Tarea Integrada con Pcas Profesionalizantes II

Análisis de las Implementaciones Actuales

MongoDB (con `mongoose`)

- La conexión a MongoDB se realiza mediante `mongoose.connect`, lo cual es ideal para MongoDB, pero no es compatible con bases de datos SQL.
- `mongoose.set('strictQuery', false)` configura el comportamiento de las consultas. No se requiere SQL directo en esta implementación, ya que `mongoose` maneja la abstracción de las operaciones CRUD.

MySQL (con `mysql2/promise`)

- La configuración se realiza mediante `mysql.createPool`, que establece una conexión con MySQL. Aquí, cada solicitud puede acceder al pool de conexiones a través de `req.db`.
- La tabla `todos` se verifica o crea usando la función `createTodosTable`, que depende de SQL directo.
- Para adaptar el proyecto a otras bases de datos, sería ideal eliminar el SQL directo y manejar los modelos y consultas a través de un ORM.

Propuesta de Implementación de un ORM (Alternativas a Sequelize)

Aquí se presentan dos ORMs que permiten flexibilidad para usar SQL y, en algunos casos, NoSQL. Esto facilita que el cambio de base de datos sólo requiera una modificación en la configuración de conexión.

1. TypeORM

- **Ventajas**: Compatible con múltiples bases de datos como MySQL, PostgreSQL, SQLite e incluso MongoDB (en modo experimental). Define modelos de datos como entidades y permite cambiar la base de datos modificando solo la configuración.
- **Implementación**: Se pueden definir las entidades en lugar de modelos SQL directos, lo cual simplifica el cambio de base de datos.
- **Ejemplo de Configuración**:

```
import { DataSource } from 'typeorm';

const AppDataSource = new DataSource({

  type: 'mysql', // Cambiar a 'mongodb' o cualquier otro si se desea
```

```
  host: 'localhost',
  port: 3306,
  username: 'todolist',
  password: 'todolist',
  database: 'todolist',
  synchronize: true,
  entities: [Todo], // Importar el modelo Todo como entidad
});
AppDataSource.initialize()
  .then(() => console.log('Conectado a la base de datos'))
  .catch((err) => console.error('Error al conectar a la base de datos', err));
```

Definición de Modelo:

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity()
export class Todo {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  title: string;

  @Column()
  completed: boolean;
}
```

—----------

2. Objection.js

- Ventajas: Aunque está diseñado principalmente para SQL, Objection.js usa un enfoque basado en modelos similar al de los ORMs de SQL y tiene compatibilidad con bases de datos SQL a través de `knex.js`.
- Implementación: Define los modelos usando clases JavaScript y simplifica la interacción con bases de datos SQL como MySQL y PostgreSQL.
- Ejemplo de Configuración:

```
const { Model } = require('objection');

const Knex = require('knex');


// Configuración de Knex para MySQL (o cualquier otro tipo de base de datos SQL)

const knex = Knex({

  client: 'mysql',

  connection: {

    host: 'localhost',

    user: 'todolist',

    password: 'todolist',

    database: 'todolist',

  },

});


Model.knex(knex);


console.log('Conectado a la base de datos');
```

—----------

Definición de Modelo:

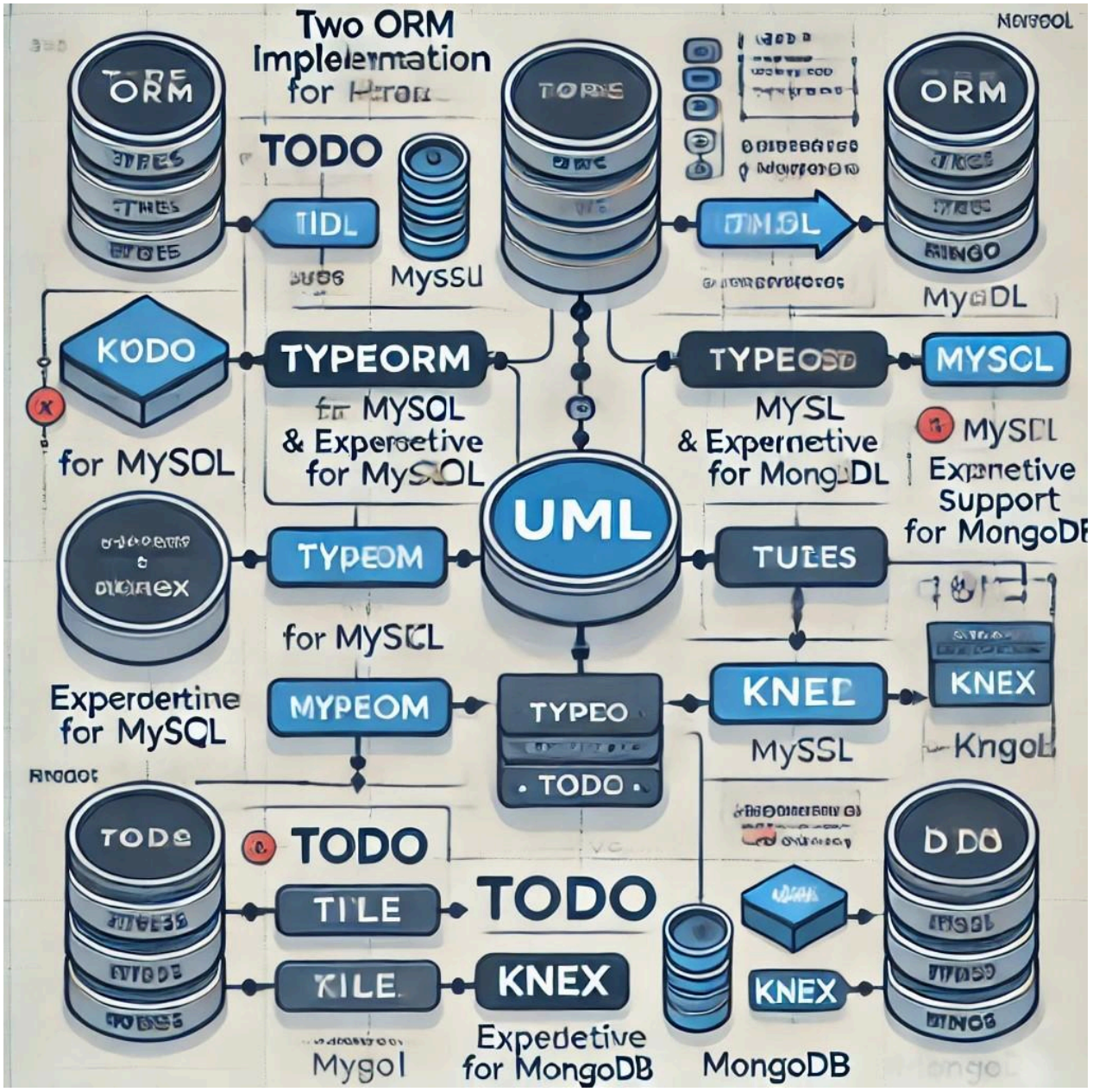
```
const { Model } = require('objection');

class Todo extends Model {
  static get tableName() {
    return 'todos';
  }
}

module.exports = Todo;
```

.

Diagrama que representa las 2 propuestas de implementacion de ORM



Resumen

Usar **TypeORM** o **Objection.js** puede resolver la necesidad de cambiar solo la configuración de conexión. Esto hace que el código sea adaptable a diferentes bases de datos sin requerir reescritura de consultas SQL o configuraciones específicas de NoSQL. Puedes elegir uno de estos ORMs y adaptar las configuraciones del `server.js` para reducir el código dependiente de SQL o `mongoose`, facilitando una transición entre bases de datos según lo necesites