Mastering Sequelize with PostgreSQL

A Comprehensive Guide for Beginners

Achmad Zaenuri

November 25, 2024

Contents

Chapter 1

Setting Up the Environment Properly

1.1 Introduction

This chapter will guide you through the process of setting up your development environment for working with Sequelize and PostgreSQL. As a beginner, you'll learn everything from installing the necessary software to creating your first database connection.

1.2 Prerequisites

Before we begin, ensure you have the following installed on your system:

- Node.js (version 14.x or higher)
- npm (usually comes with Node.js)
- PostgreSQL (version 12.x or higher)

1.2.1 Installing Node.js

Follow these steps to install Node.js:

1. Visit the official Node.js website: https://nodejs.org
2. Download the LTS (Long Term Support) version
3. Follow the installation wizard for your operating system
4. Verify installation by opening a terminal and running:

```
$ node --version
$ npm --version
```

1.2.2 Installing PostgreSQL

Follow these steps to install PostgreSQL:

1. Visit: https://www.postgresql.org/download/
2. Choose your operating system and follow the installation instructions
3. Note down the superuser (postgres) password during installation
4. Verify installation by running:

```
$ psql --version
```

1.3 Project Setup

1.3.1 Creating a New Node.js Project

```
$ mkdir my-sequelize-project
$ cd my-sequelize-project
$ npm init -y
```

1.3.2 Installing Required Dependencies

Installing Packages

```
$ npm install sequelize sequelize-cli pg pg-hstore
$ npm install nodemon --save-dev
```

Note

The `pg` package is the PostgreSQL driver, and `pg-hstore` is required for storing and retrieving JSON data.

1.4 Database Configuration

1.4.1 Creating a Database

PostgreSQL Commands

```
-- Connect to PostgreSQL
$ psql -U postgres

-- Create a new database
postgres=# CREATE DATABASE my_sequelize_db;

-- Verify database creation
postgres=# \l
```

1.4.2 Project Structure

Create the following directory structure:

```
my-sequelize-project/
    config/
        database.js
    models/
        index.js
    migrations/
    seeders/
    package.json
```

1.4.3 Database Configuration File

Create `config/database.js`:

```
module.exports = {
        development: {
                username: 'postgres',
                password: 'your_password',
                database: 'my_sequelize_db',
                host: '127.0.0.1',
                dialect: 'postgres',
                logging: console.log,
                pool: {
                        max: 5,
                        min: 0,
                        acquire: 30000,
                        idle: 10000
                }
        }
};
```

1.5 Database Connection

1.5.1 Setting Up the Connection

Create `models/index.js`:

```
const { Sequelize } = require('sequelize');
const config = require('../config/database.js');

const env = process.env.NODE_ENV || 'development';
const dbConfig = config[env];

const sequelize = new Sequelize(
dbConfig.database,
dbConfig.username,
dbConfig.password,
{
        host: dbConfig.host,
        dialect: dbConfig.dialect,
        logging: dbConfig.logging,
        pool: dbConfig.pool
}
);

module.exports = sequelize;
```

1.5.2 Testing the Connection

Create a test file (`test.js`):

```
const sequelize = require('./models');

async function testConnection() {
        try {
                await sequelize.authenticate();
                console.log('Connection successful!');
        } catch (error) {
                console.error('Connection failed:', error);
        } finally {
                await sequelize.close();
        }
}

testConnection();
```

1.6 Environment Variables

1.6.1 Setting Up Environment Variables

First, install the dotenv package:

```
$ npm install dotenv
```

Create a `.env` file:

```
DB_USERNAME=postgres
DB_PASSWORD=your_password
DB_DATABASE=my_sequelize_db
DB_HOST=127.0.0.1
DB_DIALECT=postgres
NODE_ENV=development
```

1.7 Common Issues and Solutions

Connection Refused
- Check if PostgreSQL service is running
- Verify port availability (default: 5432)

Authentication Failed
- Verify username and password
- Check database permissions

Database Not Found
- Ensure database was created
- Check database name spelling

1.8 Best Practices

1.8.1 Security

- Never commit .env files to version control
- Use environment variables for sensitive data
- Implement proper error handling
- Use connection pooling for better performance

1.8.2 Configuration

- Set up different configurations per environment
- Use appropriate logging levels
- Implement connection timeouts
- Set reasonable pool sizes

1.9 Summary

In this chapter, we covered:

- Environment setup
- Database configuration
- Connection management
- Environment variables
- Best practices
- Common issues and solutions

Chapter 2

Model Design Best Practices

2.1 Introduction

In this chapter, we'll explore how to design and implement database models effectively using Sequelize. You'll learn about data types, relationships, validations, and best practices for structuring your models.

2.2 Understanding Models in Sequelize

2.2.1 What is a Model?

A model in Sequelize is a representation of a database table. It defines the structure of your data and how it should be stored and retrieved.

Basic Model Structure

```
const { Model, DataTypes } = require('sequelize');
const sequelize = require('../config/database');

class User extends Model {}

User.init({
        // Model attributes here
}, {
        sequelize,
        modelName: 'User'
});
```

2.3 Data Types and Attributes

2.3.1 Common Data Types

Sequelize provides various data types that map to PostgreSQL types:

```
const { DataTypes } = require('sequelize');

const User = sequelize.define('User', {
        // String types
        username: {
                type: DataTypes.STRING(100),    // VARCHAR(100)
                allowNull: false
        },
        description: {
                type: DataTypes.TEXT            // TEXT
        },

        // Numeric types
        age: {
                type: DataTypes.INTEGER         // INTEGER
        },
        balance: {
                type: DataTypes.DECIMAL(10, 2)  // DECIMAL(10,2)
        },

        // Date types
        birthDate: {
                type: DataTypes.DATE            // TIMESTAMP WITH TIME ZONE
        },

        // Boolean type
        isActive: {
                type: DataTypes.BOOLEAN         // BOOLEAN
        },

        // JSON type
        settings: {
                type: DataTypes.JSONB           // JSONB
        },

        // UUID type
        id: {
                type: DataTypes.UUID,
                defaultValue: DataTypes.UUIDV4,
                primaryKey: true
        }
});
```

2.3.2 Column Options

Important column options to consider:

```
const Product = sequelize.define('Product', {
        name: {
                type: DataTypes.STRING,
                allowNull: false,               // NOT NULL constraint
                unique: true,                   // UNIQUE constraint
                defaultValue: 'New Product',    // DEFAULT value
                validate: {
                        notEmpty: true,             // Custom validation
                        len: [3, 50]                // Length validation
                }
        },
        price: {
                type: DataTypes.DECIMAL(10, 2),
                validate: {
                        min: 0                      // Minimum value validation
                }
        }
});
```

2.4 Model Relationships

2.4.1 Types of Relationships

Sequelize supports various types of relationships:

```
// One-to-One relationship
User.hasOne(Profile);
Profile.belongsTo(User);

// One-to-Many relationship
User.hasMany(Post);
Post.belongsTo(User);

// Many-to-Many relationship
User.belongsToMany(Role, { through: 'UserRoles' });
Role.belongsToMany(User, { through: 'UserRoles' });
```

2.4.2 Relationship Options

Advanced relationship configurations:

```
// One-to-Many with cascading delete
User.hasMany(Post, {
        foreignKey: {
                name: 'userId',
                allowNull: false
        },
        onDelete: 'CASCADE',
        onUpdate: 'CASCADE'
});
```

```
// Many-to-Many with extra attributes
const UserRoles = sequelize.define('UserRoles', {
        grantedAt: DataTypes.DATE,
        grantedBy: DataTypes.STRING
});

User.belongsToMany(Role, {
        through: UserRoles,
        foreignKey: 'userId'
});

Role.belongsToMany(User, {
        through: UserRoles,
        foreignKey: 'roleId'
});
```

2.5 Model Validations

2.5.1 Built-in Validators

Sequelize provides several built-in validators:

```
const User = sequelize.define('User', {
        email: {
                type: DataTypes.STRING,
                validate: {
                        isEmail: true,          // Email format
                        notNull: true,          // NOT NULL
                        notEmpty: true          // Not empty string
                }
        },
        age: {
                type: DataTypes.INTEGER,
                validate: {
                        min: 0,                 // Minimum value
                        max: 120,               // Maximum value
                        isInt: true             // Must be integer
                }
        },
        website: {
                type: DataTypes.STRING,
                validate: {
                        isUrl: true             // Must be URL
                }
        }
});
```

2.5.2 Custom Validators

Creating custom validation rules:

```
const User = sequelize.define('User', {
        password: {
                type: DataTypes.STRING,
                validate: {
                        isStrongPassword(value) {
                                if (!/[A-Z]/.test(value)) {
                                        throw new Error('Password must contain uppercase letter');
                                }
                                if (!/[0-9]/.test(value)) {
                                        throw new Error('Password must contain number');
                                }
                                if (value.length < 8) {
                                        throw new Error('Password must be at least 8 characters');
                                }
                        }
                }
        }
});
```

2.6 Model Hooks

2.6.1 Lifecycle Hooks

Hooks allow you to trigger actions before or after specific events:

```
const User = sequelize.define('User', {
        username: DataTypes.STRING,
        password: DataTypes.STRING
}, {
        hooks: {
                beforeCreate: async (user) => {
                        // Hash password before saving
                        user.password = await bcrypt.hash(user.password, 10);
                },
                beforeUpdate: async (user) => {
                        // Hash password if it's changed
                        if (user.changed('password')) {
                                user.password = await bcrypt.hash(user.password, 10);
                        }
                }
        }
});
```

2.7 Best Practices

2.7.1 Naming Conventions

- Use PascalCase for model names (e.g., `User`, `BlogPost`)
- Use camelCase for attribute names (e.g., `firstName`, `createdAt`)

- Use underscores for table names (e.g., `blog_posts`, `user_roles`)

2.7.2 Model Organization

Keep your models organized:

```
// models/index.js
const User = require('./user.model');
const Post = require('./post.model');
const Comment = require('./comment.model');

// Set up associations
User.hasMany(Post);
Post.belongsTo(User);
Post.hasMany(Comment);
Comment.belongsTo(Post);

module.exports = {
        User,
        Post,
        Comment
};
```

2.8 Performance Considerations

2.8.1 Indexing

Add indexes for frequently queried fields:

```
const User = sequelize.define('User', {
        email: {
                type: DataTypes.STRING,
                unique: true,
                index: true
        }
}, {
        indexes: [
        {
                fields: ['createdAt'],
                name: 'user_created_at_idx'
        },
        {
                fields: ['email', 'status'],
                name: 'user_email_status_idx'
        }
        ]
});
```

2.9 Practical Examples

2.9.1 E-Commerce System

Let's build a comprehensive e-commerce system model structure:

```
// models/user.model.js
const User = sequelize.define('User', {
        id: {
                type: DataTypes.UUID,
                defaultValue: DataTypes.UUIDV4,
                primaryKey: true
        },
        email: {
                type: DataTypes.STRING,
                allowNull: false,
                unique: true,
                validate: {
                        isEmail: true
                }
        },
        password: {
                type: DataTypes.STRING,
                allowNull: false
        },
        role: {
                type: DataTypes.ENUM('customer', 'admin', 'vendor'),
                defaultValue: 'customer'
        },
        lastLoginAt: {
                type: DataTypes.DATE
        },
        status: {
                type: DataTypes.ENUM('active', 'inactive', 'suspended'),
                defaultValue: 'active'
        }
}, {
        hooks: {
                beforeCreate: async (user) => {
                        user.password = await bcrypt.hash(user.password, 10);
                }
        },
        indexes: [
        { fields: ['email'] },
        { fields: ['status', 'role'] }
        ]
});

// models/product.model.js
const Product = sequelize.define('Product', {
        id: {
                type: DataTypes.UUID,
                defaultValue: DataTypes.UUIDV4,
                primaryKey: true
        },
        name: {
```

```
                type: DataTypes.STRING,
                allowNull: false,
                validate: {
                        len: [3, 100]
                }
        },
        slug: {
                type: DataTypes.STRING,
                unique: true
        },
        description: {
                type: DataTypes.TEXT
        },
        price: {
                type: DataTypes.DECIMAL(10, 2),
                allowNull: false,
                validate: {
                        min: 0
                }
        },
        stock: {
                type: DataTypes.INTEGER,
                allowNull: false,
                defaultValue: 0,
                validate: {
                        min: 0
                }
        },
        metadata: {
                type: DataTypes.JSONB,
                defaultValue: {}
        }
}, {
        hooks: {
                beforeValidate: (product) => {
                        if (product.name && !product.slug) {
                                product.slug = product.name
                                .toLowerCase()
                                .replace(/[^a-zA-Z0-9]/g, '-')
                                .replace(/-+/g, '-');
                        }
                }
        }
});

// models/order.model.js
const Order = sequelize.define('Order', {
        id: {
                type: DataTypes.UUID,
                defaultValue: DataTypes.UUIDV4,
                primaryKey: true
        },
        status: {
                type: DataTypes.ENUM(
                'pending',
                'processing',
                'shipped',
                'delivered',
```

```
                'cancelled'
                ),
                defaultValue: 'pending'
        },
        totalAmount: {
                type: DataTypes.DECIMAL(10, 2),
                allowNull: false
        },
        shippingAddress: {
                type: DataTypes.JSONB,
                allowNull: false
        },
        paymentStatus: {
                type: DataTypes.ENUM('pending', 'paid', 'failed', 'refunded'),
                defaultValue: 'pending'
        }
});

// models/orderItem.model.js
const OrderItem = sequelize.define('OrderItem', {
        quantity: {
                type: DataTypes.INTEGER,
                allowNull: false,
                validate: {
                        min: 1
                }
        },
        priceAtTime: {
                type: DataTypes.DECIMAL(10, 2),
                allowNull: false
        }
});

// Set up relationships
User.hasMany(Order);
Order.belongsTo(User);

Order.hasMany(OrderItem);
OrderItem.belongsTo(Order);

Product.hasMany(OrderItem);
OrderItem.belongsTo(Product);
```

2.9.2 Blog System with Categories and Tags

Example of a blog system with categories and tags:

```
// models/category.model.js
const Category = sequelize.define('Category', {
        id: {
                type: DataTypes.UUID,
                defaultValue: DataTypes.UUIDV4,
                primaryKey: true
        },
        name: {
```

```
                type: DataTypes.STRING,
                allowNull: false,
                unique: true
        },
        slug: {
                type: DataTypes.STRING,
                unique: true
        },
        description: DataTypes.TEXT
}, {
        hooks: {
                beforeValidate: (category) => {
                        if (category.name && !category.slug) {
                                category.slug = category.name
                                .toLowerCase()
                                .replace(/[^a-zA-Z0-9]/g, '-');
                        }
                }
        }
});

// models/post.model.js
const Post = sequelize.define('Post', {
        id: {
                type: DataTypes.UUID,
                defaultValue: DataTypes.UUIDV4,
                primaryKey: true
        },
        title: {
                type: DataTypes.STRING,
                allowNull: false,
                validate: {
                        len: [3, 150]
                }
        },
        slug: {
                type: DataTypes.STRING,
                unique: true
        },
        content: {
                type: DataTypes.TEXT,
                allowNull: false
        },
        status: {
                type: DataTypes.ENUM('draft', 'published', 'archived'),
                defaultValue: 'draft'
        },
        publishedAt: DataTypes.DATE,
        metadata: {
                type: DataTypes.JSONB,
                defaultValue: {
                        views: 0,
                        likes: 0,
                        readingTime: null
                }
        }
}, {
        hooks: {
```

```
                beforeValidate: (post) => {
                        if (post.title && !post.slug) {
                                post.slug = post.title
                                .toLowerCase()
                                .replace(/[^a-zA-Z0-9]/g, '-');
                        }
                },
                beforeCreate: (post) => {
                        if (post.status === 'published' && !post.publishedAt) {
                                post.publishedAt = new Date();
                        }
                }
        },
        indexes: [
        { fields: ['status', 'publishedAt'] }
        ]
});

// models/tag.model.js
const Tag = sequelize.define('Tag', {
        name: {
                type: DataTypes.STRING,
                allowNull: false,
                unique: true
        },
        slug: {
                type: DataTypes.STRING,
                unique: true
        }
});

// models/postTag.model.js
const PostTag = sequelize.define('PostTag', {
        id: {
                type: DataTypes.UUID,
                defaultValue: DataTypes.UUIDV4,
                primaryKey: true
        }
});

// Set up relationships
Category.hasMany(Post);
Post.belongsTo(Category);

Post.belongsToMany(Tag, { through: PostTag });
Tag.belongsToMany(Post, { through: PostTag });

User.hasMany(Post);
Post.belongsTo(User, {
        as: 'author',
        foreignKey: {
                name: 'authorId',
                allowNull: false
        }
});
```

2.9.3 Social Network Connections

Example of handling social network connections:

```
// models/friendship.model.js
const Friendship = sequelize.define('Friendship', {
        status: {
                type: DataTypes.ENUM('pending', 'accepted', 'blocked'),
                defaultValue: 'pending'
        },
        blockedBy: {
                type: DataTypes.UUID,
                allowNull: true
        }
}, {
        indexes: [
        {
                fields: ['status']
        }
        ]
});

// models/user.model.js
User.belongsToMany(User, {
        as: 'friends',
        through: Friendship,
        foreignKey: 'userId',
        otherKey: 'friendId'
});

// Helper methods for the User model
User.prototype.sendFriendRequest = async function(friendId) {
        return await Friendship.create({
                userId: this.id,
                friendId: friendId,
                status: 'pending'
        });
};

User.prototype.acceptFriendRequest = async function(friendId) {
        const friendship = await Friendship.findOne({
                where: {
                        userId: friendId,
                        friendId: this.id,
                        status: 'pending'
                }
        });

        if (!friendship) {
                throw new Error('Friend request not found');
        }

        await friendship.update({ status: 'accepted' });

        // Create reverse friendship
        await Friendship.create({
                userId: this.id,
```

```
                friendId: friendship.userId,
                status: 'accepted'
        });

        return friendship;
};

User.prototype.blockUser = async function(userId) {
        const [friendship] = await Friendship.findOrCreate({
                where: {
                        userId: this.id,
                        friendId: userId
                },
                defaults: {
                        status: 'blocked',
                        blockedBy: this.id
                }
        });

        if (friendship.status !== 'blocked') {
                await friendship.update({
                        status: 'blocked',
                        blockedBy: this.id
                });
        }

        // Remove reverse friendship if exists
        await Friendship.destroy({
                where: {
                        userId: userId,
                        friendId: this.id
                }
        });

        return friendship;
};
```

2.9.4 File Management System

Example of a file management system with folders:

```
// models/folder.model.js
const Folder = sequelize.define('Folder', {
        id: {
                type: DataTypes.UUID,
                defaultValue: DataTypes.UUIDV4,
                primaryKey: true
        },
        name: {
                type: DataTypes.STRING,
                allowNull: false,
                validate: {
                        len: [1, 255]
                }
        },
```

```
        path: {
                type: DataTypes.STRING,
                allowNull: false
        },
        parentId: {
                type: DataTypes.UUID,
                allowNull: true
        }
}, {
        hooks: {
                beforeValidate: async (folder) => {
                        if (folder.parentId) {
                                const parent = await Folder.findByPk(folder.parentId);
                                folder.path = `${parent.path}/${folder.name}`;
                        } else {
                                folder.path = `/${folder.name}`;
                        }
                }
        },
        indexes: [
        { fields: ['path'], unique: true },
        { fields: ['parentId'] }
        ]
});

// models/file.model.js
const File = sequelize.define('File', {
        id: {
                type: DataTypes.UUID,
                defaultValue: DataTypes.UUIDV4,
                primaryKey: true
        },
        name: {
                type: DataTypes.STRING,
                allowNull: false,
                validate: {
                        len: [1, 255]
                }
        },
        mimeType: {
                type: DataTypes.STRING,
                allowNull: false
        },
        size: {
                type: DataTypes.BIGINT,
                allowNull: false,
                validate: {
                        min: 0
                }
        },
        path: {
                type: DataTypes.STRING,
                allowNull: false
        },
        metadata: {
                type: DataTypes.JSONB,
                defaultValue: {}
        }
```

```
}, {
        hooks: {
                beforeValidate: async (file) => {
                        if (file.folderId) {
                                const folder = await Folder.findByPk(file.folderId);
                                file.path = `${folder.path}/${file.name}`;
                        } else {
                                file.path = `/${file.name}`;
                        }
                }
        },
        indexes: [
        { fields: ['path'], unique: true },
        { fields: ['mimeType'] }
        ]
});

// Set up relationships
Folder.hasMany(Folder, {
        as: 'subfolders',
        foreignKey: 'parentId'
});
Folder.belongsTo(Folder, {
        as: 'parent',
        foreignKey: 'parentId'
});

Folder.hasMany(File);
File.belongsTo(Folder);

User.hasMany(Folder);
Folder.belongsTo(User, {
        as: 'owner'
});

User.hasMany(File);
File.belongsTo(User, {
        as: 'owner'
});
```

These examples demonstrate:

- Complex model relationships
- Advanced validation rules
- Custom hooks for data processing
- Proper indexing strategies
- Real-world business logic implementation
- Proper use of JSONB for flexible data
- Comprehensive error handling

2.10 Common Pitfalls

- Circular dependencies in relationships
- Missing indexes on foreign keys
- Improper data type selection
- Inefficient relationship definitions

2.11 Summary

In this chapter, we covered:

- Model definition and structure
- Data types and attributes
- Relationships and associations
- Validations and hooks
- Best practices and performance considerations

Chapter 3

Query Optimization Techniques

3.1 Introduction

Understanding how to write efficient queries is crucial for building performant applications. In this chapter, we'll explore various techniques for querying your database using Sequelize, starting from basic operations and progressing to more advanced concepts. We'll focus not just on how to write queries, but also on understanding why certain approaches are more efficient than others.

3.2 Basic Query Operations

Before diving into optimization, it's essential to understand the fundamental query operations in Sequelize. These operations form the building blocks of all database interactions in your application.

3.2.1 Finding Records

When working with databases, retrieving data is one of the most common operations. Sequelize provides several methods to find records, each suited for different scenarios.

Finding a Single Record

The most basic operation is finding a single record. Sequelize provides multiple ways to achieve this:

Finding by Primary Key

The `findByPk` method is the most straightforward way to retrieve a record when you know its primary key:

```
const user = await User.findByPk(userId);
```

Under the hood, this generates a SQL query like:

```
SELECT * FROM "users" WHERE id = userId LIMIT 1;
```

Use this method when:

- You know the exact primary key
- You need to retrieve a specific record
- You're sure the record exists (returns null if not found)

Finding One Record with Conditions

When you need to find a record based on specific conditions, use `findOne`:

```
const admin = await User.findOne({
        where: {
                role: 'admin',
                status: 'active'
        }
});
```

This generates:

```
SELECT * FROM "users"
WHERE role = 'admin' AND status = 'active'
LIMIT 1;
```

Use this method when:

- You need to find a record based on conditions
- You want only the first matching record
- The record might not exist

Finding Multiple Records

Often, you'll need to retrieve multiple records that match certain criteria. The `findAll` method is your primary tool for this purpose.

Basic FindAll Usage

```
const activeUsers = await User.findAll({
        where: {
                status: 'active'
        }
});
```

This translates to:

```
SELECT * FROM "users" WHERE status = 'active';
```

Important considerations:

- Without limits, this will retrieve ALL matching records
- For large datasets, always implement pagination
- Consider selecting only needed fields

3.2.2 Understanding Operators

Sequelize provides a powerful set of operators that allow you to create complex queries. These operators are similar to SQL operators but with a more JavaScript-friendly syntax.

Basic Operators

Let's explore the most commonly used operators:

Sequelize provides various comparison operators through the `Op` object:

```
const { Op } = require('sequelize');

// Greater than
const expensiveProducts = await Product.findAll({
        where: {
                price: {
                        [Op.gt]: 100    // price > 100
                }
        }
});

// Between range
const midRangeProducts = await Product.findAll({
        where: {
                price: {
                        [Op.between]: [20, 100]    // 20 <= price <= 100
                }
        }
});
```

```
// In a list
const specificProducts = await Product.findAll({
        where: {
                category: {
                        [Op.in]: ['electronics', 'books', 'games']
                }
        }
});
```

Common operators include:

Op.eq : Equal to (=)

Op.ne : Not equal to (!=)

Op.gt : Greater than (>)

Op.gte : Greater than or equal to (>=)

Op.lt : Less than (<)

Op.lte : Less than or equal to (<=)

Op.between : Between two values

Op.in : In a list of values

Op.notIn : Not in a list of values

Logical Operators

Combining multiple conditions is essential for complex queries:

```
const results = await Product.findAll({
        where: {
                [Op.and]: [
                {
                        price: {
                                [Op.gte]: 50
                        }
                },
                {
                        category: 'electronics'
                },
                {
                        [Op.or]: [
                        { brand: 'Samsung' },
                        { brand: 'Apple' }
                        ]
                }
                ]
        }
});
```

This creates a SQL query equivalent to:

```sql
SELECT * FROM products
WHERE price >= 50
AND category = 'electronics'
AND (brand = 'Samsung' OR brand = 'Apple');
```

Understanding the structure:

Op.and : All conditions must be true

Op.or : At least one condition must be true

- Conditions can be nested for complex logic
- Use parentheses in the code to maintain clarity

Pattern Matching Operators

When searching text fields, pattern matching is essential:

```
const searchResults = await Product.findAll({
        where: {
                name: {
                        // Case-insensitive search
                        [Op.iLike]: '%phone%'    // PostgreSQL specific
                },
                description: {
                        // Regular expression match
                        [Op.regexp]: '\\d{3}-\\d{2}-\\d{4}'
                }
        }
});
```

Key pattern matching operators:

Op.like : Case-sensitive pattern match

Op.iLike : Case-insensitive pattern match (PostgreSQL)

Op.regexp : Regular expression match

Op.startsWith : Starts with pattern

Op.endsWith : Ends with pattern

Op.substring : Contains substring

Important Note

When using pattern matching:

- Consider performance implications on large datasets
- Use indexes appropriately for text search fields
- Consider using full-text search for better performance
- Be cautious with leading wildcards (’

3.3 Advanced Querying Techniques

Understanding advanced querying techniques is crucial for building efficient applications. These techniques help you retrieve complex data structures while minimizing database load.

3.3.1 Eager Loading

One of the most important concepts in Sequelize is eager loading, which helps solve the N+1 query problem. Let’s understand what this means and how to use it effectively.

Understanding the N+1 Problem

The N+1 Query Problem

Consider this common scenario:

```
//  Bad Practice: N+1 Problem
const posts = await Post.findAll();
for (const post of posts) {
        const author = await post.getAuthor();
        console.log(author.name);
}
```

This code generates N+1 queries:

- 1 query to fetch all posts
- N queries (one for each post) to fetch its author

If you have 100 posts, this generates 101 database queries!

Basic Eager Loading

Here's how to solve the N+1 problem using eager loading:

```
// ✅ Good Practice: Single Query with Eager Loading
const posts = await Post.findAll({
        include: [{
                model: User,
                as: 'author',
                attributes: ['id', 'name', 'email']
        }]
});

// Now you can access author directly
posts.forEach(post => {
        console.log(post.author.name);
});
```

This generates a single SQL query with a JOIN:

```
SELECT
"Post".*,
"author"."id" AS "author.id",
"author"."name" AS "author.name",
"author"."email" AS "author.email"
FROM "posts" AS "Post"
LEFT OUTER JOIN "users" AS "author"
ON "Post"."authorId" = "author"."id";
```

Key benefits:

- Reduced number of queries
- Better performance
- Less network overhead
- Simpler code

Nested Eager Loading

Sometimes you need to load multiple levels of related data:

```
const orders = await Order.findAll({
        include: [
        {
                model: User,
                attributes: ['id', 'email'],
                include: [{
                        model: Address,
                        attributes: ['street', 'city']
                }]
        },
```

```
        {
                model: OrderItem,
                include: [{
                        model: Product,
                        attributes: ['id', 'name', 'price']
                }]
        }
        ]
});
```

Important considerations:

- Deep nesting can lead to complex queries
- Consider performance impact with large datasets
- May need to split into multiple queries for better performance
- Use `separate: true` for heavy nested relations

3.3.2 Attributes and Aggregations

Understanding how to select specific attributes and perform calculations is crucial for optimizing query performance.

Selecting Specific Attributes

Optimizing Field Selection

Instead of selecting all fields, choose only what you need:

```
// Bad Practice: Selecting everything
const users = await User.findAll();

// Good Practice: Select specific fields
const users = await User.findAll({
        attributes: [
        'id',
        'email',
        ['firstName', 'name'], // Alias
        [sequelize.fn('DATE', sequelize.col('createdAt')), 'joinDate']
        ]
});
```

Benefits of selective attributes:

- Reduced data transfer
- Less memory usage
- Improved query performance
- Clearer data structure

Working with Aggregations

Sequelize provides powerful tools for data aggregation:

Common Aggregation Functions

```
// Basic aggregation
const orderStats = await Order.findAll({
        attributes: [
        [sequelize.fn('DATE', sequelize.col('createdAt')), 'date'],
        [sequelize.fn('COUNT', sequelize.col('id')), 'orderCount'],
        [sequelize.fn('SUM', sequelize.col('totalAmount')), 'revenue']
        ],
        group: [sequelize.fn('DATE', sequelize.col('createdAt'))]
});

// Complex aggregations with conditions
const productStats = await OrderItem.findAll({
        attributes: [
        'productId',
        [sequelize.fn('COUNT', sequelize.col('id')), 'totalOrders'],
        [sequelize.fn('SUM', sequelize.col('quantity')), 'totalQuantity'],
        [
        sequelize.fn(
        'SUM',
        sequelize.literal('quantity * price')
        ),
        'totalRevenue'
        ]
        ],
        include: [{
                model: Product,
                attributes: ['name'],
                required: true
        }],
        group: ['productId', 'Product.id', 'Product.name']
});
```

Common aggregation functions:

- COUNT: Count records
- SUM: Sum values
- AVG: Calculate average
- MAX: Find maximum value
- MIN: Find minimum value

3.4 Performance Optimization

Optimizing query performance is crucial for maintaining a responsive application. Let's explore various techniques to improve query efficiency.

3.4.1 Query Optimization Strategies

Using Indexes Effectively

Index Optimization

Indexes are crucial for query performance:

```
// Define indexes in your model
const User = sequelize.define('User', {
        email: {
                type: DataTypes.STRING,
                unique: true,  // Creates unique index
                index: true    // Creates basic index
        }
}, {
        indexes: [
        {
                name: 'user_status_role',
                fields: ['status', 'role']
        },
        {
                name: 'user_email_status',
                unique: true,
                fields: ['email', 'status']
        }
        ]
});
```

When to use indexes:

- Frequently queried fields
- Fields used in WHERE clauses
- Fields used for sorting (ORDER BY)
- Foreign key fields
- Unique constraint fields

Remember:

- Indexes improve read performance but slow down writes
- Don't over-index - each index takes space and maintenance
- Consider compound indexes for common query patterns
- Monitor index usage

Implementing Pagination

Efficient Pagination

Proper pagination is essential for handling large datasets:

```
const PAGE_SIZE = 20;
const page = req.query.page || 1;

// Basic offset pagination
const products = await Product.findAndCountAll({
        limit: PAGE_SIZE,
        offset: (page - 1) * PAGE_SIZE,
        order: [['createdAt', 'DESC']],
        attributes: ['id', 'name', 'price'] // Select only needed fields
});

// Cursor-based pagination (more efficient for large datasets)
const products = await Product.findAll({
        where: {
                createdAt: {
                        [Op.lt]: cursor // timestamp of last item
                }
        },
        limit: PAGE_SIZE,
        order: [['createdAt', 'DESC']]
});
```

Pagination best practices:

- Use cursor-based pagination for large datasets
- Always include a reasonable page size limit
- Consider caching frequently accessed pages
- Include total count only when necessary

3.5 Transaction Management

Transactions ensure data consistency by grouping multiple database operations into a single unit of work.

3.5.1 Understanding Transactions

Transaction Basics

Transactions follow the ACID principles:

- Atomicity: All operations succeed or all fail
- Consistency: Data remains valid
- Isolation: Transactions don't interfere
- Durability: Changes are permanent

3.5.2 Implementing Transactions

Basic Transaction Usage

Here's a typical transaction implementation:

```
// Using async/await with transaction
const createOrder = async (userId, items) => {
        // Start transaction
        const t = await sequelize.transaction();

        try {
                // Create order
                const order = await Order.create({
                        userId,
                        status: 'pending'
                }, { transaction: t });

                // Create order items
                await Promise.all(items.map(item =>
                OrderItem.create({
                        orderId: order.id,
                        productId: item.productId,
                        quantity: item.quantity
                }, { transaction: t })
                ));

                // Update product stock
                await Promise.all(items.map(item =>
                Product.decrement('stock', {
                        by: item.quantity,
                        where: { id: item.productId },
                        transaction: t
                })
                ));

                // Commit transaction
                await t.commit();
                return order;
        } catch (error) {
                // Rollback on error
                await t.rollback();
                throw error;
        }
};
```

Key points:

- Always use try-catch blocks
- Remember to commit or rollback
- Pass transaction object to all operations
- Consider isolation levels

3.6 Best Practices

3.6.1 Query Optimization Checklist

- Select only needed fields
- Use appropriate indexes
- Implement proper pagination
- Use eager loading to avoid N+1 queries
- Consider using raw queries for complex operations
- Monitor query performance
- Use transactions for data consistency

3.6.2 Performance Monitoring

Monitoring Query Performance

```
// Log slow queries
const sequelize = new Sequelize(config, {
        benchmark: true,
        logging: (sql, timing) => {
                if (timing > 1000) { // Queries taking > 1 second
                        console.warn(`Slow query (${timing}ms):`, sql);
                }
        }
});

// Analyze query plans
const analyzePlan = async () => {
        const [plan] = await sequelize.query(`
        EXPLAIN ANALYZE
        SELECT * FROM users
        JOIN orders ON users.id = orders.user_id
        WHERE users.status = 'active'
        `);
        console.log(plan);
};
```

3.7 Summary

In this chapter, we covered:

- Understanding and using basic queries
- Advanced querying techniques

- Performance optimization strategies
- Transaction management
- Best practices and monitoring

Chapter 4

Advanced Query Optimization Techniques

4.1 Introduction

This chapter delves deeper into advanced query optimization techniques in Sequelize. We'll explore complex queries, performance tuning, and scalability considerations that become crucial as your application grows.

4.2 Complex Query Patterns

4.2.1 Subqueries

Subqueries allow you to nest one query within another. Understanding how to use them effectively is crucial for complex data operations.

Using Subqueries in WHERE Clauses

Subquery Examples

Here's how to use subqueries effectively:

```
// Find users who have placed orders over $1000
const users = await User.findAll({
        where: {
                id: {
                        [Op.in]: sequelize.literal(`(
                        SELECT DISTINCT "userId"
                        FROM "Orders"
                        WHERE "totalAmount" > 1000
                        )`)
                }
        }
});

// More readable alternative using include
const users = await User.findAll({
        include: [{
                model: Order,
                where: {
                        totalAmount: {
                                [Op.gt]: 1000
                        }
                },
                required: true,
                attributes: []  // Don't select Order fields
        }]
});
```

Key considerations:

- Use `sequelize.literal` carefully (risk of SQL injection)
- Consider using includes when possible
- Understand performance implications
- Use indexes on joined fields

Correlated Subqueries

Correlated Subqueries

Subqueries that reference the outer query:

```
// Find products with above-average price in their category
const products = await Product.findAll({
        where: {
                price: {
                        [Op.gt]: sequelize.literal(`(
                        SELECT AVG(price)
                        FROM "Products" AS p2
                        WHERE p2."categoryId" = "Product"."categoryId"
                        )`)
                }
        },
        include: [{
                model: Category,
                attributes: ['name']
        }]
});
```

When to use:

- Comparing against aggregated values
- Finding records relative to their group
- Complex filtering conditions

Performance tips:

- Use indexes on compared columns
- Consider materializing frequently used calculations
- Monitor query execution time

4.3 Advanced Joins and Relationships

4.3.1 Complex Join Operations

Understanding how to handle complex joins efficiently is crucial for large applications.

Multiple Join Types

Different Join Types

```
// Inner Join (default with required: true)
const results = await Order.findAll({
        include: [{
                model: User,
                required: true  // INNER JOIN
        }]
});

// Left Outer Join (default)
const results = await Order.findAll({
        include: [{
                model: User,
                required: false  // LEFT OUTER JOIN
        }]
});

// Right Join (using literal SQL)
const results = await sequelize.query(`
SELECT "Orders".*, "Users".*
FROM "Orders"
RIGHT JOIN "Users" ON "Orders"."userId" = "Users"."id"
`, {
        type: QueryTypes.SELECT,
        model: Order,
        include: [User]
});
```

Understanding join types:

- INNER JOIN: Only matching records
- LEFT JOIN: All records from left table
- RIGHT JOIN: All records from right table
- FULL JOIN: All records from both tables

When to use each:

- INNER JOIN: When you need data from both tables
- LEFT JOIN: When you want all records from main table
- RIGHT JOIN: Rarely used, prefer LEFT JOIN
- FULL JOIN: When you need all possible combinations

4.3.2 Advanced Association Patterns

Polymorphic Associations

Implementing Polymorphic Associations

Handling relationships where a model can belong to multiple types:

```
// Comment can belong to either Post or Image
const Comment = sequelize.define('Comment', {
        content: DataTypes.TEXT,
        commentableId: DataTypes.INTEGER,
        commentableType: DataTypes.STRING
});

const Post = sequelize.define('Post', {
        title: DataTypes.STRING,
        content: DataTypes.TEXT
});

const Image = sequelize.define('Image', {
        url: DataTypes.STRING,
        caption: DataTypes.STRING
});

// Helper function to get comments
Post.prototype.getComments = function() {
        return Comment.findAll({
                where: {
                        commentableId: this.id,
                        commentableType: 'Post'
                }
        });
};

Image.prototype.getComments = function() {
        return Comment.findAll({
                where: {
                        commentableId: this.id,
                        commentableType: 'Image'
                }
        });
};
```

Important considerations:

- Index both commentableId and commentableType
- Consider using a discriminator pattern
- Handle cascading deletes carefully
- Maintain data integrity

4.4 Query Performance Optimization

4.4.1 Query Planning and Analysis

Understanding Query Plans

Analyzing Query Execution

Use EXPLAIN ANALYZE to understand query performance:

```
const analyzeQuery = async () => {
        const [analysis] = await sequelize.query(`
        EXPLAIN ANALYZE
        SELECT "Users".*, COUNT("Orders"."id") as "orderCount"
        FROM "Users"
        LEFT JOIN "Orders" ON "Users"."id" = "Orders"."userId"
        GROUP BY "Users"."id"
        `);

        console.log(analysis);
};
```

Key metrics to analyze:

- Scan type (Sequential vs Index)
- Execution time
- Number of rows examined
- Join strategies used

Optimization strategies:

- Add appropriate indexes
- Rewrite queries to use indexes
- Consider materialized views
- Optimize JOIN conditions

4.4.2 Caching Strategies

Query Result Caching

Implementing Caching

Using Redis for query caching:

```
const Redis = require('ioredis');
const redis = new Redis();

const getCachedUsers = async () => {
        const cacheKey = 'users:active';

        // Try to get from cache
        let users = await redis.get(cacheKey);

        if (users) {
                return JSON.parse(users);
        }

        // If not in cache, query database
        users = await User.findAll({
                where: { status: 'active' },
                include: [{
                        model: Profile,
                        attributes: ['avatar']
                }]
        });

        // Cache the result
        await redis.setex(cacheKey, 3600, JSON.stringify(users));

        return users;
};
```

Caching considerations:

- Choose appropriate cache duration
- Handle cache invalidation
- Consider memory usage
- Monitor cache hit rates

4.5 Database Scaling Strategies

4.5.1 Read Replicas

Configuring Read Replicas

Setting up read replicas in Sequelize:

```
const sequelize = new Sequelize({
        dialect: 'postgres',
        replication: {
                read: [
                { host: 'read-replica-1', username: '...', password: '...' },
                { host: 'read-replica-2', username: '...', password: '...' }
                ],
                write: { host: 'master', username: '...', password: '...' }
        }
});

// Queries automatically use read replicas
const users = await User.findAll();  // Uses read replica

// Force master for critical reads
const user = await User.findByPk(id, {
        useMaster: true
});
```

Best practices:

- Use read replicas for heavy read operations
- Monitor replication lag
- Handle eventual consistency
- Implement proper failover

4.6 Performance Monitoring

4.6.1 Query Monitoring

Implementing Query Monitoring

```
const sequelize = new Sequelize(config, {
        benchmark: true,
        logging: (sql, timing) => {
                // Log slow queries
                if (timing > 1000) {
                        console.warn(`Slow query (${timing}ms):`, sql);

                        // Send to monitoring service
                        monitor.trackQuery({
                                sql,
                                timing,
                                timestamp: new Date()
                        });
                }

                // Track query patterns
                const queryType = getQueryType(sql);
                metrics.incrementCounter(`query.${queryType}`);
        }
});

// Custom query logger
const QueryLogger = {
        logQuery: async (sql, timing) => {
                await QueryLog.create({
                        sql,
                        executionTime: timing,
                        timestamp: new Date()
                });
        }
};
```

Monitoring metrics:

- Query execution time
- Query patterns and frequency
- Slow query analysis
- Resource utilization

4.7 Best Practices

4.7.1 Query Optimization Checklist

- Profile queries before optimization
- Use appropriate indexes
- Implement caching where appropriate
- Monitor query performance
- Use read replicas for heavy read operations
- Implement proper error handling
- Regular maintenance and monitoring

4.8 Summary

In this chapter, we covered:

- Complex query patterns
- Advanced joins and relationships
- Query performance optimization
- Database scaling strategies
- Performance monitoring

Chapter 5

Database Migrations and Schema Management

5.1 Introduction

Database migrations are essential for managing schema changes in a versioned and organized way. This chapter covers how to effectively manage database schema evolution using Sequelize migrations.

5.2 Understanding Migrations

5.2.1 What Are Migrations?

Migration Basics

Migrations are like version control for your database schema. They allow you to:

- Track database changes
- Share schema changes with team members
- Roll back changes when needed
- Maintain data integrity during updates

5.2.2 Setting Up Migrations

Initial Setup

Migration Configuration

First, install the Sequelize CLI:

```
npm install --save-dev sequelize-cli
```

Create a configuration file (`.sequelizerc`):

```
const path = require('path');

module.exports = {
        'config': path.resolve('config', 'database.js'),
        'models-path': path.resolve('models'),
        'seeders-path': path.resolve('seeders'),
        'migrations-path': path.resolve('migrations')
};
```

Initialize Sequelize project structure:

```
npx sequelize-cli init
```

This creates:

- `config/database.js` - Database configuration
- `models/` - Model definitions
- `migrations/` - Migration files
- `seeders/` - Seed data files

5.3 Creating Migrations

5.3.1 Basic Migration Operations

Creating Tables

Table Creation Migration

Generate a new migration:

```
npx sequelize-cli migration:generate --name create-users-table
```

Implement the migration:

```
module.exports = {
        async up(queryInterface, Sequelize) {
                await queryInterface.createTable('Users', {
                        id: {
                                type: Sequelize.UUID,
                                defaultValue: Sequelize.UUIDV4,
                                primaryKey: true
                        },
                        email: {
                                type: Sequelize.STRING,
                                allowNull: false,
                                unique: true
                        },
                        password: {
                                type: Sequelize.STRING,
                                allowNull: false
                        },
                        status: {
                                type: Sequelize.ENUM('active', 'inactive'),
                                defaultValue: 'active'
                        },
                        createdAt: {
                                type: Sequelize.DATE,
                                allowNull: false
                        },
                        updatedAt: {
                                type: Sequelize.DATE,
                                allowNull: false
                        }
                }, {
                        indexes: [
                        {
                                name: 'users_email_status_idx',
                                fields: ['email', 'status']
                        }
                        ]
                });
        },

        async down(queryInterface, Sequelize) {
                await queryInterface.dropTable('Users');
        }
};
```

Key concepts:

- up method defines changes to apply

Modifying Tables

Table Modification Migration

Adding columns:

```
module.exports = {
        async up(queryInterface, Sequelize) {
                // Add new column
                await queryInterface.addColumn('Users', 'lastName', {
                        type: Sequelize.STRING,
                        allowNull: true,
                        after: 'firstName'  // Position the column
                });

                // Add multiple columns
                await queryInterface.addColumns('Users', {
                        phoneNumber: {
                                type: Sequelize.STRING,
                                allowNull: true
                        },
                        dateOfBirth: {
                                type: Sequelize.DATE,
                                allowNull: true
                        }
                });
        },

        async down(queryInterface, Sequelize) {
                // Remove columns in reverse order
                await queryInterface.removeColumn('Users', 'dateOfBirth');
                await queryInterface.removeColumn('Users', 'phoneNumber');
                await queryInterface.removeColumn('Users', 'lastName');
        }
};
```

Modifying columns:

```
module.exports = {
        async up(queryInterface, Sequelize) {
                await queryInterface.changeColumn('Users', 'email', {
                        type: Sequelize.STRING(100),
                        allowNull: false,
                        unique: true,
                        validate: {
                                isEmail: true
                        }
                });
        },

        async down(queryInterface, Sequelize) {
                await queryInterface.changeColumn('Users', 'email', {
                        type: Sequelize.STRING,
                        allowNull: true,
                        unique: false
                });
        }
};
```

5.3.2 Advanced Migration Patterns

Data Migrations

Migrating and transforming data:

```
module.exports = {
        async up(queryInterface, Sequelize) {
                // Create temporary column
                await queryInterface.addColumn('Users', 'fullName', {
                        type: Sequelize.STRING
                });

                // Update data
                await queryInterface.sequelize.query(`
                UPDATE "Users"
                SET "fullName" = CONCAT("firstName", ' ', "lastName")
                WHERE "firstName" IS NOT NULL
                AND "lastName" IS NOT NULL
                `);

                // Remove old columns
                await queryInterface.removeColumn('Users', 'firstName');
                await queryInterface.removeColumn('Users', 'lastName');
        },

        async down(queryInterface, Sequelize) {
                // Restore original structure
                await queryInterface.addColumn('Users', 'firstName', {
                        type: Sequelize.STRING
                });
                await queryInterface.addColumn('Users', 'lastName', {
                        type: Sequelize.STRING
                });

                // Split data back
                await queryInterface.sequelize.query(`
                UPDATE "Users"
                SET
                "firstName" = SPLIT_PART("fullName", ' ', 1),
                "lastName" = SPLIT_PART("fullName", ' ', 2)
                WHERE "fullName" IS NOT NULL
                `);

                // Remove temporary column
                await queryInterface.removeColumn('Users', 'fullName');
        }
};
```

Important considerations:

- Handle large datasets in batches
- Consider data integrity
- Plan for rollback scenarios

- Test with representative data

5.4 Migration Best Practices

5.4.1 Planning Migrations

Migration Guidelines

Key principles:

- Make migrations atomic and focused
- Test migrations thoroughly
- Include proper rollback logic
- Document complex migrations
- Consider performance impact

Example checklist:

1. Backup database before migration
2. Test migration on staging
3. Plan deployment window
4. Prepare rollback strategy
5. Monitor system during migration

5.4.2 Zero-Downtime Migrations

Example of adding a non-nullable column:

```
// Step 1: Add nullable column
module.exports = {
        async up(queryInterface, Sequelize) {
                await queryInterface.addColumn('Users', 'role', {
                        type: Sequelize.STRING,
                        allowNull: true
                });
        },
        async down(queryInterface, Sequelize) {
                await queryInterface.removeColumn('Users', 'role');
        }
};

// Step 2: Set default values
```

```
module.exports = {
        async up(queryInterface, Sequelize) {
                await queryInterface.sequelize.query(`
                UPDATE "Users"
                SET role = 'user'
                WHERE role IS NULL
                `);
        },
        async down(queryInterface, Sequelize) {
                await queryInterface.sequelize.query(`
                UPDATE "Users"
                SET role = NULL
                `);
        }
};

// Step 3: Make column non-nullable
module.exports = {
        async up(queryInterface, Sequelize) {
                await queryInterface.changeColumn('Users', 'role', {
                        type: Sequelize.STRING,
                        allowNull: false
                });
        },
        async down(queryInterface, Sequelize) {
                await queryInterface.changeColumn('Users', 'role', {
                        type: Sequelize.STRING,
                        allowNull: true
                });
        }
};
```

Best practices:

- Break changes into smaller steps
- Maintain backward compatibility
- Use feature flags when possible
- Monitor database performance

5.5 Schema Version Control

5.5.1 Managing Migration History

Migration Management

Running migrations:

```
# Run all pending migrations
npx sequelize-cli db:migrate

# Undo last migration
npx sequelize-cli db:migrate:undo

# Undo all migrations
npx sequelize-cli db:migrate:undo:all

# Run migrations up to a specific one
npx sequelize-cli db:migrate --to XXXXXX-migration-name.js
```

Checking status:

```
# View pending migrations
npx sequelize-cli db:migrate:status
```

5.6 Summary

Key takeaways:

- Migration fundamentals
- Best practices for schema changes
- Zero-downtime migration patterns
- Version control strategies

Chapter 6

Advanced Model Patterns and Data Validation

6.1 Introduction

This chapter explores advanced patterns for model design, complex validation scenarios, and sophisticated data handling techniques using Sequelize.

6.2 Advanced Model Patterns

6.2.1 Model Inheritance

Model inheritance allows you to share common attributes and methods between models.

Single Table Inheritance

Example of a content management system:

```
// Base Content model
class Content extends Model {
        static init(sequelize) {
                return super.init({
                        id: {
                                type: DataTypes.UUID,
                                defaultValue: DataTypes.UUIDV4,
                                primaryKey: true
                        },
                        title: {
                                type: DataTypes.STRING,
                                allowNull: false
                        },
                        type: {
```

```
                        type: DataTypes.STRING,
                        allowNull: false
                },
                content: {
                        type: DataTypes.TEXT
                },
                metadata: {
                        type: DataTypes.JSONB,
                        defaultValue: {}
                }
        }, {
                sequelize,
                tableName: 'contents',
                discriminator: 'type'
        });
    }

    static includeOptions() {
        return {};
    }
}

// Article model extending Content
class Article extends Content {
    static init(sequelize) {
        super.init(sequelize);
        this.addHook('beforeCreate', (instance) => {
            instance.type = 'article';
        });
        return this;
    }

    static includeOptions() {
        return {
            ...super.includeOptions(),
            where: { type: 'article' }
        };
    }

    get excerpt() {
        return this.content.substring(0, 150) + '...';
    }
}

// Video model extending Content
class Video extends Content {
    static init(sequelize) {
        super.init(sequelize);
        this.addHook('beforeCreate', (instance) => {
            instance.type = 'video';
        });
        return this;
    }

    static includeOptions() {
        return {
            ...super.includeOptions(),
            where: { type: 'video' }
```

```
                };
        }

        get duration() {
                return this.metadata.duration || 0;
        }
}
```

Usage:

```
// Create different content types
const article = await Article.create({
        title: 'Understanding STI',
        content: 'Single Table Inheritance...'
});

const video = await Video.create({
        title: 'STI Tutorial',
        content: 'Video description',
        metadata: { duration: 360 }
});

// Query specific content types
const articles = await Article.findAll();
const videos = await Video.findAll();
```

Key concepts:

- Use discriminator column to differentiate types
- Share common attributes in base model
- Extend functionality in child models
- Maintain type-specific behavior

6.2.2 Model Mixins

Example of a timestamped soft-delete mixin:

```
// Soft Delete Mixin
const SoftDeleteMixin = {
        addSoftDelete(Model) {
                Model.addHook('beforeFind', (options) => {
                        if (!options.withDeleted) {
                                options.where = {
                                        ...options.where,
                                        deletedAt: null
                                };
                        }
                });

                Model.prototype.softDelete = async function() {
```

```
                this.deletedAt = new Date();
                await this.save();
        };

        Model.prototype.restore = async function() {
                this.deletedAt = null;
                await this.save();
        };

        Model.withDeleted = function(options = {}) {
                return this.findAll({
                        ...options,
                        withDeleted: true
                });
        };

        return Model;
    }
};

// Timestamp Mixin
const TimestampMixin = {
    addTimestamps(Model) {
        Model.addHook('beforeCreate', (instance) => {
            instance.createdAt = new Date();
            instance.updatedAt = new Date();
        });

        Model.addHook('beforeUpdate', (instance) => {
            instance.updatedAt = new Date();
        });

        return Model;
    }
};

// Using mixins
class User extends Model {
    static init(sequelize) {
        super.init({
            // Model attributes
        }, { sequelize });

        SoftDeleteMixin.addSoftDelete(this);
        TimestampMixin.addTimestamps(this);
        return this;
    }
}
```

Usage:

```
// Regular operations
const user = await User.findByPk(1);
await user.softDelete();

// Find including deleted
const allUsers = await User.withDeleted();
```

```
// Restore deleted user
await user.restore();
```

6.3 Advanced Validation

6.3.1 Custom Validators

Example of custom validation rules:

```
class Order extends Model {
        static init(sequelize) {
                return super.init({
                        items: {
                                type: DataTypes.JSONB,
                                validate: {
                                        isValidItems(value) {
                                                if (!Array.isArray(value)) {
                                                        throw new Error('Items must be an array');
                                                }

                                                if (value.length === 0) {
                                                        throw new Error('Order must contain at least
                                                }

                                                const invalidItems = value.filter(item =>
                                                !item.productId ||
                                                !item.quantity ||
                                                item.quantity <= 0
                                                );

                                                if (invalidItems.length > 0) {
                                                        throw new Error('Invalid items in order');
                                                }
                                        }
                                }
                        },
                        deliveryAddress: {
                                type: DataTypes.JSONB,
                                validate: {
                                        async isValidAddress(value) {
                                                if (!value.street || !value.city || !value.country)
                                                        throw new Error('Incomplete address');
                                                }

                                                // Example: Validate postal code format based on cou
                                                const postalCodeFormats = {
                                                        US: /^\d{5}(-\d{4})?$/,
                                                        UK: /^[A-Z]{1,2}\d[A-Z\d]? ?\d[A-Z]{2}$/
                                                };

                                                const format = postalCodeFormats[value.country];
                                                if (format && !format.test(value.postalCode)) {
                                                        throw new Error('Invalid postal code format'
```

```
                                        }

                                        // Example: External API validation
                                        try {
                                                const isValid = await validateAddressWithAPI
                                                if (!isValid) {
                                                        throw new Error('Invalid address');
                                                }
                                        } catch (error) {
                                                throw new Error('Address validation failed')
                                        }
                                }
                        }
                },
                status: {
                        type: DataTypes.STRING,
                        validate: {
                                isIn: {
                                        args: [['pending', 'processing', 'shipped', 'deliver
                                        msg: 'Invalid order status'
                                },
                                async isValidStatusTransition(value) {
                                        if (this.changed('status')) {
                                                const validTransitions = {
                                                        pending: ['processing'],
                                                        processing: ['shipped'],
                                                        shipped: ['delivered'],
                                                        delivered: []
                                                };

                                                const oldStatus = this.previous('status');
                                                const allowedStatuses = validTransitions[old

                                                if (!allowedStatuses.includes(value)) {
                                                        throw new Error(`Cannot transition f
                                                }
                                        }
                                }
                        }
                }
        }, {
                sequelize,
                hooks: {
                        beforeValidate: async (order) => {
                                // Additional validation logic
                        }
                }
        });
    }
}
```

6.3.2 Context-Aware Validation

Example of validation that depends on context:

```
class Product extends Model {
        static init(sequelize) {
                return super.init({
                        price: {
                                type: DataTypes.DECIMAL(10, 2),
                                validate: {
                                        async isPriceValid(value) {
                                                // Different validation based on product type
                                                if (this.type === 'subscription') {
                                                        if (value < 0) {
                                                                throw new Error('Subscription price
                                                        }
                                                } else {
                                                        if (value <= 0) {
                                                                throw new Error('Product price must
                                                        }
                                                }

                                                // Price change validation
                                                if (this.changed('price')) {
                                                        const maxChange = this.type === 'subscriptio
                                                        const oldPrice = this.previous('price');

                                                        if (oldPrice) {
                                                                const changePercent = Math.abs(value
                                                                if (changePercent > maxChange) {
                                                                        throw new Error(
                                                                        `Price change cannot exceed
                                                                        );
                                                                }
                                                        }
                                                }
                                        }
                                }
                        },
                        stock: {
                                type: DataTypes.INTEGER,
                                validate: {
                                        async isStockValid(value) {
                                                // Skip stock validation for digital products
                                                if (this.type === 'digital') {
                                                        return;
                                                }

                                                if (value < 0) {
                                                        throw new Error('Stock cannot be negative');
                                                }

                                                // Validate against pending orders
                                                const pendingOrdersCount = await OrderItem.sum('quan
                                                        where: {
                                                                productId: this.id,
                                                                status: 'pending'
                                                        }
                                                });

                                                if (value < pendingOrdersCount) {
                                                        throw new Error(
```

```
                                                        'Stock cannot be less than pending order qua
                                                        );
                                                }
                                        }
                                }
                        }, { sequelize });
        }
}
```

6.4 Model Hooks

6.4.1 Advanced Hook Patterns

Example of sophisticated hook usage:

```
class Order extends Model {
        static init(sequelize) {
                return super.init({
                        // ... attributes
                }, {
                        hooks: {
                                // Validate complex business rules
                                beforeValidate: async (order) => {
                                        if (order.isNewRecord) {
                                                await validateUserPurchaseLimit(order);
                                        }
                                },

                                // Ensure data consistency
                                beforeCreate: async (order) => {
                                        await lockInventory(order);
                                },

                                // Handle side effects
                                afterCreate: async (order) => {
                                        await Promise.all([
                                        updateUserPurchaseHistory(order),
                                        sendOrderNotifications(order),
                                        updateInventoryStats(order)
                                        ]);
                                },

                                // Cleanup on failure
                                afterCreate: async (order, options) => {
                                        if (options.transaction) {
                                                options.transaction.afterCommit(() => {
                                                        processOrderSuccess(order);
                                                });
                                        }
                                },

                                // Validate status transitions
                                beforeUpdate: async (order) => {
```

```
                                        if (order.changed('status')) {
                                                await validateStatusTransition(order);
                                        }
                                },

                                // Handle cascading updates
                                afterUpdate: async (order) => {
                                        if (order.changed('status')) {
                                                await updateRelatedRecords(order);
                                        }
                                }
                        }
                });
        }
}

// Helper functions
async function validateUserPurchaseLimit(order) {
        const user = await order.getUser();
        const monthlyOrders = await Order.sum('totalAmount', {
                where: {
                        userId: user.id,
                        createdAt: {
                                [Op.gte]: moment().startOf('month').toDate()
                        }
                }
        });

        if (monthlyOrders + order.totalAmount > user.monthlyLimit) {
                throw new Error('Monthly purchase limit exceeded');
        }
}

async function lockInventory(order) {
        const items = await order.getItems();

        for (const item of items) {
                const product = await item.getProduct();

                if (product.stock < item.quantity) {
                        throw new Error(`Insufficient stock for ${product.name}`);
                }

                await product.decrement('stock', {
                        by: item.quantity,
                        lock: true
                });
        }
}

async function validateStatusTransition(order) {
        const validTransitions = {
                pending: ['processing', 'cancelled'],
                processing: ['shipped', 'cancelled'],
                shipped: ['delivered'],
                delivered: [],
                cancelled: []
        };
```

```
        const oldStatus = order.previous('status');
        const newStatus = order.get('status');

        if (!validTransitions[oldStatus].includes(newStatus)) {
                throw new Error(
                `Invalid status transition from ${oldStatus} to ${newStatus}`
                );
        }
}
```

6.5 Best Practices

6.5.1 Model Organization

- Separate business logic into services
- Use mixins for shared functionality
- Implement proper error handling
- Document complex validation rules
- Test edge cases thoroughly

6.6 Summary

Key takeaways:

- Advanced model patterns
- Complex validation strategies
- Sophisticated hook implementations
- Best practices for model organization

Chapter 7

Deployment and Production Considerations

7.1 Introduction

Deploying a Sequelize application to production requires careful planning and consideration of various factors including performance, security, and maintainability. This chapter covers essential aspects of deployment and production management.

7.2 Production Configuration

7.2.1 Environment Setup

Proper configuration for production environments:

```
// config/database.js
require('dotenv').config();

module.exports = {
        production: {
                dialect: 'postgres',
                host: process.env.DB_HOST,
                database: process.env.DB_NAME,
                username: process.env.DB_USER,
                password: process.env.DB_PASSWORD,
                pool: {
                        max: parseInt(process.env.DB_POOL_MAX || '10'),
                        min: parseInt(process.env.DB_POOL_MIN || '2'),
                        acquire: 30000,
                        idle: 10000
                },
                dialectOptions: {
                        ssl: {
                                require: true,
```

```
                        rejectUnauthorized: false
                    },
                    keepAlive: true
                },
                logging: false, // Disable SQL logging in production
                benchmark: true // Enable query timing
        }
};

// src/database.js
const { Sequelize } = require('sequelize');
const config = require('../config/database');

const env = process.env.NODE_ENV || 'development';
const dbConfig = config[env];

const sequelize = new Sequelize({
        ...dbConfig,
        define: {
                timestamps: true,
                underscored: true,
                paranoid: true // Enable soft deletes globally
        },
        hooks: {
                beforeConnect: async (config) => {
                        console.log('Connecting to database...');
                },
                afterConnect: async (connection) => {
                        console.log('Database connected successfully');
                }
        }
});

module.exports = sequelize;
```

7.2.2 Connection Management

Implementing robust connection handling:

```
// src/database/connectionManager.js
class DatabaseConnectionManager {
        constructor(sequelize) {
                this.sequelize = sequelize;
                this.maxRetries = 5;
                this.retryDelay = 5000;
        }

        async connect() {
                let retries = 0;

                while (retries < this.maxRetries) {
                        try {
                                await this.sequelize.authenticate();
                                console.log('Database connection established');
                                return true;
```

```
                        } catch (error) {
                                retries++;
                                console.error(
                                `Failed to connect to database (attempt ${retries}/${this.maxRetries
                                error.message
                                );

                                if (retries === this.maxRetries) {
                                        throw new Error('Failed to connect to database');
                                }

                                await new Promise(resolve =>
                                setTimeout(resolve, this.retryDelay)
                                );
                        }
                }
        }

        async healthCheck() {
                try {
                        await this.sequelize.query('SELECT 1');
                        return true;
                } catch (error) {
                        console.error('Database health check failed:', error);
                        return false;
                }
        }
}
```

7.3 Performance Optimization

7.3.1 Query Optimization

Implementing query optimization strategies:

```
// src/services/queryOptimizer.js
class QueryOptimizer {
        static async findWithPagination(model, options = {}) {
                const {
                        page = 1,
                        pageSize = 20,
                        order = [['createdAt', 'DESC']],
                        ...queryOptions
                } = options;

                const offset = (page - 1) * pageSize;

                // Use separate count query for better performance
                const [count, rows] = await Promise.all([
                model.count({ where: queryOptions.where }),
                model.findAll({
                        ...queryOptions,
                        limit: pageSize,
                        offset,
```

```
                        order
                ])
                ]);

                return {
                        rows,
                        pagination: {
                                page,
                                pageSize,
                                totalPages: Math.ceil(count / pageSize),
                                totalItems: count
                        }
                };
        }

        static createQueryLoggingInterceptor() {
                return {
                        async before(options) {
                                options._startTime = Date.now();
                        },
                        async after(options) {
                                const duration = Date.now() - options._startTime;
                                if (duration > 1000) { // Log slow queries
                                        console.warn(`Slow query (${duration}ms):`, options.sql);
                                }
                        }
                };
        }
}
```

7.4 Monitoring and Logging

7.4.1 Implementing Monitoring

Setting up comprehensive monitoring:

```
// src/monitoring/databaseMonitor.js
class DatabaseMonitor {
        constructor(sequelize) {
                this.sequelize = sequelize;
                this.metrics = {
                        queries: {
                                total: 0,
                                failed: 0,
                                slow: 0
                        },
                        connections: {
                                active: 0,
                                idle: 0,
                                failed: 0
                        }
                };
        }
```

```
        setupQueryLogging() {
                this.sequelize.options.benchmark = true;
                this.sequelize.options.logging = (sql, timing) => {
                        this.metrics.queries.total++;

                        if (timing > 1000) {
                                this.metrics.queries.slow++;
                                console.warn(`Slow query (${timing}ms):`, sql);
                        }
                };
        }

        async getPoolStatus() {
                const pool = this.sequelize.connectionManager.pool;
                return {
                        total: pool.size,
                        idle: pool.idle,
                        active: pool.size - pool.idle
                };
        }

        async getMetrics() {
                const poolStatus = await this.getPoolStatus();
                return {
                        ...this.metrics,
                        pool: poolStatus,
                        timestamp: new Date()
                };
        }
}
```

7.4.2 Error Handling

Implementing robust error handling:

```
// src/middleware/errorHandler.js
class DatabaseErrorHandler {
        static handle(error) {
                if (error instanceof Sequelize.ConnectionError) {
                        return {
                                status: 503,
                                message: 'Database connection error',
                                retryAfter: 30
                        };
                }

                if (error instanceof Sequelize.ValidationError) {
                        return {
                                status: 400,
                                message: 'Validation error',
                                errors: error.errors.map(err => ({
                                        field: err.path,
                                        message: err.message
                                }))
                        };
```

```
                }

                if (error instanceof Sequelize.UniqueConstraintError) {
                        return {
                                status: 409,
                                message: 'Duplicate entry',
                                errors: error.errors.map(err => ({
                                        field: err.path,
                                        message: 'Already exists'
                                }))
                        };
                }

                // Default error
                return {
                        status: 500,
                        message: 'Internal server error'
                };
        }
}
```

7.5 Scaling Strategies

7.5.1 Read Replicas

Configuring read replicas for better performance:

```
// config/database.js
module.exports = {
        production: {
                dialect: 'postgres',
                replication: {
                        read: [
                        { host: 'read-replica-1', username: '...' },
                        { host: 'read-replica-2', username: '...' }
                        ],
                        write: { host: 'master', username: '...' }
                },
                pool: {
                        max: 20,
                        idle: 30000
                }
        }
};

// Usage in services
class UserService {
        async findUsers(criteria) {
                // Read from replica
                return await User.findAll({
                        where: criteria
                });
        }
```

```
        async createUser(data) {
                // Write to master
                return await User.create(data, {
                        useMaster: true
                });
        }
}
```

7.5.2 Connection Pooling

Implementing advanced connection pooling:

```
// src/database/poolConfig.js
const poolConfig = {
        max: 20, // Maximum pool size
        min: 5,  // Minimum pool size
        acquire: 30000, // Maximum time to acquire connection
        idle: 10000,    // Maximum idle time
        evict: 1000,    // Run eviction every 1 second
        validate: async (connection) => {
                try {
                        await connection.query('SELECT 1');
                        return true;
                } catch (e) {
                        return false;
                }
        }
};

const sequelize = new Sequelize({
        ...config,
        pool: poolConfig,
        hooks: {
                beforeConnect: async (config) => {
                        // Log connection attempts
                },
                afterDisconnect: async () => {
                        // Clean up resources
                }
        }
});
```

7.6 Deployment Process

7.6.1 Migration Strategy

Implementing safe database migrations:

```
// scripts/deploy.js
async function deploy() {
        try {
```

```
                // 1. Backup database
                await backupDatabase();

                // 2. Run migrations
                await sequelize.authenticate();
                await runMigrations();

                // 3. Verify migrations
                await verifyMigrations();

                // 4. Update application
                await updateApplication();

        } catch (error) {
                // Rollback if needed
                await rollback();
                throw error;
        }
}

async function runMigrations() {
        const umzug = new Umzug({
                migrations: {
                        path: './migrations',
                        params: [sequelize.getQueryInterface()]
                },
                storage: 'sequelize',
                storageOptions: { sequelize }
        });

        return umzug.up();
}
```

7.7 Maintenance Procedures

7.7.1 Database Maintenance

Regular maintenance tasks:

```
// scripts/maintenance.js
class DatabaseMaintenance {
        static async vacuum() {
                await sequelize.query('VACUUM ANALYZE');
        }

        static async reindex() {
                const tables = await sequelize.getQueryInterface()
                .showAllTables();

                for (const table of tables) {
                        await sequelize.query(`REINDEX TABLE "${table}"`);
                }
        }
```

```
        static async cleanupSoftDeleted() {
                const models = Object.values(sequelize.models)
                .filter(model => model.options.paranoid);

                for (const model of models) {
                        await model.destroy({
                                where: {
                                        deletedAt: {
                                                [Op.lt]: moment().subtract(3, 'months')
                                        }
                                },
                                force: true
                        });
                }
        }
}
```

7.8 Security Considerations

7.8.1 SQL Injection Prevention

Implementing secure query practices:

```
// src/services/secureQueryBuilder.js
class SecureQueryBuilder {
        static buildWhereClause(filters) {
                const where = {};
                const allowedOperators = ['eq', 'gt', 'lt', 'like'];

                for (const [key, value] of Object.entries(filters)) {
                        if (typeof value === 'object') {
                                const operator = Object.keys(value)[0];
                                if (!allowedOperators.includes(operator)) {
                                        throw new Error(`Invalid operator: ${operator}`);
                                }
                                where[key] = { [Op[operator]]: value[operator] };
                        } else {
                                where[key] = value;
                        }
                }

                return where;
        }

        static sanitizeOrder(orderBy) {
                const allowedFields = ['id', 'createdAt', 'updatedAt'];
                const [field, direction] = orderBy.split(' ');

                if (!allowedFields.includes(field)) {
                        throw new Error(`Invalid order field: ${field}`);
                }

                return [[field, direction.toUpperCase()]];
```

```
        }
}
```

7.9 Best Practices

7.9.1 Deployment Checklist

- Backup database before deployment
- Run migrations in a transaction
- Verify database connections
- Monitor query performance
- Set up proper logging
- Configure connection pools
- Implement health checks
- Set up monitoring alerts

7.10 Summary

Key takeaways:

- Production configuration best practices
- Performance optimization techniques
- Monitoring and logging strategies
- Scaling approaches
- Security considerations
- Maintenance procedures