Documento de Arquitectura de Software

Sistema de Autenticación Único (SSO) - Clean Architecture

Fecha: diciembre 2025

1. Introducción General

1.1 Propósito del documento

Este documento describe en detalle la arquitectura del Sistema de Autenticación Único (SSO), diseñado para centralizar la gestión de identidades, proveer servicios de autenticación y autorización robustos, y facilitar la administración de usuarios y roles. El diseño sigue los principios de **Clean Architecture** para asegurar la independencia de frameworks, testabilidad y mantenibilidad a largo plazo.

1.2 Alcance del sistema

El sistema abarca la totalidad del ciclo de vida de la identidad digital dentro de la organización:

- **Autenticación:** Inicio de sesión seguro mediante credenciales y tokens JWT.
- **Seguridad Avanzada:** Implementación de Segundo Factor de Autenticación (2FA) con TOTP (Google Authenticator).
- **Autorización:** Control de acceso granular basado en Roles (RBAC) y Permisos (Claims/Policies).
- **Gestión Administrativa:** Panel web (Frontend) para la administración de usuarios, asignación dinámica de roles y gestión del catálogo de permisos.
- **Infraestructura:** Despliegue contenerizado y orquestado para entornos de desarrollo y producción.

1.3 Audiencia y nivel técnico esperado

Este documento está dirigido a desarrolladores de software (Backend .NET y Frontend React), arquitectos de soluciones, ingenieros DevOps y stakeholders técnicos encargados de la seguridad y despliegue del sistema.

2. Visión Arquitectónica General

2.1 Estilo arquitectónico utilizado

Se adopta el estilo de **Clean Architecture (Arquitectura Limpia)**, estructurando el sistema en capas concéntricas donde la dependencia siempre apunta hacia el interior (el Dominio).

- **Frontend:** Single Page Application (SPA) desacoplada.
- **Backend:** API RESTful modularizada.
- **Despliegue:** Arquitectura basada en contenedores (Docker).

2.2 Decisiones arquitectónicas clave (ADR)

- **ADR-001 - Clean Architecture:** Se elige para desacoplar la lógica de negocio de la UI y la base de datos, permitiendo cambiar tecnologías externas sin afectar el núcleo.
- **ADR-002 - Patrón CQRS (Mediator):** Uso de la librería MediatR para separar las operaciones de lectura (Queries) y escritura (Commands), reduciendo el acoplamiento en los controladores.
- **ADR-003 - Identity & JWT:** Uso de ASP.NET Core Identity para la gestión de usuarios y JSON Web Tokens (JWT) para mantener la autenticación *stateless* (sin estado), facilitando la escalabilidad.
- **ADR-004 - Runtime Configuration:** Inyección de variables de entorno en tiempo de ejecución (tanto en Backend como Frontend) para permitir la portabilidad entre entornos sin recompilar ("Build once, deploy anywhere").

2.3 Diagramas de alto nivel

El sistema se compone de tres contenedores principales orquestados:

1. **Frontend Container (Nginx + React):** Interfaz de usuario.
2. **Backend Container (.NET 8 API):** Lógica de negocio y seguridad.
3. **Database Container (SQL Server):** Persistencia de datos.

3. Componentes del Sistema

3.1 Módulos principales y responsabilidades

El backend se divide en cuatro proyectos físicos según Clean Architecture:

- **SSO.Domain:** Núcleo central. Contiene entidades (Usuario, SystemPermission), objetos de valor y constantes de seguridad. Sin dependencias externas.
- **SSO.Application:** Capa de orquestación. Define casos de uso (Handlers), interfaces (IAuthService, IUserService) y DTOs.
- **SSO.Infrastructure:** Capa de implementación técnica. Contiene el DbContext (EF Core), la implementación de Identity, servicios de JWT, lógica de 2FA y repositorios.
- **SSO.WebApi:** Capa de presentación. Expone endpoints REST, configura la inyección de dependencias y gestiona middlewares (CORS, Auth).

3.2 Interfaces y APIs expuestas

El sistema expone una API REST documentada vía Swagger, con los siguientes recursos principales:

- POST /api/Auth/login: Autenticación y emisión de JWT.
- POST /api/Auth/2fa-*: Configuración y validación de doble factor.
- GET/POST /api/Users: Gestión CRUD de usuarios (Solo Admins).
- GET/POST /api/Roles: Gestión dinámica de roles y permisos.

3.3 Comunicación entre componentes

- **Frontend -> Backend:** Comunicación asíncrona mediante HTTP/HTTPS (JSON) utilizando Axios.
- **Backend -> Base de Datos:** Comunicación TCP/IP hacia el contenedor de SQL Server utilizando Entity Framework Core.
- **Interna (Backend):** Comunicación en memoria mediante Inyección de Dependencias y mediación de mensajes (IMediator).

3.4 Integración con sistemas externos

- **Google Authenticator:** Integración mediante el estándar TOTP (Time-based One-Time Password) para generar códigos de 6 dígitos validados por el servidor.

4. Detalle del Estilo Arquitectónico

4.1 Arquitectura en Capas (Clean Architecture)

- **Capa de Dominio:** Entidades puras y reglas de negocio invariantes.
- **Capa de Aplicación:** Lógica de la aplicación específica. No conoce la base de datos ni la web, solo interfaces.
- **Capa de Infraestructura:** "El mundo sucio". Aquí residen las implementaciones concretas de acceso a datos, envío de correos, generación de tokens, etc.
- **Capa de Presentación (API):** Punto de entrada. Traduce peticiones HTTP a Comandos/Queries de la aplicación.

5. Seguridad

5.1 Autenticación y autorización

- **Autenticación:** Gestionada por **ASP.NET Core Identity**. Las contraseñas se almacenan con hash seguro. Se emiten tokens **JWT** firmados digitalmente (HMACSHA256) que contienen la identidad del usuario.
- **Autorización:** Implementación de **RBAC (Role-Based Access Control)** enriquecido con **Políticas de Permisos (Claims)**. Los controladores no solo verifican el rol (ej: "Admin"),

sino la posesión de permisos específicos (ej: Permissions.Users.Delete), permitiendo una granularidad fina.

5.2 Gestión de secretos y cifrado

- **Secretos:** Las cadenas de conexión y claves de firma de tokens **NO** se almacenan en el código fuente. Se inyectan mediante **Variables de Entorno** en el contenedor Docker.
- **Cifrado:** Todo el tráfico externo debe cursar sobre HTTPS (TLS). Las contraseñas de usuario y claves 2FA están cifradas o hasheadas en la base de datos.

5.3 Políticas de acceso

- **CORS (Cross-Origin Resource Sharing):** Configurado dinámicamente para permitir peticiones solo desde orígenes confiables definidos en la configuración del entorno.
- **Protección de Endpoints:** Uso del atributo [Authorize] global o específico por política en los controladores.

6. Escalabilidad y Rendimiento

6.1 Estrategias de escalabilidad

La arquitectura es **Stateless** (sin estado). El servidor no mantiene sesiones de usuario en memoria (usa JWT). Esto permite escalar el backend horizontalmente (múltiples instancias del contenedor API) detrás de un balanceador de carga sin problemas de afinidad.

6.2 Balanceo de carga

El frontend (React) es servido por Nginx, que es altamente eficiente para contenido estático. En un entorno de orquestación (como Kubernetes), el tráfico hacia la API se balancea nativamente entre los pods disponibles.

6.3 Tolerancia a fallos y alta disponibilidad

El despliegue con Docker Compose incluye políticas de reinicio automático (restart: on-failure o always) y *Healthchecks* para asegurar que los servicios dependientes (como la API esperando a la Base de Datos) inicien en el orden correcto y se recuperen ante fallos transitorios.

7. DevOps y Despliegue

7.1 Estrategia de CI/CD

El proyecto está configurado para la construcción de imágenes Docker optimizadas (Multi-stage build). Las imágenes se versionan y se suben a un registro de contenedores (Docker Hub) bajo los repositorios sergiovallegarma/sso-api-final y sergiovallegarma/sso-web-final.

7.2 Infraestructura como código

Toda la infraestructura necesaria para ejecutar el sistema está definida en el archivo docker-compose.yml. Esto garantiza paridad entre los entornos de desarrollo, pruebas y producción, eliminando el problema de "funciona en mi máquina".

7.3 Ambientes de despliegue

El sistema soporta la configuración de múltiples ambientes mediante inyección de variables.

- **Frontend:** Utiliza un script de arranque (env.sh) que inyecta la URL de la API en tiempo de ejecución en el navegador.
- **Backend:** Lee la cadena de conexión y configuraciones de seguridad desde las variables de entorno del host.

8. Calidad y Mantenibilidad

8.1 Estrategias de pruebas

La separación de capas facilita enormemente las pruebas:

- **Unitarias:** Se pueden crear pruebas para los Handlers de la Capa de Aplicación "mockeando" las interfaces de Infraestructura.
- **Integración:** La facilidad de levantar una base de datos SQL Server en Docker permite realizar pruebas de integración reales en entornos efímeros.

8.2 Observabilidad

Los contenedores emiten logs estructurados a la salida estándar (stdout/stderr), permitiendo su recolección centralizada. El backend incluye manejo global de excepciones para registrar errores críticos sin exponer detalles sensibles al cliente.

8.3 Gestión de deuda técnica

El uso estricto de Clean Architecture previene la deuda técnica arquitectónica, evitando dependencias circulares y "código espagueti". Las migraciones de Entity Framework permiten un control de versiones estricto sobre el esquema de la base de datos.

9. Anexos y Referencias

9.1 Glosario

- **JWT:** JSON Web Token.
- **TOTP:** Time-based One-Time Password.
- **IdP:** Identity Provider.
- **SPA:** Single Page Application.

9.2 Referencias y normativas

- Estándares OAuth 2.0 y OpenID Connect (como referencia de diseño).
- Guías de arquitectura de aplicaciones .NET de Microsoft.

9.3 Documentación técnica relacionada

- Repositorio GitHub del proyecto.
- Imágenes Docker en Docker Hub.
- Guías de instalación y despliegue (docker-compose.prod.yml).