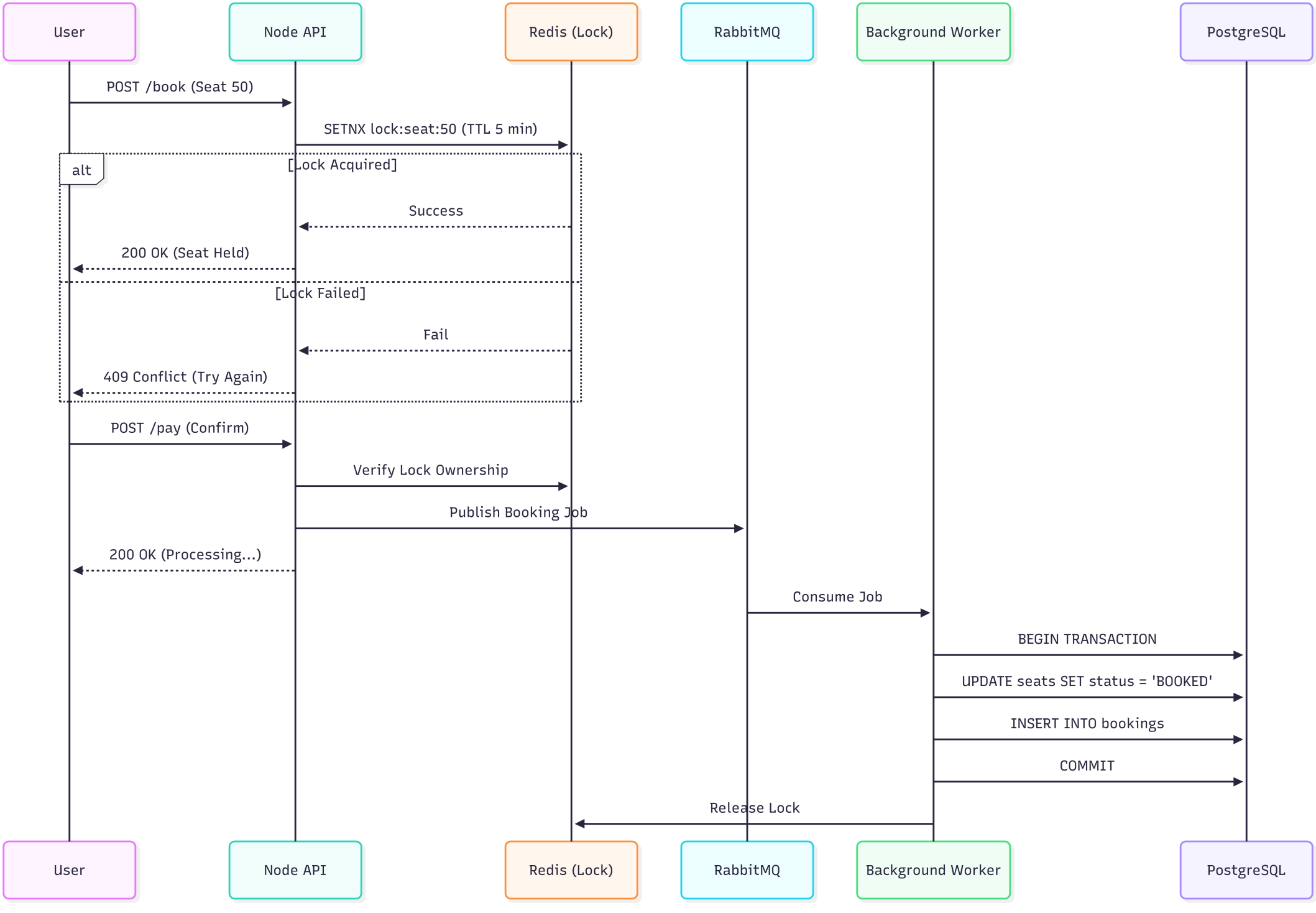
How to Handle 10,000 Users in 1 Second

Solving the “Taylor Swift Problem” with Node.js & Redis

Kavishka Niluminda

The Logic

```
// THE CORE LOGIC: Distributed Locking with Redis
// Goal: Prevent double-booking when 10k users hit the button at once.

export const bookSeat = async (req: Request, res: Response) => {
  const { seatId, userId } = req.body;
  const lockKey = `lock:seat:${seatId}`;

  try {
    // 1. Try to acquire the lock in Redis (Mutex)
    // 'NX': Only set if Not Exists (The Atomic Guard)
    // 'EX': Auto-expire in 300s to prevent deadlocks
    const acquiredLock = await redisClient.set(lockKey, userId, {
      NX: true,
      EX: 300
    });

    // 2. If Redis says "False", someone else beat us by a millisecond
    if (!acquiredLock) {
      return res.status(409).json({
        error: "Seat is currently on hold by another user."
      });
    }

    // 3. Lock Acquired! The user now has exclusive access
    // We can now safely proceed to payment processing...
    return res.status(200).json({
      message: "Seat held! Proceed to payment."
    });

  } catch (error) {
    console.error("Concurrency Error:", error);
    res.status(500).json({ error: "System Busy" });
  }
};
```


User
Node API
Redis (Lock)
RabbitMQ
Background Worker
PostgreSQL
POST /book (Seat 50)
SETNX lock:seat:50 (TTL 5 min)
alt
[Lock Acquired]
Success
200 OK (Seat Held)
[Lock Failed]
Fail
409 Conflict (Try Again)
POST /pay (Confirm)
Verify Lock Ownership
Publish Booking Job
200 OK (Processing...)
Consume Job
BEGIN TRANSACTION
UPDATE seats SET status = 'BOOKED'
INSERT INTO bookings
COMMIT
Release Lock
User
Node API
Redis (Lock)
RabbitMQ
Background Worker
PostgreSQL

K6 Results

```
  ▉ TOTAL RESULTS

    checks_total.......: 1000   97.186916/s
    checks_succeeded...: 50.00% 500 out of 1000
    checks_failed......: 50.00% 500 out of 1000

    ✗ is status 200 (Success)
      ↳  0% — ✓ 1 / ✗ 499
    ✗ is status 409 (Locked)
      ↳  99% — ✓ 499 / ✗ 1
      ↳  99% — ✓ 499 / ✗ 1

    HTTP
    http_req_duration..............: avg=20.76ms min=551.2µs med=13.2ms  max=75.1ms  p(90)=57.86ms
p(95)=63.47ms
      { expected_response:true }...: avg=39.83ms min=39.83ms med=39.83ms max=39.83ms p(90)=39.83ms
p(95)=39.83ms
    http_req_failed................: 99.80% 499 out of 500
    http_reqs......................: 500    48.593458/s

      ↳  99% — ✓ 499 / ✗ 1

    HTTP
    http_req_duration..............: avg=20.76ms min=551.2µs med=13.2ms  max=75.1ms  p(90)=57.86ms
p(95)=63.47ms
      { expected_response:true }...: avg=39.83ms min=39.83ms med=39.83ms max=39.83ms p(90)=39.83ms
p(95)=39.83ms
    http_req_failed................: 99.80% 499 out of 500
    http_reqs......................: 500    48.593458/s

    EXECUTION
    iteration_duration.............: avg=1.02s   min=1s      med=1.01s   max=1.08s   p(90)=1.06s
p(95)=1.07s
    iterations.....................: 500    48.593458/s
    vus............................: 50     min=50         max=50
    vus_max........................: 50     min=50         max=50

    NETWORK
    data_received..................: 160 kB 16 kB/s
    data_sent......................: 79 kB  7.7 kB/s


running (10.3s), 00/50 VUs, 500 complete and 0 interrupted iterations
default ✓ [======================================] 50 VUs  10s
```