ODM y ORM en MySQL y MongoDB

Julian F. Latorre

Julio 31 de 2024

Índice

1. Marco Conceptual

La siguiente tabla presenta los conceptos clave necesarios para comprender ORM y ODM en profundidad:

2. ORM en MySQL

2.1. Definición y Funcionamiento de ORM

Object-Relational Mapping (ORM) es una técnica que permite convertir datos entre sistemas de tipos incompatibles en lenguajes de programación orien-

Categoría	Conceptos
Bases de Datos	Base de Datos Relacional, Base de Datos NoSQL
Bases de Datos Relacionales	Tabla, Columna, Fila, Clave Primaria, Clave Foránea, Índice, Consulta SQL
Bases de Datos NoSQL (MongoDB)	Documento, Colección, Campo, _id
Programación Orientada a Objetos	Clase, Objeto, Atributo, Método
Web y JavaScript	API, REST, JSON, Promise, async/await
Específicos de ORM/ODM	Mapeo, Esquema, Modelo, Migración, Seeding, Lazy Loading, Eager Loading, Query Builder, Middleware

Cuadro 1: Conceptos clave para ORM y ODM

tados a objetos. En el contexto de MySQL, ORM mapea:

- Tablas a Clases
- Filas a Objetos
- Columnas a Atributos

2.2. Uso de Conceptos Específicos en ORM

2.2.1. Mapeo

El mapeo es el núcleo de ORM. Por ejemplo, con Sequelize:

```
const User = sequelize.define('User', {
  firstName: {
    type: DataTypes.STRING,
    allowNull: false
  },
  lastName: {
    type: DataTypes.STRING
  }
});
```

Aquí, estamos mapeando una tabla 'User' a una clase 'User' en JavaScript.

2.2.2. Modelo

El modelo es la representación de la tabla en el código:

```
// Crear un nuevo usuario
const newUser = await User.create({ firstName: 'Pepito', lastName:
    'Perez' });

// Buscar un usuario
const user = await User.findOne({ where: { firstName: 'Pepito' } })
    ;
```

2.2.3. Migración

Las migraciones permiten gestionar cambios en el esquema de la base de datos:

```
module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.addColumn('Users', 'email', {
      type: Sequelize.STRING,
      allowNull: false,
      unique: true
    });
  },
  down: async (queryInterface, Sequelize) => {
    await queryInterface.removeColumn('Users', 'email');
  }
};
```

2.2.4. Query Builder

Los ORM suelen proporcionar un query builder para construir consultas complejas:

```
const users = await User.findAll({
  where: {
    age: {
      [Op.gte]: 18
    }
  },
  order: [['createdAt', 'DESC']]
});
```

3. ODM en MongoDB

3.1. Definición y Funcionamiento de ODM

Object-Document Mapping (ODM) es similar a ORM, pero está diseñado para bases de datos orientadas a documentos como MongoDB. ODM mapea:

- Colecciones a Clases
- Documentos a Objetos
- Campos a Atributos

3.2. Uso de Conceptos Específicos en ODM

3.2.1. Esquema

En ODM, los esquemas definen la estructura de los documentos:

```
const userSchema = new mongoose.Schema({
  firstName: { type: String, required: true },
  lastName: String,
  email: { type: String, required: true, unique: true }
});
```

3.2.2. Modelo

El modelo es una clase constructora compilada a partir de un Schema:

```
const User = mongoose.model('User', userSchema);

// Crear un nuevo usuario
const newUser = await User.create({ firstName: 'Pepito', lastName:
    'Perez', email: 'pepito@example.com' });

// Buscar un usuario
const user = await User.findOne({ firstName: 'Pepito' });
```

3.2.3. Middleware

ODM como Mongoose permiten el uso de middleware para procesar documentos:

```
userSchema.pre('save', function(next) {
  this.updatedAt = Date.now();
  next();
});
```

3.2.4. Lazy Loading vs Eager Loading

ODM suele soportar tanto lazy loading como eager loading:

```
// Lazy loading
const user = await User.findById(id);
const posts = await user.getPosts();

// Eager loading
const user = await User.findById(id).populate('posts');
```

4. Implementación en React

En una aplicación React, ORM o ODM se utilizan típicamente en el backend. El frontend de React se comunica con este backend a través de una API. Aquí hay un ejemplo de cómo se podría implementar un componente que utiliza datos de un backend con ODM:

```
import React, { useState, useEffect } from 'react';

function UserList() {
  const [users, setUsers] = useState([]);
```

```
  useEffect(() => {
    fetchUsers();
  }, []);

  const fetchUsers = async () => {
    try {
      const response = await fetch('http://localhost:3000/api/users
    ');
      const data = await response.json();
      setUsers(data);
    } catch (error) {
      console.error('Error fetching users:', error);
    }
  };

  const createUser = async (userData) => {
    try {
      const response = await fetch('http://localhost:3000/api/users
    ', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify(userData),
      });
      const newUser = await response.json();
      setUsers([...users, newUser]);
    } catch (error) {
      console.error('Error creating user:', error);
    }
  };

  return (
    <div>
      <h1>User List</h1>
      <ul>
        {users.map(user => (
          <li key={user._id}>{user.firstName} {user.lastName}</li>
        ))}
      </ul>
      {/* Aquí iría un formulario para crear nuevos usuarios */}
    </div>
  );
}

export default UserList;
```

Este componente utiliza los conceptos de API y Promise/async-await para interactuar con un backend que probablemente esté utilizando ODM para manejar los datos de usuarios.

5. Conclusión

ORM y ODM son herramientas que simplifican la interacción entre las aplicaciones y las bases de datos. Utilizan conceptos como mapeo, modelos, esquemas

y migraciones para proporcionar una abstracción sobre la base de datos subyacente. En el contexto del desarrollo Fullstack, estas herramientas se utilizan típicamente en el backend, con el Frontend comunicándose con el servidor a través de una API RESTful.