

# Clean Architecture en Frontend

Estructura Robusta con Next.js y TypeScript

Computación en Red 3 • Semana 11 • Universidad Icesi

# El Dilema: Políticas vs. Detalles

## Las Políticas (El Negocio)

### Reglas Esenciales

#### Lo que el sistema hace y valida

Definen la lógica central de la organización: cálculo de totales, validación de un ejercicio o restricciones de acceso.

### Alta Estabilidad

#### Cambian poco y despacio

No dependen de qué librería de UI esté de moda ni de qué base de datos use el backend.

## Los Detalles (Las Herramientas)

### Mecanismos de Entrega

#### Cómo se muestra y se transporta

React, Next.js, Tailwind CSS, clientes HTTP como `fetch` o bases de datos como PostgreSQL.

### Riesgo Arquitectónico

#### Acoplamiento peligroso

El problema surge cuando las políticas dependen de los detalles: cada cambio externo obliga a reescribir el negocio.

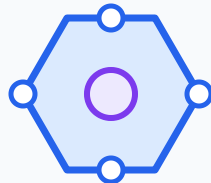
# Origen y Evolución de las Arquitecturas Limpias

Una idea que se repite: el dominio al centro

**2005**

**HEXAGONAL**

Alistair Cockburn

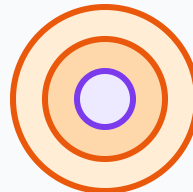


Puertos y adaptadores

**2008**

**ONION**

Jeffrey Palermo

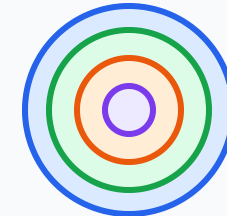


Capas concéntricas

**2012**

**CLEAN**

Robert C. Martin



4 círculos + regla de dependencia

## La Regla de Dependencia



### La Regla Central

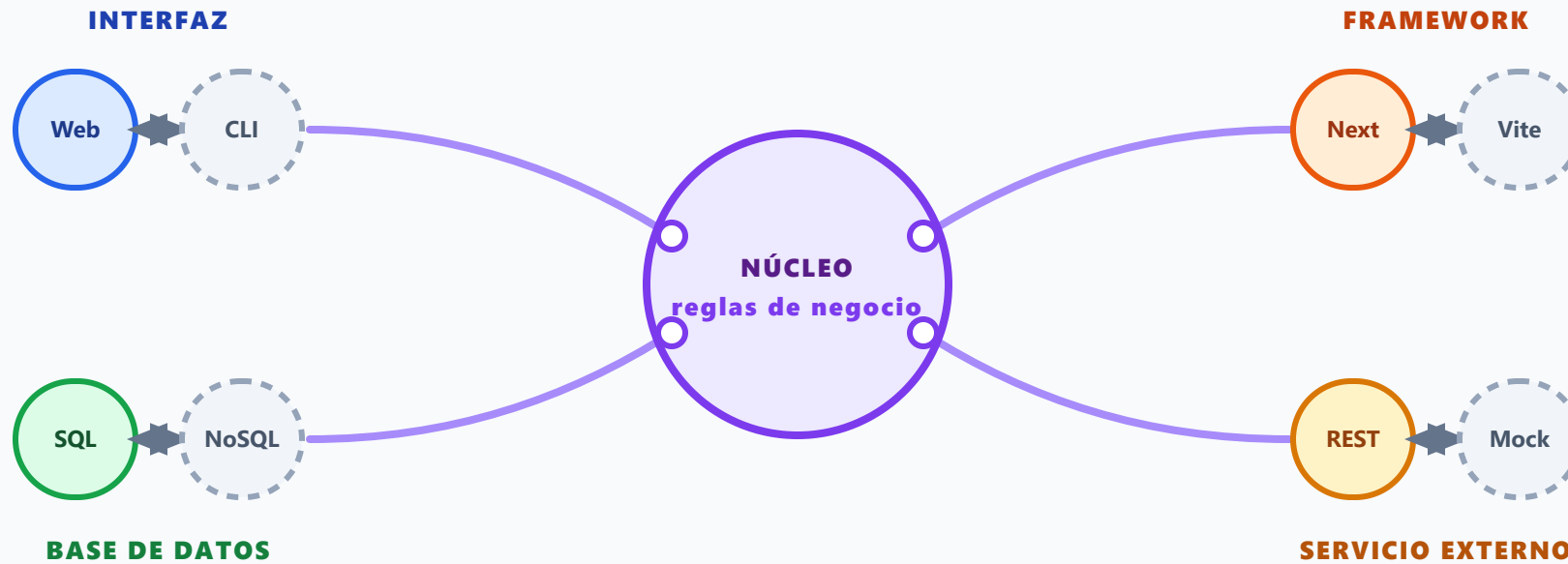
Las dependencias solo apuntan hacia adentro

El código de un círculo interno **nunca puede nombrar ni importar** nada declarado en un círculo externo.

- **Aislamiento total:** El núcleo no sabe si se renderiza en Next.js, en React Native o en Node.js.
- **Inversión de Dependencias:** Si el dominio necesita guardar datos, declara una interfaz abstracta; la infraestructura externa la implementa.
- **Flujo de control vs. Dependencia:** La ejecución viaja hacia afuera, pero la dependencia de código fuente apunta siempre hacia el centro.

## Independencia: El Núcleo Protegido

Lo que Clean Architecture permite cambiar sin tocar el núcleo

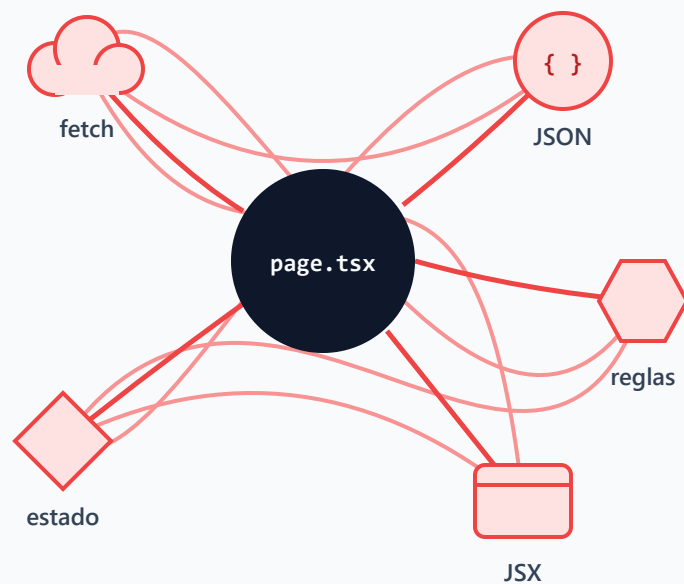


## El Antipatrón del Componente Monolítico

## Caos Monolítico vs. Separación en Capas

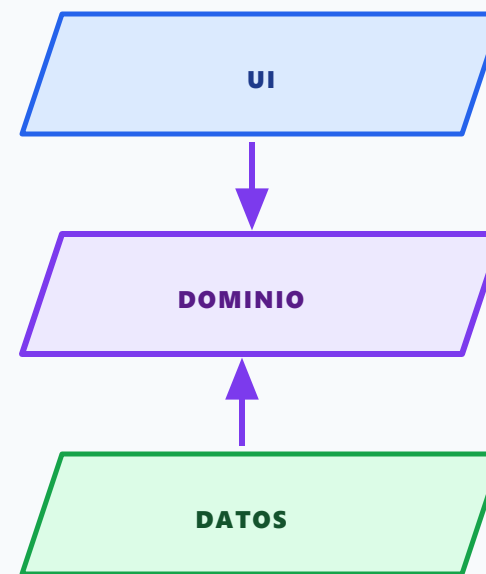
Todo mezclado vs. separado por capas

**ANTES**



vs

**DESPUÉS**



# Anatomía del Fat Component

## 5 Responsabilidades en un Solo Archivo

### 1. Red y Transporte

Conoce URLs fijas ( `/api/exercises` ) y verbos HTTP.

### 2. Acoplamiento al JSON

Usa `snake_case` del backend directamente en la vista.

### 3. Reglas en el Botón

Valida longitud del nombre e invariantes en `handleCreate` .

### 4. Estado Reactivo y JSX

Mezcla `useState` con decenas de clases de Tailwind.

## El Antipatrón en Código Simplificado

```
// src/app/exercises/page.tsx (Antipatrón)
export default function ExercisePage() {
  const [list, setList] = useState([]);

  // 1. Acceso a red acoplado a la vista
  useEffect(() => {
    fetch('/api/exercises')
      .then(r => r.json())
      .then(setList);
  }, []);

  // 2. Regla de negocio en el botón
  const handleCreate = () => {
    if (name.trim().length < 3) return;
    // 3. Payload formateado a mano para el backend
    fetch('/api/exercises', { method: 'POST', ... });
  };

  return <div>{/* 4. JSX con estilos */}</div>;
}
```

**VISTA (JSX)**

Presenta datos al usuario

**CASO DE USO**

Orquesta la intención

**REPOSITORIO**

Recupera y persiste

**MAPPER**

Traduce formatos incompatibles

## Principio de Responsabilidad Única

**Una Sola Razón para Cambiar**

**Cada módulo cumple un solo rol especializado**

Si una pantalla contiene red, validaciones y diseño, tiene **múltiples razones distintas para romperse**.

- **La Vista (JSX):** Solo cambia si el diseñador actualiza la interfaz visual.
- **El Caso de Uso:** Solo cambia si las reglas del negocio o flujos de usuario evolucionan.
- **El Repositorio:** Solo cambia si la fuente técnica de almacenamiento se modifica.
- **El Mapper:** Solo cambia si el contrato JSON del servidor se actualiza.

# Inversión de Dependencias (DIP)

## Sin DIP (Acoplamiento Rígido)

### Dependencia Directa

#### La vista importa la implementación

El componente React conoce directamente la clase concreta de transporte o la URL física.

```
// Acoplado directamente a la red
import { ApiClient } from './api';

function Screen() {
  const api = new ApiClient();
  // Si cambia ApiClient, se rompe Screen
}
```

## Con DIP (Contratos Abstractos)

### Inversión Lograda

#### El centro define la interfaz

El caso de uso solo conoce la `interface`; la infraestructura externa la implementa.

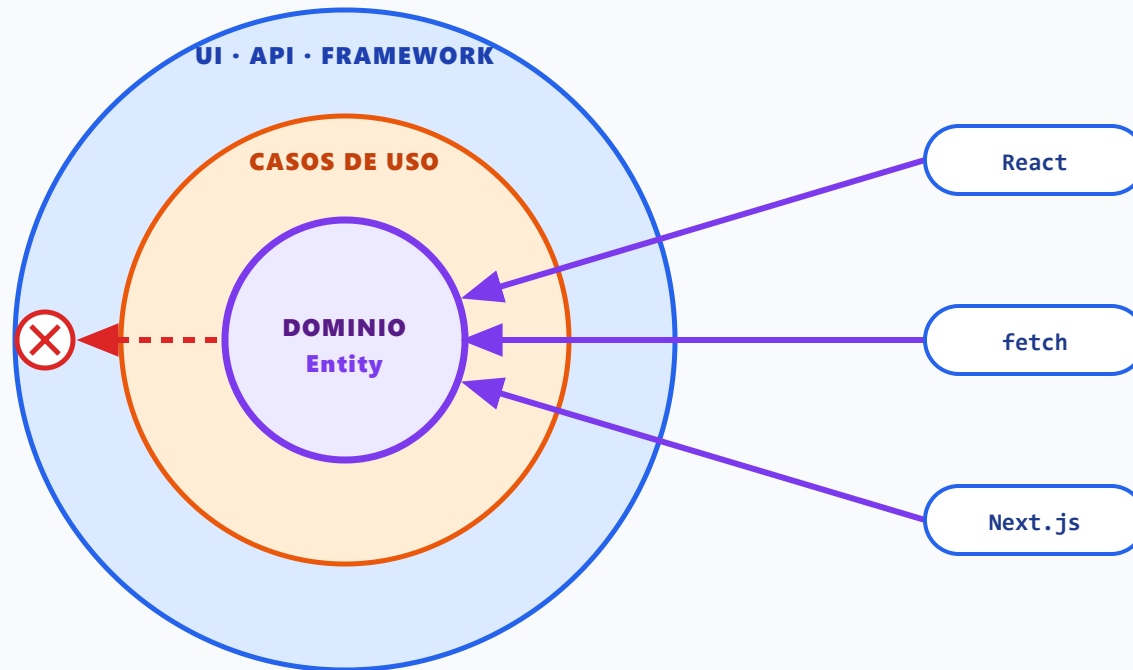
```
// El Dominio define el contrato QUÉ
interface ExerciseRepository {
  getAll(): Promise<Exercise[]>;
}

// El Caso de Uso solo conoce la interface
class GetExercisesUseCase {
  constructor(private repo: ExerciseRepository) {}
}
```

## Estructura por Capas y Vertical Slices

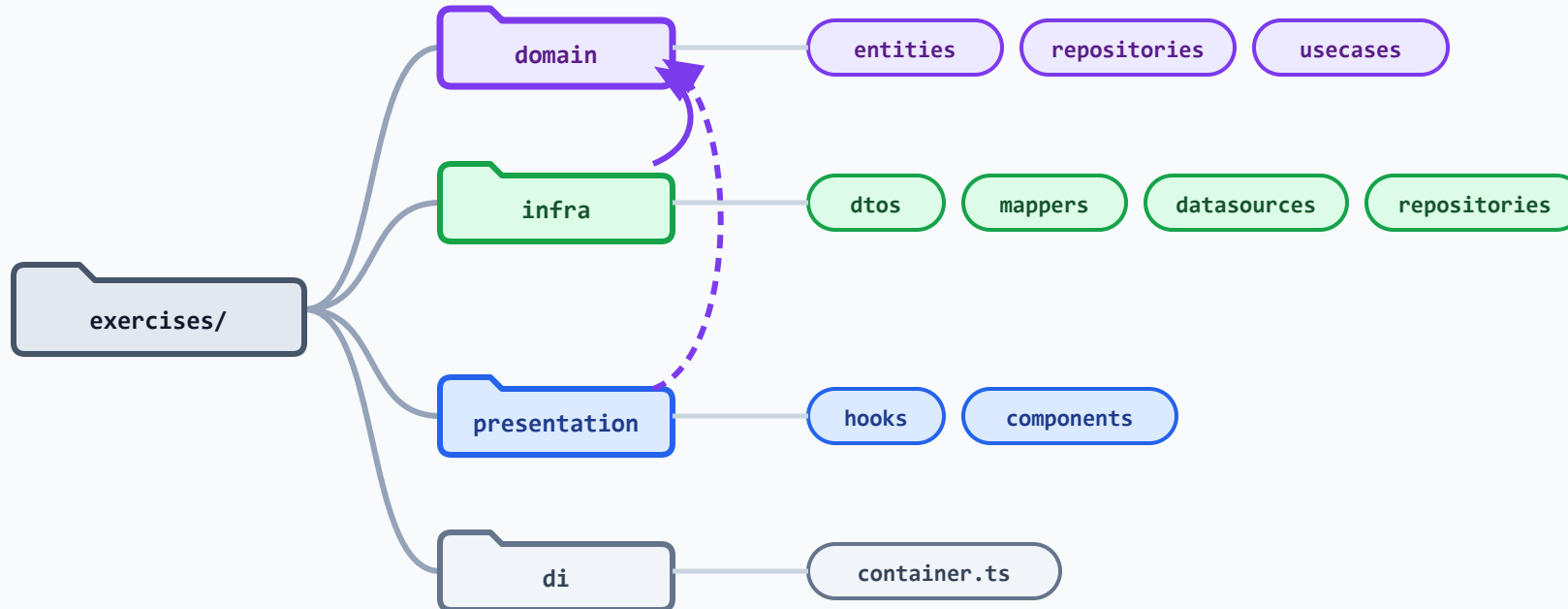
## Círculos Concéntricos de Clean Architecture

La regla de dependencia: las flechas solo entran



## Organización por Features en Next.js

Una feature, tres capas hacia el dominio



# Anatomía del Módulo por Feature

## 1. Dominio e Infraestructura

`features/exercises/domain/`

### TypeScript 100% Puro

`entities/` (modelos inmutables), `repositories/` (contratos abstractos) y `usecases/` (interactors).

`features/exercises/infrastructure/`

### Adaptadores Externos

`dtos/` (JSON de la red), `mappers/` (conversores) y `datasources/` (Mock o NestJS).

## 2. Contenedor y Presentación

`features/exercises/di/`

### Composition Root

`exercise.container.ts` : ensambla dependencias e inyecta la fuente de datos física.

`features/exercises/presentation/`

### UI Desacoplada (React / Next.js)

`hooks/` (presenter con estado local reactivo) y `components/` (tarjetas y formularios puros).

## Capa de Dominio: El Núcleo de Negocio

# Entidades de Dominio Inmutables

## Reglas Protegidas en el Constructor

### TypeScript Puro

#### Cero Dependencias Externas

El dominio no importa React, ni hooks, ni `next/navigation`, ni librerías de terceros.

- **Inmutabilidad estricta:** Todas las propiedades son `readonly` para evitar mutaciones accidentales.
- **Validación de Invariantes:** El constructor arroja excepciones si los datos son inválidos antes de que contaminen el sistema.

## Entidad Simplificada en Código

```
// src/features/exercises/domain/entities/exercise.ts
export type MuscleGroup = 'CHEST' | 'BACK' | 'LEGS';

export class Exercise {
  readonly id: string;
  readonly name: string;
  readonly muscleGroup: MuscleGroup;

  constructor(props: { id: string; name: string; muscleGroup: MuscleGroup }) {
    if (!props.name || props.name.trim().length < 3) {
      throw new Error('Nombre inválido: mínimo 3 letras');
    }
    this.id = props.id;
    this.name = props.name;
    this.muscleGroup = props.muscleGroup;
  }
}
```

# Contratos de Repositorio

## El Dominio Declara el QUÉ

### Abstracción Pura

#### Una Interface de TypeScript

El dominio declara qué operaciones de persistencia necesita sin especificar si vendrán de una base SQL, una API REST o un mock.

- **Independencia del transporte:** No contiene `fetch`, ni headers, ni endpoints.
- **Tipos limpios:** Retorna y recibe siempre Entidades de Dominio, nunca DTOs de red.

## El Contrato en Código

```
// domain/repositories/exercise.repository.ts
import { Exercise, MuscleGroup } from '../entities/exercise';

export interface CreateExerciseParams {
  name: string;
  muscleGroup: MuscleGroup;
}

export interface ExerciseRepository {
  // Contrato abstracto del dominio
  getAll(): Promise<Exercise[]>;
  create(params: CreateExerciseParams): Promise<Exercise>;
}
```

## Casos de Uso (Interactors)

### Una Intención, Un Método

#### Responsabilidad Única

#### Método `execute()` Especializado

Cada caso de uso representa una acción concreta del usuario (ej. `GetExercises` o `CreateExercise`).

- **Orquestación:** Coordina entidades y contratos de repositorio.
- **Reglas de aplicación:** Aplica lógica como ordenamiento alfabético o filtros antes de entregar datos a la UI.

### Caso de Uso en Código Simplificado

```
// domain/usecases/get-exercises.usecase.ts
import { ExerciseRepository } from '../repositories/exercise.repository';

export class GetExercisesUseCase {
  constructor(private readonly repo: ExerciseRepository) {}

  async execute() {
    const items = await this.repo.getAll();
    // Regla de aplicación: ordenar alfabéticamente
    return items.sort((a, b) => a.name.localeCompare(b.name));
  }
}
```



**TEST PASSED**

Tiempo: ~12 ms

0 dependencias de navegador

## Pruebas Unitarias Instantáneas

### Pruebas en Milisegundos

**Sin navegador, sin DOM, sin backend**

Al no tener acoplamiento con la interfaz gráfica, la lógica de negocio se verifica en Node.js puro usando un repositorio simulado (Mock).

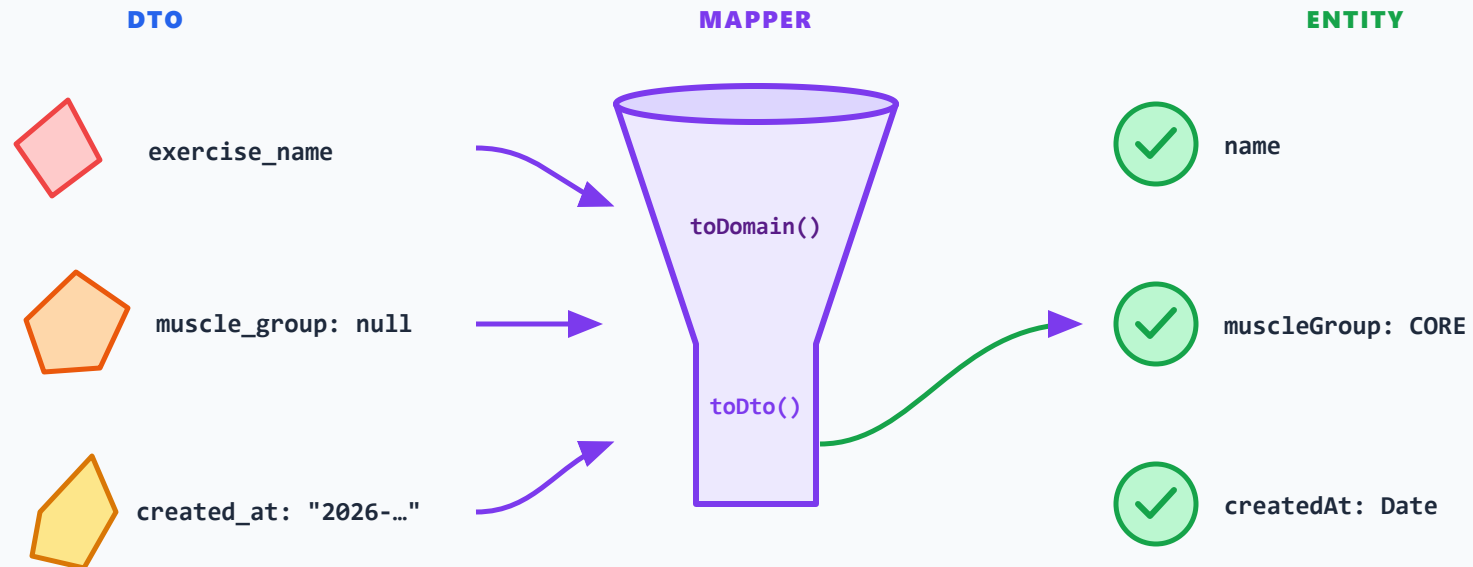
```
// create-exercise.usecase.test.ts
it('falla si el nombre tiene menos de 3 caracteres', async () => {
  const mockRepo = { getAll: jest.fn(), create: jest.fn() };
  const useCase = new CreateExerciseUseCase(mockRepo);

  await expect(useCase.execute({ name: 'Ab' }))
    .rejects.toThrow();
  expect(mockRepo.create).not.toHaveBeenCalled();
});
```

## Capa de Infraestructura: Adaptadores y Acceso a Datos

## La Aduana Defensiva del Mapper

### El Mapper: del JSON irregular a la Entity limpia



# DTO vs. Entidad en Código

## 1. DTO Crudo de la API

### Formato de Red

#### Refleja fielmente el JSON

Usa `snake_case` y tipos no normalizados propios de la API externa.

```
export interface ExerciseResponseDto {  
  id: string;  
  exercise_name: string;  
  muscle_group: string | null;  
}
```

## 2. Mapper Defensivo

### Transformador Seguro

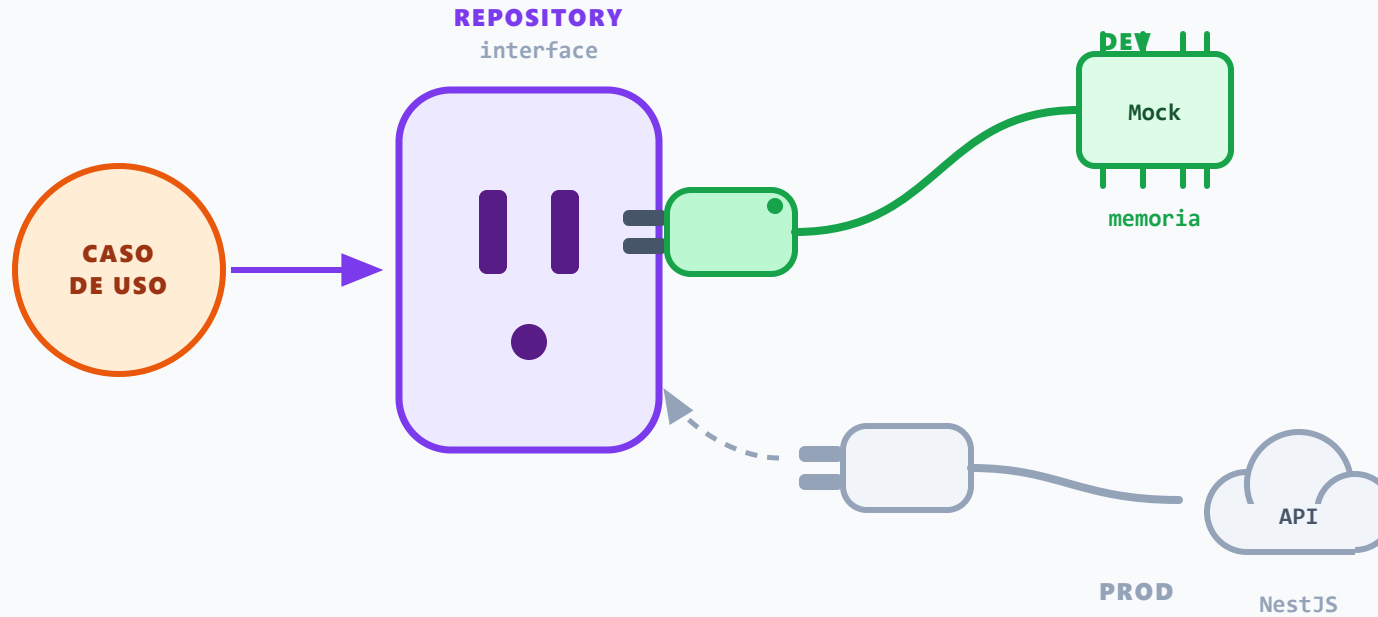
#### Normaliza y crea la Entidad

Si el backend renombra un campo, solo se ajusta este archivo.

```
export class ExerciseMapper {  
  static toDomain(dto: ExerciseResponseDto): Exercise {  
    return new Exercise({  
      id: dto.id,  
      name: dto.exercise_name,  
      muscleGroup: (dto.muscle_group ?? 'CHEST') as MuscleGroup,  
    });  
  }  
}
```

## Datasources Desacoplados: Socket y Plugs

**Repository: un enchufe, varias fuentes de datos**



## El Repositorio Concreto

### Coordinador de Infraestructura

#### Cumple el Contrato

#### ExerciseRepositoryImpl

Ensambla el flujo técnico: invoca la fuente de datos física y traduce los resultados con el mapper.

- **Inyección del Datasource:** Recibe `ExerciseDatasource` por constructor; no usa `new` internamente.
- **Frontera segura:** Devuelve solo Entidades limpias al Dominio.

### Implementación en Código

```
// infrastructure/repositories/exercise.repository.impl.ts
export class ExerciseRepositoryImpl implements ExerciseRepository {
  constructor(private readonly ds: ExerciseDatasource) {}

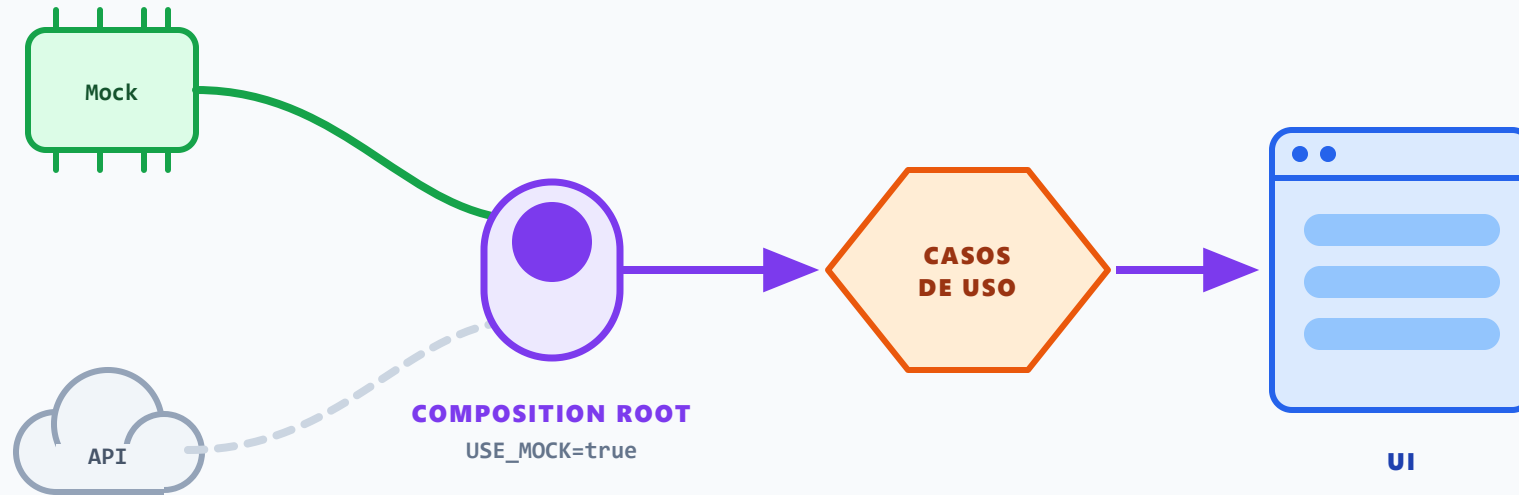
  async getAll(): Promise<Exercise[]> {
    const dtos = await this.ds.fetchExercises();
    return dtos.map(ExerciseMapper.toDomain);
  }

  async create(params: CreateExerciseParams) {
    const dto = await this.ds.createExercise(params);
    return ExerciseMapper.toDomain(dto);
  }
}
```

## Inyección de Dependencias y Conexión con Next.js

## Inyección de Dependencias: El Interruptor

**Composition Root: un interruptor decide la fuente**



# El Composition Root

## Ensamblador Funcional

### Un Solo Punto Central

#### `exercise.container.ts`

Decide qué fuente instanciar según la variable de entorno, inyecta el repositorio y expone los casos de uso listos para usar.

- **Alternancia sin esfuerzo:** Cambiar de Mock a API real es cambiar un valor en el archivo `.env`.
- **Cero librerías pesadas:** Tipado estricto con TypeScript estándar.

## El Contenedor en Código

```
// features/exercises/di/exercise.container.ts
const isMock = process.env.NEXT_PUBLIC_USE MOCK === 'true';

// 1. Selección dinámica de fuente
const datasource = isMock
  ? new ExerciseMockDatasource()
  : new ExerciseApiDatasource('http://localhost:3000/api');

// 2. Ensamblaje e inyección
const repository = new ExerciseRepositoryImpl(datasource);

// 3. Exportación de Casos de Uso listos para la UI
export const exerciseContainer = {
  getExercises: new GetExercisesUseCase(repository),
  createExercise: new CreateExerciseUseCase(repository),
};
```

# El Custom Hook como Presenter

## Presenter / ViewModel en React

### useExercises

#### Puente entre Dominio y Vista

Orquesta la ejecución de los casos de uso y gestiona el estado reactivo ( `loading` , `exercises` , `error` ).

### ¿Por qué useState y no Redux aún?

Para dominar primero Clean Architecture. Cuando el estado sea global, solo cambiaremos este hook sin tocar el Dominio.

## El Hook en Código Simplificado

```
// presentation/hooks/use-exercises.ts
export function useExercises() {
  const [exercises, setExercises] = useState<Exercise[]>([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    // Invoca el Caso de Uso a través del contenedor
    exerciseContainer.getExercises.execute()
      .then(setExercises)
      .finally(() => setLoading(false));
  }, []);

  return { exercises, loading };
}
```

# Vistas Puras y Página en Next.js

## Componentes Puros (Dumb)

### ExerciseCard

Recibe la entidad por props y renderiza diseño visual puro con Tailwind CSS.

```
export function ExerciseCard({ exercise }: { exercise: Exercise }) {  
  return (  
    <div className="p-4 border rounded shadow">  
      <h3>{exercise.name}</h3>  
      <span>{exercise.muscleGroup}</span>  
    </div>  
  );  
}
```

## Página Declarativa (< 20 líneas)

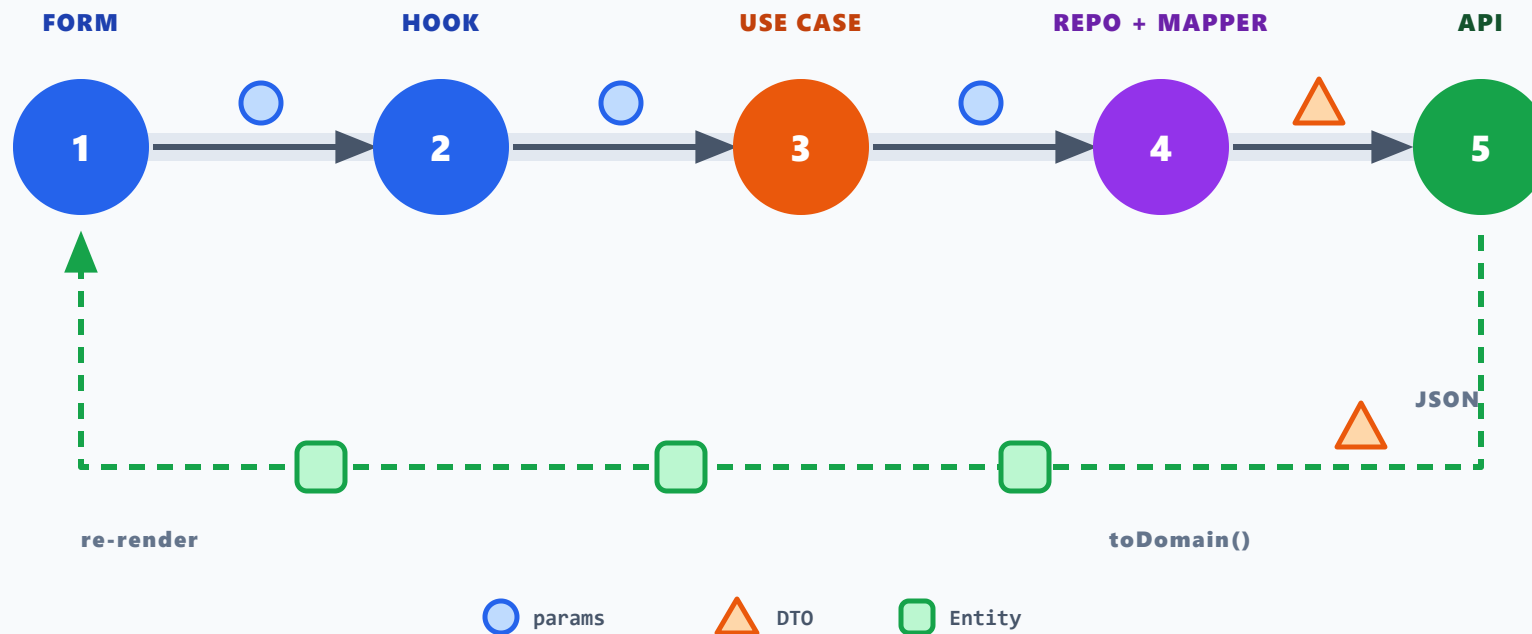
### src/app/exercises/page.tsx

La página de Next.js solo ensambla el Hook con los componentes visuales.

```
'use client';  
export default function ExercisesPage() {  
  const { exercises, loading } = useExercises();  
  
  if (loading) return <p>Cargando...</p>;  
  
  return (  
    <main className="grid grid-cols-2 gap-4 p-8">  
      {exercises.map(e => <ExerciseCard key={e.id} exercise={e} />)}  
    </main>  
  );  
}
```

## Tubería de Ejecución: Del Click al Backend

Un clic: ida al backend y vuelta a la pantalla



## Criterio Práctico: ¿Cuándo Aplicarlo?

### Cuándo SÍ Aplicar Clean Architecture

#### Sistemas en Crecimiento

##### Software que perdura en el tiempo

Aplicaciones empresariales que van a recibir nuevas funcionalidades y soporte continuo durante meses o años.

#### Reglas de Negocio Reales

##### Lógica más allá del simple CRUD

Validaciones complejas, cálculos, múltiples orígenes de datos o necesidad estricta de pruebas automatizadas.

### Cuándo NO Aplicarlo (Evitar Sobreingeniería)

#### Prototipos Desechables

##### MVPs de Fin de Semana

Para validar una hipótesis o en una hackathon, escribir todas las capas añade fricción innecesaria.

#### CRUDs Triviales

##### Páginas de Solo Lectura Simple

Si una pantalla solo muestra una tabla estática sin lógica de negocio, un Server Component directo es suficiente.



## SISTEMA MANTENIBLE

Desacoplado y Testeable

**Next.js + TypeScript Limpio**

# Los 4 Pilares en Frontend

### 1. Regla de Dependencia

#### Dependencias hacia adentro

El código fuente siempre apunta al núcleo de mayor nivel de abstracción.

### 2. Dominio Puro

#### TypeScript 100% independiente

Entidades inmutables y casos de uso sin librerías de UI ni transporte de red.

### 3. Aduana Defensiva

#### Mappers y DTOs

Aíslan el sistema frente a cambios de estructura o nomenclatura del backend.

### 4. Sustituibilidad

#### Composition Root

Permite alternar entre Mock y API real tocando una sola variable de entorno.