Database Migration MongoDB to PostgreSQL

1. Objective

The aim to migrate the backend database of the MOP project from MongoDB (NoSQL) to PostgreSQL (Relational) to:

a. Improve data consistency, query performance, and relational modelling
b. Enable advanced analytics, data validation, and long-term scalability
c. Facilitate integration with modern developer tools and ORMs

2. Why?

a. No enforced schema validation beyond Mongoose model definitions
b. Complex aggregation pipelines for statistics queries
c. Limited relational modelling, causing redundant or embedded data
d. Inconsistent development experience when querying deeply nested or grouped data

Benefit	Description
Strict schema	Data types, lengths, and constraints are enforced by default
Faster queries	Especially for grouping (e.g., tag + semester stats)
Easier analytics	Native SQL GROUP BY and JOIN operations simplify reporting logic
ORM compatibility	Prisma integration enables typed, safe, and simple access across the app
Cleaner data model	No need for nested arrays — normalized design makes associations clear

3. Full Migration Workflow

Step 1: Backup and Export MongoDB

a. Export current MongoDB data to JSON:

```
mongoexport --db=mop --collection=items --out=items.json
mongoexport --db=mop --collection=usecases --out=usecases.json
```

Alternatively, use a custom Node.js or Python script to export with transformations.

Step 2: Define PostgreSQL Schema via Prisma

a. Install Prisma:

```
npm install @prisma/client
npx prisma init
```

b. Update .env:

```
DATABASE_URL="postgresql://postgres:pass@localhost:5432/mop"
```

c. **In prisma/schema.prisma:**

```
model Item {
  id          Int     @id @default(autoincrement())
  name        String  @db.VarChar(20)
  description String
  price       Float
}

model UseCase {
  id         Int     @id @default(autoincrement())
  tag        String  @db.VarChar(30)
  popularity Int
  semester   String  @db.VarChar(10)
}
```

d. **Apply schema:**

```
npx prisma migrate dev --name init
npx prisma generate
```

Step 3: Migrate Data from MongoDB to PostgreSQL

Write a script in Python or JS to:

a. Read usecases.json and items.json
b. Transform them into flat rows
c. Insert them into the new PostgreSQL tables

Step 4: Replace Mongoose Models with Prisma

a. **Remove:**

   i. models/Item.js, models/UseCase.js
   ii. lib/mongodb.js
   iii. mongoose from package.json

b. **Add:**

Create lib/prisma.js:

```
import { PrismaClient } from '@prisma/client';
const globalForPrisma = global;
const prisma = globalForPrisma.prisma || new PrismaClient();
if (process.env.NODE_ENV !== 'production') globalForPrisma.prisma = prisma;
export default prisma;
```

Step 5: Refactor API Routes

a. **/api/items.js:**

```
import prisma from '../../lib/prisma';

export default async function handler(req, res) {
  const { method } = req;

  switch (method) {
    case 'GET':
      const items = await prisma.item.findMany();
      return res.status(200).json({ success: true, data: items });
    case 'POST':
      const item = await prisma.item.create({ data: req.body });
      return res.status(201).json({ success: true, data: item });
    default:
      res.setHeader('Allow', ['GET', 'POST']);
      return res.status(405).end(`Method ${method} Not Allowed`);
  }
}
```

b. **/api/usecases.js:**

```
import prisma from '../../lib/prisma';

export default async function handler(req, res) {
  const { method, query } = req;

  if (method === 'GET') {
    if (query.stats === 'true') {
      const stats = await prisma.useCase.groupBy({
        by: ['tag', 'semester'],
        _avg: { popularity: true },
        _count: true
      });

      const formatted = stats.map(stat => ({
        tag: stat.tag,
        semester: stat.semester,
        averagePopularity: Number(stat._avg.popularity.toFixed(2)),
        count: stat._count
      }));

      return res.status(200).json({ success: true, data: formatted });
    }

    const useCases = await prisma.useCase.findMany();
    return res.status(200).json({ success: true, data: useCases });
  }

  if (method === 'POST') {
    const useCase = await prisma.useCase.create({ data: req.body });
    return res.status(201).json({ success: true, data: useCase });
  }

  res.setHeader('Allow', ['GET', 'POST']);
  return res.status(405).end(`Method ${method} Not Allowed`);
}
```

Step 6: Rebuild Dummy Data Seeder

To translate current dummy data logic into a seed file.

a. **Create prisma/seed.js:**

```
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();

async function main() {
  await prisma.useCase.deleteMany();
  await prisma.useCase.createMany({
    data: [
      { tag: 'Frontend', semester: '2024-T1', popularity: 75 },
      { tag: 'Backend', semester: '2024-T2', popularity: 60 },
      { tag: 'Frontend', semester: '2024-T3', popularity: 80 },
      { tag: 'Frontend', semester: '2024-T1', popularity: 55 },
      { tag: 'Backend', semester: '2024-T3', popularity: 70 },
    ]
  });

  console.log('Dummy data inserted.');
}

main().finally(() => prisma.$disconnect());
```

b. Run:

```
node prisma/seed.js
```

<u>Step 7: Deploy and Test</u>

1. Update .env.local for production PostgreSQL
2. Deploy backend using Vercel, GCP, or Azure
3. Test /api/items, /api/usecases, and /api/usecases?stats=true using Postman or browser
4. Confirm all outputs match original MongoDB behavior

4. Conclusion:

PostgreSQL migration with Prisma brings a more robust, scalable, and query-optimized backend to the MOP project. This eliminates the need for manual aggregation logic, reduces code complexity, and opens the door for better analytics and developer experience.