

JalaUniversity

A large, light blue watermark of the Jala University logo is centered in the background. It features a stylized 'U' shape on the left and the text 'Jala University' on the right.

Programación IV

Lab Week2

Name: Aldo Ochoa Condori

Tutor: Luis Morales Ledezma

Topic: Introducción a ES6 y Principios de programación funcional

13/07/2024

ORM Best Practices for Node.js Applications

Introduction

Object-Relational Mapping (ORM) libraries simplify the interaction between applications and databases by allowing developers to interact with the database using JavaScript objects instead of raw SQL queries. This guide explores best practices for integrating and using ORM libraries in Node.js applications, covering ORM selection, setup/configuration, model definition, querying, performance optimization, and handling relationships.

ORM Selection

When selecting an ORM for a Node.js application, it's crucial to consider the features, community support, and suitability for the database type (SQL vs NoSQL). Here are comparisons of some popular ORM libraries:

Sequelize

- **Features:** Sequelize is a promise-based ORM for Node.js and supports various SQL dialects (Postgres, MySQL, MariaDB, SQLite, and Microsoft SQL Server). It offers features like migrations, schema synchronization, and transaction management.
- **Community Support:** Strong community with extensive documentation and numerous plugins.
- **Suitability:** Best suited for SQL databases.

TypeORM

- **Features:** TypeORM is an ORM that supports both SQL and NoSQL databases (MySQL, MariaDB, PostgreSQL, SQLite, Microsoft SQL Server, Oracle, and MongoDB). It offers support for migrations, decorators for defining models, and advanced querying.
- **Community Support:** Growing community with good documentation and active development.
- **Suitability:** Suitable for both SQL and NoSQL databases.

Mongoose

- **Features:** Mongoose is an ODM (Object Data Modeling) library for MongoDB and Node.js. It provides schema-based solutions to model data, built-in type casting, validation, query building, and business logic hooks.
- **Community Support:** Very strong community with extensive documentation and tutorials.

- **Suitability:** Best suited for MongoDB (NoSQL).

Setup and Configuration

Database Connection Management:

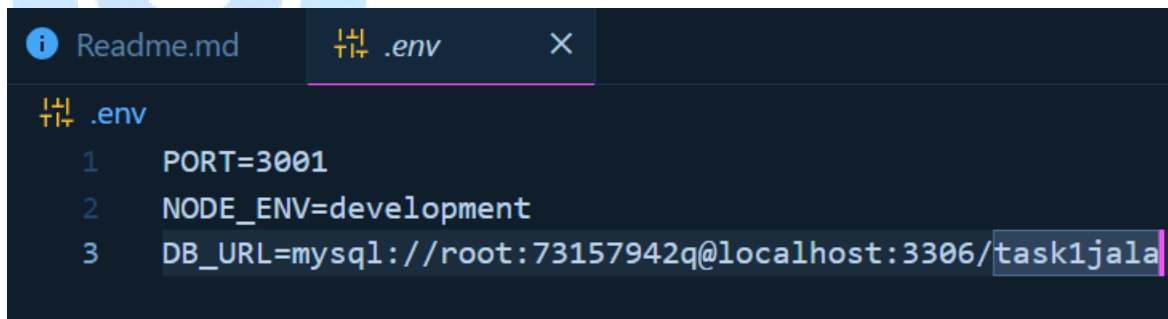
- Use connection pooling to manage database connections efficiently.
- Close database connections gracefully when the application shuts down.

Environment-Specific Configurations:

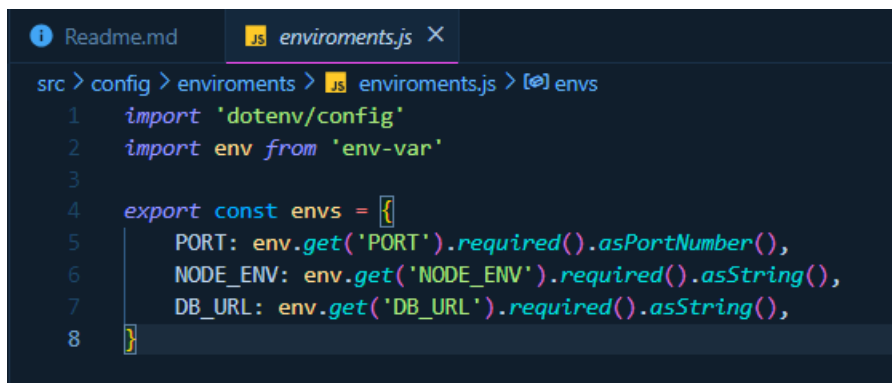
- Use environment variables to manage database configurations for different environments (development, testing, production).
- Store sensitive information such as database credentials in environment variables or secure configuration files.

ORM Initialization:

- Initialize the ORM library at the start of the application.
- Ensure that the database connection is established before handling any requests.



```
Readme.md .env X
1 PORT=3001
2 NODE_ENV=development
3 DB_URL=mysql://root:73157942q@localhost:3306/task1jala
```



```
Readme.md JS enviroments.js X
src > config > enviroments > JS enviroments.js > [0] envs
1 import 'dotenv/config'
2 import env from 'env-var'
3
4 export const envs = {
5   PORT: env.get('PORT').required().asPortNumber(),
6   NODE_ENV: env.get('NODE_ENV').required().asString(),
7   DB_URL: env.get('DB_URL').required().asString(),
8 }
```

```
src > config > database > .js database.js > [e] authenticated
1  import { Sequelize } from "sequelize";
2  import { envs } from "../enviroments/enviroments.js";
3
4  export const sequelize = new Sequelize(envs.DB_URL,{
5    dialect:'mysql',
6    logging:console.log,
7    define:{
8      timestamps:false,
9      freezeTableName:true
10   }
11 })
12
13 export const authenticated = async() =>{
14   try{
15     await sequelize.authenticate()
16     console.log('Connection has been sucessfull.')
17   }catch(error){
18     console.log(error)
19   }
20 }
21
22
23 export const syncUp = async()=>{
24   try{
25     await sequelize.sync({/*force:true*/});
26     console.log('Synced has been sucessfull')
27   }catch(error){
28     console.log(error)
29   }
30 }
```

Model Definition

Defining Models:

- Use clear and consistent naming conventions for model definitions.
- Define models in separate files to keep the codebase organized.

Relationships:

- Use associations (hasOne, hasMany, belongsTo, belongsToMany) to define relationships between models.
- Ensure referential integrity by defining foreign keys correctly.

Validations:

- Use built-in validation features provided by the ORM to enforce data integrity.
- Define custom validation functions for complex validation logic.

```
src > Module > Juego > JS Juego.model.js > ...
1  import { DataTypes } from "sequelize";
2  import { sequelize } from "../../config/database/database.js";
3
4  const Juego = sequelize.define("Juegos",{
5    id: {
6      primaryKey: true,
7      type: DataTypes.INTEGER,
8      autoIncrement: true,
9      allowNull:false
10   },
11   name: {
12     type: DataTypes.STRING(60),
13     allowNull:false
14   },
15   description:{
16     type:DataTypes.STRING(100),
17     allowNull:false
18   },
19   genre: {
20     type:DataTypes.ENUM('Adventure','Racing'),
21     allowNull:false,
22   },
23   platform:{
24     type: DataTypes.ENUM('Mobile','PC'),
25     allowNull:false
26   }
27 });
28
29
30 export default Juego;
```

Querying and Transactions

Basic CRUD Operations:

- Use the ORM's built-in methods for basic CRUD operations to ensure consistency and security.

Advanced Querying:

- Utilize features like eager loading to fetch related data in a single query.
- Use raw queries judiciously for complex operations that are not well-supported by the ORM.

Transaction Management:

- Use transactions to ensure data consistency in complex operations that involve multiple queries.
- Handle transaction commits and rollbacks appropriately to manage errors

```
src > Module > Juego > Juego.service.js > ...
1  import Juego from "../Juego.model.js";
2
3  export class GameService{
4
5      static async findGame(id){
6          return await Juego.findOne({
7              attributes:{
8                  exclude:[
9                      'createdAt',
10                     'updatedAt',
11                 ]
12             },
13             where: {
14                 id:id
15             }
16         })
17     }
18
19     static async findAllGames(){
20         return await Juego.findAll({
21             attributes:{
22                 exclude:[
23                     'createdAt',
24                     'updatedAt',
25                 ]
26             }
27         })
28     }
29
30     static async createGame(data) {
31         return await Juego.create(data)
32     }
33
34     static async updateGame(juego,data){
35         return await juego.update(data)
36     }
37 }
```

Performance Optimization

1. Eager Loading:

Use eager loading to minimize the number of database queries and fetch related data in a single query.

2. Query Optimization:

- Optimize queries by selecting only the required fields.
- Use indexes to improve query performance for frequently accessed fields.

3. Caching:

- Implement caching strategies to reduce the load on the database for frequently accessed data.

4. Indexing:

- Use database indexes to speed up queries on large datasets.
- Ensure that indexes are used judiciously to avoid performance degradation due to excessive indexing.

