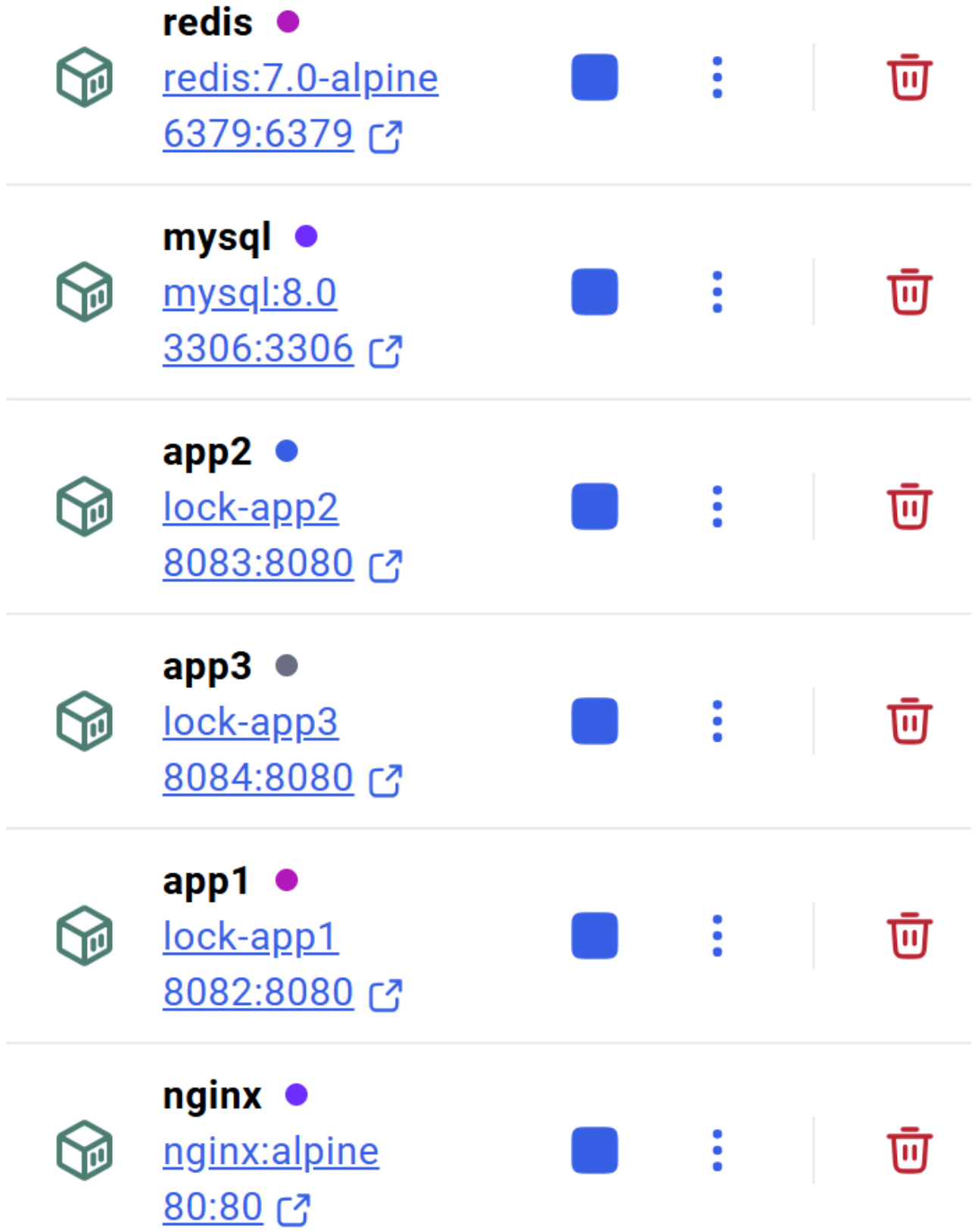
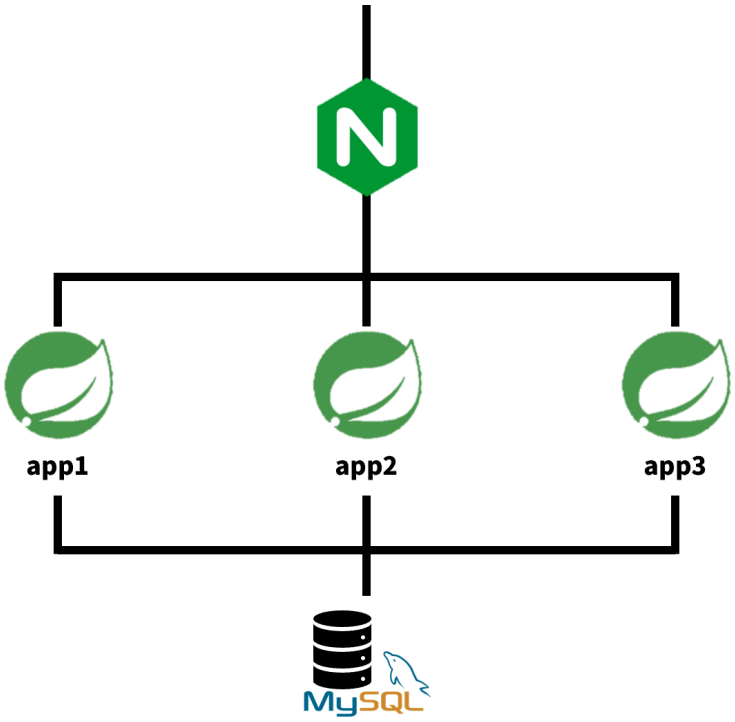
백엔드 아키텍처


redis
redis:7.0-alpine
6379:6379
mysql
mysql:8.0
3306:3306
app2
lock-app2
8083:8080
app3
lock-app3
8084:8080
app1
lock-app1
8082:8080
nginx
nginx:alpine
80:80



- `Nginx`로 들어오는 `/api/*` 요청들을 백엔드 서버들로 분산해서 전송해줌.
- 분산 로직은 `Nginx`에서 정의한 `Round-Robin` 방식을 채택함.

`nginx.conf`

```
events {
    worker_connections 1024;
}

http {
    # 로드밸런싱 대상 서버 정의
    upstream backend {
        # 라운드로빈 방식 (기본값)
        server app1:8080;
        server app2:8080;
        server app3:8080;
    }

    server {
        listen 80;
        server_name localhost;

        # 로그 설정
        access_log /var/log/nginx/access.log;
        error_log /var/log/nginx/error.log;

        # API 요청을 백엔드로 프록시
        location /api/ {
            proxy_pass http://backend;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
```

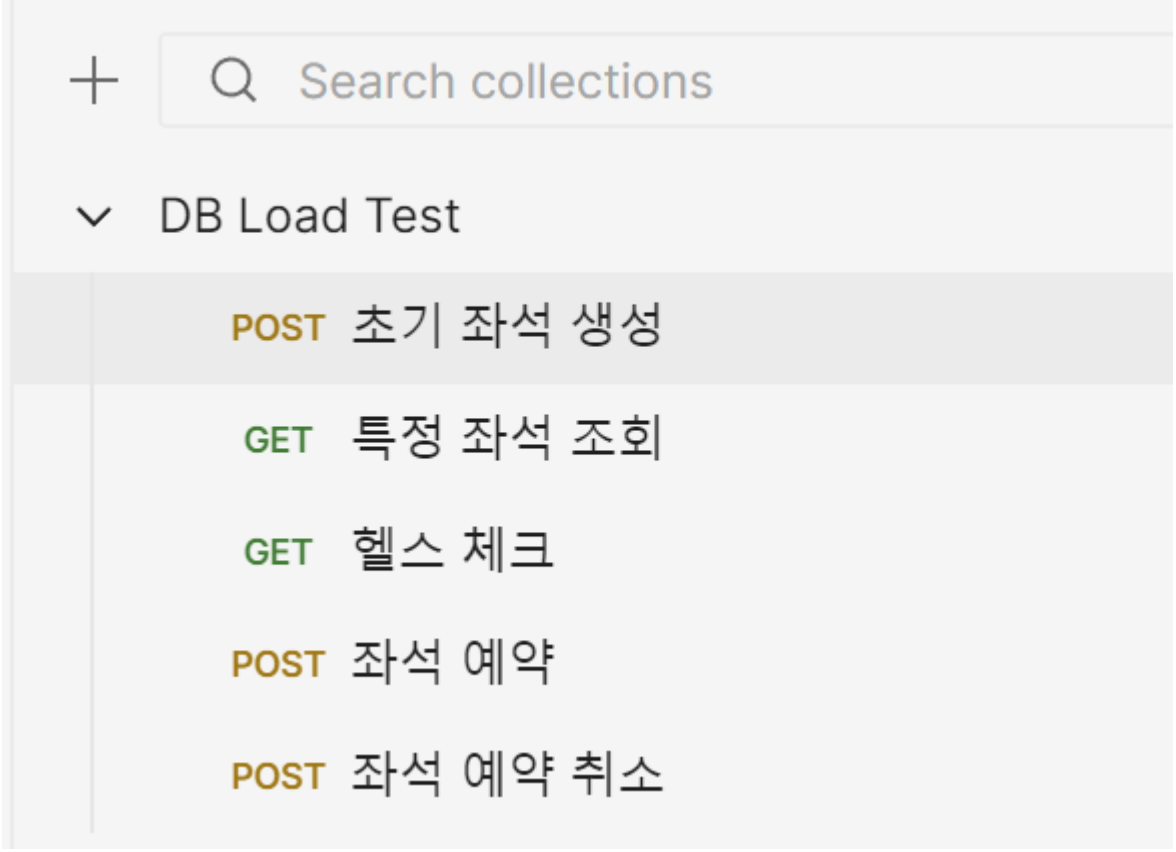
```
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;

            # 타임아웃 설정
            proxy_connect_timeout 60s;
            proxy_send_timeout 60s;
            proxy_read_timeout 60s;
        }

        # 헬스체크
        location /health {
            access_log off;
            return 200 "OK\n";
            add_header Content-Type text/plain;
        }
    }
}
```

백엔드 로직

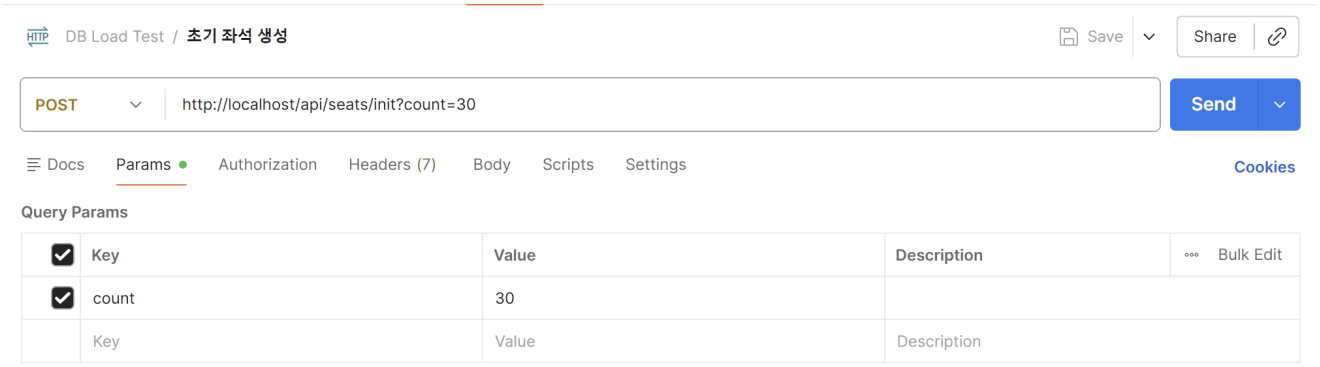
```
package com.dbstudy.lock.controller;

import com.dbstudy.lock.domain.Seat;
import com.dbstudy.lock.dto.ReservationRequest;
import com.dbstudy.lock.dto.ReservationResponse;
import com.dbstudy.lock.service.SeatService;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.Map;

@Slf4j
@RestController
@RequestMapping("/api/seats")
@RequiredArgsConstructor
public class SeatController {

    private final SeatService seatService;

    /**
     * 좌석 예약 API (동시성 제어 없음)
     */    @PostMapping("/reserve")
    public ResponseEntity<ReservationResponse> reserveSeat(@RequestBody
ReservationRequest request) {
        log.info("예약 요청 수신 - seatId: {}, userName: {}",
request.getSeatId(), request.getUserName());
```

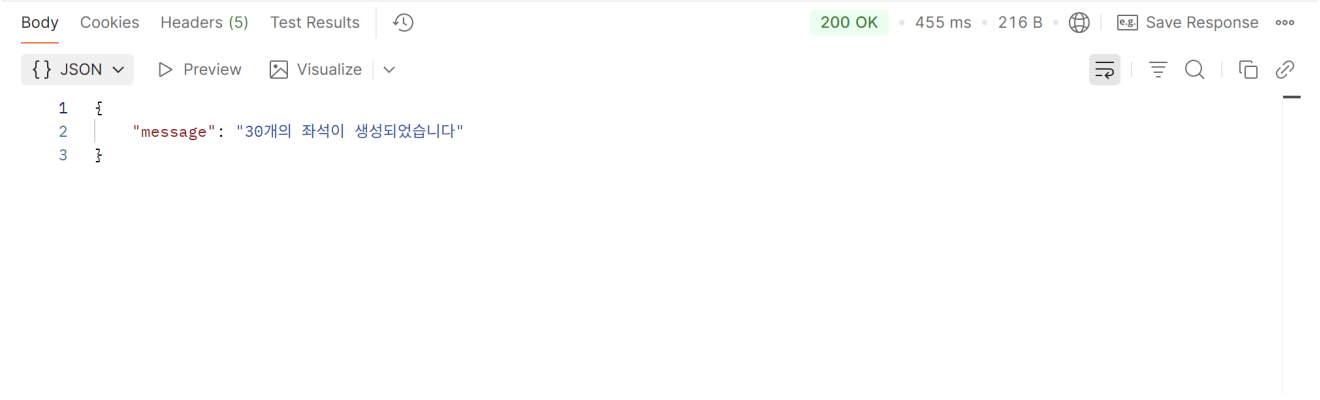
```
        ReservationResponse response = seatService.reserveSeatWithoutLock(
                request.getSeatId(),
                request.getUserName()
        );

        return ResponseEntity.ok(response);
    }

    /**
     * 좌석 조회 API
     */    @GetMapping("/{id}")
    public ResponseEntity<Seat> getSeat(@PathVariable Long id) {
        Seat seat = seatService.getSeat(id);
        return ResponseEntity.ok(seat);
    }

    /**
     * 초기 데이터 생성 API (테스트용)
     */    @PostMapping("/init")
    public ResponseEntity<Map<String, String>>
initSeats(@RequestParam(defaultValue = "10") int count) {
        seatService.initSeats(count);
        return ResponseEntity.ok(Map.of("message", count + "개의 좌석이 생성되
었습니다"));
    }

    /**
     * 예약 취소 API (테스트용)
     */    @PostMapping("/{id}/cancel")
    public ResponseEntity<Map<String, String>>
cancelReservation(@PathVariable Long id) {
        seatService.cancelReservation(id);
        return ResponseEntity.ok(Map.of("message", "예약이 취소되었습니다"));
    }

    /**
     * 헬스체크 API
     */    @GetMapping("/health")
    public ResponseEntity<Map<String, String>> health() {
        return ResponseEntity.ok(Map.of(
            "status", "UP",
            "server", System.getenv().getOrDefault("HOSTNAME", "unknown")
        ));
    }
}
```

API 목록



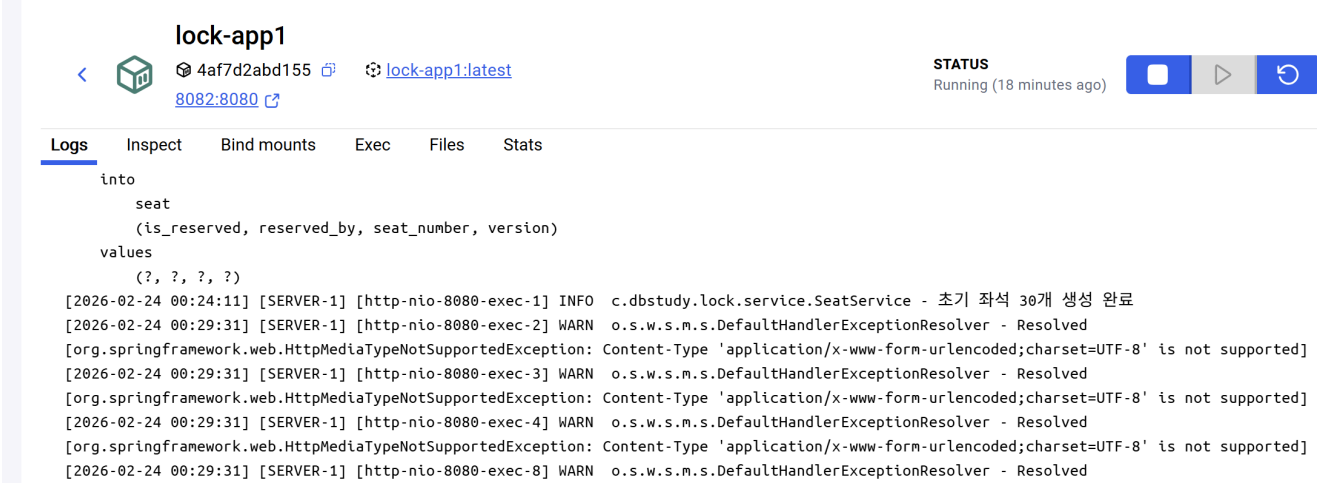
초기 좌석 생성



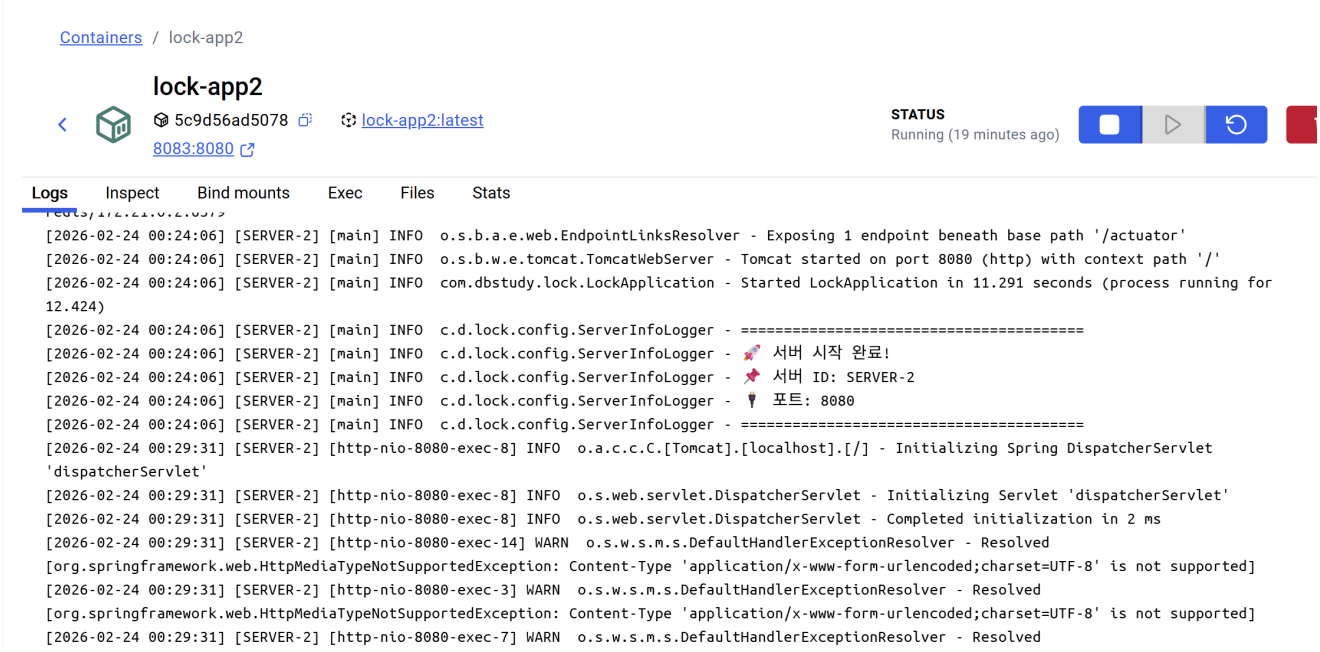


- 라운드 로빈 알고리즘에 의해 `1번 서버` 에만 로그가 찍힘.

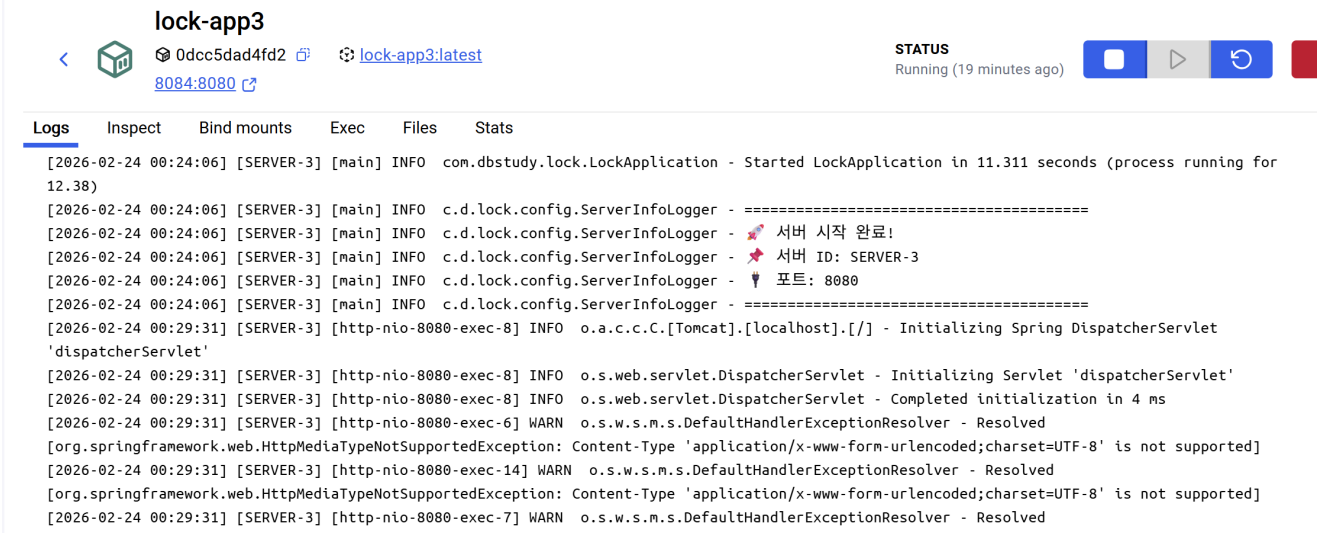
1번 서버



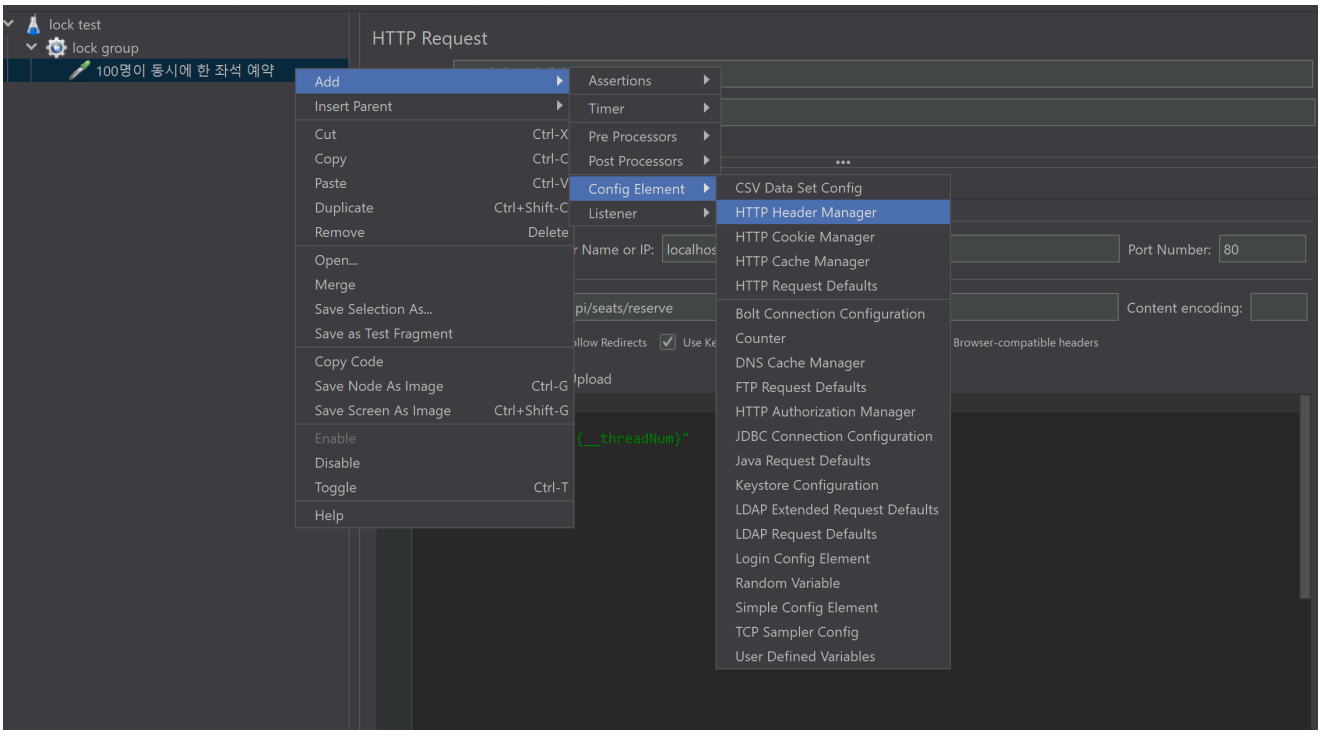
2번 서버

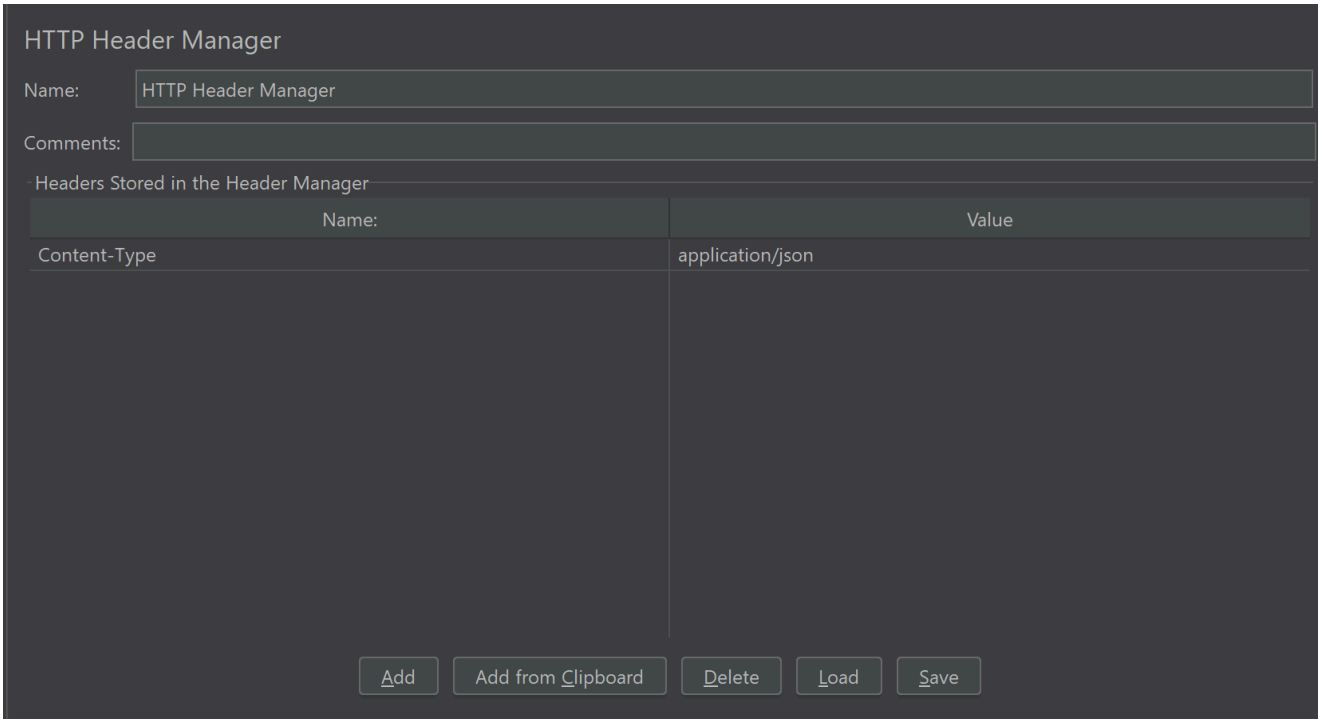


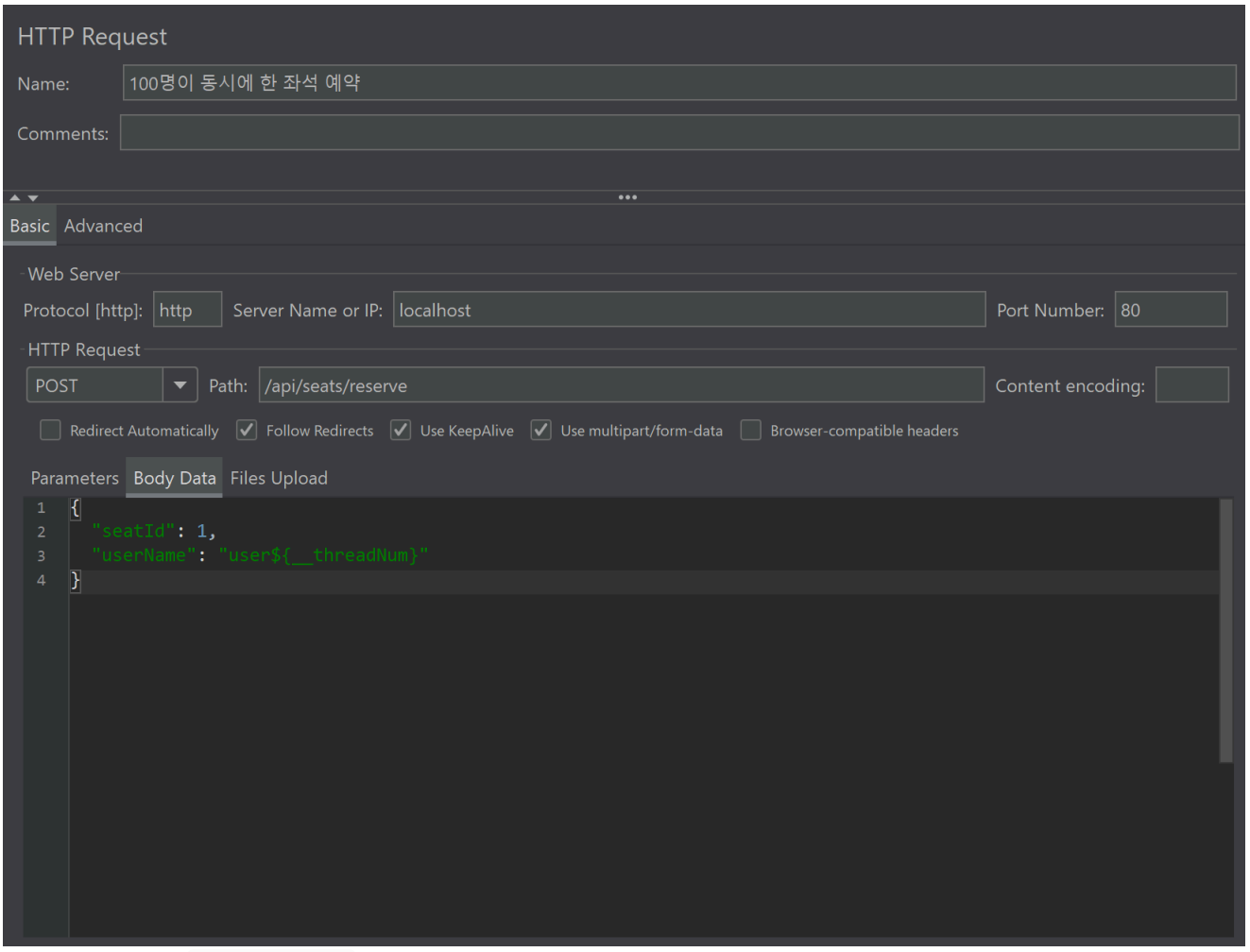
3번 서버



JMeter 세팅

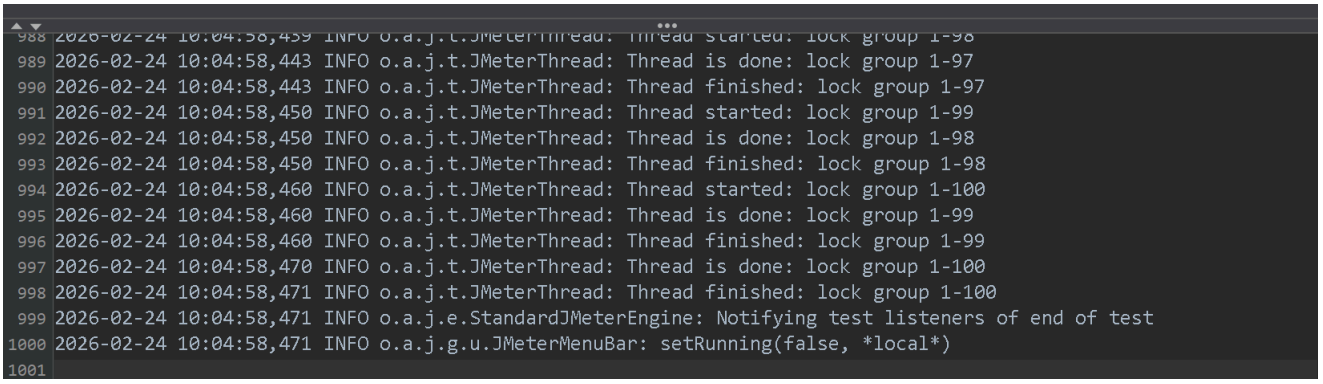

lock test
lock group
100명이 동시에 한 좌석 예약
HTTP Request
Add
Insert Parent
Cut Ctrl-X
Copy Ctrl-C
Paste Ctrl-V
Duplicate Ctrl+Shift-C
Remove Delete
Open...
Merge
Save Selection As...
Save as Test Fragment
Copy Code
Save Node As Image Ctrl-G
Save Screen As Image Ctrl+Shift-G
Enable
Disable
Toggle Ctrl-T
Help
Assertions
Timer
Pre Processors
Post Processors
Config Element
Listener
CSV Data Set Config
HTTP Header Manager
HTTP Cookie Manager
HTTP Cache Manager
HTTP Request Defaults
Bolt Connection Configuration
Counter
DNS Cache Manager
FTP Request Defaults
HTTP Authorization Manager
JDBC Connection Configuration
Java Request Defaults
Keystore Configuration
LDAP Extended Request Defaults
LDAP Request Defaults
Login Config Element
Random Variable
Simple Config Element
TCP Sampler Config
User Defined Variables
Name or IP:
localhos
Port Number:
80
pi/seats/reserve
Content encoding:
Browser-compatible headers
{__threadNum}"


HTTP Header Manager
Name:
HTTP Header Manager
Comments:
Headers Stored in the Header Manager
Name:
Value
Content-Type
application/json
Add
Add from Clipboard
Delete
Load
Save



좌석 번호 1번(`seatId = 1`)을 100명이 동시에 요청하는 상황을 시뮬레이션 하기 위한 세팅

- JMeter 내장 함수를 사용해서 쓰레드 번호를 고유한 유저 번호로 할당했음.



- 요청 성공

로그 분석

```
app1  | 2026-02-24 10:20:32.880 | [2026-02-24 01:20:32] [SERVER-1] [http-nio-8080-exec-1] INFO  c.dbstudy.lock.service.SeatService - 📋 [SERVER-1] 초기 좌석 30개 생성 완료
nginx | 2026-02-24 10:20:32.941 | 172.21.0.1 - - [24/Feb/2026:01:20:32 +0000] "POST /api/seats/init?count=30 HTTP/1.1" 200 65 "-" "PostmanRuntime/7.51.1"
app2  | 2026-02-24 10:20:36.784 | [2026-02-24 01:20:36] [SERVER-2] [http-nio-8080-exec-2] INFO  o.a.c.c.C.[Tomcat].[localhost].[/] - Initializing
```

```
Spring DispatcherServlet 'dispatcherServlet'
app2  | 2026-02-24 10:20:36.784 | [2026-02-24 01:20:36] [SERVER-2] [http-
nio-8080-exec-2] INFO  o.s.web.servlet.DispatcherServlet - Initializing
Servlet 'dispatcherServlet'
app2  | 2026-02-24 10:20:36.786 | [2026-02-24 01:20:36] [SERVER-2] [http-
nio-8080-exec-2] INFO  o.s.web.servlet.DispatcherServlet - Completed
initialization in 2 ms
app1  | 2026-02-24 10:20:36.801 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-3] INFO  c.d.lock.controller.SeatController - 예약 요청 수신 -
seatId: 1, userName: user6
app1  | 2026-02-24 10:20:36.802 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-2] INFO  c.d.lock.controller.SeatController - 예약 요청 수신 -
seatId: 1, userName: user3
app1  | 2026-02-24 10:20:36.803 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-2] INFO  c.dbstudy.lock.service.SeatService - 🔍 [SERVER-1]
[http-nio-8080-exec-2] STEP 1: 좌석 조회 시작 - seatId: 1, userName: user3
app1  | 2026-02-24 10:20:36.803 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-3] INFO  c.dbstudy.lock.service.SeatService - 🔍 [SERVER-1]
[http-nio-8080-exec-3] STEP 1: 좌석 조회 시작 - seatId: 1, userName: user6
```

- 현재, 초기 30개의 좌석 생성 이후 `01:20:36`에 server 3에서 가장 첫 번째 좌석 예약 요청이 들어왔고, 동시에 최초의 `좌석 예약`을 위한 `좌석 조회 요청`이 들어왔음.
  - 이때의 상황은, `SELECT seat WHERE id=1 -> is_reserved=false` 좌석이 예매되어 있지 않다고 판단하고 있는 상황임.

```
app1  | 2026-02-24 10:20:36.811 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-3] DEBUG org.hibernate.SQL -
app1  | 2026-02-24 10:20:36.811 |     select
app1  | 2026-02-24 10:20:36.811 |         s1_0.id,
app1  | 2026-02-24 10:20:36.811 |         s1_0.is_reserved,
app1  | 2026-02-24 10:20:36.811 |         s1_0.reserved_by,
app1  | 2026-02-24 10:20:36.811 |         s1_0.seat_number,
app1  | 2026-02-24 10:20:36.811 |         s1_0.version
app1  | 2026-02-24 10:20:36.811 |     from
app1  | 2026-02-24 10:20:36.811 |         seat s1_0
app1  | 2026-02-24 10:20:36.811 |     where
app1  | 2026-02-24 10:20:36.811 |         s1_0.id=?
app1  | 2026-02-24 10:20:36.811 | Hibernate:
app1  | 2026-02-24 10:20:36.811 |     select
app1  | 2026-02-24 10:20:36.811 |         s1_0.id,
app1  | 2026-02-24 10:20:36.811 |         s1_0.is_reserved,
app1  | 2026-02-24 10:20:36.811 |         s1_0.reserved_by,
app1  | 2026-02-24 10:20:36.811 |         s1_0.seat_number,
app1  | 2026-02-24 10:20:36.811 |         s1_0.version
app1  | 2026-02-24 10:20:36.811 |     from
app1  | 2026-02-24 10:20:36.811 |         seat s1_0
app1  | 2026-02-24 10:20:36.811 |     where
app1  | 2026-02-24 10:20:36.811 |         s1_0.id=?
```

```
app1  | 2026-02-24 10:20:36.816 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-2] INFO  c.dbstudy.lock.service.SeatService - 🚦 [SERVER-1]
[http-nio-8080-exec-2] STEP 2: 예약 가능 체크 - seatId: 1, isReserved: false
- ✅ 통과!
app1  | 2026-02-24 10:20:36.816 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-3] INFO  c.dbstudy.lock.service.SeatService - 🚦 [SERVER-1]
[http-nio-8080-exec-3] STEP 2: 예약 가능 체크 - seatId: 1, isReserved: false
- ✅ 통과!
```

- 현재 `seatId = 1`에 대한 좌석에 두 명의 유저(`user3`, `user6`)가 모두 예약 가능함을 확인 받은 상황.

여기서 킥이 들어가는데,

```
app1  | 2026-02-24 10:20:36.943 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-8] INFO  c.dbstudy.lock.service.SeatService - ⏳ [SERVER-1]
[http-nio-8080-exec-8] STEP 3: 100ms 지연 시작 (동시성 문제 확대 구간)
```

```
// STEP 2: 예약 가능 체크
if (seat.isReserved()) { return "이미 예약됨"; }  // 여기서 "비었다!" 판단

// STEP 3: 100ms 지연
Thread.sleep(100);  // 이 100ms 동안 아무것도 안 함

// STEP 4: 예약 처리
seat.reserve(userName);  // 여기서 실제로 예약 처리
```

- 비었음을 판단한 뒤 의도적으로 100ms 지연을 준 뒤 그 사이에 다른 스레드가 들어올 수 있는 틈을 주는 것.

이때 동시에 두 번째 서버도 요청을 받기 시작함

```
app2  | 2026-02-24 10:20:36.912 | [2026-02-24 01:20:36] [SERVER-2] [http-
nio-8080-exec-5] INFO  c.d.lock.controller.SeatController - 예약 요청 수신 -
seatId: 1, userName: user13
app2  | 2026-02-24 10:20:36.912 | [2026-02-24 01:20:36] [SERVER-2] [http-
nio-8080-exec-3] INFO  c.d.lock.controller.SeatController - 예약 요청 수신 -
seatId: 1, userName: user7
app2  | 2026-02-24 10:20:36.912 | [2026-02-24 01:20:36] [SERVER-2] [http-
nio-8080-exec-1] INFO  c.d.lock.controller.SeatController - 예약 요청 수신 -
seatId: 1, userName: user1
app2  | 2026-02-24 10:20:36.912 | [2026-02-24 01:20:36] [SERVER-2] [http-
nio-8080-exec-6] INFO  c.d.lock.controller.SeatController - 예약 요청 수신 -
seatId: 1, userName: user16
app2  | 2026-02-24 10:20:36.912 | [2026-02-24 01:20:36] [SERVER-2] [http-
nio-8080-exec-2] INFO  c.d.lock.controller.SeatController - 예약 요청 수신 -
seatId: 1, userName: user4
```

```
app2  | 2026-02-24 10:20:36.912 | [2026-02-24 01:20:36] [SERVER-2] [http-
nio-8080-exec-4] INFO  c.d.lock.controller.SeatController - 예약 요청 수신 -
seatId: 1, userName: user10
```

그런 와중에 첫 번째 서버에서는 최초의 예약 성공 로그가 찍히기 시작함.

```
app1  | 2026-02-24 10:20:36.920 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-3] INFO  c.dbstudy.lock.service.SeatService - ✅ [SERVER-1]
[http-nio-8080-exec-3] STEP 4: 예약 성공! seatId: 1 - userName: user6
app1  | 2026-02-24 10:20:36.920 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-2] INFO  c.dbstudy.lock.service.SeatService - ✅ [SERVER-1]
[http-nio-8080-exec-2] STEP 4: 예약 성공! seatId: 1 - userName: user3
app1  | 2026-02-24 10:20:36.923 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-4] INFO  c.dbstudy.lock.service.SeatService - ✅ [SERVER-1]
[http-nio-8080-exec-4] STEP 4: 예약 성공! seatId: 1 - userName: user9
app2  | 2026-02-24 10:20:36.925 | [2026-02-24 01:20:36] [SERVER-2] [http-
nio-8080-exec-6] INFO  c.dbstudy.lock.service.SeatService - 🔍 [SERVER-2]
[http-nio-8080-exec-6] STEP 1: 좌석 조회 시작 - seatId: 1, userName: user16
app2  | 2026-02-24 10:20:36.925 | [2026-02-24 01:20:36] [SERVER-2] [http-
nio-8080-exec-1] INFO  c.dbstudy.lock.service.SeatService - 🔍 [SERVER-2]
[http-nio-8080-exec-1] STEP 1: 좌석 조회 시작 - seatId: 1, userName: user1
app2  | 2026-02-24 10:20:36.925 | [2026-02-24 01:20:36] [SERVER-2] [http-
nio-8080-exec-5] INFO  c.dbstudy.lock.service.SeatService - 🔍 [SERVER-2]
[http-nio-8080-exec-5] STEP 1: 좌석 조회 시작 - seatId: 1,
```

그런데, 이상한 게 어떻게 한 좌석에 대해서 `user6`, `user3`, `user9`가 동시에 예약을 성공했을까?

다시 제대로 흐름을 보자.

```
app1  | 2026-02-24 10:20:36.817 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-4] INFO  c.dbstudy.lock.service.SeatService - 🔍 [SERVER-1]
[http-nio-8080-exec-4] STEP 1: 좌석 조회 시작 - seatId: 1, userName: user9
app1  | 2026-02-24 10:20:36.803 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-2] INFO  c.dbstudy.lock.service.SeatService - 🔍 [SERVER-1]
[http-nio-8080-exec-2] STEP 1: 좌석 조회 시작 - seatId: 1, userName: user3
app1  | 2026-02-24 10:20:36.803 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-3] INFO  c.dbstudy.lock.service.SeatService - 🔍 [SERVER-1]
[http-nio-8080-exec-3] STEP 1: 좌석 조회 시작 - seatId: 1, userName: user6
```

- 좌석을 조회했고 (`스레드 2`, `스레드 3`, `스레드 4`)

```
app1  | 2026-02-24 10:20:36.816 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-2] INFO  c.dbstudy.lock.service.SeatService - 🚦 [SERVER-1]
[http-nio-8080-exec-2] STEP 2: 예약 가능 체크 - seatId: 1, isReserved: false
- ✅ 통과!
app1  | 2026-02-24 10:20:36.816 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-3] INFO  c.dbstudy.lock.service.SeatService - 🚦 [SERVER-1]
```

```
[http-nio-8080-exec-3] STEP 2: 예약 가능 체크 - seatId: 1, isReserved: false
- ✅ 통과!
app1  | 2026-02-24 10:20:36.821 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-4] INFO  c.dbstudy.lock.service.SeatService - 🚦 [SERVER-1]
[http-nio-8080-exec-4] STEP 2: 예약 가능 체크 - seatId: 1, isReserved: false
- ✅ 통과!
```

- 예약 가능함을 확인 받았고 (`스레드 2` , `스레드 3` , `스레드 4`)

```
app1  | 2026-02-24 10:20:36.920 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-3] INFO  c.dbstudy.lock.service.SeatService - ✅ [SERVER-1]
[http-nio-8080-exec-3] STEP 4: 예약 성공! seatId: 1 - userName: user6
app1  | 2026-02-24 10:20:36.920 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-2] INFO  c.dbstudy.lock.service.SeatService - ✅ [SERVER-1]
[http-nio-8080-exec-2] STEP 4: 예약 성공! seatId: 1 - userName: user3
app1  | 2026-02-24 10:20:36.923 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-4] INFO  c.dbstudy.lock.service.SeatService - ✅ [SERVER-1]
[http-nio-8080-exec-4] STEP 4: 예약 성공! seatId: 1 - userName: user9
```

- 예약을 모두 성공했음.

즉, 예약 가능함을 확인 받는 순간이 거의 동시에 이루어졌기 때문에, 각자가 당시에 보던 DB의 상황에서는 예약이 가능했음

예약이 가능해서 또 거의 동시에 예약을 시도했더니 예약이 됐고, 결국에는 db에 user9의 정보가 덮어씌워지게 될 것임.

심지어

```
app1  | 2026-02-24 10:20:36.953 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-5] INFO  c.dbstudy.lock.service.SeatService - ✅ [SERVER-1]
[http-nio-8080-exec-5] STEP 4: 예약 성공! seatId: 1 - userName: user12
```

- `user12` 도 예약을 성공한 걸 볼 수 있는데

```
app1  | 2026-02-24 10:20:36.821 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-4] INFO  c.dbstudy.lock.service.SeatService - ⏳ [SERVER-1]
[http-nio-8080-exec-4] STEP 3: 100ms 지연 시작 (동시성 문제 확대 구간)
app1  | 2026-02-24 10:20:36.847 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-5] INFO  c.d.lock.controller.SeatController - 예약 요청 수신 -
seatId: 1, userName: user12
app1  | 2026-02-24 10:20:36.848 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-5] INFO  c.dbstudy.lock.service.SeatService - 🔍 [SERVER-1]
[http-nio-8080-exec-5] STEP 1: 좌석 조회 시작 - seatId: 1, userName: user12
```

- 해당 유저는 `스레드 4` 가 100ms 지연이 이루어진 뒤에 예약 요청을 수신했음.

- 즉, `스레드 4 = user 9`가 예약을 확정하기 전 100ms의 틈에 `스레드 5 = user 12`가 파고들어서 예약을 시도(`조회 - 가능 체크 - 지연 시작 - 성공`)하려고 하는 상황임.

또한 `스레드 9 = user 25`는 `user 9`가 예약에 성공하고 `user 12`가 예약을 확정 짓기도 전에 접근해서

```
app1  | 2026-02-24 10:20:36.975 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-9] INFO  c.dbstudy.lock.service.SeatService - 🚦 [SERVER-1]
[http-nio-8080-exec-9] STEP 2: 예약 가능 체크 - seatId: 1, isReserved: true -
❌ 차단
app1  | 2026-02-24 10:20:36.976 | [2026-02-24 01:20:36] [SERVER-1] [http-
nio-8080-exec-9] WARN  c.dbstudy.lock.service.SeatService - 🚫 [SERVER-1]
[http-nio-8080-exec-9] 이미 예약된 좌석 - seatId: 1, reservedBy: user9
```

`user 9`가 예약한 좌석이라 예매에 실패했다고 안내받음.

서버 2의 경우에는 서버 1의 요청보다 늦게 도달해서

```
nginx | 2026-02-24 10:20:37.064 | 172.21.0.1 - - [24/Feb/2026:01:20:37
+0000] "POST /api/seats/reserve HTTP/1.1" 500 129 "-" "Apache-
HttpClient/4.5.14 (Java/21.0.9)"
app2  | 2026-02-24 10:20:37.064 | [2026-02-24 01:20:37] [SERVER-2] [http-
nio-8080-exec-6] INFO  c.dbstudy.lock.service.SeatService - 🚦 [SERVER-2]
[http-nio-8080-exec-6] STEP 2: 예약 가능 체크 - seatId: 1, isReserved: true -
❌ 차단
app2  | 2026-02-24 10:20:37.064 | [2026-02-24 01:20:37] [SERVER-2] [http-
nio-8080-exec-8] INFO  c.dbstudy.lock.service.SeatService - 🚦 [SERVER-2]
[http-nio-8080-exec-8] STEP 2: 예약 가능 체크 - seatId: 1, isReserved: true -
❌ 차단
app2  | 2026-02-24 10:20:37.064 | [2026-02-24 01:20:37] [SERVER-2] [http-
nio-8080-exec-1] INFO  c.dbstudy.lock.service.SeatService - 🚦 [SERVER-2]
[http-nio-8080-exec-1] STEP 2: 예약 가능 체크 - seatId: 1, isReserved: true -
❌ 차단
app2  | 2026-02-24 10:20:37.064 | [2026-02-24 01:20:37] [SERVER-2] [http-
nio-8080-exec-7] INFO  c.dbstudy.lock.service.SeatService - 🚦 [SERVER-2]
[http-nio-8080-exec-7] STEP 2: 예약 가능 체크 - seatId: 1, isReserved: true -
❌ 차단
app2  | 2026-02-24 10:20:37.064 | [2026-02-24 01:20:37] [SERVER-2] [http-
nio-8080-exec-6] WARN  c.dbstudy.lock.service.SeatService - 🚫 [SERVER-2]
[http-nio-8080-exec-6] 이미 예약된 좌석 - seatId: 1, reservedBy: user9
app2  | 2026-02-24 10:20:37.064 | [2026-02-24 01:20:37] [SERVER-2] [http-
nio-8080-exec-8] WARN  c.dbstudy.lock.service.SeatService - 🚫 [SERVER-2]
[http-nio-8080-exec-8] 이미 예약된 좌석 - seatId: 1, reservedBy: user9
app2  | 2026-02-24 10:20:37.064 | [2026-02-24 01:20:37] [SERVER-2] [http-
nio-8080-exec-7] WARN  c.dbstudy.lock.service.SeatService - 🚫 [SERVER-2]
[http-nio-8080-exec-7] 이미 예약된 좌석 - seatId: 1, reservedBy: user9
app2  | 2026-02-24 10:20:37.064 | [2026-02-24 01:20:37] [SERVER-2] [http-
```

```
nio-8080-exec-1] WARN  c.dbstudy.lock.service.SeatService - 🚫 [SERVER-2]
[http-nio-8080-exec-1] 이미 예약된 좌석 - seatId: 1, reservedBy: user9
```

`user 9`가 예약한 좌석에 조회 요청을 날려서 차단당한 것도 확인할 수 있음.

타임라인을 정리하면,

`user 9` 100ms 대기 시작 -> `user 12` 예약 시도 -> `user 9` 예약 성공 -> `user 25` 및 `서버 2` 접근 시도했으나, `user 9`가 예약 성공됐기에 차단 -> `user 12` 예약 성공 -> 최종 예약자 `user 12`

100ms 사이의 치열한 타임라인 - by gemini

- **1단계 [~ .821]: `user9`의 조회 및 대기**
  - `user9`가 좌석을 조회합니다. 아무도 없음을 확인하고 예약 처리를 위해 100ms 대기(수면) 상태에 들어갑니다.
- **2단계 [.848]: `user12`의 파고들기**
  - `user9`가 자고 있는 사이 `user12`가 조회를 시도합니다. `user9`가 아직 DB에 도장을 찍지 않았으므로 여전히 '빈자리'로 조회됩니다. `user12` 역시 통과하고 100ms 대기에 들어갑니다.
- **3단계 [.923]: `user9`의 예약 성공**
  - 잠에서 깬 `user9`가 예약을 확정 짓습니다. **(DB 상태: `user9` 예약됨)**
- **4단계 [.925 ~ .976]: `user25` 및 `서버2` 유저들의 접근과 차단**
  - `user25`와 서버 2(app2)로 들어온 유저들이 DB를 조회합니다. 이때는 `user9`가 방금 DB에 도장을 찍은 직후이므로, 이들은 `reservedBy: user9`라는 데이터를 읽게 됩니다.
  - 당연히 "이미 예약된 좌석"이라며 차단당합니다. (이들은 `user9`가 최종 승리자인 줄 압니다.)
- **5단계 [.953]: `user12`의 최종 덮어쓰기 (Lost Update 발생)**
  - 드디어 잠에서 깬 `user12`가 예약을 시도합니다.
  - `user12`는 이미 2단계에서 '빈자리'라는 검증을 통과했기 때문에, 현재 DB 상태가 `user9`로 바뀌었는지 어쨌는지 확인하지 않고 **그냥 묻지도 따지지도 않고 자신의 데이터로 덮어써 버립니다. (DB 상태: `user12` 예약됨)**

결과만 보자면, 100ms 틈 사이에서 `스레드 5`가 예약에 관한 모든 과정을 통과해서 `user 12`가 최종 예약자로 덮어씌워지는 문제가 발생한 것.

최종 정리

> 전형적인 경쟁 상태에 의한 갱신 손실

1. **조회 시점의 착각 (Read-Modify-Write의 한계):** 여러 스레드가 거의 동시에 `SELECT`를 때림.
   - 이때는 아직 어느 스레드도 예약을 확정(DB Update) 짓지 않았기 때문에, `user3`, `user6`, `user9` 모두가 `is_reserved=false`라는 똑같은 과거의 데이터를 읽고 "예약 가능하군!" 하고 판단해 버린 상황.
2. **비동기적인 틈새 (100ms 지연):** 의도적으로 걸어둔 100ms 지연 시간(`Thread.sleep(100)`)

- 앞선 스레드들이 아직 예약을 처리(`STEP 4`)하기도 전에, 뒤늦게 들어온 `user12` 스레드까지 이 틈을 파고들어서 예약 관련 로직들을 진행해버림.

3. **무자비한 덮어쓰기 (Lost Update):** 각 스레드들은 자기가 조회했던 사실만 믿고 동시다발적으로 예약을 진행.
   - 결국 먼저 DB에 기록된 예약 정보들은 0.001초 차이로 뒤따라온 스레드들에 의해 덮어씌워지고, 가장 마지막에 `UPDATE`를 친 `user12`가 최종 예약자가 됨.

이게 바로 동시성 제어 처리가 안 된 시스템에서 다수의 트래픽이 몰렸을 때 발생하는 아주 교과서적인 **정합성 붕괴** 상태라고 볼 수 있음.

`(근데 굳이 서버를 3개나 만든 이유는 최종적으로 적용할 구조인 레디스 분산 락의 도입을 위한 정당성 확보 때문임. 즉, 단일 서버로도 충분히 재현 가능한 상황임.)`

다음에 계속