Diseño de Arquitectura del Sistema - **(SISTEC)**

**Fecha:** 09/09/2025

**Autor: Raurau Quispe, Juan Boris; Munares Chamorro, Juan Carlos**

**Versión:** 1.0

# 1. Introducción

Este documento describe de manera detallada la arquitectura del sistema SISTEC, una aplicación web monolítica desarrollada en PHP cuyo propósito es permitir la gestión de seguridad y monitoreo de equipos de laboratorio mediante tecnología RFID. Se define la estructura general del sistema, los componentes que lo conforman, sus interacciones y las tecnologías empleadas.

# 2. Visón General de la Arquitectura

# Visión General

El sistema SISTEC es una aplicación web monolítica desarrollada con PHP y Frontend en HTML/CSS/JS. Se ejecuta sobre un servidor LAMP (Linux, Apache, MySQL/MariaDB, PHP) o Nginx+PHP-FPM. Dado el packaging actual (archivos PHP que contienen la lógica, vistas y acceso a BD), el estilo arquitectónico es MONOLÍTICO.

# Componentes de la Arquitectura Monolítica

Lista y descripción de los componentes y responsabilidades:

2.1. Servidor Web (Apache / Nginx):

- Expone endpoints HTTP/HTTPS.  
- Sirve archivos estáticos (CSS, JS, imágenes) y procesa peticiones PHP.  
- Recomendación: usar Nginx como reverse proxy y PHP-FPM para mejor rendimiento.

2.2. Aplicación PHP (Capa única):

- Contiene controladores (scripts PHP en /php/), vistas (/views/) y acceso a BD (php/db.php).  
- Lógica: autenticación, autorización por rol, CRUD de usuarios, equipos, procesamiento de alertas y generación de reportes.  
- Comunicación sin API REST separada: la misma aplicación sirve vistas y endpoints AJAX.

2.3. Base de Datos (MariaDB/MySQL):

- Persistencia de usuarios, equipos y alertas. Ver TP2. E6 para el diseño detallado.

2.4. Dispositivo Lector RFID (ESP32/MFRC522):

- Dispositivo con conectividad Wi‑Fi/HTTP que hace POST a php/alertas/registrar.php con número\_tarjeta\_rfid.  
- La aplicación consulta equipos por numero\_tarjeta\_rfid y crea registros en alertas.

# 3. Flujo de Datos y Secuencias (monolito)

Caso: Lectura RFID -> Generación de alerta:

1. El ESP32 lee el UID de la tarjeta y realiza una petición HTTP POST a /php/alertas/registrar.php con el campo numero\_tarjeta\_rfid.  
2. El script PHP recibe la solicitud y valida el payload (presencia y formato del UID).  
3. Se ejecuta una SELECT sobre equipos WHERE numero\_tarjeta\_rfid = ? para obtener nombre y ubicacion.  
4. El backend construye el mensaje de alerta (ej. '📟 PANTALLA 01 ha salido del laboratorio') y decide el nivel (peligro/cuidado) según reglas de negocio.  
5. Se inserta la fila en alertas (mensaje, nivel, ubicacion, fecha, numero\_tarjeta\_rfid) dentro de una transacción si aplica.  
6. Se responde al ESP32 con JSON que indica éxito o error.  
7. Opcionalmente, se dispara notificación (email/SMS) o se almacena en logs para auditoría.

# 4. Estructura de Carpetas (recomendación)

/ (raíz del proyecto)  
├─ index.php  
├─ login.php, register.php  
├─ /php/ -> controladores y endpoints (usuarios/, equipos/, alertas/)  
├─ /views/ -> vistas HTML + plantillas  
├─ /assets/ -> css/, js/, img/  
├─ /config/ -> config.php, parametros de despliegue  
├─ /logs/ -> logs de aplicación (no accesible públicamente)  
├─ /vendor/ -> dependencias si las hay

# 5. Seguridad en la Arquitectura

Controles y recomendaciones prácticas:

- Autenticación y sesiones: usar session\_start(), cookies con flags Secure y HttpOnly, regenerar session\_id al login, tiempo de expiración.  
- Contraseñas: password\_hash() y password\_verify(). Longitud de hash y costo controlado.  
- CSRF: implementar tokens CSRF en formularios (hidden inputs) y validarlos en el backend.  
- XSS: escapar salida en vistas (htmlspecialchars) y aplicar Content Security Policy (CSP) en headers.  
- Inyección SQL: usar sentencias preparadas (mysqli->prepare) siempre y validación/saneamiento de entradas.  
- HTTPS obligatorio en producción con HSTS.  
- Gestión de roles: control de acceso por rol (admin, tecnico, invitado) en cada endpoint crítico.  
- Logs de auditoría: registrar acciones críticas (login, CRUD usuarios, eliminaciones) con timestamp y usuario actor.

# 6. Disponibilidad, Backup y Recuperación

- Respaldos: mysqldump periódicos, binlogs para PITR.  
- Alta disponibilidad (opcional): replicación Maestra-Esclavo o uso de servicio DB gestionado (RDS).  
- Recuperación: playbook paso a paso para restaurar (restaurar dump + aplicar binlogs desde X fecha).  
- Monitoreo: Prometheus / Grafana o soluciones de logs (ELK) para métricas y alertas operativas.

# 7. Escalabilidad y Evolución desde Monolito

- Escalamiento vertical (inicial): aumentar recursos del servidor (CPU, RAM, IOPS) para manejar más lectores y concurrencia.  
- Escalamiento horizontal (a futuro): separar estático (NGINX) y PHP-FPM en múltiples instancias detrás de un balanceador. Mantener BD central o migrar a cluster.  
- Refactor sugerido para microservicios: extraer módulo de ingestión de alertas -> servicio independiente con cola (RabbitMQ/Redis Streams) para desacoplar lecturas y persistencia.  
- Contenerización: empaquetar el monolito en Docker (imagen PHP + Apache), definir docker-compose para desarrollo y CI/CD.

# 8. Observabilidad y Mantenimiento

- Logs estructurados (JSON) para facilitar ingestión en ELK/Graylog.  
  
- Métricas clave: latencia de endpoints, tasa de alertas/segundo, errores 5xx, uso de BD (queries lentas).  
  
- Política de despliegue: entornos dev -> staging -> prod con CI (tests básicos de integración y scripts de migración DB).

# 9. Anexos y diagramas

ERD:

Imagen que contiene Diagrama

El contenido generado por IA puede ser incorrecto.

Diagrama de secuencia: sección 3 para el flujo de alertas.