Bootcamp de Desarrollo Web

Clase 18

Code Whisperer & Unit Testing

1

¿Qué es CodeWhisperer?

Amazon CodeWhisperer es un generador de código de propósito general basado en aprendizaje automático que le proporciona recomendaciones de código en tiempo real.

2

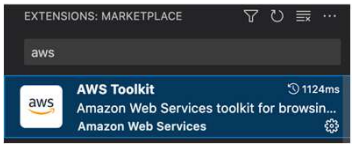
¿Qué es CodeWhisperer?

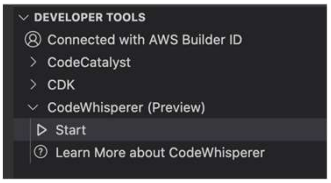
A medida que escribe código, CodeWhisperer genera automáticamente sugerencias basadas en su código y comentarios existentes. Sus recomendaciones personalizadas pueden variar en tamaño y alcance, desde un comentario de una sola línea hasta funciones completamente formadas.

Nota: Actualmente soporta los lenguajes C#, Typescript, Python, Java y Javascript

Instalar CodeWhisperer

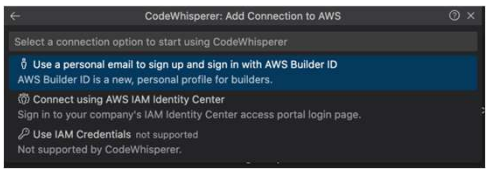
- En el marketplace de extensiones de VS Code busca AWS Toolkit e instálalo
- Una vez instalado ingresa en el menú que aparecerá en la barra principal de VS Code
- Abre el menú Developer Tools, abre CodeWhisperer y presiona en Start

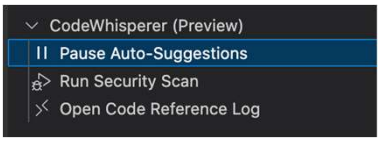




Instalar CodeWhisperer

- Te solicitará conectar una de estas cuentas.
- Una vez conectada la cuenta debería comenzar a correr el **Auto-Suggestions**





5

Ejemplo

- escribe: //crea una función que sume dos variables

```
//crea una función que sume dos variables
const suma = (a, b) => {
    return a + b;
}
//llama a la funci3n suma
const resultado = suma(2, 3);
//muestra el resultado
console.log(resultado);
```


6

Bootcamp de Desarrollo Web

Clase 18

Unit Testing

7

¿Qué es el testing?

El testing es nuestra herramienta de feedback, nos permite detectar los errores en el proceso de desarrollo para poder solventarlos a tiempo. También podemos describirlo como el proceso de verificación y validación de una aplicación.

Hay que tener en cuenta que es un proceso paralelo al de desarrollo, ya que a medida que avanzamos con el proyecto aparecerán errores que corregir.

8

¿Por qué es tan importante el testing?

A la hora de desarrollar una aplicación es importante que se cumplan todos los requisitos solicitados para poder entregar un producto de calidad y fiable.

Hay varios puntos en el proceso de desarrollo que pueden ser críticos a causa del error humano y que pueden llevar a que un software no cumpla los requisitos del cliente. Estos son algunos:

- El cliente no tiene que saber lo necesario para realizar este software y solo se encarga de solicitar lo que necesita o cree que necesita.

¿Por qué es tan importante el testing?

- Las personas que revisan lo solicitado por el cliente pueden malinterpretar o no documentar correctamente la información.
- Durante el desarrollo se pueden introducir errores debidos falta de experiencia, de tiempo, errores humanos, etc.
- Es posible que el cliente no revise el proyecto y lo publique pensando que ha sido testado con anterioridad por los desarrolladores haciendo que se publique una aplicación incompleta y con fallos que se encontrarán los usuarios finales.

Beneficios del testing

Los motivos que consideramos principales para la realización de los **tests** en el proceso de desarrollo los siguientes:

- Intentar encontrar los errores en etapas tempranas del desarrollo que supondría un ahorro de tiempo y costes
- Identificación de errores en cualquier fase de desarrollo
- Ahorro en costes posteriores de mantenimiento.
- Garantizar que un proyecto sea fiable y sea del gusto del cliente.
- Garantizar la calidad del software.
- Ofrecer una visión de calidad y confianza a la hora vender los productos.

Tipos de Pruebas

a. Unit Testing (Pruebas Unitarias)

Las pruebas unitarias se centran en comprobar la funcionalidad de pequeñas unidades de código, como funciones o métodos individuales. Estas pruebas son rápidas y fáciles de escribir, y ayudan a detectar errores en el nivel más bajo del software.

b. Integration Testing (Pruebas de Integración)

Las pruebas de integración verifican que las diferentes unidades de código funcionan bien juntas. Estas pruebas son cruciales para asegurar que los módulos interactúan correctamente y que la combinación de componentes individuales no introduce errores.

Tipos de Pruebas

c. End-to-End Testing (Pruebas de Extremo a Extremo)

Las pruebas de extremo a extremo simulan escenarios completos de usuario, verificando que el sistema como un todo funciona como se espera. Estas pruebas son más lentas y complejas, pero proporcionan una alta confianza en la funcionalidad general del sistema.

d. Functional Testing (Pruebas Funcionales)

Las pruebas funcionales aseguran que el sistema cumple con los requisitos especificados, enfocándose en el output correcto dado un input específico. Estas pruebas se centran en lo que hace el sistema sin considerar cómo lo hace.

Tipos de Pruebas

e. Performance Testing (Pruebas de Rendimiento)

Estas pruebas evalúan cómo se comporta el sistema bajo diferentes cargas de trabajo, identificando problemas de rendimiento, como cuellos de botella y tiempos de respuesta lentos.

Herramientas Populares para Testing en JavaScript

a. Jest

Desarrollado por Facebook, Jest es un framework de testing completo y fácil de configurar. Es particularmente popular en proyectos de React.

b. Mocha

Mocha es un framework de pruebas flexible y extenso. Se suele usar junto con otras librerías como Chai para aserciones y Sinon para mocks y stubs.

c. Jasmine

Jasmine es un framework de testing behavior-driven development (BDD) que no depende de otras herramientas.

Herramientas Populares para Testing en JavaScript

d. Karma

Karma es un test runner que permite ejecutar pruebas en múltiples navegadores simultáneamente.

e. Cypress

Cypress es una herramienta para pruebas end-to-end moderna y robusta que proporciona una experiencia de usuario intuitiva.

Conceptos Clave en Testing

a. Assertions (Aserciones)

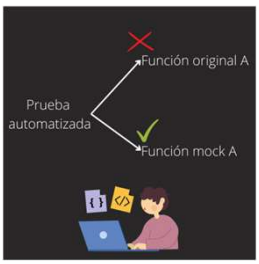
Las aserciones son afirmaciones que verifican si un valor es el esperado. Si la afirmación resulta falsa, la prueba falla.

```
$ jest
 FAIL  test/specs/urlParser.spec.js
  ● urlParser › returns a formatted URL

    expect(received).toBe(expected)

    Expected value to be (using ===):
      true
    Received:
      false

      at Object.<anonymous> (test/specs/urlParser.spec.js:7:19)
          at Promise (<anonymous>)
          at <anonymous>
      at process._tickCallback (internal/process/next_tick.js:169:7)

urlParser
  ✕ returns a formatted URL (4ms)
```


Conceptos Clave en Testing

b. Mocks and Stubs (Mocks y Stubs)

Mocks y stubs son técnicas para simular partes del código que interactúan con componentes externos, como servicios de red o bases de datos. Esto permite probar una unidad de código en aislamiento.



Conceptos Clave en Testing

c. Test Suites

Una test suite es una colección de pruebas relacionadas. Organizar las pruebas en suites ayuda a mantener el orden y la claridad en el proceso de testing.

```
PS npm test

> learning-jest@1.0.0 test
> jest

 PASS  ./index.test.js
  FizzBuzz
    ✓ [3] should result in "fizz" (2 ms)
    ✓ [5] should result in "buzz" (1 ms)
    ✓ [15] should result in "fizzbuzz" (1 ms)
    ✓ [1,2,3] should result in "1, 2, fizz" (1 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        0.586 s, estimated 1 s
Ran all test suites.
```


Conceptos Clave en Testing

e. Code Coverage (Cobertura de Código)

La cobertura de código mide qué porcentaje del código ha sido ejecutado durante las pruebas. Herramientas como Istanbul (y su integración con Jest) ayudan a visualizar la cobertura y a identificar áreas no cubiertas por las pruebas.

```
 PASS  __tests__/sum.spec.js
  Sum
    ✓ sum two numbers (1 ms)

-------------|---------|----------|---------|---------|-------------------
File         | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------|---------|----------|---------|---------|-------------------
All files    |      50 |      100 |      50 |      50 |
 subtract.js |       0 |      100 |       0 |       0 | 1-2
 sum.js      |     100 |      100 |     100 |     100 |
-------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.42 s, estimated 1 s
Ran all test suites.
```

Buenas Prácticas en Testing

- Escribir Pruebas Pequeñas y Aisladas: Mantén las pruebas unitarias pequeñas y enfocadas en una sola responsabilidad.
- Nombrar las Pruebas Claramente: Usa nombres descriptivos para las pruebas para entender fácilmente qué están verificando.
- Ejecutar las Pruebas Frecuentemente: Corre las pruebas con regularidad para detectar errores rápidamente.
- Automatización del Testing: Usa herramientas de CI/CD para ejecutar automáticamente las pruebas en cada cambio de código.
- Mantener la Independencia de las Pruebas: Asegúrate de que las pruebas no dependan unas de otras.

Bootcamp de Desarrollo Web

Clase 18

CodeWhisperer & Unit Testing