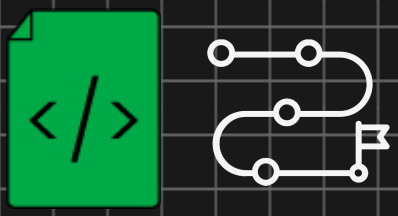

Roadmap para programadores junior

[EDICIÓN 2026]

Guía de conocimientos esenciales para desarrolladores en la era de la IA

mouredev pro

mouredev.pro/recursos

Índice

INTRODUCCIÓN

EDICIÓN 2026: FUNDAMENTOS + IA

Guía de estudio agnóstica de conocimientos esenciales para desarrolladores que inician su carrera en la era de la Inteligencia Artificial.

Su filosofía es actuar como un mapa de ruta detallado conceptual, pero como lector debes asumir la responsabilidad de investigación, el estudio activo y la búsqueda de recursos externos necesarios para la total ampliación y dominio de cada uno de los apartados aquí presentados.

Al final de la guía encontrarás un apartado de próximos pasos.

Espero que te sirva de mucha ayuda.

- **Brais Moure (@mouredev)**

Características de esta guía

✓ 100% agnóstica a tecnologías específicas
✓ Integración de IA en cada fase del aprendizaje
✓ Roadmap visual de 12 meses
✓ Fundamentos de ingeniería de software que nunca caducan
✓ Preparación para el mercado laboral actual

PARTE I: FUNDAMENTOS ESENCIALES

El desarrollador en 2026

El panorama del desarrollo de software ha cambiado drásticamente. La llegada de asistentes de código con IA, modelos de lenguaje avanzados y herramientas de automatización han transformado cómo trabajamos. Sin embargo, paradójicamente, esto hace que los fundamentos sean **más importantes que nunca**.

¿Por qué los fundamentos importan más ahora?

Las herramientas de IA pueden generar código, pero no pueden evaluar si ese código es correcto, eficiente o seguro. Esa responsabilidad recae en ti. Un desarrollador con fundamentos sólidos puede:

- Evaluar críticamente el código generado por IA.
- Detectar errores sutiles y vulnerabilidades de seguridad.
- Guiar a la IA con prompts precisos y contextualizados.
- Diseñar arquitecturas que la IA no puede concebir sola.
- Adaptarse rápidamente a cualquier tecnología nueva.
- Resolver problemas cuando la IA no tiene respuestas.
- Ampliar el contexto más allá del código.

El nuevo perfil del junior

El desarrollador junior de 2026 no es quien memoriza sintaxis o sabe usar un framework específico. Es quien entiende conceptos profundamente, usa IA como multiplicador de productividad, y tiene criterio para tomar decisiones técnicas informadas.

> 🤖 ***IA en acción:*** *La IA no reemplaza desarrolladores, pero los desarrolladores que usan IA sí reemplazan a los que no la usan.*
> *Este roadmap te prepara para ambos mundos: fundamentos sólidos + dominio de herramientas de IA.*

Filosofía de esta guía

- **Agnóstica:** Ningún lenguaje, framework o herramienta específica.
- **Transferible:** Conceptos que aplican a cualquier stack tecnológico.
- **Práctica:** Enfocada en habilidades que se usan en el trabajo real.
- **Integrada con IA:** Cada sección incluye cómo la IA potencia ese conocimiento.
- **Actualizada:** Preparada para el mercado laboral de 2026 y más allá.

Fundamentos de programación

Estos son los bloques de construcción universales que comparten todos los lenguajes de programación. Dominarlos te permite aprender cualquier lenguaje nuevo en días, no meses.

Sintaxis y Semántica

La **sintaxis** son las reglas gramaticales del lenguaje: cómo escribir instrucciones válidas. La **semántica** es el significado de esas instrucciones: qué hacen realmente. Entender la diferencia es crucial para debuggear.

Variables y Tipos de Datos

Tipos Primitivos

- **Números enteros:** valores sin decimales, positivos o negativos.
- **Números decimales:** valores con punto flotante, precisión limitada.
- **Booleanos:** verdadero o falso, base de toda lógica.
- **Caracteres:** símbolos individuales de texto.
- **Nulo/Vacío:** ausencia intencional de valor.

Tipos Compuestos

- **Cadenas de texto:** secuencias de caracteres.
- **Arrays/Listas:** colecciones ordenadas de elementos.
- **Objetos:** agrupaciones de datos relacionados.
- **Tuplas:** colecciones inmutables de tamaño fijo.

Sistemas de Tipos

- **Tipado estático:** tipos verificados antes de ejecutar (en compilación).
- **Tipado dinámico:** tipos verificados durante la ejecución.
- **Tipado fuerte:** conversiones de tipo deben ser explícitas.
- **Tipado débil:** conversiones automáticas entre tipos.

> 💡 ***Consejo:*** *Cada sistema de tipos tiene ventajas. Los estáticos capturan errores antes, los dinámicos permiten prototipar más rápido. Aprende ambos enfoques.*

Operadores

- **Aritméticos:** +, -, *, /, % (módulo), ** (potencia)
- **Comparación:** ==, !=, <, >, <=, >= (igualdad vs. identidad)
- **Lógicos:** AND, OR, NOT (evaluación de cortocircuito)
- **Asignación:** =, +=, -=, *=, /= (asignación compuesta)
- **Bit a bit:** AND, OR, XOR, NOT, shifts (nivel de bits)
- **Ternario:** condición ? valor_si : valor_no

Estructuras de Control

Condicionales

Permiten ejecutar código diferente según condiciones. El **if/else** es el más básico. El **switch/match** es útil para múltiples casos. Aprende a evitar anidamientos excesivos (early return, guard clauses).

Bucles

- **for:** cuando conoces el número de iteraciones.
- **while:** cuando la condición de parada es dinámica.
- **for-each/for-in:** para iterar sobre colecciones.
- **do-while:** cuando necesitas al menos una ejecución.

Control de Flujo

- **break:** sale del bucle completamente.
- **continue:** salta a la siguiente iteración.
- **return:** termina la función y devuelve valor.

Funciones

Las **funciones** son el mecanismo principal de abstracción y reutilización. Una buena función hace una sola cosa, tiene un nombre descriptivo, y es fácil de testear de forma aislada.

- **Declaración:** definir nombre, parámetros y cuerpo.
- **Parámetros:** por valor vs. por referencia, valores por defecto.
- **Retorno:** valores de salida, múltiples retornos, void.
- **Scope:** ámbito de variables, closures, hoisting.
- **Funciones anónimas:** lambdas, arrow functions, callbacks.
- **Recursividad:** funciones que se llaman a sí mismas.

> 🤖 ***IA en acción:*** *Usa IA para que te explique código que no entiendes, generar ejemplos de uso de funciones, o sugerir nombres descriptivos. Pero siempre verifica que el código generado hace lo que esperas. La regla de oro es acabar entendiendo todo el trabajo que puede llegar a hacer por nosotros.*

Estructuras de Datos

Las **estructuras de datos** son formas de organizar información en memoria. Elegir la estructura correcta puede significar la diferencia entre un programa que funciona instantáneamente y uno que tarda minutos.

Estructuras Lineales

Arrays y Listas

Colecciones ordenadas con acceso por índice. Los **arrays** tienen tamaño fijo y acceso O(1). Las **listas dinámicas** pueden crecer pero insertar/eliminar en medio es O(n).

> ⚠️ ***Importante:*** *La referencias a la Notación Big O, como O(1) o O(n), que aparecen en estos apartados, se explican más adelante en la guía.*

Pilas (Stacks)

LIFO: Last In, First Out. Como una pila de platos. Operaciones: push (añadir arriba), pop (quitar arriba), peek (ver arriba). Uso: deshacer operaciones, navegación de historial, evaluación de expresiones.

Colas (Queues)

FIFO: First In, First Out. Como una cola de supermercado. Operaciones: enqueue (añadir al final), dequeue (sacar del frente). Uso: procesar tareas en orden, buffers, BFS en grafos.

Estructuras Asociativas

Diccionarios / Mapas / Hash Tables

Almacenan pares clave-valor con acceso O(1) promedio por clave. Fundamentales para búsquedas rápidas, cachés, recuento de frecuencias. La clave debe ser hasheable (inmutable y única).

Conjuntos (Sets)

Colecciones de elementos únicos sin duplicados. Operaciones: pertenencia O(1), unión, intersección, diferencia. Uso: eliminar duplicados, verificar existencia.

Estructuras Jerárquicas

Árboles

Estructuras con nodos conectados jerárquicamente. Cada nodo tiene padre (excepto la raíz) e hijos. **Árboles binarios de búsqueda** permiten encontrar elementos en O(log n). Uso: sistemas de archivos, DOM, índices de Base de Datos.

Grafos

Nodos conectados por aristas, sin restricción jerárquica. Pueden ser dirigidos o no, pesados o no. Uso: estructura en redes sociales, mapas, dependencias.

Tuplas y Registros

Las **tuplas** son colecciones inmutables de tamaño fijo, útiles para retornar múltiples valores. Los **registros** (structs, records) agrupan datos relacionados con campos nombrados.

> 💡 ***Consejo:*** *No memorices todas las estructuras. Aprende a identificar qué operaciones necesitas (búsqueda, inserción, orden) y elige la estructura que las hace eficientes.*

Algoritmos y Complejidad

Un algoritmo es una secuencia de pasos para resolver un problema. La complejidad algorítmica mide cómo escala el rendimiento cuando crece la entrada.

Notación Big O

Big O describe el peor caso de cómo crece el tiempo o espacio con el tamaño de entrada (n):

- **O(1) Constante:** no importa el tamaño, mismo tiempo (acceso a array).
- **O(log n) Logarítmica:** se divide el problema en cada paso (búsqueda binaria).
- **O(n) Lineal:** proporcional al tamaño (recorrer lista).
- **O(n log n) Linearítmica:** ordenación eficiente (mergesort, quicksort).
- **O(n^2) Cuadrática:** bucles anidados (ordenación burbuja).
- **O(2^n) Exponencial:** combinaciones, evitar si es posible.

Algoritmos de Búsqueda

- **Búsqueda lineal:** recorrer uno a uno, O(n), funciona siempre.
- **Búsqueda binaria:** dividir a la mitad, O(log n), requiere datos ordenados.
- **Búsqueda en hash:** O(1) promedio usando diccionarios.

Algoritmos de Ordenación

- **Burbuja, Inserción, Selección:** O(n^2), simples pero ineficientes.
- **Quicksort:** O(n log n) promedio, muy usado, divide y vencerás.
- **Mergesort:** O(n log n) garantizado, estable, usa más memoria.
- **Ordenación por recuento/radix:** O(n) para casos específicos.

Técnicas Algorítmicas

- **Divide y vencerás:** dividir problema, resolver partes, combinar.
- **Programación dinámica:** memorizar subproblemas para no recalcular.
- **Algoritmos voraces (greedy):** elegir lo mejor localmente.
- **Backtracking:** probar opciones, retroceder si no funcionan.
- **Dos punteros:** recorrer desde extremos hacia el centro.

🤖 ***IA en acción:*** *La IA puede sugerir algoritmos para problemas específicos y explicar su complejidad. Úsala para entender soluciones, pero practica a implementarlas tú mismo para internalizar los patrones.*

Paradigmas de Programación

Los paradigmas son formas de pensar y estructurar programas. La mayoría de lenguajes modernos son multiparadigma: permiten combinar enfoques según convenga.

Programación Imperativa

Describe **cómo** hacer algo paso a paso. El programa es una secuencia de instrucciones que modifican el estado. Es el paradigma más intuitivo y la base sobre la que se construyen los demás.

Programación Orientada a Objetos (POO)

Organiza el código alrededor de objetos que combinan datos (atributos) y comportamiento (métodos). Es el paradigma dominante en la industria.

- **Clases e instancias:** plantillas y objetos concretos.
- **Encapsulamiento:** ocultar detalles, exponer interfaz pública.
- **Herencia:** crear clases basadas en otras (usar con moderación).
- **Polimorfismo:** mismo mensaje, diferentes comportamientos.
- **Abstracción:** modelar solo lo relevante del dominio.

Programación Funcional

Trata la computación como evaluación de funciones matemáticas. Enfatiza inmutabilidad y funciones puras. Produce código más predecible y testeable.

- **Funciones puras:** misma entrada = misma salida, sin efectos secundarios.
- **Inmutabilidad:** los datos no cambian, se crean nuevas versiones.
- **Funciones de orden superior:** funciones como argumentos o retornos.
- **Map, Filter, Reduce:** transformaciones declarativas de colecciones.
- **Composición:** combinar funciones simples para crear complejas.

Programación Declarativa

Describe **qué** resultado quieres, no cómo obtenerlo. El sistema decide los pasos. SQL ('dame los usuarios mayores de 18') y HTML ('esto es un párrafo') son declarativos.

> 💡 ***Consejo:*** *POO y funcional no son opuestos. Los mejores desarrolladores combinan ambos: objetos para modelar entidades del dominio, funciones puras para transformaciones.*

PARTE II: HERRAMIENTAS FUNDAMENTALES

La Línea de Comandos (Terminal)

La **terminal** es una de las herramientas más potentes de un desarrollador. Aunque parezca intimidante al principio, dominarla te hace exponencialmente más productivo y es requisito para muchas tareas de desarrollo.

¿Por qué aprender terminal?

- Muchas herramientas solo existen en línea de comandos.
- Es más rápida que interfaces gráficas para muchas tareas.
- Permite automatizar tareas repetitivas con scripts.
- Es universal: funciona igual en servidores remotos.
- Git, gestores de paquetes, y herramientas de build la requieren.
- Te da control total sobre tu sistema.

Conceptos Fundamentales

- **Shell:** el programa que interpreta tus comandos.
- **Prompt:** donde escribes los comandos.
- **Comando:** instrucción a ejecutar.
- **Argumentos:** datos que pasas al comando.
- **Flags/Opciones:** modificadores del comportamiento (-v, --help).
- **PATH:** directorios donde el sistema busca comandos.
- **Variables de entorno:** configuración del sistema.

Navegación del Sistema de Archivos

- **pwd:** muestra el directorio actual (dónde estás).
- **ls:** lista archivos y carpetas (con -la para ver todo).
- **cd:** cambia de directorio (cd .. sube un nivel).
- **mkdir:** crea un directorio nuevo.
- **rmdir:** elimina un directorio vacío.
- **Rutas absolutas:** desde la raíz (/home/usuario/proyecto).
- **Rutas relativas:** desde donde estás (./src, ../config).

Manipulación de Archivos

- **touch:** crea archivo vacío o actualiza fecha.
- **cat:** muestra contenido de archivo.
- **less/more:** ver archivos largos con paginación.
- **head/tail:** ver inicio/final de archivo.
- **cp:** copiar archivos (cp origen destino).
- **mv:** mover o renombrar archivos.
- **rm:** eliminar archivos (¡cuidado! no hay papelera).
- **find:** buscar archivos por nombre o propiedades.
- **grep:** buscar texto dentro de archivos.

Redirección y Pipes

La verdadera potencia de la terminal viene de combinar comandos:

- **>** redirige salida a archivo (sobrescribe).
- **>>** redirige salida a archivo (añade al final).
- **<** usa archivo como entrada.
- **|** (pipe) conecta la salida de un comando con la entrada de otro.
- Ejemplo: `cat archivo.txt | grep 'error' | wc -l` (cuenta líneas con 'error').

Otros Comandos Útiles

- **echo:** imprime texto (útil en scripts).
- **man:** manual de cualquier comando (man ls).
- **which:** dónde está instalado un comando.
- **chmod:** cambiar permisos de archivos.
- **curl/wget:** descargar archivos de internet.
- **tar/zip:** comprimir y descomprimir archivos.
- **ssh:** conectar a servidores remotos.
- **history:** ver comandos anteriores.

💡 ***Consejo:*** *Usa Tab para autocompletar nombres de archivos y comandos. Usa las flechas arriba/abajo para navegar el historial. Ctrl+R busca en el historial. Estos atajos te ahorrarán horas.*

🤖 ***IA en acción:*** *La IA es perfecta para generar comandos complejos. Describe lo que quieres hacer ('buscar todos los archivos .log mayores de 100MB') y te dará el comando. Pero aprende los básicos para entender qué está sugiriendo.*

Entornos de Desarrollo (IDEs)

Tu editor o **IDE** es donde pasarás la mayor parte de tu tiempo. Configurarlo bien y conocer sus funciones te hace significativamente más productivo.

Editor vs. IDE

- **Editor de texto avanzado:** ligero, rápido, extensible (VS Code, Cursor, Vim).
- **IDE completo:** más pesado, más funciones integradas (IntelliJ, Visual Studio).
- **Recomendación:** empieza con un editor extensible, añade lo que necesites.

Funciones Esenciales que Debes Dominar

Navegación

- Ir a archivo por nombre (Ctrl/Cmd + P)
- Ir a símbolo/función (Ctrl/Cmd + Shift + O)
- Ir a definición (F12 o Ctrl/Cmd + Click)
- Volver atrás (Alt + ←)
- Buscar en todo el proyecto (Ctrl/Cmd + Shift + F)

Edición

- Selección múltiple (Ctrl/Cmd + D)
- Mover líneas arriba/abajo (Alt + ↑/↓)
- Duplicar línea (Ctrl/Cmd + Shift + D)
- Comentar/descomentar (Ctrl/Cmd + /)
- Formatear documento (Shift + Alt + F)

Refactoring

- Renombrar símbolo (F2)
- Extraer a función/variable
- Organizar imports

Extensiones Imprescindibles

- **Linter:** detecta errores mientras escribes.
- **Formateador:** formato consistente automático (Prettier, Black).
- **Autocompletado inteligente:** sugerencias contextuales.
- **Git integrado:** ver cambios, hacer commits sin salir.
- **Snippets:** plantillas de código frecuente.
- **Temas:** reduce fatiga visual (tema oscuro recomendado).

El Debugger Integrado

El debugger del IDE es una de las herramientas más poderosas y subutilizadas:

- **Breakpoints:** pausar ejecución en una línea específica.
- **Step over:** ejecutar línea actual, ir a la siguiente.
- **Step into:** entrar dentro de la función llamada.
- **Step out:** salir de la función actual.
- **Watch:** observar el valor de variables.
- **Call stack:** ver la cadena de llamadas que llegó hasta aquí.

> 💡 ***Consejo:*** *Dedica una tarde a aprender los atajos de teclado de tu editor. El tiempo invertido se recupera en días. Imprime una cheatsheet y ponla junto a tu monitor.*

> 🤖 ***IA en acción:*** *Los asistentes de código integrados en el IDE (Copilot, Cursor, Claude Code, etc.) son muy útiles. Aprende a usarlos junto con los atajos tradicionales.*

Control de Versiones con Git

Git es el sistema de control de versiones dominante en la industria. No es opcional: es tan fundamental como saber programar. Permite rastrear cambios, colaborar con otros, y mantener un historial completo de tu proyecto.

Conceptos Fundamentales

- **Repositorio (repo):** carpeta con todo el código y su historial.
- **Commit:** instantánea del proyecto en un momento dado.
- **Branch (rama):** línea paralela de desarrollo independiente.
- **HEAD:** puntero al commit actual donde estás.
- **Staging area:** zona de preparación antes de commit.
- **Remote:** repositorio en servidor (GitHub, GitLab, etc.).
- **Clone:** copiar un repositorio remoto a tu máquina.

Comandos Esenciales de Git

Configuración inicial

- **git config:** configurar nombre, email, editor.
- **git init:** crear un nuevo repositorio en carpeta actual.
- **git clone [url]:** descargar repositorio remoto.

Flujo básico de trabajo

- **git status:** ver estado actual (archivos modificados, staged).
- **git add [archivo]:** añadir archivo al staging area.
- **git add .:** añadir todos los archivos modificados.
- **git commit -m 'mensaje':** crear commit con mensaje.
- **git log:** ver historial de commits (--oneline para resumido).
- **git diff:** ver cambios no staged.

Trabajar con ramas

- **git branch:** listar ramas locales.
- **git branch [nombre]:** crear nueva rama.
- **git checkout [rama]:** cambiar a otra rama.
- **git checkout -b [rama]:** crear y cambiar a nueva rama.
- **git merge [rama]:** fusionar rama en la actual.
- **git branch -d [rama]:** eliminar rama.

Sincronizar con remoto

- **git remote add origin [url]:** conectar con repositorio remoto.
- **git push:** subir commits locales al remoto.
- **git push -u origin [rama]:** primera vez que subes una rama.
- **git pull:** descargar y fusionar cambios del remoto.
- **git fetch:** descargar cambios sin fusionar.

Deshacer cambios

- **git checkout -- [archivo]:** descartar cambios no staged.
- **git reset HEAD [archivo]:** quitar archivo del staging.
- **git reset --soft HEAD~1:** deshacer último commit (mantiene cambios).
- **git reset --hard HEAD~1:** deshacer último commit (pierde cambios).
- **git stash:** guardar cambios temporalmente.

Plataformas de Hosting (GitHub, GitLab, etc.)

Las plataformas de hosting añaden colaboración sobre Git:

- **Pull/Merge Request:** proponer cambios para revisión antes de merge.
- **Issues:** rastrear bugs, tareas y mejoras.
- **Fork:** crear tu copia de un repositorio ajeno.
- **Code Review:** revisar código de otros antes de aceptar.
- **Actions/CI:** automatización de tests y despliegues.
- **Wiki/Docs:** documentación del proyecto.

Flujos de Trabajo

- **Feature branches:** una rama por funcionalidad, merge cuando está lista.
- **Trunk-based:** commits pequeños directo a main, integración continua.
- **Gitflow:** ramas para develop, release, hotfix (más complejo).
- **Fork workflow:** para contribuir a proyectos open source.

Buenas Prácticas

- Commits pequeños, atómicos y frecuentes.
- Mensajes descriptivos: 'Añade validación de email en registro'.
- NUNCA commits de credenciales o secretos (.env, API keys).
- Usa .gitignore para excluir: node_modules, .env, builds, logs.
- Revisa con git diff antes de commit.
- Mantén main/master siempre funcional.
- Haz pull antes de empezar a trabajar cada día.

⚠️ ***Importante:*** *Si subes credenciales o secretos por error, cambiarlos NO es suficiente. Quedan en el historial de Git. Debes rotar las credenciales inmediatamente y considerar reescribir el historial.*

🤖 ***IA en acción:*** *La IA puede generar mensajes de commit descriptivos, explicar qué hace un comando Git, ayudarte a resolver conflictos de merge, y sugerir el flujo de trabajo adecuado para tu equipo.*

Debugging y Resolución de Problemas

Esta es probablemente la habilidad más importante que diferencia a un junior productivo de uno que se bloquea constantemente. Debuggear no es magia: es un proceso sistemático que se puede aprender.

Mentalidad de Debugging

- **Los bugs no son aleatorios:** el código hace exactamente lo que le dijiste, no lo que querías.
- **Reproduce primero:** no puedes arreglar lo que no puedes reproducir.
- **Un cambio a la vez:** si cambias muchas cosas, no sabrás qué lo arregló.
- **No asumas:** verifica cada suposición con datos.

Cómo Leer Mensajes de Error

Los mensajes de error son tu mejor amigo. Aprende a leerlos:

1. **Tipo de error:** SyntaxError, TypeError, NullPointerException...
2. **Mensaje:** descripción de qué salió mal.
3. **Stack trace:** la cadena de llamadas que llegó al error.
4. **Archivo y línea:** dónde ocurrió (empieza por arriba del stack).

```Shell
TypeError: Cannot read property 'name' of undefined
    at getUserName (user.js:15)      ← El error está aquí
    at displayProfile (profile.js:42)
    at main (app.js:10)
```

Técnicas de Debugging

Print Debugging (el básico)

Añadir prints/logs para ver valores en puntos específicos:

- ¿Qué valor tiene esta variable aquí?
- ¿Se está ejecutando esta línea?
- ¿Cuántas veces entra a este bucle?

Binary Search Debugging

Cuando no sabes dónde está el bug:

1. Pon un log a la mitad del código sospechoso.
2. ¿El bug ocurre antes o después?
3. Repite en la mitad relevante.
4. Reduce hasta encontrar la línea exacta.

Rubber Duck Debugging

Explica el código línea por línea a un pato de goma (o a ti mismo en voz alta). Frecuentemente encuentras el error al verbalizarlo.

Debugging con el Debugger

Usa breakpoints para:

- Pausar en una línea específica
- Inspeccionar todas las variables en ese momento
- Ejecutar paso a paso
- Ver el call stack completo

Proceso Sistemático

1. **Reproduce el bug** de forma consistente.
2. **Aísla el problema** - ¿cuál es el input mínimo que lo causa?
3. **Formula hipótesis** - ¿qué crees que está mal?
4. **Verifica la hipótesis** - con logs, debugger, o tests.
5. **Arregla y verifica** - asegúrate de que realmente está arreglado.
6. **Escribe un test** - para que no vuelva a pasar.

Cómo Buscar Soluciones Efectivamente

Antes de buscar

- Lee el error completo.
- Intenta entenderlo tú primero.
- Simplifica el problema.

Cómo buscar

- Copia el mensaje de error exacto (sin datos específicos de tu código).
- Incluye el lenguaje/framework.
- Busca en: IA > documentación oficial > Stack Overflow > GitHub Issues.

Cómo evaluar respuestas

- ¿Es reciente? Las soluciones viejas pueden no aplicar.
- ¿Tiene votos/aceptada? Pero no confíes ciegamente.
- ¿Entiendes la solución? No copies código que no entiendas.
- ¿Hay efectos secundarios? Lee los comentarios.

Errores Comunes por Categoría

Errores de sintaxis

- Paréntesis/llaves sin cerrar.
- Punto y coma faltante (en lenguajes que lo requieren).
- Typos en nombres de variables.

Errores de tipos

- Null/undefined donde se esperaba un valor.
- String donde se esperaba número (o viceversa).
- Índice fuera de rango.

Errores de lógica

- Condición invertida (< en vez de >).
- Off-by-one (empezar en 1 en vez de 0).
- Variables con el valor de la iteración anterior.

Errores de estado

- Variable no inicializada.
- Estado modificado en lugar inesperado.
- Race condition en código asíncrono.

> 💡 ***Consejo:*** *Cuando estés completamente bloqueado, aléjate 15 minutos. Muchos bugs se resuelven después de un descanso, una ducha, o una noche de sueño.*

> 🤖 ***IA en acción:*** *Pega el error y el código relevante a la IA. Pregunta no solo cómo arreglarlo, sino POR QUÉ ocurre. Usa la IA para aprender, no solo para solucionar.*

PARTE III: CONCEPTOS INTERMEDIOS

Modularidad y Gestión de Dependencias

La **modularidad** es dividir el código en piezas independientes que se pueden desarrollar, testear y mantener por separado. Es esencial para proyectos que crecen más allá de unos pocos archivos.

Conceptos de Modularidad

- **Módulo:** unidad de código con responsabilidad definida.
- **Exportar:** hacer disponible para otros módulos.
- **Importar:** usar funcionalidad de otros módulos.
- **Namespace:** evitar colisiones de nombres.
- **Encapsulación:** ocultar detalles internos del módulo.

Gestión de Dependencias

Las **dependencias** son bibliotecas externas que tu proyecto usa. Gestionarlas bien es crucial para la reproducibilidad y seguridad:

- **Gestor de paquetes:** herramienta para instalar/actualizar dependencias.
- **Archivo de manifiesto:** lista de dependencias y sus versiones.
- **Archivo de lock:** versiones exactas para reproducibilidad.
- **Versionado semántico:** MAJOR.MINOR.PATCH (1.2.3).
- **Dependencias transitivas:** dependencias de tus dependencias.

Buenas Prácticas

- Mantén dependencias actualizadas (seguridad).
- Revisa qué instalas (supply chain attacks).
- Prefiere dependencias bien mantenidas y populares.
- Minimiza dependencias cuando sea práctico.
- Documenta por qué se usa cada dependencia.

> ⚠️ ***Importante:*** *Las dependencias son código de terceros que ejecutas. Una dependencia maliciosa puede comprometer todo tu proyecto. Verifica siempre.*

Manejo de Errores

El **manejo de errores** es lo que separa el código de juguete del código de producción. Los errores van a ocurrir; la pregunta es cómo los manejas.

Tipos de Errores

Errores de programación (bugs)

- Errores de lógica, typos, casos no contemplados.
- **Solución:** arreglar el código, no manejarlos silenciosamente.

Errores esperados (fallos operacionales)

- Red no disponible, archivo no existe, input inválido.
- **Solución:** manejarlos graciosamente, informar al usuario.

Errores irrecuperables

- Memoria agotada, disco lleno.
- **Solución:** fallar limpiamente, loguear para diagnóstico.

Mecanismos de Manejo

Excepciones (try/catch/finally)

```
None
try {
    // Código que puede fallar
} catch (error) {
    // Manejar el error
} finally {
    // Siempre se ejecuta (limpieza)
}
```

Valores de retorno

- Retornar null/undefined para indicar ausencia.
- Retornar objetos Result/Either con éxito o error.
- Códigos de error (menos común en lenguajes modernos).

Principios de Manejo de Errores

Fail Fast

Detectar y reportar errores lo antes posible. No continuar con datos inválidos esperando que "tal vez funcione".

No Atrapar Todo

```
None
// MAL - oculta todos los errores
try {
    hacerAlgo();
} catch (e) {
    // silencio
}

// BIEN - maneja errores específicos
try {
    hacerAlgo();
} catch (NetworkError e) {
    reintentar();
} catch (ValidationError e) {
    mostrarMensaje(e);
}
```

Mensajes de Error Útiles

- **Malo:** "Error"
- **Malo:** "Algo salió mal"
- **Bueno:** "No se pudo conectar al servidor de pagos después de 3 intentos"
- **Bueno:** "El email 'abc' no es válido: falta el símbolo @"

Propagar vs. Manejar

- **Maneja** si puedes hacer algo útil con el error.
- **Propaga** si la capa superior puede manejarlo mejor.
- **Nunca** ignores errores silenciosamente.

Errores en Código Asíncrono

Los errores en código asíncrono necesitan tratamiento especial:

- **Callbacks:** el primer parámetro suele ser el error.
- **Promesas:** usar .catch() o try/catch con async/await.
- **Eventos:** escuchar eventos de error.

```
None
// Promesas
fetchData()
    .then(data => process(data))
    .catch(error => handleError(error));

// Async/await
try {
    const data = await fetchData();
    process(data);
} catch (error) {
    handleError(error);
}
```

Patrones Comunes

Guard Clauses

Validar condiciones al inicio y retornar/lanzar temprano:

```
None
function processUser(user) {
    if (!user) throw new Error('User required');
    if (!user.email) throw new Error('Email required');

    // Lógica principal sin anidamiento
}
```

Default Values

Proporcionar valores por defecto seguros:

```
None
const name = user.name || 'Anónimo';
const port = config.port ?? 3000;
```

> 💡 ***Consejo:*** *Un buen mensaje de error debe responder: ¿Qué pasó? ¿Por qué? ¿Cómo solucionarlo?*

Asincronía y Concurrencia

La programación **asíncrona** permite ejecutar operaciones sin bloquear el programa. Es fundamental para aplicaciones que interactúan con redes, archivos o usuarios.

Conceptos Fundamentales

- **Síncrono:** una operación debe terminar antes de empezar la siguiente.
- **Asíncrono:** una operación puede iniciarse sin esperar a otras.
- **Bloqueante:** la operación detiene la ejecución hasta completarse.
- **No bloqueante:** permite continuar mientras la operación se procesa.
- **Concurrencia:** manejar múltiples tareas (no necesariamente simultáneas).
- **Paralelismo:** ejecutar múltiples tareas realmente al mismo tiempo.

Patrones Asíncronos

Callbacks

Funciones que se pasan como argumento y se ejecutan cuando la operación termina. Simples pero pueden crear 'callback hell' cuando se anidan.

Promesas / Futuros

Objetos que representan un valor que estará disponible en el futuro. Permiten encadenar operaciones (.then) y manejar errores (.catch) de forma más limpia que callbacks.

Async/Await

Sintaxis que hace que código asíncrono parezca síncrono. Más legible que promesas encadenadas. La función se 'pausa' en await sin bloquear.

Problemas Comunes

Callback Hell

Callbacks anidados que crean código ilegible:

```
None
// Evitar esto
getData(function(a) {
    getMoreData(a, function(b) {
        getEvenMore(b, function(c) {
            // Pirámide de la perdición
        });
    });
});
```

Race Conditions

Cuando el resultado depende del orden de operaciones asíncronas:

```
None
// ¿Cuál termina primero?
let result;
fetchA().then(a => result = a);
fetchB().then(b => result = b);
// result es impredecible
```

Errores No Capturados

Las excepciones en callbacks asíncronos no se propagan normalmente.

Consideraciones

- El manejo de errores es diferente en código asíncrono.
- Race conditions: cuidado con el orden de operaciones.
- Deadlocks: evitar esperas circulares.
- No todo debe ser asíncrono - añade complejidad.

> 💡 ***Consejo:*** *Empieza con código síncrono. Añade asincronía solo donde realmente la necesites: operaciones de red, archivos, o tareas que tardan.*

Programación Orientada a Eventos

En la **programación orientada a eventos**, el flujo del programa está determinado por eventos externos: clicks de usuario, mensajes de red, temporizadores. Es el modelo dominante en interfaces de usuario y servidores.

Conceptos Fundamentales

- **Evento:** notificación de que algo ha ocurrido.
- **Emisor:** origen que genera el evento.
- **Listener/Handler:** función que responde al evento.
- **Event loop:** ciclo que procesa eventos continuamente.
- **Event queue:** cola de eventos pendientes de procesar.

Patrones de Eventos

- **Observer:** objetos se suscriben para recibir notificaciones.
- **Pub/Sub:** publicadores y suscriptores desacoplados.
- **Event emitter:** objeto que puede emitir y escuchar eventos.
- **Event delegation:** un handler para múltiples elementos similares.

Buenas Prácticas

- Evita handlers que hacen demasiado y delega a funciones.
- Limpia listeners cuando ya no se necesitan (memory leaks).
- Usa debounce/throttle para eventos frecuentes.
- Documenta qué eventos emite cada componente.

Debounce y Throttle

Debounce

Espera a que paren los eventos antes de ejecutar. Útil para búsqueda mientras escribes:

- Usuario escribe: a...b...c...d
- Solo busca una vez cuando para de escribir.

Throttle

Ejecuta como máximo una vez cada X tiempo. Útil para scroll o resize:

- Usuario hace scroll continuo.
- Ejecuta cada 100ms, no en cada pixel.

Expresiones Regulares (Regex)

Las **expresiones regulares** son patrones para buscar y manipular texto. Son extremadamente potentes pero pueden ser crípticas. Todo desarrollador necesita conocer los básicos.

¿Cuándo usar Regex?

Buenos casos de uso:

- Validar formatos (email, teléfono, código postal).
- Buscar patrones en texto.
- Extraer partes de un string.
- Reemplazar patrones.

Malos casos de uso:

- Parsear HTML/XML (usa un parser).
- Validaciones muy complejas (usa lógica normal).
- Cuando un método de string simple basta.

Sintaxis Básica

Caracteres literales

- `abc` → encuentra "abc" exactamente

Metacaracteres

- `.` → cualquier carácter (excepto nueva línea)
- `^` → inicio de línea
- `$` → fin de línea
- `\d` → cualquier dígito [0-9]
- `\w` → cualquier carácter de palabra [a-zA-Z0-9_]
- `\s` → cualquier espacio en blanco

Cuantificadores

- `*` → cero o más
- `+` → uno o más
- `?` → cero o uno (opcional)
- `{3}` → exactamente 3
- `{2,5}` → entre 2 y 5

Clases de caracteres

- `[abc]` → a, b, o c
- `[^abc]` → cualquier cosa excepto a, b, c
- `[a-z]` → cualquier letra minúscula

Grupos

- `(abc)` → grupo de captura
- `(?:abc)` → grupo sin captura
- `a|b` → a o b

Ejemplos Comunes

None

```
# Email (simplificado)
^[\w.-]+@[\w.-]+\.\w+$

# Teléfono (formato español)
^(\+34|0034)?[6-9]\d{8}$

# URL
^https?://[\w.-]+(/[\w.-]*)*$

# Código postal español
^\d{5}$

# Contraseña (mínimo 8 chars, una mayúscula, un número)
^(?=.*[A-Z])(?=.*\d).{8,}$
```

Consejos Prácticos

- **Usa herramientas online** para probar regex (regex101.com).
- **Comenta tus regex** - en 6 meses no recordarás qué hace.
- **Empieza simple** y ve añadiendo complejidad.
- **Considera alternativas** - a veces código normal es más legible.
- **Cuidado con el rendimiento** - regex mal escritas pueden ser muy lentas.

💡 ***Consejo:*** *No intentes memorizar toda la sintaxis. Aprende los básicos y usa referencias cuando necesites algo específico.*

🤖 ***IA en acción:*** *La IA es excelente para generar y explicar regex. Describe lo que quieres encontrar en lenguaje natural y pide que te genere el patrón.*

PARTE IV: CALIDAD Y DISEÑO

Testing y Calidad del Código

Los **tests** son tu red de seguridad. Te permiten modificar código con confianza, documentan comportamiento esperado, y detectan errores antes de que lleguen a producción.

Pirámide de Testing

Tests Unitarios (base de la pirámide)

Prueban funciones o métodos aislados. Son rápidos, específicos y deberías tener muchos. Un test unitario verifica que una unidad de código hace lo que se espera.

Tests de Integración (medio)

Verifican que módulos o servicios funcionan correctamente juntos. Más lentos que unitarios pero detectan problemas de interfaces.

Tests End-to-End (cima)

Simulan el comportamiento real del usuario. Los más lentos pero validan flujos completos. Ten pocos pero cubran caminos críticos.

Principios de Testing

- **AAA - Arrange, Act, Assert:** preparar, ejecutar, verificar.
- **Tests independientes:** no depender del orden o estado previo.
- **Tests deterministas:** mismo resultado siempre.
- **Tests rápidos:** si tardan, no se ejecutarán frecuentemente.
- **Un assert por test:** idealmente, verificar una sola cosa.

TDD - Test Driven Development

- **Red:** escribe un test que falle.
- **Green:** escribe el código mínimo para que pase.
- **Refactor:** mejora el código manteniendo tests verdes.
- Repite el ciclo para cada funcionalidad.

Herramientas de Calidad

- **Linters:** detectan problemas de estilo y errores potenciales.
- **Formateadores:** formato consistente automático.
- **Type checkers:** verifican tipos en lenguajes dinámicos.
- **Analizadores estáticos:** bugs sin ejecutar código.

> 🤖 ***IA en acción:*** *La IA puede generar tests unitarios automáticamente, sugerir casos límite que no consideraste, y explicar por qué un test falla. Úsala para aumentar tu cobertura de tests.*

Principios de Diseño de Software

Los **principios de diseño** son heurísticas probadas por décadas de experiencia colectiva. No son reglas absolutas, pero ignorarlos suele tener consecuencias.

Principios SOLID

- **S - Single Responsibility:** una clase, una razón para cambiar.
- **O - Open/Closed:** abierto a extensión, cerrado a modificación.
- **L - Liskov Substitution:** subtipos sustituibles por tipos base.
- **I - Interface Segregation:** interfaces específicas mejor que una grande.
- **D - Dependency Inversion:** depender de abstracciones, no implementaciones.

Otros Principios Clave

- **DRY:** Don't Repeat Yourself - evita duplicación de lógica.
- **KISS:** Keep It Simple - la simplicidad es preferible.
- **YAGNI:** You Aren't Gonna Need It - no implementes lo que no necesitas.
- **Composición > Herencia:** preferir componer a heredar.
- **Separation of Concerns:** cada módulo, un aspecto.
- **Fail Fast:** detectar y reportar errores temprano.

Clean Code

- Nombres descriptivos y pronunciables
- Funciones pequeñas que hacen una cosa
- Comentarios que explican 'por qué', no 'qué'
- Formato consistente
- Sin código muerto ni comentado
- Manejo explícito de errores

Patrones de Diseño

Los **patrones de diseño son soluciones** probadas a problemas recurrentes en el desarrollo de software. No necesitas memorizarlos todos, pero reconocerlos te ayuda a comunicarte con otros desarrolladores y a no reinventar la rueda.

¿Qué son los Patrones de Diseño?

Un patrón de diseño describe un problema común, su solución y cuándo aplicarla. Fueron popularizados por el libro 'Gang of Four' y se clasifican en tres categorías:

- **Creacionales:** cómo crear objetos de forma flexible.
- **Estructurales:** cómo componer clases y objetos.
- **De comportamiento:** cómo interactúan y comunican los objetos.

Patrones Creacionales

Factory / Factory Method

Delega la creación de objetos a un método o clase especializada. Útil cuando no sabes de antemano qué tipo exacto de objeto necesitas crear, o cuando la creación es compleja.

Singleton

Garantiza que una clase tenga solo una instancia global. Útil para recursos compartidos (conexiones de BD, configuración). **Usar con moderación**: puede dificultar testing y crear acoplamiento oculto.

Builder

Construye objetos complejos paso a paso. Útil cuando un objeto tiene muchos parámetros opcionales. Evita los constructores con demasiados argumentos.

Dependency Injection

En lugar de crear sus dependencias, un objeto las recibe desde fuera. Facilita enormemente el testing y reduce el acoplamiento. Fundamental en aplicaciones modernas.

Patrones Estructurales

Adapter

Permite que interfaces incompatibles trabajen juntas. Como un adaptador de enchufe: traduce entre dos interfaces diferentes.

Decorator

Añade funcionalidad a un objeto dinámicamente sin modificar su clase. Alternativa flexible a la herencia para extender el comportamiento.

Facade

Proporciona una interfaz simplificada a un sistema complejo. Oculta la complejidad interna tras una API simple y fácil de usar.

Composite

Trata objetos individuales y composiciones de objetos de manera uniforme. Útil para estructuras jerárquicas como árboles de componentes UI.

Patrones de Comportamiento

Observer / Pub-Sub

Define una dependencia uno-a-muchos: cuando un objeto cambia de estado, todos sus dependientes son notificados. Base de sistemas de eventos y arquitecturas reactivas.

Strategy

Define una familia de algoritmos intercambiables. Permite cambiar el comportamiento de un objeto en runtime sin modificar su código.

Command

Encapsula una petición como un objeto. Permite parametrizar operaciones, encolarlas, y soportar undo/redo.

Iterator

Proporciona una forma de acceder secuencialmente a elementos de una colección sin exponer su representación interna. Base de los bucles for-each.

State

Permite a un objeto cambiar su comportamiento cuando cambia su estado interno. Alternativa limpia a grandes bloques if-else basados en estado.

Patrones Arquitectónicos

- **Repository:** abstrae el acceso a datos tras una interfaz.
- **MVC:** separa modelo, vista y controlador.
- **MVVM:** modelo, vista y viewmodel (binding bidireccional).
- **Service Layer:** capa de lógica de negocio.
- **Event Sourcing:** almacena cambios de estado como eventos.

> 💡 ***Consejo:*** *No fuerces patrones donde no son necesarios. Un patrón es una solución a un problema; si no tienes el problema, el patrón solo añade complejidad. Aprende a reconocer cuándo un problema encaja con un patrón conocido.*

> 🤖 ***IA en acción:*** *La IA es excelente para sugerir qué patrón aplicar a un problema específico, generar implementaciones de patrones en tu lenguaje, y explicar cuándo y por qué usar cada uno. Describe tu problema y pregunta qué patrón aplica.*

Refactoring y Deuda Técnica

Refactoring es mejorar la estructura del código sin cambiar su comportamiento. La **deuda técnica** es el costo acumulado de decisiones rápidas que complican el futuro.

¿Qué es Refactoring?

Cambiar la estructura interna del código mientras mantiene el mismo comportamiento externo:

- Renombrar variables/funciones.
- Extraer funciones de código repetido.
- Simplificar condicionales complejos.
- Reorganizar clases y módulos.

Cuándo Refactorizar

Sí refactorizar

- Antes de añadir nueva funcionalidad.
- Cuando encuentres código difícil de entender.
- Cuando veas duplicación.
- Durante code review.

No refactorizar

- Sin tests que validen que no rompes nada.
- Código que funciona y no necesitas tocar.
- Cuando hay deadline crítico (anótalo para después).
- Todo a la vez (hazlo incrementalmente).

Técnicas Comunes de Refactoring

Extract Method

Extraer un bloque de código a su propia función:

```
None
// Antes
function processOrder(order) {
    // 20 líneas validando
    // 30 líneas calculando total
    // 15 líneas enviando email
}

// Después
function processOrder(order) {
    validateOrder(order);
    const total = calculateTotal(order);
    sendConfirmationEmail(order, total);
}
```

Rename

Cambiar nombres para reflejar mejor la intención:

```
None
// Antes
const d = new Date();
const x = calculateX(a, b);

// Después
const orderDate = new Date();
const shippingCost = calculateShippingCost(weight, distance);
```

Replace Magic Numbers

Usar constantes con nombre en lugar de valores literales:

```
None
// Antes
if (age >= 18) { ... }
if (status === 3) { ... }

// Después
const LEGAL_AGE = 18;
const STATUS_APPROVED = 3;
if (age >= LEGAL_AGE) { ... }
if (status === STATUS_APPROVED) { ... }
```

Simplify Conditionals

```
None
// Antes
if (user !== null && user !== undefined && user.isActive === true) {

// Después
if (user?.isActive) {
```

Deuda Técnica

Qué es

Decisiones de implementación que facilitan el presente pero complican el futuro:

- Código duplicado.
- Falta de tests.
- Arquitectura inadecuada.
- Dependencias desactualizadas.

Cómo gestionarla

1. **Hacerla visible:** mantén una lista de deuda conocida.
2. **Pagar intereses:** dedica tiempo regular a reducirla.
3. **No acumular sin control:** a veces es necesaria, pero conscientemente.
4. **Priorizar:** ataca primero la que más fricción causa.

> 💡 ***Consejo:*** *El mejor momento para refactorizar es antes de añadir código nuevo. "Deja el código mejor de como lo encontraste" (Boy Scout Rule).*

PARTE V: DATOS Y COMUNICACIÓN

Redes Básicas: Cómo Funciona Internet

Entender cómo viajan los datos por la red te ayuda a debuggear problemas, entender errores, y diseñar mejor tus aplicaciones.

El Modelo Cliente-Servidor

- **Cliente:** quien inicia la petición (navegador, app móvil, otro servidor).
- **Servidor:** quien responde a las peticiones.
- **Petición (Request):** mensaje del cliente al servidor.
- **Respuesta (Response):** mensaje del servidor al cliente.

Conceptos Fundamentales

IP y Puertos

- **Dirección IP:** identificador único de un dispositivo en la red (192.168.1.1).
- **Puerto:** "puerta" específica en ese dispositivo (80 para HTTP, 443 para HTTPS).
- **localhost (127.0.0.1):** tu propia máquina.

DNS

El sistema que traduce nombres de dominio a IPs:

```
None
google.com → 142.250.184.46
```

Cuando escribes una URL, primero se consulta el DNS para obtener la IP.

Protocolos de la Capa de Transporte

- **TCP:** confiable, ordenado, con conexión (HTTP, email).
- **UDP:** rápido, sin garantías (video streaming, juegos).

HTTP/HTTPS en Detalle

Estructura de una Petición HTTP

```
None
GET /api/users HTTP/1.1        ← Método, ruta, versión
Host: api.ejemplo.com          ← Headers
Authorization: Bearer token123
Content-Type: application/json

{"name": "Brais"}               ← Body (opcional)
```

Estructura de una Respuesta HTTP

```
None
HTTP/1.1 200 OK                ← Versión, código, mensaje
Content-Type: application/json
Cache-Control: max-age=3600

{"id": 1, "name": "Brais"}      ← Body
```

Códigos de Estado Importantes

- **200 OK:** éxito.
- **201 Created:** recurso creado.
- **400 Bad Request:** petición mal formada.
- **401 Unauthorized:** no autenticado.
- **403 Forbidden:** no autorizado (autenticado pero sin permiso).
- **404 Not Found:** recurso no existe.
- **500 Internal Server Error:** error del servidor.

CORS (Cross-Origin Resource Sharing)

Mecanismo de seguridad del navegador que bloquea peticiones a dominios diferentes:

- Tu web en `miapp.com` quiere llamar a `api.otro.com`
- El navegador bloquea por defecto.
- El servidor debe enviar headers CORS permitiendo el origen.

Errores CORS comunes

```
None
Access to fetch at 'https://api.com' from origin 'https://miapp.com'
has been blocked by CORS policy
```

Solución: configurar el servidor para permitir el origen.

HTTPS y Seguridad

- **HTTP:** texto plano, cualquiera puede leerlo.
- **HTTPS:** cifrado con TLS/SSL.
- **Certificado:** prueba la identidad del servidor.
- **Siempre usa HTTPS** para datos sensibles.

Herramientas de Diagnóstico

- **ping:** verificar conectividad básica.
- **nslookup/dig:** consultar DNS.
- **curl:** hacer peticiones HTTP desde terminal.
- **DevTools > Network:** ver todas las peticiones del navegador.

> 💡 ***Consejo:*** *Cuando algo falla en red, usa las DevTools del navegador (F12 > Network) para ver exactamente qué peticiones se hacen y qué respuestas llegan.*

Bases de Datos

Casi toda aplicación necesita persistir datos. Entender los fundamentos de **bases de datos** es esencial para cualquier desarrollador.

Bases de Datos Relacionales (SQL)

- **Tablas:** estructuras con filas y columnas.
- **Clave primaria:** identificador único de cada fila.
- **Clave foránea:** referencia a otra tabla.
- **Normalización:** organizar para evitar redundancia.
- **Índices:** acelerar búsquedas (con costo de escritura).
- **Transacciones ACID:** operaciones atómicas y consistentes.

SQL Básico

- **SELECT:** leer datos, con WHERE, ORDER BY, GROUP BY.
- **INSERT:** crear nuevos registros.
- **UPDATE:** modificar registros existentes.
- **DELETE:** eliminar registros.
- **JOIN:** combinar datos de múltiples tablas.

Bases de Datos NoSQL

- **Documentales:** documentos JSON flexibles, sin esquema fijo.
- **Clave-valor:** pares simples, muy rápidas, ideal para caché.
- **Columnares:** optimizadas para análisis de grandes volúmenes.
- **Grafos:** especializadas en relaciones entre datos.

Cuándo Usar Cada Una

SQL (Relacional)

- Datos estructurados con relaciones claras.
- Necesitas transacciones ACID.
- Consultas complejas con JOINs.
- La integridad de datos es crítica.

NoSQL

- Esquema flexible o cambiante.
- Escala horizontal masiva.
- Datos no estructurados (documentos, logs).
- Rendimiento sobre consistencia.

> 💡 ***Consejo:*** *SQL es una habilidad fundamental. Aprende a escribir consultas básicas manualmente antes de depender de ORMs o generadores de queries.*

APIs y Comunicación entre Sistemas

Las **APIs** son contratos que permiten a sistemas comunicarse. Son el pegamento del software moderno.

Conceptos HTTP

- **Métodos:** GET (leer), POST (crear), PUT/PATCH (actualizar), DELETE.
- **Códigos de estado:** 2xx éxito, 4xx error cliente, 5xx error servidor.
- **Headers:** metadatos de la petición/respuesta.
- **Body:** datos enviados o recibidos.
- **URL/Endpoint:** dirección del recurso.

Estilos de API

REST

- Recursos identificados por URLs.
- Verbos HTTP para operaciones.
- Sin estado entre peticiones.
- El más común en web.

```
None
GET    /api/users          → listar usuarios
GET    /api/users/123      → obtener usuario 123
POST   /api/users          → crear usuario
PUT    /api/users/123      → actualizar usuario 123
DELETE /api/users/123      → eliminar usuario 123
```

GraphQL

- El cliente especifica exactamente qué datos quiere.
- Un solo endpoint.
- Evita over-fetching y under-fetching.

gRPC

- Alto rendimiento, Protocol Buffers.
- Ideal entre microservicios.
- Tipado fuerte.

Formatos de Datos

- **JSON:** ligero, legible, el estándar.
- **XML:** más verboso, usado en sistemas legacy.
- **Protocol Buffers:** binario, compacto, tipado.

Autenticación y Autorización

- **API Keys:** identificadores simples.
- **OAuth 2.0:** autorización delegada estándar.
- **JWT:** tokens autocontenidos.
- **HTTPS:** siempre cifrar comunicaciones.

Autenticación vs. Autorización

- **Autenticación:** ¿Quién eres? (login).
- **Autorización:** ¿Qué puedes hacer? (permisos).

Seguridad Básica

La **seguridad** no es opcional ni algo que se añade después. Como desarrollador, eres la primera línea de defensa.

Vulnerabilidades Comunes (OWASP Top 10)

- **Inyección SQL:** validar y parametrizar SIEMPRE las consultas.
- **XSS:** escapar todo contenido de usuarios antes de mostrarlo.
- **CSRF:** tokens anti-forgery en formularios.
- **Autenticación rota:** passwords hasheados, sesiones seguras.
- **Exposición de datos:** cifrar datos sensibles, HTTPS obligatorio.

Principios de Seguridad

- **Defensa en profundidad:** múltiples capas de protección.
- **Mínimo privilegio:** solo los permisos necesarios.
- **Validar entrada, escapar salida:** nunca confiar en datos externos.
- **Fail secure:** ante errores, denegar por defecto.
- **Keep it simple:** código complejo = más vulnerabilidades.

Manejo de Contraseñas

- **NUNCA** almacenar en texto plano ni usar MD5 o SHA1 para passwords.
- **SÍ** usar bcrypt, argon2, o PBKDF2.
- **SÍ** usar un salt único por usuario.
- **SÍ** usar autenticación de dos factores cuando sea posible.

Secretos y Configuración

- No commits de credenciales en Git.
- Usar variables de entorno.
- Diferentes credenciales por entorno y rotar credenciales regularmente.

> ⚠️ ***Importante:*** *Asume que toda entrada del usuario es maliciosa. Valida, sanitiza, y escapa siempre.*

PARTE VI: PRÁCTICAS PROFESIONALES

Lectura y Comprensión de Código

Los juniors pasan más tiempo leyendo código que escribiéndolo. Saber navegar y entender código ajeno es una habilidad fundamental.

Por Qué es Importante

- Pasarás ~70% del tiempo leyendo código.
- El código legacy es la norma, no la excepción.
- Entender antes de modificar previene bugs.
- Aprendes de código de otros desarrolladores.

Cómo Abordar un Codebase Nuevo

Vista de pájaro

- Lee el README.
- Mira la estructura de carpetas.
- Identifica el stack tecnológico.
- Busca documentación o diagramas.

Encuentra los puntos de entrada

- **Web:** rutas, controladores.
- **API:** endpoints.
- **CLI:** función main.
- **Tests:** a menudo documentan comportamiento

Sigue el flujo de datos

- ¿Dónde entra el input?
- ¿Qué transformaciones sufre?
- ¿Dónde sale el output?

Identifica los componentes principales

- Modelos/Entidades.
- Servicios/Lógica de negocio.
- Controladores/Handlers.
- Utilidades/Helpers.

Técnicas de Lectura

Lectura en espiral

1. Empieza por la función/clase principal.
2. Anota las funciones que llama (sin entrar todavía).
3. Entiende el flujo general.
4. Profundiza en cada función llamada.
5. Repite recursivamente.

Usa el debugger para entender

- Pon breakpoints y ejecuta.
- Observa valores reales de variables.
- Sigue el flujo de ejecución real.

Toma notas

- Dibuja diagramas de flujo.
- Anota qué hace cada componente.
- Marca las partes que no entiendas para volver después.

Señales para Identificar Código Importante

- Archivos grandes o muy modificados (git log).
- Código con muchos tests.
- Código referenciado desde muchos lugares.
- Código mencionado en la documentación.

Lidiar con Código Malo

El código legacy suele ser confuso. Estrategias:

- No juzgues: había contexto que no conoces.
- Añade tests antes de modificar.
- Refactoriza incrementalmente.
- Documenta lo que descubras para el siguiente.

> 💡 ***Consejo:*** *Antes de preguntar "qué hace este código", intenta leerlo tú mismo durante 30 minutos. Llegarás con mejores preguntas.*

> 🤖 ***IA en acción:*** *Pega fragmentos de código y pide explicaciones. "¿Qué hace esta función?" "¿Por qué se usa este patrón aquí?" La IA es excelente para esto.*

Code Review

El **code review** es el proceso de revisar código de otros antes de integrarlo. Es una de las prácticas más valiosas para la calidad y el aprendizaje.

Por qué hacemos Code Review

- Detectar bugs antes de que lleguen a producción.
- Compartir conocimiento en el equipo.
- Mantener consistencia en el código.
- Mejorar como desarrolladores (dando y recibiendo).

Cómo hacer una buena Review

Preparación

- Entiende el contexto (lee la descripción del PR/MR).
- Conoce los requisitos o la issue relacionada.
- Revisa sin prisa (es una tarea importante).

Qué Buscar

- **Correctitud:** ¿Funciona? ¿Cubre casos límite?
- **Claridad:** ¿Se entiende sin explicación?
- **Mantenibilidad:** ¿Será fácil modificar después?
- **Tests:** ¿Hay tests? ¿Cubren lo importante?
- **Seguridad:** ¿Hay vulnerabilidades obvias?
- **Rendimiento:** ¿Hay problemas evidentes?

Cómo dar Feedback

- **Sé específico:** "Esta función hace demasiado" vs. "Considera extraer las líneas 15-30 a una función `validateInput()`"
- **Explica el porqué:** No solo qué cambiar, sino por qué.
- **Distingue crítico de sugerencia:** "Bloqueante: esto causa bug" vs. "Sugerencia: podrías usar X"
- **Sé amable:** Hay una persona al otro lado.
- **Reconoce lo bueno:** No todo es criticar.

Ejemplos de Comentarios

Malo:

> "Esto está mal"

Bueno:

> "Este bucle podría causar un problema de rendimiento si `users` tiene miles de elementos. ¿Consideraste paginar o usar `users.find()` en lugar de `users.filter()[0]`?"

Malo:

> "No me gusta este nombre"

Bueno:

> "El nombre `data` es muy genérico. ¿Qué te parece `userProfile` para que sea más claro qué contiene?"

Cómo Recibir Feedback

- **No lo tomes personal:** El código no eres tú.
- **Agradece los comentarios:** Alguien dedicó tiempo a ayudarte.
- **Pregunta si no entiendes:** "¿Puedes explicarme por qué es mejor así?"
- **No discutas por discutir:** A veces es cuestión de gustos.
- **Aprende de cada review:** Es una oportunidad de mejora.

Etiqueta de Code Review

- Responde a todos los comentarios (aunque sea "Hecho").
- Haz reviews pequeños frecuentes (no PRs de 2000 líneas).
- No dejes reviews pendientes mucho tiempo.
- Si el PR es grande, ofrece una llamada para discutirlo.
- Usa herramientas de sugerencia cuando tu editor lo permita.

> 💡 ***Consejo:*** *Como junior, pide que revisen tu código siempre. Como reviewer, recuerda cómo te sentías cuando empezabas.*

Logging y Observabilidad

El **logging** es lo que te permite entender qué pasó cuando algo falla en producción. Sin logs, estás ciego.

Por Qué es Importante

- En producción no puedes poner breakpoints.
- Los usuarios no saben explicar bugs técnicamente.
- Los errores intermitentes son difíciles de reproducir.
- Los logs son tu caja negra cuando algo explota.

Niveles de Log

En orden de severidad:

- **DEBUG:** Información detallada para desarrollo. No en producción.
- **INFO:** Eventos normales significativos. "Usuario X inició sesión".
- **WARN:** Algo inesperado pero no crítico. "Reintentando conexión".
- **ERROR:** Algo falló y necesita atención. "No se pudo guardar pedido".
- **FATAL:** La aplicación no puede continuar. "No hay conexión a BD".

Qué Loguear

Sí loguear

- Inicio y fin de operaciones importantes.
- Errores con contexto suficiente.
- Decisiones de negocio importantes.
- Métricas de rendimiento (tiempo de respuesta).
- Acciones de seguridad (login, cambios de permisos).

No loguear

- Datos sensibles (contraseñas, tokens, tarjetas).
- Información personal innecesaria.
- Cada línea de código (genera ruido).
- Datos que violan privacidad (GDPR).

Cómo Loguear Bien

Incluye contexto

```
None
// Malo
log.error("Error");

// Bueno
log.error("No se pudo procesar pedido", {
    orderId: order.id,
    userId: user.id,
    error: err.message
});
```

Usa IDs de correlación

Para trazar una petición a través de múltiples servicios:

```
None
[req-abc123] Recibida petición POST /orders
[req-abc123] Validando datos del pedido
[req-abc123] Guardando en base de datos
[req-abc123] Petición completada en 234ms
```

Logs estructurados

Usa JSON para que sean procesables por herramientas:

```
JSON
{"level":"error","timestamp":"2024-01-15T10:30:00Z","message":"DB connection failed","retry":3}
```

Observabilidad Más allá de Logs

- **Métricas:** Números que puedes agregar (requests/segundo, latencia).
- **Trazas:** Seguimiento de una petición a través del sistema.
- **Alertas:** Notificaciones cuando algo está mal.

> 💡 ***Consejo:*** *Cuando debuggees un problema, piensa "¿qué log me habría ayudado a encontrar esto más rápido?" y añádelo para la próxima vez.*

Entornos y Configuración

Entender los diferentes entornos y cómo configurarlos es crucial para no romper producción.

Los Tres Entornos típicos

Development (Dev)

- Tu máquina local.
- Datos de prueba.
- Errores detallados.
- Hot reload.

Staging/QA

- Réplica de producción.
- Datos similares a producción (pero no reales).
- Para testing final antes de deploy.

Production (Prod)

- El entorno real con usuarios reales.
- Datos reales.
- Errores genéricos (sin exponer detalles).
- Monitorización activa.

Variables de Entorno

Configuración que cambia entre entornos:

```
None
# .env.development
DATABASE_URL=localhost:5432/myapp_dev
DEBUG=true
API_KEY=dev-key-not-secret
```

```
# .env.production
DATABASE_URL=prod-server:5432/myapp
DEBUG=false
API_KEY=real-secret-key
```

Buenas Prácticas

- Nunca commitear `.env` con secretos reales.
- Usar `.env.example` con valores de ejemplo.
- Diferentes credenciales por entorno.
- Validar que las variables requeridas existan al iniciar.

Reglas de Oro

1. **Nunca testear en producción** (a menos que sea feature flags controlados).
2. **Los datos de producción son sagrados** - no los modifiques manualmente.
3. **Lo que funciona en dev puede fallar en prod** - prueba en staging.
4. **Las credenciales de prod solo las tiene quien las necesita.**

Feature Flags

Interruptores para activar/desactivar funcionalidades sin deploy:

- Lanzar features gradualmente
- A/B testing
- Desactivar rápidamente algo que falla

```
None
if (featureFlags.isEnabled('newCheckout')) {
    renderNewCheckout();
} else {
    renderOldCheckout();
}
```

> ⚠️ ***Importante:*** *Si no estás seguro de si algo afecta a producción, pregunta. Es mejor preguntar antes que arreglar después.*

Arquitectura Básica de Aplicaciones

La **arquitectura** define la estructura de alto nivel de tu aplicación: cómo se organizan los componentes y cómo se comunican.

Patrones Arquitectónicos

MVC - Model View Controller

Separa datos (Model), presentación (View) y lógica de control (Controller). Clásico en aplicaciones web. Facilita el mantenimiento y testing.

Capas (Layered Architecture)

Organiza en capas horizontales: presentación, negocio, datos. Cada capa solo habla con la inferior. Simple y muy común.

Clean Architecture / Hexagonal

El dominio en el centro, independiente de frameworks y BD. Más compleja pero muy testeable.

Monolito vs. Microservicios

Monolito

- Todo en una unidad desplegable.
- Simple de empezar y debuggear.
- **Recomendado para empezar.**

Microservicios

- Múltiples servicios independientes.
- Escala mejor, equipos independientes.
- Mucha más complejidad operacional.
- **No para proyectos pequeños ni juniors solos.**

> 💡 ***Consejo:*** *Empieza siempre con un monolito bien estructurado. Puedes extraer microservicios después si realmente lo necesitas.*

Capas Típicas

Presentación	UI, API endpoints
Aplicación/Servicios	Casos de uso, lógica de negocio
Dominio	Entidades, reglas de negocio
Infraestructura	BD, APIs externas, email

Patrones de Diseño Comunes

- **Factory:** crear objetos sin especificar clase exacta.
- **Singleton:** una única instancia global (usar con cuidado).
- **Repository:** abstraer acceso a datos.
- **Strategy:** algoritmos intercambiables.
- **Observer:** notificar cambios a interesados.

> 💡 ***Consejo:*** *No necesitas conocer todos los patrones. Aprende los más comunes y reconoce cuándo un problema se parece a uno que ya tiene solución conocida.*

PARTE VII: EL DESARROLLADOR MODERNO

Fundamentos de IA para Desarrolladores

Entender cómo funciona la IA que usas te hace un usuario más efectivo y te prepara para integrarla en tus aplicaciones. No necesitas ser experto en machine learning, pero sí conocer los conceptos fundamentales.

Modelos de Lenguaje (LLMs)

Qué son

Los **Large Language Models** son redes neuronales entrenadas con enormes cantidades de texto para predecir la siguiente palabra. Esa simple tarea, a escala masiva, produce capacidades emergentes sorprendentes.

Conceptos clave

Tokens

- La unidad mínima que procesa el modelo (no siempre es una palabra).
- "Hola mundo" → ["Hola", " mundo"] (2 tokens).
- "programación" → ["program", "ación"] (2 tokens).
- Los modelos tienen límite de tokens por petición (contexto).

Ventana de contexto

- Cantidad máxima de tokens que el modelo puede "recordar".
- Incluye tu prompt + la respuesta.
- Modelos actuales: 8K, 32K, 100K, 200K+ tokens.
- Más contexto = más información disponible, pero más costo.

Temperatura

- Controla la "creatividad" vs. "determinismo".
- Temperatura 0: respuestas más predecibles y consistentes.
- Temperatura alta (0.7-1): respuestas más variadas y creativas.
- Para código: temperatura baja. Para escritura creativa: temperatura alta.

System prompt

- Instrucciones iniciales que definen el comportamiento del modelo.
- "Eres un experto en Python que responde de forma concisa".
- Establece el contexto, tono, restricciones y formato.

Tipos de Modelos

Por capacidad

- **Modelos base:** entrenados solo para predecir texto.
- **Modelos instruction-tuned:** ajustados para seguir instrucciones.
- **Modelos chat:** optimizados para conversación multi-turno.
- **Modelos de código:** especializados en programación.

Por acceso

- **Propietarios/API:** Claude, GPT, Gemini - acceso vía API de pago.
- **Open source:** Llama, Mistral, CodeLlama - puedes ejecutarlos localmente.
- **Open weights:** pesos públicos pero con restricciones de uso.

Por modalidad

- **Solo texto:** entrada y salida de texto.
- **Multimodal:** pueden procesar imágenes, audio, video.
- **Generación de imágenes:** DALL-E, Midjourney, Stable Diffusion.

Prompt Engineering

El arte de comunicarte efectivamente con modelos de IA:

Técnicas fundamentales

Zero-shot Pedir directamente sin ejemplos:

```
None
Clasifica este texto como positivo o negativo: "El producto llegó roto"
```

Few-shot Dar ejemplos antes de la tarea:

```
None
Clasifica estos textos:
"Me encanta" → positivo
"Es horrible" → negativo
"El producto llegó roto" → ?
```

Chain of Thought (CoT) Pedir que razone paso a paso:

```
None
Resuelve este problema paso a paso, mostrando tu razonamiento:
Si tengo 3 manzanas y compro 2 bolsas de 4 manzanas cada una...
```

Role prompting Asignar un rol específico:

```
None
Actúa como un arquitecto de software senior revisando código de un junior.
```

Estructura de un buen prompt

1. **Contexto:** información de fondo necesaria.
2. **Instrucción:** qué quieres que haga.
3. **Input:** los datos a procesar.
4. **Formato de salida:** cómo quieres la respuesta.
5. **Ejemplos:** si aplica (few-shot).

Agentes de IA

Qué son

Sistemas donde un LLM puede tomar acciones, usar herramientas, y trabajar de forma autónoma para completar tareas complejas.

Componentes de un agente

- **LLM:** el "cerebro" que razona y decide.
- **Herramientas:** funciones que el agente puede invocar (buscar, calcular, ejecutar código).
- **Memoria:** historial de acciones y resultados.
- **Loop de razonamiento:** observar → pensar → actuar → repetir.

Patrones comunes

ReAct (Reasoning + Acting)

```
None
Pensamiento: Necesito buscar información sobre X
Acción: buscar("X")
Observación: [resultados]
Pensamiento: Ahora puedo responder...
Respuesta: ...
```

Tool use / Function calling El modelo decide cuándo llamar a funciones externas:

```
JSON
{
  "function": "get_weather",
  "arguments": {"city": "Madrid"}
}
```

Ejemplos de agentes

- **Asistentes de código:** pueden leer archivos, ejecutar comandos, modificar código
- **Agentes de investigación:** buscan, leen, sintetizan información
- **Agentes de automatización:** realizan tareas en aplicaciones

Reglas y Configuración para Agentes (AGENTS.md)

Qué son las Reglas de Agente

Archivos de configuración que instruyen a los agentes de IA sobre cómo comportarse en tu proyecto. Son como un "onboarding" automático para la IA.

Archivos comunes

Archivo	Herramienta	Propósito
`AGENTS.md`	Cursor, Claude	Instrucciones generales del proyecto
`.cursorrules`	Cursor	Reglas específicas de Cursor
`.claude`	Claude Code	Configuración para Claude
`CLAUDE.md`	Claude Code	Instrucciones en la raíz del proyecto
`.github/copilot-instructions.md`	GitHub Copilot	Reglas para Copilot
`rules/`	Varios	Carpeta con múltiples archivos de reglas

Por qué son importantes

- **Consistencia:** todos los desarrolladores (y la IA) siguen las mismas convenciones.
- **Contexto:** la IA entiende tu stack, arquitectura y preferencias.
- **Productividad:** menos correcciones manuales del código generado.
- **Onboarding:** nuevos miembros del equipo (humanos o IA) entienden el proyecto rápido.

Qué incluir en un AGENTS.md

```
None
# Instrucciones para Agentes de IA

## Sobre el Proyecto
- Descripción breve del proyecto
- Stack tecnológico (lenguaje, frameworks, BD)
- Arquitectura general

## Convenciones de Código
- Estilo de nombrado (camelCase, snake_case)
- Estructura de archivos y carpetas
- Patrones preferidos (repository, service layer, etc.)

## Reglas Específicas
- Siempre usar TypeScript strict mode
- Preferir composición sobre herencia
- Los componentes van en /src/components
- Las queries SQL van en /src/repositories

## Qué NO hacer
- No usar any en TypeScript
- No hacer commits directos a main
- No hardcodear credenciales

## Testing
- Framework de testing usado
- Convenciones de nombrado de tests
- Cobertura mínima esperada

## Dependencias
- Librerías preferidas para X tarea
- Librerías prohibidas o a evitar
```

Ejemplo real simplificado

```
None
# AGENTS.md - Proyecto E-commerce

## Stack
- Backend: Node.js + Express + TypeScript
- BD: PostgreSQL con Prisma ORM
- Frontend: React + TailwindCSS
- Testing: Jest + React Testing Library

## Convenciones
- Nombres de archivos: kebab-case (user-service.ts)
- Nombres de funciones: camelCase
- Nombres de tipos/interfaces: PascalCase
- Indentación: 2 espacios

## Arquitectura
/src
  /controllers  → Manejan HTTP requests
  /services     → Lógica de negocio
  /repositories → Acceso a datos
  /models       → Tipos y validaciones
  /utils        → Helpers puros

## Reglas
- Toda función pública debe tener JSDoc
- Los errores se manejan con clases custom en /src/errors
- Las validaciones usan Zod
- NO usar console.log, usar el logger en /src/utils/logger

## Comandos útiles
- npm run dev → desarrollo
- npm run test → tests
- npm run lint → linting
```

Reglas por carpeta

Algunos proyectos usan reglas específicas por módulo:

```
None
/src
  /auth
    AGENTS.md         → Reglas específicas de autenticación
  /payments
    AGENTS.md         → Reglas específicas de pagos (PCI compliance, etc.)
  /api
    AGENTS.md         → Convenciones de endpoints
```

Buenas prácticas

- **Mantenlo actualizado:** reglas desactualizadas confunden más que ayudan.
- **Sé específico:** "usa funciones pequeñas" es vago; "funciones de máximo 20 líneas" es claro.
- **Incluye ejemplos:** un ejemplo vale más que mil explicaciones.
- **Versiona con el código:** el AGENTS.md va en Git con el proyecto.
- **Revísalo en PRs:** si cambian convenciones, actualiza las reglas.

El futuro de la configuración de agentes

- **Estandarización:** MCP y otros protocolos pueden unificar formatos.
- **Validación automática:** herramientas que verifican que el código cumple las reglas.
- **Generación asistida:** IA que ayuda a crear las reglas basándose en el código existente.
- **Reglas dinámicas:** contexto que se ajusta según el archivo que estás editando.

> 💡 ***Consejo:*** *Empieza simple. Un AGENTS.md básico de 20 líneas es mejor que ninguno. Itera según las necesidades del equipo.*

MCP (Model Context Protocol)

Qué es

MCP es un protocolo estándar abierto para conectar modelos de IA con fuentes de datos y herramientas externas. Piensa en él como "USB para IA" - una interfaz universal.

Por qué importa

- **Antes de MCP:** cada integración era custom, frágil, incompatible.
- **Con MCP:** un estándar que cualquier herramienta puede implementar.

Componentes de MCP

Servers (servidores MCP) Exponen datos y herramientas:

- Servidor de archivos: lee/escribe archivos locales.
- Servidor de base de datos: ejecuta queries.
- Servidor de API: conecta con servicios externos.
- Servidor de Git: operaciones de repositorio.

Clients (clientes MCP) Aplicaciones que se conectan a servidores MCP:

- IDEs con IA integrada.
- Chatbots.
- Agentes autónomos.

Recursos Datos que el servidor expone (archivos, registros BD, etc.)

Herramientas Acciones que el servidor permite ejecutar

Ejemplo de flujo MCP

```
None
Usuario: "Resume los últimos commits de mi proyecto"
    ↓
Cliente (Claude) detecta que necesita Git
    ↓
Llama al servidor MCP de Git
    ↓
Servidor ejecuta: git log --oneline -10
    ↓
Retorna resultados al cliente
    ↓
Claude genera el resumen
```

Por qué aprenderlo

- Es el estándar emergente para integración de IA.
- Te permite crear herramientas que cualquier IA puede usar.
- Entenderlo te da ventaja en el mercado laboral 2026+.

RAG (Retrieval Augmented Generation)

El problema

Los LLMs tienen conocimiento limitado a su entrenamiento. No conocen tus documentos privados ni información actualizada.

La solución: RAG

Combinar búsqueda de información con generación:

1. **Indexar:** convertir documentos en embeddings (vectores).
2. **Buscar:** encontrar fragmentos relevantes para la pregunta.
3. **Generar:** dar esos fragmentos al LLM como contexto.

Flujo de RAG

```
None
Usuario: "¿Cuál es nuestra política de vacaciones?"
    ↓
1. Convertir pregunta a embedding
2. Buscar en base de datos vectorial
3. Recuperar fragmentos relevantes del manual de empleados
4. Prompt: "Usando este contexto: [fragmentos], responde: [pregunta]"
    ↓
Respuesta basada en TUS documentos
```

Conceptos clave

Embeddings

- Representación numérica del significado del texto.
- Textos similares → vectores cercanos.
- Permiten búsqueda semántica (por significado, no keywords).

Base de datos vectorial

- Almacena y busca embeddings eficientemente.
- Ejemplos: Pinecone, Weaviate, Chroma, pgvector.

Casos de uso

- Chatbots sobre tu documentación.
- Búsqueda en bases de conocimiento.
- Asistentes con información privada/actualizada.

APIs de IA: Integración Básica

Anatomía de una llamada a API de LLM

```python
# Ejemplo conceptual (pseudocódigo)
response = llm_api.chat(
    model="modelo",
    messages=[
        {"role": "system", "content": "Eres un asistente útil"},
        {"role": "user", "content": "Explica qué es una API"}
    ],
    max_tokens=500,
    temperature=0.7
)

print(response.content)
```

Parámetros comunes

- **model:** qué modelo usar.
- **messages:** historial de conversación.
- **max_tokens:** límite de longitud de respuesta.
- **temperature:** creatividad (0-1).
- **stop:** secuencias donde parar de generar.

Consideraciones de integración

- **Costo:** se paga por tokens (entrada + salida).
- **Latencia:** las respuestas tardan segundos.
- **Rate limits:** límites de peticiones por minuto.
- **Streaming:** recibir respuesta token a token.
- **Manejo de errores:** timeouts, límites, errores de API.

Fine-tuning vs. Prompting

Prompting (lo que harás 99% del tiempo)

- Ajustar comportamiento mediante instrucciones.
- Rápido, barato, flexible.
- Limitado por la ventana de contexto.
- **Úsalo para:** la mayoría de casos.

Fine-tuning

- Reentrenar el modelo con tus datos.
- Caro, lento, requiere expertise.
- El modelo "aprende" nuevo conocimiento.
- **Úsalo para:** casos muy específicos con muchos datos.

Alternativas intermedias

- **Few-shot learning:** ejemplos en el prompt.
- **RAG:** información externa en contexto.
- **System prompts:** instrucciones persistentes.

> 💡 ***Consejo:*** *Agota las posibilidades de prompting y RAG antes de considerar fine-tuning.*

Limitaciones y Consideraciones

Limitaciones técnicas

- **Alucinaciones:** inventan información con confianza.
- **Conocimiento desactualizado:** no saben eventos recientes.
- **Razonamiento limitado:** fallan en lógica compleja.
- **Contexto finito:** olvidan en conversaciones largas.

Consideraciones éticas

- **Sesgo:** reflejan sesgos de los datos de entrenamiento.

- **Privacidad:** cuidado con qué datos envías a APIs externas.
- **Atribución:** el contenido generado no es "tuyo".
- **Dependencia:** no pierdas tus habilidades fundamentales.

Seguridad

- **Prompt injection:** usuarios maliciosos manipulando el comportamiento.
- **Data leakage:** el modelo revelando información del prompt.
- **Jailbreaking:** técnicas para evadir restricciones.

> ⚠️ ***Importante:*** *Nunca confíes ciegamente en la salida de un LLM. Verifica información crítica, especialmente código que afecte seguridad o datos.*

El Ecosistema actual

Tendencias a observar

- **Modelos más pequeños y eficientes:** correr localmente.
- **Multimodalidad:** texto + imagen + audio + video.
- **Agentes más autónomos:** tareas complejas sin supervisión.
- **Especialización:** modelos específicos por dominio.
- **Estandarización:** protocolos como MCP ganando adopción.

Habilidades valiosas

- Prompt engineering efectivo.
- Integración de APIs de IA.
- Diseño de sistemas con IA (cuándo sí, cuándo no).
- Evaluación de calidad de outputs.
- Entender limitaciones y riesgos.

> 🤖 ***IA en acción:*** *Este capítulo te da el vocabulario y conceptos para entender las herramientas que usarás a diario. No necesitas ser experto en ML, pero sí entender qué hay "bajo el capó" para usarlas mejor.*

DevOps y CI/CD Conceptos

DevOps es la cultura y prácticas que unifican desarrollo y operaciones. CI/CD automatiza el proceso desde el código hasta producción.

Integración Continua (CI)

Integrar código frecuentemente en un repositorio compartido, con verificación automática mediante builds y tests:

- Commits frecuentes al repositorio principal.
- Build automático en cada commit.
- Tests automáticos que deben pasar.
- Feedback rápido si algo falla.
- El código siempre debe estar en estado desplegable.

Entrega/Despliegue Continuo (CD)

- **Entrega continua:** código siempre listo para desplegar (manual).
- **Despliegue continuo:** despliegue automático a producción.
- Pipelines que automatizan: build → test → deploy.
- Rollback automatizado si algo falla.

Contenedores (Concepto)

Los contenedores empaquetan aplicación + dependencias en una unidad portable. 'Funciona en mi máquina' ya no es excusa. Garantizan que el entorno de desarrollo sea idéntico al de producción.

Infraestructura como Código

Definir infraestructura (servidores, redes, bases de datos) en archivos de configuración versionados. Permite reproducir entornos exactos.

💡 ***Consejo:*** *Como junior, no necesitas dominar DevOps, pero entender estos conceptos te ayuda a colaborar mejor con el equipo y entender el ciclo de vida del software.*

IA como Herramienta de Aprendizaje

Una de las aplicaciones más poderosas de la IA para un junior es usarla como tutor personal disponible 24/7. Bien utilizada, puede acelerar dramáticamente tu aprendizaje.

La IA como Tutor Personal

A diferencia de tutoriales estáticos, la IA puede adaptarse a tu nivel, responder tus preguntas específicas, y explicar conceptos de múltiples formas hasta que los entiendas:

- **Explicaciones adaptadas:** 'explícame recursividad como si tuviera 10 años'.
- **Múltiples perspectivas:** 'explícalo de otra forma' o 'usa una analogía'.
- **Preguntas sin vergüenza:** puedes preguntar lo más básico sin juzgar.
- **Ritmo personalizado:** profundiza donde necesites, avanza donde domines.
- **Disponibilidad:** aprende a las 3 AM si es cuando tienes tiempo.

> 💡 ***Consejo:*** *Utiliza la IA para poner en práctica el método Socrático.*

Técnicas de Estudio con IA

- **Explica código desconocido:** pega código y pregunta qué hace línea por línea.
- **Genera ejercicios:** 'dame 5 ejercicios de dificultad creciente sobre arrays'.
- **Verifica tu comprensión:** explica un concepto y pide que te corrija.
- **Crea flashcards:** pide que genere preguntas y respuestas para repasar.
- **Simula entrevistas:** practica preguntas técnicas con feedback.
- **Compara tecnologías:** '¿cuáles son las diferencias entre X e Y?'
- **Roadmap personalizado:** 'tengo X conocimientos, ¿qué debería aprender después?'

Aprender de Errores con IA

Los errores son oportunidades de aprendizaje. La IA puede ayudarte a entenderlos profundamente:

- Pega el error completo y pregunta qué significa.
- Pregunta no solo cómo solucionarlo, sino POR QUÉ ocurre.
- Pide que te explique cómo prevenir errores similares.
- Solicita ejemplos de otros errores relacionados.
- Pregunta cuáles son las causas más comunes de ese error.

Explorar Documentación y Código

- **Resumir documentación:** 'resume los conceptos clave de esta API'.
- **Entender código open source:** 'explica qué hace este componente'.
- **Comparar enfoques:** '¿cuál de estas dos soluciones es mejor y por qué?'
- **Traducir entre lenguajes:** 'muéstrame cómo sería este código en otro lenguaje'.

> 💡 ***Consejo:*** *El mejor uso de la IA para aprender no es pedir respuestas, sino pedir explicaciones. 'Cómo hago X' te da código. 'Por qué funciona X' te da conocimiento.*

Limitaciones para el Aprendizaje

- **No reemplaza la práctica:** entender no es lo mismo que saber hacer.
- **Puede tener errores:** verifica información importante con otras fuentes.
- **No sustituye la comunidad:** los humanos aportan contexto y experiencia.
- **Riesgo de pasividad:** no solo leas, implementa lo que aprendes.
- **Conocimiento superficial:** profundiza más allá de lo que la IA explica.

> 🤖 ***IA en acción:*** *Usa la IA como punto de partida, no como destino final. Que te explique un concepto, luego practica. Que te sugiera recursos, luego estúdialos. La IA acelera, pero tú haces el trabajo.*

IA como Herramienta de Desarrollo

Esta es la habilidad diferenciadora del desarrollador de 2026. La IA no es una moda pasajera: ha cambiado fundamentalmente cómo trabajamos.

Tipos de Herramientas de IA

- **Autocompletado de código:** sugiere código mientras escribes.
- **Chat/Asistente:** responde preguntas, explica código, debugging.
- **Generación de código:** crea funciones o archivos completos.
- **Revisión de código:** detecta bugs, sugiere mejoras.
- **Generación de tests:** crea casos de prueba automáticamente.
- **Documentación:** genera docs y comentarios.
- **Traducción de código:** convierte entre lenguajes.

Prompt Engineering para Desarrolladores

La calidad de lo que obtienes de la IA depende de cómo le pidas las cosas:

- **Sé específico:** lenguaje, contexto, restricciones, formato esperado.
- **Da ejemplos:** muestra input/output esperado.
- **Divide tareas grandes:** mejor varios prompts pequeños.
- **Especifica el rol:** 'actúa como experto en seguridad'.
- **Pide explicaciones:** no solo código, también el razonamiento.
- **Itera:** refina el prompt según los resultados.

Flujo de Trabajo con IA

- **Entender el problema:** tú defines qué hay que hacer.
- **Generar solución:** la IA propone código o enfoque.
- **Revisar críticamente:** verificar lógica, errores, seguridad.
- **Testear:** el código generado debe pasar tus tests.
- **Refinar:** iterar hasta que sea correcto y limpio.
- **Documentar:** explica decisiones para tu yo futuro.

Casos de Uso Productivos

- Generar boilerplate y código repetitivo.
- Escribir tests unitarios y casos límite.
- Explicar código legacy o desconocido.
- Debugging: 'este código da error X, ¿por qué?'
- Revisar código buscando mejoras.
- Generar documentación y docstrings.
- Refactorizar código existente.
- Convertir código entre lenguajes.

Limitaciones y Precauciones

- **Alucinaciones:** la IA inventa código que parece correcto pero no lo es.
- **Contexto limitado:** no conoce tu proyecto completo.
- **Código inseguro:** puede generar vulnerabilidades.
- **Conocimiento desactualizado:** no conoce APIs/versiones recientes.
- **Dependencia excesiva:** no dejes de desarrollar tus habilidades.
- **Privacidad:** cuidado con código propietario en herramientas externas.

> ⚠️ ***Importante:*** *NUNCA asumas que el código generado por IA es correcto o seguro. Revísalo siempre como revisarías código de un junior: con atención y escepticismo constructivo.*

El Futuro: Desarrollador + IA

El desarrollador del futuro no es quien escribe más líneas de código, sino quien mejor sabe definir problemas, guiar a la IA, evaluar soluciones y tomar decisiones de diseño. Los fundamentos que aprendes en este roadmap son precisamente lo que te permite hacer esto.

> 🤖 ***IA en acción:*** *La IA amplifica tus habilidades. Un desarrollador con fundamentos sólidos + dominio de IA es mucho más productivo que uno que solo tiene uno de los dos. Invierte en ambos.*

Soft Skills y Carrera Profesional

Las habilidades técnicas te consiguen entrevistas; las soft skills, que ya deberían llamarse power skills, determinan tu crecimiento profesional. El desarrollo es fundamentalmente una actividad de equipo.

Comunicación

- Explica conceptos técnicos a diferentes audiencias.
- Documenta tu código y decisiones.
- Escucha activamente antes de responder.
- Escribe mensajes claros y estructurados.
- Participa constructivamente en code reviews.

Trabajo en Equipo

- Comparte conocimiento proactivamente.
- Acepta y da feedback constructivo.
- Cumple compromisos, comunica problemas temprano.
- Respeta diferentes perspectivas y enfoques.
- Colabora en lugar de competir.

Resolución de Problemas

- Descompón problemas grandes en partes.
- Investiga antes de preguntar, pero pregunta cuando estés bloqueado.
- Documenta lo que has intentado.
- Aprende a debuggear sistemáticamente.
- Reconoce cuando algo es 'suficientemente bueno'.

Aprendizaje Continuo

- Dedica tiempo regular a aprender.
- Lee código de proyectos open source.

- Construye proyectos personales para experimentar.
- Participa en comunidades de desarrolladores.
- Acepta que siempre hay más por aprender.

Gestión del Tiempo

- Estima tareas con margen realista.
- Evita multitasking. El contexto tiene coste.
- Toma descansos. La fatiga reduce la productividad.
- Aprende a decir 'no' o negociar plazos.

Metodologías Ágiles

Las metodologías ágiles son formas de organizar el trabajo en equipo que enfatizan entregas frecuentes, adaptación al cambio y colaboración.

Principios Ágiles

- Individuos e interacciones sobre procesos y herramientas.
- Software funcionando sobre documentación extensiva.
- Colaboración con el cliente sobre negociación contractual.
- Responder al cambio sobre seguir un plan.

Scrum (Conceptos)

- **Sprint:** iteración de 1-4 semanas con objetivo claro.
- **Product Backlog:** lista priorizada de trabajo pendiente.
- **Sprint Planning:** planificar qué se hará en el sprint.
- **Daily Standup:** sincronización diaria breve.
- **Sprint Review:** demostrar lo completado.
- **Retrospectiva:** qué mejorar para el siguiente sprint.

Kanban (Conceptos)

- Visualizar el flujo de trabajo en un tablero.
- Limitar el trabajo en progreso (WIP).
- Mover tareas de izquierda a derecha.
- Identificar y eliminar cuellos de botella.
- Mejora continua sin sprints fijos.

> 💡 ***Consejo:*** *Como junior, tu rol es entender el proceso del equipo, cumplir con las ceremonias, y comunicar impedimentos temprano. Con experiencia podrás aportar mejoras al proceso.*

PARTE VIII: PLAN DE ACCIÓN

Roadmap visual: plan de 12 meses

Este plan estructura el aprendizaje en fases progresivas. Los tiempos son orientativos (debes adaptarlos a tu ritmo). Lo importante es la progresión, no la velocidad.

FASE 1: FUNDAMENTOS DE PROGRAMACIÓN

Meses 1-2

- ✓ Elige un primer lenguaje (cualquier lenguaje popular).
- ✓ Domina variables, tipos, operadores.
- ✓ Practica condicionales y bucles hasta que sean naturales.
- ✓ Entiende funciones: parámetros, retorno, scope.
- ✓ Configura tu entorno: editor/IDE, terminal básico.
- ✓ **IA:** Usa chat de IA para explicarte conceptos y errores.

FASE 2: ESTRUCTURAS DE DATOS Y ALGORITMOS

Meses 3-4

- ✓ Domina arrays, listas, diccionarios, sets.
- ✓ Entiende pilas y colas: cuándo usar cada una.
- ✓ Aprende búsqueda lineal y binaria.
- ✓ Conoce Big O básico: $O(1)$, $O(n)$, $O(n^2)$.
- ✓ Practica problemas de lógica simple.
- ✓ **IA:** Pide explicaciones de algoritmos con ejemplos visuales.

FASE 3: CONTROL DE VERSIONES Y MODULARIDAD

Mes 5

- ✓ Domina comandos básicos de control de versiones.
- ✓ Practica: commit, branch, merge, push, pull.
- ✓ Crea tu primer repositorio público.
- ✓ Aprende a organizar código en módulos.
- ✓ Usa un gestor de dependencias.
- ✓ **IA:** Genera mensajes de commit descriptivos.

FASE 4: PARADIGMAS Y PRINCIPIOS

Meses 6-7

- ✓ Profundiza en POO: clases, herencia, polimorfismo.
- ✓ Explora funcional: funciones puras, map/filter/reduce.
- ✓ Estudia principios SOLID (al menos S, O, D).
- ✓ Practica escribir código limpio.
- ✓ Empieza un proyecto personal aplicando todo.
- ✓ **IA:** Pide revisiones de código y sugerencias de mejora.

FASE 5: TESTING Y BASES DE DATOS

Meses 8-9

- ✓ Escribe tus primeros tests unitarios.
- ✓ Practica TDD en funcionalidades pequeñas.
- ✓ Aprende SQL: SELECT, INSERT, UPDATE, DELETE, JOIN.
- ✓ Diseña esquemas de base de datos simples.
- ✓ Implementa persistencia en tu proyecto.
- ✓ **IA:** Genera tests unitarios y casos límite.

FASE 6: APIs Y SEGURIDAD

Meses 10-11

- ✓ Entiende HTTP: métodos, códigos de estado, headers.
- ✓ Consume APIs públicas en tu proyecto.
- ✓ Crea una API REST básica.
- ✓ Implementa autenticación simple.
- ✓ Estudia OWASP Top 10 y aplica validaciones.
- ✓ **IA:** Audita tu código buscando vulnerabilidades.

FASE 7: CONSOLIDACIÓN Y PREPARACIÓN LABORAL

Mes 12

- ✓ Completa y pule tu proyecto personal (es tu portafolio).
- ✓ Documenta profesionalmente con README.
- ✓ Aprende conceptos básicos de CI/CD.
- ✓ Contribuye a algún proyecto open source.
- ✓ Prepara tu perfil profesional y CV.
- ✓ Practica entrevistas técnicas y soft skills (power skills).
- ✓ **IA:** Simula entrevistas técnicas, prepara explicaciones.

> 💡 ***Consejo:*** *Este roadmap es una guía, no una regla rígida. Adapta los tiempos a tu situación, profundiza donde te interese más, y recuerda: la constancia, paciencia y práctica supera a la intensidad.*

Recursos y Próximos Pasos

Estrategias de Aprendizaje

- **Aprende haciendo:** los proyectos enseñan más que los tutoriales.
- **Enseña lo que aprendes:** escribir o explicar consolida conocimiento.
- **Lee código ajeno:** proyectos open source son tu escuela.
- **Practica deliberadamente:** sal de tu zona de confort.
- **Mantén consistencia:** 1 hora diaria > 7 horas un día.

Construye tu Portafolio

- 2-5 proyectos que demuestren diferentes habilidades.
- README claro en cada proyecto: qué, por qué, cómo.
- Código limpio, con tests, bien organizado.
- Variedad: diferentes tipos de aplicaciones.
- Un proyecto 'estrella' más completo que el resto.

Preparación para Entrevistas

- Practica problemas de código en plataformas de ejercicios.
- Prepara explicaciones claras de tus proyectos.
- Desarrolla la habilidad de pensar en voz alta.
- Investiga las empresas antes de aplicar.
- Prepara preguntas inteligentes para tus entrevistadores.

Mentalidad de Crecimiento

- El síndrome del impostor es normal (la constancia es clave).
- Compárate con tu yo de ayer, no con otros.
- Los errores son oportunidades de aprendizaje.
- Pedir ayuda es una fortaleza.
- Celebra los pequeños logros.
- La curiosidad es tu mejor activo.

¡El momento de empezar es ahora!

El mundo del desarrollo de software está lleno de oportunidades para quienes combinan fundamentos sólidos con las herramientas modernas de IA.

Comienza a dar tus primeros pasos en mi campus de estudiantes **mouredev pro**: **https://mouredev.pro**

(Utiliza el cupón "PRO" para acceder con un 10% de descuento a todas las suscripciones y cursos del campus).

Allí encontrarás cursos de fundamentos para aprender a estudiar programación desde cero y una ruta de estudio estructurada. También contarás con soporte, comunidad, test de conocimientos, mentorías semanales y certificado de finalización.

También puedes encontrar todos mis recursos gratis y seguirme en redes desde: **https://moure.dev**

mouredev pro

Estudia programación y desarrollo de software de manera diferente

mouredev.pro