Ingeniería del Software II

TP 3 - Despliegue y Clases

Docentes a cargo:
Victor Valotto, Cielo Godoy

Alumnas:
Inderkumer, Priscila
Murzi, Ana Sol

13 - OCT - 2025

1. Objetivo

Establecer el puente entre los modelos de arquitectura lógica (componentes y clases) y los modelos de infraestructura física (despliegue).

2. Fechas de trabajo

Semana del 06/10/2025 al 17/10/2025, inclusive.

3. Contextualización y estado de situación

A esta altura, los grupos ya:

- Identificaron requerimientos funcionales y no funcionales.
- Definieron una arquitectura preliminar de alto nivel con el modelo C4 (al menos hasta la vista de componentes).
- Modelaron comportamiento con diagramas de secuencia.

El siguiente paso es profundizar en el diseño detallado de su solución, vinculando:

- UML de clases (estructura interna de los componentes),
- UML de despliegue (infraestructura y asignación de software a hardware)

4. Directivas

1. Revisión teórica. Repasar los conceptos de:
    - Diagrama de clases UML

        Es una representación estática del código fuente, donde se muestran las relaciones entre las clases, los atributos y los métodos de cada clase

        Cada elemento y cada vínculo en el diagrama de clases tiene una interpretación específica, y todas las reglas sintácticas debe ser respetadas

    - Diagrama de despliegue UML

        Tiene como objetivo principal representar y visualizar la distribución física del sistema

        Se representan dispositivos, hardware y los vínculos entre ellos, incluyendo conexiones físicas como cableado y conexiones inalámbricas como Wi-Fi, satélite, entre otras

        El propósito fundamental de este diagrama es comprender qué hardware se necesita para la solución de software que se está construyendo

2. Ejercicio de análisis
    - **Explicar qué comunica cada uno de estos diagramas y qué diferencia hay entre ellos.**

        Cada uno de estos diagramas comunican:

        - Diagrama de clases UML

            Los bloques de construcción lógicos del sistema y cómo se relacionan entre sí en el código
        - Diagrama de despliegue UML

            Cómo el sistema se distribuye físicamente en servidores, contenedores y redes

        Ambos diagramas son diferentes entre sí, en donde las diferencias rigen en:

Aspecto en que difieren	Diagrama de clases UML	Diagrama de despliegue UML
Naturaleza	Representación estática del código fuente	Representación de la distribución física del sistema
Qué muestra	Clases, atributos, métodos y relaciones entre clases	Dispositivos, hardware y vínculos físicos/inalámbricos
Propósito	Definir comportamientos y responsabilidades del software	Comprender qué hardware se necesita para la solución
Elementos	Clases, interfaces, asociaciones, composiciones, agregaciones, herencias	Nodos (servidores, dispositivos), artefactos, conexiones físicas
Sintaxis	Reglas estrictas de UML (cada elemento tiene interpretación específica)	Representación de topología física y comunicaciones
Representa	Código (estructura lógica)	Procesos e infraestructura (no representa código)

- **Justificar por qué es importante mantener la trazabilidad entre C4 → Clases → Despliegue.**

  La trazabilidad es el hilo conductor que garantiza coherencia entre la arquitectura conceptual, su implementación en código y su materialización física.
  Las razones por las que es importante mantener la trazabilidad entre C4 → Clases → Despliegue son:

  1. Garantiza que lo diseñado en C4 sea lo implementado (mostrado en las clases) y lo desplegado (mostrado en la infraestructura)
  2. Las decisiones arquitectónicas se materializan consistentemente en todos los niveles
  3. Las relaciones de diseño se validan contra la infraestructura real
  4. Los cambios fluyen ordenadamente: C4 → Clases → Despliegue
  5. Cada nivel valida la factibilidad del anterior
  6. Se evitan decisiones aisladas por equipo que generan inconsistencias
  7. Se detectan inconsistencias antes de implementar
  8. Se sabe qué clases con datos sensibles están en qué servidores
  9. Se valida que la infraestructura soporta las necesidades del código
  10. Se identifican componentes críticos para replicación

3. Aplicación a su proyecto

A partir de la solución que se viene diseñando:

- **C4 (revisión): ajustar o completar su vista de componentes.**
- **Clases: para al menos tres componentes críticos (ej. autenticación, gestión de datos, procesamiento principal), desarrollar el diagrama de clases detallado.**
  Los componentes críticos de nuestro sistema son:

- Autenticación
  Código UML

```
@startuml
title Componente de Autenticación

package "Autenticación" {
  class AuthController <<Controller>> {
    - authService: AuthService
    + registrar(datos: UsuarioDTO): Response
    + login(credenciales: CredencialesDTO): TokenResponse
    + logout(token: String): Response
    + refrescarToken(token: String): TokenResponse
  }

  class AuthService <<Service>> {
    - usuarioRepository: UsuarioRepository
    - passwordEncoder: PasswordEncoder
    - tokenGenerator: TokenGenerator
    + registrarUsuario(datos: UsuarioDTO): Usuario
    + autenticar(email: String, password: String): Token
    + validarToken(token: String): Boolean
    + verificarPermisos(usuario: Usuario, recurso: String): Boolean
    + cambiarContraseña(usuario: Usuario, nuevaPassword: String): Boolean
  }

  class Usuario <<Entity>> {
    - id: Long
    - nombre: String
    - apellido: String
    - email: String
    - passwordHash: String
    - telefono: String
    - direccion: String
    - rol: RolUsuario
    - fechaRegistro: Date
    - activo: Boolean
    + validarEmail(): Boolean
    + tienePermiso(recurso: String): Boolean
    + esAdministrador(): Boolean
  }

  class TokenGenerator <<Component>> {
    - secretKey: String
    - expirationTime: Long
    + generarToken(usuario: Usuario): String
    + validarToken(token: String): Boolean
    + extraerUsuario(token: String): Usuario
    + estaExpirado(token: String): Boolean
  }

  enum RolUsuario {
    DUEÑO
```

```
    VECINO
    MUNICIPALIDAD
    VETERINARIA
    ADMINISTRADOR
  }

  AuthController --> AuthService : usa
  AuthService --> Usuario : gestiona
  AuthService --> TokenGenerator : utiliza
  Usuario --> RolUsuario : tipo
}

@enduml
```

UML-Autenticacion

- Gestión de Mascotas

```
@startuml
title Componente de Gestión de Datos de Mascotas

package "Gestión de Mascotas" {
  class MascotaController <<Controller>> {
    - mascotaService: MascotaService
    + registrarMascota(datos: MascotaDTO): Response
    + actualizarMascota(id: Long, datos: MascotaDTO): Response
    + obtenerMascota(id: Long): MascotaDTO
    + listarMascotasPorDueño(idDueño: Long): List<MascotaDTO>
    + eliminarMascota(id: Long): Response
  }

  class MascotaService <<Service>> {
    - mascotaRepository: MascotaRepository
    - validador: ValidadorMascota
    - archivoService: ArchivoService
    + crearMascota(datos: MascotaDTO, dueño: Usuario): Mascota
    + actualizarDatos(id: Long, datos: MascotaDTO): Mascota
    + obtenerPorId(id: Long): Mascota
    + obtenerPorDueño(dueño: Usuario): List<Mascota>
    + validarDatos(mascota: Mascota): Boolean
    + cargarFoto(id: Long, foto: File): String
  }

  class Mascota <<Entity>> {
    - id: Long
    - nombre: String
    - especie: EspecieMascota
    - raza: String
    - edad: Integer
    - sexo: String
    - color: String
```

```
  - tamano: TamanoMascota
  - caracteristicasDistintivas: String
  - numeroChip: String
  - urlFoto: String
  - dueño: Usuario
  - fechaRegistro: Date
  - activa: Boolean
  + tieneChip(): Boolean
  + obtenerEdadEnMeses(): Integer
  + esPerro(): Boolean
  + esGato(): Boolean
  + coincideCaracteristicas(otra: Mascota): Double
}

class CaracteristicaFisica {
  - tipo: String
  - valor: String
}

enum EspecieMascota {
  PERRO
  GATO
  AVE
  CONEJO
  OTRO
}

enum TamanoMascota {
  PEQUEÑO
  MEDIANO
  GRANDE
  MUY_GRANDE
}

MascotaController --> MascotaService : usa
MascotaService --> Mascota : gestiona
Mascota --> Usuario : pertenece
Mascota --> EspecieMascota : tipo
Mascota --> TamanoMascota : tipo
Mascota --> CaracteristicaFisica : composición
}

@enduml
```

UML - Gestión de Datos de Mascotas

- Reportes

```
@startuml
title Componente de Reportes y Alertas

package "Reportes y Alertas" {
  class ReporteController <<Controller>> {
    - reporteService: ReporteService
    + reportarPerdida(datos: ReporteDTO): Response
    + reportarAvistamiento(datos: AvistamientoDTO): Response
    + actualizarEstado(id: Long, estado: EstadoReporte): Response
    + obtenerReporte(id: Long): ReporteDTO
    + buscarCoincidencias(id: Long): List<CoincidenciaDTO>
    + listarReportesActivos(): List<ReporteDTO>
  }

  class ReporteService <<Service>> {
    - reporteRepository: ReporteRepository
    - matchingEngine: MatchingEngine
    - notificacionService: NotificacionService
    - geolocalizacionService: GeolocalizacionService
    + crearReportePerdida(datos: ReporteDTO): ReportePerdida
    + registrarAvistamiento(datos: AvistamientoDTO): Avistamiento
    + buscarCoincidencias(reporte: ReportePerdida): List<Coincidencia>
    + cambiarEstado(reporte: ReportePerdida, nuevoEstado: EstadoReporte): void
    + notificarCoincidencia(coincidencia: Coincidencia): void
  }

  class ReportePerdida <<Entity>> {
    - id: Long
    - mascota: Mascota
    - fechaPerdida: Date
    - ubicacionPerdida: Ubicacion
    - descripcion: String
    - reportante: Usuario
    - estado: EstadoReporte
    - fechaCreacion: Date
    - avistamientos: List<Avistamiento>
    + estaActivo(): Boolean
    + agregarAvistamiento(av: Avistamiento): void
    + cerrarReporte(): void
  }

  class Avistamiento <<Entity>> {
    - id: Long
    - reportePerdida: ReportePerdida
    - ubicacion: Ubicacion
    - fechaAvistamiento: Date
    - descripcion: String
    - reportante: Usuario
    - foto: String
    - verificado: Boolean
    + marcarComoVerificado(): void
```

```
    + calcularDistanciaAlReporte(): Double
  }

  class Coincidencia <<Entity>> {
    - id: Long
    - reportePerdida: ReportePerdida
    - mascotaEncontrada: Mascota
    - porcentajeSimilitud: Double
    - factoresCoincidentes: List<String>
    - fechaDeteccion: Date
    - revisada: Boolean
    + esAltaCoincidencia(): Boolean
    + marcarComoRevisada(): void
  }

  class MatchingEngine <<Component>> {
    - umbralCoincidencia: Double
    + calcularSimilitud(mascota1: Mascota, mascota2: Mascota): Double
    + encontrarCandidatos(reporte: ReportePerdida): List<Mascota>
  }

  enum EstadoReporte {
    ACTIVO
    EN_SEGUIMIENTO
    RESUELTO
    CERRADO
    CANCELADO
  }

  ReporteController --> ReporteService : usa
  ReporteService --> ReportePerdida : gestiona
  ReporteService --> MatchingEngine : utiliza
  ReportePerdida --> Avistamiento : contiene
  ReportePerdida --> Usuario : asociado
  ReportePerdida --> Mascota : asociado
  Coincidencia --> ReportePerdida : asocia
  Coincidencia --> Mascota : referencia
  ReportePerdida --> EstadoReporte : tipo
}

@enduml
```

UML-Repotes

- **Despliegue: representar cómo los contenedores identificados en C4 se distribuyen en nodos físicos/lógicos de ejecución (servidores, bases de datos, clientes, etc.).**
  Diagrama de despliegue

4. Relación entre modelos

   Incluir una breve explicación donde muestren explícitamente:

- Cómo los componentes de C4 se traducen en clases UML.
- Cómo esos mismos componentes/clases se asignan en el diagrama de despliegue.

5. Entregables

Informe en PDF con:

1. Explicación breve de cada tipo de diagrama.
2. Diagramas elaborados (C4 – componentes, UML clases, UML despliegue).
3. Relación y justificación entre los tres niveles

Relación entre los modelos

De componentes C4 a Clases UML

La vista de componentes del modelo C4 identifica **módulos funcionales cohesivos** del sistema, mientras que el diagrama de clases UML especifica la **estructura interna** de cada componente mediante clases, atributos, métodos y relaciones.

Mapeo Componente → Clases

Componente C4	Clases UML Principales	Descripción
Autenticación y Autorización	AuthController, AuthService, Usuario, TokenGenerator, RolUsuario	El componente se materializa en un controlador que expone endpoints REST, un servicio que contiene la lógica de negocio, y entidades que modelan usuarios y sus roles. El TokenGenerator es una clase utilitaria.
Gestión de Datos de Mascotas	MascotaController, MascotaService, Mascota, EspecieMascota, TamanoMascota, ValidadorMascota	El componente se traduce en un patrón MVC/Service, donde el controlador maneja peticiones HTTP, el servicio implementa reglas de negocio, y la entidad Mascota representa el modelo de dominio con sus características.
Procesamiento de Reportes	ReporteController, ReporteService, ReportePerdida, Avistamiento, Coincidencia, EstadoReporte	El componente incluye controladores y servicios para gestionar reportes, entidades que modelan los reportes y sus estados.

Gestión de Alertas y Notificaciones	NotificacionController, NotificacionService, Notificacion, CanalNotificacion, PreferenciasNotificacion	Similar estructura de controlador-servicio-entidad. Las clases modelan notificaciones, canales de envío (email, push, in-app) y preferencias de usuario.
Geolocalización	GeolocalizacionService, Ubicacion, CalculadorDistancias	Componente más orientado a servicios, con una entidad Ubicacion (coordenadas GPS) y clases utilitarias para cálculos geográficos.
Integración Externa	IntegracionService, ClienteRedesSociales, ClienteChipRFID, AdaptadorAPI	Componente de servicios con clases adaptadoras que encapsulan la comunicación con APIs externas.
Reportes y Estadísticas	EstadisticaController, EstadisticaService, ReporteEstadistico, GeneradorPDF, ExportadorCSV	Controlador y servicio para generar analytics, con clases utilitarias para exportar datos en diferentes formatos.

De clases UML a despliegue

Las clases UML definen la **estructura lógica del software**, mientras que el diagrama de despliegue muestra cómo estas clases se **empaquetan, distribuyen y ejecutan** en infraestructura física o virtualizada.

Clases UML	Contenedor	Nodo de Despliegue	Justificación
Todas las clases Controller	Backend/API	Servidor de Aplicaciones	Los controladores manejan peticiones HTTP y se ejecutan en el servidor de aplicaciones como parte de la aplicación backend.
Todas las clases Service	Backend/API	Servidor de Aplicaciones	Los servicios contienen lógica de negocio y se ejecutan en el mismo proceso que los controladores.

Justificación de decisiones de diseño

Separación en componentes

La descomposición en componentes sigue el **principio de responsabilidad única (SRP)** y facilita:

- **Mantenibilidad**: Cada componente tiene un propósito claro.
- **Escalabilidad**: Los componentes pueden escalarse independientemente (ej: más instancias del componente de Notificaciones si hay picos de tráfico).
- **Testabilidad**: Cada componente puede testearse de forma aislada mediante mocks.
- **Reusabilidad**: Componentes como Autenticación pueden reutilizarse en otros proyectos.

Patrón de arquitectura en capas

Se utiliza una arquitectura en capas:

- **Capa de presentación** (Controllers): Maneja peticiones HTTP, validación de entrada, serialización JSON.
- **Capa de negocio** (Services): Contiene la lógica de dominio, reglas de negocio, orquestación de operaciones.
- **Capa de persistencia** (Repositories/Entities): Maneja el acceso a datos, mapeo objeto-relacional.

Ventajas:

- Clara separación de responsabilidades.
- Facilita cambios en una capa sin afectar las demás (ej: cambiar la base de datos sin modificar la lógica de negocio).

Uso de DTOs (Data Transfer Objects)

Los DTO se utilizan para:

- **Desacoplar** la API de la estructura interna de las entidades.
- **Validar** datos de entrada antes de procesarlos.
- **Controlar** qué información se expone al cliente (evitar exponer contraseñas, datos sensibles).
- **Versionar** APIs sin romper clientes existentes.

Uso de servicios externos

Delegar funcionalidades a servicios especializados (email, push notifications, mapas) permite:

- **Reducir complejidad**: No reinventar la rueda.
- **Confiabilidad**: Usar servicios probados y de alta disponibilidad.