Corporación Universitaria del Huila – CORHUILA

Asignatura: Arquitectura de Software

PROPUESTA – FASE 3

Propuesta de Rediseño Arquitectónico y Aplicación de Patrones de Diseño

Docente:
Luis Ángel Vargas Narvaez

Integrantes:
Juan José Torrejano Rojas - jjtorrejano-2032b@corhuila.edu.co
Jhon Edinson Marín Tapias - jemarin-2032b@corhuila.edu.co
Karina Cantillo Plaza - kcantillo-2032b@corhuila.edu.co
Danay Mariana Pereira Ospina - dmpereira-2032b@corhuila.edu.co

Neiva, Huila

23 de febrero de 2026

Introducción

En la presente fase se desarrolla la propuesta de rediseño arquitectónico del sistema "Espagueti de Encuestas", a partir de los anti-patrones y malas prácticas identificados en la Fase 2.

A diferencia de la fase anterior, cuyo enfoque fue diagnóstico, esta etapa tiene un carácter propositivo. El objetivo principal es plantear una arquitectura mejorada basada en patrones de diseño reconocidos, principios SOLID y buenas prácticas de Clean Code, orientada a mejorar la seguridad, mantenibilidad, escalabilidad y organización estructural del sistema.

La propuesta contempla mejoras tanto en el backend desarrollado en Spring Boot como en el frontend implementado en Angular, así como una visión futura de descomposición del monolito hacia una arquitectura basada en microservicios con bounded contexts claramente definidos.

Este documento no implica la modificación directa del código fuente original, sino que presenta una propuesta técnica fundamentada que ejemplifica cómo debería refactorizarse el sistema para cumplir con estándares modernos de arquitectura de software.

Actividad 3.1 - Propuesta de Patrones de Diseño

3.1.1 Propuesta Backend (Spring Boot)

Archivo analizado: *EncuestaController.java*

A continuación, se presentan los anti-patrones identificados en el backend del sistema, indicando las líneas del código afectadas y su impacto técnico.

Con base en estos hallazgos, se propone la aplicación de patrones de diseño que permitan mejorar la arquitectura, seguridad, mantenibilidad y cumplimiento de principios SOLID del sistema.

Layered Architecture (MVC) - Arquitectura

Problema identificado:

- Líneas: Todo el archivo

El archivo EncuestaController.java concentra múltiples responsabilidades: manejo de solicitudes HTTP, validaciones, construcción de consultas SQL y acceso directo a la base de datos. Esto viola el principio de Responsabilidad Única (SRP), ya que la clase asume distintos motivos de cambio, aumentando el acoplamiento y la complejidad del sistema.

Impacto:

- Dificulta la mantenibilidad del código
- Complica la realización de pruebas unitarias
- Incrementa el acoplamiento entre capas
- Hace el sistema más rígido ante cambios futuros

Patrón propuesto:

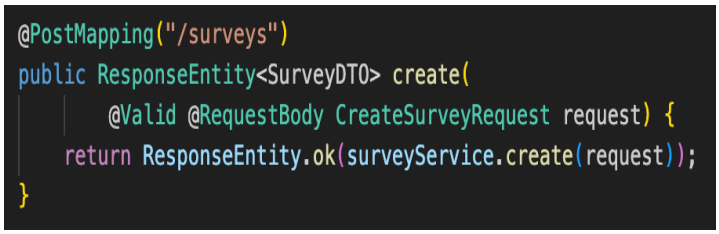
- Layered Architecture (Controller → Service → Repository)
- Estructura:
    - controller/
    - service/
    - repository/
    - entity/
    - dto/

Justificación arquitectónica:

La arquitectura en capas propone separar el sistema en componentes con responsabilidades bien definidas:

- Controller: gestión de solicitudes HTTP
- Service: lógica de negocio
- Repository: acceso a datos

Esta separación reduce el acoplamiento, mejora la cohesión y facilita pruebas unitarias mediante inyección de dependencias. Además, fortalece el cumplimiento de los principios SRP y DIP de SOLID, y prepara el sistema para una futura evolución arquitectónica.

Código actual	Propuesta
	

```
@PostMapping("/crear")
public Map crear(@RequestBody Map body) {
    String sql = "INSERT INTO encuestas ...";
    jdbc().update(sql);
}
```

```
@PostMapping("/surveys")
public ResponseEntity<SurveyDTO> create(
        @Valid @RequestBody CreateSurveyRequest request) {
    return ResponseEntity.ok(surveyService.create(request));
}
```

En el código actual, el controlador ejecuta directamente consultas SQL, mezclando responsabilidades de presentación, lógica y persistencia. En la propuesta, el controlador delega la operación al `surveyService`, logrando una estructura modular, más mantenible y alineada con buenas prácticas de arquitectura.

Repository Pattern + Spring Data JPA - Arquitectura

Problema identificado:

- Líneas: 41, 78, 101–105, 109

En el archivo `EncuestaController.java` se construyen consultas SQL mediante concatenación directa de parámetros recibidos del usuario. Esta práctica expone el sistema a vulnerabilidades de seguridad y acopla la lógica de negocio con detalles de persistencia.

Impacto:

- Vulnerabilidad a ataques de **SQL Injection**
- Código difícil de mantener y propenso a errores
- Alto acoplamiento entre lógica y acceso a datos

Patrón propuesto:

- Repository Pattern con Spring Data JPA

Justificación arquitectónica:

El Repository Pattern permite abstraer el acceso a datos mediante una interfaz que encapsula las operaciones de persistencia. Al utilizar Spring Data JPA, las consultas se gestionan

internamente mediante sentencias preparadas (prepared statements), eliminando el riesgo de SQL Injection.

Además, esta aproximación reduce código repetitivo (boilerplate), desacopla la lógica de negocio del acceso a datos y facilita la implementación de pruebas unitarias mediante la simulación del repositorio.

<table>
<tr><th>Código actual</th><th>Propuesta</th></tr>
<tr>
<td>

```
String sql = "SELECT * FROM encuestas WHERE id = " + id;
```

</td>
<td>

```
@Repository
public interface SurveyRepository
        extends JpaRepository<Survey, Long> {
}
```

Uso en service:

```
public SurveyDTO findById(Long id) {
    Survey survey = repository.findById(id)
        .orElseThrow(() -> new SurveyNotFoundException());
    return mapper.toDTO(survey);
}
```

</td>
</tr>
</table>

En el código actual, la consulta se construye concatenando directamente el identificador recibido, lo que representa un riesgo de inyección SQL y mezcla responsabilidades dentro del controlador. En la propuesta, el acceso a datos se delega al repositorio, el cual gestiona las consultas de manera segura y desacoplada, mejorando la seguridad, mantenibilidad y claridad estructural del sistema.

DTO Pattern - Arquitectura

Problema identificado:

- Líneas: 30, 46, 72, 87

En el controlador se utilizan objetos `Map` genéricos para recibir y retornar información en los endpoints. Esta práctica elimina el tipado fuerte y carece de un contrato claro para los datos intercambiados entre cliente y servidor.

Impacto:

- No existe validación automática de los datos
- No hay un contrato claro de la API
- Mayor probabilidad de errores en tiempo de ejecución
- Código menos legible y mantenible

Patrón propuesto:

- DTO Pattern (Data Transfer Object)

Justificación arquitectónica:

El patrón DTO define estructuras de datos específicas para representar la información que se recibe y se envía a través de la API. Esto permite establecer un contrato explícito entre cliente y servidor, mejorar el tipado fuerte y habilitar validaciones mediante anotaciones como `@NotBlank` y `@Size`.

Además, favorece la claridad del modelo expuesto, reduce errores en tiempo de ejecución y facilita la documentación automática de la API (por ejemplo, con OpenAPI/Swagger).

Código actual	Propuesta

`public Map crear(@RequestBody Map body)`	`public record CreateSurveyRequest(`
	`  @NotBlank @Size(min = 3) String pregunta`
	`) {}`
	
	`public record SurveyDTO(`
	`  Long id,`
	`  String pregunta,`
	`  int siCount,`
	`  int noCount`
	`) {}`

En el código actual, el uso de `Map` obliga a realizar conversiones manuales y validaciones imperativas, aumentando el riesgo de errores. En la propuesta, los DTOs definen explícitamente la estructura de entrada y salida, permiten validación declarativa mediante anotaciones y establecen un contrato claro de datos, mejorando la robustez y claridad del sistema.

Global Exception Handler - Manejo de Errores

Problema identificado:

- Líneas: 51–53, 66–68, 82–84, 112–114

En los distintos métodos del controlador se capturan excepciones genéricas utilizando `printStackTrace()` y retornando `null` como respuesta. Esta práctica no proporciona información estructurada al cliente y expone detalles internos del sistema.

Impacto:

- Respuestas HTTP inconsistentes
- Posible fuga de información sensible en logs
- Mala experiencia para el cliente
- Código repetitivo y poco mantenible

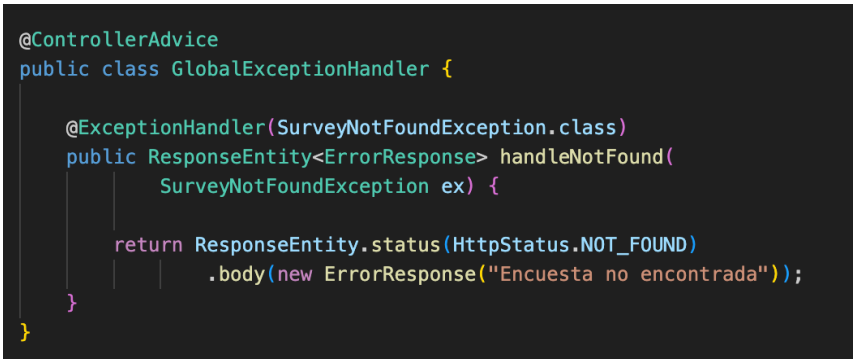
Patrón propuesto:

- Global Exception Handler con @ControllerAdvice

Justificación arquitectónica:

El uso de un manejador global de excepciones permite centralizar el tratamiento de errores en una sola clase, separando esta responsabilidad del controlador.

Esta aproximación garantiza respuestas HTTP consistentes, mejora la seguridad al evitar la exposición de detalles internos y reduce la duplicación de código. Además, contribuye a una arquitectura más limpia y alineada con buenas prácticas REST.

Código actual	Propuesta
	

En el código actual, cada método maneja las excepciones de forma local y devuelve `null`, lo que genera respuestas ambiguas e inconsistentes. En la propuesta, las excepciones se gestionan de manera centralizada, permitiendo definir respuestas HTTP claras y estandarizadas, mejorando la seguridad y la mantenibilidad del sistema.

Externalized Configuration - Seguridad

Problema identificado:

- Líneas: 16–18

Las credenciales de conexión a la base de datos se encuentran definidas directamente dentro del código fuente como valores hardcodeados. Esta práctica acopla la configuración al código y expone información sensible.

Impacto:

- Riesgo de seguridad al exponer credenciales
- Dificultad para manejar múltiples ambientes (dev, test, prod)
- Bajo nivel de flexibilidad en despliegues
- Violación del principio de configuración externa

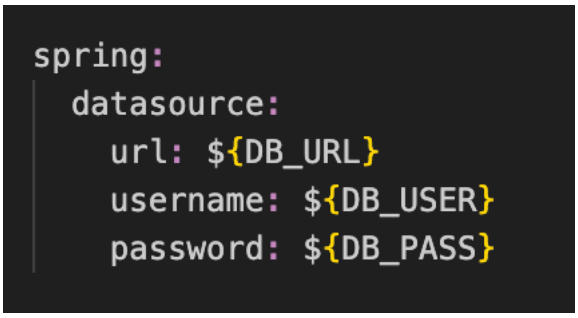
Patrón propuesto:

- Externalized Configuration (Spring Boot application.yml)

Justificación arquitectónica:

La configuración externalizada permite separar los parámetros sensibles y específicos del entorno del código fuente de la aplicación. Al utilizar variables de entorno dentro del archivo `application.yml`, se mejora la seguridad, se facilita la gestión de múltiples ambientes y se simplifica el despliegue en contenedores como Docker o en arquitecturas de microservicios.

Esta práctica promueve una arquitectura más flexible, segura y alineada con estándares modernos de desarrollo.

Código actual	Propuesta
	

En el código actual, las credenciales están incrustadas directamente en la clase, lo que implica modificar el código para cualquier cambio de entorno. En la propuesta, los valores se externalizan mediante variables de entorno, permitiendo cambiar configuraciones sin alterar el código fuente, mejorando la seguridad y la portabilidad del sistema.

DataSource + Connection Pool - Rendimiento

Problema identificado:

- Líneas: 20–27

El método `jdbc()` crea manualmente un `DriverManagerDataSource` cada vez que se invoca, generando una nueva configuración de conexión para cada request. Esta implementación evita el uso de un pool de conexiones y acopla la gestión de recursos directamente al controlador.

Impacto:

- Degradación del rendimiento bajo carga
- Alto consumo de recursos
- Gestión ineficiente de conexiones
- Código innecesariamente repetitivo

Patrón propuesto:

- Dependency Injection + DataSource gestionado por Spring

Justificación arquitectónica:

Spring Boot configura automáticamente un `DataSource` con pool de conexiones (por defecto, HikariCP), permitiendo la reutilización eficiente de conexiones a la base de datos. Al delegar la gestión del `DataSource` al contenedor de Spring e inyectar `JdbcTemplate` mediante constructor, se mejora el rendimiento, se optimiza el uso de recursos y se desacopla la infraestructura de la lógica de la aplicación.

Esta práctica también fortalece el principio de Inversión de Dependencias (DIP), al depender de componentes gestionados por el framework en lugar de instanciarlos manualmente.

<table>
<tr><th>Código actual</th><th>Propuesta</th></tr>
<tr>
<td>

```
private JdbcTemplate jdbc() {
    DriverManagerDataSource ds = new DriverManagerDataSource();
    ds.setDriverClassName("org.postgresql.Driver");
    ds.setUrl(URL);
    ds.setUsername(USER);
    ds.setPassword(PASS);
    return new JdbcTemplate(ds);
}
```

Uso en cada método:

```
jdbc().update(sql);
```

</td>
<td>

```
spring:
  datasource:
    url: jdbc:postgresql://db:5432/encuestas
    username: ${DB_USER}
    password: ${DB_PASS}
    driver-class-name: org.postgresql.Driver
```

```
@RestController
@RequestMapping("/surveys")
public class SurveyController {

    private final JdbcTemplate jdbcTemplate;

    public SurveyController(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

</td>
</tr>
</table>

En el código actual, se crea manualmente una nueva configuración de conexión en cada invocación, lo que impide la reutilización eficiente de recursos. En la propuesta, Spring gestiona el `DataSource` y el pool de conexiones de forma automática, permitiendo reutilizar conexiones y mejorar significativamente el rendimiento y la escalabilidad del sistema.

3.1.2 Propuesta Frontend (Angular)

Archivos analizados:

- crear.component.ts
- encuesta.component.ts
- respuestas.component.ts

- home.component.ts

A continuación, se presentan los anti-patrones identificados en el frontend del sistema, indicando los archivos afectados y su impacto técnico. Con base en estos hallazgos, se propone la aplicación de patrones y buenas prácticas propias de Angular para mejorar la arquitectura, mantenibilidad, seguridad y tipado del sistema.

Service Pattern (Separación de acceso HTTP) - Arquitectura

Problema identificado:

Archivos:

- crear.component.ts (línea 21)
- encuesta.component.ts (línea 22)
- respuestas.component.ts (línea 18)

Los componentes realizan directamente las llamadas HTTP utilizando `HttpClient`, integrando lógica de presentación con comunicación remota. Esta implementación mezcla responsabilidades dentro del componente, el cual debería enfocarse únicamente en la interacción con la vista.

Impacto:

- Violación de separación de responsabilidades
- Dificulta pruebas unitarias
- Repetición de lógica HTTP
- Mayor acoplamiento entre UI y backend

Patrón propuesto:

- Service Pattern (EncuestaService)

Justificación arquitectónica:

En Angular, la capa de servicios permite encapsular la comunicación con el backend y desacoplarla de los componentes visuales. Esta separación favorece una arquitectura más modular, donde los componentes se enfocan en la representación de datos y los servicios en la lógica de acceso remoto.

Además, facilita la simulación de dependencias (mocks) en pruebas unitarias, mejora la reutilización del código y fortalece el principio de Responsabilidad Única (SRP), alineándose con una arquitectura más limpia y mantenible.

Código actual	Propuesta

`this.http.post(this.url + '/crear', { pregunta: this.pregunta })`	`@Injectable({ providedIn: 'root' })`
	`export class EncuestaService {`
	
	`  constructor(private http: HttpClient) {}`
	
	`  crear(pregunta: string) {`
	`  return this.http.post('/crear', { pregunta });`
	`  }`
	`}`
	
	`#Uso en componente`
	
	`this.encuestaService.crear(this.pregunta)`
	`  .subscribe(...)`

En la implementación actual, cada componente administra sus propias llamadas HTTP. En la propuesta, la comunicación se centraliza en un servicio inyectable, reduciendo el acoplamiento y mejorando la coherencia estructural del frontend.

Externalized Configuration (environment.ts) - Arquitectura

Problema identificado:

Archivos:

- crear.component.ts (línea 18)
- encuesta.component.ts (línea 18)
- respuestas.component.ts (línea 15)

La URL del backend se encuentra definida manualmente en cada componente, generando duplicación de configuración y dependencia directa del entorno de ejecución.

Impacto:

- Dificultad para cambiar entre ambientes (desarrollo, pruebas, producción)
- Riesgo de inconsistencias
- Mayor mantenimiento ante cambios de infraestructura

Patrón propuesto:

- Externalized Configuration mediante environment.ts

Justificación arquitectónica:

Angular permite definir configuraciones específicas por entorno a través de archivos `environment.ts`. Centralizar la URL del backend en estos archivos desacopla la configuración del código funcional y facilita el despliegue en distintos ambientes.

Esta práctica mejora la mantenibilidad, reduce errores humanos y se alinea con principios de configuración externa utilizados en arquitecturas modernas.

Código actual	Propuesta

`url: any = 'http://localhost:8080';`	`# environment.ts`
	`export const environment = {`
	`  apiUrl: 'http://localhost:8080'`
	`};`
	
	`# Uso en componente`
	`import { environment } from '../environments/environment';`
	`this.http.get(environment.apiUrl + '/encuestas')`

En el código actual, la URL está repetida en varios componentes. En la propuesta, la configuración se centraliza, permitiendo modificar el entorno sin alterar múltiples archivos del sistema.

Strong Typing + Interfaces (SOLID / Type Safety) - Tipado

Problema identificado:

Archivos: múltiples componentes

Uso generalizado del tipo `any` en variables y respuestas HTTP, eliminando contratos explícitos de datos.

Impacto:

- Pérdida de validación en tiempo de compilación
- Mayor probabilidad de errores en ejecución
- Baja claridad del modelo de datos

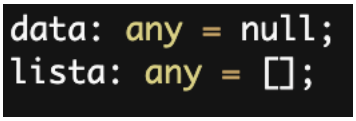
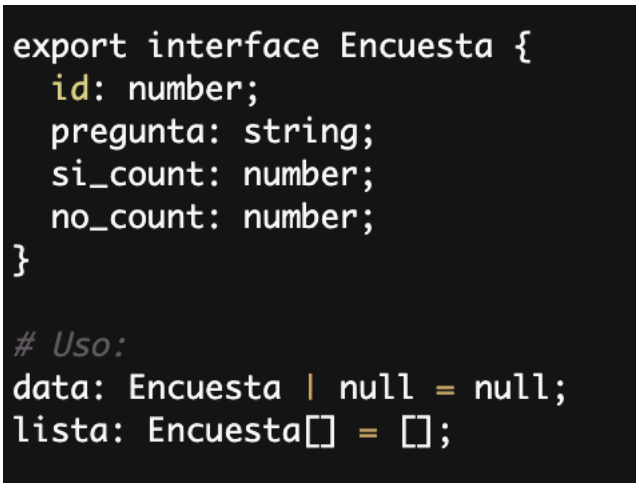
Patrón propuesto:

- Uso de Interfaces Tipadas

Justificación arquitectónica:

TypeScript ofrece tipado estático que permite definir contratos explícitos para los datos que circulan en la aplicación. Al utilizar `any`, se pierde el beneficio principal del lenguaje, debilitando la robustez del sistema.

Definir interfaces fortalece el control en tiempo de compilación, mejora la legibilidad del código y reduce la probabilidad de errores en runtime. Esta práctica refuerza principios de diseño orientado a contratos y mejora la calidad general del frontend.

Código actual	Propuesta
	

El uso de `any` desactiva el sistema de tipos. Con interfaces definidas, el compilador valida estructuras de datos y previene inconsistencias antes de ejecutar la aplicación.

Eliminación de Manipulación Directa del DOM – Clean Code

Problema identificado:

Archivos:

- home.component.ts (línea 18)
- crear.component.ts (línea 29)
- encuesta.component.ts

Uso de `document.getElementById()` para modificar elementos del DOM manualmente.

Impacto:

- Ruptura del enfoque declarativo de Angular
- Código frágil ante cambios en el template
- Dificultad para pruebas automatizadas

Patrón propuesto:

- Data Binding y Template Binding de Angular

Justificación arquitectónica:

Angular está diseñado bajo un modelo declarativo basado en binding y detección automática de cambios. Manipular el DOM directamente evita que el framework controle el ciclo de vida de la vista, introduciendo inconsistencias y reduciendo la coherencia arquitectónica.

El uso de interpolación y property binding permite que el framework gestione la actualización del DOM de forma reactiva y estructurada, manteniendo la cohesión con el modelo del framework.

Código actual	Propuesta
`const t = document.getElementById('tituloEncuesta');` `if (t) { t.innerHTML = this.data.pregunta + ' ???'; }`	`<h2>{{ data?.pregunta }} ???</h2>`

La implementación actual modifica el DOM manualmente. En la propuesta, Angular actualiza la vista automáticamente mediante binding declarativo, reduciendo complejidad y mejorando mantenibilidad.

Reemplazo de Polling Manual por RxJS - Rendimiento

Problema identificado:

Archivo: encuesta.component.ts (líneas 27–29)

Uso de `setInterval` para actualizar datos periódicamente.

Impacto:

- Riesgo de fugas de memoria
- Gestión manual del ciclo de vida del temporizador
- Código imperativo poco alineado con el modelo reactivo

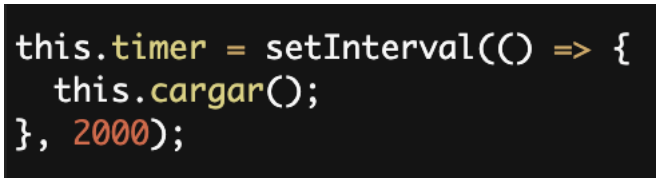
Patrón propuesto:

- RxJS (interval + switchMap + takeUntil)

Justificación arquitectónica:

Angular se basa en un modelo reactivo soportado por RxJS. Utilizar `setInterval` rompe esta coherencia y obliga a gestionar manualmente la cancelación del temporizador. Al emplear operadores como `interval`, `switchMap` y `takeUntil`, se integra el flujo periódico dentro del ecosistema reactivo de Angular, permitiendo una cancelación controlada y evitando fugas de memoria.

Esta solución mejora la escalabilidad, estabilidad y coherencia arquitectónica del frontend.

Código actual	Propuesta
	

El enfoque actual usa temporizadores imperativos que requieren limpieza manual. La propuesta integra el polling en un flujo reactivo controlado, mejorando robustez y alineación con el framework.

Actividad 3.2 - Propuesta de Principios Clean Code

En esta sección se identifican violaciones a principios de Clean Code presentes en el backend (Spring Boot) y frontend (Angular), proponiendo mejoras orientadas a incrementar la legibilidad, mantenibilidad, cohesión y robustez del sistema.

Las mejoras aplicadas se alinean con principios de Clean Code y con fundamentos de SOLID, fortaleciendo la calidad estructural del proyecto.

Principio	Código actual (archivo + línea / problema)	Propuesta de mejora
Nombres significativos	*EncuestaController.java* (30, 46) → Uso de Map r, Map body, variable e sin significado semántico claro	Reemplazar por SurveyDTO response, CreateSurveyRequest request, Survey survey para expresar intención y definir contratos explícitos
DRY (Don't Repeat Yourself)	*EncuestaController.java* (41, 78, 101–105) → Validación pregunta.length() < 3 repetida en múltiples métodos	Centralizar validación en CreateSurveyRequest usando @NotBlank y @Size(min=3) validado automáticamente por Spring
Funciones pequeñas / SRP	*EncuestaController.java* (25–60) → Método crear() de más de 20 líneas mezclando validación, SQL y construcción de respuesta	Delegar lógica a SurveyService, separando validación, persistencia y mapeo en métodos específicos
No retornar null	*EncuestaController.java* (53, 68, 84, 114) → Uso de return null en bloques catch	Lanzar SurveyNotFoundException y manejarla globalmente con @ControllerAdvice
Configuración no limpia (Hardcode)	*EncuestaController.java* (16–18) → Credenciales embebidas en el código (USER = "espagueti")	Externalizar configuración en application.yml con variables de entorno (${DB_USER})
Creación repetida de objetos (baja eficiencia)	*EncuestaController.java* (20–27) → Creación de DriverManagerDataSource en cada request	Inyección de JdbcTemplate gestionado por Spring con pool de conexiones

Tipado débil (Frontend)	*encuesta.component.ts* (14), *crear.component.ts* (18) → Uso de any en respuestas HTTP	Definir interfaces Survey, VoteRequest y tipo `VoteOption = 'SI'
Baja cohesión en componentes Angular	*crear.component.ts* (21), *encuesta.component.ts* (22) → Llamadas HTTP directas en el componente	Centralizar comunicación en SurveyService (Service Pattern)
Uso imperativo en entorno reactivo	*respuestas.component.ts* (35) → Uso de setInterval() para polling manual	Reemplazar por interval() + switchMap() con RxJS para control reactivo

3.2.1 Snippets de código

Nombres significativos

EncuestaController.java (30, 46)

Código actual	Propuesta

`public Map crear(@RequestBody Map body) {`
`  Map r = new HashMap();`
`  r.put("error", "Pregunta muy corta");`
`}`
	`public SurveyDTO createSurvey(`
	`  @Valid @RequestBody CreateSurveyRequest request) {`
	`}`
	
	`# DTO tipado`
	`public record CreateSurveyRequest(`
	`  @NotBlank`
	`  @Size(min = 3)`
	`  String pregunta`
	`) {}`

Se reemplazó el uso de tipos genéricos (`Map`) y nombres ambiguos por clases y métodos con nombres descriptivos (`CreateSurveyRequest`, `SurveyDTO`, `createSurvey`). Esto mejora la legibilidad, la intención del código y permite entender su propósito sin necesidad de revisar su implementación.

DRY (Don't Repeat Yourself)

EncuestaController.java (41, 78, 101–105)

Código actual	Propuesta

`if (body == null || body.get("pregunta") == null || body.get("pregunta").toString().trim().length() < 3) { Map r = new HashMap(); r.put("error", "Pregunta muy corta"); return r; }`	`public record CreateSurveyRequest( @NotBlank @Size(min = 3) String pregunta ) {}`
	
	`# Y en el controller`
	`public SurveyDTO createSurvey( @Valid @RequestBody CreateSurveyRequest request)`

Se eliminó la lógica de validación repetida en múltiples endpoints trasladándola a un DTO con anotaciones de validación. Esto evita duplicación, reduce errores futuros y centraliza la responsabilidad de validación en un solo punto.

Funciones pequeñas / SRP

EncuestaController.java (25–60)

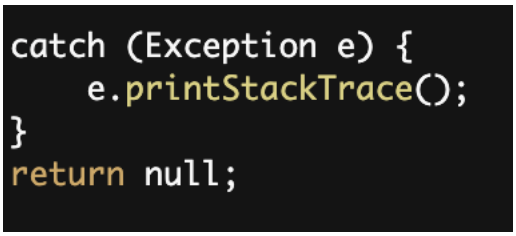
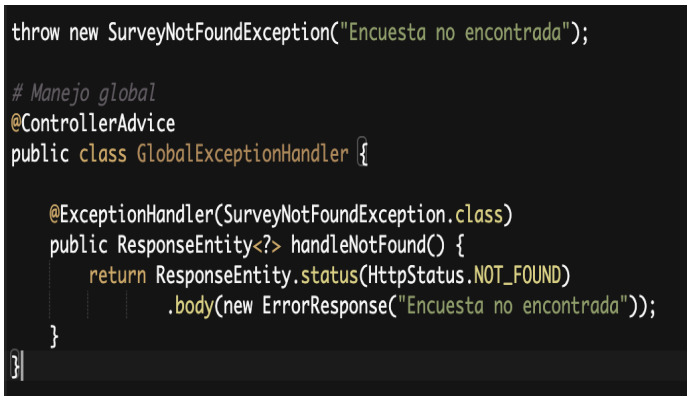
Código actual	Propuesta

`public Map crear(@RequestBody Map body) { // validación String sql = "INSERT INTO encuestas(pregunta, si_count, no_count) VALUES ('" + body.get("pregunta") + "', 0, 0)"; jdbc().update(sql); Integer id = jdbc().queryForObject("SELECT MAX(id) FROM encuestas", Integer.class); Map resp = new HashMap(); resp.put("id", id); return resp; }`	`public SurveyDTO createSurvey(CreateSurveyRequest request) { return surveyService.create(request); }`
	
	`# Servicio separado`
	`@Service`
	`public class SurveyService {`
	
	`public Survey create(CreateSurveyRequest request) { // lógica de persistencia aquí } }`

Se separó la lógica de negocio del controlador delegándola a un servicio. Esto cumple el principio de responsabilidad única, ya que el controlador ahora solo gestiona peticiones HTTP y el servicio contiene la lógica de persistencia.

No retornar null

EncuestaController.java (53, 68, 84, 114)

Código actual	Propuesta
	

Se reemplazó el retorno de `null` por el uso de excepciones personalizadas y un manejador global. Esto mejora el manejo de errores, evita posibles `NullPointerException` y hace el comportamiento del sistema más explícito.

Configuración no limpia (Hardcode)

EncuestaController.java (16–18)

Código actual	Propuesta
	

Se eliminaron credenciales y configuraciones incrustadas en el código, trasladándolas a variables de entorno. Esto mejora la seguridad, la mantenibilidad y permite configurar distintos entornos sin modificar el código fuente.

Creación repetida de objetos

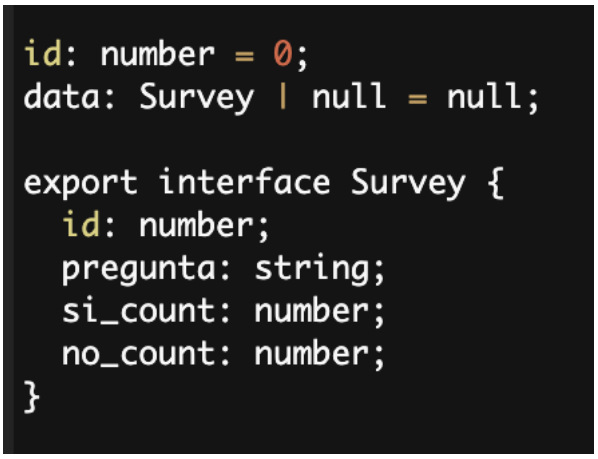
EncuestaController.java (20–27)

Código actual	Propuesta
	

Se eliminó la creación manual de `JdbcTemplate` en favor de inyección de dependencias. Esto mejora la cohesión, facilita pruebas unitarias y aprovecha el contenedor de Spring para la gestión de objetos.

Tipado débil (Frontend)

encuesta.component.ts (14)
crear.component.ts (18)

Código actual	Propuesta
	

Se reemplazó el uso de `any` por interfaces tipadas. Esto mejora la seguridad de tipos, previene errores en tiempo de ejecución y permite mayor soporte del compilador y del IDE.

Baja cohesión en componentes Angular

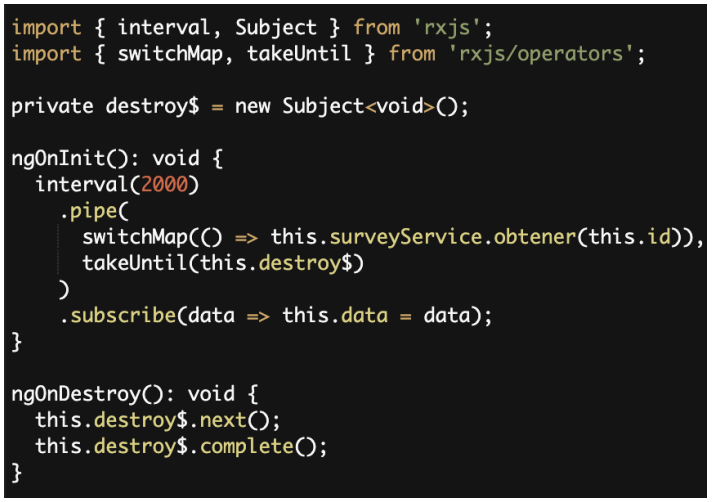
crear.component.ts (21)
encuesta.component.ts (22)

Código actual	Propuesta
	

Se extrajo la lógica HTTP de los componentes hacia un servicio dedicado. Esto mejora la cohesión, facilita pruebas y mantiene los componentes enfocados únicamente en la presentación.

Uso imperativo en entorno reactivo

encuesta.component.ts (27 aprox)

Código actual	Propuesta
	

Se reemplazó `setInterval` por un flujo reactivo con RxJS y manejo adecuado del ciclo de vida. Esto previene fugas de memoria y se alinea con el paradigma reactivo de Angular.

Actividad 3.3 - Diseño de Arquitectura de Microservicios

En esta sección se propone la descomposición del sistema monolítico en una arquitectura basada en microservicios, definiendo responsabilidades claras por dominio (Bounded Context). Cada microservicio es autónomo, mantiene su propia base de datos y se comunica con los demás mediante APIs REST o eventos, garantizando bajo acoplamiento, alta cohesión y escalabilidad independiente.}

Microservicio	Responsabilidad	Endpoints (Ejemplo)	Base de datos
api-gateway	Punto de entrada único al sistema. Enrutamiento dinámico hacia los microservicios, autenticación/autorización (JWT), CORS, rate limiting, logging y manejo de errores comunes.	Proxy dinámico: `/surveys/**` → survey-service `/votes/**` → voting-service `/auth/**` → identity-service `/notifications/**` → notification-service `/reports/**` → reporting-service	—

survey-service	Gestión de encuestas (CRUD) y ciclo de vida (borrador, publicada, cerrada). Administra preguntas y opciones. Es el dueño del dominio "Encuesta".	`POST /surveys`
		`GET /surveys`
		`GET /surveys/{id}`
		`PUT /surveys/{id}`
		`PATCH /surveys/{id}/status`
			PostgreSQL (surveys)

voting-service	Registro de votos y aplicación de reglas de negocio (validar encuesta activa, evitar duplicados). Calcula resultados simples (conteo por opción). Puede emitir eventos como `VoteRegistered`.	`POST /votes` `GET /results/{surveyId}`	PostgreSQL (votes)

identity-service	Gestión de usuarios, autenticación y autorización. Registro, login, emisión y validación de JWT, roles y permisos.	`POST /auth/register`
		`POST /auth/login`
		`GET /users/{id}`
		`GET /roles`
			PostgreSQL (identity)

notification-service	Envío de notificaciones (correo o push). Consume eventos del sistema como `SurveyPublished` o `VoteRegistered`. No contiene lógica de negocio, solo comunicación.	`POST /notifications/email` `POST /notifications/push`	PostgreSQL (notifications) *(opcional)*
reporting-service	Generación de reportes y analítica de negocio: encuestas más respondidas, participación por fechas, métricas agregadas. Puede consumir eventos para mantener vistas optimizadas.	`GET /reports/top-surveys` `GET /reports/activity?from=&to=`	PostgreSQL (reporting)

Relación con la Actividad 3.1

Cada microservicio propuesto mantiene internamente la arquitectura en capas definida en la Actividad 3.1 (Controller – Service – Repository). Los patrones de diseño como Repository, Service Layer y uso de DTO no desaparecen al pasar a microservicios, sino que se encapsulan dentro de cada bounded context, garantizando coherencia entre el diseño interno del backend y la arquitectura distribuida propuesta.

3.3.1 Componentes Transversales de Infraestructura

Componente	Propósito en la arquitectura
Service Discovery (Netflix Eureka / Consul)	Permite que los microservicios se registren y descubran dinámicamente sin depender de direcciones IP fijas. Facilita escalabilidad horizontal y balanceo de carga.
Config Server (Spring Cloud Config)	Centraliza la configuración de todos los servicios (URLs, credenciales, entornos), evitando hardcoding y permitiendo cambios sin recompilar cada servicio.
Message Broker (RabbitMQ / Apache Kafka)	Permite comunicación asíncrona basada en eventos entre microservicios (ej. `VoteRegistered`, `SurveyPublished`), reduciendo acoplamiento.
Circuit Breaker (Resilience4j)	Previene fallos en cascada cuando un servicio no responde, mejorando la resiliencia del sistema distribuido.
Centralized Logging (ELK Stack)	Centraliza logs de todos los servicios para monitoreo, trazabilidad y diagnóstico de errores.

Estos componentes no representan dominios de negocio, sino capacidades transversales necesarias para garantizar escalabilidad, resiliencia, configuración centralizada y observabilidad en una arquitectura de microservicios.

3.3.2 Arquitectura propuesta por capas – Diagramas C4

Diagrama C4 – Nivel 1 (Contexto)

El diagrama de contexto muestra el sistema como una única unidad desde una perspectiva externa. Se identifican los actores principales (por ejemplo, usuario final o administrador) y los sistemas externos que interactúan con la plataforma de encuestas. En este nivel no se detallan componentes internos, sino las relaciones de alto nivel entre el sistema y su entorno.

El objetivo de esta vista es comprender el alcance del sistema y cómo se integra dentro del ecosistema general, sin entrar en detalles técnicos de implementación.

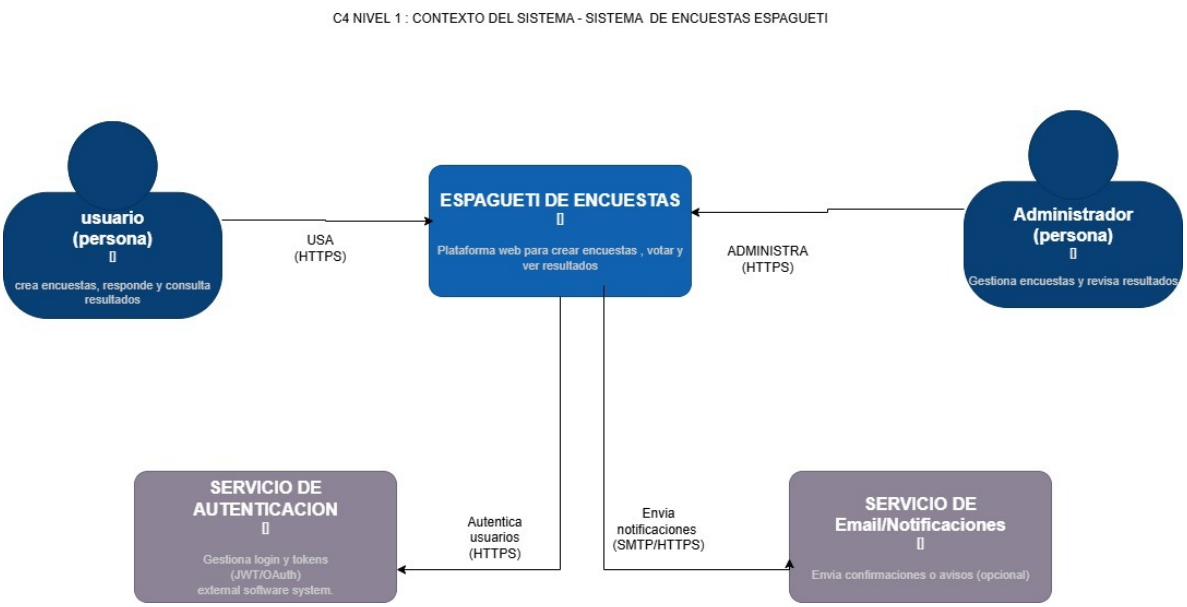


Diagrama C4 – Nivel 2 (Contenedores – Arquitectura Actual)

El diagrama de contenedores actual representa la arquitectura monolítica original del sistema. En esta versión, la aplicación backend concentra controladores, lógica de negocio y acceso a datos en un único contenedor que se comunica directamente con una base de datos compartida.

Esta estructura genera alto acoplamiento entre componentes, dificulta la escalabilidad independiente y aumenta el impacto de cambios internos. Además, cualquier fallo puede afectar a la totalidad del sistema.

Esta vista permite visualizar las limitaciones arquitectónicas que motivan la propuesta de refactorización hacia microservicios.

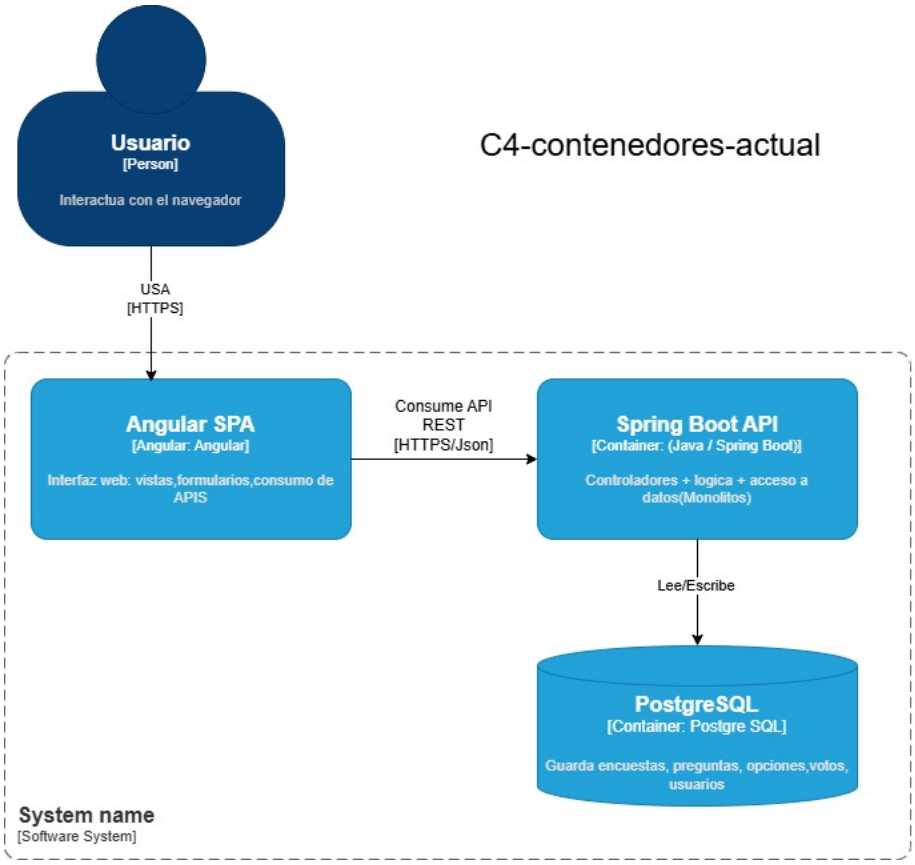
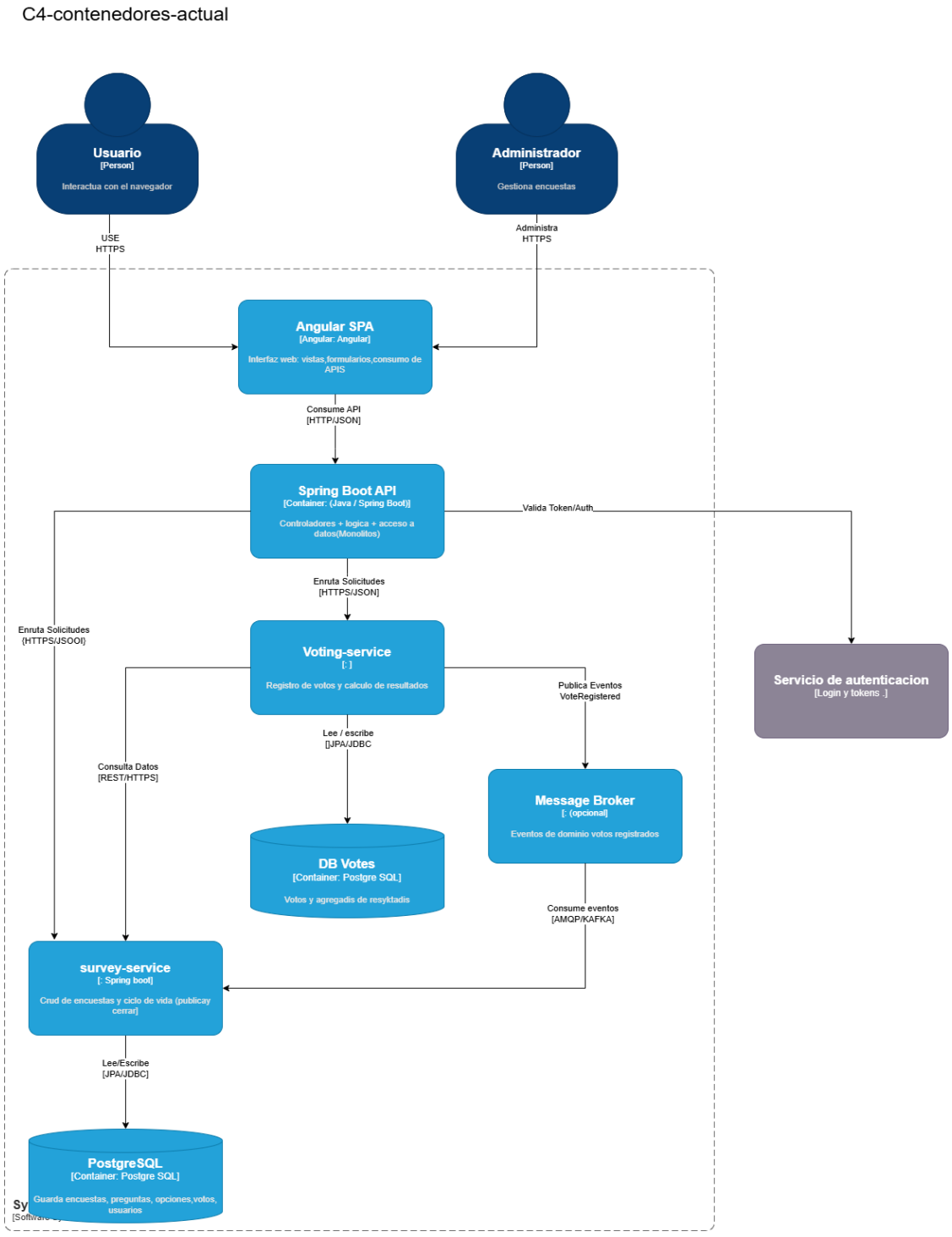


Diagrama C4 – Nivel 2 (Contenedores – Arquitectura Propuesta)

El diagrama de contenedores propuesto muestra la transición hacia una arquitectura basada en microservicios. El sistema se divide en servicios independientes organizados por dominio (Bounded Context), como survey-service, voting-service, identity-service, notification-service y reporting-service, coordinados a través de un API Gateway.

Cada microservicio es autónomo y mantiene su propia base de datos, respetando el principio de separación de responsabilidades y evitando el acceso directo entre esquemas. La comunicación entre servicios se realiza mediante APIs REST o eventos a través de un Message Broker, reduciendo el acoplamiento y permitiendo escalabilidad independiente.

Esta arquitectura mejora la mantenibilidad, resiliencia y capacidad de evolución del sistema en comparación con el monolito original.



En conjunto, los tres niveles del modelo C4 permiten visualizar la evolución del sistema desde una estructura monolítica hacia una arquitectura distribuida, manteniendo coherencia entre el diseño interno definido en la Actividad 3.1 y la descomposición estructural propuesta en esta sección.

Conclusiones de la Fase 3

Durante la Fase 3 se evidenció que la aplicación de patrones de diseño y principios de Clean Code no son conceptos aislados, sino complementarios. Los patrones como Repository, Service Layer y el uso de DTO permitieron estructurar el backend de manera organizada, promoviendo separación de responsabilidades, bajo acoplamiento y alta cohesión. A su vez, estos mismos principios se alinean con las buenas prácticas de Clean Code, como nombres significativos, eliminación de duplicación (DRY) y funciones con responsabilidad única.

La transición hacia una arquitectura basada en microservicios no reemplaza los patrones previamente definidos, sino que los encapsula dentro de cada bounded context. Cada microservicio mantiene internamente una arquitectura en capas, mientras que externamente opera de forma autónoma con su propia base de datos y comunicación desacoplada mediante APIs REST o eventos. Esto demuestra coherencia entre el diseño interno del sistema y su evolución hacia una arquitectura distribuida.

Asimismo, la inclusión de componentes transversales como Service Discovery, Config Server, Message Broker y mecanismos de resiliencia refuerza la visión de una arquitectura escalable y preparada para entornos productivos reales.

En conjunto, esta fase permitió comprender que una arquitectura sólida no depende únicamente de dividir el sistema en múltiples servicios, sino de aplicar correctamente principios de diseño y organización tanto a nivel interno como a nivel estructural.

Reflexión final:

¿Vale la pena refactorizar el sistema de Monolito a Microservicios?

La decisión de migrar de una arquitectura monolítica a microservicios no debe tomarse únicamente por tendencia tecnológica, sino en función del contexto, la complejidad y las necesidades reales del sistema.

En el caso del sistema “Espagueti de Encuestas”, el análisis realizado evidenció múltiples problemas estructurales: alto acoplamiento, falta de separación de responsabilidades, dificultades para escalar componentes de manera independiente y vulnerabilidades de seguridad. Desde este punto de vista, una arquitectura basada en microservicios aporta beneficios claros como:

- Escalabilidad independiente por dominio.
- Mejor aislamiento de fallos.
- Mayor mantenibilidad en equipos grandes.
- Despliegue independiente de funcionalidades.
- Mejor alineación con bounded contexts del dominio.

Sin embargo, también es importante reconocer que los microservicios introducen mayor complejidad operativa: gestión de red, latencia, monitoreo distribuido, consistencia de datos y mayor carga DevOps. Para sistemas pequeños, con bajo tráfico o equipos reducidos, un monolito bien estructurado (modular y con arquitectura en capas) puede ser suficiente y más eficiente.

Por lo tanto, la refactorización sí puede ser valiosa si el sistema proyecta crecimiento, aumento de usuarios o ampliación funcional significativa. De lo contrario, una mejora del monolito hacia una arquitectura modular limpia podría ser una alternativa más costo-efectiva.

En conclusión, la migración a microservicios no es obligatoria, pero en el contexto analizado representa una oportunidad estratégica de evolución arquitectónica si se justifica por escalabilidad y crecimiento futuro.