{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 I, DE ACUERDO A LAS APLICACIONES WEB, MÓVILES Y APIS.

Contenidos

Objetivo General del Curso 2

Objetivo específico del Módulo 2

Módulo 4: 4

CYPRESS – AUTOMATIZACIÓN WEB I 4

Capítulo 1: INSTALACIÓN Y ESTRUCTURA DE UN PROYECTO CYPRESS 6

Instalación y estructura de un proyecto Cypress 6

Estructura de un proyecto Cypress 8

Capítulo 2: COMANDOS ESENCIALES Y VALIDACIONES BÁSICAS 10

Capítulo 3: MANEJO DE DATOS (FIXTURES) Y BUENAS PRÁCTICAS DE TESTING 17

¿Qué son los fixtures en Cypress? 17

Capítulo 4: CAPTURA DE EVIDENCIA (SCREENSHOTS, VIDEOS, REPORTES) 21

Capturas de pantalla automáticas y manuales 22

Integración con CI/CD y utilidad práctica 25

Módulo 4:
CYPRESS – AUTOMATIZACIÓN WEB I

En el mundo del testing automatizado, Cypress se ha consolidado como una de las herramientas más modernas, accesibles y potentes para realizar pruebas end-to-end (E2E) en aplicaciones web. A diferencia de otros frameworks, Cypress se ejecuta directamente en el navegador, lo que le permite interactuar con la aplicación en tiempo real, ofreciendo una experiencia visual clara del flujo de las pruebas. Su arquitectura innovadora permite realizar pruebas rápidas, con una sintaxis intuitiva basada en JavaScript, que facilita la adopción tanto por parte de testers como de desarrolladores.

Además, incorpora funcionalidades como recarga automática, captura de errores, grabación de pruebas y un potente panel de depuración, convirtiéndolo en una herramienta de alto valor para equipos ágiles y proyectos con ciclos de desarrollo continuos.

Capítulo 1: INSTALACIÓN Y ESTRUCTURA DE UN PROYECTO CYPRESS

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.

Instalación y estructura de un proyecto Cypress

¿Qué es Cypress?

Cypress es una herramienta de testing end-to-end de código abierto diseñada específicamente para pruebas automatizadas de aplicaciones web modernas. Está desarrollada sobre JavaScript y funciona directamente en el navegador, permitiendo observar en tiempo real cómo se ejecutan los tests.

Su propuesta de valor se basa en la simplicidad de uso, una instalación sencilla, una documentación clara y una interfaz visual muy completa. A diferencia de otras herramientas como Selenium, que operan en un entorno externo al navegador, Cypress se ejecuta dentro del mismo contexto que la aplicación, lo que permite acceder al DOM, interceptar peticiones, controlar el tiempo de ejecución y depurar con gran precisión.

Instalación de Cypress

Para comenzar a trabajar con Cypress, es necesario tener instalado Node.js en el equipo. Cypress se instala como una dependencia del proyecto, lo que facilita su integración con distintos entornos de desarrollo.

A continuación, los pasos básicos para instalarlo:

1.- Inicializar el proyecto con npm

Si aún no existe un proyecto, puede iniciarlo ejecutando en la terminal:

```bash
npm init -y
```

2.- Instalar Cypress

Luego, se instala Cypress como una dependencia de desarrollo:

```bash
npm install cypress --save-dev
```

3.- Abrir Cypress

Una vez instalado, Cypress se puede abrir por primera vez usando:

```bash
npx cypress open
```

Este comando abrirá la interfaz gráfica de Cypress y generará automáticamente una estructura de carpetas básicas dentro del proyecto.

Estructura de un proyecto Cypress

Al ejecutar Cypress por primera vez, se crea automáticamente una carpeta llamada cypress/, además del archivo de configuración principal cypress.config.js o cypress.json (dependiendo de la versión).

La estructura general incluye:

- cypress/
  Carpeta principal que contiene los archivos de prueba y soporte.
  - e2e/
    Contiene los archivos de pruebas end-to-end. Aquí se definen los casos de prueba utilizando sintaxis JavaScript.
  - fixtures/
    Almacena archivos JSON u otros datos simulados que se pueden usar en los tests (por ejemplo, usuarios, respuestas de API, etc.).
  - support/
    Archivos que se cargan automáticamente antes de cada test. Comúnmente se usa para definir comandos personalizados o configuración global.
- cypress.config.js
  Archivo de configuración donde se pueden personalizar rutas, tiempos de espera, reportes, entre otros parámetros del comportamiento de Cypress.

Ejemplo básico de test

Un test básico en Cypress se ve así:

```javascript
describe('Mi primera prueba', () => {
  it('Visita el sitio y verifica el título', () => {
    cy.visit('https://ejemplo.com')
    cy.title().should('include', 'Ejemplo')
  })
})
```

Este código abre una página web y verifica que el título contenga la palabra "Ejemplo".

Ventajas de Cypress:

- Fácil instalación e integración.
- Recarga automática de pruebas al guardar.
- Panel visual con ejecución paso a paso.
- Acceso completo al DOM y a las peticiones de red.
- Interfaz intuitiva para debugging.
- Amplia comunidad y soporte constante.

Capítulo 2: COMANDOS ESENCIALES Y VALIDACIONES BÁSICAS

Al comenzar con Cypress, es fundamental dominar un conjunto de comandos esenciales y validaciones básicas que permiten interactuar con la interfaz de usuario y comprobar el comportamiento esperado de una aplicación web. Estos comandos constituyen la base para desarrollar pruebas automatizadas fiables, legibles y efectivas. Cypress ofrece una sintaxis sencilla, basada en JavaScript, que permite describir de forma clara y concisa las interacciones que un usuario realizaría en una aplicación.

Además, incorpora mecanismos de aserción integrados mediante la librería Chai, que facilitan la validación de resultados esperados sin necesidad de dependencias externas.

Cypress ofrece una amplia gama de comandos integrados que permiten interactuar con la aplicación, validar comportamientos y manipular el DOM, todo de forma fluida y legible. Estos comandos comienzan con el prefijo cy. y se pueden encadenar para crear pruebas más precisas y legibles.

Comandos de navegación y acción

- cy.visit(url)
  Navega a una URL específica.

```js
cy.visit('https://miweb.com')
```

- cy.reload()
  Recarga la página actual.

```js
cy.reload()
```

- cy.go('back') / cy.go('forward')
  Navega hacia atrás o adelante en el historial del navegador.

```js
cy.go('back')
```

Interacción con elementos

- cy.get(selector)
  Obtiene un elemento del DOM usando un selector CSS.

```js
cy.get('#boton-enviar')
```

- cy.click()
  Hace clic en un elemento.

```js
cy.get('.btn').click()
```

- cy.type('texto')
  Escribe texto dentro de un input.

```js
cy.get('input[name="usuario"]').type('usuario123')
```

- cy.select('opción')
  Selecciona una opción en un <select>.

```js
cy.get('select').select('Chile')
```

- cy.check() / cy.uncheck()
  Marca o desmarca checkboxes.

```js
cy.get('input[type="checkbox"]').check()
```

Asersiones (verificaciones)

- cy.should('contain', texto)
  Verifica que el elemento contenga cierto texto.

```js
cy.get('h1').should('contain', 'Bienvenido')
```

- cy.should('have.length', número)
  Verifica la cantidad de elementos seleccionados.

```js
cy.get('.producto').should('have.length', 5)
```

- cy.should('be.visible')
  Verifica que un elemento sea visible en la pantalla.

```js
cy.get('#mensaje').should('be.visible')
```

Esperas y temporizadores

- cy.wait(tiempo)
  Pausa la prueba por la cantidad de milisegundos indicada.

```js
cy.wait(1000) // espera 1 segundo
```

- cy.intercept()
  Intercepta peticiones HTTP para simular respuestas (mocking).

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

Comandos especiales y utilitarios

- cy.screenshot()
  Toma una captura de pantalla del estado actual de la prueba.

```js
cy.screenshot('pantalla-login')
```

- cy.url()
  Obtiene la URL actual del navegador.

```js
cy.url().should('include', '/dashboard')
```

- cy.title()
  Obtiene el título de la página.

```js
cy.title().should('eq', 'Mi Aplicación')
```

- cy.contains(texto)
  Selecciona un elemento que contenga cierto texto.

```js
cy.contains('Iniciar sesión').click()
```

Buenas prácticas con comandos

- Encadenar comandos para evitar el uso de variables intermedias.
- Evitar cy.wait() cuando sea posible; preferir cy.intercept() o cy.get().should().
- Usar selectores únicos y semánticos para mayor estabilidad.
- Agrupar comandos personalizados usando Cypress.Commands.add() para reutilizar lógica común.

Capítulo 3: MANEJO DE DATOS (FIXTURES) Y BUENAS PRÁCTICAS DE TESTING

El manejo eficiente de datos es un pilar fundamental en la automatización de pruebas, especialmente cuando se busca simular distintos escenarios sin depender directamente del backend. Cypress facilita esta tarea mediante el uso de fixtures, archivos que contienen datos estáticos en formato JSON, los cuales pueden ser reutilizados en múltiples pruebas.

Esto permite mantener las pruebas limpias, evitar la repetición de datos y mejorar la mantenibilidad del código de testing. Además, incorporar buenas prácticas al momento de diseñar, escribir y mantener pruebas automatizadas incrementa su confiabilidad y facilita su comprensión por parte de todo el equipo de desarrollo.

¿Qué son los fixtures en Cypress?

Los fixtures son archivos de datos ubicados en la carpeta cypress/fixtures, comúnmente en formato .json, aunque también pueden estar en .txt, .csv, .js, etc. Su propósito es proporcionar información predefinida que puede ser utilizada dentro de las pruebas para simular respuestas de API, datos de formularios, usuarios ficticios, productos, entre otros.

Estructura típica de un archivo fixture
Archivo: cypress/fixtures/usuario.json

```json
{
  "nombre": "Juan",
  "email": "juan@ejemplo.com",
  "password": "1234"
}
```

Cómo cargar un fixture en una prueba

```js
describe('Login con datos del fixture', () => {
  it('debe completar el formulario usando datos de un archivo', () => {
    cy.fixture('usuario').then((usuario) => {
      cy.visit('/login')
      cy.get('input[name="email"]').type(usuario.email)
      cy.get('input[name="password"]').type(usuario.password)
      cy.get('form').submit()
    })
  })
})
```

Ventajas de usar fixtures

- Permiten reutilizar datos en múltiples pruebas.
- Facilitan la separación entre lógica de prueba y datos.
- Ayudan a construir pruebas más limpias, legibles y fáciles de mantener.
- Son útiles para mockear peticiones de API cuando el backend no está disponible.
- Facilitan la simulación de datos realistas para pruebas más representativas.

Buenas prácticas de testing con Cypress

Aplicar buenas prácticas al desarrollar pruebas automatizadas asegura un código robusto, reutilizable y mantenible en el tiempo. Aquí algunas recomendaciones clave:

1. Nombrar pruebas de forma clara y descriptiva
Cada prueba debe tener un nombre que indique con claridad lo que se está verificando.

```js
it('debe mostrar un error si el email es inválido', () => { ... })
```

2. Separar lógica y datos
Evitar hardcodear datos en las pruebas. En su lugar, usar fixtures u objetos definidos en archivos externos.

3. Utilizar before() y beforeEach() para reutilización de código

```js
beforeEach(() => {
  cy.visit('/login')
})
```

4. Evitar el uso innecesario de cy.wait()
A menos que sea estrictamente necesario, evitar esperas artificiales. En su lugar, preferir comandos como cy.get().should() o interceptores.

5. Encadenar comandos para mayor legibilidad

```js
cy.get('input#email').type('test@correo.com').should('have.value', 'test@correo.com')
```

6. Limitar pruebas a un solo objetivo
Cada it() debe testear un comportamiento específico. Evita incluir múltiples validaciones que no estén relacionadas directamente.

7. Centralizar selectores y rutas repetitivas
Usar variables o archivos de configuración para manejar selectores y URLs frecuentes, evitando duplicación de código.

8. Usar comandos personalizados (Cypress.Commands.add)
Cuando una misma acción se repite, como "iniciar sesión" o "registrar usuario", crear un comando personalizado mejora la reutilización y el mantenimiento.

```js
Cypress.Commands.add('login', (usuario) => {
  cy.visit('/login')
  cy.get('input[name="email"]').type(usuario.email)
  cy.get('input[name="password"]').type(usuario.password)
  cy.get('form').submit()
})
```

El uso de fixtures en Cypress representa una herramienta poderosa para simplificar y enriquecer la automatización de pruebas, permitiendo una mayor flexibilidad en el manejo de datos. Cuando se combina con buenas prácticas de desarrollo de pruebas, se logra una base sólida que favorece el trabajo colaborativo, mejora la calidad del código y reduce los errores en producción. Cypress no solo permite testear, sino que impulsa una cultura de calidad desde el desarrollo.

Capítulo 4: CAPTURA DE EVIDENCIA (SCREENSHOTS, VIDEOS, REPORTES)

En el mundo del testing automatizado, uno de los aspectos más relevantes para el análisis y la trazabilidad de errores es la capacidad de capturar evidencia visual y textual de la ejecución de pruebas.

Cypress destaca en este aspecto al proporcionar, de forma nativa, herramientas para registrar el comportamiento de las pruebas mediante capturas de pantalla, grabaciones de video y generación de reportes estructurados, lo que resulta fundamental en procesos de integración continua (CI), auditorías de calidad o comunicación con otros miembros del equipo de desarrollo.

Estas evidencias no solo ayudan a detectar y corregir errores con mayor rapidez, sino que también refuerzan la transparencia y fiabilidad del proceso de pruebas automatizadas.

Capturas de pantalla automáticas y manuales

Cypress permite capturar screenshots automáticos cuando una prueba falla, lo cual es especialmente útil en entornos de integración continua. Estas imágenes se guardan por defecto en la carpeta cypress/screenshots y pueden personalizarse fácilmente.

Captura automática al fallar una prueba

Esto ocurre sin necesidad de configuración adicional cuando se corre Cypress con el comando:

```bash
npx cypress run
```

Captura manual en cualquier punto de la prueba

Si se desea capturar un momento específico, se puede usar:

```js
cy.screenshot('nombre-personalizado')
```

También se puede capturar un elemento en particular:

```js
cy.get('.boton-enviar').screenshot()
```

Esto permite tener mayor control sobre la documentación visual del proceso de testing.

Grabación de video de las pruebas

Otra característica potente de Cypress es la grabación en video de la ejecución de pruebas en modo headless (cypress run). Estos archivos se almacenan en la carpeta cypress/videos y permiten revisar, paso a paso, el comportamiento de una prueba en tiempo real, lo que es muy útil para identificar condiciones de fallo que no se reflejan claramente en los logs o pantallas estáticas.

Ejecución con grabación de video

```bash
npx cypress run
```

Se genera automáticamente un video por cada archivo de prueba ejecutado. Si todas las pruebas del archivo pasan, Cypress puede eliminar automáticamente el video (configurable).

Configuración opcional en cypress.config.js

```js
video: true,
videosFolder: 'cypress/videos',
trashAssetsBeforeRuns: true,
```

Generación de reportes de ejecución

Aunque Cypress no incluye por defecto una herramienta de reportes visuales detallados, se puede integrar fácilmente con librerías externas como Mochawesome, Allure o incluso Jenkins, para generar reportes personalizados en HTML, JSON o PDF.

Generación de reportes con Mochawesome (ejemplo)

1.- Instalar dependencias:

```bash
npm install --save-dev mochawesome mochawesome-merge mochawesome-report-generator
```

2.- Configurar en cypress.config.js o cypress.json:

```json
{
  "reporter": "mochawesome",
  "reporterOptions": {
    "reportDir": "cypress/reports",
    "overwrite": false,
    "html": false,
    "json": true
  }
}
```

3.- Combinar y generar el HTML:

```bash
npx mochawesome-merge cypress/reports/*.json > merged-report.json
npx marge merged-report.json -f report -o cypress/reports/html
```

Este flujo permite tener reportes gráficos con detalles como escenarios pasados, fallidos, tiempos de ejecución, logs, etc.

Integración con CI/CD y utilidad práctica

La utilidad de contar con capturas y reportes es aún más evidente en entornos donde las pruebas se ejecutan automáticamente en pipelines de Integración Continua y Entrega Continua (CI/CD). Aquí, contar con evidencias claras puede ser la diferencia entre un despliegue exitoso y una regresión inadvertida. Herramientas como GitHub Actions, GitLab CI, CircleCI, Jenkins, entre otros, pueden configurarse para almacenar y mostrar estas evidencias como artefactos del job, facilitando el acceso por parte de todo el equipo.

Ventajas de capturar evidencia en pruebas automatizadas

- **Facilita el debugging**: Ver el momento exacto donde falla una prueba permite identificar errores de lógica, sincronización o visualización.
- **Documenta el proceso de prueba**: Ideal para auditar procesos o entregar resultados a áreas no técnicas.
- **Mejora la colaboración**: Otros desarrolladores o QA pueden acceder a la evidencia sin necesidad de reproducir la prueba.
- **Aumenta la trazabilidad**: Se puede asociar cada error reportado con su video, screenshot y reporte.
- **Permite comparar regresiones visuales**: Especialmente útil en proyectos con cambios frecuentes de UI.

La captura de evidencia en Cypress no es solo un complemento, sino una herramienta estratégica para lograr una automatización de pruebas de calidad profesional. Con capturas, videos y reportes detallados, el proceso de testing se vuelve más claro, trazable y eficiente, favoreciendo la colaboración entre equipos y la detección temprana de errores. Incorporar estas prácticas en el flujo de trabajo asegura no solo pruebas más confiables, sino también un producto final más robusto.