Implementación del pat´ron Repository

Julian F. Latorre

Índice

1. Introducción a MongoDB

MongoDB es una base de datos NoSQL orientada a documentos, diseñada para almacenar grandes volúmenes de datos en documentos similares a JSON. A diferencia de las bases de datos relacionales tradicionales, MongoDB no requiere un esquema predefinido, lo que permite una mayor flexibilidad en la estructura de los datos.

1.1. Características principales

- Modelo de datos basado en documentos
- Esquema flexible
- Escalabilidad horizontal
- Consultas ad hoc y indexación
- Agregación y MapReduce
- Replicación y alta disponibilidad
- Sharding (particionamiento de datos)

2. Instalación y configuración

2.1. Instalación en diferentes sistemas operativos

2.1.1. Windows

Para instalar MongoDB en Windows:

1. Descarga el instalador MSI de la página oficial de MongoDB.
2. Ejecuta el instalador y sigue las instrucciones del asistente.
3. Agrega la ruta de MongoDB a las variables de entorno del sistema.

2.1.2. macOS

Para instalar MongoDB en macOS usando Homebrew:

```
brew tap mongodb/brew
brew install mongodb-community
```

2.1.3. Linux (Ubuntu)

Para instalar MongoDB en Ubuntu:

```
sudo apt-get update
sudo apt-get install -y mongodb
```

2.2. Configuración básica

Después de la instalación, es necesario configurar MongoDB:

1. Crea un directorio para almacenar los datos:

```
sudo mkdir -p /data/db
```

2. Asigna los permisos adecuados:

```
sudo chown -R `id -un` /data/db
```

3. Inicia el servidor MongoDB:

```
mongod
```

3. Operaciones CRUD básicas

3.1. Creación de documentos

Para insertar un documento en MongoDB:

```
db.collection.insertOne({
    nombre: "Juan",
    edad: 30,
    ciudad: "Madrid"
})
```

3.2. Lectura de documentos

Para leer documentos de MongoDB:

```
// Buscar todos los documentos
db.collection.find()

// Buscar documentos con condiciones
db.collection.find({ edad: { $gt: 25 } })
```

3.3. Actualización de documentos

Para actualizar documentos en MongoDB:

```
db.collection.updateOne(
    { nombre: "Juan" },
    { $set: { edad: 31 } }
)
```

3.4. Eliminación de documentos

Para eliminar documentos de MongoDB:

```
db.collection.deleteOne({ nombre: "Juan" })
```

4. Modelado de datos en MongoDB

4.1. Documentos embebidos vs. referencias

4.1.1. Documentos embebidos

Los documentos embebidos son una característica poderosa de MongoDB que permite almacenar datos relacionados en una estructura jerárquica dentro de un único documento. Este enfoque es ideal cuando:

- Los datos relacionados siempre se consultan juntos.
- La relación es de uno a pocos (one-to-few).
- Los datos embebidos no crecen significativamente con el tiempo.

Ejemplo detallado:

```
{
    _id: ObjectId("5f8a7b"),
    nombre: "Juan Pérez",
    edad: 30,
    direcciones: [
        {
            tipo: "casa",
            direccion: "Calle 321 No. 123",
            ciudad: "Bogotá",
            codigoPostal: "266001"
        },
        {
            tipo: "trabajo",
            direccion: "Calle 123 No. 321",
            ciudad: "Bogotá",
            codigoPostal: "266002"
        }
    ],
    contactos: [
        {
            tipo: "email",
            valor: "juan.perez@ejemplo.com"
        },
        {
            tipo: "telefono",
            valor: "+57 123 456 789"
        }
    ]
}
```

En este ejemplo, las direcciones y los contactos están embebidos dentro del documento del usuario. Esto permite recuperar toda la información relacionada con Juan Pérez en una sola consulta.

4.1.2. Referencias

Las referencias, por otro lado, son útiles cuando:

- Los datos relacionados se consultan por separado con frecuencia.
- La relación es de uno a muchos (one-to-many) o de muchos a muchos (many-to-many).
- Los datos referenciados son compartidos por múltiples documentos.

Ejemplo detallado:

```
// Documento de usuario
{
    _id: ObjectId("5f8a7b"),
    nombre: "Juan Pérez",
    edad: 30,
    direcciones: [ObjectId("5f8a7c"), ObjectId("5f8a7d")],
    pedidos: [ObjectId("5f8a7e"), ObjectId("5f8a7f")]
}

// Documentos de direcciones
{
    _id: ObjectId("5f8a7c"),
    usuario_id: ObjectId("5f8a7b"),
    tipo: "casa",
    direccion: "Calle 321 No. 123",
            ciudad: "Bogotá",
            codigoPostal: "266001"
}
{
    _id: ObjectId("5f8a7d"),
    usuario_id: ObjectId("5f8a7b"),
    tipo: "trabajo",
    direccion: "Calle 123 No. 321",
    ciudad: "Bogotá",
    codigoPostal: "266002"
}

// Documentos de pedidos
{
    _id: ObjectId("5f8a7e"),
    usuario_id: ObjectId("5f8a7b"),
    fecha: ISODate("2023-07-30"),
    total: 100.50,
    estado: "completado"
}
{
    _id: ObjectId("5f8a7f"),
    usuario_id: ObjectId("5f8a7b"),
    fecha: ISODate("2023-08-05"),
```

```
    total: 75.25,
    estado: "en proceso"
}
```

En este ejemplo, las direcciones y los pedidos se almacenan en colecciones separadas y se referencian desde el documento del usuario. Esto permite una mayor flexibilidad y escalabilidad, especialmente si necesitamos consultar o actualizar direcciones o pedidos de forma independiente.

4.2. Patrones de diseño comunes

4.2.1. Patrón de subdocumentos

Este patrón es una extensión del concepto de documentos embebidos y es útil para representar relaciones jerárquicas complejas.

Ejemplo detallado:

```
{
    _id: ObjectId("5f8a7b"),
    nombre: "Tienda A",
    ubicacion: {
        direccion: "Calle 123",
        ciudad: "Bogotá",
        codigoPostal: "000001",
        coordenadas: {
            latitud: 72.3851,
            longitud: 4.1734
        }
    },
    inventario: [
        {
            producto: "Laptop",
            marca: "TechBrand",
            modelo: "TB2000",
            especificaciones: {
                procesador: "Intel i7",
                ram: "16GB",
                almacenamiento: "512GB SSD"
            },
            precio: 999999,
            stock: 50
        },
        {
            producto: "Smartphone",
            marca: "MobileTech",
            modelo: "MT100",
            especificaciones: {
                pantalla: "6.5 pulgadas",
                camara: "48MP",
                bateria: "4500mAh"
            },
            precio: 599990,
            stock: 100
        }
    ]
}
```

Este patrón permite representar estructuras de datos complejas y anidadas dentro de un solo documento, lo que facilita la recuperación de toda la información relacionada en una sola consulta.

4.2.2. Patrón de referencias bidireccionales

Este patrón es útil cuando necesitamos navegar eficientemente en ambas direcciones de una relación.

Ejemplo detallado:

```
// Documento de autor
{
    _id: ObjectId("5f8a7b"),
    nombre: "Gabriel García Márquez",
    libros: [ObjectId("5f8a7c"), ObjectId("5f8a7d")]
}

// Documentos de libros
{
    _id: ObjectId("5f8a7c"),
    titulo: "Cien años de soledad",
    autor_id: ObjectId("5f8a7b"),
    año_publicacion: 1967,
    genero: "Realismo mágico"
}
{
    _id: ObjectId("5f8a7d"),
    titulo: "El amor en los tiempos del cólera",
    autor_id: ObjectId("5f8a7b"),
    año_publicacion: 1985,
    genero: "Novela"
}
```

Este patrón permite consultar eficientemente tanto los libros de un autor como el autor de un libro específico.

5. Indexación y optimización de consultas

5.1. Tipos de índices

MongoDB soporta varios tipos de índices para optimizar diferentes tipos de consultas:

5.1.1. Índices simples

Son índices sobre un solo campo del documento.

```
db.usuarios.createIndex({ "nombre": 1 })
```

Este índice mejorará el rendimiento de las consultas que busquen o ordenen por el campo "nombre".

5.1.2. Índices compuestos

Son índices que incluyen múltiples campos.

```
db.usuarios.createIndex({ "ciudad": 1, "edad": -1 })
```

Este índice optimizará las consultas que filtren por ciudad y ordenen por edad en orden descendente.

5.1.3. Índices multikey

Se crean automáticamente cuando indexamos un campo que contiene un array.

```
db.productos.createIndex({ "categorias": 1 })
```

Este índice mejorará las búsquedas en el array de categorías de los productos.

5.1.4. Índices geoespaciales

Optimizan consultas basadas en ubicación geográfica.

```
db.lugares.createIndex({ "ubicacion": "2dsphere" })
```

Este índice permitirá realizar consultas eficientes de proximidad geográfica.

5.1.5. Índices de texto

Permiten realizar búsquedas de texto completo.

```
db.articulos.createIndex({ "contenido": "text" })
```

Este índice facilitará las búsquedas de texto dentro del contenido de los artículos.

5.2. Estrategias de indexación

5.2.1. Índices de cobertura

Un índice de cobertura contiene todos los campos necesarios para responder a una consulta, sin necesidad de acceder a los documentos.

```
db.usuarios.createIndex({ "nombre": 1, "edad": 1 })

// Consulta que utiliza el índice de cobertura
db.usuarios.find({ "nombre": "Juan" }, { "nombre": 1, "edad": 1, "
    _id": 0 })
```

5.2.2. Índices parciales

Permiten indexar solo un subconjunto de documentos que cumplen una condición específica.

```
db.pedidos.createIndex(
    { "fecha_entrega": 1 },
    { partialFilterExpression: { "estado": "pendiente" } }
)
```

Este índice solo incluirá los pedidos con estado "pendiente", optimizando las consultas sobre estos documentos específicos.

5.3. Análisis de consultas

El método ‘explain()‘ es una herramienta poderosa para analizar el rendimiento de las consultas:

```
db.usuarios.find({ "edad": { $gt: 30 } }).explain("executionStats")
```

Este comando proporcionará información detallada sobre cómo MongoDB ejecuta la consulta, incluyendo:

- El plan de consulta seleccionado
- El número de documentos examinados
- El tiempo de ejecución
- Los índices utilizados (si los hay)

6. Agregación en MongoDB

El framework de agregación de MongoDB proporciona un conjunto poderoso de herramientas para realizar operaciones complejas de transformación y análisis de datos.

6.1. Etapas del pipeline de agregación

6.1.1. $match

Filtra los documentos para pasar solo aquellos que cumplen con los criterios especificados.

```
db.ventas.aggregate([
    { $match: { fecha: { $gte: new Date("2023-01-01") } } }
])
```

Este ejemplo filtra las ventas realizadas a partir del 1 de enero de 2023.

6.1.2. $group

Agrupa documentos por una clave especificada y permite realizar cálculos sobre los grupos de documentos.

```
db.ventas.aggregate([
    { $group: {
        _id: "$categoria",
        totalVentas: { $sum: "$monto" },
        promedioVenta: { $avg: "$monto" },
        cantidadVentas: { $sum: 1 }
    }}
])
```

Este ejemplo agrupa las ventas por categoría y calcula el total de ventas, el promedio y la cantidad de ventas para cada categoría.

6.1.3. $project

Pasa los documentos con los campos especificados al siguiente paso del pipeline. Puede ser usado para incluir, excluir o renombrar campos.

```
db.empleados.aggregate([
    { $project: {
        nombreCompleto: { $concat: ["$nombre", " ", "$apellido"] },
        edad: 1,
        salario: 1,
        _id: 0
    }}
])
```

Este ejemplo crea un campo "nombreCompletoçoncatenando nombre y apellido, y mantiene los campos edad y salario, excluyendo el _id.

6.1.4. $sort

Ordena los documentos según los criterios especificados.

```
db.productos.aggregate([
    { $sort: { precio: -1, nombre: 1 } }
])
```

Este ejemplo ordena los productos por precio en orden descendente y luego por nombre en orden ascendente.

6.1.5. $limit y $skip

Limitan el número de documentos que pasan al siguiente paso del pipeline ($limit) o saltan un número específico de documentos ($skip).

```
db.noticias.aggregate([
    { $sort: { fecha: -1 } },
    { $skip: 10 },
    { $limit: 5 }
])
```

Este ejemplo obtiene las noticias 11 a 15 más recientes.

6.1.6. $unwind

Descompone un campo de array en múltiples documentos.

```
db.pedidos.aggregate([
    { $unwind: "$items" }
])
```

Si un pedido tiene múltiples items, este paso creará un documento separado para cada item del pedido.

6.1.7. $lookup

Realiza un "left outerjoinçon otra colección.

```
db.pedidos.aggregate([
    { $lookup: {
        from: "clientes",
        localField: "cliente_id",
        foreignField: "_id",
        as: "cliente_info"
    }}
])
```

Este ejemplo une la información del cliente a cada pedido.

6.2. Ejemplos avanzados de agregación

6.2.1. Análisis de ventas por mes y categoría

```
db.ventas.aggregate([
    { $match: { fecha: { $gte: new Date("2023-01-01"), $lt: new
    Date("2024-01-01") } } },
    { $group: {
        _id: {
            mes: { $month: "$fecha" },
            categoria: "$categoria"
        },
        totalVentas: { $sum: "$monto" }
    }},
    { $sort: { "_id.mes": 1, "totalVentas": -1 } },
    { $group: {
        _id: "$_id.mes",
        categorias: {
            $push: {
                categoria: "$_id.categoria",
                ventas: "$totalVentas"
            }
        }
    }},
    { $project: {
        mes: "$_id",
        categorias: { $slice: ["$categorias", 3] },
        _id: 0
    }}
])
```

Este pipeline de agregación realiza las siguientes operaciones: 1. Filtra las ventas del año 2023. 2. Agrupa las ventas por mes y categoría. 3. Ordena los resultados por mes y total de ventas. 4. Reagrupa los resultados por mes, creando un array de categorías. 5. Proyecta el resultado final, mostrando solo las 3 categorías con más ventas por mes.

6.2.2. Cálculo de la mediana de precios

Calcular la mediana en MongoDB requiere un enfoque más complejo, ya que no hay un operador directo para ello:

```
db.productos.aggregate([
    { $group: {
        _id: null,
        precios: { $push: "$precio" }
    }},
    { $project: {
        mediana: {
            $let: {
                vars: {
                    preciosOrdenados: { $sort: "$precios" },
                    mitad: { $floor: { $divide: [{ $size: "$precios
    " }, 2] } }
                },
                in: {
                    $cond: {
                        if: { $eq: [{ $mod: [{ $size: "$precios" },
     2] }, 0] },
                        then: { $avg: [
                            { $arrayElemAt: ["$$preciosOrdenados",
    "$$mitad"] },
                            { $arrayElemAt: ["$$preciosOrdenados",
    { $subtract: ["$$mitad", 1] }] }
                        ]},
                        else: { $arrayElemAt: ["$$preciosOrdenados"
    , "$$mitad"] }
                    }
                }
            }
        }
    }}
])
```

Este pipeline: 1. Agrupa todos los precios en un array. 2. Utiliza operadores de expresión para ordenar los precios y calcular la mediana. 3. Maneja tanto el caso de un número par como impar de precios.

6.3. Implementación del Repository

Ahora, implementemos la interfaz del repository y sus implementaciones específicas para MongoDB y MySQL:

```
// Archivo: src/repositories/usuarioRepository.js
class UsuarioRepository {
    async obtenerTodos() {
```

```
        throw new Error("Método no implementado");
    }

    async obtenerPorId(id) {
        throw new Error("Método no implementado");
    }

    async crear(usuario) {
        throw new Error("Método no implementado");
    }

    async actualizar(id, usuario) {
        throw new Error("Método no implementado");
    }

    async eliminar(id) {
        throw new Error("Método no implementado");
    }
}

module.exports = UsuarioRepository;

// Archivo: src/repositories/mongoUsuarioRepository.js
const UsuarioRepository = require('./usuarioRepository');
const { connectMongo } = require('../config/database');
const Usuario = require('../models/usuario');

class MongoUsuarioRepository extends UsuarioRepository {
    constructor() {
        super();
        this.collection = 'usuarios';
    }

    async obtenerTodos() {
        const db = await connectMongo();
        const usuarios = await db.collection(this.collection).find
    ().toArray();
        return usuarios.map(u => new Usuario(u._id.toString(), u.
    nombre, u.edad));
    }

    async obtenerPorId(id) {
        const db = await connectMongo();
        const usuario = await db.collection(this.collection).
    findOne({ _id: ObjectId(id) });
        return usuario ? new Usuario(usuario._id.toString(),
    usuario.nombre, usuario.edad) : null;
    }

    async crear(usuario) {
        const db = await connectMongo();
        const resultado = await db.collection(this.collection).
    insertOne(usuario);
        return new Usuario(resultado.insertedId.toString(), usuario
    .nombre, usuario.edad);
    }
```

```
    async actualizar(id, usuario) {
        const db = await connectMongo();
        await db.collection(this.collection).updateOne(
            { _id: ObjectId(id) },
            { $set: { nombre: usuario.nombre, edad: usuario.edad }
    }
        );
        return new Usuario(id, usuario.nombre, usuario.edad);
    }

    async eliminar(id) {
        const db = await connectMongo();
        await db.collection(this.collection).deleteOne({ _id:
    ObjectId(id) });
    }
}

module.exports = MongoUsuarioRepository;

// Archivo: src/repositories/mysqlUsuarioRepository.js
const UsuarioRepository = require('./usuarioRepository');
const { connectMySQL } = require('../config/database');
const Usuario = require('../models/usuario');

class MySQLUsuarioRepository extends UsuarioRepository {
    async obtenerTodos() {
        const pool = connectMySQL();
        const [rows] = await pool.query('SELECT * FROM usuarios');
        return rows.map(r => new Usuario(r.id, r.nombre, r.edad));
    }

    async obtenerPorId(id) {
        const pool = connectMySQL();
        const [rows] = await pool.query('SELECT * FROM usuarios
    WHERE id = ?', [id]);
        return rows.length ? new Usuario(rows[0].id, rows[0].nombre
    , rows[0].edad) : null;
    }

    async crear(usuario) {
        const pool = connectMySQL();
        const [result] = await pool.query(
            'INSERT INTO usuarios (nombre, edad) VALUES (?, ?)',
            [usuario.nombre, usuario.edad]
        );
        return new Usuario(result.insertId, usuario.nombre, usuario
    .edad);
    }

    async actualizar(id, usuario) {
        const pool = connectMySQL();
        await pool.query(
            'UPDATE usuarios SET nombre = ?, edad = ? WHERE id = ?'
    ,
            [usuario.nombre, usuario.edad, id]
        );
        return new Usuario(id, usuario.nombre, usuario.edad);
```

```
    }

    async eliminar(id) {
        const pool = connectMySQL();
        await pool.query('DELETE FROM usuarios WHERE id = ?', [id])
    ;
    }
}

module.exports = MySQLUsuarioRepository;
```

6.4. Rutas de la API

Implementemos las rutas de nuestra API utilizando el patrón Repository:

```
// Archivo: src/routes/usuarios.js
const express = require('express');
const router = express.Router();
const MongoUsuarioRepository = require('../repositories/
    mongoUsuarioRepository');
const MySQLUsuarioRepository = require('../repositories/
    mysqlUsuarioRepository');
const Usuario = require('../models/usuario');

// Elegir el repositorio basado en una variable de entorno
const usuarioRepository = process.env.DB_TYPE === 'mysql'
    ? new MySQLUsuarioRepository()
    : new MongoUsuarioRepository();

router.get('/', async (req, res) => {
    try {
        const usuarios = await usuarioRepository.obtenerTodos();
        res.json(usuarios);
    } catch (error) {
        res.status(500).json({ error: 'Error al obtener usuarios'
    });
    }
});

router.get('/:id', async (req, res) => {
    try {
        const usuario = await usuarioRepository.obtenerPorId(req.
    params.id);
        if (usuario) {
            res.json(usuario);
        } else {
            res.status(404).json({ error: 'Usuario no encontrado'
    });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error al obtener el usuario'
     });
    }
});

router.post('/', async (req, res) => {
```

```
    try {
        const nuevoUsuario = new Usuario(null, req.body.nombre, req
    .body.edad);
        const usuarioCreado = await usuarioRepository.crear(
    nuevoUsuario);
        res.status(201).json(usuarioCreado);
    } catch (error) {
        res.status(500).json({ error: 'Error al crear el usuario'
    });
    }
});

router.put('/:id', async (req, res) => {
    try {
        const usuarioActualizado = new Usuario(req.params.id, req.
    body.nombre, req.body.edad);
        const resultado = await usuarioRepository.actualizar(req.
    params.id, usuarioActualizado);
        res.json(resultado);
    } catch (error) {
        res.status(500).json({ error: 'Error al actualizar el
    usuario' });
    }
});

router.delete('/:id', async (req, res) => {
    try {
        await usuarioRepository.eliminar(req.params.id);
        res.status(204).send();
    } catch (error) {
        res.status(500).json({ error: 'Error al eliminar el usuario
    ' });
    }
});

module.exports = router;
```

6.5. Servidor principal

Finalmente, configuremos nuestro servidor principal:

```
// Archivo: src/server.js
const express = require('express');
const usuariosRouter = require('./routes/usuarios');
require('dotenv').config();

const app = express();
const port = process.env.PORT || 3000;

app.use(express.json());
app.use('/api/usuarios', usuariosRouter);

app.listen(port, () => {
    console.log(`Servidor corriendo en http://localhost:${port}`);
});
```

7. Pruebas con Postman

Para probar el correcto funcionamiento del patrón Repository implementado, podemos utilizar Postman para realizar solicitudes HTTP a nuestra API. A continuación, se detallan los comandos de Postman para probar cada operación:

7.1. Obtener todos los usuarios

- Método: GET
- URL: http://localhost:3000/api/usuarios
- Headers:
  - Content-Type: application/json

7.2. Obtener un usuario por ID

- Método: GET
- URL: http://localhost:3000/api/usuarios/id
- Headers:
  - Content-Type: application/json

Reemplaza id con el ID real del usuario que deseas obtener.

7.3. Crear un nuevo usuario

- Método: POST
- URL: http://localhost:3000/api/usuarios
- Headers:
  - Content-Type: application/json
- Body (raw JSON):

```
{
    "nombre": "Nuevo Usuario",
    "edad": 30
}
```

7.4. Actualizar un usuario existente

- Método: PUT
- URL: http://localhost:3000/api/usuarios/id
- Headers:
  - Content-Type: application/json
- Body (raw JSON):

```
{
    "nombre": "Usuario Actualizado",
    "edad": 31
}
```

Reemplaza id con el ID real del usuario que deseas actualizar.

7.5. Eliminar un usuario

- Método: DELETE
- URL: http://localhost:3000/api/usuarios/id
- Headers:
  - Content-Type: application/json

Reemplaza id con el ID real del usuario que deseas eliminar.

7.6. Verificación del funcionamiento con diferentes bases de datos

Para verificar que el patrón Repository funciona correctamente tanto con MongoDB como con MySQL, sigue estos pasos:

1. Configura la variable de entorno DB_TYPE en "mongoτ realiza todas las operaciones CRUD anteriores.
2. Cambia la variable de entorno DB_TYPE a "mysqlτ repite las mismas operaciones CRUD.
3. Compara los resultados para asegurarte de que son consistentes independientemente de la base de datos utilizada.

Al realizar estas pruebas, deberías poder confirmar que:

- Las operaciones CRUD funcionan correctamente en ambas bases de datos.
- Los datos se almacenan y recuperan de manera consistente.

- El cambio entre bases de datos es transparente para el cliente de la API.

Recuerda que para cambiar entre bases de datos, solo necesitas modificar la variable de entorno DB_TYPE y reiniciar tu aplicación. Esto demuestra la flexibilidad y poder del patrón Repository en la abstracción de la capa de persistencia.

8. Conclusión

En este documento, hemos explorado en profundidad los conceptos avanzados de MongoDB, incluyendo el modelado de datos, la indexación y optimización de consultas, y las poderosas capacidades de agregación. Además, hemos implementado el patrón Repository para permitir la flexibilidad entre MongoDB y MySQL, demostrando cómo se puede abstraer la lógica de acceso a datos de la lógica de negocio.

La implementación del patrón Repository nos permite:

- Cambiar fácilmente entre diferentes bases de datos sin afectar el resto de la aplicación.
- Mejorar la mantenibilidad y testabilidad del código.
- Centralizar la lógica de acceso a datos, facilitando la implementación de cachés, logging, y otras optimizaciones.

Al combinar las capacidades avanzadas de MongoDB con una arquitectura bien diseñada, hemos creado una aplicación robusta y flexible que puede adaptarse a diferentes requisitos de almacenamiento de datos y escalar eficientemente según las necesidades del proyecto.