Comunicación Técnica Efectiva para Desarrolladores

Manual del Estudiante

MODULO 0

LECCIÓN 0.1: PRECONCEPTOS FUNDAMENTALES DE LA COMUNICACIÓN TÉCNICA

¡Hola Futuro Technical Writer! 👋

Soy **GemPreConceptos**, tu profesor de nivelación. Antes de sumergirnos en la redacción de RFCs complejos o ADKs, necesitamos calibrar nuestra brújula. He analizado el plan curricular y detectado 3 pilares invisibles que sostienen todo el curso. Si estos cimientos fallan, el edificio se cae.

Aquí tienes los conceptos "Boleto de Entrada" para este viaje:

---

1. DOCS AS CODE (Documentación como Código)

📝 Definición

Es la filosofía de tratar la documentación con el mismo rigor y herramientas que el código fuente.

💡 ¿Por qué es fundamental?

Imagina que construyes una casa (tu software) pero los planos (la documentación) están dibujados en servilletas dispersas (Word, Wikis desactualizadas, emails). Si cambias una pared, nadie actualiza la servilleta. **Docs as Code** significa guardar los planos en la misma caja fuerte que los ladrillos (Git), revisarlos con inspectores (Code Review) y publicarlos automáticamente cuando la casa se termina (CI/CD).

⭐ Importancia: 10/10

Sin esto, la documentación siempre estará desincronizada de la realidad.

---

2. DEUDA TÉCNICA DOCUMENTAL

📝 Definición

El costo implícito de no documentar una decisión o cambio en el momento en que ocurre.

💡 ¿Por qué es fundamental?

Es como lavar los platos. Si lavas tu plato (documentas tu cambio) justo después de comer, toma 30 segundos. Si esperas al final del mes (el lanzamiento), tienes una montaña de platos sucios con comida pegada. El esfuerzo para "ponerse al día" es exponencialmente mayor y mucho más doloroso.

⭐ Importancia: 9/10

Entender este costo te motivará a escribir *mientras* programas, no después.

---

3. VOZ ACTIVA vs. VOZ PASIVA

📝 Definición

- **Voz Activa**: El sujeto realiza la acción. ("El sistema valida los datos").
- **Voz Pasiva**: La acción es recibida por el sujeto. ("Los datos son validados por el sistema").

💡 ¿Por qué es fundamental?

En código, buscamos la ruta más corta de A a B. En escritura técnica, la **Voz Activa** es esa ruta directa ($O(1)$). La Voz Pasiva es un algoritmo ineficiente ($O(n^2)$) que obliga al cerebro del lector a procesar quién hizo qué.

- *Malo*: "El error debe ser corregido por el usuario." (¿Quién manda? ¿Qué pasa?)
- *Bueno*: "Corrija el error." (Instrucción clara, sujeto implícito "usted").

⭐ Importancia: 8/10

Es la herramienta #1 para la claridad.

🛠️ TU PRIMERA MISIÓN (CHECKLIST DE NIVELACIÓN)

Antes de pasar al Módulo 1, asegúrate de:

1. [ ] Saber qué es **Markdown** (si no, busca un tutorial de 5 min).
2. [ ] Tener un editor de texto (VS Code recomendado).
3. [ ] Entender que escribir es una iteración, igual que programar (Draft → Review → Refactor → Deploy).

¡Estás listo para dejar de ser solo un “coder” y convertirte en un **Ingeniero Comunicador**! 🚀

MODULO 1

TEMA 1.1.1: ELIMINACIÓN DE VOZ PASIVA

Tiempo estimado: 30 minutos **Nivel**: Intermedia **Prerrequisitos**: Conceptos de Módulo 0 (Agencia gramatical)

¿Por qué importa este concepto?

En la documentación técnica, la ambigüedad se considera un defecto crítico (bug). Cuando se escribe una frase como "Los datos fueron procesados", se oculta información vital para el lector: ¿Quién o qué procesó los datos? ¿Fue el usuario manualmente? ¿El backend automáticamente? ¿Un proceso en segundo plano?

La **Voz Pasiva** diluye la responsabilidad y aumenta lo que llamamos "carga cognitiva": el esfuerzo mental necesario para procesar la información. Al eliminarla, logramos tres objetivos fundamentales de ingeniería:

1. **Precisión**: Se define explícitamente el actor del sistema.
2. **Concisión**: Las oraciones activas son, en promedio, un 20-30% más cortas.
3. **Accionabilidad**: El usuario sabe exactamente qué debe hacer o qué esperar del sistema.

Ejemplo real: En la documentación de plataformas como Stripe o Google Cloud, raramente encontrará construcciones como "El pago es recibido". En su lugar, encontrará "Stripe recibe el pago".

Conexión con conocimientos previos

En el Módulo 0 establecimos que la escritura técnica busca la ruta más eficiente para transmitir información. Podemos usar la analogía de la complejidad algorítmica:

- La **Voz Activa** es una operación de complejidad constante ($O(1)$): el cerebro conecta "Sujeto > Acción" (Acceso directo).

- La **Voz Pasiva** es una operación más costosa, similar a $O(n^2)$, porque obliga al cerebro a buscar en el contexto quién realizó la acción.

Comprensión intuitiva

Para los desarrolladores, es útil pensar en la gramática como si fuera código:

- **Voz Pasiva** es comparable a ejecutar una función anónima sin argumentos claros: `process_stuff()`. ¿Qué entra? ¿Qué contexto usa? El comportamiento es opaco.
- **Voz Activa** es comparable a una función con tipos explícitos: `User.click(Button)`. La estructura es `Sujeto -> Verbo -> Objeto`.

Ejemplo motivador

Imagine que un desarrollador lee lo siguiente en un archivo README:

> "La configuración debe ser actualizada antes del despliegue."

El desarrollador se detiene y pregunta: "¿Por quién? ¿Debo hacerlo yo manualmente? ¿Lo hace el pipeline de CI/CD automáticamente?". Esta duda detiene el flujo de trabajo y puede provocar errores en producción.

Definición formal

La **Voz Activa** sigue una estructura gramatical directa y lineal: $$ Sujeto + Verbo + Objeto $$

La **Voz Pasiva** invierte esta estructura, convirtiendo al objeto de la acción en el sujeto gramatical, generalmente introduciendo el verbo auxiliar "ser": $$ Objeto + ser + participio (+ por + Agente) $$

Propiedades fundamentales

1. **Agencia Explícita**: El actor siempre precede a la acción.
2. **Direccionalidad**: El flujo de información es lineal (de izquierda a derecha).
3. **Economía Léxica**: Elimina verbos auxiliares innecesarios como "fue", "ha sido" o "es".

Implementación práctica

En ingeniería de software no siempre es posible evitar la voz pasiva (a veces el actor es irrelevante), pero debemos refactorizar agresivamente cuando la acción es crítica.

Algoritmo de Refactorización

Para corregir frases pasivas, siga este procedimiento paso a paso:

1. **Detectar**: Busque construcciones del tipo “ser + participio” (ejemplos: es hecho, fue enviado, será cancelado).
2. **Identificar Agente**: Pregúntese “¿Quién realiza la acción?”.
3. **Reubicar**: Mueva ese Agente al inicio de la oración.
4. **Eliminar Auxiliares**: Borre los verbos auxiliares “ser” o “estar”.

Ejemplos de Refactorización (Antes vs. Después)

Caso 1: Instrucciones al Usuario

Pasiva (Incorrecto):

> “El botón ‘Deploy’ deberá ser presionado cuando los tests hayan pasado.”
> *Problema: Ambigüedad. ¿Es automático o manual?*

Activa (Correcto):

> “Presione el botón ‘Deploy’ una vez que pasen los tests." *Solución: Instrucción imperativa directa al lector.*

Caso 2: Descripción del Sistema

Pasiva (Incorrecto):

> “Una excepción es lanzada por el servidor si el token es inválido." *Problema: Excesivamente largo (12 palabras).*

Activa (Correcto):

> “El servidor lanza una excepción si el token es inválido." *Solución: Conciso (9 palabras). Ahorro del 25% de texto.*

Práctica Guiada

Intente refactorizar este párrafo técnico común:

Texto Original (Legacy): “Los logs son rotados cada 24 horas por el sistema. Si el espacio en disco es excedido, una alerta será enviada al administrador.”

Análisis de Refactorización:

1. *Segmento 1*: “son rotados por el sistema” → Agente identificado: “el sistema”.
2. *Segmento 2*: “espacio es excedido” → Agente implícito: “los logs”.
3. *Segmento 3*: “alerta será enviada” → Agente: “el sistema”.

Texto Refactorizado: “El sistema rota los logs cada 24 horas. Si los logs exceden el espacio en disco, el sistema envía una alerta al administrador.”

Errores frecuentes

Error 1: La pasiva “Zombie”

Ocurre cuando se oculta al actor deliberadamente para no sonar “agresivo” o directo.

- *Incorrecto*: “Se cometieron errores en la migración.” (Nadie asume la culpa).
- *Correcto*: “El equipo subestimó la carga en la migración.” (Honestidad y responsabilidad).

Error 2: Confundir Pasiva con Estado

No toda construcción “ser + participio” es pasiva. A veces describe un estado del sistema.

- *Válido*: “El sistema está caído.” (Esto describe un estado, no una acción pasiva).

Resumen del concepto

En una frase: Coloque al actor (Sujeto) antes de la acción (Verbo) para eliminar dudas sobre la responsabilidad.

Cuándo usarlo: En la gran mayoría instrucciones, descripciones de procesos y mensajes de error.

Excepción: Úsela solo cuando el actor es desconocido o irrelevante (ejemplo: “La función fue deprecada en v2.0”, donde no importa qué desarrollador específico la deprecor).

Siguiente paso: Estructura de pirámide invertida (Tema 1.1.2).

BANCO DE EJERCICIOS: MÓDULO 1 - ELIMINACIÓN DE VOZ PASIVA

METADATA

- **Módulo**: Principios de Escritura Técnica
- **Objetivos evaluados**:
    1. Identificar construcciones pasivas.
    2. Transformar oraciones pasivas a activas sin perder significado.
    3. Aplicar criterio de agencia en párrafos complejos.
- **Tiempo total estimado**: 25 minutos
- **Tipo**: Formativa
- **Nivel de ruta**: Intermedia

EJERCICIO 1: Detección de Agentes Ocultos

METADATA

- **ID**: `EJ-MOD1-001`
- **Dificultad**: ⭐ Básico
- **Tiempo estimado**: 5 minutos
- **Nivel Bloom**: Recordar/Comprender
- **Tipo**: Análisis gramatical

ENUNCIADO

Identifica cuáles de las siguientes oraciones están en **Voz Pasiva** y subraya el verbo auxiliar ("ser").

1. "El servidor de integración continua ha compilado el proyecto exitosamente."
2. "Los errores fueron ignorados por el script de migración."
3. "La base de datos será restaurada automáticamente a las 03:00 AM."
4. "Estamos monitorizando la latencia del API Gateway."
5. "Se ha decidido deprecar la versión 1.0 de la librería."

RESPUESTA MODELO

1. **Activa**. (Sujeto: El servidor).

2. **Pasiva**. (“**fueron** ignorados”). Agente: script de migración.
3. **Pasiva**. (“**será** restaurada”). Agente: Oculto/Implícito (el sistema).
4. **Activa**. (Sujeto implícito: Nosotros/El equipo).
5. **Pasiva**. (“**ha** decidido” → Forma impersonal “Se”). Agente: Oculto (El equipo de producto).

EJERCICIO 2: Refactorización Atómica

METADATA

- **ID**: `EJ-MOD1-002`
- **Dificultad**: ⭐ ⭐ Intermedio
- **Tiempo estimado**: 10 minutos
- **Nivel Bloom**: Aplicar
- **Tipo**: Reescritura

ENUNCIADO

Reescribe las siguientes oraciones técnicas para usar **Voz Activa**. Si el agente no es explícito, infiere uno lógico (ej: “el sistema”, “el usuario”, “el administrador”).

Input:

1. “El acceso es denegado si la contraseña es introducida incorrectamente tres veces.”
2. “Un correo de confirmación será enviado al usuario una vez que el pago sea procesado.”
3. “Los datos son encriptados por el cliente antes de ser enviados al servidor.”

SOLUCIÓN MODELO

1. **Input**: “El acceso es denegado si la contraseña es introducida incorrectamente tres veces.” **Solución**: “El sistema deniega el acceso si el usuario introduce la contraseña incorrectamente tres veces.” *(Cambio: Agregamos “El sistema” y “el usuario”).*

2. **Input**: “Un correo de confirmación será enviado al usuario una vez que el pago sea procesado.” **Solución**: “El sistema envía un correo de confirmación al usuario una vez que procesa el pago.” *(Cambio: Eliminamos “será” y unificamos la acción).*

3. **Input**: “Los datos son encriptados por el cliente antes de ser enviados al servidor.” **Solución**: “El cliente encripta los datos antes de enviarlos al servidor.” *(Cambio: “El cliente” pasa al inicio).*

EJERCICIO 3: El “Postmortem” Pasivo

METADATA

- **ID**: `EJ-MOD1-003`
- **Dificultad**: ⭐ ⭐ ⭐ Avanzado
- **Tiempo estimado**: 10 minutos
- **Nivel Bloom**: Evaluar/Crear
- **Tipo**: Edición de párrafo

ENUNCIADO

El siguiente fragmento de un Postmortem oculta responsables y acciones clave. Reescríbelo en voz activa para clarificar la cronología del incidente.

Texto Original (Pasivo):

> “A las 14:00, una caída en la latencia fue detectada por el equipo de SRE. Una investigación fue iniciada. Se descubrió que una configuración errónea había sido aplicada al balanceador de carga. La configuración fue revertida y el servicio fue restaurado a las 14:15.”

Tarea:

1. Identificar los agentes (SRE, configuración, balanceador).
2. Reescribir en voz activa.
3. Mantener el tono profesional (sin culpar, solo reportar hechos).

RÚBRICA DE EVALUACIÓN

Criterio	Puntos	Descripción
Agencia	40%	Todos los verbos principales tienen sujeto explícito.
Concisión	30%	El texto resultante es más corto o igual en longitud.
Claridad	30%	La secuencia causa-efecto es lineal.

SOLUCIÓN MODELO

> “A las 14:00, el equipo de SRE detectó una caída en la latencia e inició una investigación. El equipo descubrió que alguien aplicó una configuración errónea

> al balanceador de carga. SRE revirtió la configuración y restauró el servicio a las 14:15.”

(Nota: En un entorno real "blameless", podríamos usar "El sistema tenía una configuración errónea", pero la acción de "aplicar" y "revertir" debe ser activa).

ERRORES COMUNES

❌ **Error: La Pasiva defensiva**

> "Se cometió un error en la configuración." **Diagnóstico**: Miedo a señalar la causa. **Corrección**: "Una configuración errónea causó el fallo." (El objeto inanimado puede ser el sujeto activo).

AUTOEVALUACIÓN

- [ ] ¿Entiendo la diferencia entre sujeto gramatical y actor lógico?
- [ ] ¿Puedo reducir el conteo de palabras eliminando "ser/estar"?
- [ ] ¿Sé cuándo es aceptable usar la pasiva (ej. actor desconocido)?

EVALUACIÓN: ELIMINACIÓN DE VOZ PASIVA

FICHA TÉCNICA

- **Tema**: 1.1.1 Eliminación de Voz Pasiva
- **Tiempo límite**: 10 minutos
- **Puntaje total**: 100 puntos (20 pts por pregunta)
- **Nivel**: Intermedio

CUESTIONARIO

Pregunta 1: Identificación

¿Cuál de las siguientes oraciones utiliza correctamente la **Voz Activa**?

- [ ] **a)** Los datos son validados por el controlador antes de ser guardados.
- [ ] **b)** El controlador valida los datos antes de guardarlos.
- [ ] **c)** Se validan los datos en el controlador.

- [ ] **d)** La validación es realizada por el controlador.

Pregunta 2: Impacto Cognitivo

Según la analogía del curso, ¿por qué la Voz Pasiva es menos eficiente para el lector?

- [ ] **a)** Porque ocupa menos espacio en disco.
- [ ] **b)** Porque tiene una complejidad equivalente a O(n^2) cognitiva.
- [ ] **c)** Porque es gramaticalmente incorrecta en español.
- [ ] **d)** Porque suena menos profesional.

Pregunta 3: Refactorización

Selecciona la mejor refactorización para: *"El despliegue fue abortado por el sistema de seguridad debido a un error."*

- [ ] **a)** El sistema de seguridad abortó el despliegue debido a un error.
- [ ] **b)** Debido a un error, el despliegue fue abortado.
- [ ] **c)** Se abortó el despliegue por un error en el sistema de seguridad.
- [ ] **d)** El error causó que el despliegue fuera abortado.

Pregunta 4: Agentes Ocultos

En la frase *"Se decidió aumentar el timeout a 30 segundos"*, ¿cuál es el problema principal?

- [ ] **a)** La frase es demasiado larga.
- [ ] **b)** "Decidió" es un verbo débil.
- [ ] **c)** No se especifica quién tomó la decisión (Agencia oculta).
- [ ] **d)** El timeout es muy alto.

Pregunta 5: Excepciones

¿En qué escenario es **aceptable** usar Voz Pasiva?

- [ ] **a)** Cuando queremos sonar más académicos.
- [ ] **b)** Cuando el actor es desconocido o irrelevante para el contexto.
- [ ] **c)** En los títulos de los documentos.
- [ ] **d)** Nunca, está prohibida en ingeniería.

SOLUCIONARIO

Respuesta 1: b) El controlador valida los datos antes de guardarlos.

> **Justificación**: Sigue la estructura Sujeto (Controlador) → Verbo (Valida) → Objeto (Datos). Las otras opciones ocultan o posponen al actor.

Respuesta 2: b) Porque tiene una complejidad equivalente a O(n^2) cognitiva.

> **Justificación**: El cerebro debe esperar al final de la frase para identificar al actor y reconstruir la relación, aumentando la carga mental.

Respuesta 3: a) El sistema de seguridad abortó el despliegue debido a un error.

> **Justificación**: Coloca al agente responsable (Sistema de seguridad) al principio, tomando control de la acción.

Respuesta 4: c) No se especifica quién tomó la decisión (Agencia oculta).

> **Justificación**: El uso de "Se + verbo" (pasiva refleja) difumina la responsabilidad. ¿Fue el equipo? ¿El CTO? ¿Un proceso automático?

Respuesta 5: b) Cuando el actor es desconocido o irrelevante para el contexto.

> **Justificación**: Ejemplo: "El servidor se reinició". No importa qué electrón específico causó el reinicio, importa el estado resultante.

TEMA 1.1.2: ESTRUCTURA DE PIRÁMIDE INVERTIDA

Tiempo estimado: 15 minutos **Nivel**: Principiante **Prerrequisitos**: Tema 1.1.1 (Voz Pasiva)

¿Por qué importa este concepto?

En la escuela nos enseñaron a escribir ensayos: Introducción → Desarrollo → Clímax → Conclusión. Esto funciona para novelas de misterio. Para documentación técnica, **es un desastre**.

Un ingeniero que busca un error a las 3 AM no quiere misterio. Quiere la solución **YA**. Si escondes la respuesta al final del párrafo, estás quemando su tiempo.

Conexión con conocimientos previos

- **Tema 1.1.1**: Usar voz activa hace que las oraciones sean directas. La Pirámide Invertida hace que el *párrafo* sea directo.
- **TL;DR** (Too Long; Didn't Read): Es la aplicación más pura de este concepto.

Comprensión intuitiva

Imagina que eres un periodista de 1920 enviando una noticia por telégrafo durante la guerra. La línea puede cortarse en cualquier momento. Tienes que decir lo más importante **primero**.

1. "El Titanic se hundió". (Lead)
2. "Murieron 1500 personas". (Detalles clave)
3. "La orquesta tocó hasta el final". (Color/Contexto)

Si se corta la línea después de la frase 1, la noticia principal llegó.

Definición formal

Implementación práctica

Ejemplo: Reporte de Estatus

Mal (Cronológico):

> "Ayer revisamos los logs y vimos errores. Luego reiniciamos el servidor. Pareció funcionar, pero luego falló de nuevo. Investigamos la DB y vimos bloqueos. Al final, liberamos los bloqueos y el sistema subió."

Bien (Pirámide Invertida):

> **El sistema está estable ahora.** Causa raíz: Bloqueos en la base de datos que liberamos manualmente. *Cronología*: Detectamos errores ayer, reiniciamos (fallido), encontramos bloqueos y resolvimos.

Ejemplo: Documentación de Función

Mal:

> "Esta función primero inicializa un buffer, luego itera sobre el array, verifica nulos... y finalmente retorna el promedio."

Bien:

> **Retorna el promedio de un array de números.** (Ignora nulos automáticamente). *Detalles de implementación*: Usa un buffer interno para...

Errores frecuentes

Error 1: "Enterrar el Lead" (Burying the Lead)

Empezar con "Debido a los recientes cambios en la arquitectura de microservicios..." en lugar de "El endpoint /users está deprecado".

Error 2: Asumir que el lector leerá todo

Los usuarios **escanean** (patrón de lectura en F). Solo leen las primeras palabras de cada párrafo. Si la información importante está en la línea 5, es invisible.

Resumen del concepto

En una frase: Dile al lector qué pasó antes de explicarle cómo pasó.

Regla de Oro: Si el usuario deja de leer después de la primera frase, ¿se llevó la información vital? Si es sí, usaste bien la pirámide.

Siguiente paso: Ejercicios para reescribir textos cronológicos.

BANCO DE EJERCICIOS: MÓDULO 1 - PIRÁMIDE INVERTIDA

METADATA

- **Módulo**: Principios de Escritura Técnica
- **Objetivos evaluados**:
    1. Identificar "leads" enterrados en textos densos.
    2. Redactar resúmenes ejecutivos (TL;DR).
    3. Reestructurar actualizaciones de estado cronológicas.
- **Tiempo total estimado**: 20 minutos
- **Nivel**: Básico/Intermedio

EJERCICIO 1: Arqueología del Lead

ENUNCIADO

Lee el siguiente correo y subraya la única frase que realmente importa (el "Lead").

> "Hola equipo, espero que estén bien. Estuve revisando el ticket sobre la latencia en el dashboard. Hice varias pruebas de carga con JMeter y al principio todo parecía normal. Luego noté que el uso de CPU subía mucho en la instancia de reportes. Después de perfilar el código, vi que el bucle de generación de PDF no está paginado. Básicamente, el servidor se queda sin memoria cuando el reporte es grande. Así que **necesitamos posponer el lanzamiento de mañana** hasta arreglarlo."

SOLUCIÓN

Lead Enterrado: "Necesitamos posponer el lanzamiento de mañana".

EJERCICIO 2: El Arte del TL;DR

ENUNCIADO

Tienes un RFC de 10 páginas proponiendo migrar de Jenkins a GitHub Actions. Escribe un bloque **TL;DR** de 3 líneas para poner al inicio del documento.

Datos clave:

- Costo: Ahorro de $500/mes.
- Esfuerzo: 2 sprints.
- Beneficio: Builds 40% más rápidos.
- Riesgo: Curva de aprendizaje del equipo.

SOLUCIÓN MODELO

```
## TL;DR
Propuesta para migrar CI/CD a GitHub Actions.
- **Beneficio**: Reducción del tiempo de build en 40% y ahorro de $500/mes.
- **Costo**: 2 sprints de ingeniería.
- **Recomendación**: Proceder con PoC en Q3.
```

EJERCICIO 3: Refactorización de Status Report

ENUNCIADO

Transforma este update de Slack cronológico en una Pirámide Invertida.

Texto Original:

> "@channel Ayer empecé a trabajar en la integración con la API de pagos. Leí la documentación y vi que cambió la autenticación. Intenté usar nuestras llaves viejas y fallaron. Tuve que generar nuevas llaves en el portal de developer. Luego actualicé las variables de entorno en staging. Finalmente pude hacer un pago de prueba. Así que la integración ya funciona en staging."

SOLUCIÓN MODELO

> "@channel **Integración de Pagos completada en Staging**. ⭐ Ya pueden probar el flujo de checkout en el entorno de pruebas.
>
> **Detalles**:
>
> - Se actualizaron las API Keys (el proveedor cambió el método de auth).

- Variables de entorno `STRIPE_KEY` actualizadas en Staging."

AUTOEVALUACIÓN

- [ ] ¿Mis correos empiezan con la conclusión?
- [ ] ¿Uso negritas para destacar la acción requerida?

EVALUACIÓN: PIRÁMIDE INVERTIDA

FICHA TÉCNICA

- **Tema**: 1.1.2 Estructura de Pirámide Invertida
- **Nivel**: Básico

CUESTIONARIO

Pregunta 1

¿Cuál es el objetivo principal de la Pirámide Invertida en documentación técnica?

- [ ] **a)** Hacer que los documentos se vean más estéticos.
- [ ] **b)** Permitir que el lector obtenga la información crítica sin leer todo el texto.
- [ ] **c)** Ocultar los detalles técnicos para no asustar al usuario.

Pregunta 2

¿Qué significa **TL;DR**?

- [ ] **a)** Too Long; Don't Rewrite
- [ ] **b)** Technical Log; Data Recovery
- [ ] **c)** Too Long; Didn't Read

Pregunta 3

¿Dónde debe ir la **conclusión** o acción requerida en un correo de ingeniería?

- [ ] **a)** Al final, después de presentar toda la evidencia.
- [ ] **b)** En el asunto y en la primera línea.

- [ ] **c)** En un archivo adjunto.

Pregunta 4

Selecciona el mejor título según el principio de “Inform Fast”:

- [ ] **a)** “Informe de estado semanal del proyecto Alpha”
- [ ] **b)** “Retraso en el módulo de pagos por bloqueo en API de terceros”
- [ ] **c)** “Actualización importante sobre el avance”

SOLUCIONARIO

1. **b)**. La escaneabilidad y la entrega inmediata de valor son la clave.
2. **c)**. Es un resumen ejecutivo para quienes no tienen tiempo de leer todo.
3. **b)**. Front-loading: la información vital va primero.
4. **b)**. El título debe contener la noticia, no solo la categoría.

TEMA 1.2.1: MAPEO DE AUDIENCIA

Tiempo estimado: 20 minutos **Nivel**: Intermedio **Prerrequisitos**: Tema 1.1.X (Claridad y Estructura)

¿Por qué importa este concepto?

El error #1 en documentación técnica no es la gramática, es la **Falta de Empatía**. Escribirle a un Senior Architect como si fuera un Junior, o explicarle a un Usuario Final cómo funciona la base de datos interna.

Si no defines tu audiencia, escribes para nadie. Distinguir entre **Usuario** (quien usa tu código) y **Mantenedor** (quien arregla tu código) es la diferencia entre una librería exitosa y “abandonware”.

Conexión con conocimientos previos

En programación orientada a objetos, tenemos `public` (interfaz para el usuario) y `private` (implementación para el mantenedor). La documentación debe respetar ese mismo encapsulamiento.

- **Docs de Usuario**: API pública. Contrato. Cómo se usa.
- **Docs de Mantenedor**: Internals. Lógica de negocio. Por qué se hizo así.

Comprensión intuitiva

Imagina que compras un coche.

1. **Manual del Conductor (Usuario)**: Cómo arrancar, cómo poner aire acondicionado, qué gasolina usa. (Caja Negra).
2. **Manual de Taller (Mantenedor)**: Torque de los tornillos de la culata, diagrama eléctrico del alternador. (Caja Blanca).

Si el manual del conductor explicara el diagrama eléctrico en la página 1, nadie conduciría. Si el manual de taller solo dijera "gire la llave", nadie podría repararlo.

Definición formal

El **Mapeo de Audiencia** es el proceso de identificar el modelo mental y las necesidades del lector antes de escribir.

Matriz de Audiencia Técnica

Dimensión	USUARIO (Consumidor)	MANTENEDOR (Colaborador)
Objetivo	Resolver un problema rápido.	Entender/Modificar el sistema.
Perspectiva	Caja Negra (Black Box).	Caja Blanca (White Box).
Pregunta clave	"¿Cómo hago X?"	"¿Por qué X funciona así?"
Artefacto	README, Guías, API Reference.	CONTRIBUTING, Architecture Docs, Comentarios inline.

Implementación práctica

Algoritmo de Segmentación

Antes de escribir un documento, ejecuta este `switch`:

1. **Caso: README / Tutorial → Modo Usuario**.
    - *Ocultar*: Detalles de implementación, dependencias internas, deuda técnica.
    - *Mostrar*: Instalación, configuración, ejemplos de uso (Happy Path).
2. **Caso: Design Doc / Pull Request → Modo Mantenedor**.
    - *Ocultar*: Marketing, promesas vacías.
    - *Mostrar*: Trade-offs, riesgos, complejidad algorítmica ($O(n)$), alternativas rechazadas.

Ejemplo Comparativo: Endpoint de Login

Versión Usuario (Swagger/Docs)

> **POST /login** Autentica un usuario y devuelve un JWT.
>
> - **Input**: `{user, pass}`
> - **Output**: `200 OK { token }`
> - **Error**: `401 Invalid Credentials`

(Nota: Al usuario no le importa si usas un hash bcrypt o argon2, solo quiere su token).

Versión Mantenedor (Comentario/Wiki interna)

> **Módulo de Autenticación** Usamos `bcrypt` con work factor 12. **Warning**: No aumentar el factor a >14 porque la latencia de login excede los 500ms y rompe el SLA. La validación de password ocurre *antes* de la conexión a DB para mitigar ataques DoS.

Errores frecuentes

Error 1: “Leaking Internals” (La Fuga)

Explicar la implementación en la guía de usuario.

- *Mal*: “Para guardar el archivo, instanciamos un `FileStream` con buffer de 4kb...”
- *Bien*: “Para guardar el archivo, llame a `save()`.”

Error 2: Asumir el contexto del autor

Escribir pensando que el lector lleva 6 meses en el proyecto.

- *Síntoma*: “Ejecuta el script de siempre.”

- *Solución*: "Ejecuta `./scripts/deploy.sh`."

Resumen del concepto

En una frase: Decide si estás escribiendo para quien *conduce* el coche (Usuario) o para quien *repara* el motor (Mantenedor), y no mezcles los manuales.

Cuándo usarlo: Siempre. Define tu audiencia en la primera línea mentalmente.

Prerrequisito crítico: Teoría de la Mente (capacidad de atribuir desconocimiento a otros).

Siguiente paso: Módulo 2 - Documentación en el Código (donde aplicaremos esto a Comentarios vs. Docstrings).

BANCO DE EJERCICIOS: MÓDULO 1 - MAPEO DE AUDIENCIA

METADATA

- **Módulo**: Principios de Escritura Técnica
- **Objetivos evaluados**:
  1. Diferenciar necesidades de Usuario vs. Mantenedor.
  2. Identificar el nivel de abstracción adecuado por audiencia.
  3. Reescribir contenido técnico para diferentes stakeholders.
- **Tiempo total estimado**: 20 minutos
- **Nivel**: Intermedio

EJERCICIO 1: Clasificación de Documentos

ENUNCIADO

Clasifica los siguientes fragmentos según su audiencia ideal: **(U) Usuario** o **(M) Mantenedor**.

1. "La función `retry_logic` usa un backoff exponencial para no saturar el servidor aguas arriba."

2. “Para obtener un API Key, navega a Settings > Developer Tools y haz clic en ‘Generate New Key’.”
3. “Advertencia: Cambiar el tamaño del pool de conexiones requiere reiniciar la instancia.”
4. “Migramos de Axios a Fetch porque Axios añadía 15kb al bundle size innecesariamente.”
5. “Este endpoint devuelve código 429 si haces más de 100 peticiones por minuto.”

SOLUCIÓN

1. **(M)**. Explica el *cómo* interno y la justificación de estabilidad.
2. **(U)**. Instrucción paso a paso de uso.
3. **(U)**. Advertencia operativa para quien administra el sistema.
4. **(M)**. Justificación de decisión técnica/arquitectura.
5. **(U)**. Contrato de interfaz (Límite de uso).

EJERCICIO 2: El Camaleón Técnico

ENUNCIADO

Tienes un incidente: “La base de datos se cayó por falta de disco”. Escribe un mensaje de 1 línea para cada una de las siguientes audiencias:

1. **Usuario Final** (Cliente de la app).
2. **CTO / Manager** (Negocio).
3. **Equipo de DevOps** (Técnico).

SOLUCIÓN MODELO

1. **Usuario Final**: “Estamos experimentando problemas de conexión. El servicio volverá pronto." *(Caja Negra: No mencionar “disco” ni “base de datos”).*

3. **DevOps**: “PostgreSQL master `disk_usage` al 100% en `/var/lib/postgresql`. Necesitamos purgar logs o extender el volumen EBS." *(Caja Blanca: Detalles accionables y técnicos).*

EJERCICIO 3: Depuración de Documentación

ENUNCIADO

Este párrafo está en la guía de “Primeros Pasos” (Onboarding de Usuario). Identifica la frase que **LEAKS INTERNALS** (fuga detalles de implementación) y reescríbela.

> “Para crear una cuenta, envía un POST a `/signup`. El sistema validará tu email usando una Regex compleja que importamos de la librería `email-validator` v2.0 para asegurar que no sea un dominio temporal. Si es válido, recibirás un 201 Created.”

SOLUCIÓN

Fuga: “usando una Regex compleja que importamos de la librería `email-validator` v2.0...” **Por qué**: Al usuario no le importa qué librería usas. **Refactor**: “El sistema validará que tu email tenga un formato correcto y no pertenezca a un dominio temporal.”

AUTOEVALUACIÓN

- [ ] ¿Antes de escribir, me pregunto “¿Qué quiere lograr el lector?”?
- [ ] ¿Sé ocultar la complejidad innecesaria para el usuario final?

EVALUACIÓN: MAPEO DE AUDIENCIA

FICHA TÉCNICA

- **Tema**: 1.2.1 Usuario vs. Mantenedor
- **Nivel**: Intermedio

CUESTIONARIO

Pregunta 1

¿Cuál es la diferencia principal entre un documento para Usuario y uno para Mantenedor?

- [ ] **a)** El de Usuario es más corto.
- [ ] **b)** El de Usuario trata el sistema como Caja Negra; el de Mantenedor como Caja Blanca.
- [ ] **c)** El de Mantenedor solo contiene código.

Pregunta 2: Escenario

Estás escribiendo un `README.md` para una librería pública. ¿Qué información **NO** debería estar en la sección de "Inicio Rápido"?

- [ ] **a)** Comandos de instalación (`npm install`).
- [ ] **b)** Explicación de por qué elegiste el patrón Singleton para la conexión a DB.
- [ ] **c)** Ejemplo de código "Hello World".

Pregunta 3: Anti-patrones

¿Qué es "Leaking Internals" (Fuga de Internos)?

- [ ] **a)** Cuando la memoria RAM se llena.
- [ ] **b)** Cuando expones detalles de implementación irrelevantes en la documentación de usuario.
- [ ] **c)** Cuando publicas contraseñas en GitHub.

Pregunta 4

Si un lector pregunta "¿Cómo extiendo esta clase para añadir una funcionalidad?", ¿qué rol está asumiendo?

- [ ] **a)** Usuario.
- [ ] **b)** Mantenedor / Colaborador.
- [ ] **c)** Project Manager.

SOLUCIONARIO

1. **b)**. La perspectiva de abstracción es la clave.
2. **b)**. Las decisiones de arquitectura pertenecen a `CONTRIBUTING` o `docs/architecture`, no al Quickstart.

4. **b)**. Quiere modificar o extender el código, por lo tanto necesita visión de Caja Blanca.

MODULO 2

TEMA 2.1.1: COMENTARIOS Y DOCSTRINGS (EL POR QUÉ)

Tiempo estimado: 25 minutos **Nivel**: Básico **Prerrequisitos**: Tema 1.2.1 (Usuario vs Mantenedor)

¿Por qué importa este concepto?

El código explica el **Qué** y el **Cómo**. Los comentarios solo deben existir para explicar el **Por Qué**.

Un comentario que repite lo que dice el código es ruido ("Code Smell"). Un comentario que explica una decisión de negocio compleja o una trampa oculta es oro puro. Tu objetivo es escribir código que no necesite comentarios, y luego escribir comentarios donde el código no sea suficiente.

Conexión con conocimientos previos

En el Módulo 1.2.1 aprendimos sobre "Caja Negra" vs "Caja Blanca".

- **Docstrings (Caja Negra)**: Para el Usuario. Explican el contrato (inputs/outputs).
- **Comentarios Inline (Caja Blanca)**: Para el Mantenedor. Explican la implementación (hacks, optimizaciones).

Comprensión intuitiva

Piensa en los subtítulos de una película.

- **Buen comentario**: (Personaje duda antes de abrir) → Subtítulo: "*Teme que el asesino esté dentro*". (Contexto invisible).

El código es la acción. El comentario es el contexto invisible.

Definición formal

1. **Comentario Pragmático**: Aporta información que no existe en la sintaxis del lenguaje (intención, limitaciones de negocio, referencias a tickets de Jira).

2. **Docstring (Documentation String)**: Texto estructurado asociado a una función/clase que describe su interfaz pública.

La Regla de Oro

> "No comentes código malo; reescríbelo." — Brian Kernighan

Antes de escribir `// Calcula el total`, renombra la variable `t` a `total_invoice_amount`.

Implementación práctica

Caso 1: El comentario redundante (Anti-patrón)

```
# Incrementa i en 1
i = i + 1

# Función que obtiene usuarios
def get_users():
    ...
```

Crítica: El lector sabe leer código. Esto insulta su inteligencia y añade deuda de mantenimiento (si el código cambia, el comentario miente).

Caso 2: El comentario de "Por Qué" (Patrón)

```
# Usamos un sleep de 2s porque la API de legacy-bank
# tiene una race condition si reintentamos inmediatamente.
# Ver Ticket JIRA-1234.
time.sleep(2)
```

Valor: Explicación crítica. Si borras el sleep para "optimizar", rompes la integración.

Caso 3: Docstrings Estructurados

Usa formatos estándar (Google, NumPy, Sphinx) para que los IDEs ayuden al usuario.

```
def connect_db(timeout: int = 30) -> Connection:
    """
    Establece conexión con la base de datos principal.

    Args:
        timeout: Segundos máximos de espera. Si excede, lanza TimeoutError.
                 PRECAUCIÓN: No usar >60s en prod para evitar bloqueos de gunicorn.
```

```
    Returns:
        Objeto Connection activo.
    """
```

Guía de Estilo Rápida

1. **TODOs**: Úsalos para deuda técnica reconocida. `// TODO(autor): Refactorizar esto cuando migremos a v2.`
2. **Hacks**: Admite cuando el código es feo. `// FIXME: Solución temporal. Esto es O(n^2), mover a worker en el futuro.`
3. **Referencias**: Linkea a StackOverflow o Docs oficiales. `// Adaptado de: https://stackoverflow.com/...`

Resumen del concepto

En una frase: El código dice qué hace. Los comentarios dicen por qué lo hace así (y por qué no de otra forma).

Cuándo comentar: Cuando tomaste una decisión no obvia, cuando hay un bug conocido de una librería externa, o cuando la lógica es matemáticamente compleja.

Siguiente paso: El README (El documento más importante de tu repositorio).

BANCO DE EJERCICIOS: MÓDULO 2 -

METADATA

- **Módulo**: Documentación en el Código
- **Objetivos evaluados**:
    1. Identificar comentarios redundantes (Anti-patrón).
    2. Refactorizar código para eliminar necesidad de comentarios.
    3. Escribir Docstrings estructurados que aporten valor.
- **Tiempo total estimado**: 20 minutos
- **Nivel**: Básico

EJERCICIO 1: Matando el Ruido

ENUNCIADO

Clasifica los siguientes comentarios como ✅ **ÚTIL** o ❌ **RUIDO**.

1. `i += 1 // Incrementa el contador`
2. `// TODO: Refactorizar a O(n) antes del lanzamiento`
3. `// Esta función obtiene usuarios` (sobre funcion `get_users()`)
4. `// Workaround para bug en Safari < 14 (Ticket #99)`
5. `return x * 0.16 // Calcula IVA`

SOLUCIÓN

1. ❌ **RUIDO**. El código ya dice que incrementa.
2. ✅ **ÚTIL**. Deuda técnica explícita.
3. ❌ **RUIDO**. El nombre de la función ya lo dice.
4. ✅ **ÚTIL**. Contexto crítico externo (Why).
5. ❌ **RUIDO**. Mejor renombrar variable: `return subtotal * TAX_RATE`.

EJERCICIO 2: Refactorización (Code > Comments)

ENUNCIADO

Elimina los comentarios del siguiente código renombrando variables y funciones para que sea auto-explicativo.

Código Original:


```
# Comprueba si el usuario puede entrar
# a es la edad, s es el estado (1 activo, 0 inactivo)
def check(a, s):
    # Si es mayor de 18 y está activo
    if a >= 18 and s == 1:
        return True
    return False
```

SOLUCIÓN MODELO

```
def is_user_eligible_for_entry(age, status):
    is_adult = age >= 18
```

```
    is_active = status == 1

    return is_adult and is_active
```

(Nota: Cero comentarios necesarios. El código se lee como inglés).

EJERCICIO 3: El Docstring Perfecto

ENUNCIADO

Escribe un Docstring para esta función usando el formato `Args` y `Returns`. Inventa los detalles lógicos.

```
def calculate_insurance_premium(age, risk_factor):
    ...
```

SOLUCIÓN MODELO

```
def calculate_insurance_premium(age: int, risk_factor: float) -> float:
    """
    Calcula la prima mensual del seguro basada en perfil de riesgo.

    La fórmula penaliza exponencialmente el factor de riesgo para
    desincentivar perfiles altos (Política de Negocio 2024).

    Args:
        age: Edad del asegurado (debe ser 18-99).
        risk_factor: Índice de 0.0 a 1.0.
```



```
    Returns:
        Monto mensual en USD. Retorna 0 si es rechazado.
    """
```

AUTOEVALUACIÓN

- [ ] ¿He borrado comentarios obvios hoy?
- [ ] ¿Mis variables explican el “qué” para que mis comentarios solo expliquen el “por qué”?

EVALUACIÓN: COMENTARIOS Y DOCSTRINGS

FICHA TÉCNICA

- **Tema**: 2.1.1 El Por Qué sobre el Qué
- **Nivel**: Básico

CUESTIONARIO

Pregunta 1

¿Cuál es la función principal de un “Comentario Inline” Pragmático?

- [ ] **a)** Explicar la sintaxis del lenguaje para programadores junior.
- [ ] **b)** Traducir el código a lenguaje natural línea por línea.
- [ ] **c)** Explicar el “Por Qué” (Intención, Negocio) que no es obvio en el código.

Pregunta 2

Según la Regla de Kernighan, ¿qué debes hacer con el código malo?

- [ ] **a)** Comentarlo extensamente para advertir a otros.
- [ ] **b)** Reescribirlo para que sea claro sin comentarios.
- [ ] **c)** Borrarlo y empezar de cero.

Identifica el comentario **REDUNDANTE**:

- [ ] **a)** `// TODO: Refactorizar esto cuando salga v2.0`
- [ ] **b)** `// Usamos sleep(1) por race condition en API externa`
- [ ] **c)** `total = precio * 1.16 // Multiplica precio por 1.16`

Pregunta 4

¿Cuál es la diferencia entre un Docstring y un Comentario Inline?

- [ ] **a)** El Docstring es para el Usuario (API pública); el Comentario es para el Mantenedor (Implementación).
- [ ] **b)** El Docstring es opcional; el Comentario es obligatorio.

- [ ] **c)** No hay diferencia.

SOLUCIONARIO

1. **c)**. El código explica el Qué; el comentario explica el Por Qué.
2. **b)**. "Don't comment bad code — rewrite it."
3. **c)**. El código ya explica matemáticamente la operación; mejor sería renombrar la constante 1.16 a `TAX_RATE`.
4. **a)**. Docstring define el contrato de interfaz; Comentario explica detalles internos.

TEMA 2.2.1: ESTRUCTURA DE UN README GANADOR

Tiempo estimado: 30 minutos **Nivel**: Intermedio **Prerrequisitos**: Tema 1.2.1 (Usuario vs Mantenedor)

¿Por qué importa este concepto?

El README es la **Landing Page** de tu código. Es lo primero que ve un desarrollador en GitHub. Es tu única oportunidad para causar una buena impresión. Un mal README grita "Proyecto abandonado" o "Difícil de usar". Un buen README convierte visitantes en usuarios (o colaboradores).

1. ¿Qué hace esto?
2. ¿Cómo lo hago correr en mi máquina?

Conexión con conocimientos previos

Aplicamos la **Pirámide Invertida** (Tema 1.1.2) y el **Mapeo de Audiencia** (Tema 1.2.1).

- *Pirámide*: Pon el propósito y el "Quickstart" arriba. Mueve la configuración avanzada abajo.
- *Audiencia*: La mayoría de los lectores del README son **Usuarios** que quieren probarlo, no **Mantenedores** que quieren compilarlo desde cero.

Comprensión intuitiva

Imagina que entras a una tienda de Apple (Landing Page).

- **Lo que ves**: Mesas con productos listos para usar (El “Qué” y el “Cómo”).
- **Lo que no ves**: Los planos eléctricos de la tienda o el reglamento de los empleados (El “Internals”).

Tu README debe sentirse como la tienda, no como el almacén.

Definición formal

Un README efectivo debe contener, en orden de prioridad:

1. **Nombre y One-Liner**: Qué es.
2. **Badges**: Estado del build, versión, licencia (Prueba social).
3. **Descripción / Problema**: Qué soluciona.
4. **Quick Start**: El camino más corto para ver el “Hello World”.
5. **Features**: Lista de capacidades clave.
6. **Configuración**: Opciones avanzadas.

Implementación práctica

Anatomía de un README Perfecto


````
# 🚀 RocketLib (El Título)

> La librería más rápida para enviar cohetes a Marte en Python. (El One-Liner)

[![Build Status](file://E:/repositories/teach-laoz-courses-generator/cursos/curso_comunic

## ¿Por qué usar RocketLib? (El Gancho)
Las librerías actuales son lentas. RocketLib usa Rust por debajo para lograr...

## Instalación (La Acción)
```bash
pip install rocket-lib
````
````

Quick Start (El “Hello World”)

```
import rocket

rocket.launch(target="Mars")
# > 🚀 Despegando en 3, 2, 1...
```

Configuración Avanzada (La Cola de la Pirámide)

```
…


### El Anti-Patrón: "El Muro de Texto"

*   Empezar con la historia del proyecto. "Este proyecto nació en 2019 cuando..." (A nadi
*   Empezar con los requisitos de compilación. "Instala GCC v9, Make v4..." (Aburrido. So

### El Patrón: "Zero to Hero"

Tu sección de instalación debe ser copiables y pegables.
*   *Mal*: "Descarga el código y compílalo."
*   *Bien*:
    ```bash
    git clone ...
    cd ...
    ./install.sh
```



```
---

## Resumen del concepto

**En una frase**: Trata a tu README como un producto de marketing. Vende la solución y fa

**La Regla de los 10 Segundos**: Si no puedo ejecutar tu código en 10 segundos (o entende

**Siguiente paso**: Generación de Ejercicios para auditar READMEs reales.
```

BANCO DE EJERCICIOS: MÓDULO 2 - ESTRUCTURA DE README
```

METADATA

- **Módulo**: Documentación en el Código
- **Objetivos evaluados**:
    1. Auditar READMEs usando la regla de los 10 segundos.
    2. Redactar "Hooks" (Ganchos) efectivos.
    3. Diferenciar entre Quickstart y Configuración Avanzada.
- **Tiempo total estimado**: 20 minutos
- **Nivel**: Intermedio

EJERCICIO 1: Auditoría de 10 Segundos

ENUNCIADO

Abre el último README que escribiste (o un repositorio open source al azar). Cronometra 10 segundos. Intenta responder estas 3 preguntas:

1. ¿Qué problema resuelve este proyecto?
2. ¿Cuál es el comando para instalarlo?
3. ¿Cuál es el estado del build (funciona o no)?

Diagnóstico:

- Si fallaste la #1: Te falta un "One-Liner" claro al inicio.

- Si fallaste la #2: Tu sección de instalación "entierra el lede" o no tienes bloques de código copiables.
- Si fallaste la #3: Te faltan Badges de estado.

EJERCICIO 2: El Gancho (The Hook)

ENUNCIADO

Transforma la siguiente descripción genérica en un "Gancho" de producto atractivo.

Original (Aburrido):

> "Esta es una librería escrita en Python que permite a los usuarios manipular archivos CSV de manera rápida utilizando multiprocesamiento. Fue creada en 2023."

Tu Misión:

- Escribe un Título.
- Escribe un One-Liner (Slogan).
- Escribe un bloque de “Por qué usar esto”.

SOLUCIÓN MODELO

```
# ⚡ TurboCSV

> Manipulación de CSVs 10x más rápida usando todos tus núcleos de CPU.

## ¿Por qué TurboCSV?
Las librerías estándar como `pandas` son lentas cargando terabytes de datos.
TurboCSV usa multiprocesamiento nativo para parsear archivos gigantes en segundos, no hor
```

EJERCICIO 3: Ordenando la Casa

ENUNCIADO

Ordena las siguientes secciones para un README ideal (de arriba a abajo):

A. Créditos y Agradecimientos a los autores. B. Comando `npm install fast-logger`. C. Explicación detallada de la arquitectura interna de la clase Logger. D. Título y Descripción corta. E. Ejemplo de uso: `logger.log("Hola")`.

SOLUCIÓN

1. **D** (Título/Descripción): Lo primero que ves.
2. **B** (Instalación): La acción inmediata.
3. **E** (Ejemplo): Verlo funcionar (“Quickstart”).
4. **C** (Arquitectura): Para mantenedores (Caja Blanca), va después.
5. **A** (Créditos): Al final (La cola de la pirámide).

AUTOEVALUACIÓN

- [ ] ¿Mi README tiene bloques de código (```` ```bash ````) listos para copiar?
- [ ] ¿El primer párrafo explica el valor, no la historia?

EVALUACIÓN: ESTRUCTURA DE README

FICHA TÉCNICA

- **Tema**: 2.2.1 README como Producto
- **Nivel**: Intermedio

CUESTIONARIO

Pregunta 1

¿Cuál es la función principal de un README según el curso?

- [ ] **a)** Documentar legalmente la propiedad intelectual.
- [ ] **b)** Servir como "Landing Page" para convencer al desarrollador de usar el código.
- [ ] **c)** Listar todos los archivos del repositorio.

Pregunta 2

Según la Regla de los 10 Segundos, ¿qué debe encontrar el usuario inmediatamente?

- [ ] **a)** La lista de autores y agradecimientos.
- [ ] **b)** Qué hace el proyecto y cómo instalarlo/correrlo.

Pregunta 3

¿Cuál es el mejor formato para la sección de instalación?

- [ ] **a)** Una explicación detallada en prosa sobre las dependencias.
- [ ] **b)** Un enlace a la documentación oficial del lenguaje.
- [ ] **c)** Un bloque de código (`bash`) listo para copiar y pegar.

Pregunta 4

¿Dónde deberías colocar los "Badges" (Build passing, License)?

- [ ] **a)** Al final, como pie de página.
- [ ] **b)** En `CONTRIBUTING.md`.
- [ ] **c)** Inmediatamente después del título, para dar "Prueba Social".

SOLUCIONARIO

1. **b)**. El README es marketing técnico.
2. **b)**. Problema + Solución (Quickstart).
3. **c)**. Reduce la fricción cognitiva al mínimo.
4. **c)**. Generan confianza inmediata sobre la salud del proyecto.

MODULO 3

TEMA 3.1.1: OPENAPI/SWAGGER (CONTRACT-FIRST)

Tiempo estimado: 35 minutos **Nivel**: Intermedio **Prerrequisitos**: Módulo 2 (Documentación en Código)

¿Por qué importa este concepto?

Imagina construir un puente empezando desde los dos extremos sin planos, esperando encontrarse en el medio. Eso es desarrollar APIs **Code-First** (primero el código, luego la doc). Resultado: El Frontend espera un array y el Backend manda un objeto. El puente se cae.

Contract-First significa dibujar el plano (OpenAPI Spec) antes de poner un solo ladrillo (código).

Conexión con conocimientos previos

- **Tema 2.2.1 (README)**: Un README explica *cómo instalar*. Una Spec de OpenAPI explica *cómo interactuar*.
- **Tema 1.2.1 (Audiencia)**: La audiencia de tu API es una máquina (que necesita precisión estricta) y un humano (que necesita ejemplos claros).

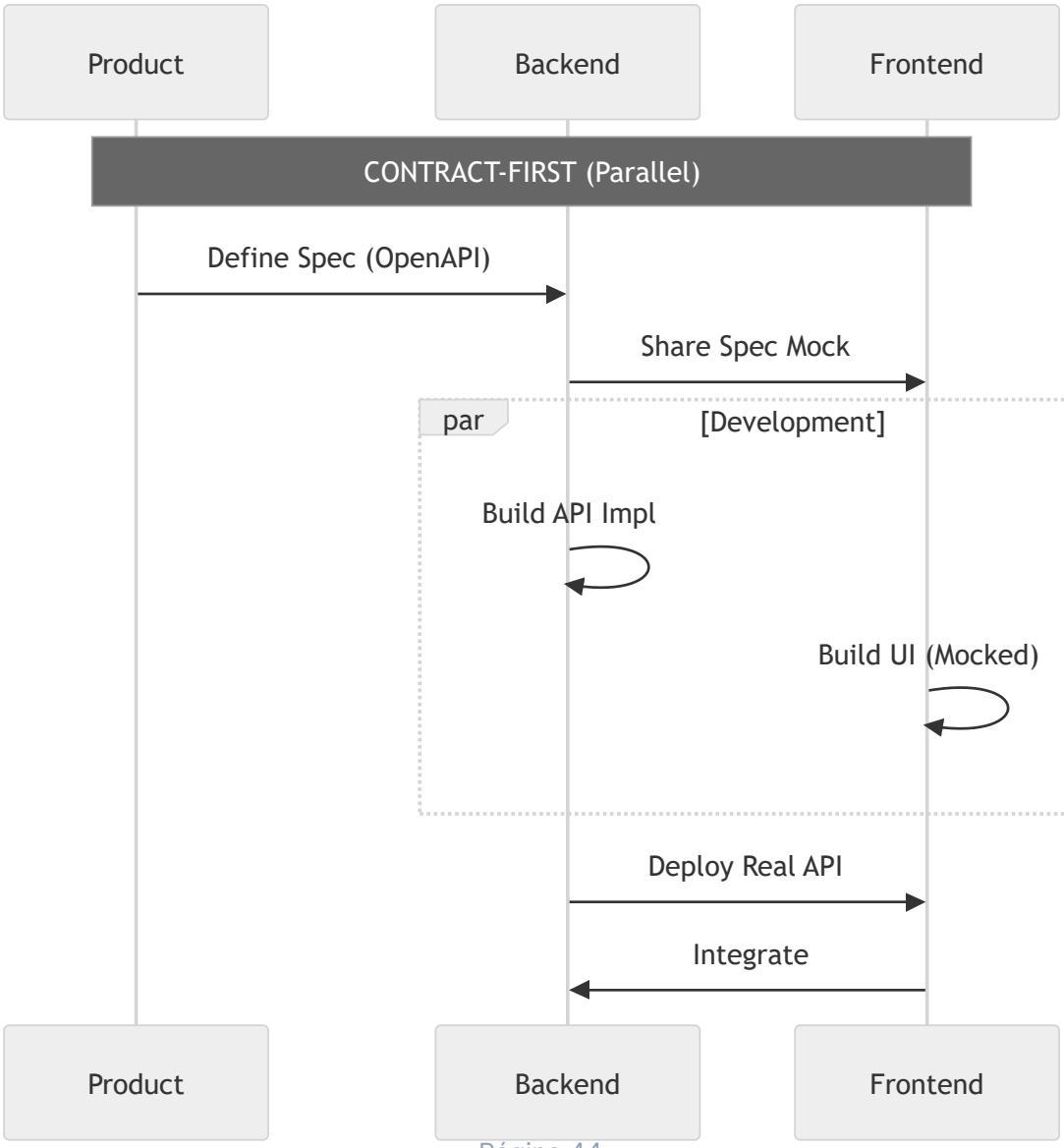
Comprensión intuitiva

OpenAPI es el **Contrato de Alquiler** entre el Backend y el Frontend. Firmas el contrato *antes* de mudarte. Si el contrato dice "El alquiler son 500 euros", el casero no puede pedirte 600 después. Si la API Spec dice "devuelvo un string", el Backend no puede devolver un null.

Definición formal

OpenAPI (antes Swagger) es un estándar para describir APIs REST en formato YAML o JSON. Es agnóstico del lenguaje (funciona para Node, Python, Java, etc.).

Flujo de Trabajo




```
'200':
          description: Éxito
          content:
            application/json:
              schema:
                type: object
                properties:
                  temp:
                    type: integer
                    example: 24
```

Herramientas

- **Swagger Editor**: Para escribir y previsualizar.
- **Stoplight**: Interfaz visual para no lidiar con YAML puro.
- **Postman**: Puede importar OpenAPI para generar tests.

Errores frecuentes

Error 1: Generar la documentación desde el código (Code-First)

Usar anotaciones en Java/Python para generar el Swagger al final. *Problema*: La documentación siempre llega tarde y refleja "lo que el código hace" (bugs incluidos), no "lo que debería hacer".

Error 2: "Enums" no documentados

Un campo `status` que devuelve `1`, `2` o `3`. Sin documentación, el consumidor tiene que adivinar qué significa cada número mágicos.

Resumen del concepto

En una frase: Escribe la documentación *antes* de escribir el código. Ahorrarás semanas de refactorización.

Regla de Oro: Si cambias la API, actualizas el contrato (Spec) primero, y el código después.

Siguiente paso: Ejercicios para leer y escribir YAML de OpenAPI.

BANCO DE EJERCICIOS: MÓDULO 3 - OPENAPI Y CONTRACT-FIRST

METADATA

- **Módulo**: Documentación de APIs
- **Objetivos evaluados**:
    1. Leer y modificar especificaciones OpenAPI (YAML).
    2. Identificar riesgos del enfoque "Code-First".
    3. Resolver conflictos de integración mediante contratos.
- **Tiempo total estimado**: 25 minutos
- **Nivel**: Avanzado

EJERCICIO 1: El Inspector de Contratos

ENUNCIADO

Analiza el siguiente fragmento YAML y encuentra los **3 Errores Críticos** que romperían la integración con el Frontend.

```
paths:
  /users:
    get:
      summary: Obtener usuarios
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: array  # Error: No define qué hay dentro del array
        '404':
          description: No encontrado
    post:
      summary: Crear usuario
```



```
- name: id   # Error: El ID debe ser generado por el server, no enviado en URL
          in: path
          required: true
          schema:
            type: integer
```

SOLUCIÓN

1. **Array sin items**: `type: array` necesita `items: { $ref: ... }`. El Front no sabe qué datos esperar.
2. **ID en Path para POST**: Un `POST` a `/users` no debería llevar ID (`/users/{id}`). El ID se crea al guardar.
3. **Falta RequestBody**: El `POST` no define qué datos enviar (Nombre, Email) en el cuerpo.

EJERCICIO 2: Code-First vs Contract-First

ENUNCIADO

Eres el Tech Lead. Tu equipo Backend sugiere: “Hagamos el código en Python primero, y luego usamos una librería para auto-generar el Swagger”. Escribe 2 argumentos técnicos para **rechazar** esa propuesta y forzar Contract-First.

SOLUCIÓN MODELO

1. **Bloqueo del Frontend**: “Si esperamos a terminar el código Python para tener el Swagger, el equipo de Frontend estará bloqueado 2 semanas esperando saber qué campos enviaremos. Con Contract-First, pueden usar Mocks hoy mismo.”
2. **Leaking Internals**: “Los generadores automáticos tienden a exponer nuestros modelos de Base de Datos (ej. `created_at_db_index`) en la API pública, acoplándonos innecesariamente.”

EJERCICIO 3: Diseñando la Hamburguesa

ENUNCIADO

Escribe el `schema` OpenAPI para un objeto `Product` en una tienda e-commerce. Requisitos:

- `id`: UUID, solo lectura.
- `name`: String, obligatorio.
- `price`: Número, obligatorio, mayor a 0.

- `tags`: Array de strings (opcional).

SOLUCIÓN MODELO

```
components:
  schemas:
    Product:
      type: object
      required:
        - name
        - price
      properties:
        id:
          type: string
          format: uuid
          readOnly: true
        name:
          type: string
        price:
```

```
        type: number
        minimum: 0.01
      tags:
        type: array
        items:
          type: string
```

AUTOEVALUACIÓN

- [ ] ¿Entiendo que el YAML es la ley, y el código es solo la implementación?
- [ ] ¿Puedo leer un archivo swagger.yaml sin usar la UI gráfica?

EVALUACIÓN: OPENAPI Y CONTRACT-FIRST

FICHA TÉCNICA

- **Tema**: 3.1.1 Diseño de Contratos de API
- **Nivel**: Avanzado

CUESTIONARIO

Pregunta 1

¿Qué es el enfoque **Contract-First**?

- [ ] **a)** Escribir el código primero y luego generar la documentación automáticamente.
- [ ] **b)** Definir y acordar la especificación (YAML/JSON) antes de escribir cualquier línea de código.
- [ ] **c)** Contratar consultores externos para diseñar la API.

Pregunta 2: Diagnóstico

Si tu API devuelve un campo llamado `mongoose_internal_id_v2`, ¿qué error de diseño estás cometiendo?

- [ ] **a)** Leaking Internals (Fuga de implementación).
- [ ] **b)** Over-fetching.
- [ ] **c)** Falta de autenticación.

Pregunta 3

¿Cuál es la función principal de **Swagger UI**?

- [ ] **a)** Compilar el código Backend.
- [ ] **b)** Generar bases de datos desde cero.
- [ ] **c)** Renderizar la especificación OpenAPI de forma visual e interactiva (“Try it out”).

Pregunta 4: YAML

En una especificación OpenAPI, ¿qué define la sección `components/schemas`?

- [ ] **a)** Las credenciales de acceso a la base de datos.
- [ ] **b)** Los modelos de datos reutilizables (ej. User, Product, Error).
- [ ] **c)** La lista de desarrolladores backend.

Pregunta 5

¿Qué ventaja tiene el Frontend al usar Contract-First?

- [ ] **a)** Puede empezar a programar inmediatamente usando servidores Mock basados en el contrato.
- [ ] **b)** No tiene que validar los datos.

- [ ] **c)** Puede escribir el código del Backend ellos mismos.

SOLUCIONARIO

1. **b)**. Es un acuerdo previo que desacopla el desarrollo.
2. **a)**. Estás exponiendo detalles de tu ORM (Mongoose) que deberían ser privados.
3. **c)**. Es la herramienta de visualización estándar.
4. **b)**. Permite definir la estructura de los objetos (tipos, requeridos) una sola vez.
5. **a)**. Permite paralelismo real entre equipos.

TEMA 3.1.2: AUTENTICACIÓN Y ERRORES

Tiempo estimado: 30 minutos **Nivel**: Intermedio **Prerrequisitos**: Tema 3.1.1 (OpenAPI)

¿Por qué importa este concepto?

Una API no se juzga por cómo funciona cuando todo va bien, sino por cómo se comporta cuando algo sale mal. Los errores son la interfaz de usuario del desarrollador. Un error críptico (`500 Internal Server Error`) hace perder horas de debugging. Un error claro (`400 Bad Request: Missing field 'email'`) se soluciona en segundos.

Igualmente, la autenticación es la puerta de entrada. Si es confusa, nadie entrará.

Conexión con conocimientos previos

En el Tema 2.1.1 (Comentarios), aprendimos a explicar el "Por Qué". Los errores de API deben hacer lo mismo: explicar *por qué* falló el request y *qué* hacer para arreglarlo.

Comprensión intuitiva

Los Códigos HTTP son Semáforos

Imagina conducir por una ciudad sin semáforos. Caos total. Los códigos HTTP son señales universales:

- **2xx (Verde)**: Todo bien. Pasa.

El Payload de Error es el Tablero

Si el coche se detiene, no quieres solo una luz roja genérica. Quieres un indicador específico: "Falta aceite" o "Puerta abierta". Devolver `200 OK` con un cuerpo `{ "status": "error" }` es como poner una pegatina de "Todo está bien" sobre el indicador de "Motor incendiado". **Es mentir**.

Definición formal

1. La Santísima Trinidad de los Errores

Nunca inventes códigos. Usa los estándares:

- **400 Bad Request**: Error de sintaxis o validación (falta campo).
- **401 Unauthorized**: "¿Quién eres?" (Falta token).
- **403 Forbidden**: "Sé quién eres, pero no puedes pasar" (Falta permiso).
- **404 Not Found**: Lo que buscas no existe.
- **429 Too Many Requests**: Calma, estás haciendo spam.
- **500 Internal Server Error**: Bug en el código (NullPointer, DB caída).

2. Estructura de Error (Standard Problem Details - RFC 7807)

No devuelvas solo texto. Devuelve JSON estructurado.

```
{
  "type": "about:blank",
  "title": "Saldo insuficiente",
  "status": 403,
  "detail": "Tu saldo actual es 10.00, pero la transacción requiere 25.00.",
  "instance": "/transactions/12345"
}
```

Implementación práctica

El Anti-Patrón: “200 OK con Error” (El Mentiroso)

```
HTTP/1.1 200 OK
Content-Type: application/json
```



```
{
  "success": false,
  "error": "Card declined"
}
```

- *Por qué es malo*: Las herramientas de monitoreo (Datadog, New Relic) ven un “200” y piensan que el sistema está saludable. El cliente tiene que parsear el cuerpo para saber si funcionó.

El Patrón: Autenticación Clara

Define explícitamente cómo autenticarse en tu OpenAPI:

1. **API Key**: Simple. `Authorization: ApiKey <secret>`. Para s2s (server-to-server).
2. **Bearer Token (JWT)**: Estándar para usuarios. `Authorization: Bearer <token>`.

```
components:
  securitySchemes:
    BearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
```

```
security:
  - BearerAuth: []
```

Errores frecuentes

Error 1: Leakear Stack Traces

Nunca devuelvas el stack trace de Java/Python en producción.

- *Mal*: `ValueError: at app.py line 45...`
- *Riesgo*: Expones rutas de archivos y versiones de librerías a atacantes.

Error 2: Mensajes Genéricos

- *Mal*: `400 Bad Request`.
- *Bien*: `400 Bad Request: Field 'age' must be an integer > 18`.

Resumen del concepto

En una frase: Usa los códigos HTTP como se diseñaron (no mientas con 200) y da mensajes de error que ayuden al desarrollador a arreglar el problema sin llamarte.

Regla de Oro: Un buen error enseña al usuario cómo usar la API.

Siguiente paso: Generación de Ejercicios para diagnosticar APIs mentirosas.

BANCO DE EJERCICIOS: MÓDULO 3 - AUTENTICACIÓN Y ERRORES

METADATA

- **Módulo**: Documentación de APIs
- **Objetivos evaluados**:
    1. Seleccionar el código de estado HTTP correcto.
    2. Escribir respuestas de error estructuradas (RFC 7807).
    3. Diferenciar errores de cliente (4xx) vs servidor (5xx).
- **Tiempo total estimado**: 20 minutos
- **Nivel**: Intermedio

EJERCICIO 1: El Semáforo HTTP

ENUNCIADO

Asocia cada escenario con el Código HTTP más apropiado (No uses 200 ni 500 genérico).

1. El usuario intenta borrar un post que no es suyo.
2. El usuario envía un JSON mal formado (falta una coma).
3. La base de datos se quedó sin conexiones.
4. El token de acceso ha expirado.
5. El recurso solicitado `/users/999` no existe.

SOLUCIÓN

1. **403 Forbidden**. (Sabe quién eres, pero no tienes permiso).
2. **400 Bad Request**. (Error de sintaxis).
3. **503 Service Unavailable**. (Problema temporal del servidor).
4. **401 Unauthorized**. (Vuelve a loguearte).

5. **404 Not Found**.

EJERCICIO 2: Diseñador de Payload

ENUNCIADO

Escribe el JSON de respuesta para el siguiente error:

- **Escenario**: Usuario intenta registrarse con el email `bob@gmail`, pero faltan el `.com`.
- **Requisito**: Usa el estándar RFC 7807 (Problem Details).

SOLUCIÓN MODELO

```
{
  "type": "https://api.myapp.com/errors/invalid-email",
  "title": "Invalid Input",
  "status": 400,
  "detail": "El campo 'email' no es válido. Falta el dominio.",
  "instance": "/register",
  "invalidParams": [
    {
      "name": "email",
```

```
      "reason": "Invalid format"
    }
  ]
}
```

EJERCICIO 3: El Mentiroso

ENUNCIADO

Refactoriza esta respuesta de API (Anti-patrón) a una respuesta RESTful correcta.

Respuesta Original (Mala):

```
HTTP/1.1 200 OK
{
  "status": "error",
  "msg": "No se encontró el producto"
}
```


Tu Refactorización: *(Escribe el Header y el Body)*

SOLUCIÓN MODELO

```
HTTP/1.1 404 Not Found
Content-Type: application/problem+json

{
  "title": "Product Not Found",
  "status": 404,
  "detail": "El producto con ID solicitado no existe en el inventario."
}
```

AUTOEVALUACIÓN

- [ ] ¿He dejado de usar `200 OK` para errores?
- [ ] ¿Mis errores explican CÓMO arreglar el problema?

EVALUACIÓN: AUTENTICACIÓN Y ERRORES

FICHA TÉCNICA

- **Tema**: 3.1.2 Protocolo HTTP y Errores
- **Nivel**: Intermedio

CUESTIONARIO

Pregunta 1

¿Qué significa un código de estado de la familia **4xx**?

- [ ] **a)** El servidor falló internamente.
- [ ] **b)** La petición del cliente tiene un error (Client Error).
- [ ] **c)** La petición fue procesada exitosamente.

Pregunta 2: Escenario

Un usuario intenta acceder a `/admin/panel` pero no tiene privilegios de administrador. Está logueado correctamente. ¿Qué código debes devolver?

- [ ] **a)** 401 Unauthorized.
- [ ] **b)** 403 Forbidden.
- [ ] **c)** 404 Not Found (para ocultarlo).

Pregunta 3: Anti-patrón

¿Por qué es malo devolver `200 OK` con un cuerpo JSON `{ "error": "failed" }`?

- [ ] **a)** Porque consume más ancho de banda.
- [ ] **b)** Porque las herramientas de monitoreo creerán que el sistema está saludable cuando no lo está.
- [ ] **c)** No es malo, es una práctica común en GraphQL.

Pregunta 4

Según el estándar RFC 7807 (Problem Details), ¿qué campos debería tener un error JSON?

- [ ] **a)** Solo un string con el mensaje.
- [ ] **b)** `type`, `title`, `status`, `detail`, `instance`.
- [ ] **c)** El stack trace completo del servidor.

Pregunta 5

¿Cuál es la forma estándar de enviar un token JWT en una petición HTTP?

- [ ] **a)** En la URL: `?token=...`
- [ ] **b)** En una cookie no segura.
- [ ] **c)** En el Header: `Authorization: Bearer <token>`

SOLUCIONARIO

1. **b)**. Indica que el cliente debe corregir algo antes de reintentar.
2. **b)**. 401 es "¿Quién eres?"; 403 es "Sé quién eres, pero no puedes pasar".
3. **b)**. Rompe la semántica del protocolo HTTP y dificulta la observabilidad.
4. **b)**. Provee una estructura estándar para que las máquinas entiendan el error.
5. **c)**. Es el estándar de industria para autenticación stateless.

MODULO 4

TEMA 4.1.1: ANATOMÍA DE UN DESIGN DOC

Tiempo estimado: 30 minutos **Nivel**: Avanzado **Prerrequisitos**: Módulo 3 (APIs)

¿Por qué importa este concepto?

Escribir código es caro. Reescribir código es 10 veces más caro. Un **Design Doc** (Documento de Diseño) es la herramienta más barata para detectar errores arquitectónicos *antes* de escribir una sola línea de código. Es "pensar en papel".

Conexión con conocimientos previos

- **Tema 1.1.2 (Pirámide Invertida)**: Un Design Doc debe tener un Resumen Ejecutivo al principio.

- **Tema 3.1.1 (Contract-First)**: El Design Doc es el contrato, pero de la arquitectura interna, no solo de la API pública.

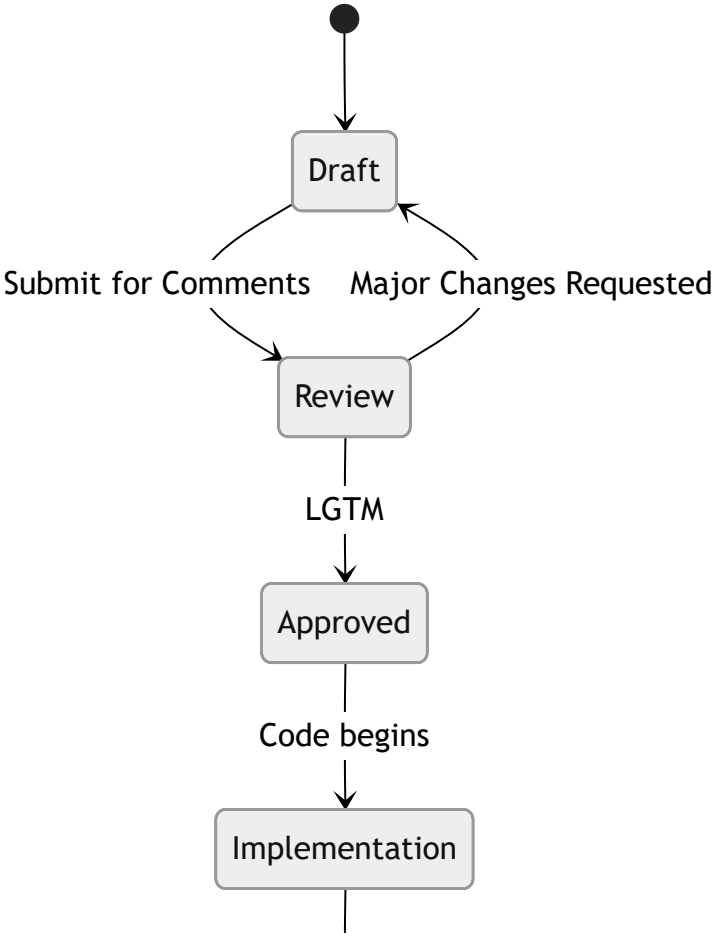
Comprensión intuitiva

Imagina que vas a construir un rascacielos. ¿Empiezas a poner ladrillos el día 1? No. Haces un plano (Blueprint). Se lo enseñas a los ingenieros estructurales, a los bomberos, a los electricistas. Ellos te dicen: *"Oye, pusiste los baños lejos de las tuberías de agua"*. Corriges el plano con un borrador. Costo: 0 euros. Si hubieras construido la pared, el costo sería millonario.

Definición formal

Un **Design Doc** (o RFC - Request for Comments) es un documento técnico que describe la solución propuesta a un problema complejo.

Ciclo de vida



En una frase: El código gana argumentos; los Design Docs evitan que los argumentos ocurran.

Regla de Oro: Busca el feedback negativo. Si todos dicen "LGTM" (Looks Good To Me) sin leer, tu Design Doc falló. Quieres que alguien encuentre el agujero de seguridad ahora, no los hackers después.

Siguiente paso: Ejercicios para analizar alternativas y trade-offs.

BANCO DE EJERCICIOS: MÓDULO 4 - DESIGN DOCS Y RFCs

METADATA

- **Módulo**: Documentación de Arquitectura
- **Objetivos evaluados**:
    1. Redactar el contexto de un problema sin saltar a la solución.
    2. Evaluar Trade-offs en la sección de "Alternativas Consideradas".
    3. Distinguir entre "Diseño de Alto Nivel" y "Detalles de Implementación".
- **Tiempo total estimado**: 30 minutos
- **Nivel**: Avanzado

EJERCICIO 1: El One-Liner Ejecutivo

ENUNCIADO

Eres el Tech Lead. Tienes que convencer al CTO de reescribir el módulo de pagos de Python a Go. Escribe el **Resumen (TL;DR)** de tu Design Doc.

- **Restricción**: Máximo 2 oraciones.
- **Debe incluir**: El cambio propuesto y el beneficio de negocio (no solo técnico).

SOLUCIÓN MODELO

> "Proponemos migrar el servicio de checkout a Go para reducir la latencia de P99 de 500ms a 50ms durante el Black Friday. Esto evitará la pérdida estimada de $50k en ventas por timeouts observada el año pasado."

EJERCICIO 2: Matriz de Decisiones (Trade-offs)

ENUNCIADO

Estás eligiendo una base de datos para guardar logs de auditoría (write-heavy, read-rarely). Completa la sección "Alternativas Consideradas" para **PostgreSQL** vs **Elasticsearch**.

Criterio	PostgreSQL	Elasticsearch	Decisión
Escritura	(A)	(B)	
Búsqueda	SQL estándar, lenta en texto libre	Búsqueda full-text nativa y rápida	
Costo	Bajo (ya tenemos infraestructura)	©	

(Rellena A, B y C)

SOLUCIÓN

- **(A)**: ACID, más lenta para ingesta masiva sin sharding.
- **(B)**: Optimizada para append-only, ingesta masiva casi real-time.
- **©**: Alto (requiere cluster dedicado y mucha RAM).

EJERCICIO 3: El Detective de Design Docs

ENUNCIADO

Lee el siguiente fragmento de un Design Doc y marca qué partes **NO** deberían estar ahí (son detalles de implementación o ruido).

> "Para el sistema de notificaciones, vamos a crear una clase `NotificationManager` que hereda de `BaseManager` (línea 45 de `utils.py`). Usaremos un `for loop` para iterar los usuarios. La base de datos estará en la IP 192.168.1.50. El objetivo es enviar emails a usuarios inactivos."

SOLUCIÓN

- ❌ `hereda de BaseManager`: Detalle de implementación irrelevante para la arquitectura.
- ❌ `línea 45 de utils.py`: Demasiado granular, cambiará mañana.
- ❌ `Usaremos un for loop`: Obvio.

- ❌ `IP 192.168.1.50`: Hardcoded configuration.
- ✅ `El objetivo es enviar emails a usuarios inactivos`: Esto es **Contexto**, y es lo único valioso del párrafo.

AUTOEVALUACIÓN

- [ ] ¿He mencionado los "Riesgos" de mi solución propuesta?
- [ ] ¿He explicado "Por qué NO" elegí las otras alternativas?

EVALUACIÓN: DESIGN DOCS Y RFCs

FICHA TÉCNICA

- **Tema**: 4.1.1 Documentación de Arquitectura
- **Nivel**: Avanzado

CUESTIONARIO

Pregunta 1

¿Cuál es el objetivo principal de escribir un Design Doc **antes** de codificar?

- [ ] **a)** Cumplir con los requisitos de RRHH para promoción.
- [ ] **b)** Detectar fallos de arquitectura cuando el costo de corrección es bajo (en papel).
- [ ] **c)** Generar documentación automática para el usuario final.

Pregunta 2

En la sección **"Alternativas Consideradas"**, ¿qué debes documentar?

- [ ] **a)** Solo la opción que elegiste.
- [ ] **b)** Todas las opciones evaluadas y la razón técnica/negocio por la cual fueron descartadas.
- [ ] **c)** Un historial de chats de Slack.

Pregunta 3: Diagnóstico

Estás revisando un Design Doc y ves el siguiente texto: *"En la línea 54 del archivo user_controller.py, usaremos un bucle while..."*. ¿Qué feedback darías?

- [ ] **a)** "Excelente detalle".
- [ ] **b)** "Esto es un Detalle de Implementación. El Design Doc debe enfocarse en la Arquitectura (Alto Nivel), no en el código línea por línea."
- [ ] **c)** "Cambia el while por un for".

Pregunta 4

¿Qué significan las siglas **RFC** en este contexto?

- [ ] **a)** Request for Comments (Solicitud de Comentarios).
- [ ] **b)** Ready for Coding (Listo para Codificar).
- [ ] **c)** Real Fast Compiler.

Pregunta 5

¿Cuándo se debe escribir un Design Doc?

- [ ] **a)** Al finalizar el proyecto, para dejar registro.
- [ ] **b)** Durante la implementación, actualizándolo cada día.
- [ ] **c)** Antes de escribir código significativo (ej. tareas de >3 días), como herramienta de planificación.

SOLUCIONARIO

1. **b)**. “Shift left” en la detección de errores.
2. **b)**. El valor está en los trade-offs analizados.
3. **b)**. El código cambia; la arquitectura debe ser estable.
4. **a)**. Es un estándar adoptado del IETF para propuestas colaborativas.
5. **c)**. Es una herramienta de diseño, no de autopsia.

TEMA 4.2.1: ADRs (ARCHITECTURE DECISION RECORDS)

Tiempo estimado: 30 minutos **Nivel**: Avanzado **Prerrequisitos**: Tema 4.1.1 (Design Docs)

¿Por qué importa este concepto?

¿Alguna vez has mirado una base de datos y te has preguntado: *“¿Por qué diablos separaron el nombre en 4 campos distintos?”*? Y nadie sabe la respuesta. La persona que lo decidió se fue hace 3 años.

La arquitectura de software sufre de amnesia. Un **ADR** es la cura para esa amnesia. Es un registro inmutable (como un commit de git) que explica una decisión en el momento en que se tomó.

Conexión con conocimientos previos

- **Tema 4.1.1 (Design Docs)**: Un Design Doc es para *proponer* un cambio grande. Un ADR es para *registrar* una decisión concreta (haya habido design doc o no).
- **Git Commit**: Un ADR es a la arquitectura lo que un mensaje de commit es al código.

Comprensión intuitiva

Imagina la **Bitácora del Capitán** de un barco.

- *Día 1*: “Decidimos ir al norte para evitar la tormenta.”
- *Día 5*: “La ruta norte está bloqueada por hielo. Decidimos girar al oeste.”

Si lees la bitácora, entiendes el viaje. Si solo ves el mapa final, pensarás que el capitán estaba borracho por dar tantas vueltas. Los ADRs son la bitácora de tu proyecto.

Definición formal

Un **ADR** es un archivo de texto ligero (Markdown) guardado en el repositorio (usualmente en `/doc/adr`) que captura una decisión de arquitectura significativa.

Formato Estándar (Michael Nygard)

Un ADR tiene 4 secciones obligatorias:

1. **Título**: `001-usar-postgres-para-usuarios.md`
2. **Contexto**: Qué problema teníamos. "Necesitamos transacciones ACID para el saldo."
3. **Decisión**: Qué elegimos. "Usaremos PostgreSQL 14."

4. **Consecuencias**: Lo bueno y lo malo. "Ganamos consistencia, pero perdemos la facilidad de sharding de Mongo."

Estado del ADR

- **Proposed**: En discusión.
- **Accepted**: Aprobado y vigente.
- **Deprecated**: Fue reemplazado por un ADR posterior. (Nunca borres un ADR, solo márcalo como obsoleto).

Implementación práctica

Ejemplo de ADR Real

```
# ADR 009: Usar UUIDs para Claves Primarias

**Fecha**: 2025-10-15
**Estado**: Accepted

## Contexto
Actualmente usamos `AUTO_INCREMENT` (Integers) para los IDs de usuarios.
Esto revela el número de usuarios a la competencia (si alguien se registra y recibe el ID
Además, dificulta la fusión de bases de datos en el futuro (colisión de IDs).

## Decisión
Usaremos **UUID v4** para todas las nuevas tablas y migraremos la tabla `Users`.
```

```
## Consecuencias
### Positivas
*   Seguridad: Los IDs no son enumerables.
*   Descentralización: Podemos generar IDs en el cliente o en múltiples nodos sin coordin

### Negativas
*   Rendimiento: Los índices B-Tree son más lentos y grandes con UUIDs que con Integers.
*   Legibilidad: Es difícil dictar un UUID por teléfono para soporte (`a4b5...`).
```

Herramientas

No necesitas software complejo. Usa la CLI `adr-tools`:


```
adr init
adr new "Usar UUIDs"
```

Resumen del concepto

En una frase: Si tomas una decisión que afecta cómo trabajarán los demás, escríbela. Si no está escrita, no es una decisión, es solo un rumor.

Regla de Oro: Un ADR explica el "Por qué", no solo el "Qué". Incluso "No hacer nada" es una decisión que merece un ADR.

Siguiente paso: Generación de Ejercicios para redactar tus primeros ADRs.

BANCO DE EJERCICIOS: MÓDULO 4 - ADRs

METADATA

- **Módulo**: Documentación de Arquitectura
- **Objetivos evaluados**:
    1. Estructurar una decisión siguiendo el formato Nygard (Contexto, Decisión, Consecuencias).
    2. Identificar el impacto negativo (Trade-offs) de una decisión técnica.
    3. Diferenciar entre una "Preferencia Personal" y una "Decisión Arquitectónica".
- **Tiempo total estimado**: 25 minutos

- **Nivel**: Avanzado

EJERCICIO 1: El Arquitecto de Caching

ENUNCIADO

Tu equipo decidió implementar **Redis** para cachear las respuestas de la API de Productos. Redacta la sección de **"Consecuencias"** de ese ADR.

- **Requisito**: Debes listar al menos 1 consecuencia positiva y 2 consecuencias negativas (riesgos).

SOLUCIÓN MODELO

Consecuencias

- **Positivas**:
  - Reducción de carga en la base de datos principal en un 80%.
  - Tiempo de respuesta promedio baja de 200ms a 20ms.
- **Negativas**:
  - **Consistencia Eventual**: Los usuarios podrían ver precios viejos por hasta 5 minutos (TTL).
  - **Complejidad Operativa**: Necesitamos mantener un clúster de Redis altamente disponible; si cae, la DB principal podría colapsar por el tráfico repentino (Thundering Herd).

EJERCICIO 2: Arqueología de Software

ENUNCIADO

Llegas a un proyecto y ves que todo el código está escrito en **Java 8**, aunque estamos en 2025. Nadie sabe por qué. Escribe un ADR tipo "Retroactivo" para documentar este hecho y decidir qué hacer.

- **Título**: `ADR-005: Mantener vs Actualizar Java 8`
- **Decisión**: (Elige una: Actualizar a Java 17 o Mantener Java 8) y justifícala brevemente.

SOLUCIÓN MODELO (Opción: Mantener)

Contexto: El sistema usa librerías legacy `sun.misc.*` que fueron eliminadas en Java 9+. **Decisión**: Mantendremos Java 8 por los próximos 6 meses.

Justificación: El costo de reescribir las dependencias legacy supera el beneficio de las nuevas features de Java 17 en este momento crítico de negocio.

EJERCICIO 3: Detectando “Opiniones” disfrazadas de ADRs

ENUNCIADO

Analiza este borrador de ADR y di por qué está mal:

> **Título**: Usar Linter **Contexto**: El código está feo. **Decisión**: Usaremos el linter de AIRBNB porque me gusta más que el Standard, odio no poner puntos y comas. **Consecuencias**: El código se verá mejor.

CRÍTICA

1. **Contexto débil**: “El código está feo” es subjetivo. Debería hablar de consistencia o errores prevenibles.
2. **Decisión sesgada**: “Porque me gusta más” no es un argumento técnico válido. Debería comparar reglas objetivas o adopción en la comunidad.
3. **Falta de Consecuencias Negativas**: ¿Qué pasa con el código existente? ¿Habrá que arreglar 5000 archivos? ¿Romperá el build?

AUTOEVALUACIÓN

- [ ] ¿He listado al menos una consecuencia negativa para mi decisión?
- [ ] ¿Es mi ADR inmutable (se entiende sin contexto externo)?

EVALUACIÓN: ADRs

FICHA TÉCNICA

- **Tema**: 4.2.1 Architecture Decision Records
- **Nivel**: Avanzado

CUESTIONARIO

Pregunta 1

¿Cuál es la diferencia principal entre un **Design Doc** y un **ADR**?

- [ ] **a)** El Design Doc es para propuestas futuras (mutables); el ADR es un registro histórico de una decisión tomada (inmutable).
- [ ] **b)** Los ADRs son solo para bases de datos.
- [ ] **c)** El Design Doc se escribe en Word y el ADR en Excel.

Pregunta 2

Según el formato de **Michael Nygard**, ¿cuáles son las secciones obligatorias de un ADR?

- [ ] **b)** Contexto, Decisión y Consecuencias.
- [ ] **c)** Presupuesto, Plazos y Responsables.

Pregunta 3

Si una decisión arquitectónica cambia (ej. decidimos dejar de usar Redis y volver a Memcached), ¿qué haces con el ADR original de Redis?

- [ ] **a)** Lo borras del repositorio para no confundir.
- [ ] **b)** Lo editas y cambias el texto.
- [ ] **c)** Creas un nuevo ADR que "Depreca" (Supersedes) al anterior, manteniendo el historial intacto.

Pregunta 4

¿Qué debe incluirse en la sección de **Consecuencias**?

- [ ] **a)** Solo los beneficios.
- [ ] **b)** Solo los riesgos.
- [ ] **c)** Tanto los beneficios (Pros) como los riesgos/costos (Cons) aceptados.

Pregunta 5

¿Dónde deben vivir los archivos ADR?

- [ ] **a)** En una carpeta compartida de Google Drive.
- [ ] **b)** En el mismo repositorio que el código (ej. `/doc/adr`), bajo control de versiones.
- [ ] **c)** En la wiki de la empresa (Confluence/Notion) solamente.

SOLUCIONARIO

1. **a)**. El ADR congela el “por qué” en el tiempo.
2. **b)**. Es el estándar de industria para ligereza y claridad.
3. **c)**. La historia no se borra; se evoluciona.
4. **c)**. Toda decisión técnica tiene Trade-offs.
5. **b)**. “Docs as Code”: la documentación debe viajar con el software.

MODULO 5

TEMA 5.1.1: POSTMORTEMS BLAMELESS (CAUSA RAÍZ)

Tiempo estimado: 35 minutos **Nivel**: Intermedio **Prerrequisitos**: Tema 4.1.1 (Design Docs)

¿Por qué importa este concepto?

Cuando un sistema de producción cae, la reacción instintiva es buscar un culpable: *“¿Quién hizo el deploy?”*. Esta mentalidad garantiza que el error se repetirá. Si despides al ingeniero que borró la base de datos, has despedido a la única persona que aprendió la lección (y que costó millones entrenar).

Un **Postmortem Blameless** transforma un desastre costoso en un activo de conocimiento.

Conexión con conocimientos previos

- **Tema 3.1.2 (Errores HTTP)**: Un error `500` en producción es el detonante habitual de un Postmortem.

- **Tema 5.2.1 (Code Reviews)**: La cultura de “no culpar” es la misma que usaremos en los Code Reviews.

Comprensión intuitiva

Piensa en una **Investigación de Accidente Aéreo**. El objetivo no es meter al piloto en la cárcel. El objetivo es rediseñar la cabina para que ningún otro piloto pueda cometer ese mismo error, incluso si está cansado o estresado.

- *Culpable*: “Juan borró la tabla.”

Definición formal

Un **Postmortem** es un documento escrito *después* de que el incidente ha sido resuelto. No es un chat de Slack. Es un reporte formal que responde:

1. **¿Qué pasó?** (Timeline).
2. **¿Por qué pasó?** (Root Cause Analysis).
3. **¿Cómo evitamos que pase de nuevo?** (Action Items).

Técnica: Los 5 Porqués (Toyota)

No te detengas en la primera respuesta.

- **Problema**: El sitio se cayó.

1. *¿Por qué?*: La base de datos CPU llegó al 100%.
2. *¿Por qué?*: Una query sin índice se ejecutó millones de veces.
3. *¿Por qué?*: Se lanzó una nueva feature que usaba esa query.
4. *¿Por qué?*: El desarrollador no probó con datos reales.
5. *¿Por qué? (Causa Raíz)*: **No tenemos un entorno de Staging con volumen de datos similar a Producción.**

Solución: Crear un script de anonimización para Staging. (No “Regañar al desarrollador”).

Estructura del Documento

1. **Resumen Ejecutivo**: Impacto en el negocio (duración, usuarios afectados, dinero perdido).
2. **Cronología (Timeline)**: Segundo a segundo.
    - `10:00` - Se inicia deploy v2.
    - `10:05` - Alertas de latencia se disparan.
    - `10:15` - Rollback iniciado.
3. **Causa Raíz**: El resultado de los 5 Porqués.

4. **Action Items**: Tareas en Jira con dueño y fecha. “Arreglarlo” no es un action item. “Implementar Rate Limiting en endpoint X” sí lo es.

Errores frecuentes

Error 1: Lenguaje Acusatorio

- *Mal*: “El error fue causado porque Pedro olvidó el `WHERE`.”
- *Bien*: “El comando `DELETE` fue ejecutado sin cláusula `WHERE`, lo que el sistema permitió sin advertencia.”

Error 2: “Ser más cuidadosos” como Solución

“Action Item: Los devs deben tener más cuidado.” Esto NO sirve. Los humanos fallan. La solución debe ser técnica (Tooling, Automatización, Linter), no psicológica.

Resumen del concepto

En una frase: No desperdicies una buena crisis. Úsala para endurecer el sistema, no para castigar a las personas.

Regla de Oro: Si en tu postmortem aparece el nombre de una persona como “Causa”, has fallado. La causa siempre es el proceso o la herramienta.

Siguiente paso: Ejercicios para practicar la técnica de los 5 Porqués.

BANCO DE EJERCICIOS: MÓDULO 5 - POSTMORTEMS

METADATA

- **Módulo**: Comunicación en Crisis y Feedback
- **Objetivos evaluados**:
    1. Aplicar la técnica de los “5 Porqués”.

- **Nivel**: Intermedio

EJERCICIO 1: Los 5 Porqués

ENUNCIADO

Incidente: La facturación mensual falló para el 30% de los usuarios. **Causa Inmediata**: El proceso batch de facturación se quedó sin memoria (OOM).

Aplica 5 niveles de profundidad para encontrar la Causa Raíz. *(Usa tu imaginación técnica para rellenar los huecos)*.

1. **¿Por qué falló por memoria?** → *R: Porque cargó todos los usuarios en memoria a la vez.*
2. **¿Por qué cargó todos los usuarios?** → *R: ...*
3. **¿Por qué...?** → *R: ...*
4. **¿Por qué...?** → *R: ...*
5. **¿Por qué...? (Causa Raíz)** → *R: ...*

SOLUCIÓN MODELO

1. Porque cargó todos los usuarios en un array.
2. **¿Por qué?**: Porque el código usaba `User.all()` sin paginación.
3. **¿Por qué?**: Porque el dev asumió que había pocos usuarios (como en local).
4. **¿Por qué?**: Porque no probó con un dataset de tamaño producción.
5. **¿Por qué?**: Porque **Staging no tiene datos seed automatizados** que repliquen el volumen de Prod. (Causa Raíz).

EJERCICIO 2: El Detector de Culpa

ENUNCIADO

Convierte estas frases de “Culpables” a “Sistémicas”.

1. “María se olvidó de renovar el certificado SSL.”
2. “El intern borró la base de datos de producción por error.”
3. “Nadie revisó el PR con suficiente atención.”

SOLUCIÓN

1. “El sistema de monitoreo no alertó sobre la expiración inminente del certificado SSL con suficiente antelación.”
2. “La base de datos de producción era accesible con permisos de escritura desde la terminal de un usuario sin MFA, y no había protección contra borrado masivo.”
3. “El proceso de CI/CD permitió mergear código sin requerir la aprobación explícita de un CODEOWNER.”

EJERCICIO 3: Action Items Reales

ENUNCIADO

Para el incidente del Ejercicio 1 (OOM en facturación), escribe 2 Action Items.

- Uno debe ser **Mitigación Inmediata**.
- Uno debe ser **Prevención Definitiva**.

SOLUCIÓN MODELO

1. **Inmediata**: Refactorizar el script para usar paginación (batch size: 1000) y liberar memoria en cada iteración. (Dueño: Tech Lead. Fecha: Hoy).
2. **Preventiva**: Crear un script de “Data Seeding” en Staging que genere 1 millón de usuarios dummy para testear rendimiento antes de cada release. (Dueño: DevOps. Fecha: Próximo Sprint).

AUTOEVALUACIÓN

- [ ] ¿He evitado usar nombres de personas en mis respuestas?
- [ ] ¿Mis soluciones requieren que la gente “tenga más cuidado” (Mal) o cambian el sistema (Bien)?

EVALUACIÓN: POSTMORTEMS BLAMELESS

FICHA TÉCNICA

- **Tema**: 5.1.1 Análisis de Causa Raíz
- **Nivel**: Intermedio

CUESTIONARIO

Pregunta 1

¿Cuál es el propósito principal de un Postmortem?

- [ ] **a)** Asignar responsabilidad financiera al equipo causante.
- [ ] **b)** Aprender del incidente para prevenir que se repita sistémicamente.
- [ ] **c)** Pedir disculpas públicas a los clientes.

Pregunta 2: Los 5 Porqués

Al aplicar los "5 Porqués", ¿cuándo sabes que has llegado a la Causa Raíz?

- [ ] **a)** Cuando encuentras a la persona que cometió el error.
- [ ] **b)** Cuando llegas a un fallo de proceso, herramienta o cultura que puede ser arreglado permanentemente.
- [ ] **c)** Después de exactamente 5 preguntas, ni una más.

Pregunta 3: Diagnóstico

Analiza este Action Item: *"Los desarrolladores deben tener más cuidado al ejecutar DELETE en producción"*. ¿Es válido?

- [ ] **a)** Sí, es un buen recordatorio.
- [ ] **b)** No, porque depende de la voluntad humana y no cambia el sistema (es frágil).
- [ ] **c)** Sí, pero debería incluir multas.

Pregunta 4

¿Qué significa "Blameless" (Sin Culpa)?

- [ ] **a)** Que nadie es responsable de nada.

- [ ] **b)** Que se asume buena intención y se enfoca la investigación en el sistema, no en los individuos.
- [ ] **c)** Que los errores se ocultan bajo la alfombra.

Pregunta 5

¿Qué sección del Postmortem es crucial para la trazabilidad futura?

- [ ] **a)** La lista de asistentes a la reunión.

- [ ] **c)** La firma del CEO.

SOLUCIONARIO

1. **b)**. Transformar fallo en aprendizaje.
2. **b)**. La causa raíz siempre es sistémica, no individual.
3. **b)**. La esperanza no es una estrategia de ingeniería.
4. **b)**. Seguridad psicológica es requisito para la honestidad técnica.
5. **b)**. Permite entender la secuencia de eventos y la velocidad de respuesta.

TEMA 5.2.1: CODE REVIEWS EMPÁTICOS

Tiempo estimado: 30 minutos **Nivel**: Intermedio **Prerrequisitos**: Tema 5.1.1 (Blameless Culture)

¿Por qué importa este concepto?

El Code Review (PR) es el punto de mayor fricción social en el desarrollo. Es donde el "Mi código" se convierte en "Nuestro código". Una mala cultura de review crea silos: *"No reviso el código de Ana porque siempre se ofende"* o *"Álex es un pedante que me bloquea por espacios en blanco"*.

Un **Code Review Empático** mejora la calidad del código SIN destruir la moral del equipo.

Conexión con conocimientos previos

- **Tema 1.1.2 (Pirámide Invertida)**: Tus comentarios en el PR deben ser directos pero amables.
- **Tema 5.1.1 (Postmortems)**: El objetivo es mejorar el sistema, no juzgar al autor.

Comprensión intuitiva

Piensa en un **Editor de Libros**. El editor no odia al escritor. Quiere que el libro sea un best-seller. Si el editor dice: *"Este capítulo es aburrido, bórralo"*, el escritor se pone a la defensiva. Si dice: *"Creo que la trama ganaría velocidad si unimos estos dos capítulos"*, el escritor colabora.

El Code Review es **Edición Colaborativa**, no un examen que se aprueba o reprueba.

Definición formal

Un buen comentario de Code Review tiene 3 características (Google Standard):

1. **Accionable**: Dice claramente qué cambiar.
2. **Justificado**: Explica por qué (Performance, Estilo, Seguridad).
3. **Etiquetado**: Define la severidad.

Niveles de Severidad (Etiquetas)

Usa prefijos para eliminar la ansiedad:

- **[BLOCKER]**: "Esto rompe producción o introduce un bug de seguridad. No se puede mergear."
- **[NIT]** (Nitpick): "Detalle menor (espacios, nombres). Arréglalo si quieres, pero no bloquea el merge."
- **[OPTIONAL]**: "Sugerencia de refactoring para el futuro. No bloquea."
- **[QUESTION]**: "No entiendo esto, ¿me lo explicas? (No es una crítica, es duda)."

La Técnica "Nosotros" vs "Tú"

Jamás uses "Tú" para señalar un error. Ataca al código, no a la persona.

- *Violento*: "**Tú** olvidaste cerrar la conexión aquí."
- *Neutral*: "El código no cierra la conexión."
- *Empático*: "**Nos** falta cerrar la conexión aquí para evitar memory leaks."

Implementación práctica

El Sandwich de Feedback (Uso con precaución)

A veces ayuda empezar con algo positivo. *"¡Gran refactorización de la clase User! Solo tengo un par de dudas sobre el manejo de errores en este método..."*

Ejemplo de Comentario Constructivo

> **[BLOCKER]**: Veo que estamos iterando la lista `users` dentro del loop de `orders`.
> **Por qué**: Esto crea una complejidad O(n^2) que tumbará el servidor si tenemos

> más de 100 usuarios. **Sugerencia**: Podríamos crear un mapa de usuarios antes del loop para bajarlo a O(n). ¿Qué opinas?

Errores frecuentes

Error 1: "LGTM" Automático (Looks Good To Me)

Aprobar sin leer es peligroso. Si no entiendes el código, di: *"No tengo contexto suficiente para aprobar esto"*. Es más profesional.

Error 2: Comentarios Pasivo-Agresivos

- *"¿Por qué harías esto?"*
- *"Esto es obvio..."* Evita preguntas retóricas. Sé explícito.

Resumen del concepto

En una frase: Trata el código de tu compañero con el mismo respeto con el que tratarías el tuyo propio si tuvieras amnesia.

Regla de Oro: Si tienes que escribir más de 3 comentarios seguidos sobre el mismo tema, **deja de escribir y haz una videollamada**. El texto es malo para el debate.

Siguiente paso: Ejercicios para reescribir comentarios tóxicos.

BANCO DE EJERCICIOS: MÓDULO 5 - CODE REVIEWS EMPÁTICOS

METADATA

- **Módulo**: Comunicación en Crisis y Feedback
- **Objetivos evaluados**:
    1. Transformar comentarios agresivos en feedback constructivo.
    2. Aplicar etiquetas de severidad (NIT, BLOCKER, OPTIONAL).
    3. Practicar la técnica del “Nosotros” vs “Tú”.
- **Tiempo total estimado**: 20 minutos
- **Nivel**: Intermedio

EJERCICIO 1: El Traductor Diplomático

ENUNCIADO

Reescribe estos comentarios reales y tóxicos para que sean accionables y empáticos, usando la técnica “Nosotros”.

1. “¿Por qué diablos usaste un bucle anidado aquí? Es horrible.”
2. “Este nombre de variable no tiene sentido. Cámbialo.”
3. “Leíste la documentación antes de escribir esto?”

SOLUCIÓN MODELO

1. “Veo que usamos un bucle anidado. **Nos** preocupa que esto afecte el rendimiento con grandes volúmenes de datos. ¿Podríamos usar un HashMap para evitar la complejidad cuadrática?”
2. “Este nombre de variable (`x`) es un poco ambiguo. **Sugerencia**: ¿Qué tal si lo renombramos a `userSessionId` para que sea más claro para el próximo que lea el código?”
3. “Creo que esta implementación difiere de lo que dicen los docs sobre el manejo de fechas. ¿Podríamos revisar juntos esa parte para asegurarnos de que no estamos introduciendo un bug?”

EJERCICIO 2: Clasificación de Severidad

ENUNCIADO

Etiqueta los siguientes hallazgos en un PR con `[BLOCKER]`, `[NIT]` o `[QUESTION]`.

1. El código permite inyección SQL en el login.
2. Falta un espacio después del `if`.
3. El desarrollador usó una librería de 2018 que tiene vulnerabilidades conocidas.

4. No entiendo por qué se divide por 100 aquí.
5. Sería mejor (pero no urgente) mover esta función a otro archivo.

SOLUCIÓN

1. **[BLOCKER]**: Seguridad crítica.
2. **[NIT]**: Estilo, no afecta funcionalidad.

4. **[QUESTION]**: Necesita contexto antes de juzgar.
5. **[OPTIONAL]** o **[NIT]**: Sugerencia de refactoring.

EJERCICIO 3: El Estancamiento (Stalemate)

ENUNCIADO

Llevas 3 días discutiendo en los comentarios de un PR con un compañero sobre si usar `Tabs` o `Spaces`. El tono está subiendo. ¿Cuál es la acción profesional correcta?

- [ ] **a)** Escalarlo al CTO.
- [ ] **b)** Escribir un comentario de 500 palabras explicando por qué tienes razón.
- [ ] **c)** Detener la escritura, agendar una llamada de 5 minutos, llegar a un acuerdo verbal, y documentarlo.

SOLUCIÓN

c). El texto es terrible para resolver conflictos emocionales/opiniones. Una llamada humaniza la interacción y desbloquea el PR.

AUTOEVALUACIÓN

- [ ] ¿He eliminado la palabra “Tú” de mis críticas?
- [ ] ¿He explicado el “Por qué” en mis Blockers?

EVALUACIÓN: CODE REVIEWS EMPÁTICOS

FICHA TÉCNICA

- **Tema**: 5.2.1 Revisión de Código
- **Nivel**: Intermedio

CUESTIONARIO

Pregunta 1

¿Cuál es la diferencia entre un comentario `[BLOCKER]` y un `[NIT]`?

- [ ] **a)** `[BLOCKER]` es para errores de sintaxis; `[NIT]` es para errores de lógica.
- [ ] **b)** `[BLOCKER]` impide el merge por riesgo grave; `[NIT]` es una sugerencia menor que no impide el merge.
- [ ] **c)** `[BLOCKER]` lo escribe el jefe; `[NIT]` lo escriben los juniors.

Pregunta 2: Comunicación No Violenta

Elige la mejor forma de reportar un bug en un PR:

- [ ] **a)** "¿Por qué hiciste esto así? Es muy lento."
- [ ] **b)** "Has introducido una regresión de performance."
- [ ] **c)** "Veo que este bucle anidado podría causar lentitud con muchos datos. ¿Podríamos optimizarlo?"

Pregunta 3

¿Por qué se recomienda evitar el pronombre "Tú" en los comentarios?

- [ ] **a)** Porque es informal.
- [ ] **b)** Porque tiende a sonar acusatorio y pone al autor a la defensiva.
- [ ] **c)** Porque Google lo prohíbe.

Pregunta 4

Si ves un error de indentación (espacios vs tabs) en un archivo que no tiene linter configurado, ¿qué haces?

- [ ] **a)** Marcarlo como `[BLOCKER]` y detener el deploy.
- [ ] **b)** Marcarlo como `[NIT]` o arreglarlo tú mismo si tienes permisos, y sugerir configurar un Linter automático (Action Item).
- [ ] **c)** Ignorarlo.

Pregunta 5

¿Cuál es el objetivo principal de un Code Review?

- [ ] **a)** Encontrar culpables.

- [ ] **b)** Asegurar la calidad del código y compartir conocimiento (mentoring).
- [ ] **c)** Retrasar el lanzamiento para molestar a Producto.

SOLUCIONARIO

1. **b)**. Claridad en la severidad reduce ansiedad.
2. **c)**. Ataca el problema (“bucle”), no a la persona, y propone solución.
3. **b)**. Separa la identidad del autor del artefacto (código).
4. **b)**. Los problemas de estilo deben ser resueltos por máquinas (Linters), no humanos.
5. **b)**. Es un mecanismo de control de calidad y educación cruzada.

MODULO 6

TEMA 6.1.1: EL PORTAL DEL DESARROLLADOR (PROYECTO BOOMERANG)

Tiempo estimado: 45 minutos (Lectura + Inicio de Proyecto) **Nivel**: Experto **Prerrequisitos**: Módulos 1 al 5

¿Por qué importa este concepto?

Hasta ahora has escrito “piezas” sueltas: un README, un endpoint de API, un Postmortem. Pero los desarrolladores no consumen piezas sueltas. Consumen una **Experiencia de Producto**.

Un **Developer Portal** es la tienda donde vendes tu tecnología. Si la tienda está desordenada, nadie comprará, por muy bueno que sea el producto. El “Proyecto Boomerang” se llama así porque todo lo que has aprendido regresa para integrarse aquí.

Conexión con conocimientos previos

- **Módulo 1 (Escritura)**: El tono de tu portal.
- **Módulo 2 (Código)**: Los SDKs y ejemplos.
- **Módulo 3 (APIs)**: La referencia Swagger/OpenAPI.
- **Módulo 4 (Arquitectura)**: Decisiones de diseño (ADRs).
- **Módulo 5 (Soporte)**: Cómo contactar cuando algo falla.

Anatomía de un Developer Portal de Clase Mundial

Analiza portales como *Stripe Docs* o *Twilio Docs*. Todos tienen 4 pilares:

1. La Landing Page (El Gancho)

Responde en 5 segundos: *"¿Qué hace esto y por qué debería importarme?"*.

- **Héroe**: Título claro + Botón "Get Started".
- **Value Props**: "Integra pagos en 5 líneas de código".
- **Hello World**: Un snippet de código funcional que muestra simplicidad.

2. Guías y Tutoriales (El "Cómo")

Aquí aplicas la **Pirámide Invertida**.

- **Quickstart**: De 0 a "Hello World" en 15 minutos.
- **Tutoriales Temáticos**: "Cómo manejar reembolsos", "Cómo autenticar usuarios".

3. Referencia de API (El Diccionario)

Generada automáticamente (Swagger), pero curada humanamente.

- Debe incluir: Endpoints, Parámetros, Códigos de Error (Traffic Lights), y Ejemplos de Request/Response en varios lenguajes (cURL, Python, Node).

4. Soporte y Comunidad (La Red de Seguridad)

- **Status Page**: ¿Está caída la API?
- **Changelog**: ¿Qué cambió recientemente?
- **Stack Overflow / Discord**: Dónde pedir ayuda humana.

EL PROYECTO FINAL: "Boomerang"

Tu misión es diseñar la estructura de documentación para una API ficticia: **"Meteor Weather API"**.

Escenario

Meteor es una API que provee datos del clima histórico y predicciones futuras para aplicaciones de agricultura. Acaban de lanzar la versión 2.0 y la documentación actual es un

PDF de 50 páginas. Tu trabajo es modernizarla.

Entregable Requerido

No tienes que programar el portal. Tienes que crear la **Arquitectura de Información** en un archivo `PORTAL_STRUCTURE.md` que contenga:

1. **Home Copy**: Título y propuesta de valor.
2. **Índice de Navegación**: ¿Qué secciones tendrá el menú lateral?
3. **Diseño del Quickstart**: Pasos numerados del tutorial "Obtén el clima de tu ciudad en 5 minutos".
4. **Matriz de Errores**: Lista de 3 errores comunes de la API y cómo los documentarías (Status Code + Mensaje Humano).

Este proyecto demostrará que puedes pensar no solo como escritor, sino como **Product Manager de Documentación**.

Resumen del concepto

En una frase: La documentación es un producto, no un artefacto. Trátala como tal.

Regla de Oro: La métrica de éxito de un portal es el **TTHW (Time to Hello World)**. ¿Cuánto tarda un dev nuevo en hacer su primera llamada exitosa?

Siguiente paso: Manos a la obra. Generaremos ejercicios para bocetar este portal.

PROYECTO BOOMERANG: EJECUCIÓN

METADATA

- **Módulo**: Proyecto Integrador Final
- **Objetivos evaluados**:
    1. Diseñar la Arquitectura de Información de un Developer Portal.
    2. Aplicar principios de copywriting técnico (Value Props).
    3. Estructurar un Quickstart efectivo.
- **Entregable**: Un archivo `PORTAL_STRUCTURE.md` (simulado).

PASO 1: La Landing Page (El Pitch)

INSTRUCCIONES

Estás diseñando la Home de “Meteor Weather API”. Escribe los siguientes 3 elementos para la portada:

1. **H1 (Título Principal)**: Debe ser claro y orientado a la acción. No uses “Meteor API”. Usa algo que diga *qué obtiene el usuario*.
2. **Subtítulo (Propuesta de Valor)**: En 2 líneas, explica por qué somos mejores que la competencia (ej. OpenWeatherMap). Pista: Habla de latencia o precisión histórica.
3. **Snippet “Hello World”**: Escribe una llamada `curl` ficticia que devuelva el clima de Tokio. Debe parecer simple.

PASO 2: Arquitectura de Navegación (Card Sorting)

INSTRUCCIONES

Tienes los siguientes contenidos desordenados. Organízalos en un Menú Lateral con Categorías lógicas (ej. “Inicio”, “Guías”, “Referencia”, “Soporte”).

- Cómo autenticarse con OAuth2.
- Endpoint GET /weather/history.
- Librería para Python.
- Preguntas Frecuentes (FAQ).
- Endpoint GET /weather/forecast.
- Primeros pasos: Tu primera request.
- Códigos de Error y Límites de Rate.
- SDK para Node.js.
- Estado del Servicio (Uptime).

PASO 3: El Quickstart (De 0 a Hero)

INSTRUCCIONES

Escribe el índice (pasos) para el tutorial “Quickstart”. No escribas el contenido completo, solo los **títulos de los pasos**. El objetivo es que el usuario tenga un JSON con el clima en su terminal en menos de 5 pasos.

Ejemplo:

1. *Regístrate para obtener una API Key.*
2. *...*

SOLUCIÓN MODELO (PARA AUTO-REVISIÓN)

Paso 1

- **H1**: “Pronósticos hiper-locales para agricultura de precisión.”
- **Subtítulo**: “Accede a 50 años de datos históricos y predicciones minuto a minuto con nuestra API de baja latencia. Diseñada para agrónomos y desarrolladores.”
- **Snippet**:

  ```
  curl "https://api.meteor.com/v2/weather?city=Tokyo&key=YOUR_KEY"
  ```

Paso 2

- **Getting Started**: Primeros pasos, Autenticación (OAuth2), Códigos de Error.
- **Guides**: Quickstart.
- **API Reference**: GET /forecast, GET /history.
- **SDKs**: Python, Node.js.
- **Support**: FAQ, Status.

Paso 3

1. Obtén tu API Key gratuita.
2. Instala nuestro cliente (o usa cURL).
3. Haz tu primera petición (Copy-Paste).
4. Interpreta la respuesta JSON.

EVALUACIÓN: EL PORTAL DEL DESARROLLADOR

FICHA TÉCNICA

- **Tema**: 6.1.1 Proyecto Final

- **Nivel**: Experto

Pregunta 1

¿Qué métrica es fundamental para medir el éxito de la Landing Page de un Developer Portal?

- [ ] **a)** El número de visitas totales.
- [ ] **b) TTHW (Time To Hello World)**: El tiempo que tarda un dev en hacer su primera llamada exitosa.
- [ ] **c)** La cantidad de palabras en la documentación.

Pregunta 2

¿Cuál es la función principal de la sección “Quickstart”?

- [ ] **a)** Explicar teóricamente cómo funciona la arquitectura de la API.
- [ ] **b)** Proveer una gratificación instantánea (una victoria rápida) para motivar al desarrollador a seguir explorando.
- [ ] **c)** Listar todos los errores posibles.

Pregunta 3

En la anatomía de un portal, ¿dónde debería vivir la especificación técnica detallada de cada endpoint (parámetros, tipos, headers)?

- [ ] **a)** En la Landing Page.
- [ ] **b)** En las Guías Conceptuales.
- [ ] **c)** En la **Referencia de API** (API Reference).

Pregunta 4

Si tu API cambia y rompe compatibilidad (Breaking Change), ¿dónde debe comunicarse esto principalmente?

- [ ] **a)** En un ADR interno solamente.
- [ ] **b)** En el **Changelog** y mediante comunicación proactiva a los usuarios afectados.
- [ ] **c)** En el footer de la página web.

Pregunta 5

¿Por qué llamamos a la documentación un “Producto”?

- [ ] **a)** Porque se vende por separado.

- [ ] **c)** Es solo una metáfora sin sentido.

SOLUCIONARIO

1. **b)**. La velocidad de activación es clave en Developer Experience (DX).
2. **b)**. Reduce la fricción inicial y genera confianza.
3. **c)**. Es el lugar canónico para los detalles técnicos precisos.
4. **b)**. Transparencia y respeto por el tiempo del usuario.
5. **b)**. Requiere mantenimiento y evolución constante basada en feedback.

Tabla de Contenidos