Implementación de Patrones de Diseño en el Proyecto

1. Patrón Singleton

1.1 Cambios Realizados

Archivo	Líneas afectadas	Antes	Después
db.ts	1–11	export const db = drizzle(...)	export function getDb(): … con caché
page.txt	1, 7, 14, 21, 28	import { db } from "@/db"	import { getDb } from "@/db"
	8, 15, 22, 29	Uso directo de db	const db = getDb();
layout.txt	—	Sin cambios	Sin cambios

1.2 Motivación

El módulo original exportaba una instancia de Drizzle directamente, lo cual permitía crear nuevas instancias mediante drizzle(...). El patrón Singleton busca:

a) Asegurar la existencia de una única instancia compartida globalmente.
b) Evitar la creación no controlada de nuevas instancias.

1.3 Implementación Actual

1. Se define una variable privada estática (instance) fuera del alcance de los consumidores.
2. Se crea una fábrica pública (getDb()) que:
   - Si instance === undefined → crea la conexión y la almacena.
   - Si instance ya existe → devuelve la misma referencia.
3. El constructor no se exporta, por lo que la única forma legal de obtener una conexión es mediante getDb().

1.4 Cumplimiento del Patrón (GoF, 1995)

Requisito Singleton	Implementación actual
Única instancia compartida	✅ instance se crea una sola vez

Acceso global controlado	✅ getDb()
Creación centralizada y controlada	✅ Constructor oculto
Inicialización perezosa (lazy)	✅ En la primera llamada

1.5 Diagrama UML Simplificado

```
+------------------+
|  <<module>> db  |
+------------------+
| - instance       |
+------------------+
| + getDb(): DB    |
+------------------+
```

1.6 Uso en el Código

```
import { getDb } from '@/db';

const db = getDb();   // siempre retorna la misma instancia
```

1.7 Conclusión

El módulo db.ts pasa de simular un singleton a implementar formalmente el patrón, asegurando que toda la aplicación utilice una única conexión Drizzle controlada y eficiente.

2. Patrón Builder

2.1 Archivo Implementado

src/builders/QueryBuilder.ts

Propósito: Permitir la construcción de consultas SELECT de Drizzle de forma incremental, sin exponer la complejidad interna de SQL.

2.2 Descripción del Patrón

El patrón Builder (Gamma et al., 1995) separa la construcción de un objeto complejo de su representación, permitiendo generar diferentes variantes del mismo proceso de creación.

En este caso:
- El objeto complejo es una consulta SQL.
- La representación varía según los filtros, uniones o límites aplicados.

2.3 Estructura

Elemento	Implementación
Director	No se usa; el cliente cumple este rol.
Builder (interfaz implícita)	QueryBuilder<T>
Producto	PgSelect<T, {}, {}> retornado por build()
Pasos de construcción	where(condition), y futuros orderBy(), limit(), etc.

2.4 Código Fuente Resumido

```
export class QueryBuilder<T extends ReturnType<typeof pgTable>> {
  constructor(private table: T) {}
  private whereConditions: SQL[] = [];

  where(condition: SQL): this {
    this.whereConditions.push(condition);
    return this;
  }

  build() {
    const db = (dbModule as any).getDb();
    const query = db.select().from(this.table as any);
    return this.whereConditions.length
      ? query.where(and(...this.whereConditions))
      : query;
  }
}
```

2.5 Flujo de Trabajo

1. Crear una instancia de QueryBuilder con la tabla objetivo.
2. Encadenar pasos opcionales (where, orderBy, limit, etc.).
3. Llamar a build() para obtener la consulta (PgSelect).
4. Ejecutar con .execute() o enviarla a otra capa.

2.6 Ejemplos de Uso

a) Consulta sin filtros:
```
const todos = await new QueryBuilder(todoTable).build().execute();
```

b) Consultar tareas completadas:
```
const todos = await new QueryBuilder(todoTable)
  .where(eq(todoTable.completed, true))
  .build()
  .execute();
```

c) Filtros dinámicos:
```
const qb = new QueryBuilder(todoTable);
if (completed !== undefined) qb.where(eq(todoTable.completed, completed));
if (search) qb.where(like(todoTable.title, `%${search}%`));

const todos = await qb.build().execute();
```

2.7 Ventajas del Patrón

Ventaja	Descripción
Encapsulación	El cliente no conoce los detalles de construcción SQL.
Fluent API	Sintaxis legible y encadenada: builder.where(...).build().
Extensibilidad	Se pueden agregar nuevos métodos sin alterar código existente.
Reutilización	Permite crear múltiples consultas sobre la misma tabla.
Responsabilidad Única	QueryBuilder solo construye; Drizzle ejecuta.

2.8 Reglas de Uso

1. No ejecutar SQL dentro del builder; solo construirla.
2. No instanciar drizzle fuera de getDb().
3. Sí encadenar todos los pasos requeridos antes de build().

2.9 Extensiones Planeadas

```
qb.where(...)
  .orderBy(...)
  .limit(...)
  .offset(...)
  .innerJoin(...);
```

Estas extensiones se agregarán sin afectar la compatibilidad existente (Principio Abierto/Cerrado).

2.10 Conclusión

QueryBuilder representa una implementación pura del patrón Builder, ya que permite construir consultas SQL paso a paso de manera flexible, extensible y coherente con los principios de diseño orientado a objetos. Su uso conjunto con el Singleton de base de datos garantiza una arquitectura modular, limpia y fácil de mantener.