«Talento Tech»

Desarrollo Web 4

Clase 08

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

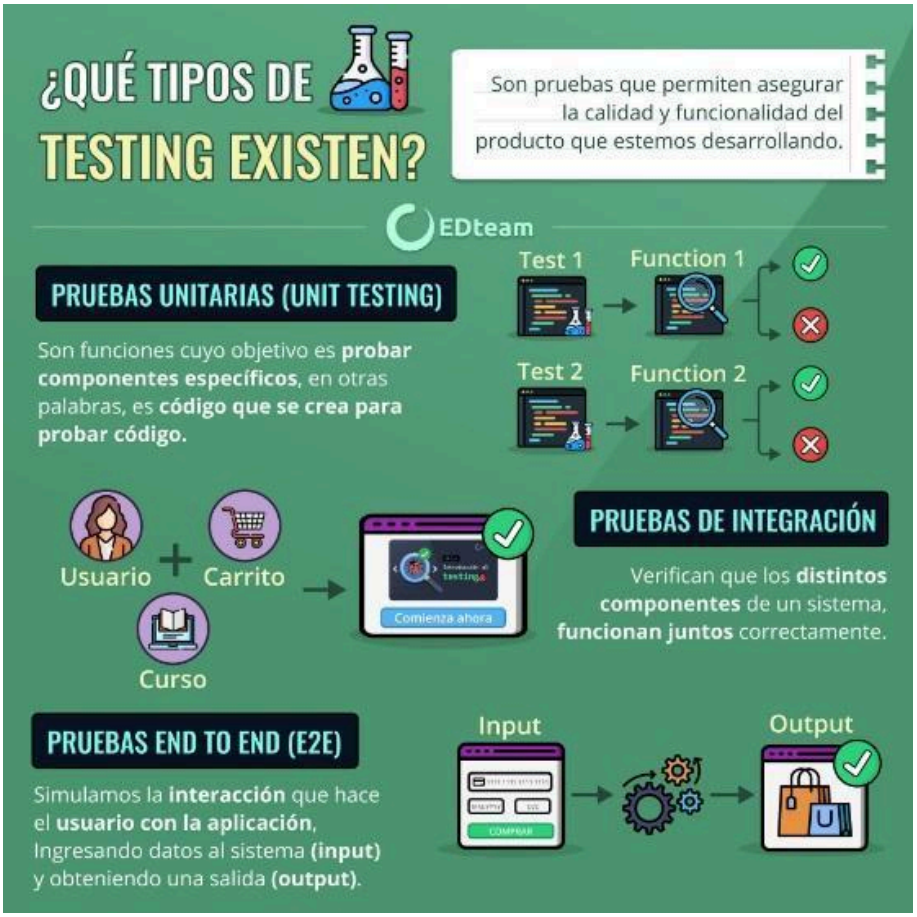
Clase N° 8 | Testing

Temario:

- Introducción a Pruebas Unitarias y End to End
- Frameworks, Estructuras y Asertaciones en Pruebas E2E y Unitarias

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Introducción a Pruebas Unitarias y End to End



Las pruebas unitarias y las pruebas end-to-end (E2E) son prácticas fundamentales en el desarrollo de software que se centran en garantizar la calidad y el rendimiento de las aplicaciones.

Definición y Propósito de las Pruebas Unitarias

Propósito:
Las **pruebas unitarias** son procesos automatizados diseñados para evaluar cada unidad individual (como funciones, métodos o bloques de código) de manera aislada.

Estas pruebas verifican que cada componente funcione según lo esperado, asegurando la corrección de pequeñas unidades antes de integrarlas en el sistema completo.

Propósito:

1. **Validación de la Funcionalidad:** Las pruebas unitarias garantizan que cada componente cumpla con su propósito específico, detectando y corrigiendo errores en las primeras etapas del desarrollo.

2. **Mantenimiento de la Calidad del Código:** Al probar unidades individuales, se mejora la calidad del código, facilitando la identificación y corrección de problemas antes de que se vuelvan críticos.

3. **Facilitación del Refactorizado:** Las pruebas unitarias brindan confianza al realizar cambios en el código, permitiendo ajustes y mejoras con menor riesgo de introducir errores.

Definición y Propósito de las Pruebas End-to-End

Las **pruebas end-to-end** evalúan la aplicación en su conjunto, simulando la experiencia del usuario desde el inicio hasta el final. Estas pruebas verifican la interacción de todos los componentes, desde la interfaz de usuario hasta la base de datos, asegurando la funcionalidad general.

Propósito:

1**. Validación del Flujo Completo:** Las pruebas E2E confirman que todas las partes de la aplicación trabajan armoniosamente, garantizando que el flujo de trabajo completo sea funcional y libre de errores.

2. **Identificación de Problemas de Integración:** Estas pruebas son cruciales para detectar problemas de integración entre diferentes componentes del sistema, como la interfaz de usuario y la lógica de negocios.

3. **Garantía de Experiencia del Usuario:** Las pruebas E2E se centran en simular las acciones del usuario, asegurando que la aplicación brinde una experiencia coherente y satisfactoria.

Beneficios de Implementar Ambos Tipos de Pruebas

1. Detección Temprana y Validación Completa: Las pruebas unitarias ofrecen detección temprana de errores en componentes individuales, mientras que las pruebas E2E validan el sistema en su totalidad, asegurando una cobertura completa.

2. Mantenimiento Eficiente y Corrección Proactiva: Las pruebas unitarias facilitan la corrección proactiva de errores en unidades pequeñas, mientras que las pruebas E2E identifican problemas de integración y garantizan la corrección a nivel de sistema.

3. Eficiencia en Desarrollo y Confianza General: La combinación de pruebas unitarias y E2E agiliza el desarrollo, brinda confianza en la calidad del código y mejora la eficiencia en la entrega de software.

Al comprender y aplicar tanto las pruebas unitarias como las pruebas E2E, los desarrolladores pueden asegurar la robustez y la funcionalidad integral de sus aplicaciones. La combinación de estas prácticas contribuye significativamente a un proceso de desarrollo de software sólido y confiable.

Frameworks, Estructuras y Asertaciones en Pruebas E2E y Unitarias

Introducción a los Frameworks de Pruebas

Los **frameworks de pruebas** son conjuntos estructurados de herramientas y convenciones que facilitan la planificación, escritura y ejecución de pruebas en el desarrollo de software. Estos frameworks proporcionan una estructura organizada para las pruebas, estableciendo normas que simplifican la creación de casos de prueba y la interpretación de resultados. Su objetivo principal es mejorar la eficiencia y la calidad del proceso de desarrollo al ofrecer herramientas estandarizadas para la automatización y verificación de la funcionalidad del código.
Algunos frameworks, como **Jest y Mocha,** están especializados en pruebas unitarias, mientras que otros, como **Selenium**, se centran en pruebas end-to-end. La elección del framework depende de los requisitos específicos del proyecto y del tipo de pruebas que se estén realizando.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Los frameworks más conocidos en Javascript: Jest y Mocha

Jest es un framework de pruebas unitarias para **JavaScript/TypeScript.** Destaca por su configuración sencilla, potentes aserciones y capacidades integradas de "**mocking**". Su estructura amigable y su capacidad para ejecutar pruebas de forma paralela lo convierten en una opción popular. Jest ofrece funciones adicionales como "snapshot testing" para facilitar la comparación visual de resultados.

Mocha es un flexible framework de pruebas unitarias para **JavaScript/TypeScript**, ampliamente utilizado tanto en entornos **Node.js** como en navegadores. Su fuerza radica en su modularidad, permitiendo la integración con diversas **bibliotecas de aserciones**, como **Chai**. Mocha facilita la ejecución de pruebas asíncronas y ofrece informes detallados, siendo una opción versátil para diferentes necesidades de pruebas en proyectos JavaScript.

A continuación vamos a hacer un ejemplo de prueba **E2E** de una sencilla api rest que tiene dos endpoints get y post.

Para poder comenzar necesitamos inicializar el proyecto de la siguiente forma:

```
npm init -y
```

Es importante que el package.json que vamos a crear agreguemos debajo de la línea “main” la siguiente línea

```
"type": "module"
```

Dato: El nombre de la carpeta la llame “dw4-clase-testing” en este caso podes llamarla de la forma que quieras.

El archivo package.json nos debería quedar de la siguiente forma:

```
{
  "name": "dw4-clase-testing",
  "version": "1.0.0",
  "description": "",
  "main": "index.js"
  "type": "module"
  "scripts": {
  "test": "mocha test.js"
  },
   "keyword": []
   "author": "",
   "license": "ISC",
   "dependencies": {
      "body-parser": "^1.20.2",
      "express": "^4.18.2",
      "mysql": "^2.18.1"
   },
  "devDependencies": {
  "chai": "^5.0.0!,
  "chai-http": "^4.4.0"
  "mocha": "^10.2.0"
  "supertest": "^6.3.4"
  }
}
```

Para poder inicializar el server necesitamos hacer las instalaciones:

```
npm i express mysql pkg
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Este sería el server:

```
// Definir una ruta para crear la tabla
app.get('/crear-tabla'.(req.res) => {
   const createTableQuery =
   'CREATE TABLE productos (
      id INT AUTO_INCREMENT PRIMARY KEY,
      nombre VACHAR(255) NOT NULL,
      precio DECIMAL(10.2) NOT NULL,
      descripcion TEXT


   )'
   ;

   connection.query(createTableQuery. (err.results) => {
   if (err) {
      console,error('Error al crear la tabla', err);
      res.status(500).send('Error al crear la tabka');
   } else {
      console.log('Tabla creada exitosamente');
      res.send('Tabla creada exitosamente');
   }
 });
});
```

```
app.post('/agregar-producto'.(req.res) => {
   const { nombre, precio, descripcion } = req.body;
   const insertQuery =
    INSERT INTO productos (nombre, precio, descripcion)
    VALUES (?, ?, ?)
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

```
';

 connection.query(insertQuery, [nombre, precio, descripcion]. (err,results) => {
 if (err) {
   console.error('Error al agregar producto:', err);
   } else {
   console.log('PRoducto agregado exitosamente');
   res.send('Producto agregado exitosamente');
   }
  });
});
```

```
app.post('/productos'.(req.res) => {
   const selectQuery = 'SELECT ' FROM productos';

   connection.query(selectQuery.(err.results) => {
    if (err) {
      console.error('Error al seleccionar productos');
    } else {
      console.log('Productos seleccionados exitosamente');
      res.json(results);
    }
 });
});

app.listen(PORT.() => {
   console.log(`Servidor Express escuchando en el puerto ${PORT}`);
});
```

Testing

A continuación vamos a realizar el testing, debemos instalar las siguientes librerías:

```
npm i chau supertest chaiHttp
```

En una primera instancia vamos a importar las librerías de la siguiente forma:

```
import * as chai from 'chai';
import app from './index.js';
import supertest from 'supertest';
import chaiHttp from 'chai-http'
```

Debajo vamos a agregar la siguiente línea que será explicada a continuación:

```
chai.use(chaiHttp);
```

Chai.use(chaiHttp) se utiliza para agregar el middleware proporcionado por Chai HTTP a la instancia de Chai, permitiendo así realizar solicitudes HTTP desde las pruebas.

Ahora vamos a analizar la siguiente línea :

```
const request = supertest.agent(app);
```

La línea “const request = supertest.agent(app)” está creando un objeto request utilizando “supertest”. Este objeto se utiliza para realizar solicitudes HTTP a la aplicación Express (app) que estás probando.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Por último vamos a analizar las pruebas “Api Tests”:

```
describe('API Tests', () => {
  // Test para la ruta POST /agregar-producto
  it('Debería agregar un nuevo producto', (done) => {
    const newProduct = {
      nombre: 'Nuevo Producto',
      precio: 19.99,
      descripcion: 'Descripción del nuevo producto'
    };

    request
    .post('/agregar-producto')
    .send(newProduct)
    .end((err, res) => {
      chai.expect(res).to.have.status(200);
      chai.expect(res.text).to.equal(
'Producto agregado exitosamente');
      done();
    });
  });

  // Test para la ruta GET /productos
  it('Debería obtener todos los productos', (done) => {
    request
      .get('/productos')
      .end((err, res) => {
        chai.expect(res).to.have.status(200);
        chai.expect(res.body).to.be.an('array');
        done();
      });
  });
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ahora vamos a ver qué hace cada método para poder comprender qué hace este código:

1. **describe:** Esta palabra clave inicia un bloque de pruebas y se utiliza para describir el conjunto de pruebas que sigue. En este caso, describe un conjunto de pruebas para las API.

2. **it:** Indica un caso de prueba individual dentro del bloque `describe`. Especifica qué debería hacer la prueba y contiene la lógica de la prueba.

3. **request**: Es una instancia de supertest que se utiliza para hacer solicitudes HTTP a la aplicación Express. `supertest` es una biblioteca que facilita las pruebas de integración para aplicaciones web.

4. **post y get:** Son métodos de la instancia `request` y se utilizan para realizar solicitudes POST y GET, respectivamente.

5. **.send(newProduct):** Método encadenado a la solicitud para adjuntar un cuerpo a la solicitud POST. En este caso, envía el objeto `newProduct` como el cuerpo de la solicitud.

6. **.end((err, res) => { ... }):** Método encadenado que indica el final de la solicitud y define una función de devolución de llamada que se ejecuta cuando la solicitud se completa. En esta función, se realizan aserciones sobre la respuesta.

7. **chai.expect(res):** Utiliza la biblioteca de aserciones Chai para realizar afirmaciones sobre la respuesta `res`.

8. **to.have.status(200):** Asegura que la respuesta tenga un código de estado 200.

9. **to.equal('Producto agregado exitosamente'):** Asegura que el cuerpo de la respuesta sea exactamente igual al texto proporcionado.

10. **done():** Una función que se llama al final de la prueba para indicar que la prueba ha finalizado. Es particularmente útil en pruebas asincrónicas.

Todo el código de test quedaría así ⬇️

Buenos Aires aprende
Agencia de Habilidades para el Futuro

```
import * as chai from 'chai';
import app from './index.js';
import supertest from 'supertest';
import chaiHttp from 'chai-http'

chai.use(chaiHttp);

const request = supertest.agent(app);
```

```
describe('API Tests', () => {
    // Test para la ruta POST /agregar-producto
    it('Debería agregar un nuevo producto', (done) => {
        const newProduct = {
            nombre: 'Nuevo Producto',
            precio: 19.99,
            descripcion: 'Descripción del nuevo producto'
        };

        request
        .post('/agregar-producto')
        .send(newProduct)
        .end((err, res) => {
            chai.expect(res).to.have.status(200);
            chai.expect(res.text).to.equal(
  'Producto agregado exitosamente');
            done();
        });
    });

    // Test para la ruta GET /productos
    it('Debería obtener todos los productos', (done) => {
        request
```

```
            .get('/productos')
            .end((err, res) => {
                chai.expect(res).to.have.status(200);
                chai.expect(res.body).to.be.an('array');
                done();
            });
    });
```

¿Qué te pareció el contenido que vimos en la clase de hoy?
¿Usarías estas herramientas de testing como parte de tu proceso de desarrollo?


Buenos Aires
aprende
Buenos
Aires
Ciudad