## Elementos Principales en el Diseño de Proyectos de Software (con TDD integrado)

1. **Requisitos del Proyecto**
   * Funcionales: Qué debe hacer el sistema.
   * No funcionales: Rendimiento, seguridad, usabilidad, escalabilidad.
2. **Arquitectura del Sistema**
   * Definición de componentes, módulos y su interacción.
   * Selección de patrones arquitectónicos (por ejemplo, MVC, microservicios).
   * Decisiones sobre tecnología, bases de datos, APIs.
   * **Diseño para testabilidad:** crear componentes modulares y fácilmente testeables para facilitar TDD.
3. **Diseño de Datos**
   * Modelado de la base de datos.
   * Estructura de datos y esquemas.
   * Integridad y relaciones.
4. **Diseño de la Interfaz de Usuario (UI/UX)**
   * Wireframes y prototipos.
   * Experiencia de usuario.
   * Accesibilidad y responsividad.
   * **Pruebas TDD de componentes UI:** implementar tests unitarios e integración para componentes front-end.
5. **Diseño de la Seguridad**
   * Autenticación y autorización.
   * Manejo de datos sensibles.
   * Políticas de acceso y cifrado.
6. **Planificación de Testing**
   * Tipos de pruebas: unitarias, integración, sistema, aceptación.
   * **Incorporación de TDD:** escribir tests antes del código, definir flujos para desarrollo guiado por pruebas.
   * Automatización de pruebas.
   * Definición de criterios de calidad.
7. **Planificación de Implementación y Despliegue**
   * Estrategias de desarrollo: iterativo, incremental.
   * **Integración continua / entrega continua (CI/CD) con enfoque en TDD:** ejecución automática de tests en pipelines para validar cambios.
   * Gestión de versiones y releases.
8. **Documentación**
   * Documentos de diseño, arquitectura, APIs y usuarios.
   * Manuales y guías.
   * **Tests como documentación viva:** los tests escritos en TDD funcionan también como referencia del comportamiento esperado.
9. **Gestión del Proyecto**
   * Cronograma, recursos, riesgos.
   * Metodologías ágiles o tradicionales.
   * **Cultura y adopción de TDD:** capacitar al equipo, asignar tiempos para pruebas, promover calidad.

### 1. ****"Software Engineering" – Ian Sommerville****

### 2. ****"The Mythical Man-Month" – Frederick P. Brooks Jr.****

### 3. ****"Clean Architecture" – Robert C. Martin (Uncle Bob)****

### 4. ****ISO/IEC/IEEE 12207 – Software Life Cycle Processes****

### 5. ****IEEE Software Engineering Standards****

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_-

## ¿Qué son los Requisitos Funcionales?

* Describen **qué debe hacer el sistema**: funciones, comportamientos, procesos que debe realizar.
* Ejemplo: "El sistema debe permitir al usuario registrarse con email y contraseña."

## ¿Cómo identificar los requisitos funcionales?

1. **Entrevistas con stakeholders:** Hablar con clientes, usuarios, expertos.
2. **Análisis de procesos actuales:** Entender cómo se hacen las tareas que el sistema reemplazará o apoyará.
3. **Estudio de documentos:** Normativas, manuales, ejemplos previos.
4. **Casos de uso y escenarios:** Describir situaciones en que el usuario interactúa con el sistema.
5. **Talleres colaborativos:** Sesiones con el equipo para levantar y validar requisitos.

## Orden típico entre Casos de Uso y Requisitos Funcionales

### 1. ****Identificación inicial de Requisitos Funcionales****

* Primero se levantan los requisitos funcionales de alto nivel: qué funciones o servicios debe ofrecer el sistema.
* Estos son descripciones generales de las capacidades que el sistema debe tener.

### 2. ****Definición de Casos de Uso****

* A partir de esos requisitos funcionales, se desarrollan casos de uso que describen **escenarios específicos** de interacción entre usuarios (actores) y el sistema.
* Los casos de uso detallan el flujo de acciones para cumplir una función concreta y ayudan a entender cómo se usan esos requisitos en la práctica.

### 3. ****Refinamiento de Requisitos Funcionales****

* Con base en los casos de uso, los requisitos funcionales pueden ser ajustados o detallados para cubrir todos los escenarios y excepciones.
* Esto asegura que el sistema sea completo y cubra las necesidades reales.

|  |  |
| --- | --- |
| **Campo** | **Descripción** |
| **ID** | Identificador único del caso de uso (ejemplo: CU001) |
| **Nombre** | Nombre descriptivo y claro del caso de uso |
| **Actor(es)** | Usuario(s) o sistema(s) que interactúan con el caso de uso |
| **Descripción** | Breve resumen de qué hace el caso de uso |
| **Precondiciones** | Condiciones que deben cumplirse antes de que se pueda ejecutar el caso de uso |
| **Flujo Principal** | Secuencia paso a paso de acciones que ocurren cuando el caso de uso se ejecuta con éxito |
| **Flujos Alternativos** | Descripción de variaciones o situaciones excepcionales en el flujo principal |
| **Postcondiciones** | Estado del sistema después de que el caso de uso se completa |
| **Requisitos Relacionados** | Referencia a los requisitos funcionales o no funcionales que cubre este caso de uso |
| **Notas / Comentarios** | Información adicional o aclaraciones importantes |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **ID** | **Nombre del Requisito** | **Descripción Detallada** | **Prioridad (Alta/Media/Baja)** | **Responsable** | **Estado** | **Criterios de Aceptación** | **Comentarios / Notas** |
| RF001 | Registro de Usuario | El sistema debe permitir a los usuarios registrarse mediante email y contraseña, validando formato y duplicados. | Alta | Ana Pérez | Pendiente | - Usuario recibe email de confirmación.- No se permiten emails duplicados.- Contraseña con mínimo 8 caracteres. | Revisar política de privacidad. |
| RF002 | Inicio de Sesión | Permitir a los usuarios autenticarse con email y contraseña para acceder a su cuenta. | Alta | Juan Gómez | En desarrollo | - Sesión iniciada correctamente.- Sesión expira tras 30 min de inactividad. | Validar integración con sistema de MFA. |
| RF003 | Recuperación de Contraseña | Enviar enlace seguro para restablecer contraseña cuando el usuario lo solicite. | Media | Ana Pérez | En pruebas | - Enlace expira en 24 horas.- Enlace solo usable una vez. | Probar en diferentes navegadores. |
| RF004 | Modificación de Perfil | El usuario puede actualizar su información personal y contraseña. | Media | Equipo Backend | Completado | - Cambios reflejados en base de datos.- Notificación de éxito al usuario. | Confirmar validación de campos. |

|  |  |
| --- | --- |
| **Estado** | **Descripción** |
| **Pendiente** | Requisito identificado pero aún no iniciado su análisis ni desarrollo. |
| **En análisis** | Requisito en estudio para definir detalles técnicos y funcionales. |
| **En desarrollo** | Equipo implementando la funcionalidad en el sistema. |
| **En pruebas** | Funcionalidad desarrollada y sometida a pruebas de calidad y validación. |
| **Completado** | Requisito implementado, probado y aprobado para producción. |
| **Bloqueado** | Avance detenido por impedimentos como falta de información, recursos o dependencias. |

**Definición de Requisitos No Funcionales**

Los requisitos no funcionales definen **cómo debe comportarse el sistema** en aspectos como rendimiento, seguridad, usabilidad, etc. Más que funcionalidades específicas, establecen **criterios de calidad**.

**¿Cómo definirlos?**

1. **Rendimiento**
   * Ejemplo: El sistema debe responder a cualquier solicitud en menos de 2 segundos bajo una carga de 1000 usuarios concurrentes.
   * Indicadores: Tiempo de respuesta, throughput (transacciones por segundo), tasa de errores bajo carga.
2. **Seguridad**
   * Ejemplo: El sistema debe soportar autenticación multifactor y cifrar datos sensibles en reposo y tránsito.
   * Indicadores: Número de vulnerabilidades detectadas, porcentaje de cumplimiento de normativas (GDPR, OWASP), tiempo para corregir vulnerabilidades.
3. **Usabilidad**
   * Ejemplo: El sistema debe permitir que un usuario nuevo complete el registro sin ayuda en menos de 3 minutos.
   * Indicadores: Tiempo promedio para completar tareas, tasa de error de usuarios, resultados de encuestas de satisfacción.
4. **Escalabilidad**
   * Ejemplo: El sistema debe soportar un aumento del 50% en usuarios sin degradar el rendimiento.
   * Indicadores: Capacidad máxima de usuarios concurrentes, tiempo de escalado automático, métricas de carga.

**Ejemplo de formato para definir requisitos no funcionales con indicadores**

|  |  |  |  |
| --- | --- | --- | --- |
| **Requisito No Funcional** | **Descripción** | **Indicador/Medida** | **Valor Objetivo** |
| Rendimiento | Tiempo de respuesta promedio para solicitudes críticas. | Tiempo de respuesta (ms) | ≤ 2000 ms |
| Seguridad | Implementación de cifrado de datos sensibles. | % de datos cifrados | 100% |
| Usabilidad | Facilidad para completar registro sin ayuda. | Tiempo promedio (minutos) | ≤ 3 minutos |
| Escalabilidad | Capacidad para aumentar usuarios sin afectar rendimiento. | Usuarios concurrentes soportados | 1500 usuarios concurrentes |

**Definición de Componentes, Módulos y su Interacción**

**¿Qué es?**

* **Componentes y módulos:** Son las partes **independientes o semi-independientes** que componen el sistema.
* Cada componente tiene una función o responsabilidad clara.
* Los módulos son grupos lógicos de código o funcionalidades que trabajan juntos para cumplir un propósito.
* **Interacción:** Es cómo estos componentes se comunican o colaboran para que el sistema funcione como un todo.

**¿Cómo se hace?**

1. **Análisis de requisitos y funcionalidades:**
   * Identificar qué funcionalidades principales debe tener el sistema.
2. **División en componentes/módulos:**
   * Agrupar funcionalidades relacionadas en módulos claros y con responsabilidades bien definidas (principio de separación de responsabilidades).
3. **Definir interfaces de comunicación:**
   * Especificar cómo se comunicarán los módulos entre sí (APIs internas, eventos, mensajes, llamadas directas).
4. **Seleccionar patrones arquitectónicos:**
   * Por ejemplo, arquitectura en capas, microservicios, arquitectura orientada a eventos, etc.
5. **Diagramar la arquitectura:**
   * Usar diagramas de componentes, diagramas de secuencia o diagramas de flujo para mostrar interacción.

**Guía para Definir la Arquitectura de un Proyecto de Software**

**1. Entender los Requisitos**

* Revisa los requisitos funcionales y no funcionales.
* Determina necesidades clave de rendimiento, seguridad, escalabilidad y usabilidad.

**2. Identificar Componentes y Módulos**

* Divide el sistema en partes lógicas que cumplen funciones específicas.
* Define responsabilidades claras para cada componente o módulo.
* Ejemplo: Módulo de autenticación, módulo de gestión de usuarios, módulo de pagos.

**3. Definir la Interacción entre Componentes**

* Decide cómo los módulos se comunicarán:
  + Llamadas directas (funciones, métodos).
  + APIs REST o GraphQL.
  + Mensajería asíncrona o eventos.
* Define protocolos, formatos de datos y contratos de interfaces.

**4. Seleccionar Patrones Arquitectónicos**

* Elige un patrón que se adapte a las necesidades y el contexto del proyecto:
  + **Monolito**: Todo en un solo despliegue (ideal para proyectos pequeños o MVPs).
  + **MVC (Modelo-Vista-Controlador)**: Separación clara de lógica y presentación.
  + **Microservicios**: Servicios independientes y escalables, buena opción para sistemas grandes.
  + **Arquitectura basada en eventos**: Sistemas reactivos y desacoplados.

**5. Elegir Tecnologías y Herramientas**

* Selecciona lenguajes de programación, frameworks y librerías que se adapten al equipo y requisitos.
* Decide la base de datos (relacional, NoSQL, en memoria).
* Define sistemas de mensajería, balanceadores, y plataformas de despliegue.

**6. Documentar la Arquitectura**

* Usa diagramas:
  + Diagramas de componentes.
  + Diagramas de flujo o secuencia para mostrar interacciones.
* Describe decisiones clave y justificaciones.
* Indica puntos de escalabilidad, seguridad y mantenimiento.

**7. Validar con Stakeholders**

* Revisa la arquitectura con el equipo y partes interesadas.
* Ajusta según feedback y nuevas necesidades.

**8. Plan para Evolución y Mantenimiento**

* Define cómo la arquitectura podrá escalar y adaptarse a cambios futuros.
* Considera modularidad y separación para facilitar actualizaciones.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_-

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Este proyecto sigue los estándares más altos para desarrollo profesional moderno:

- Código modular y documentado

- Testing con Jest

- Dockerización y CI/CD con GitHub Actions

- Especificación OpenAPI para APIs

my-software-project/

├── README.md # Descripción general del proyecto y cómo usar la plantilla

├── docs/

│ ├── architecture.md # Documentación de arquitectura del proyecto

│ ├── requirements.md # Requisitos funcionales y no funcionales

│ ├── api-spec.yaml # Especificación OpenAPI para API REST

│ └── style-guide.md # Guía de estilo para el código

├── src/

│ ├── index.ts # Punto de entrada principal (ejemplo Node.js con TS)

│ ├── modules/ # Código modularizado

│ │ └── exampleModule.ts

│ └── tests/ # Tests unitarios y de integración (Jest)

│ └── exampleModule.test.ts

├── .github/

│ └── workflows/

│ └── ci.yml # Pipeline de integración continua (GitHub Actions)

├── Dockerfile # Para contenerizar la aplicación

├── docker-compose.yml # Orquestación local con servicios dependientes

├── package.json # Dependencias y scripts (Node.js ejemplo)

├── tsconfig.json # Configuración TypeScript

├── .eslintrc.json # Linter para mantener calidad de código

├── .prettierrc # Formato de código

└── .gitignore # Archivos y carpetas a ignorar en Git

**¿Qué es un Ingeniero de Software?**

Un **Ingeniero de Software** es un profesional especializado en el diseño, desarrollo, mantenimiento y gestión de sistemas de software. Utiliza principios de ingeniería para construir software de manera eficiente, escalable, confiable y segura.

**¿Qué hace un Ingeniero de Software?**

* **Análisis de requisitos:** Entiende qué necesita el cliente o usuario para definir qué debe hacer el software.
* **Diseño:** Planea la arquitectura y estructura del software para cumplir los requisitos.
* **Desarrollo:** Escribe el código fuente aplicando buenas prácticas de programación.
* **Testing:** Realiza pruebas para garantizar que el software funcione correctamente y sea robusto.
* **Mantenimiento:** Corrige errores y adapta el software a nuevas necesidades o tecnologías.
* **Gestión de proyectos:** Participa en la planificación, seguimiento y entrega de proyectos.

**Habilidades clave**

* Programación en varios lenguajes.
* Conocimiento de metodologías ágiles (Scrum, Kanban).
* Manejo de herramientas de control de versiones (Git).
* Conocimientos de bases de datos, redes y seguridad.
* Comunicación efectiva para trabajar en equipo.

**Diferencia con un programador**

* Un programador suele enfocarse en escribir código.
* Un ingeniero de software tiene una visión más amplia que incluye análisis, diseño, pruebas y mantenimiento.

## Etapas del Desarrollo de Software

### 1. ****Análisis de Requisitos****

* Se identifican y documentan las necesidades del cliente y usuarios.
* Se definen requisitos funcionales (qué debe hacer el software) y no funcionales (rendimiento, seguridad, usabilidad).
* Se establecen prioridades y criterios de aceptación.

### 2. ****Diseño****

* Se planifica la arquitectura del sistema.
* Se diseñan diagramas (UML, de flujo, de componentes).
* Se especifican tecnologías, bases de datos y interfaces.

### 3. ****Implementación o Desarrollo****

* Se escribe el código según el diseño.
* Se realizan pruebas unitarias para verificar cada componente.
* Se integran los módulos progresivamente.

### 4. ****Pruebas (Testing)****

* Se ejecutan pruebas de integración, sistema y aceptación.
* Se detectan y corrigen errores.
* Se valida que el software cumple con los requisitos.

### 5. ****Despliegue****

* Se instala y configura el software en el entorno de producción.
* Se realiza la entrega formal al cliente.
* Se capacita a usuarios si es necesario.

### 6. ****Mantenimiento****

* Se corrigen errores detectados en producción.
* Se realizan actualizaciones y mejoras.
* Se adapta el software a nuevos requerimientos.

### Metodologías comunes

* **Cascada:** Etapas secuenciales, cada una termina antes de comenzar la siguiente.
* **Ágil (Scrum, Kanban):** Desarrollo iterativo e incremental, con ciclos cortos y adaptativos.
* **DevOps:** Integra desarrollo y operaciones para despliegues continuos y automatizados.

## ¿Qué es DevOps?

**DevOps** es una cultura, práctica y conjunto de herramientas que unifican el desarrollo de software (Dev) con las operaciones de TI (Ops). Su objetivo principal es acelerar la entrega de software con alta calidad, fomentando la colaboración entre desarrolladores, testers y administradores de sistemas.

## Componentes clave de DevOps

1. **Integración Continua (CI)**
   * Automatiza la construcción y pruebas del código cada vez que alguien hace un cambio (push).
   * Detecta errores temprano.
2. **Entrega Continua (CD)**
   * Automatiza la entrega del software a entornos de prueba o producción.
   * Permite despliegues frecuentes y confiables.
3. **Infraestructura como Código (IaC)**
   * Define y gestiona infraestructura (servidores, redes, bases de datos) usando código, lo que facilita replicar entornos.
4. **Monitoreo y Feedback**
   * Supervisión continua del software y la infraestructura.
   * Retroalimentación rápida para corregir fallos y mejorar.

## Prácticas de Ciberseguridad para Aplicaciones de Alto Nivel

### 1. ****Autenticación y Autorización Seguras****

* Uso de protocolos robustos como OAuth 2.0 y OpenID Connect.
* Implementar autenticación multifactor (MFA).
* Control estricto de permisos y roles para limitar acceso según necesidad (principio de menor privilegio).

### 2. ****Cifrado de Datos****

* Cifrar datos en tránsito usando HTTPS/TLS.
* Cifrado de datos sensibles en reposo (bases de datos, almacenamiento).
* Uso de algoritmos modernos y actualizados.

### 3. ****Validación y Saneamiento de Entradas****

* Validar toda entrada del usuario para evitar inyección de código (SQL, XSS).
* Utilizar librerías y frameworks que mitiguen vulnerabilidades comunes.

### 4. ****Gestión de Sesiones Seguras****

* Implementar sesiones con tokens seguros (ej. JWT con expiración).
* Usar cookies seguras y con atributos HttpOnly y Secure.
* Evitar exposición de tokens en URLs.

### 5. ****Registro y Monitoreo****

* Mantener logs de acceso y actividades importantes.
* Monitoreo en tiempo real para detectar comportamientos anómalos o intentos de ataque.
* Alertas automáticas ante eventos sospechosos.

### 6. ****Actualizaciones y Parcheo Constante****

* Mantener el software y dependencias actualizados.
* Aplicar parches de seguridad tan pronto como estén disponibles.

### 7. ****Protección contra Ataques Comunes****

* Implementar mecanismos contra ataques DDoS.
* Uso de firewalls de aplicaciones web (WAF).
* Mitigar ataques de fuerza bruta y de relleno de credenciales.

### 8. ****Seguridad en la Configuración****

* Evitar configuraciones por defecto inseguras.
* Minimizar exposición de servicios y puertos.
* Gestionar secretos y claves con herramientas específicas (ej. Vault).

### 9. ****Pruebas de Seguridad****

* Realizar análisis de vulnerabilidades y pruebas de penetración periódicas.
* Revisiones de código enfocadas en seguridad (code reviews).

## **Contextualización de la identidad en el backend** o **Server-Side Identity Injection**

### ¿Qué significa?

* El cliente (frontend o consumidor de API) **no envía directamente el userId ni datos sensibles de identidad** en el cuerpo de la solicitud (POST).
* En lugar de eso, envía un **token JWT (JSON Web Token)** que contiene la identidad y permisos.
* El backend **verifica y decodifica el token**, y **extrae el userId desde allí**, para luego usarlo en operaciones como registro en auditoría, asociación de datos, etc.
* Esto evita manipulaciones maliciosas, porque el usuario no puede alterar el userId arbitrariamente.

En el backend se hace el **“token introspection”** o validación del JWT para extraer claims como sub (subject, normalmente userId).  
  
**JWA** significa **JSON Web Algorithms**.

Es el conjunto de algoritmos criptográficos estándar que se utilizan para:

* Firmar (sign)
* Verificar firmas (verify)
* Cifrar (encrypt)
* Descifrar (decrypt)
* Derivar claves, etc.

**¿Dónde se usa JWA?**

JWA define las especificaciones de los algoritmos que puedes usar en:

* **JWS (JSON Web Signature)** para firmar y verificar tokens.
* **JWE (JSON Web Encryption)** para cifrar y descifrar datos.
* **JWT (JSON Web Token)** que es un tipo específico de JWS o JWE que se usa mucho para autenticación.

**Algunos ejemplos de algoritmos definidos en JWA**

| **Tipo de Algoritmo** | **Ejemplos** | **Uso** |
| --- | --- | --- |
| Algoritmos de firma | RS256, ES256, PS256, HS256 | Firmar/verificar JWT y JWS |
| Algoritmos de cifrado | A128GCM, A256GCM | Cifrar/descifrar JWE |
| Algoritmos de clave | RSA-OAEP, ECDH-ES | Intercambio de claves para cifrado |
| Algoritmos hash | SHA-256, SHA-384 | Hash usado en firmas y otros procesos |

# Definición y Clasificación de Datos Sensibles

## 1. ¿Qué son Datos Sensibles?

* **Definición:**  
  Datos que, si son divulgados, alterados o destruidos sin autorización, pueden causar daño a personas, organizaciones o sistemas.
* **Características principales:**
  + Confidencialidad requerida
  + Riesgo alto en caso de exposición
  + Protección legal o regulatoria

## 2. ¿Por qué clasificar Datos Sensibles?

* Para aplicar controles de seguridad adecuados.
* Para cumplir con normativas y estándares.
* Para minimizar riesgos de fugas o mal uso.

## 3. Clasificación Conceptual de Datos Sensibles

| **Categoría** | **Ejemplos** | **Riesgos si se Exponen** |
| --- | --- | --- |
| **Datos Personales** | Nombre, dirección, DNI, fecha de nacimiento | Robo de identidad, suplantación |
| **Datos de Autenticación** | Contraseñas, tokens, preguntas de seguridad | Acceso no autorizado, fraude |
| **Datos Financieros** | Números de tarjetas, cuentas bancarias | Fraude financiero, robo |
| **Datos de Salud** | Historial médico, diagnósticos | Violación de privacidad, discriminación |
| **Datos Empresariales** | Propiedad intelectual, planes estratégicos | Competencia desleal, pérdidas económicas |
| **Datos Regulatorios** | Información regulada por leyes (ej. GDPR) | Multas, sanciones legales |

## 4. Niveles de Sensibilidad

* **Alto:** Información crítica que requiere máxima protección.
* **Medio:** Información que debe protegerse, pero con menor impacto si se filtra.
* **Bajo:** Información que puede ser pública o tiene bajo impacto si se expone.

## 5. Pasos para Clasificar Datos

1. Identificar el tipo de dato.
2. Evaluar impacto potencial de su exposición.
3. Determinar requisitos legales o regulatorios.
4. Asignar nivel de sensibilidad.
5. Implementar controles y políticas de protección.

# Firma de Tokens, Cifrado en Tránsito (HTTPS) y Seguridad en Reposo

## 1. Firma de Tokens (ejemplo: JWT firmado)

* El token (como JWT) contiene datos (payload) y una firma digital.
* **Firma digital:** Garantiza que el token no ha sido modificado y que proviene de una fuente confiable.
* El payload está **codificado (base64) pero no cifrado**, por lo que es legible si alguien accede al token.

## 2. Cifrado en Tránsito con HTTPS (TLS)

* Cuando el token viaja entre cliente y servidor, la comunicación se protege con **TLS**.
* TLS cifra todos los datos intercambiados, incluyendo el token, evitando que terceros puedan interceptar y leer la información.
* Esto protege el token y otros datos sensibles durante la transmisión.

## 3. Token estático en reposo

* Una vez que el token está guardado en el cliente (por ejemplo, en localStorage o cookies) o en el servidor (logs, bases de datos), el token **no está cifrado por sí mismo**.
* El token puede ser leído por quien tenga acceso al almacenamiento o sistema donde se guardó.
* Por eso, se deben tomar medidas adicionales para proteger el almacenamiento (uso de cookies HttpOnly, almacenamiento seguro, políticas de acceso, etc.).

## Resumen

| **Aspecto** | **Protección y Riesgo** |
| --- | --- |
| Firma del token | Protege integridad y autenticidad |
| Cifrado en tránsito | Protege el token y datos contra espionaje y manipulación durante la comunicación |
| Token estático | No está cifrado por sí mismo, es legible si alguien accede al almacenamiento; requiere protección adicional |

# ¿Cómo resuelven los JOSE/JWA la protección de tokens?

## 1. **JWS (JSON Web Signature) — Firma digital**

* Usando algoritmos JWA para firmar tokens (como RS256, HS256), JWS asegura la **integridad y autenticidad** del token.
* Esto garantiza que el token no ha sido alterado y proviene de quien dice ser.
* **Pero el contenido no está cifrado**, por lo que puede ser leído por quien tenga el token.

## 2. **JWE (JSON Web Encryption) — Cifrado**

* JWE utiliza algoritmos JWA para **cifrar el contenido** del token o mensaje.
* Esto protege la confidencialidad del payload, haciéndolo ilegible sin la clave correcta.
* Ideal para proteger datos sensibles en reposo o tránsito.

## 3. **JWA (JSON Web Algorithms)**

* Define los algoritmos criptográficos usados para firmar (JWS) y cifrar (JWE), como RSA, ECDSA, AES-GCM, etc.
* Permite elegir algoritmos seguros y estándares para proteger tokens.

## En resumen

| **Estándar** | **Función** | **Protección que ofrece** |
| --- | --- | --- |
| JWS | Firma digital | Integridad y autenticidad (no confidencialidad) |
| JWE | Cifrado | Confidencialidad y protección de datos |
| JWA | Algoritmos para JWS/JWE | Seguridad criptográfica para firmar y cifrar |

## ¿Por qué es importante?

* Si solo firmas el token (JWS), cualquiera puede leer el payload si obtiene el token.
* Si usas cifrado (JWE), solo quien tenga la clave puede leer el contenido, aumentando la seguridad para datos sensibles.

## Bibliografía y Normativas Internacionales Relevantes

### 1. ****RFCs de JOSE (JSON Object Signing and Encryption)****

* **RFC 7515:** JSON Web Signature (JWS)
* **RFC 7516:** JSON Web Encryption (JWE)
* **RFC 7517:** JSON Web Key (JWK)
* **RFC 7518:** JSON Web Algorithms (JWA)  
  Estas RFCs son las especificaciones oficiales sobre cómo firmar, cifrar y manejar tokens y claves de forma segura. Son la base técnica para librerías como **jose**.

### 2. ****OWASP (Open Web Application Security Project)****

* Proporciona guías y buenas prácticas para el desarrollo seguro de aplicaciones.
* En particular, el proyecto OWASP Top 10 lista las vulnerabilidades más críticas que deben evitarse.
* OWASP también tiene guías específicas sobre autenticación, gestión de sesiones y criptografía.

### 3. ****ISO/IEC 27001 e ISO/IEC 27002****

* Normas internacionales para sistemas de gestión de seguridad de la información (SGSI).
* Establecen controles y buenas prácticas para proteger la información, incluyendo datos sensibles.

### 4. ****GDPR (Reglamento General de Protección de Datos, Unión Europea)****

* Normativa legal que regula el tratamiento y protección de datos personales en la UE.
* Define qué se considera dato personal y sensible, y obliga a implementar medidas técnicas y organizativas para protegerlos.

### 5. ****NIST (National Institute of Standards and Technology)****

* Publica guías de seguridad y criptografía, como el NIST Special Publication 800-63 (Autenticación Digital).
* Proporciona recomendaciones sobre algoritmos criptográficos y gestión de identidad.

## Recomendación

* Para diseño y desarrollo seguro, combinar estas fuentes: seguir las RFCs técnicas, aplicar buenas prácticas OWASP, cumplir normativas legales como GDPR y adoptar estándares ISO y NIST para la gestión.

## Elementos Principales en el Diseño de Proyectos de Software

### 1. ****Requisitos del Proyecto****

* Funcionales: Qué debe hacer el sistema.
* No funcionales: Rendimiento, seguridad, usabilidad, escalabilidad.

### 2. ****Arquitectura del Sistema****

* Definición de componentes, módulos y su interacción.
* Selección de patrones arquitectónicos (por ejemplo, MVC, microservicios).
* Decisiones sobre tecnología, bases de datos, APIs.

### 3. ****Diseño de Datos****

* Modelado de la base de datos.
* Estructura de datos y esquemas.
* Integridad y relaciones.

### 4. ****Diseño de la Interfaz de Usuario (UI/UX)****

* Wireframes y prototipos.
* Experiencia de usuario.
* Accesibilidad y responsividad.

### 5. ****Diseño de la Seguridad****

* Autenticación y autorización.
* Manejo de datos sensibles.
* Políticas de acceso y cifrado.

### 6. ****Planificación de Testing****

* Tipos de pruebas: unitarias, integración, sistema, aceptación.
* Automatización de pruebas.
* Definición de criterios de calidad.

### 7. ****Planificación de Implementación y Despliegue****

* Estrategias de desarrollo: iterativo, incremental.
* Integración continua / entrega continua (CI/CD).
* Gestión de versiones y releases.

### 8. ****Documentación****

* Documentos de diseño, arquitectura, APIs y usuarios.
* Manuales y guías.

### 9. ****Gestión del Proyecto****

* Cronograma, recursos, riesgos.
* Metodologías ágiles o tradicionales.