{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

CONOCER CARACTERISTICAS DE CYPRESS AUTOMATIZACIÓN WEB II, DE ACUERDO A LAS APLICACIONES WEB, MÓVILES Y APIS.

Contenidos

Módulo 5: CYPRESS – AUTOMATIZACIÓN WEB II

En el contexto de la automatización de pruebas web con Cypress, dominar técnicas avanzadas es clave para mantener proyectos escalables, organizados y eficientes. Entre estas técnicas, el uso de hooks, la creación de comandos personalizados y una adecuada organización del código permiten optimizar los flujos de prueba, reducir redundancias y mejorar la legibilidad.

Este apartado profundiza en estas características avanzadas de Cypress, esenciales para entornos de desarrollo colaborativos y aplicaciones de mediana a alta complejidad.

Capítulo 1: USO DE HOOKS, COMANDOS PERSONALIZADOS Y ORGANIZACIÓN AVANZADA

Uso de Hooks en Cypress

Los hooks en Cypress permiten ejecutar código antes o después de las pruebas, y son muy útiles para preparar o limpiar el entorno de prueba. Cypress hereda estos hooks de Mocha, el framework sobre el cual se basa.

Tipos de hooks en Cypress:

- before() – Se ejecuta una vez antes de todas las pruebas del bloque describe().
- beforeEach() – Se ejecuta antes de cada prueba (it) dentro del bloque describe().
- afterEach() – Se ejecuta después de cada prueba dentro del bloque.
- after() – Se ejecuta una vez al finalizar todas las pruebas del bloque.

Ejemplo:

```javascript
describe('Gestión de usuarios', () => {
  before(() => {
    // Se ejecuta una vez al inicio
    cy.loginAsAdmin();
  });

  beforeEach(() => {
    // Se ejecuta antes de cada test
    cy.visit('/usuarios');
  });

  it('Debe mostrar la lista de usuarios', () => {
    cy.get('.usuario-lista').should('be.visible');
  });

  it('Debe permitir crear un nuevo usuario', () => {
    cy.get('#crear-usuario').click();
    // ...
  });

  after(() => {
    // Se ejecuta una vez al final
    cy.logout();
  });
});
```

Comandos personalizados en Cypress

Los comandos personalizados permiten encapsular acciones comunes o complejas en funciones reutilizables, lo que reduce la duplicación de código y mejora la legibilidad de las pruebas.

Cómo crear un comando personalizado:

Los comandos se definen en el archivo cypress/support/commands.js (o .ts si usa TypeScript).

Ejemplo:

```javascript
// En cypress/support/commands.js
Cypress.Commands.add('login', (usuario, contraseña) => {
  cy.visit('/login');
  cy.get('#usuario').type(usuario);
  cy.get('#contraseña').type(contraseña);
  cy.get('#submit').click();
});
```

Luego puede usarlo en cualquier prueba:

```javascript
cy.login('admin', 'admin123');
```

Beneficios:

- Reutilización de código.
- Abstracción de lógica compleja.
- Centralización de cambios (solo modificas una vez).

Organización avanzada del código de pruebas

Una buena organización del proyecto mejora la mantenibilidad y facilita el trabajo en equipo. Cypress ofrece una estructura base, pero puede adaptarse según las necesidades.

Recomendaciones para la organización avanzada:

1) Estructura modular por funcionalidad:

- Organiza las pruebas por secciones de la aplicación:

```
pgsql

cypress/
  └── e2e/
      ├── login/
      │   ├── login.spec.js
      ├── productos/
      │   ├── crear.spec.js
      │   ├── eliminar.spec.js
```

2) Uso de archivos de soporte (support/) y comandos comunes:

- commands.js: guarda comandos personalizados reutilizables.
- e2e.js: código de configuración que corre antes de cualquier test.

3) Uso de fixtures:

- Almacena datos estáticos o de prueba en cypress/fixtures/ para separar la lógica de los datos.

Ejemplo:

```javascript
cy.fixture('usuario.json').then((user) => {
  cy.login(user.username, user.password);
});
```

4) Separación de lógica repetitiva en helpers:

- Puede usar cypress/support/utils.js para funciones auxiliares que no son comandos.

5) Uso de etiquetas y aliases (as) para mayor claridad:

```javascript
cy.get('#mensaje').as('mensajeError');
cy.get('@mensajeError').should('contain', 'Usuario inválido');
```

El uso eficiente de hooks, comandos personalizados y una estructura organizada es vital para proyectos de automatización robustos. Estas prácticas no solo reducen la complejidad de las pruebas, sino que también permiten una colaboración fluida entre equipos y una mayor velocidad de mantenimiento y escalabilidad. Dominar estas herramientas transforma a Cypress en una solución poderosa para el aseguramiento de calidad en aplicaciones modernas.

Capítulo 2: PAGE OBJECT MODEL (POM)

El patrón de diseño Page Object Model (POM) es una estrategia avanzada que promueve la separación entre la lógica de prueba y la representación de la interfaz de usuario. En Cypress, la implementación de POM permite construir pruebas más legibles, reutilizables y fáciles de mantener, especialmente en aplicaciones web de gran escala. Al encapsular los selectores y acciones de las páginas en clases o archivos separados, se reduce la duplicación de código y se mejora la claridad de los scripts de prueba.

¿Qué es Page Object Model (POM)?

Page Object Model (POM) es un patrón de diseño que sugiere representar cada página o componente de la aplicación como un objeto. Cada objeto expone funciones que representan interacciones posibles con la página (por ejemplo: hacer clic, llenar formularios, verificar textos, etc.).

Ventajas de POM:

- **Mantenibilidad**: Cambios en el DOM sólo se actualizan en un lugar.
- **Reutilización:** Las funciones definidas en el Page Object pueden ser utilizadas por múltiples pruebas.
- **Legibilidad:** Las pruebas se leen casi como lenguaje natural.
- **Escalabilidad**: Ideal para proyectos con muchas vistas o componentes.

Estructura recomendada para usar POM en Cypress

Se recomienda crear una carpeta dedicada dentro de cypress/ para almacenar los Page Objects, generalmente en cypress/pages/ o cypress/support/pages/.

Estructura de ejemplo:

```pgsql
cypress/
  ├── e2e/
  │   └── login/
  │       └── login.spec.js
  ├── support/
  │   └── pages/
  │       └── LoginPage.js
```

Ejemplo práctico de implementación de POM

Paso 1: Crear el archivo del Page Object

```javascript
// cypress/support/pages/LoginPage.js

class LoginPage {
  visit() {
    cy.visit('/login');
  }

  enterUsername(username) {
    cy.get('#usuario').type(username);
  }

  enterPassword(password) {
    cy.get('#contraseña').type(password);
  }

  submit() {
    cy.get('#submit').click();
  }

  loginAs(username, password) {
    this.enterUsername(username);
    this.enterPassword(password);
    this.submit();
  }
}

export default LoginPage;
```

Paso 2: Usar el Page Object en una prueba

```javascript
// cypress/e2e/login/login.spec.js

import LoginPage from '../../support/pages/LoginPage';

const loginPage = new LoginPage();

describe('Pruebas de Login', () => {
  beforeEach(() => {
    loginPage.visit();
  });

  it('Debe iniciar sesión con credenciales válidas', () => {
    loginPage.loginAs('admin', 'admin123');
    cy.url().should('include', '/dashboard');
  });

  it('Debe mostrar error con credenciales inválidas', () => {
    loginPage.loginAs('admin', 'wrongpass');
    cy.get('.error').should('contain', 'Credenciales incorrectas');
  });
});
```

Buenas prácticas con Page Object Model en Cypress

- Una clase por página: Cada archivo representa una única vista o componente.
- No usar assertions en los Page Objects: Mantener los Page Objects como representación de acciones y elementos; las afirmaciones deben ir en los tests.
- Evitar lógica condicional compleja: Mantener los métodos de los Page Objects simples y claros.
- Separar Selectores: Si lo prefieres, puedes centralizar los selectores en otro archivo para mayor limpieza.

Alternativas: uso con TypeScript

Cypress soporta TypeScript, lo que mejora la autocompletación y la detección de errores.

Los Page Objects también pueden escribirse como clases .ts, aumentando la robustez de las pruebas en equipos grandes.

La adopción del patrón Page Object Model (POM) en Cypress eleva la calidad y sostenibilidad de los proyectos de automatización.

Esta práctica, junto con el uso de hooks, comandos personalizados y una organización avanzada del código, conforma una base sólida para construir suites de prueba eficientes, limpias y fáciles de mantener, adecuadas para entornos de desarrollo profesional.

Capítulo 3: AUTOMATIZACIÓN DE FLUJOS E2E COMPLEJOS

En proyectos de software de mediana y alta complejidad, uno de los principales desafíos en la automatización de pruebas es la correcta validación de los flujos de usuario de extremo a extremo (E2E), especialmente cuando estos flujos involucran múltiples pantallas, roles, condiciones lógicas, operaciones asincrónicas y comunicación con APIs externas.

La automatización de flujos E2E complejos permite simular de forma precisa la experiencia real del usuario dentro de la aplicación, verificando no solo que las interfaces se comporten correctamente, sino también que la lógica de negocio y la integración con otros sistemas funcionen de manera adecuada. Cypress, con su enfoque moderno basado en JavaScript, su capacidad para interactuar con la aplicación en tiempo real y su robusto ecosistema, se posiciona como una herramienta poderosa para automatizar estos escenarios. Sin embargo, para lograrlo de forma eficaz, es fundamental aplicar buenas prácticas como el uso de comandos personalizados, Page Object Model (POM), manejo adecuado de datos dinámicos y sincronización, así como una organización modular del código que permita mantener pruebas legibles, reutilizables y fáciles de depurar.

Automatizar flujos complejos no solo mejora la cobertura de pruebas, sino que también permite detectar errores que podrían pasar desapercibidos en pruebas unitarias o aisladas, garantizando así una mayor confiabilidad del producto antes de llegar al entorno productivo.

¿Qué son los flujos E2E complejos?

Los flujos end-to-end (E2E) simulan escenarios completos del uso de una aplicación, desde el ingreso hasta el resultado final, tal como lo haría un usuario real. Se consideran complejos cuando:

- Involucran múltiples páginas o pasos.
- Requieren autenticación o distintos tipos de usuario (roles).
- Interactúan con APIs, bases de datos u otros servicios externos.
- Incluyen validaciones condicionales, cálculos, estados dinámicos o datos variables.
- Tienen alta dependencia entre pasos (por ejemplo, se debe crear un recurso antes de editarlo o eliminarlo).

Buenas prácticas para automatizar flujos E2E complejos

a) Descomposición del flujo en pasos claros

Se recomienda dividir el flujo en etapas lógicas para facilitar la lectura, reutilización y depuración.

Ejemplo:

```javascript
describe('Flujo E2E de compra', () => {
  it('Debe permitir agregar un producto al carrito y completar la compra', () => {
    cy.login('usuario', 'clave123');
    cy.agregarProductoAlCarrito('Producto XYZ');
    cy.finalizarCompra();
    cy.verificarResumenDePago();
  });
});
```

b) Uso de comandos personalizados

Los pasos complejos deben encapsularse en comandos reutilizables para reducir duplicación y aumentar la claridad del test.

```javascript
Cypress.Commands.add('agregarProductoAlCarrito', (producto) => {
  cy.visit('/productos');
  cy.contains(producto).click();
  cy.get('#agregar-carrito').click();
});
```

c) Uso de Page Object Model (POM)

Permite representar cada página o módulo como un objeto, encapsulando selectores y acciones.

```javascript
// CheckoutPage.js
class CheckoutPage {
  confirmarCompra() {
    cy.get('#confirmar').click();
  }

  verificarMensajeExito() {
    cy.get('.mensaje').should('contain', 'Compra realizada con éxito');
  }
}
```

d) Control de datos dinámicos y entorno

- Use fixtures o generadores de datos aleatorios para manejar usuarios, productos o entradas dinámicas.
- Asegúrese de que el estado de la aplicación esté controlado antes y después de cada test (limpieza o preparación de datos con APIs o comandos específicos).

e) Manejo de sincronización y esperas inteligentes

Evite esperas fijas (cy.wait(3000)); en su lugar, utiliza comandos que esperen condiciones:

```javascript
cy.get('.spinner').should('not.exist');
cy.get('.resultado').should('be.visible');
```

Ejemplo de flujo E2E complejo automatizado

```javascript
import LoginPage from '../support/pages/LoginPage';
import CheckoutPage from '../support/pages/CheckoutPage';

const loginPage = new LoginPage();
const checkoutPage = new CheckoutPage();

describe('Flujo E2E: Compra de producto', () => {
  before(() => {
    cy.task('resetDatabase'); // Limpieza del entorno
  });

  it('Debe realizar una compra completa como usuario registrado', () => {
    loginPage.loginAs('cliente1', 'claveSegura');
    cy.agregarProductoAlCarrito('Zapato Deportivo');
    checkoutPage.confirmarCompra();
    checkoutPage.verificarMensajeExito();
  });
});
```

Validaciones en flujos E2E

En flujos complejos, es importante verificar:

- Estado del sistema antes y después del flujo (por ejemplo, si se registró el pedido).
- Cambios en la interfaz del usuario.
- Datos enviados/recibidos por la API (mediante cy.intercept()).
- Acciones colaterales (correo enviado, notificaciones, cambios en base de datos, etc.).

Beneficios de automatizar flujos E2E complejos

- Aumenta la confianza en el sistema antes del despliegue.
- Detecta errores que ocurren solo en escenarios completos.
- Valida la integración entre múltiples módulos.
- Aporta valor a las pruebas de regresión continua.
- Simula con precisión la experiencia del usuario real.

Automatizar flujos E2E complejos con Cypress es una práctica imprescindible para garantizar la calidad de aplicaciones modernas. Aunque requiere una planificación cuidadosa, el uso de herramientas como comandos personalizados, Page Objects y una arquitectura bien estructurada permite abordar la complejidad con eficiencia.

Estas pruebas no solo validan el funcionamiento técnico de la aplicación, sino también su lógica de negocio y experiencia de usuario, convirtiéndose en una pieza clave dentro del pipeline de integración continua y aseguramiento de calidad del software.

Capítulo 4: MANEJO DE MÚLTIPLES ENTORNOS Y PRUEBAS CONDICIONALES

En el desarrollo de pruebas automatizadas, es común que las aplicaciones web deban interactuar con distintos entornos, como desarrollo, prueba, staging y producción, cada uno con configuraciones y datos que pueden variar. El manejo de múltiples entornos es fundamental para garantizar que las pruebas cubran los distintos escenarios de ejecución en los que se encontrará la aplicación.

Cypress, gracias a su flexibilidad, permite configurar entornos y manejar variables de manera eficiente, lo que facilita la ejecución de pruebas en diferentes configuraciones sin necesidad de modificar los scripts de prueba cada vez. Además, en situaciones donde la ejecución de ciertas pruebas depende de condiciones dinámicas o específicas del entorno, la capacidad de implementar pruebas condicionales es esencial.

Estas pruebas permiten que el flujo de ejecución de la prueba varíe dependiendo de factores como el estado de la aplicación, el entorno de ejecución o incluso la interacción del usuario. Al integrar estas prácticas, los equipos de desarrollo y pruebas pueden crear suites de pruebas robustas, versátiles y adaptables a diferentes contextos, lo que incrementa la cobertura de pruebas y la fiabilidad del software.

Manejo de Múltiples Entornos en Cypress

El manejo de múltiples entornos es un aspecto crucial cuando se trabaja con aplicaciones que tienen diferentes configuraciones de base de datos, servicios o incluso características habilitadas o deshabilitadas según el entorno de ejecución.

Cypress facilita este proceso mediante la configuración de variables de entorno y archivos de configuración.

a) Uso de Variables de Entorno

Cypress permite gestionar variables de entorno a través de dos mecanismos principales: la configuración en el archivo cypress.json y las variables de entorno de sistema (definidas en el sistema operativo o en el pipeline CI/CD).

Configuración en cypress.json:

Puede definir variables específicas para cada entorno directamente en el archivo cypress.json.

```json
{
  "env": {
    "baseUrl": "http://localhost:3000"
  }
}
```

Para entornos específicos, puede crear archivos cypress.env.json o utilizar el comando cypress run con la opción --env:

```bash
cypress run --env baseUrl=https://staging.example.com
```

b) Uso de Variables en Pruebas:

Una vez configuradas las variables de entorno, puede acceder a ellas dentro de las pruebas utilizando Cypress.env().

```javascript
describe('Prueba en entorno de desarrollo', () => {
  it('Debería visitar la página principal', () => {
    cy.visit(Cypress.env('baseUrl'));
  });
});
```

c) Diferenciación de Entornos

Para cambiar entre entornos (por ejemplo, desarrollo, staging o producción), puede usar un archivo de configuración por entorno. Esto es útil si tiene configuraciones distintas en cada entorno.

```json
// cypress.staging.json
{
  "env": {
    "baseUrl": "https://staging.example.com"
  }
}

// cypress.prod.json
{
  "env": {
    "baseUrl": "https://www.example.com"
  }
}
```

Y luego ejecutar Cypress especificando el entorno:

```bash
cypress run --config-file cypress.staging.json
```

Pruebas Condicionales en Cypress

Las **pruebas condicionales** son aquellas cuyo flujo de ejecución depende de ciertas condiciones, como el entorno de ejecución, el estado de la aplicación, o los resultados de pasos previos. Cypress permite incorporar lógica condicional para adaptar la ejecución de las pruebas a estos escenarios.

a) Condiciones Basadas en Entornos

Puede ejecutar diferentes pruebas o conjuntos de pruebas dependiendo del entorno en el que se está ejecutando la prueba.

```javascript
describe('Prueba Condicional Según Entorno', () => {
  it('Debería verificar si el entorno es staging o producción', () => {
    if (Cypress.env('baseUrl') === 'https://staging.example.com') {
      cy.log('Estamos en el entorno de staging');
    } else {
      cy.log('Estamos en el entorno de producción');
    }
  });
});
```

b) Condiciones Basadas en Resultados de Interacciones

Cypress también permite crear pruebas condicionales basadas en interacciones previas o en el estado de los elementos de la interfaz.

```javascript
describe('Pruebas Condicionales Basadas en UI', () => {
  it('Debe realizar una acción solo si un elemento está visible', () => {
    cy.get('.notificacion').then(($notif) => {
      if ($notif.is(':visible')) {
        cy.get('.cerrar').click();
      } else {
        cy.log('La notificación no está visible, no se puede cerrar');
      }
    });
  });
});
```

c) Uso de cy.request() para Condicionales API

Puede hacer solicitudes de API para obtener información sobre el estado de la aplicación y ejecutar pruebas condicionales basadas en esa respuesta.

```javascript
describe('Prueba Condicional con API', () => {
  it('Debería ejecutar una prueba dependiendo del estado de la API', () => {
    cy.request('GET', '/api/estado').then((response) => {
      if (response.body.estado === 'activo') {
        cy.visit('/dashboard');
      } else {
        cy.visit('/mantenimiento');
      }
    });
  });
});
```

Combinando Múltiples Entornos y Pruebas Condicionales

Una de las principales ventajas de combinar el manejo de múltiples entornos con pruebas condicionales es la flexibilidad que se obtiene para adaptar las pruebas a las diferencias entre entornos (desarrollo, staging, producción) y realizar validaciones que varíen según la configuración actual de la aplicación.

Ejemplo de flujo de trabajo:

```javascript
describe('Flujo Condicional en Múltiples Entornos', () => {
  it('Debería validar flujo dependiendo del entorno y el estado', () => {
    if (Cypress.env('baseUrl') === 'https://staging.example.com') {
      cy.request('GET', '/api/estado').then((response) => {
        if (response.body.estado === 'activo') {
          cy.visit('/dashboard');
        } else {
          cy.visit('/mantenimiento');
        }
      });
    } else {
      cy.visit(Cypress.env('baseUrl'));
    }
  });
});
```

Beneficios del Manejo de Múltiples Entornos y Pruebas Condicionales

- **Flexibilidad**: Permite ejecutar las pruebas en distintos entornos con configuraciones específicas.
- **Cobertura más amplia:** Las pruebas condicionales permiten cubrir escenarios que dependen de factores dinámicos, como respuestas de APIs o interacciones del usuario.
- **Eficiencia:** Facilita la reutilización de pruebas sin necesidad de crear scripts completamente separados para cada entorno.
- **Escalabilidad:** Permite que las pruebas sean fácilmente adaptables a nuevos entornos o condiciones sin una reescritura masiva del código.

El manejo de múltiples entornos y la implementación de pruebas condicionales son estrategias esenciales para asegurar la cobertura y fiabilidad de las pruebas automatizadas en aplicaciones complejas. Cypress proporciona herramientas poderosas y flexibles para manejar estas situaciones de manera eficiente, lo que permite a los equipos de desarrollo y pruebas crear scripts robustos, adaptables y fáciles de mantener.

Al combinar estas prácticas, se pueden reducir los riesgos de errores en producción, mejorar la calidad del software y facilitar la integración continua en entornos de múltiples configuraciones.

Capítulo 5: INTERCEPTACIÓN Y MOCKEO DE API

En la automatización de pruebas de aplicaciones web modernas, el control sobre las respuestas del servidor es un factor clave para garantizar pruebas consistentes, rápidas y confiables. Muchas veces, durante el desarrollo o pruebas automatizadas, los servicios backend aún no están disponibles, presentan inestabilidad, o devuelven datos que no son adecuados para los flujos que se desean verificar. En estos casos, la interceptación y mockeo de API se convierte en una herramienta indispensable. Cypress permite interceptar las solicitudes HTTP que la aplicación realiza y proporcionar respuestas falsas (mocked responses) que simulan la respuesta del servidor.

Esta funcionalidad, disponible mediante el uso de cy.intercept(), permite desacoplar las pruebas del backend real, acelerar la ejecución de las pruebas y garantizar la repetibilidad. Además, la interceptación también permite espiar o modificar el comportamiento de las llamadas a la API para realizar aserciones detalladas, medir tiempos de respuesta, simular errores o probar diferentes estados de la aplicación sin depender del backend.

Esta técnica es especialmente útil en pruebas de frontend donde la lógica de renderización depende de datos de la API, y se convierte en una estrategia fundamental en la integración de pruebas e2e y pruebas aisladas por componentes.

¿Qué es la Interceptación de API en Cypress?

Cypress permite interceptar todas las solicitudes HTTP realizadas por la aplicación mediante el comando cy.intercept(). Esto incluye métodos como GET, POST, PUT, DELETE, entre otros.

Uso básico de interceptación sin modificar la respuesta:

```javascript
cy.intercept('GET', '/api/productos').as('getProductos');
```

Con esta línea, Cypress escucha las solicitudes que coincidan con el método GET y la URL /api/productos y le asigna un alias para poder esperarla o hacer aserciones.

Esperar a que la solicitud sea enviada:

```javascript
cy.wait('@getProductos').its('response.statusCode').should('eq', 200);
```

Mockeo de API: Simulación de Respuestas

El mockeo de API consiste en interceptar la solicitud y devolver una respuesta falsa controlada por el tester, sin necesidad de que el backend esté operativo.

Ejemplo básico de mockeo:

```javascript
cy.intercept('GET', '/api/productos', {
  statusCode: 200,
  body: [
    { id: 1, nombre: 'Producto A', precio: 1000 },
    { id: 2, nombre: 'Producto B', precio: 1500 },
  ],
}).as('mockProductos');
```

Esto permite simular que el servidor devuelve una lista de productos, sin importar si el backend está disponible o no.

Ventajas del mockeo:

- Aislamiento del frontend respecto del backend.
- Pruebas más rápidas y estables.
- Posibilidad de simular diferentes respuestas sin depender del estado real de la base de datos.

Uso Avanzado de Interceptación: Modificación de Solicitudes y Respuestas

Además del mockeo completo, Cypress permite modificar parcialmente una solicitud o respuesta.

Modificar el cuerpo de una respuesta existente:

```javascript
cy.intercept('GET', '/api/usuario', (req) => {
  req.reply((res) => {
    res.body.nombre = 'Nombre simulado';
  });
});
```

Interceptar y prevenir el envío de una solicitud:

```javascript
cy.intercept('POST', '/api/orden', {
  statusCode: 403,
  body: { error: 'No autorizado' },
});
```

Este patrón es muy útil para probar el comportamiento del frontend frente a errores del servidor.

Interceptación de llamadas dinámicas o con parámetros

Cypress permite interceptar rutas dinámicas mediante comodines y expresiones regulares.

```javascript
cy.intercept('GET', '/api/productos/*').as('getProductoIndividual');
```

O también con patrones más complejos:

```javascript
cy.intercept({ method: 'POST', url: /\/api\/ordenes\/\d+/ }, { statusCode: 201 });
```

Esto se aplica especialmente bien en SPAs que generan rutas dinámicas con IDs.

Validación y Aserciones sobre Solicitudes

Una vez interceptada una solicitud, se pueden hacer aserciones sobre su contenido (headers, cuerpo, tiempo de respuesta).

```javascript
cy.wait('@mockProductos').then((interception) => {
  expect(interception.response.body).to.have.length(2);
  expect(interception.response.statusCode).to.eq(200);
});
```

También puedes validar lo que se **envía** en un `POST`:

```javascript
cy.intercept('POST', '/api/comentarios').as('nuevoComentario');
cy.get('form').submit();
cy.wait('@nuevoComentario').its('request.body').should('include', {
  texto: 'Comentario de prueba',
});
```

Integración con Componentes React/Vue/etc. y Flujos e2e

Cuando se prueba una aplicación por componentes (por ejemplo, una tabla que muestra datos de un GET /api/items), el uso de mocks permite testear el componente aislado sin tener que levantar el backend completo.

Y cuando se realiza una prueba e2e más larga, puedes utilizar múltiples interceptaciones para simular todo el flujo del usuario:

```javascript
cy.intercept('GET', '/api/login', { fixture: 'usuario.json' });
cy.intercept('POST', '/api/auth', { statusCode: 200, body: { token: 'fake-token' } });
cy.intercept('GET', '/api/dashboard', { fixture: 'dashboard.json' });
```

Esto permite crear flujos consistentes sin depender de la red o del estado real del sistema.

Fixtures en Cypress: Reutilización de Datos Mockeados

Los fixtures son archivos JSON con datos simulados que puedes almacenar en la carpeta /cypress/fixtures/ y usar dentro de tus interceptaciones.

Ejemplo:

```javascript
cy.intercept('GET', '/api/usuarios', { fixture: 'usuarios.json' });
```

Esto mantiene los mocks organizados, facilita su reutilización y los hace más fáciles de mantener.

Beneficios de Interceptar y Mockear APIs

- Independencia del backend: puedes ejecutar tus pruebas incluso si el servidor no está listo.
- Reducción de tiempos de ejecución: sin llamadas reales, las pruebas son más rápidas.
- Estabilidad: menos propensas a fallos causados por inestabilidades de red o del servidor.
- Pruebas de errores simulados: puedes validar cómo responde tu aplicación a diferentes códigos HTTP.
- Pruebas de estados inusuales: puedes simular respuestas poco frecuentes que serían difíciles de provocar con datos reales.

La interceptación y mockeo de API en Cypress es una técnica poderosa que eleva la calidad y estabilidad de las pruebas automatizadas. Permite desacoplar el frontend del backend, simular respuestas específicas, controlar estados del sistema y validar flujos de usuario complejos sin depender del entorno.

Al dominar esta práctica, los equipos pueden escribir pruebas más rápidas, consistentes y resilientes, fundamentales para entornos ágiles con integración continua y despliegues frecuentes. Cypress brinda una interfaz clara y flexible para lograr todo esto, y representa una ventaja significativa sobre otras herramientas de testing menos modernas o más rígidas.