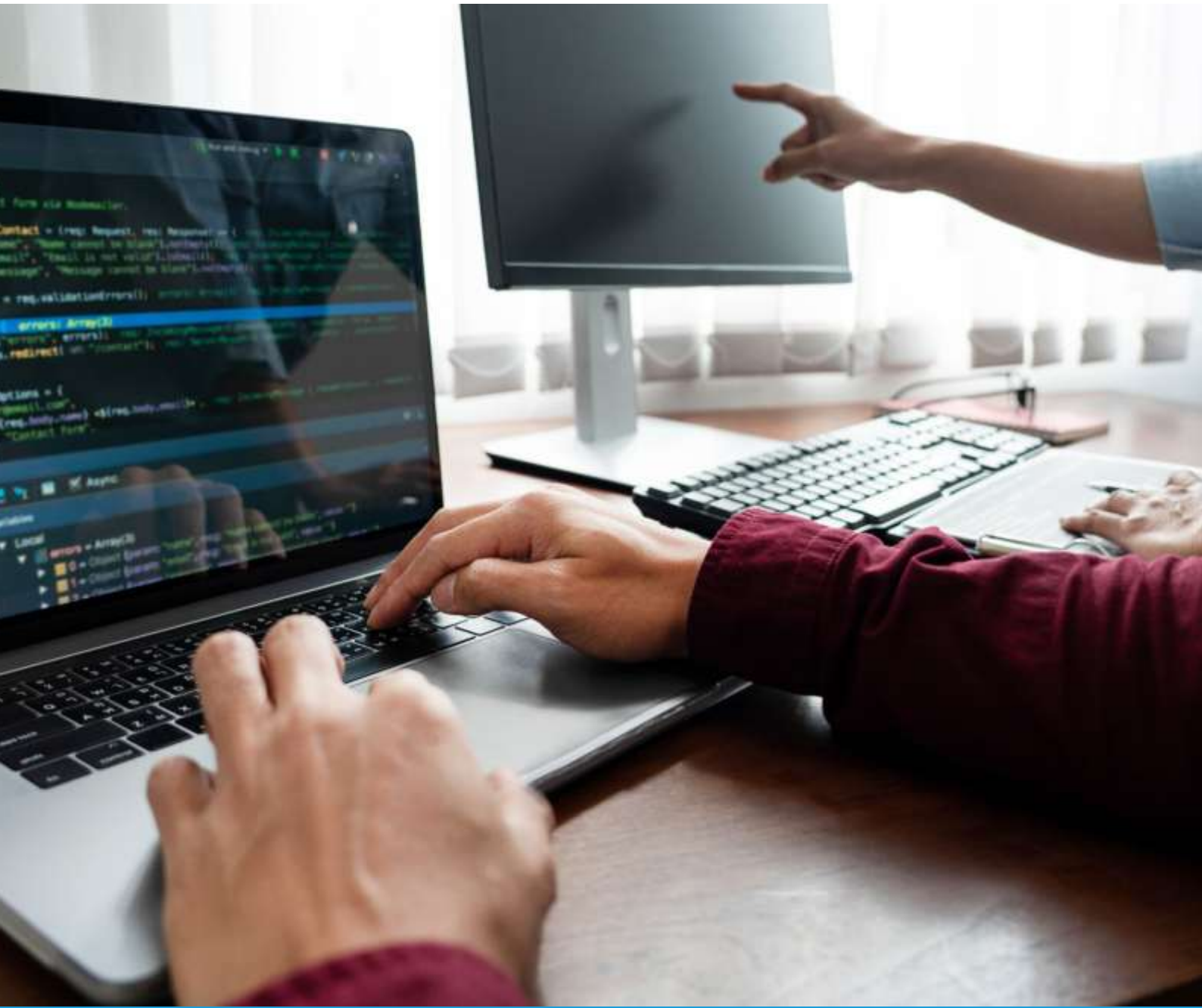
{desafío}
latam_

CURSO

TEST AUTOMATION ENGINEER
FORMACIÓN INTEGRAL



Objetivo General del Curso

ANALIZAR LAS HERRAMIENTAS DE DISEÑO DE TEST AUTOMATION ENGINEER, DE ACUERDO A LAS APLICACIONES WEB, MÓVILES Y APIS.

Objetivo específico del Módulo

DISTINGUIR LOS FUNDAMENTOS DE PROGRAMACIÓN APLICADOS AL TESTING (JAVASCRIPT), DE ACUERDO A LAS APLICACIONES WEB, MÓVILES Y APIS.

Contenidos

Módulo 3: FUNDAMENTOS DE PROGRAMACIÓN APLICADOS AL TESTING (JAVASCRIPT)

Capítulo 1: TIPOS DE DATOS, FUNCIONES, ESTRUCTURAS CONDICIONALES Y REPETITIVAS

El conocimiento de los fundamentos de programación es esencial para comprender y aplicar técnicas de testing de software de manera eficaz. En particular, JavaScript, al ser uno de los lenguajes más utilizados en el desarrollo web, es clave en la validación del comportamiento y la funcionalidad de las aplicaciones.

Comprender cómo se estructuran y funcionan los tipos de datos, las funciones, y las estructuras condicionales y repetitivas permite al tester escribir scripts de prueba, automatizar casos y detectar errores de forma precisa, optimizando así la calidad del software.

Tipos de datos

JavaScript utiliza diferentes tipos de datos para representar y manipular información. Los tipos primitivos más comunes son:

- Number: representa tanto enteros como decimales. Ej: let edad = 30;
- String: cadenas de texto. Ej: let nombre = "María";
- Boolean: valores lógicos true o false. Ej: let esActivo = true;
- Null: representa la ausencia intencional de valor. Ej: let resultado = null;
- Undefined: indica que una variable ha sido declarada pero aún no tiene un valor asignado. Ej: let x;

También existen tipos de objeto, como:

Object: estructura de datos que agrupa pares clave-valor. Ej: let persona = { nombre: "Ana", edad: 28 };

- Array: lista ordenada de elementos. Ej: let numeros = [1, 2, 3];

En el testing, identificar y validar el tipo de dato esperado es crucial para prevenir errores y asegurar el comportamiento correcto del sistema.

Funciones

Las funciones en JavaScript permiten encapsular bloques de código reutilizable. Se utilizan tanto para estructurar la lógica de un programa como para definir los pasos de una prueba automatizada.

Ejemplo básico:

```javascript
function saludar(nombre) {
  return "Hola, " + nombre;
}
```

También pueden definirse como expresiones o funciones flecha:

```javascript
const sumar = (a, b) => a + b;
```

En testing, se suelen usar funciones para definir pruebas unitarias o para agrupar comandos que se repiten, como validaciones o preparación de datos.

Estructuras condicionales

Las estructuras condicionales permiten ejecutar bloques de código según se cumpla o no una condición. Las más comunes son:

```javascript
if (edad >= 18) {
  console.log("Es mayor de edad");
} else {
  console.log("Es menor de edad");
}
```

También se puede utilizar switch para evaluar múltiples valores posibles:

```javascript
switch (estado) {
  case "activo":
    console.log("Usuario activo");
    break;
  case "inactivo":
    console.log("Usuario inactivo");
    break;
  default:
    console.log("Estado desconocido");
}
```

En testing, estas estructuras ayudan a validar distintos escenarios y respuestas del sistema ante diferentes entradas.

Estructuras repetitivas

Las estructuras repetitivas o bucles permiten ejecutar un bloque de código varias veces. Las más usadas son:

for:

```javascript
for (let i = 0; i < 5; i++) {
  console.log("Iteración", i);
}
```

while:

```javascript
let contador = 0;
while (contador < 5) {
  console.log("Contador", contador);
  contador++;
}
```

for...of (para recorrer arrays):

```javascript
let nombres = ["Ana", "Luis", "Carlos"];
for (let nombre of nombres) {
  console.log(nombre);
}
```

Estas estructuras permiten, por ejemplo, ejecutar múltiples casos de prueba sobre diferentes datos o repetir una acción hasta que se cumpla una condición específica.

Capítulo 2: OBJETOS, ARRAYS, DESESTRUCTURACIÓN Y MANEJO DE ERRORES

Dominar estructuras como objetos, arrays, técnicas de desestructuración y el manejo de errores es fundamental en el proceso de testing automatizado.

Estas herramientas permiten organizar la información de manera estructurada, acceder eficientemente a los datos y responder adecuadamente a situaciones imprevistas durante la ejecución de pruebas.

En JavaScript, estos elementos son ampliamente utilizados tanto en la escritura de scripts como en el uso de frameworks de testing, facilitando la lectura, el mantenimiento y la robustez del código de pruebas.

Objetos

En JavaScript, los objetos son estructuras fundamentales que permiten representar entidades del mundo real o estructuras de datos más complejas. Un objeto está compuesto por propiedades, que a su vez consisten en una clave (key) y un valor (value). Esta forma de organización facilita el acceso, modificación y validación de información específica durante las pruebas de software.

Por ejemplo, al testear una API que retorna información de un usuario, es común recibir una respuesta en formato JSON que se transforma directamente en un objeto de JavaScript. Desde ahí, se pueden verificar condiciones como si el campo nombre está presente, si el email tiene un formato válido o si el campo activo es true.

Además, los objetos permiten anidar otras estructuras, como arrays u otros objetos, lo cual es habitual en respuestas más complejas, y requieren una navegación cuidadosa para acceder a los datos necesarios.

En testing automatizado, también se utilizan para definir configuraciones, estados simulados, datos mock y resultados esperados.

Ejemplo:

```javascript
let usuario = {
  nombre: "María",
  edad: 30,
  activo: true
};
```

Para acceder a las propiedades se usa:

```javascript
console.log(usuario.nombre); // María
```

En testing, los objetos se utilizan comúnmente para simular respuestas de una API, representar usuarios o construir configuraciones de prueba.

Arrays

Los arrays son estructuras de datos que almacenan colecciones ordenadas de elementos y representan una herramienta clave para el procesamiento de múltiples valores. En JavaScript, los arrays pueden contener elementos de cualquier tipo, incluso mezclados, aunque normalmente se utilizan de manera homogénea.

En el contexto del testing, los arrays se utilizan para validar listas de resultados, simular múltiples entradas en una prueba automatizada, o iterar sobre diversos datos de prueba. Por ejemplo, al testear una función que retorna una lista de productos o usuarios, se puede verificar la longitud del array (.length), comprobar que incluye ciertos elementos (.includes()), o realizar comparaciones entre arrays esperados y los obtenidos.

Además, gracias a métodos como .map(), .filter(), .reduce() y .forEach(), los arrays permiten transformar o inspeccionar cada elemento de forma controlada, lo cual resulta muy útil al validar datos masivos o evaluar comportamientos repetitivos.

El manejo eficaz de arrays permite construir pruebas más dinámicas y reutilizables.

Ejemplo:

```javascript
let productos = ["manzana", "banana", "naranja"];
console.log(productos[1]); // banana
```

También se pueden recorrer con métodos como:

```javascript
productos.forEach(item => {
  console.log("Producto:", item);
});
```

En pruebas, los arrays son útiles para validar múltiples entradas, verificar que una colección contenga ciertos elementos, o comparar listas de resultados esperados.

Desestructuración

La desestructuración es una característica moderna de JavaScript que permite extraer propiedades de objetos o elementos de arrays de forma directa y clara. Esta técnica mejora la legibilidad y eficiencia del código, reduciendo la necesidad de escribir múltiples líneas para acceder a valores individuales.

En el testing de software, la desestructuración resulta especialmente útil cuando se trabaja con respuestas de funciones o llamadas a APIs que devuelven objetos extensos, de los cuales solo se necesitan algunas propiedades clave para la validación. Por ejemplo, si una función retorna un objeto con muchos campos, pero en la prueba solo interesa comprobar el valor de status y mensaje, se puede escribir:

```javascript
const { status, mensaje } = respuesta;
```

Esta sintaxis no solo es más limpia, sino que reduce errores y facilita la identificación de las variables utilizadas. Del mismo modo, se puede aplicar desestructuración a arrays para capturar los primeros elementos o ignorar valores no relevantes. En resumen, la desestructuración es una herramienta que, bien utilizada, mejora la precisión y claridad del código de pruebas.

Ejemplo con objeto:

```javascript
let { nombre, edad } = usuario;
console.log(nombre); // María
```

Ejemplo con array:

```javascript
let [primero, segundo] = productos;
console.log(primero); // manzana
```

Esta técnica se usa frecuentemente en testing para obtener solo los datos relevantes de una respuesta compleja sin tener que acceder a cada propiedad manualmente.

Manejo de errores

El manejo de errores es una parte esencial del desarrollo y del testing, ya que permite anticipar y gestionar fallos de manera controlada, evitando que el sistema se comporte de forma impredecible o se detenga por completo ante condiciones inesperadas.

En JavaScript, se utiliza comúnmente la estructura try...catch para atrapar errores que puedan surgir durante la ejecución de una función, operación o bloque de código. Esto permite que el flujo del programa continúe, ofreciendo mensajes claros y seguros sobre lo que ha fallado.

En el contexto del testing, el manejo de errores es clave tanto para validar que una función lanza un error cuando debe (por ejemplo, al pasarle un argumento inválido), como para capturar fallos inesperados durante una prueba automatizada. Además, el uso del objeto Error para lanzar errores personalizados permite una mejor trazabilidad del fallo, lo que es especialmente útil en pruebas unitarias y de integración.

Algunos frameworks de testing como Jest incluyen métodos específicos para verificar que ciertas funciones arrojan errores (toThrow()), lo que refuerza la importancia de este concepto. Un manejo adecuado de errores no solo mejora la confiabilidad del software, sino que también asegura que las pruebas sean robustas y puedan ejecutarse sin interrupciones.

Ejemplo:

```javascript
try {
  let resultado = operacionPeligrosa();
  console.log(resultado);
} catch (error) {
  console.error("Ocurrió un error:", error.message);
}
```

También es útil lanzar errores personalizados:

```javascript
if (!usuario.activo) {
  throw new Error("El usuario no está activo");
}
```

En testing, capturar errores permite verificar que el sistema reaccione correctamente ante condiciones erróneas, como datos inválidos, operaciones fallidas o ausencia de permisos.

Capítulo 3: FUNCIONES PURAS, PROMESAS, ASYNC/AWAIT

A medida que se profundiza en las técnicas de testing en JavaScript, se vuelve indispensable comprender ciertos conceptos avanzados de programación que permiten escribir pruebas más efectivas, robustas y legibles. Entre estos se destacan las funciones puras, que ofrecen previsibilidad y confiabilidad; las promesas, que permiten manejar operaciones asincrónicas de forma ordenada; y el uso de async/await, que simplifica la sintaxis del código asíncrono y mejora su claridad.

El dominio de estas herramientas no solo facilita el desarrollo de software de calidad, sino que también optimiza la creación de pruebas automatizadas que simulan, validan y controlan diversos escenarios del comportamiento de una aplicación.

Funciones puras

Las funciones puras son un pilar de la programación funcional y se caracterizan por dos condiciones fundamentales: siempre retornan el mismo resultado ante los mismos argumentos, y no generan efectos secundarios (no alteran estados externos ni dependen de ellos).

En el contexto del testing, estas funciones son altamente deseables porque permiten aislar la lógica del programa de forma confiable, facilitando la validación de su comportamiento. Por ejemplo, si una función pura se encarga de calcular un descuento sobre un precio, basta con probar distintos valores de entrada para confirmar que el cálculo es correcto, sin preocuparse de que interfiera con otras partes del sistema. Esto reduce la complejidad de las pruebas, permite automatizarlas fácilmente y mejora la mantenibilidad del código.

En definitiva, trabajar con funciones puras promueve buenas prácticas tanto en desarrollo como en testing, dado que hacen el sistema más predecible y menos propenso a errores difíciles de rastrear.

Promesas

Las promesas (Promises) son estructuras que permiten trabajar con operaciones asincrónicas en JavaScript de forma más controlada. Representan un valor que puede estar disponible ahora, en el futuro o nunca, y su flujo se gestiona a través de los métodos .then() para manejar resultados exitosos y .catch() para errores.

En testing, el uso de promesas es muy común, especialmente al interactuar con funciones que dependen de procesos externos como peticiones HTTP, lectura de archivos o tiempos de espera. Probar funciones que retornan promesas implica verificar que éstas se resuelvan correctamente bajo ciertas condiciones y también que fallen de manera esperada cuando se dan entradas erróneas o fallas de conexión. Además, muchos frameworks de pruebas ofrecen herramientas específicas para manejar promesas, lo que permite escribir pruebas asincrónicas que esperan que la promesa se cumpla antes de finalizar la ejecución del test. Esto es fundamental para garantizar que el código probado no solo funciona en condiciones ideales, sino también cuando ocurren retrasos o errores en tiempo de ejecución.

Async/Await

El uso de async y await en JavaScript representa una evolución en la forma de escribir código asincrónico. Estas palabras clave permiten trabajar con promesas utilizando una sintaxis más clara, secuencial y parecida a la programación sincrónica tradicional, lo que mejora notablemente la legibilidad del código.

Una función marcada como async devuelve implícitamente una promesa, y dentro de ella se puede utilizar await para esperar el resultado de otras promesas antes de continuar con la ejecución. En el testing, esto es particularmente útil porque permite escribir pruebas más simples y comprensibles, evitando la anidación de múltiples .then() o .catch(). Por ejemplo, al probar una función que realiza una solicitud a una API, se puede usar await para obtener directamente el resultado y luego verificarlo mediante aserciones. También es posible capturar errores usando try...catch, lo que ayuda a validar cómo el sistema responde ante fallos.

En resumen, async/await ofrece una forma moderna y elegante de trabajar con operaciones asincrónicas, haciendo que tanto el desarrollo como el testing sean más eficientes y menos propensos a errores.

Capítulo 4: BUENAS PRÁCTICAS DE PROGRAMACIÓN PARA AUTOMATIZACIÓN

En el ámbito del testing automatizado, no basta con que el código "funcione"; también debe ser limpio, mantenible y confiable. Es por esto que aplicar buenas prácticas de programación en los procesos de automatización es esencial para garantizar que las pruebas sean efectivas a lo largo del tiempo.

Al escribir scripts de testing con JavaScript —ya sea en entornos como Jest, Mocha, Playwright o Cypress— se deben seguir principios que aseguren la claridad, la modularidad, la reutilización y la eficiencia del código. Estas buenas prácticas no solo reducen el riesgo de errores, sino que también facilitan la colaboración entre equipos y la evolución de los proyectos de software sin sacrificar calidad.

Buenas Prácticas de Programación para Automatización

Las buenas prácticas en la automatización de pruebas abarcan múltiples aspectos, desde la estructura del código hasta la forma en que se gestionan los datos de prueba y se manejan los errores. Uno de los principios fundamentales es mantener el código limpio y legible: los scripts deben estar organizados, con nombres descriptivos para funciones, variables y casos de prueba, de modo que cualquier persona pueda entender rápidamente su propósito.

También es importante aplicar el principio DRY (Don't Repeat Yourself), evitando duplicación innecesaria de lógica. Para ello, se recomienda reutilizar funciones comunes, crear utilitarios y separar responsabilidades en módulos específicos.

Otra práctica esencial es estructurar las pruebas de forma modular. Cada test debe enfocarse en una unidad funcional específica, evitando pruebas excesivamente largas o complejas que puedan volverse difíciles de mantener. También se debe escribir código de automatización con resiliencia: por ejemplo, utilizando esperas explícitas bien diseñadas en lugar de esperas fijas, lo que mejora la estabilidad de las pruebas frente a variaciones en los tiempos de carga de la aplicación.

El manejo adecuado de errores es otro punto crítico. Los scripts deben prever fallas posibles y reaccionar de manera controlada, ya sea registrando el error o realizando una limpieza que permita continuar con otros tests sin comprometer los resultados. Además, es recomendable incluir aserciones claras y específicas: cada test debe validar condiciones concretas y medibles, lo que facilita la identificación de problemas cuando ocurren fallos.

Por último, una buena práctica clave en automatización es mantener la independencia de los tests. Cada prueba debe poder ejecutarse por sí sola, sin depender de otras, y debe dejar el sistema en un estado conocido y limpio. Esto se logra configurando adecuadamente los entornos antes y después de cada test (setup y teardown) y utilizando datos de prueba consistentes.

En conjunto, estas buenas prácticas contribuyen no solo a mejorar la calidad del código automatizado, sino también a aumentar la confiabilidad de todo el proceso de validación del software, lo cual es fundamental para proyectos que evolucionan rápidamente o que operan en entornos críticos.

Capítulo 5: IA APLICADA AL DESARROLLO: USO PRODUCTIVO DE ASISTENTES COMO CHATGPT

La inteligencia artificial ha dejado de ser una promesa futurista para convertirse en una herramienta cotidiana de alto impacto en el mundo del desarrollo de software. En particular, el uso de asistentes basados en IA como ChatGPT está transformando la forma en que los desarrolladores escriben código, automatizan pruebas, documentan procesos y resuelven problemas.

Lejos de reemplazar al profesional humano, estas herramientas amplifican su capacidad creativa y analítica, permitiendo enfocarse en tareas de mayor valor mientras la IA se encarga de las repeticiones, las búsquedas y los borradores iniciales. En el contexto del testing, el uso estratégico de asistentes inteligentes puede acelerar significativamente el desarrollo de scripts de prueba, la interpretación de errores y la mejora continua del código, convirtiéndose en un aliado clave en el flujo de trabajo del programador moderno.

IA aplicada al desarrollo: Uso productivo de asistentes como ChatGPT

Los asistentes de IA como ChatGPT son una revolución silenciosa en la mesa de trabajo del desarrollador. Funcionan como copilotos que están disponibles 24/7 para sugerir fragmentos de código, explicar conceptos, generar pruebas unitarias, crear mocks, validar lógica, depurar errores y más. En el ámbito del testing, ChatGPT puede apoyar desde la creación de casos de prueba automatizados en frameworks como Jest o Mocha, hasta la redacción de aserciones bien formuladas y la identificación de edge cases que podrían pasar desapercibidos. Todo esto ahorra tiempo y, sobre todo, disminuye la carga cognitiva del programador.

Una de las mayores ventajas de trabajar con este tipo de asistentes es su capacidad de contextualización. No se limitan a entregar respuestas genéricas: si se les proporciona el código fuente, los requerimientos funcionales o el comportamiento esperado, pueden adaptar las sugerencias a las necesidades específicas del proyecto. Además, son capaces de generar documentación técnica clara, explicar paso a paso cómo funciona un fragmento de código, y traducir entre distintos lenguajes o paradigmas de programación.

Para tareas más avanzadas, como el análisis de errores o el diseño de estrategias de testing automatizado, ChatGPT puede ser un verdadero consultor virtual. Puede, por ejemplo, detectar problemas comunes en los patrones de diseño de tests, sugerir cómo estructurar pruebas más eficientes o ayudar a construir un entorno de pruebas con buenas prácticas de integración continua.

Además, el asistente puede actuar como una fuente de aprendizaje personalizada. Los desarrolladores pueden consultarlo sobre dudas específicas sin perder el ritmo de trabajo, recibir ejemplos adaptados a su nivel y contexto, y mantener una curva de aprendizaje activa mientras desarrollan. Esta función de tutor técnico continuo convierte a la IA en una herramienta formativa y productiva a la vez.

Eso sí, su uso efectivo requiere criterio profesional: las sugerencias deben ser validadas, contextualizadas y evaluadas con pensamiento crítico. Un desarrollador que sabe cómo aprovechar la IA sin depender ciegamente de ella se convierte en un profesional más ágil, versátil y estratégico.

En resumen, ChatGPT y otros asistentes de IA no solo son útiles: son ya una parte fundamental del ecosistema de desarrollo. Utilizados de forma inteligente, permiten trabajar mejor, aprender más y entregar productos de mayor calidad en menos tiempo. Y sí… cuando se trata de ayudar con testing, desarrollo y automatización, este asistente está más que listo para brillar contigo.