId));

    lockAdapter.clearLock(hotelId);
}
```

Redis

- 분산 락
  - ✓ 분산 락 실습
    - Test를 위한 LockAdapterTest 클래스를 생성하고 작성

```
class Accessor implements Runnable {
    private String hotelId;
    private Long userId;
    private CyclicBarrier cyclicBarrier;
    public Accessor(String hotelId, Long userId, CyclicBarrier cyclicBarrier) {
        this.hotelId = hotelId;
        this.userId = userId;
        this.cyclicBarrier = cyclicBarrier;
    }
    @Override
    public void run() {
        try {
            cyclicBarrier.await();
            lockAdapter.holdLock(hotelId, userId);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
}
```

Redis

- Redis Sorting
  - ✓ Sorted Set(ZSet)
    - Sorted Set에는 키 와 밸류, 밸류 와 연관된 스코어(score)를 저장할 수 있으며 저장된 스코어를 기준으로 데이터를 정렬할 수 있음
    - 스코어를 기준으로 내림차순 또는 오름차순으로 정렬할 수 있으며 offset과 count를 사용하여 특정 순위에 있는 데이터를 부분 조회할 수 있음
    - 순위별 로 특정 값을 조회하거나 비교하는 기능도 제공
    - 이 자료 구조를 이용하면 경매 기능을 쉽게 구현할 수 있으며 실시간 인기 검색어 나 게임의 사용자 랭킹 같은 기능에도 활용할 수 있음
    - RedisTemplate에서 Sorted Set 자료 구조를 사용하려면 ZSetOperations 객체를 사용
    - RedisTemplate 객체의 opsForZSet() 메서드는 ZSetOperations 객체를 리턴

      ```
      ZSetOperations<String, Long> ZSetOperations =
          biddingRedisTemplate.opsForZSet();
      ```

    - ZSetOperations는 밸류 값과 스코어 값을 저장하기 위해 TypedTuple이라는 튜플 클래스를 사용하는데 밸류의 클래스 타입을 지정할 수 있는 제네릭 타입을 설정

Redis

- ❖ Redis Sorting
  - ✓ Sorted Set(ZSet)
    - ● ZSetOperations 의 메서드
      - ◆ Boolean add(K key, V value, double score): 키와 매핑되는 Sorted Set에 밸류와 스코어를 함께 저장
      - ◆ Boolean addIfAbsent(K key, V value, double score): 키와 매핑되는 Sorted Set에 저장된 밸류가 없다면 밸류 와 스코어를 함께 저장
      - ◆ Long remove(K key, Object... values): 키와 매핑되는 Sorted Set에 values 인자와 매핑 되는 데이터들을 삭제
      - ◆ Double incrementScore(K key, V value, double delta): 키와 매핑되는 Sorted Set에 밸류와 연관된 스코어를 delta만큼 증가
      - ◆ Long rank(K key. Object o): 키와 매핑되는 Sorted Set에서 밸류의 순위 즉 인덱스 값을 리턴
      - ◆ Long reverseRank(K keyz Object o): 키와 매핑되는 Sorted Set의 밸류 순위 즉 인덱스 값을 리턴하는데 인덱스 값은 역순
      - ◆ Double score(K key. Object o): 키와 매핑되는 Sorted Set에서 밸류의 스코어를 조회

Redis

- ❖ Redis Sorting
  - ✓ Sorted Set(ZSet)
    - ● ZSetOperations 의 메서드
      - ◆ Set<TypedTuple<V>> rangeByScoreWithScores(K key, double min, double max): 키와 매핑되는'Sorted Set에서 스코어 최솟값(min), 최댓값(max) 사이에 있는 TypedTuple 객체 셋을 리턴하는데 입찰 금액이 10.000과 20,000 사이에 있는 사용자 정보를 조회할 때 유용
      - ◆ Set<V> rangeByScore(K key, double min, double max): 키와 매핑되는 Sorted Set에서 스코어 최솟값min) 과 최댓값m:ix) 사이에 있는 밸류 셋을 리턴
      - ◆ Set<V> rangeByScore(K key, double min, double max, long offset, long count): 키와 매핑되는 Sorted Set에서 스코어 최솟값(min) 과 최댓값(max) 사이에 있는 밸류 셋 중 특정 위치(offset)에서 특정 개수(count)만큼 밸류 셋을 리턴
      - ◆ Set<V> reverseRangeByScore(K key, double min, double max, long offset, long count): 키와 매핑되는 Sorted Set에서 스코어 최솟값 과 최댓값 사이에 있는 밸류 셋 중 특정 위치에서 특정 개수만큼 밸류 셋을 리턴하는데 스코어의 역순으로 정렬한 값을 리턴

Redis

- ❖ Redis Sorting
  - ✓ 경매
    - RedisTemplate을 생성해주는 코드를 RedisConfig 클래스에 추가

```
@Bean(name = "biddingRedisTemplate")
public RedisTemplate<String, Long> rankRedisTemplate() {
        RedisTemplate<String, Long> rankRedisTemplate = new RedisTemplate<>();

    rankRedisTemplate.setConnectionFactory(basicCacheRedisConnectionFactory());
       rankRedisTemplate.setKeySerializer(new StringRedisSerializer());
       rankRedisTemplate.setValueSerializer(new
    GenericToStringSerializer<>(Long.class));

       return rankRedisTemplate;
}
```

Redis

- ❖ Redis Sorting
  - ✓ 경매
    - RedisTemplate을 한 서비스 클래스 작성 – service.BiddingAdapter

```
@Component
@Slf4j
public class BiddingAdapter {

    private String PREFIX = "BIDDING::";

    @Autowired
    @Qualifier("biddingRedisTemplate")
    private RedisTemplate<String, Long> biddingRedisTemplate;

    public Boolean createBidding(Long hotelId, Long userId, Double amount) {
        String key = this.serializeKey(hotelId);
        return biddingRedisTemplate.opsForZSet().add(key, userId, amount);
    }
```

Redis

- ❖ Redis Sorting
  - ✓ 경매
    - RedisTemplate을 한 서비스 클래스 작성 – service.BiddingAdapter

```
public List<Long> getTopBidders(Long hotelId, Integer fetchCount) {
        String key = this.serializeKey(hotelId);
        return biddingRedisTemplate
                .opsForZSet()
                .reverseRangeByScore(key, 0D, Double.MAX_VALUE, 0, fetchCount)
                .stream()
                .collect(Collectors.toList());
    }

    public Double getBidAmount(Long hotelId, Long userId) {
        String key = this.serializeKey(hotelId);
        return biddingRedisTemplate.opsForZSet().score(key, userId);
    }

    public void clear(Long hotelId) {
        String key = this.serializeKey(hotelId);
        biddingRedisTemplate.delete(key);
    }
```

Redis

- ❖ Redis Sorting
  - ✓ 경매
    - ● RedisTemplate을 한 서비스 클래스 작성 – service.BiddingAdapter

```
        private String serializeKey(Long hotelId) {
            return new StringBuilder(PREFIX).append(hotelId).toString();
        }
    }
```

Redis

- ❖ Redis Sorting
  - ✓ 경매
    - RedisTemplate을 한 서비스 클래스 작성 – service.BiddingAdapter
      - 이벤트에 참가하는 호텔 아이디를 사용하여 Sorted Set의 키를 생성하는 함수로 별도의 RedisSerializer 구현체를 생성하지 않고 개발
      - 경매에 참여할 호텔 아이디와 참여하는 사용자 아이디, 비딩 금액을 각각 인자로 받는 함수로 ZSetOperations의 add() 메서드를 사용하여 호텔 아이디를 포함한 키, 사용자 아이디에 밸류 와 비딩 금액을 스코어로 저장
      - 경매에 참여한 사용자들의 비딩 금액을 역순으로 정렬한 후 fetchCount만큼 참가자의 사용자 아이디를 리스트 객체로 리턴
      - 비딩 금액이 높은 순으로 정렬해야 하므로 스코어를 역순으로 조회하는 reverseRangeByScore() 메서드를 사용하는데 이때 스코어 범위를 인자로 설정할 수 있으며 두 번째 인자인 최솟값 0D부터 세 번째 인자인 최댓값 Double.MAXVALUE까지 설정하며 네 번째 인자는 순서 인덱스로 0부터 fetchCount 개수만큼 조회
      - 참여자가 입찰한 금액을 조회하는데 메서드 내부의 score() 메서드는 스코어 값을 조회하므로 스코어에 저장한 입찰 금액을 리턴

Redis

- ❖ Redis Sorting
  - ✓ 경매
    - ● 테스트 클래스 작성 – BiddingAdapterTests

```
@SpringBootTest
class BiddingAdapterTest {

    private final Long firstUserId = 1L;
    private final Long secondUserId = 2L;
    private final Long thirdUserId = 3L;
    private final Long fourthUserId = 4L;
    private final Long fifthUserId = 5L;
    private final Long hotelId = 1000L;

    @Autowired
    private BiddingAdapter biddingAdapter;
```

Redis

- ❖ Redis Sorting
  - ✓ 경매
    - ● 테스트 클래스 작성 – BiddingAdapterTests

```
@Test
public void simulate() {
    biddingAdapter.clear(hotelId);

    //100d에서 140d까지 총 5명의 사용자가 각각 입찰
    // biddingAdapter.createBidding() 메서드는 Sorted Set 자료 구조에
 데이터를 생성
    biddingAdapter.createBidding(hotelId, firstUserId, 100d);
    biddingAdapter.createBidding(hotelId, secondUserId, 110d);
    biddingAdapter.createBidding(hotelId, thirdUserId, 120d);
    biddingAdapter.createBidding(hotelId, fourthUserId, 130d);
    biddingAdapter.createBidding(hotelId, fifthUserId, 140d);

    //첫 번째와 두 번째 사용자는 재입찰
    //첫 번째 사용자는 입찰 금액을 100d에서 200d 로 재입찰하고 두 번째
 사용자는 11Od에서 150d로 재입찰
    biddingAdapter.createBidding(hotelId, secondUserId, 150d);
    biddingAdapter.createBidding(hotelId, firstUserId, 200d);
```

Redis

- ❖ Redis Sorting
  - ✓ 경매
    - ● 테스트 클래스 작성 – BiddingAdapterTests

```
        //입찰자 중 입찰 금액 최상위 3명만 조회
        List<Long> topBidders = biddingAdapter.getTopBidders(hotelId, 3);

        //최상위 3명이 첫 번째 사용자(firstUserId), 두 번째 사용자(secondUserId),
      다섯 번째 사용자(fifthUserId)인지 확인
        Assertions.assertEquals(firstUserId, topBidders.get(0));
        Assertions.assertEquals(secondUserId, topBidders.get(1));
        Assertions.assertEquals(fifthUserId, topBidders.get(2));

        //최상위 3명이 입찰한 금액이 정확한지 다시 확인
        Assertions.assertEquals(200d, biddingAdapter.getBidAmount(hotelId,
      firstUserId));
        Assertions.assertEquals(150d, biddingAdapter.getBidAmount(hotelId,
      secondUserId));
        Assertions.assertEquals(140d, biddingAdapter.getBidAmount(hotelId,
      fifthUserId));
      }
  }
```

Redis

- ❖ Redis Sorting
  - ✓ 경매
    - ● 확인

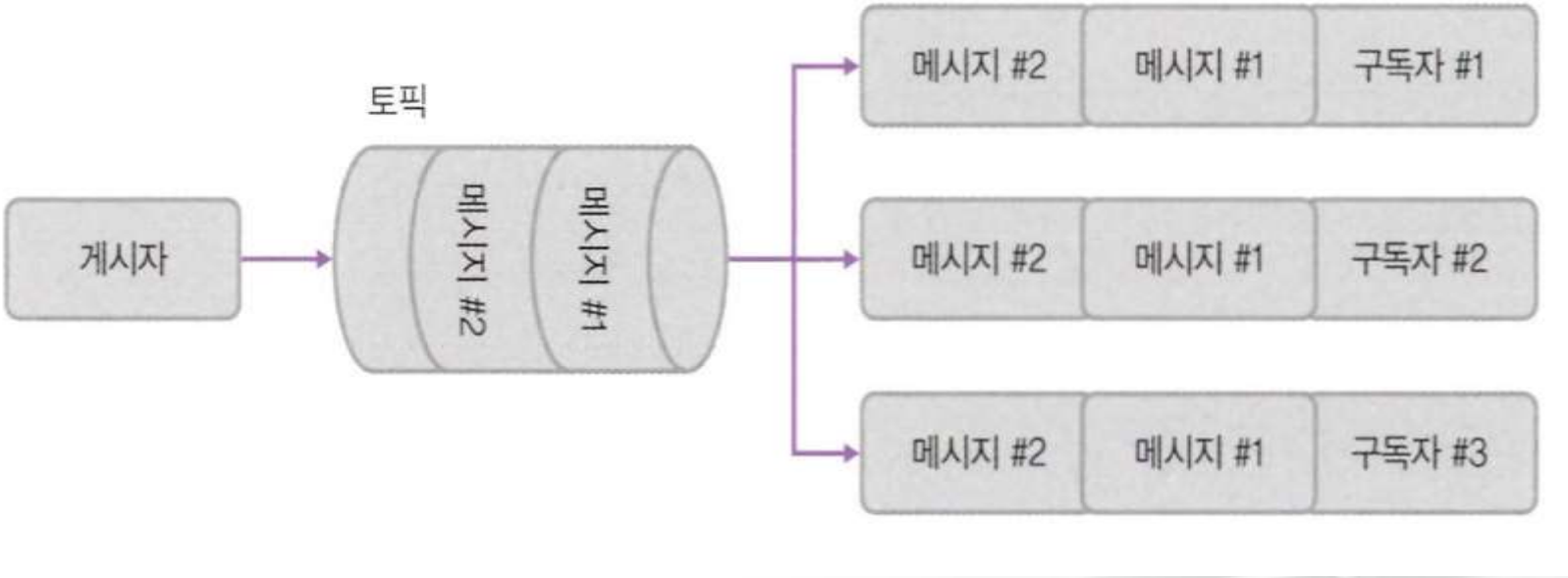
```
redis:6379> keys *
 1) "green"
 2) "member"
 3) "red"
 4) "yellow"
 5) "BIDDING::1000"
 6) "member:05e81893-76a8-49ea-9864-f829142e805b"
redis:6379> zrange "BIDDING::1000" 0 -1
 1) "3"
 2) "4"
 3) "5"
 4) "2"
 5) "1"
```

Redis

- Redis Pub-Sub 구현
  - ✓ 레디스는 Pub-Sub 기능을 제공
  - ✓ 일반적으로 서버 사이에 이벤트 데이터 를 전송 - 수신하는 일련의 과정을 메시징(messaging)이라고 하는데 Pub-Sub 기능은 이런 메시징 기능 중 하나
  - ✓ Pub-Sub 메시징 기능에 필요한 요소
    - 서버 사이에 전달되는 데이터인 메시지
    - 메시지 데이터를 생산하는 게시자(publisher)
    - 생산된 이벤트를 수신하는 구독자(subscriber)
    - 이들 사이에서 메시지를 큐잉하고 구독자가 메시지를 수신할 수 있는 토픽(topic)
  - ✓ 애플리케이션 서버들은 게시자 와 구독자가 될 수 있으며 구독자는 하나 이상 구성할 수 있음
  - ✓ 레디스는 게시자 와 구독자들이 메시지를 주고받을 수 있는 토픽을 제공하며 게시자 와 구독자는 Lettuce 라이브러리를 사용하여 토픽과 커넥션을 맺고 메시지를 송수신할 수 있음
  - ✓ Pub-Sub의 가장 큰 특징은 토픽을 구독하고 있는 모든 구독자가 메시지를 수신할 수 있다는 것

Redis

- Redis Pub-Sub 구현
  - ✓ 레디스 Pub-Sub



Redis

- Redis Pub-Sub 구현
  - ✓ 레디스 Pub-Sub 동작
    - 레디스는 하나 이상의 토픽을 설정할 수 있는데 이때 메시지 종류나 성질에 따라 여러 개의 토픽을 구성하는 것도 가능
    - 게시자는 적합한 토픽에 메시지를 게시하고 토픽은 게시된 메시지를 FIFO 형태로 관리
    - 게시된 메시지는 토픽에 연결된 모든 구독자에게 전달되는데 게시자가 순서대로 생산한 메시지 #1. 메시지 #2 가 구독자 #1~#3 모두에게 전달
    - 토픽을 수신하는 모든 구독자 에게 메시지를 FIFO 방식으로 전달
    - 레디스가 제공하는 Pub-Sub은 Kafka나 RabbitMQ 같은 메시지 큐에 비해 비교적 간단하고 빠른 기능을 제공하는데 이들과 다른 점은 구독자가 메시지 수신에 실패해도 레디스는 실패한 메시지를 재전달 할 수 없음

Redis

- Redis Pub-Sub 구현
  - ✓ 토픽 과 메시지 객체
    - 메시지 객체 – service.EventMessage

```
@Getter
@ToString
public class EventMessage {

    private Long timestamp;
    private String message;

    //게시자의 메서드
    public EventMessage(String message) {
        this.timestamp = System.currentTimeMillis();
        this.message = message;
    }

    //구독자의 메서드
    @JsonCreator
    public EventMessage(@JsonProperty("timestamp") Long timestamp,
                        @JsonProperty("message") String message) {
        this.timestamp = timestamp;
        this.message = message;
    }
}
```

Redis

- Redis Pub-Sub 구현
  - ✓ 토픽 과 메시지 객체
    - 메시지 객체 – service.EventMessage
      - 게시자가 메시지를 생성할 때 생성자 내부에서 timestamp 값을 설정
      - @JsonCreator와 @JsonProperty 애너테이션을 사용하여 JSON 메시지를 EventMessage 객체로 변환할 때 사용하는 생성자를 만들었는데 구독자 쪽에서 사용하는 생성자

Redis

- ❖ Redis Pub-Sub 구현
  - ✓ 토픽 과 메시지 객체
    - RedisConfig 파일에 추가

```
@Bean
public ChannelTopic eventTopic() {
        return new ChannelTopic("dummyTopic");
}

@Bean(name="eventRedisTemplate")
public RedisTemplate<String, String> eventRedisTemplate() {
        RedisTemplate<String, String> redisTemplate = new RedisTemplate<>();
        redisTemplate.setConnectionFactory(basicCacheRedisConnectionFactory());
        redisTemplate.setKeySerializer(new StringRedisSerializer());
        redisTemplate.setValueSerializer(new
    Jackson2JsonRedisSerializer<EventMessage>(EventMessage.class));
        return redisTemplate;
}
```

Redis

- Redis Pub-Sub 구현
  - ✓ 게시자
    - 메시지를 게시할 때 RedisTemplate의 convertAndSend() 메서드를 사용하기 때문에 RedisTemplate 스프링 빈을 정의
    - RedisTemplate 스프링 빈 키 와 밸류의 클래스 타입은 String으로 설정
    - JSON 문자열을 사용하여 메시지를 게시 및 구독하므로 밸류의 클래스 타입은 String

Redis

- ❖ Redis Pub-Sub 구현
  - ✓ 게시자
    - ● 메시지를 전송하기 위한 EventPublisher 클래스 생성

```
@Slf4j
@Component
public class EventPublisher {

    private final RedisTemplate<String, String> eventRedisTemplate;
    private final ChannelTopic eventTopic;

    public EventPublisher(RedisTemplate<String, String> eventRedisTemplate,
     ChannelTopic eventTopic) {
        this.eventRedisTemplate = eventRedisTemplate;
        this.eventTopic = eventTopic;
    }

    public void sendMessage(EventMessage eventMessage) {
        eventRedisTemplate.convertAndSend(eventTopic.getTopic(), eventMessage);
    }
}
```

Redis

- ❖ Redis Pub-Sub 구현
  - ✓ 게시자
    - ● 메시지를 전송하기 위한 EventPublisher 클래스 생성
      - ◆ EventPublisher 스프링 빈은 eventRedisTemplate과 eventTopic 스프링 빈을 주입받아야 함
      - ◆ RedisTemplate의 convertAndSencK) 메서드는 토픽 이름과 메시지를 인자로 받아서 토픽에 메시지를 전달
      - ◆ convertAndSend() 메서드 내부에는 RestTemplate의 값을 변환하는 RedisSerializer valueSerializer로 메시지가 변환되고 이 valueSerializer를 설정하려면 RedisTemplate의 setValueSerializer() 메서드를 사용

Redis

- Redis Pub-Sub 구현
  - ✓ 구독자
    - 구독자는 레디스 서버의 토픽을 주기적으로 listen하면서 새로운 메시지가 있는지 확인
    - Spring Data JPA는 이 과정을 쉽게 구현할 수 있는 클래스와 인터페이스를 제공
    - org.springframework.data,redis.listener 패키지의 RedisMessageListenerContainer 클래스와 org.springframework.data.redis.connection 의 MessageListener 인터페이스를 사용
    - RedisMessageListenerContainer는 레디스에서 메시지를 수신하는 기능을 제공하며 구독자에게 메시지를 전달하는 기능을 제공하는데 MessageListener 인터페이스의 onMessage() 메서드를 호출하고 인자 Message 객체를 주입하므로 레디스에서 전달받은 Message 객체를 사용하여 구독자가 필요한 비즈니스 로직을 개발

Redis

- Redis Pub-Sub 구현
  - ✓ 구독자
    - MessageListener 인터페이스 생성 – service.MessageListener

```
@Slf4j
public class EventListener implements MessageListener {
    private RedisTemplate<String, String> eventRedisTemplate;
    private RedisSerializer<EventMessage> valueSerializer;

    public EventListener(RedisTemplate<String, String> eventRedisTemplate) {
        this.eventRedisTemplate = eventRedisTemplate;
        this.valueSerializer = (RedisSerializer<EventMessage>)
     eventRedisTemplate.getValueSerializer();
    }

    @Override
    public void onMessage(Message message, byte[] pattern) {

        EventMessage eventMessage =
     valueSerializer.deserialize(message.getBody());
        log.warn("Subscribe Channel : {}", new String(message.getChannel()));
        log.warn("Subscribe Message : {}", eventMessage.toString());
    }
}
```

Redis

- ❖ Redis Pub-Sub 구현
  - ✓ 구독자
    - MessageListener 인터페이스 생성 – service.MessageListener
      - ◆ EventListener.java는 @Bean 애너테이션을 사용하여 자바 설정 클래스에서 스프링 빈으로 생성하므로 @Component 애너테이션 같은 스프링 빈 애너테이션이 코드 생략
      - ◆ MessageListener에서 수신한 Message 객체는 byte[ ] 타입의 로우 데이터를 리턴하므로 자바 객체로 변환하려면 RedisSerializer가 필요하므로 eventRedisTemplate의 valueSerializer를 참조하여 RedisSerializer를 변수에 할당하는데 구독자 코드에서 적절한 RedisTemplate 스프링 빈이 없다면 Jackson2JsonRedisSerializer 객체를 직접 생성해도 상관없음
      - ◆ 레디스에서 구독한 메시지 객체인 Message의 getBody() 메서드는 게시자가 토픽에 게시 한 메시지를 응답하므로 valueSerializer의 deserialize() 메서드를 사용하면 EventMessage 객체로 변환할 수 있음
      - ◆ Message의 getChannel() 메서드를 사용하면 어떤 토픽에서 메시지를 구독했는지 확인할 수 있는데 byte[] 타입을 리턴하므로 문자열로 변환하는 과정이 필요하므로 new String()을 사용

Redis

- ❖ Redis Pub-Sub 구현
  - ✓ 구독자
    - ● 구독자 등록 – RedisConfig 클래스에 추가

```
@Bean
public RedisMessageListenerContainer redisMessageListenerContainer() {
        RedisMessageListenerContainer container = new
    RedisMessageListenerContainer();
        container.setConnectionFactory(eventRedisConnectionFactory());
        container.addMessageListener(eventListener(), eventTopic());
        return container;
}

@Bean
public MessageListener eventListener(){
        return new EventListener(eventRedisTemplate());
}
```

Redis

- ❖ Redis Pub-Sub 구현
  - ✓ 구독자
    - ● 구독자 등록 – config.EventConfig
      - ◆ RedisMessageListenerContainer 객체를 생성
      - ◆ RedisMessageListenerContainer는 RedisTemplate과 마찬가지로 레디스와 연결하려고 RedisConnectionFactory 객체를 사용하는데 setConnectionFactory() 메서드를 사용
      - ◆ RedisMessageListenerContainer에 MessageListener 구현체를 등록하는데 이때 addMessageListener() 메서드는 MessageListener 객체와 ChannelTopic 객체를 인자로 받고 등록된 MessageListener 객체는 ChannelTopic의 토픽에서 받은 메시지를 처리

Redis

- ❖ Redis Pub-Sub 구현
  - ✓ 게시자 와 구독자 테스트
    - 테스트 클래스 생성 – EventTest

```
@SpringBootTest
public class EventTest {

    @Autowired
    private EventPublisher eventPublisher;

    @Test
    public void testPubSub() throws InterruptedException {
        eventPublisher.sendMessage(new EventMessage("test"));
        TimeUnit.SECONDS.sleep(3);
    }
}
```

    - 결과

```
Subscribe Channel : dummyTopic
Subscribe Message : EventMessage(timestamp=1679103268208, message=test)
```