Data Migration Guide: PostgreSQL to MySQL

This guide will help you migrate your existing data from PostgreSQL to MySQL for cPanel deployment.

Overview

There are two approaches to migrating your data:

1. **Fresh Start** (Recommended for new deployments)
2. **Data Export/Import** (For preserving existing data)

Option 1: Fresh Start (Recommended)

If you’re okay with starting fresh (no existing user data to preserve), this is the easiest approach.

Steps:

1. **Set up MySQL database** (follow CPANEL_DEPLOYMENT.md)

2. **Push the schema**:
   ```
   bash
      cd /home/ubuntu/tfg_gaming_club/nextjs_space
      yarn prisma db push
   ```

3. **Seed initial data**:
   ```
   bash
      yarn prisma db seed
   ```

This creates:
- Default admin user
- Initial games
- Default settings

You’re done! The application will work with a clean MySQL database.

Option 2: Data Export/Import (Preserve Existing Data)

Use this if you have existing users, bookings, or other data you want to keep.

Prerequisites:

- Access to your PostgreSQL database
- MySQL database set up and ready
- `psql` and `mysql` command-line tools installed

Step 1: Export Data from PostgreSQL

1.1 Export to JSON (Using Prisma Studio)

The easiest method for small to medium datasets:

1. **Start Prisma Studio** with PostgreSQL:
   `bash`
   `   # Ensure DATABASE_URL points to PostgreSQL`
   `   yarn prisma studio`

2. **Export each table**:
   - Open Prisma Studio in browser (http://localhost:5555)
   - For each model (User, Game, Booking, etc.):

     - Select all records
     - Copy data
     - Save to JSON files

1.2 Export Using pg_dump

For larger datasets:

```
# Export entire database as SQL
pg_dump -h db-b83a569b8.db003.hosteddb.reai.io \
  -U role_b83a569b8 \
  -d b83a569b8 \
  -F p \
  -f postgres_backup.sql

# Or export as CSV for each table
psql -h db-b83a569b8.db003.hosteddb.reai.io \
  -U role_b83a569b8 \
  -d b83a569b8 \
  -c "\COPY \"User\" TO 'users.csv' CSV HEADER"

psql -h db-b83a569b8.db003.hosteddb.reai.io \
  -U role_b83a569b8 \
  -d b83a569b8 \
  -c "\COPY \"Game\" TO 'games.csv' CSV HEADER"

# Repeat for other tables: Booking, PaymentLog, PaymentRecord, Settings, Account, Ses-
sion
```

Step 2: Prepare MySQL Database

```
# Connect to your MySQL database
mysql -h localhost -u your_mysql_user -p your_database_name

# Or set DATABASE_URL to MySQL and push schema
cd /home/ubuntu/tfg_gaming_club/nextjs_space
yarn prisma db push
```

This creates all tables in MySQL.

Step 3: Transform and Import Data

Method A: Manual CSV Import

If you exported to CSV:

```
# Import users
mysql -h localhost -u your_mysql_user -p your_database_name \
  -e "LOAD DATA LOCAL INFILE 'users.csv'
      INTO TABLE User
      FIELDS TERMINATED BY ','
      ENCLOSED BY '\"'
      LINES TERMINATED BY '\n'
      IGNORE 1 ROWS;"

# Repeat for other tables
```

Note: You may need to:
- Enable `local_infile` in MySQL
- Adjust field terminators based on your CSV format
- Handle special characters and NULL values

Method B: Using a Migration Script

Create a custom migration script:

```
// scripts/migrate-postgres-to-mysql.ts
import { PrismaClient as PrismaClientPostgres } from '@prisma/client';
import { PrismaClient as PrismaClientMySQL } from '@prisma/client';
import * as dotenv from 'dotenv';

dotenv.config();

// Create two Prisma clients
const postgresClient = new PrismaClientPostgres({
  datasources: {
    db: {
      url: process.env.POSTGRES_DATABASE_URL, // Old PostgreSQL URL
    },
  },
});

const mysqlClient = new PrismaClientMySQL({
  datasources: {
    db: {
      url: process.env.DATABASE_URL, // New MySQL URL
    },
  },
});

async function migrateData() {
  try {
    console.log('Starting migration...');

    // 1. Migrate Users
    console.log('Migrating users...');
    const users = await postgresClient.user.findMany();
    for (const user of users) {
      await mysqlClient.user.create({
        data: {
          id: user.id,
          username: user.username,
          password: user.password,
          realName: user.realName,
          dob: user.dob,
          discordUsername: user.discordUsername,
          membershipType: user.membershipType,
          membershipExpiry: user.membershipExpiry,
          membershipExpiredAt: user.membershipExpiredAt,
          isAdmin: user.isAdmin,
          balanceDue: user.balanceDue,
          freeWeek: user.freeWeek,
          createdAt: user.createdAt,
          updatedAt: user.updatedAt,
        },
      });
    }
    console.log(`Migrated ${users.length} users`);

    // 2. Migrate Games
    console.log('Migrating games...');
    const games = await postgresClient.game.findMany();
    for (const game of games) {
      await mysqlClient.game.create({
        data: {
          id: game.id,
          name: game.name,
          iconUrl: game.iconUrl,
        showOnFrontpage: game.showOnFrontpage,
        createdAt: game.createdAt,
      },
    });
  }
  console.log(`Migrated ${games.length} games`);

  // 3. Migrate Settings
  console.log('Migrating settings...');
  const settings = await postgresClient.settings.findMany();
  for (const setting of settings) {
    await mysqlClient.settings.create({
      data: {
        id: setting.id,
        tableCount: setting.tableCount,
        updatedAt: setting.updatedAt,
      },
    });
  }
  console.log(`Migrated ${settings.length} settings`);

  // 4. Migrate Bookings (without relations first)
  console.log('Migrating bookings...');
  const bookings = await postgresClient.booking.findMany();
  for (const booking of bookings) {
    await mysqlClient.booking.create({
      data: {
        id: booking.id,
        date: booking.date,
        tableNumber: booking.tableNumber,
        gameId: booking.gameId,
        createdById: booking.createdById,
        playersNeeded: booking.playersNeeded,
        notes: booking.notes,
        status: booking.status,
        createdAt: booking.createdAt,
        updatedAt: booking.updatedAt,
      },
    });
  }
  console.log(`Migrated ${bookings.length} bookings`);

  // 5. Migrate Booking Relations (players and paidUsers)
  console.log('Migrating booking relations...');
  const bookingsWithRelations = await postgresClient.booking.findMany({
    include: {
      players: true,
      paidUsers: true,
    },
  });

  for (const booking of bookingsWithRelations) {
    // Connect players
    if (booking.players.length > 0) {
      await mysqlClient.booking.update({
        where: { id: booking.id },
        data: {
          players: {
            connect: booking.players.map(player => ({ id: player.id })),
          },
        },
      });
    }
```

```
      // Connect paidUsers
      if (booking.paidUsers.length > 0) {
        await mysqlClient.booking.update({
          where: { id: booking.id },
          data: {
            paidUsers: {
              connect: booking.paidUsers.map(user => ({ id: user.id })),
            },
          },
        });
      }
    }
    console.log('Migrated booking relations');

    // 6. Migrate Payment Records
    console.log('Migrating payment records...');
    const paymentRecords = await postgresClient.paymentRecord.findMany();
    for (const record of paymentRecords) {
      await mysqlClient.paymentRecord.create({
        data: {
          id: record.id,
          userId: record.userId,
          amount: record.amount,
          date: record.date,
          type: record.type,
          notes: record.notes,
          createdAt: record.createdAt,
        },
      });
    }
    console.log(`Migrated ${paymentRecords.length} payment records`);

    // 7. Migrate Payment Logs
    console.log('Migrating payment logs...');
    const paymentLogs = await postgresClient.paymentLog.findMany();
    for (const log of paymentLogs) {
      await mysqlClient.paymentLog.create({
        data: {
          id: log.id,
          userId: log.userId,
          weekDate: log.weekDate,
          amountDue: log.amountDue,
          isPaid: log.isPaid,
          paymentType: log.paymentType,
          notes: log.notes,
          createdAt: log.createdAt,
        },
      });
    }
    console.log(`Migrated ${paymentLogs.length} payment logs`);

    // 8. Migrate NextAuth Data (Accounts, Sessions)
    console.log('Migrating NextAuth accounts...');
    const accounts = await postgresClient.account.findMany();
    for (const account of accounts) {
      await mysqlClient.account.create({
        data: {
          id: account.id,
          userId: account.userId,
          type: account.type,
          provider: account.provider,
          providerAccountId: account.providerAccountId,
          refresh_token: account.refresh_token,
          access_token: account.access_token,
          expires_at: account.expires_at,
          token_type: account.token_type,
          scope: account.scope,
          id_token: account.id_token,
          session_state: account.session_state,
        },
      });
    }
    console.log(`Migrated ${accounts.length} accounts`);

    console.log('Migrating NextAuth sessions...');
    const sessions = await postgresClient.session.findMany();
    for (const session of sessions) {
      // Skip expired sessions
      if (session.expires < new Date()) {
        continue;
      }
      await mysqlClient.session.create({
        data: {
          id: session.id,
          sessionToken: session.sessionToken,
          userId: session.userId,
          expires: session.expires,
        },
      });
    }
    console.log(`Migrated ${sessions.length} sessions`);

    console.log('\n=== Migration Complete ===");
    console.log('All data has been successfully migrated from PostgreSQL to MySQL');
  } catch (error) {
    console.error('Migration failed:', error);
    throw error;
  } finally {
    await postgresClient.$disconnect();
    await mysqlClient.$disconnect();
  }
}

migrateData()
  .then(() => {
    console.log('Migration script finished');
    process.exit(0);
  })
  .catch((error) => {
    console.error('Migration script failed:', error);
    process.exit(1);
  });
```

To use this script:

1. Save it as `scripts/migrate-postgres-to-mysql.ts`

2. Add both database URLs to your `.env`:
   ```env
   # Old PostgreSQL database
   POSTGRES_DATABASE_URL=”postgresql://...”
   ```

# New MySQL database
DATABASE_URL="mysql://..."
```

1. Run the migration:
   `bash`
   `yarn tsx --require dotenv/config scripts/migrate-postgres-to-mysql.ts`

### Step 4: Verify Migration

1. **Check record counts**:
   ```
   bash
     # In MySQL
     mysql -h localhost -u your_user -p your_database -e "
       SELECT 'Users' as Table, COUNT(*) as Count FROM User
       UNION ALL
       SELECT 'Games', COUNT(*) FROM Game
       UNION ALL
       SELECT 'Bookings', COUNT(*) FROM Booking
       UNION ALL
       SELECT 'PaymentLogs', COUNT(*) FROM PaymentLog;
     "
   ```

2. **Test the application**:
   - Try logging in with an existing user
   - Check if bookings display correctly
   - Verify admin functions work
   - Test creating a new booking

3. **Use Prisma Studio**:
   ```
   bash
     # Make sure DATABASE_URL points to MySQL
     yarn prisma studio
   ```
   Browse your data to ensure everything looks correct

---

## Troubleshooting

### Issue: Foreign Key Errors

**Symptom**: Errors about foreign key constraints during import

**Solution**:
- Temporarily disable foreign key checks in MySQL:
```
sql
  SET FOREIGN_KEY_CHECKS=0;
  -- Run your imports
  SET FOREIGN_KEY_CHECKS=1;
```
- Or use `relationMode = "prisma"` in schema (already configured)

### Issue: Date/Time Format Differences

**Symptom**: Dates don't import correctly
```

Solution:
- Ensure dates are in ISO 8601 format during export
- Use Prisma's built-in date handling in the migration script

Issue: Enum Values Not Recognized

Symptom: Errors with `MembershipType` or `BookingStatus` enums

Solution:
- MySQL enums are case-sensitive
- Ensure enum values match exactly: `WEEKLY`, `MONTHLY`, `YEARLY`
- Check your data for any lowercase values

Issue: Text Fields Truncated

Symptom: Long text fields (notes, tokens) are cut off

Solution:
- MySQL `TEXT` type should handle this
- Check if you need `MEDIUMTEXT` or `LONGTEXT`
- The schema uses `@db.Text` which maps correctly

Issue: Character Encoding Problems

Symptom: Special characters display incorrectly

Solution:

```sql
-- Set UTF-8 encoding for your database
ALTER DATABASE your_database CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;

-- For specific tables if needed
ALTER TABLE User CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Best Practices

1. **Always backup** your PostgreSQL data before migrating
2. **Test the migration** with a small subset of data first
3. **Verify data integrity** after migration
4. **Keep PostgreSQL running** until you're confident the MySQL migration is successful
5. **Document any custom transformations** you make during migration
6. **Test all application features** after switching to MySQL

Rollback Plan

If you need to rollback to PostgreSQL:

1. Change `prisma/schema.prisma` provider back to `"postgresql"`
2. Update `DATABASE_URL` to point to PostgreSQL
3. Run `yarn prisma generate`

4. Restart your application

Your PostgreSQL data should still be intact if you didn't delete it.

Additional Resources

- Prisma Data Migration Guide (https://www.prisma.io/docs/guides/migrate-to-prisma/migrate-from-typeorm)
- MySQL LOAD DATA Documentation (https://dev.mysql.com/doc/refman/8.0/en/load-data.html)
- PostgreSQL pg_dump Documentation (https://www.postgresql.org/docs/current/app-pgdump.html)

Good luck with your migration! If you encounter issues not covered here, consult the Prisma documentation or reach out for help.