UNIVERSIDAD NACIONAL DE SAN JUAN

FACULTAD DE CIENCIAS EXACTAS FÍSICAS Y NATURALES

DEPARTAMENTO DE INFORMÁTICA

LICENCIATURA EN SISTEMAS DE INFORMACIÓN

ANÁLISIS EVOLUTIVO DE ARQUITECTURAS DE SOFTWARE EN SISTEMAS EMPRESARIALES

Tipo de trabajo: Informe Técnico de Práctica Profesional

AUTOR
Sebastián Jesús Ariel Avila

ASESOR
Laura Aballay

COASESOR
Juan Aranda

SAN JUAN

2026

Resumen

Este trabajo analiza la evolución de arquitecturas de software y patrones de diseño en sistemas empresariales desarrollados en el ámbito de la práctica profesional, documentando el proceso de perfeccionamiento desde arquitecturas iniciales hasta el modelo de desarrollo actualmente implementado. Se combina revisión bibliográfica, análisis de código fuente de sistemas desarrollados en diferentes períodos y documentación de la progresión arquitectónica. Se presentan estilos arquitectónicos (MVC tradicional, arquitectura en capas, orientada a servicios) y patrones de diseño (Repositorio, Capa de Servicio, Especificación, Inyección de Dependencia, entre otros), mostrando cómo en cada desarrollo se incorporaron mejoras concretas y qué valor agregado se obtuvo (reutilización, testabilidad, mantenibilidad, migración gradual). Se incluyen diagramas de arquitectura por desarrollo y ejemplos de implementación que ilustran el paso de una arquitectura a otra. Como resultado se obtiene un documento que sistematiza la evolución arquitectónica, identifica el valor agregado en cada mejora, documenta el proceso de perfeccionamiento y presenta el modelo arquitectónico alcanzado actualmente, con sus características y justificación técnica.

Palabras clave: arquitectura de software, patrones de diseño, evolución de sistemas, Repositorio, Capa de Servicio, Especificación.

Abstract

This work analyses the evolution of software architectures and design patterns in enterprise systems developed in a professional practice context, documenting the improvement process from initial architectures to the development model currently in place. It combines literature review, source code analysis of systems developed in different periods, and documentation of the architectural progression. Architectural styles (traditional MVC, layered architecture, service-oriented) and design patterns (Repository, Service Layer, Specification, Dependency Injection, among others) are presented, showing how each development incorporated concrete improvements and what added value was obtained (reuse, testability, maintainability, gradual migration).

Architecture diagrams per development and implementation examples illustrate the transition from one architecture to another. The result is a document that systematises the architectural evolution, identifies the value added at each improvement, documents the improvement process, and presents the current architectural model with its characteristics and technical justification.

Keywords: software architecture, design patterns, system evolution, Repository, Service Layer, Specification.

Índice

1. Introducción

La arquitectura de software constituye el conjunto de decisiones estructurales fundamentales que determinan la organización, evolución y calidad de un sistema. En contextos empresariales dinámicos, dichas decisiones no son estáticas, sino que emergen de procesos iterativos de aprendizaje técnico y refinamiento metodológico.

El presente Informe Técnico de Práctica Profesional analiza de manera sistemática la evolución de arquitecturas y patrones de diseño aplicados en distintos sistemas desarrollados en el ámbito profesional. A partir de un enfoque comparativo y cronológico, se documenta el tránsito desde implementaciones basadas en MVC tradicional - con lógica de negocio concentrada en controladores - hacia un modelo arquitectónico en capas consolidado que incorpora Repositorio, Capa de Servicio, Especificación e Inversión de Dependencias.

El aporte central del trabajo radica en la formalización de este proceso evolutivo, identificando el valor agregado obtenido en cada transición arquitectónica y justificando técnicamente el modelo actualmente adoptado como estándar.

2. Marco teórico

2.1 Ingeniería de software y evolución de sistemas

La ingeniería de software es la disciplina que aborda el desarrollo, operación y mantenimiento de sistemas de software de manera sistemática y controlada, con el objetivo de obtener productos de calidad que satisfagan los requerimientos establecidos (Sommerville, 2011). A medida que los sistemas crecen en funcionalidad y complejidad, surge la necesidad de adaptarlos a nuevos contextos tecnológicos y organizacionales, dando lugar al proceso de evolución del software.

El software no es un producto estático, sino un sistema en constante cambio. En este sentido, la mantenibilidad se convierte en uno de los atributos de calidad más relevantes, ya que determina la facilidad con la que un sistema puede ser modificado para corregir errores, mejorar su desempeño o incorporar nuevas funcionalidades

sin comprometer su estabilidad (Pressman & Maxim, 2021). La ausencia de una estructura adecuada incrementa el acoplamiento entre componentes, dificultando la comprensión del sistema y aumentando el costo de los cambios.

Dentro de este contexto, la definición de una arquitectura adecuada y la aplicación de patrones de diseño constituyen mecanismos fundamentales para controlar la complejidad y permitir la evolución progresiva de los sistemas.

2.2 Arquitectura de software

La arquitectura de software define la estructura de alto nivel de un sistema, incluyendo sus componentes, las relaciones entre ellos y los principios que guían su diseño y evolución (Bass et al., 2012). Las decisiones arquitectónicas tienen un impacto directo en los atributos de calidad del sistema, tales como la mantenibilidad, escalabilidad, testabilidad y reutilización.

Una arquitectura bien definida permite organizar el sistema en unidades con responsabilidades claras, reducir el acoplamiento entre componentes y facilitar la incorporación de cambios sin afectar el funcionamiento global. En sistemas empresariales, donde los requerimientos evolucionan de manera constante, la arquitectura cumple un rol central al permitir la adaptación progresiva a nuevas necesidades funcionales y tecnológicas.

2.3 Estilos arquitectónicos

2.3.1 Modelo Vista Controlador (MVC)

El modelo MVC propone la separación del sistema en tres componentes principales: modelo, vista y controlador, permitiendo una organización más clara de las responsabilidades y favoreciendo la reutilización de código (Gamma et al., 1995).

Este estilo es ampliamente utilizado en frameworks como Laravel, aunque en implementaciones tradicionales la lógica de negocio suele concentrarse en los controladores, lo que genera dificultades de mantenimiento a medida que el sistema crece.

2.3.2 Arquitectura en capas

La arquitectura en capas organiza el sistema en niveles horizontales con responsabilidades bien definidas, donde cada capa ofrece servicios a la capa superior y depende únicamente de la inferior. Este enfoque favorece la separación de responsabilidades y la posibilidad de sustituir implementaciones sin afectar el resto del sistema (Fowler, 2002).

2.3.3 Arquitectura cliente-servidor

El modelo cliente-servidor establece una separación entre el frontend y el backend, comunicados a través de servicios expuestos mediante HTTP y generalmente basados en arquitecturas REST. Este enfoque permite el desarrollo de aplicaciones de una sola página (SPA), donde la interfaz de usuario evoluciona de forma independiente del servidor.

Tecnologías como React permiten implementar este tipo de soluciones desacopladas, facilitando la escalabilidad y la evolución independiente de los componentes.

En conjunto, los estilos arquitectónicos y patrones de diseño analizados constituyen mecanismos para controlar la complejidad estructural del software. La aplicación sistemática de dichos principios permite reducir el acoplamiento, mejorar la cohesión y facilitar la evolución progresiva de los sistemas. En este sentido, la arquitectura no se concibe como una decisión puntual, sino como un proceso continuo de refinamiento orientado a preservar atributos de calidad a lo largo del tiempo.

2.4 Patrones de diseño

Los patrones de diseño constituyen soluciones probadas a problemas recurrentes en el desarrollo de software y permiten mejorar la organización del código, reducir el acoplamiento y aumentar la mantenibilidad del sistema (Gamma et al., 1995).

2.4.1 Repository

El patrón Repository abstrae el acceso a los datos mediante interfaces que representan colecciones de objetos de dominio, desacoplando la lógica de negocio

de los mecanismos de persistencia (Fowler, 2002). Esto implica que la lógica de negocio depende de abstracciones y no de implementaciones concretas, permitiendo sustituir el mecanismo de persistencia - por ejemplo, cambiar de un motor relacional a uno NoSQL- sin modificar las capas superiores del sistema.

2.4.2 Service Layer

El patrón Service Layer encapsula la lógica de negocio en servicios que coordinan la interacción entre repositorios y otros componentes del sistema, manteniendo los controladores enfocados en la gestión de las solicitudes (Fowler, 2002).

2.4.3 Specification

El patrón Specification permite encapsular criterios de consulta en objetos reutilizables y combinables, mejorando la expresividad y la reutilización de las reglas de negocio (Evans, 2003).

2.4.4 Inyección de dependencias

La inyección de dependencias es una técnica que permite proporcionar a una clase sus dependencias desde el exterior en lugar de instanciarlas internamente, reduciendo el acoplamiento entre componentes y mejorando la testabilidad del sistema (Martin, 2008). Frameworks como Laravel incorporan contenedores de inversión de control que permiten gestionar estas dependencias de manera automática.

3. Metodología

El trabajo se desarrolló mediante:

1. **Revisión bibliográfica** sobre estilos arquitectónicos, patrones de diseño y buenas prácticas en arquitectura de software.
2. **Análisis de código fuente** de los sistemas seleccionados, identificando la organización del código, las capas arquitectónicas, los patrones de diseño y las tecnologías utilizadas.

3. **Relevamiento documental** de cada sistema mediante documentación técnica que describe la arquitectura full-stack (frontend y backend), los patrones identificados, las tecnologías, así como diagramas y ejemplos de código.
4. **Análisis comparativo** entre sistemas, ordenados según una secuencia de análisis que refleja el orden evolutivo y cronológico de los desarrollos, para identificar el paso de una arquitectura a otra y el valor agregado en cada mejora.
5. **Sistematización** de la evolución por desarrollo, con diagramas y ejemplos de implementación que ilustran cada transición.
6. **Redacción** de la monografía.

4. Desarrollo del trabajo

4.1 Actividades realizadas

Las actividades se organizaron en fases alineadas con el plan de trabajo del resumen presentado:

- **Fase 1 — Revisión bibliográfica y planificación:** revisión de estilos arquitectónicos y patrones; definición del alcance; selección de sistemas a estudiar.
- **Fase 2 — Relevamiento de arquitecturas y patrones:** análisis de código fuente; identificación de arquitecturas y patrones; recolección de la documentación técnica existente por sistema, incluyendo diagramas y fragmentos de implementación.
- **Fase 3 — Análisis de evolución y perfeccionamiento:** comparación entre sistemas según la secuencia de análisis adoptada; identificación de cómo se pasó de una arquitectura a otra; valor agregado en cada mejora; clasificación de patrones por desarrollo.
- **Fase 4 — Modelado arquitectónico:** elaboración de diagramas de arquitectura (full-stack) por caso de estudio; documentación técnica en los informes.

- **Fase 5 — Documentación del modelo actual:** sistematización del proceso evolutivo; descripción del modelo vigente; elaboración de lineamientos generales; inclusión de ejemplos de implementación.
- **Fase 6 — Redacción y entrega:** ajustes finales y redacción del informe completo (monografía).

4.2 Relevamiento de sistemas y arquitecturas

Se relevaron los sistemas según una **secuencia de análisis** establecida que refleja el orden evolutivo y cronológico de los casos de estudio, con el fin de documentar la evolución de las arquitecturas y de los patrones de diseño. Cada caso de estudio fue documentado de forma independiente. La secuencia adoptada se puede ver en la tabla 1:

Casos de estudio	Descripción breve	Valor agregado principal
Caso de Estudio 1	Backend CodeIgniter 3 (MVC tradicional), aplicación monolítica	Punto de partida: MVC básico sin capas intermedias
Caso de Estudio 2	Backend Laravel 11 (API), frontend SPA (Vite/React), multi-tenancy (Stancl), roles/permisos (Spatie)	Inicio de la secuencia: cliente-servidor, multi-tenancy, RBAC (Control de Acceso Basado en Roles)
Caso de Estudio 3	Backend en capas (Repository, Service), frontend SPA (Vite/React)	Testabilidad, mantenibilidad, integraciones avanzadas
Caso de Estudio 4	Backend Laravel 11 en capas (Controlador → Servicio → Repositorio → Especificación → Modelo), frontend SPA (Vite/React)	Continuidad del estándar; modelo ideal implementado desde el inicio
Caso de Estudio 5	Backend Laravel 12 en capas completo con Patrón de Especificación, Inyección de Dependencia, frontend SPA por módulos (servicios, custom hooks, Zustand, React Query)	Modelo ideal: reutilización, Patrón de Especificación, escalabilidad

En todos los casos se documentó la **arquitectura full-stack**: capa de presentación (frontend) y capa(s) de backend (APIs), con descripción de tecnologías, patrones, diagramas y flujos de comunicación (HTTP/REST).

4.3 Evolución por caso de estudio: de MVC básico al modelo en capas

El análisis de los casos de estudio en el marco de la práctica profesional permite identificar una evolución progresiva en la arquitectura adoptada. Esta evolución no responde a una planificación rígida inicial, sino a un proceso incremental de aprendizaje técnico, identificación de limitaciones y adopción progresiva de patrones orientados a mejorar atributos de calidad como mantenibilidad, escalabilidad, reutilización y testabilidad.

Cada caso de estudio introduce mejoras estructurales que responden a problemáticas detectadas en el anterior, configurando una trayectoria evolutiva coherente.

4.3.1 Caso de Estudio 1 - Sistema MVC CodeIgniter

El Sistema MVC CodeIgniter constituye el punto de partida de la secuencia evolutiva. Se trata de una aplicación web tradicional desarrollada con CodeIgniter 3, implementando el modelo MVC en su forma más básica y directa.

En este desarrollo, la arquitectura se caracteriza por ser una aplicación monolítica donde el frontend y el backend están integrados en el mismo proyecto. El framework CodeIgniter proporciona una estructura MVC nativa donde los controladores reciben las peticiones HTTP, acceden directamente a los modelos para operaciones de base de datos, y renderizan vistas que se envían al cliente.

La Figura 1 representa la arquitectura adoptada en este desarrollo.

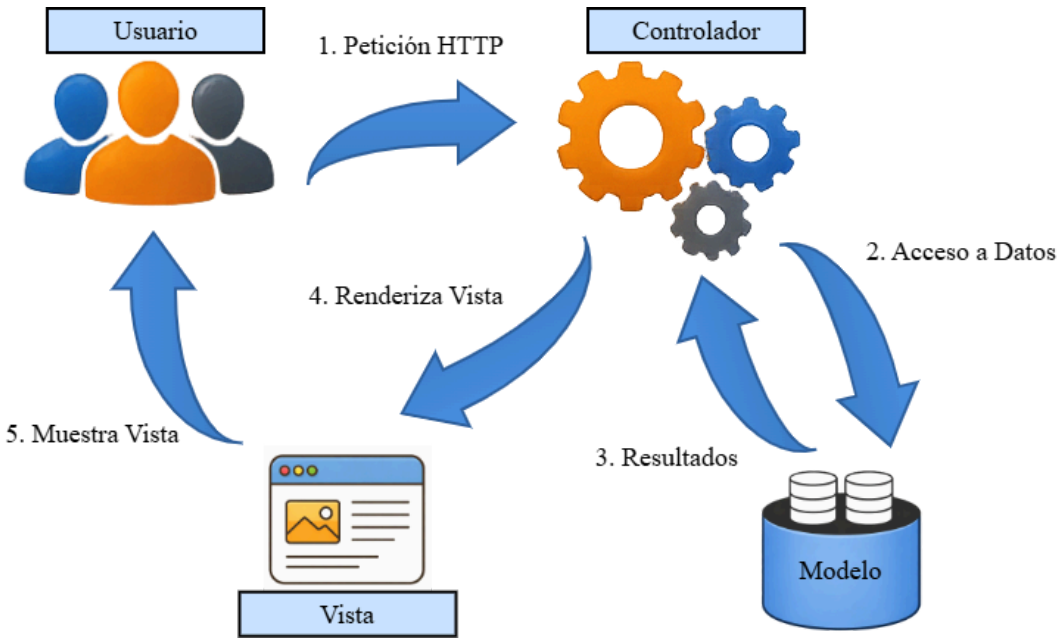

Figura 1. Caso de Estudio 1 (Sistema MVC CodeIgniter) - Arquitectura MVC tradicional monolítica.
Fuente: elaboración propia.

Análisis técnico

Este desarrollo se caracteriza por:

- Arquitectura monolítica: frontend y backend integrados en el mismo proyecto.
- MVC básico: Controlador → Modelo → Vista, sin capas intermedias.
- Acceso directo a la base de datos desde los modelos mediante Active Record de CodeIgniter.
- Lógica de negocio distribuida entre controladores y modelos.
- Renderizado de vistas en el servidor (server-side rendering).
- Ausencia de separación entre capa de presentación y lógica de negocio.

Desde la perspectiva de atributos de calidad:

- **Simplicidad inicial:** estructura clara y fácil de entender para desarrolladores con experiencia en MVC tradicional.
- **Rapidez de desarrollo:** adecuado para proyectos de alcance acotado y requisitos simples.
- **Mantenibilidad limitada:** a medida que el dominio crece, la lógica de negocio se dispersa entre controladores y modelos, dificultando su mantenimiento.
- **Baja reutilización:** no existe una capa de servicios que permita reutilizar lógica de negocio desde múltiples puntos de entrada.
- **Testabilidad reducida:** el acoplamiento entre controladores, modelos y vistas dificulta la realización de pruebas unitarias aisladas.
- **Escalabilidad limitada:** la arquitectura monolítica y la ausencia de separación de responsabilidades dificultan la escalabilidad del sistema.

Problemas detectados

A medida que el sistema crece en complejidad, surgen limitaciones estructurales:

- **Duplicación de lógica:** reglas de negocio repetidas en múltiples controladores.
- **Alto acoplamiento:** controladores fuertemente acoplados a modelos específicos y a la estructura de base de datos.
- **Dificultad para reutilizar código:** no existe una capa intermedia que permita compartir lógica entre diferentes controladores.
- **Testing complejo:** las pruebas requieren instanciar controladores completos y simular el entorno del framework.
- **Dificultad para evolucionar:** cambios en la estructura de datos o en la lógica de negocio requiere modificar múltiples puntos del código.

Este caso de estudio evidencia explícitamente las limitaciones del MVC básico cuando se aplica a sistemas empresariales de mediana o gran complejidad, estableciendo la necesidad de una arquitectura más estructurada.

El aporte principal de este caso de estudio es establecer el punto de referencia inicial: un desarrollo MVC tradicional que, si bien es adecuado para proyectos simples, evidencia claramente las limitaciones que motivaron la evolución hacia arquitecturas más estructuradas. La experiencia con este desarrollo demostró que, para sistemas empresariales complejos, se requiere una separación más clara de responsabilidades y la introducción de capas intermedias que permitan mejorar la mantenibilidad, testabilidad y escalabilidad.

Ejemplo de implementación (Caso de Estudio 1 - Sistema MVC CodeIgniter)

El controlador accede directamente al modelo y renderiza la vista. No existe capa de servicios ni repositorios.

```
// Controller: gestión de usuarios (Caso de Estudio 1)
class Usuarios extends CI_Controller {
        public function index() {
                $this->load->model('Usuario_model');
                $data['usuarios'] = $this->Usuario_model->obtener_todos();
                $this->load->view('usuarios/listar', $data);
        }

        public function crear() {
                $this->load->model('Usuario_model');
                $datos = array(
                        'nombre' => $this->input->post('nombre'),
                        'email' => $this->input->post('email')
                );
                $this->Usuario_model->insertar($datos);
                redirect('usuarios');
        }
}
```

```
// Model: acceso directo a base de datos (Caso de Estudio 1)
class Usuario_model extends CI_Model {
        public function obtener_todos() {
                return $this->db->get('usuarios')->result();
        }

        public function insertar($datos) {
                $this->db->insert('usuarios', $datos);
        }
}
```

La lógica de negocio está mezclada con la lógica de control, y no existe una capa que pueda reutilizarse desde otros puntos del sistema.

4.3.2 Caso de Estudio 2

Este caso de estudio adopta una arquitectura cliente-servidor desacoplada, con frontend SPA desarrollado en React (Vite), basado en componentes y backend API en Laravel 11.

Si bien el backend responde al modelo MVC propio del framework, la decisión de separar frontend y backend mediante servicios REST implica ya una mejora estructural respecto de arquitecturas monolíticas tradicionales.

La Figura 2 representa la arquitectura adoptada en este desarrollo.

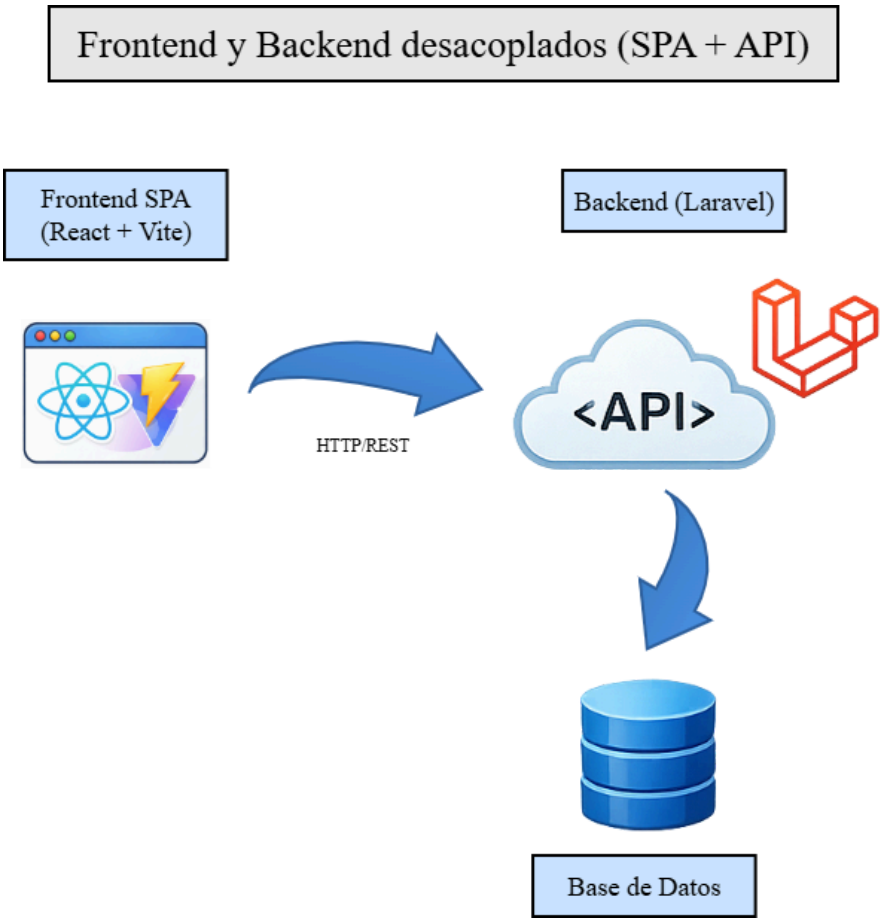

Figura 2. Caso de Estudio 2 - Arquitectura cliente-servidor con frontend y backend desacoplados.
Fuente: elaboración propia.

Análisis técnico

Este caso de estudio incorpora desde el inicio:

- Arquitectura cliente-servidor: separación clara entre frontend SPA y backend API.
- Frontend basado en componentes: la interfaz se construye con componentes reutilizables (React)
- Multi-tenancy (aislamiento lógico de datos por tenant).
- Autenticación basada en tokens (Laravel Sanctum).
- Control de acceso basado en roles (Spatie Laravel Permission).

Desde la perspectiva de atributos de calidad:

- **Escalabilidad organizacional:** soporte para múltiples instituciones (tenants) con aislamiento de datos.
- **Seguridad estructural:** autenticación y autorización formalizadas mediante mecanismos estándar.
- **Desacoplamiento parcial:** permite evolución independiente de la interfaz y la lógica de negocio.

Sin embargo, internamente el backend aún no implementa separación en capas profundas; la lógica de negocio puede tender a concentrarse en controladores, manteniendo cierta similitud estructural con el MVC básico del Caso de Estudio 1, aunque con tecnologías más modernas.

Este desarrollo establece la base tecnológica moderna y la separación arquitectónica entre cliente y servidor, pero no representa todavía una evolución en patrones internos de dominio hacia capas intermedias.

Valor evolutivo

El aporte principal de este desarrollo es la adopción de una arquitectura desacoplada frontend-backend, junto con mecanismos formales de autenticación, autorización y aislamiento de datos mediante multi-tenancy. Se establece así una base tecnológica sólida y una separación arquitectónica clara sobre la cual evolucionarán las decisiones arquitectónicas internas hacia una estructura en capas más profesional.

Ejemplo de implementación (Caso de Estudio 2) - Frontend: componente

```
// components/UsuarioList.jsx (Caso de Estudio 2)
import { useState, useEffect } from 'react';

export function UsuarioList() {
      const [usuarios, setUsuarios] = useState([]);
       useEffect(() => {
```

```
            fetch('/api/usuarios')
             .then(res => res.json())
            .then(setUsuarios);
      }, []);
      return (
            <ul>
                  {usuarios.map(u => <li key={u.id}>{u.nombre}</li>)}
            </ul>
       );
}
```

La lógica de comunicación con la API queda acoplada al componente

4.3.3 Caso de Estudio 3 - arquitectura en capas

El Caso de Estudio 3 representa el primer salto arquitectónico estructural.

Se implementa formalmente:

Controlador → Servicio → Repositorio → Modelo

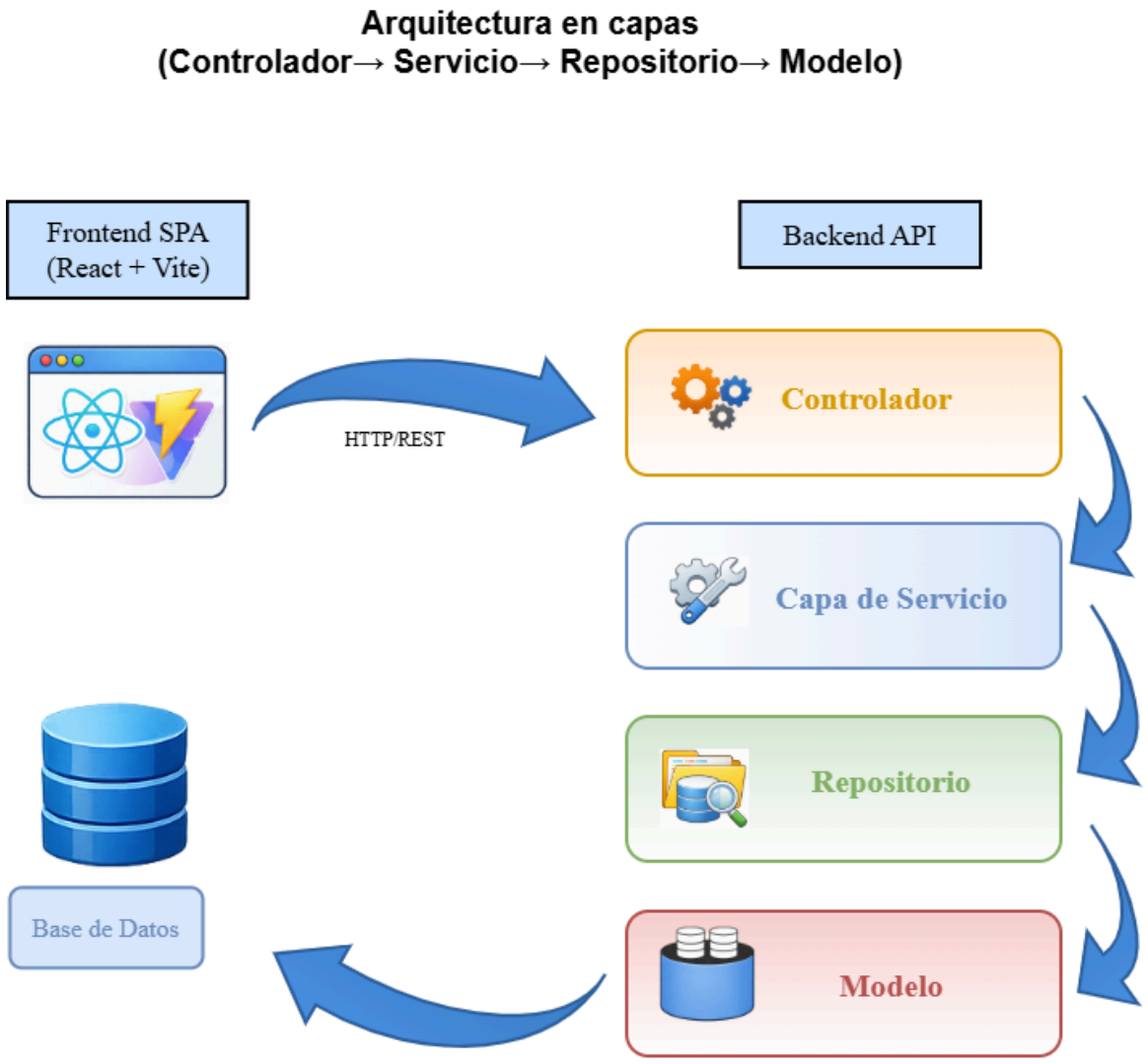

Figura 3. Caso de Estudio 3 - Introducción de arquitectura en capas.

Fuente: elaboración propia.

Mejora respecto al caso de estudio anterior

Se resuelven limitaciones previas mediante:

- Encapsulamiento de lógica de negocio en servicios.
- Abstracción del acceso a datos mediante repositorios.
- Separación entre dominio y transporte HTTP.

Impacto en atributos de calidad

- Reducción de acoplamiento
- Incremento de cohesión por capa
- Mejora significativa en testabilidad
- Mayor reutilización de reglas de negocio

Aquí comienza la profesionalización explícita de la arquitectura.

Patrones incorporados

- Servicio
- Repositorio
- Inyección de dependencia

Valor evolutivo

Se reduce el acoplamiento entre la capa HTTP y el dominio, mejora la testabilidad y se incrementa la reutilización de reglas de negocio. Este desarrollo marca el inicio de la profesionalización explícita de la arquitectura backend.

Ejemplo de implementación (Caso de Estudio 3) — Backend: Controlador → Servicio → Repositorio → Modelo

Se introduce la separación en capas: el controlador solo delega en el servicio; el servicio orquesta la lógica de negocio y utiliza el repositorio; el repositorio encapsula el acceso a datos.

```php
<?php
// Controller: delgado, solo coordina petición y respuesta (Caso de Estudio 3)
// Laravel - Arquitectura en capas

namespace App\Http\Controllers;

use App\Services\UsuarioService;

class UsuarioController extends Controller
{
    public function __construct(private UsuarioService $usuarioService) {}
```

```
    public function index()
    {
        $usuarios = $this->usuarioService->obtenerTodos();
        return response()->json($usuarios);
    }
}
<?php
// Repository: acceso a datos (Caso de Estudio 3)
// Laravel - Arquitectura en capas
namespace App\Repositories;

use App\Models\Usuario;
use Illuminate\Support\Collection;

class UsuarioRepository
{
    public function todos(): Collection
    {
        return Usuario::all();
    }
}
<?php
// Service: lógica de negocio y orquestación (Caso de Estudio 3)
// Laravel - Arquitectura en capas
namespace App\Services;

use App\Repositories\UsuarioRepository;
use Illuminate\Support\Collection;

class UsuarioService
{
    public function __construct(private UsuarioRepository $repository) {}

    public function obtenerTodos(): Collection
    {
        return $this->repository->todos();
    }
}
```

4.3.4 Caso de Estudio 4 - Patrón de Especificación

En este desarrollo el modelo en capas ya se encuentra consolidado como base estructural del sistema, incorporando Capa de Servicio, Repositorio e Inyección de Dependencias. Sin embargo, surge una nueva necesidad vinculada a la construcción de consultas dinámicas complejas, que evidencian limitaciones en la organización interna de los repositorios.

Ante esta situación, se introduce formalmente el **Patrón de Especificación**, permitiendo encapsular criterios de búsqueda, filtrado, ordenamiento y paginación en objetos reutilizables y combinables.

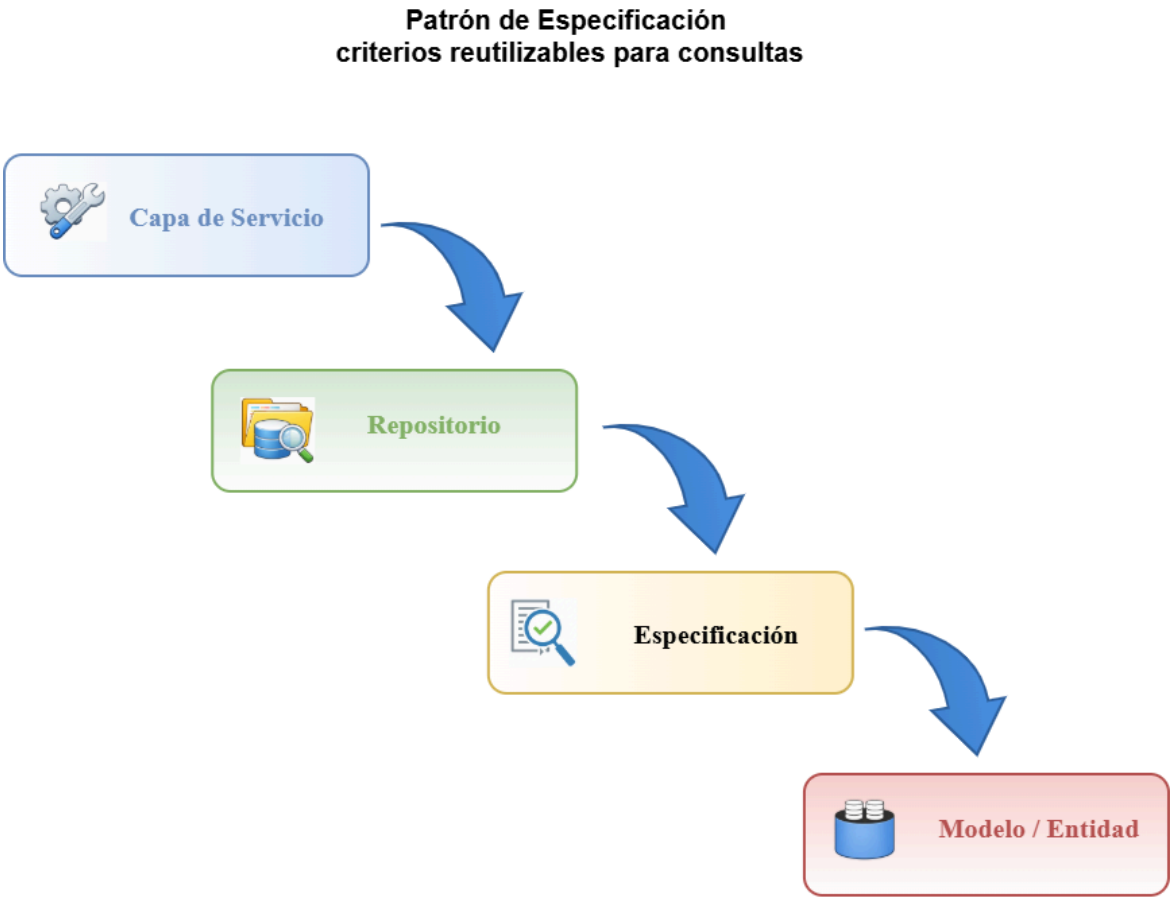

Figura 4. Caso de Estudio 4 - Patrón de Especificación.
Fuente: elaboración propia.

Evolución cualitativa

La arquitectura mantiene la estructura:

Controlador → Servicio → Repositorio → Modelo

con inyección de dependencias entre capas.

No obstante, se agrega un nuevo nivel conceptual: la externalización de la lógica de consulta mediante objetos de especificación.

Este cambio implica:

- Separación entre lógica de negocio y lógica de consulta.
- Eliminación de duplicación de filtros en repositorios.
- Mayor expresividad del dominio.
- Posibilidad de combinar criterios sin modificar código existente.

El repositorio deja de contener múltiples métodos específicos de búsqueda y pasa a recibir especificaciones que definen el comportamiento de la consulta.

Impacto en atributos de calidad

La incorporación del patrón Specification fortalece:

- **Reutilización**, mediante criterios combinables.
- **Mantenibilidad**, al centralizar reglas de consulta.
- **Extensibilidad**, permitiendo agregar nuevos filtros sin alterar código existente.
- **Testabilidad**, al poder validar especificaciones de manera aislada.

Este caso de estudio representa una evolución desde una arquitectura correctamente estructurada hacia una arquitectura más expresiva y orientada al dominio.

Valor evolutivo

El valor evolutivo del Caso de Estudio 4 radica en la formalización de la lógica de consulta como parte explícita del diseño arquitectónico. La arquitectura deja de organizar únicamente responsabilidades estructurales y comienza a modelar variabilidad del dominio de forma declarativa y reutilizable.

Ejemplo de implementación (Caso de Estudio 4) — Backend: Repositorio con especificación

El repositorio deja de exponer múltiples métodos de búsqueda y acepta objetos de especificación que encapsulan criterios de consulta reutilizables y combinables.

```php
<?php

// Interface de especificación (Caso de Estudio 4)

// Patrón Specification - criterios de consulta reutilizables

namespace App\Specifications;

use Illuminate\Database\Eloquent\Builder;

interface SpecificationInterface

{

    public function apply(Builder $query): Builder;

}
```

```php
<?php

// Especificación concreta: filtrar por estado activo (Caso de Estudio 4)


namespace App\Specifications;


use Illuminate\Database\Eloquent\Builder;


class ConEstadoActivo implements SpecificationInterface

{
```

```
    public function apply(Builder $query): Builder
    {
        return $query->where('activo', true);
    }
}
```

```
<?php
// Repositorio que aplica una especificación (Caso de Estudio 4)
namespace App\Repositories;
use App\Models\Usuario;
use App\Specifications\SpecificationInterface;
use Illuminate\Support\Collection;
class UsuarioRepository
{
    public function buscarCon(SpecificationInterface $spec): Collection
    {
        return $spec->apply(Usuario::query())->get();
    }
}
```

Uso desde el servicio: se compone el criterio sin duplicar lógica en el repositorio. En el Caso de Estudio 5 esta idea se extiende con especificaciones combinables (paginación, orden, búsqueda, etc.) y un método busqueda() que acepta varias especificaciones.

4.3.5 Caso de Estudio 5 - modelo en capas completo

En este caso de estudio se consolida un único backend (o backends nuevos) con arquitectura en capas completa: Controlador (delgado) → Servicio (lógica de

negocio) → Repositorio (acceso a datos) → Especificación → Modelo (Eloquent). Las interfaces para repositorios y servicios se registran en el contenedor (inyección de dependencias); opcionalmente se usan Traits, Policies, Jobs y Commands.

El frontend se implementa como una SPA desacoplada (React + Vite), basada en componentes y organizada por módulos. Cada módulo define su propia capa de servicios, responsable de encapsular la comunicación HTTP con la API, junto con librerías para gestión de estado y caché (TanStack React Query y Zustand). En este modelo consolidado, se incorporan explícitamente tanto la capa de servicios por módulo (como interfaz de integración con el backend) como los *custom hooks* de dominio.

Este desarrollo constituye el **modelo arquitectónico consolidado actual**, que actúa como referencia y estándar para los desarrollos recientes del ámbito de la práctica profesional.

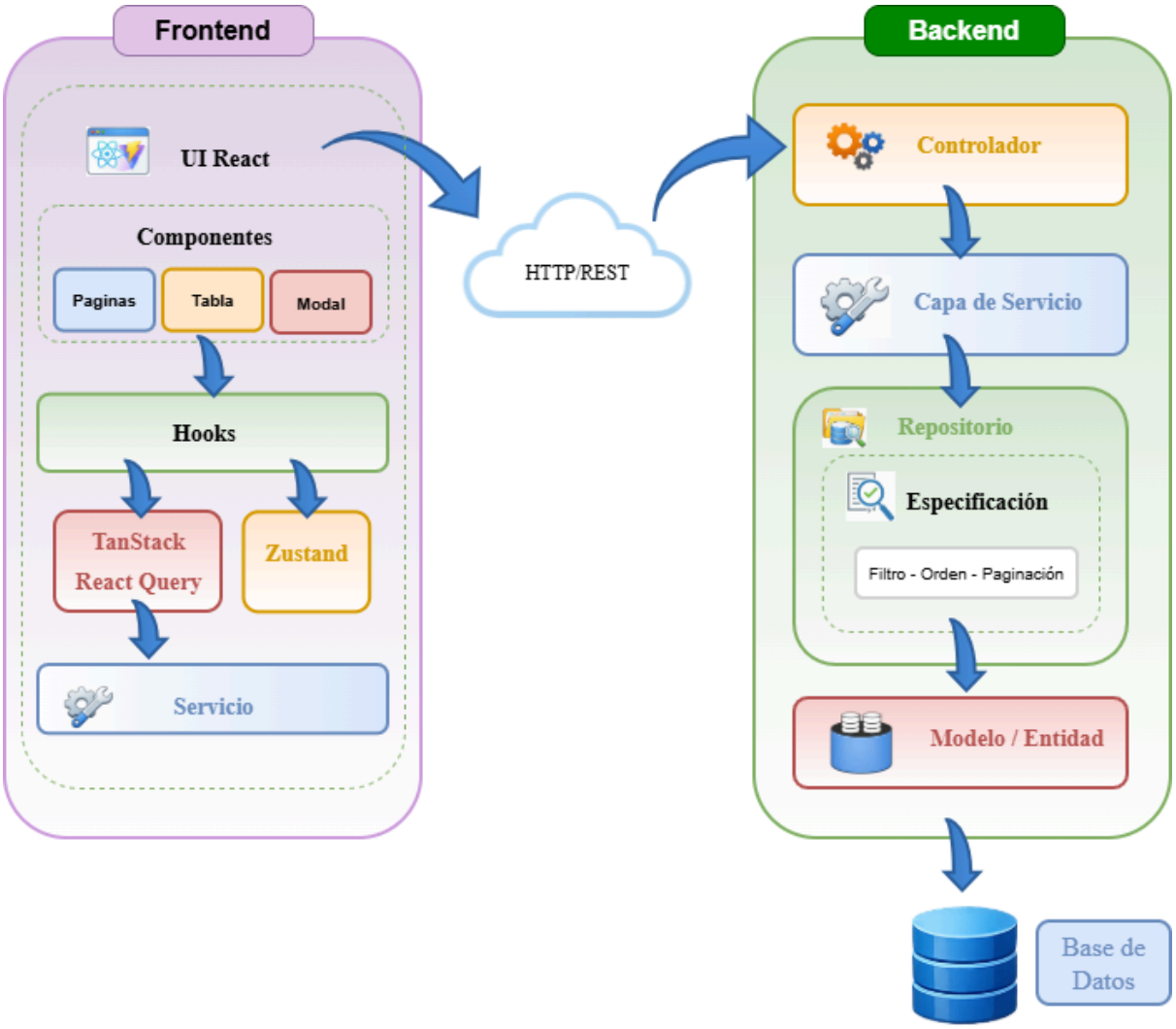

Figura 5. Caso de Estudio 5 - Modelo en capas con Especificación consolidado.
Fuente: elaboración propia.

Patrones y mecanismos arquitectónicos identificados (backend):

- **Repositorio**, con una base reutilizable (*BaseRepository*) y repositorios por entidad, incluyendo un método de búsqueda basado en especificaciones (p. ej., `busqueda(SpecificationInterface ...)`).

- **Capa de Servicio**, con servicios por dominio y una base común (*BaseService*) para encapsular reglas de negocio y orquestación.

- **Especificación**, implementada mediante especificaciones combinables para criterios de consulta (paginación, filtros, búsqueda, relaciones, ordenamiento y soft deletes).

- **Inyección de dependencia**, mediante registro de **interfaces** y sus implementaciones en el contenedor (p. ej., *BindingInterfacesProvider*), favoreciendo el desacoplamiento y el testing.

Elementos complementarios del diseño:

- **Traits** para comportamiento transversal (manejo de excepciones, respuestas API, filtrado y búsqueda).

- **Policies** para autorización basada en modelos.

- **Commands/Jobs** (cuando aplica) para tareas asíncronas y mantenimiento.

- Aplicación de principios SOLID, como **segregación de interfaces**, mediante contratos específicos por capa.

Frontend (organización y tecnologías):

Arquitectura modular basada en componentes, con servicios por módulo que encapsulan la comunicación HTTP (Axios), *custom hooks*, gestión de estado local (Zustand) y manejo de datos del servidor con caché/sincronización (TanStack React Query).

Características avanzadas

- Especificaciones combinables para filtros, búsquedas, ordenamientos y paginación.
- Interfaces desacopladas.
- Traits para comportamiento transversal.
- Soporte para Jobs y Commands.

Impacto en atributos de calidad

- Alta testabilidad
- Elevada reutilización
- Escalabilidad estructural
- Separación estricta de responsabilidades

Este caso de estudio puede considerarse el punto de estabilización técnica del modelo arquitectónico.

Valor agregado de este desarrollo:

- **Separación clara de responsabilidades:** cada capa tiene un rol bien definido; los controladores solo coordinan la petición y la respuesta.
- **Reutilización:** servicios y repositorios se usan desde múltiples controladores, comandos y jobs.
- **Especificación:** las consultas complejas se componen con objetos reutilizables (filtros, búsqueda, orden, paginación), lo que mejora la expresividad y la testabilidad.
- **Testabilidad:** la inyección de interfaces permite reemplazar repositorios y servicios por mocks en pruebas unitarias.
- **Escalabilidad:** incorporar nuevos dominios o endpoints sigue el mismo patrón; el equipo tiene un estándar claro.

Ejemplo de implementación (Caso de Estudio 5) - Repository con Specification

El repositorio base expone un método busqueda() que acepta varias especificaciones combinables:

```
abstract class BaseRepository implements BaseRepositoryInterface
{
    protected string $modelo;

    public function busqueda(SpecificationInterface ...$especificaciones):
Collection|Paginator
    {
        return $this->realizarConsulta($this->modelo::query(), ...$especificaciones);
    }

    // Métodos CRUD estándar...
}
```

Uso desde un servicio: se combinan filtros, búsqueda, ordenamiento y paginación sin duplicar lógica en cada endpoint:

```
$resultados = $repository->busqueda(
    new WithFiltersFromQueryParams($request),
    new WithSearchFromQueryParams($request, ['nombre', 'email']),
    new WithRelationships(['relacion1', 'relacion2']),
    new WithSorting($request),
    new WithPagination(15)
);
```

Ejemplo de Specification (paginación)

```
interface SpecificationInterface
{
    public function handle(Builder $query): Builder|Paginator;
}

class WithPagination implements SpecificationInterface
```

```
{
    public function __construct(private int $pageSize) {}

    public function handle(Builder $query): Paginator
    {
        return $query->paginate($this->pageSize);
    }
}
```

Ejemplo de inyección de dependencias (binding de interfaces)

Las implementaciones concretas se vinculan a interfaces en un Service Provider; el resto del código depende de las interfaces, no de las clases concretas:

```
class BindingInterfacesProvider extends ServiceProvider
{
    public array $bindings = [
        PersonaFisicaRepositoryInterface::class => PersonaFisicaRepository::class,
        PersonaFisicaServiceInterface::class => PersonaFisicaService::class,
        // ... más de 40 bindings
    ];

    public function register(): void
    {
        foreach ($this->bindings as $interface => $concrete) {
            $this->app->bind($interface, $concrete);
        }
    }
}
```

Ejemplo de implementación (Caso de Estudio 5) — Frontend: servicio por módulo y custom hook con React Query

En el modelo consolidado, cada módulo expone un servicio que encapsula las llamadas HTTP (Axios) y custom hooks que utilizan TanStack React Query para caché y sincronización:

```
// modules/expedientes/services/expedientesService.ts (Caso de Estudio 5)
// Servicio por módulo: encapsula la comunicación HTTP con la API (Axios)
import api from '@/shared/services/api';
export const expedientesService = {
  obtenerTodos: () => api.get('/expedientes').then(res => res.data),
  obtenerPorId: (id: number) => api.get(`/expedientes/${id}`).then(res =>
res.data),
  crear: (data: Record<string, unknown>) => api.post('/expedientes', data),
};
// modules/expedientes/hooks/useExpedientes.ts (Caso de Estudio 5)
// Custom hooks con TanStack React Query para caché y sincronización

import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query';
import { expedientesService } from '../services/expedientesService';
export function useExpedientes() {
  return useQuery({
    queryKey: ['expedientes'],
    queryFn: () => expedientesService.obtenerTodos(),
  });
}
export function useCrearExpediente() {
  const queryClient = useQueryClient();
  return useMutation({
    mutationFn: expedientesService.crear,
    onSuccess: () => queryClient.invalidateQueries({ queryKey: ['expedientes']
}),
```

```
  });
}
```

El servicio por módulo centraliza la comunicación con la API; los custom hooks encapsulan la lógica de datos (React Query) y se reutilizan en los componentes.

4.4. Síntesis evolutiva

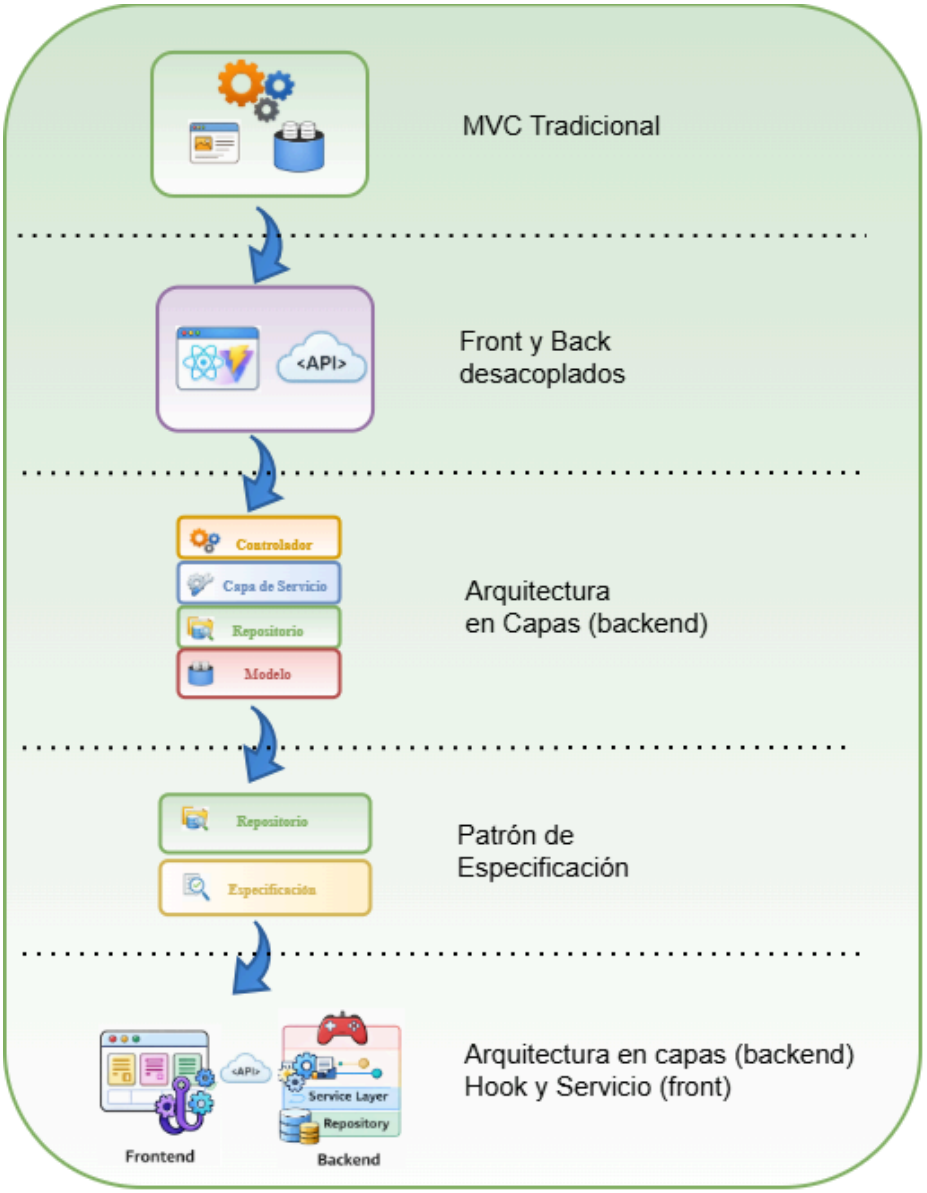

Figura 6. Síntesis de la evolución arquitectónica.

Fuente: elaboración propia.

La progresión puede sintetizarse como:

MVC básico monolítico → Cliente-servidor desacoplado → Arquitectura en capas → Capas + Specification → **Modelo consolidado (Caso de Estudio 5)**.

El Caso de Estudio 5 constituye el modelo arquitectónico consolidado actual, que actúa como referencia y estándar para desarrollos recientes. Este proceso evidencia una transición desde soluciones centradas en rapidez de implementación hacia una arquitectura madura, testable y reutilizable, alineada con principios de ingeniería de software orientados a la mantenibilidad y escalabilidad.

Tabla 2 - Comparación evolutiva de arquitecturas y atributos de calidad

Con el fin de sintetizar comparativamente la evolución arquitectónica documentada, la Tabla 2 resume los principales cambios estructurales introducidos en cada desarrollo, los problemas que motivaron dichas mejoras y los atributos de calidad fortalecidos en cada etapa.

Desarrollo	Arquitectura adoptada	Patrones principales	Problema detectado en etapa previa	Mejora introducida	Atributos de calidad fortalecidos
Caso de estudio 1	Arquitectura MVC - monolítico	MVC, Active Record (CI), SSR	No aplica	Implementación básica estructurada bajo MVC	Simplicidad inicial, rapidez de desarrollo
Caso de estudio 2	Cliente-servidor (SPA + API) con MVC interno	MVC, Multi-tenancy, RBAC, Auth API	Ausencia de desacoplamiento en arquitecturas tradicionales	Separación frontend-backend; soporte multi-tenant desde diseño inicial	Escalabilidad organizacional, seguridad, desacoplamiento

Caso de estudio 3	Arquitectura en capas (C → S → R → M)	Service Layer, Repository, DI	Baja reutilización y testabilidad en MVC básico	Separación formal de responsabilidades y abstracción de acceso a datos	Mantenibilidad, testabilidad, cohesión
Caso de estudio 4	Modelo en capas aplicado desde el inicio	Repository + Service + Specification + Interfaces + Policies	Necesidad de mayor disciplina estructural	Estandarización interna de capas, interfaces y dependencias	Consistencia, reducción de deuda técnica
Caso de estudio 5	Capas completas + Specification combinable	Repository, Service Layer, Specification avanzada, DI, Traits, Policies	Necesidad de escalabilidad y reutilización avanzada	Consolidación técnica del modelo; especificaciones reutilizables y testables	Escalabilidad estructural, alta testabilidad, reutilización

La comparación evidencia que la evolución no fue únicamente tecnológica, sino también metodológica y organizacional, culminando en la adopción sistemática de un modelo arquitectónico estandarizado.

4.5 Modelo arquitectónico actual

El modelo arquitectónico actual —denominado modelo ideal o consolidado— constituye el resultado del proceso evolutivo documentado en la sección 4.3 y corresponde a la arquitectura implementada en el **Caso de Estudio 5**. No se trata de una decisión aislada, sino de la consolidación progresiva de prácticas que demostraron mejorar atributos de calidad en los desarrollos previos (Caso de estudio 1 a 4).

4.5.1 Backend (Laravel)

El modelo consolidado adopta una arquitectura en capas estructurada como:

Controlador → Servicio → Repositorio → Especificación → Modelo

Cada capa posee responsabilidades claramente delimitadas:

- **Controlador:** gestiona la interacción HTTP y delega la lógica.
- **Servicio:** encapsula reglas de negocio y coordina operaciones.
- **Repositorio:** abstrae el acceso a datos mediante interfaces.
- **Especificación:** encapsula criterios de consulta combinables (filtros, búsqueda, ordenamiento y paginación).
- **Modelo:** representa el dominio persistente mediante ORM.

Esta organización reduce el acoplamiento entre la lógica del dominio y los mecanismos de transporte o persistencia, incrementando la cohesión interna de cada componente.

Las dependencias se gestionan mediante interfaces vinculadas a implementaciones concretas a través del contenedor de inversión de control, favoreciendo la inversión de dependencias y la posibilidad de testing aislado.

Se emplean además:

- **Traits** para comportamiento transversal.
- **Policies** para autorización basada en dominio.
- **Jobs o Commands** cuando se requiere procesamiento asíncrono o desacoplado.

4.5.2 Frontend (React/Vite)

El frontend se implementa como SPA desacoplada del backend, comunicándose mediante HTTP/REST en formato JSON.

La organización por módulos permite encapsular:

- Servicios HTTP por dominio.
- Lógica de estado del cliente.

- Componentes reutilizables.

Se incorporan herramientas modernas para:

- Gestión de datos del servidor (caching y sincronización).
- Validación estructurada de formularios.
- Construcción de interfaces consistentes.

Este enfoque mantiene coherencia estructural con el backend, extendiendo el principio de separación de responsabilidades al lado cliente.

4.5.3 Justificación técnica

La arquitectura en capas mejora significativamente la **separación de responsabilidades**, facilitando la reutilización de lógica de negocio y aumentando la testabilidad gracias a la inyección de dependencias.

La incorporación sistemática del patrón de Especificación reduce la duplicación de lógica en consultas complejas y mejora la expresividad del dominio, permitiendo encapsular criterios de búsqueda y filtrado de forma combinable y reutilizable.

Desde una perspectiva arquitectónica más amplia, el modelo consolidado se alinea con:

- El principio de inversión de dependencias.
- La separación entre políticas de negocio y detalles de infraestructura.
- Los principios de mantenibilidad y evolución progresiva del software.

En términos de atributos de calidad, el modelo fortalece:

- **Mantenibilidad**, al reducir el acoplamiento entre capas.
- **Testabilidad**, mediante interfaces y desacoplamiento estructural.
- **Escalabilidad estructural**, al permitir incorporar nuevos dominios sin alterar la arquitectura base.

- **Reutilización**, gracias a servicios y especificaciones compartidas.
- **Reducción de deuda técnica**, al adoptar el modelo desde el diseño inicial en nuevos proyectos.

En consecuencia, el modelo arquitectónico actual no representa únicamente una mejora técnica respecto de implementaciones anteriores, sino una maduración conceptual de la práctica profesional. La arquitectura deja de ser una solución puntual y pasa a constituirse como un estándar reproducible y coherente para desarrollos futuros.

5. Resultados

El análisis evolutivo realizado permitió identificar un proceso progresivo de maduración arquitectónica en los sistemas desarrollados en el marco de la práctica profesional.

A partir del relevamiento de cinco casos de estudio distintos, se observaron transformaciones estructurales que evidencian una transición desde implementaciones basadas en MVC básico monolítico hacia una arquitectura cliente-servidor desacoplada, y posteriormente hacia una arquitectura en capas consolidada e institucionalizada como estándar organizacional.

5.1 Resultados del relevamiento

Se documentaron cinco casos de estudio de sistemas mediante análisis de código fuente y documentación técnica full-stack, identificando:

- Estilo arquitectónico adoptado.
- Patrones de diseño implementados.
- Tecnologías utilizadas.
- Organización del frontend y backend.
- Ejemplos concretos de implementación.

Este relevamiento permitió establecer una secuencia evolutiva coherente, en la cual cada desarrollo incorpora mejoras respecto del anterior.

5.2 Resultados del análisis evolutivo

El análisis comparativo permitió identificar los siguientes hallazgos:

1. El Caso 1 (MVC monolítico) y el Caso 2 (cliente-servidor con MVC interno) comparten limitaciones en separación interna de responsabilidades cuando la lógica de negocio queda concentrada en controladores, afectando mantenibilidad y reutilización.

2. La incorporación de Service Layer y Repository reduce el acoplamiento entre transporte HTTP y lógica de negocio.

3. El patrón Specification mejora significativamente la expresividad del dominio y evita duplicación de lógica en consultas complejas.

4. La introducción sistemática de interfaces e inyección de dependencias incrementa la testabilidad.

5. El modelo consolidado (Caso de Estudio 5) actúa como referencia para desarrollos posteriores, reduciendo la deuda técnica cuando se adopta desde el diseño inicial.

Se constató que cada transición arquitectónica estuvo motivada por limitaciones prácticas detectadas en proyectos anteriores.

5.3 Impacto en atributos de calidad

El modelo arquitectónico consolidado fortalece múltiples atributos de calidad:

- **Mantenibilidad:** mayor facilidad para modificar reglas sin afectar otras capas.

- **Testabilidad:** posibilidad de aislar servicios y repositorios en pruebas unitarias.

- **Escalabilidad estructural:** incorporación de nuevos módulos siguiendo patrón predefinido.

- **Reutilización:** Specifications y servicios reutilizables entre proyectos.
- **Consistencia organizacional:** reducción de variabilidad estructural entre sistemas.

Estos resultados confirman que la evolución arquitectónica no fue únicamente tecnológica, sino orientada explícitamente a mejorar la calidad del software producido.

5.4 Consolidación del modelo actual

Como resultado final del proceso evolutivo, se definió un modelo arquitectónico consolidado que actúa como referencia para desarrollos recientes.

Este modelo:

- Formaliza la separación de responsabilidades.
- Estandariza la organización de capas.
- Establece patrones reproducibles.
- Reduce la curva de incorporación de nuevos desarrolladores.
- Permite el crecimiento controlado del sistema.

En consecuencia, el resultado principal de este trabajo no es solo la documentación histórica de arquitecturas, sino la sistematización de un modelo arquitectónico maduro y reproducible, alineado con principios de ingeniería de software orientados a calidad y evolución sostenible.

6. Objetivos y metas alcanzados

- **Objetivo general:** Analizar y documentar la evolución de arquitecturas de software y patrones de diseño en sistemas desarrollados en el marco de la práctica profesional.

- **Meta alcanzada:** monografía que sistematiza dicha evolución por desarrollo, el valor agregado en cada mejora, diagramas de arquitectura por desarrollo y ejemplos de implementación.
- **Objetivos específicos:**
    - (1) Relevar y documentar arquitecturas y patrones - logrado mediante la documentación técnica por sistema y los diagramas de arquitectura;
    - (2) Analizar la evolución e identificar mejoras y valor agregado por desarrollo - logrado en el análisis evolutivo (secciones 4.3 y 5);
    - (3) Identificar y catalogar patrones por sistema - logrado en la documentación de cada sistema y resumido en la monografía;
    - (4) Modelar arquitecturas - logrado con diagramas de arquitectura full-stack por caso de estudio (Figuras 1 a 5);
    - (5) Documentar el proceso de perfeccionamiento - logrado en las secciones de evolución por desarrollos y modelo actual;
    - (6) Describir el modelo arquitectónico actual y su justificación - logrado en el apartado 4.5 y en Resultados (modelo consolidado correspondiente al Caso de Estudio 5).

7. Conclusiones

El análisis evolutivo desarrollado en este trabajo evidencia un proceso sostenido de maduración arquitectónica en los sistemas implementados en el marco de la práctica profesional. La trayectoria observada demuestra que las decisiones arquitectónicas no surgieron como definiciones estáticas iniciales, sino como resultado de la experiencia acumulada y la identificación progresiva de limitaciones estructurales.

Las primeras implementaciones priorizaron simplicidad y rapidez de desarrollo, adecuadas para contextos de alcance acotado. El sistema MVC CodeIgniter (Caso de Estudio 1) representó el punto de partida con la arquitectura monolítica tradicional. Posteriormente, el Caso de Estudio 2 introdujo la separación cliente-servidor y capacidades empresariales como multi-tenancy y RBAC, pero manteniendo internamente un MVC básico. Sin embargo, estas soluciones

comenzaron a evidenciar restricciones en términos de reutilización, mantenibilidad y testabilidad a medida que el dominio crecía en complejidad.

La incorporación gradual de capas intermedias - Capa de Servicio y Repositorio - permite desacoplar la lógica de negocio del acceso a datos y del transporte HTTP, reduciendo el acoplamiento y mejorando la cohesión interna del sistema. Posteriormente, la adopción del patrón Specification introdujo un mecanismo formal para encapsular criterios de consulta reutilizables y combinables, fortaleciendo la expresividad del dominio y evitando duplicación de lógica.

El modelo arquitectónico consolidado actual no surge como una decisión aislada, sino como el resultado de un proceso empírico de perfeccionamiento técnico y conceptual. Su institucionalización como estándar organizacional representa un salto cualitativo: la arquitectura deja de ser una solución reactiva ante problemas detectados y pasa a convertirse en una estrategia de diseño anticipatoria, orientada a preservar la calidad del software frente a su evolución futura.

Desde la perspectiva del marco teórico abordado, la evolución documentada confirma la relevancia de principios como la separación de responsabilidades, la inversión de dependencias y la abstracción del acceso a datos en el fortalecimiento de atributos de calidad tales como mantenibilidad, escalabilidad y testabilidad.

En consecuencia, el trabajo no solo documenta una evolución tecnológica entre distintos sistemas, sino una evolución conceptual en la forma de abordar el diseño arquitectónico en el ámbito profesional. El modelo actual constituye una síntesis práctica de principios de ingeniería de software aplicados progresivamente en un contexto real, evidenciando una madurez técnica y organizacional alineada con buenas prácticas reconocidas en la literatura especializada.

8. Recomendaciones

El análisis realizado permite formular recomendaciones orientadas a consolidar y proyectar la madurez arquitectónica alcanzada.

8.1 Continuidad del modelo en capas

Se recomienda mantener la arquitectura en capas como estándar en los nuevos desarrollos backend (Laravel), preservando la estructura Controlador → Servicio → Repositorio → Especificación → Modelo.

La consistencia estructural favorece la mantenibilidad, reduce la variabilidad entre proyectos y consolida una identidad arquitectónica común. La experiencia documentada demuestra que la adopción sistemática del modelo disminuye la deuda técnica inicial y facilita la evolución futura del sistema.

8.2 Formalización de decisiones arquitectónicas

Se recomienda documentar formalmente las decisiones arquitectónicas significativas mediante Architecture Decision Records (ADRs), especialmente cuando se introduzcan:

- Nuevas capas.
- Nuevos patrones de diseño.
- Cambios tecnológicos relevantes.

La documentación explícita de decisiones fortalece la trazabilidad de la evolución arquitectónica y facilita la comprensión del sistema por parte de nuevos integrantes del equipo.

8.3 Evolución hacia una organización orientada al dominio

Se sugiere evaluar progresivamente la adopción de enfoques como *Screaming Architecture* en desarrollos futuros, especialmente en el frontend, organizando la estructura del código por dominios o casos de uso en lugar de únicamente por tipo técnico (components/, services/, etc.).

Este enfoque refuerza la centralidad del dominio y mejora la legibilidad conceptual del sistema, alineándose con principios de diseño orientado al dominio.

8.4 Extensión sistemática de pruebas

Dado que el modelo consolidado favorece la inyección de dependencias y el desacoplamiento estructural, se recomienda ampliar la cobertura de pruebas unitarias e integración, especialmente en:

- Servicios de dominio.
- Repositorios.
- Especificaciones.

El aprovechamiento de mocks y pruebas aisladas permitirá validar reglas de negocio y criterios de consulta de forma independiente, reforzando la confiabilidad del sistema ante cambios evolutivos.

8.5 Proyección hacia escalabilidad organizacional

Finalmente, se recomienda consolidar el modelo arquitectónico como parte de una guía interna de desarrollo, incluyendo:

- Convenciones estructurales.
- Estándares de nomenclatura.
- Lineamientos de implementación.

Esta formalización contribuirá a sostener la madurez alcanzada y permitirá que el modelo evolucione de manera controlada ante nuevos requerimientos tecnológicos o de negocio.

9. Referencias bibliográficas

Anderson, D. J. (2010). *Kanban: Cambio evolutivo exitoso para su negocio de tecnología*. Blue Hole Press.

Bass, L., Clements, P., & Kazman, R. (2012). *Arquitectura de software en la práctica* (3.ª ed.). Addison-Wesley.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifiesto para el desarrollo ágil de software*. https://agilemanifesto.org/iso/es/manifesto.html

Evans, E. (2003). *Diseño orientado al dominio: Abordando la complejidad en el corazón del software*. Addison-Wesley.

Fowler, M. (2002). *Patrones de arquitectura de aplicaciones empresariales*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Patrones de diseño: Elementos de software orientado a objetos reutilizables*. Addison-Wesley.

Martin, R. C. (2008). *Código limpio: Manual de estilo para el desarrollo ágil de software*. Anaya.

Martin, R. C. (2017). *Arquitectura limpia*. Prentice Hall.

Pressman, R. S., & Maxim, B. R. (2021). *Ingeniería del software: Un enfoque práctico* (9.ª ed.). McGraw-Hill.

Schwaber, K., & Sutherland, J. (2020). *La guía de Scrum*. Scrum.org. https://scrumguides.org/

Sommerville, I. (2011). *Ingeniería del software* (9.ª ed.). Pearson.

Mozilla. (s. f.). *Glosario de MDN Web Docs*. MDN Web Docs. Recuperado el 19 de febrero de 2026, de https://developer.mozilla.org/es/docs/Glossary

National Institute of Standards and Technology. (s. f.). *Role-Based Access Control (RBAC)*. NIST Computer Security Resource Center. Recuperado el 19 de febrero de 2026, de https://csrc.nist.gov/projects/role-based-access-control

Glosario de términos técnicos

API (Application Programming Interface). Interfaz que permite a aplicaciones comunicarse entre sí; en el contexto del trabajo, la API REST expuesta por el backend.

Binding. Asociación entre una interfaz o contrato y su implementación concreta en el contenedor de dependencias.

Build. Proceso de compilación y empaquetado del código fuente (por ejemplo, con Vite) para generar los artefactos desplegables.

CRUD. Conjunto de operaciones básicas sobre datos: Crear (Create), Leer (Read), Actualizar (Update) y Eliminar (Delete).

DI (Dependency Injection). Técnica por la cual un componente recibe sus dependencias desde el exterior en lugar de crearlas internamente, reduciendo el acoplamiento y mejorando la testabilidad.

Endpoint. Dirección o URL concreta de una API a la que el cliente envía peticiones (por ejemplo, GET /api/usuarios).

Enrutamiento (routing). Mecanismo que determina qué contenido o componente se muestra según la URL.

HTTP (HyperText Transfer Protocol). Protocolo de comunicación utilizado en la web para enviar peticiones y respuestas entre cliente y servidor.

JSON (JavaScript Object Notation). Formato ligero de intercambio de datos; en el trabajo, el formato habitual de las respuestas de la API REST.

JWT (JSON Web Token). Estándar de token utilizado para autenticación y autorización entre cliente y servidor.

Mock. Objeto simulado que sustituye a una dependencia real en pruebas para controlar su comportamiento y aislar el código bajo prueba.

Multi-tenancy. Arquitectura en la que una misma instancia del sistema sirve a varios “inquilinos” (tenants) con datos aislados entre ellos.

ORM (Object-Relational Mapping). Técnica que mapea estructuras de base de datos relacional a objetos del lenguaje de programación; en el trabajo, Eloquent en Laravel.

RBAC (Role-Based Access Control). Control de acceso basado en roles: los permisos se asignan a roles y los usuarios reciben uno o más roles.

Renderizar. Proceso por el cual el framework (por ejemplo, React) genera la representación visual de los componentes en pantalla.

REST (Representational State Transfer). Estilo arquitectónico para APIs que utilizan HTTP y recursos identificados por URI; en el trabajo, la forma habitual de comunicación frontend-backend.

SPA (Single Page Application). Aplicación web que carga una única página y actualiza el contenido dinámicamente sin recargar la página completa.

SSG (Static Site Generation). Generación de páginas estáticas en tiempo de compilación para servir contenido pre-renderizado.

SSR (Server-Side Rendering). Renderizado del contenido de la página en el servidor antes de enviarlo al cliente, en contraste con el renderizado en el navegador.