Logistic Platform For Goods Management

Technologies Used:

- Node.js
- React.js
- Express.js
- Mysql
- Redis Cache
- Websockets

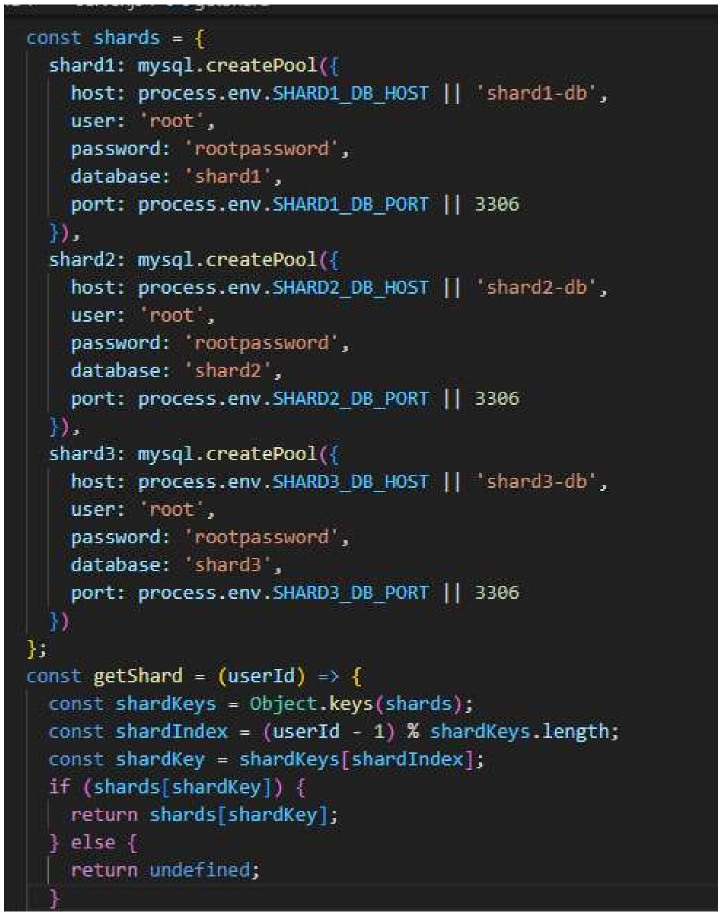
This system is designed to handle high-volume traffic efficiently using:

- *a horizontally sharded MySQL setup*
- *real-time communication through WebSockets*
- *a caching layer implemented using Redis.*

1. Horizontal Sharding and Distributed Data Handling

Horizontal sharding divides the database into smaller, manageable pieces or "shards" to spread the load evenly. This technique allows the system to scale horizontally by adding more shards as needed, ensuring the database can handle a high volume of data and traffic without overloading a single node.

This code defines the shard pools and the function getShard that maps user IDs to specific shards. By distributing users across different shards based on their IDs, the system ensures that no single shard is overloaded, which improves performance and scalability.

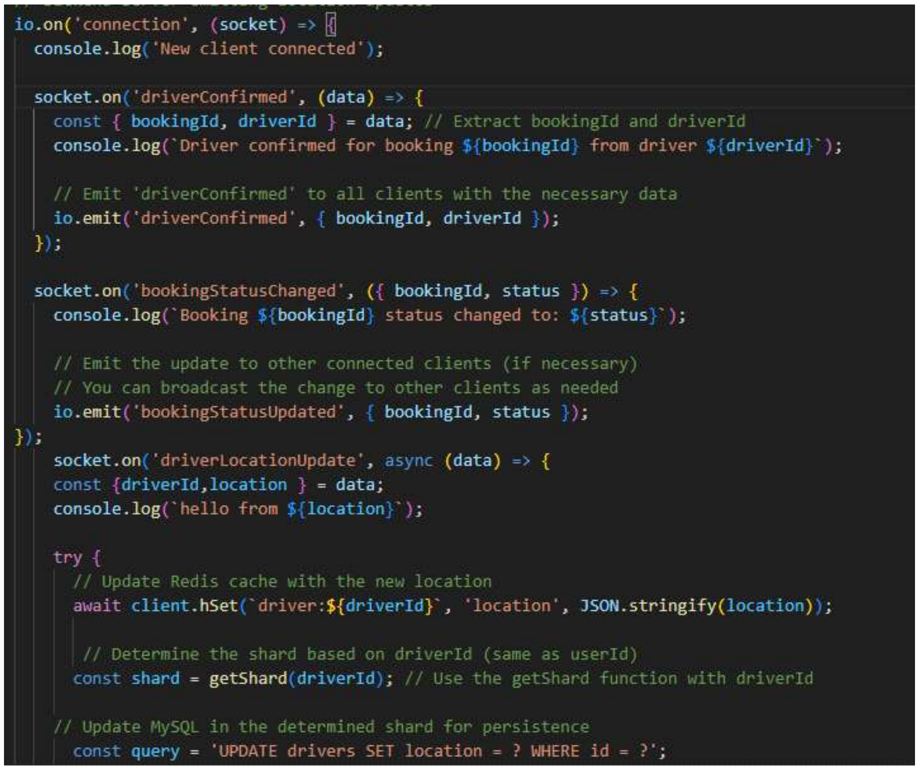


```
const shards = {
  shard1: mysql.createPool({
    host: process.env.SHARD1_DB_HOST || 'shard1-db',
    user: 'root',
    password: 'rootpassword',
    database: 'shard1',
    port: process.env.SHARD1_DB_PORT || 3306
  }),
  shard2: mysql.createPool({
    host: process.env.SHARD2_DB_HOST || 'shard2-db',
    user: 'root',
    password: 'rootpassword',
    database: 'shard2',
    port: process.env.SHARD2_DB_PORT || 3306
  }),
  shard3: mysql.createPool({
    host: process.env.SHARD3_DB_HOST || 'shard3-db',
    user: 'root',
    password: 'rootpassword',
    database: 'shard3',
    port: process.env.SHARD3_DB_PORT || 3306
  })
};
const getShard = (userId) => {
  const shardKeys = Object.keys(shards);
  const shardIndex = (userId - 1) % shardKeys.length;
  const shardKey = shardKeys[shardIndex];
  if (shards[shardKey]) {
    return shards[shardKey];
  } else {
    return undefined;
  }
```

2. Real-Time Data Handling Using WebSockets

WebSockets are used for real-time communication, particularly for tracking and updating driver locations during a ride. This approach allows the server to send and receive location updates without the overhead of repeated HTTP requests, providing efficient and fast updates.

The WebSocket server listens for events like startRide and driverLocationUpdate to update driver locations and broadcast these changes in real-time to all connected clients. This enables accurate tracking of driver movements and status updates for bookings.

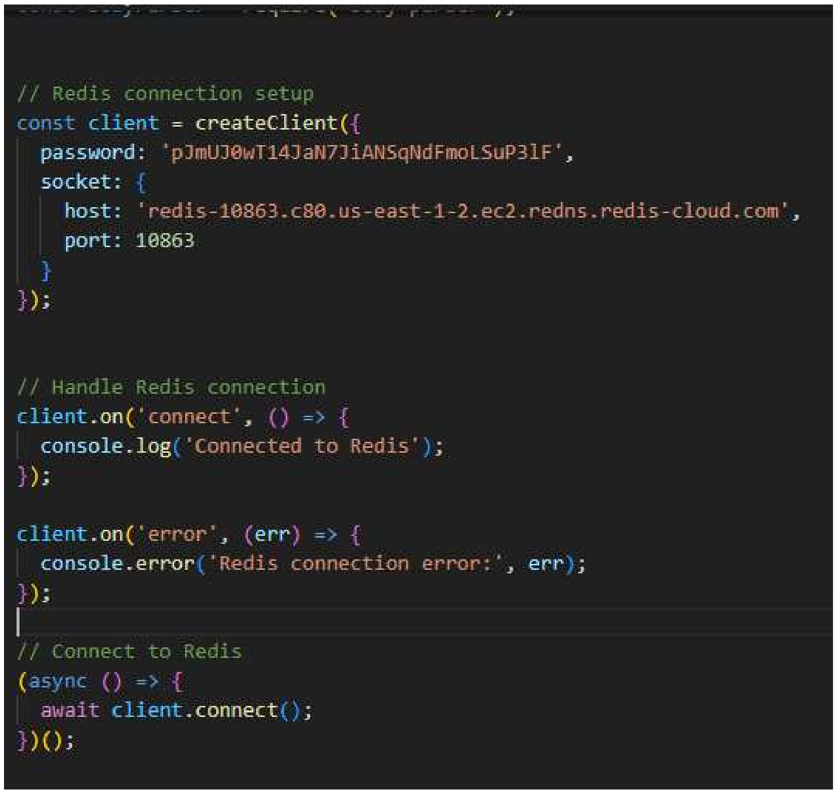


```
io.on('connection', (socket) => {
  console.log('New client connected');

  socket.on('driverConfirmed', (data) => {
    const { bookingId, driverId } = data; // Extract bookingId and driverId
    console.log(`Driver confirmed for booking ${bookingId} from driver ${driverId}`);

    // Emit 'driverConfirmed' to all clients with the necessary data
    io.emit('driverConfirmed', { bookingId, driverId });
  });

  socket.on('bookingStatusChanged', ({ bookingId, status }) => {
    console.log(`Booking ${bookingId} status changed to: ${status}`);

    // Emit the update to other connected clients (if necessary)
    // You can broadcast the change to other clients as needed
    io.emit('bookingStatusUpdated', { bookingId, status });
});
    socket.on('driverLocationUpdate', async (data) => {
    const {driverId,location } = data;
    console.log(`hello from ${location}`);

    try {
      // Update Redis cache with the new location
      await client.hSet(`driver:${driverId}`, 'location', JSON.stringify(location));

       // Determine the shard based on driverId (same as userId)
      const shard = getShard(driverId); // Use the getShard function with driverId

    // Update MySQL in the determined shard for persistence
      const query = 'UPDATE drivers SET location = ? WHERE id = ?';
```

3. Redis Caching for Improved Performance

Redis is used as an in-memory cache layer to reduce database load and improve response times. Frequently accessed data such as user information and booking details are cached in Redis. This caching layer also ensures that location updates are quickly available to clients without repeatedly querying the MySQL database.

Redis significantly reduces the time to access data by caching it closer to the application. This improves the system’s overall performance, especially under high traffic. Additionally, by caching driver locations and booking statuses, the system ensures that frequently accessed information is immediately available.



```
// Redis connection setup
const client = createClient({
  password: 'pJmUJ0wT14JaN7JiANSqNdFmoLSuP3lF',
  socket: {
    host: 'redis-10863.c80.us-east-1-2.ec2.redns.redis-cloud.com',
    port: 10863
  }
});


// Handle Redis connection
client.on('connect', () => {
  console.log('Connected to Redis');
});

client.on('error', (err) => {
  console.error('Redis connection error:', err);
});

// Connect to Redis
(async () => {
  await client.connect();
})();
```

Author:

Arman Singh

armansingh1132@gmail.com

7338947485