Pruebas unitarias

Tras analizar las ventajas de la automatización de pruebas, vamos a ver xUnit en acción.

Para ello vamos a crear un proyecto consistente en una librería de consulta de precio, en moneda fiduciaria (fiat) de cryptomonedas.

La implementación es lo de menos, lo importante es cómo realizaremos pruebas automáticas y cómo se puede mockear dependencias y configurar comportamiento de esos mocks, y así poder probar escenarios de forma determinista.

crypto-fiat-converter

Vamos a trabajar sobre un ejemplo sencillo, un conversor de cryptomoneda a EUR. Clona el repositorio

```
git clone https://gitlab.com/sunnyatticsoftware/lemoncode/crypto-fiat-converter.git
```

Observa que el proyecto consiste en una librería que soporta actualización de tipos de cambio, por un lado, y una calculadora-conversor de criptomoneda a EUR por otro lado.

Para simplificar, solamente 3 criptomonedas están soportadas: HBAR, BTC y ETH, y el resultado siempre está en EUR.

La forma de utilizarla, como cliente, es instanciando `Converter` y llamando a su único método. Por ejemplo:

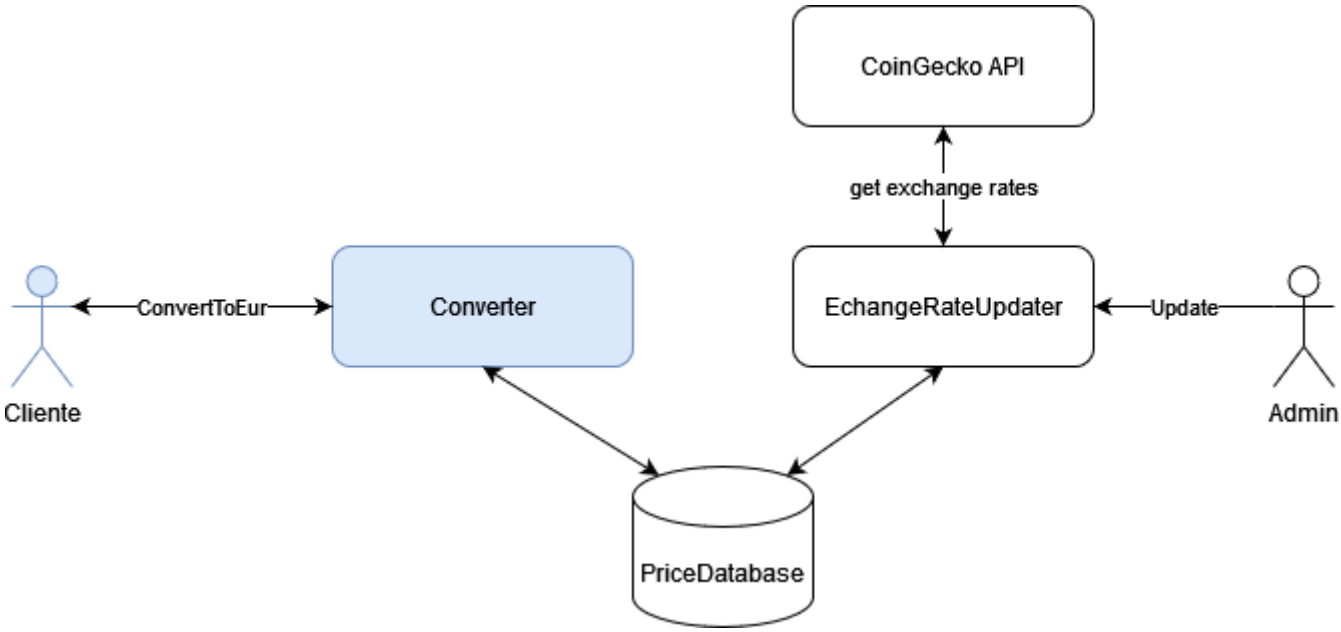
```
IPriceDatabase priceDatabase = new InMemoryPriceDatabase(); // base de datos en memoria, por ahora
var converter = new Converter(priceDatabase);
var result = converter.ConvertToEur("BTC", 2.5);
```

Hay un aspecto muy importante. Este conversor depende de una base de datos de precios, de donde obtendrá el tipo de cambio actual para realizar el cálculo. Esta base de datos está abstraída por la interfaz `IPriceDatabase`, y, por lo tanto, puede tener una implementación simple en memoria (la que utilizamos), o una implementación más compleja que utilice una base de datos real, relacional o documental. Para simplificar, nos sirve con la implementación en memoria, pero ¡ojo!, esta base de datos en memoria debe tener una única instancia.

La base de datos de precios se actualiza por otro lado gracias a una interfaz `IExchangeRateUpdater`. Hay una implementación de esta interfaz que envía peticiones http a una api externa, CoinGecko, para obtener el tipo de cambio actual y actualizar la base de datos de precios. Es decir, cachea la información de CoinGecko cuando deseemos hacerlo.

De nuevo, es importante destacar que la base de datos de precios, representada por la interfaz `IPriceDatabase`, es un recurso compartido entre el `Converter` y el `ExchangeRateUpdater` como se

muestra en esta imagen.



El comportamientod de la base de datos de precios lo define el contrato

```
public interface IPriceDatabase
{
    CurrentPrice GetPrice(string cryptoCode);

    void SetAllPrices(IDictionary<string, decimal> prices);
}
```

Como se aprecia, esta base de datos utiliza un modelo inmutable (las propiedades solamente se pueden setear al instanciar, pero luego son de solo lectura) llamado `CurrentPrice` para guardar información sobre el tipo de cambio de cada criptomoneda en EUR y sobre la fecha en la que se ha actualizado ese tipo de cambio.

```
public class CurrentPrice
{
    public string CryptoCode { get; }
    public DateTime LastUpdatedOn { get; }
    public decimal PriceInEur { get; }

    public CurrentPrice(string cryptoCode, DateTime lastUpdatedOn, decimal
priceInEur)
    {
        CryptoCode = cryptoCode;
        LastUpdatedOn = lastUpdatedOn;
        PriceInEur = priceInEur;
    }
}
```

Y como primera implementación, se ha optado por una base de datos en memoria, por simplicidad.

```
public class InMemoryPriceDatabase
    : IPriceDatabase
{
    private readonly IDictionary<string, decimal> _prices = new Dictionary<string,
decimal>();
    private DateTime _lastUpdatedOn = DateTime.MinValue;

    public CurrentPrice GetPrice(string cryptoCode)
    {
        var isStored = _prices.TryGetValue(cryptoCode, out var price);
        if(!isStored)
        {
            throw new KeyNotFoundException($"Key {cryptoCode} does not exist");
        }

        var currentPrice = new CurrentPrice(cryptoCode, _lastUpdatedOn, price);
        return currentPrice;
    }

    public void SetAllPrices(IDictionary<string, decimal> prices)
    {
        foreach (var newPrice in prices)
        {
            _prices[newPrice.Key] = newPrice.Value;
        }

        _lastUpdatedOn = DateTime.UtcNow;
    }
}
```

Por otro lado, el comportamiento del actualizador de tipo de cambio se define así

```
public interface IExchangeRateUpdater
{
    Task Update();
}
```

Esa es la acción que un administrador, o una tarea programada, deberá ejecutar para que se actualice la base de datos de precios con los tipos de cambios actuales.

Hay una implementación que es capaz de extraer los tipos de cambio de una API externa, la de CoinGecko, y modificar con esos valores la base de datos de precio.

```
public class CoinGeckoExchangeRateUpdater
    : IExchangeRateUpdater
{
    private readonly IPriceDatabase _priceDatabase;
```

```
    public CoinGeckoExchangeRateUpdater(IPriceDatabase priceDatabase)
    {
        _priceDatabase = priceDatabase;
    }

    public async Task Update()
    {
        var httpClient = new HttpClient();

        var commaSeparatedCryptoIds = string.Join(",",
SupportedCrypto.GetAllIds());

        var response = await httpClient.GetAsync(
            $"https://api.coingecko.com/api/v3/simple/price?ids=
{commaSeparatedCryptoIds}&vs_currencies=EUR");

        var json = await response.Content.ReadAsStringAsync();

        var rates = JsonSerializer.Deserialize<Dictionary<string,
Dictionary<string, decimal>>>(json);
        if (rates is null)
        {
            throw new ApplicationException("Could not retrieve rates from
CoinGecko");
        }

        var allPrices = new Dictionary<string, decimal>();

        foreach (var rate in rates)
        {
            var cryptoId = rate.Key;
            var cryptoCode = SupportedCrypto.GetCryptoCode(cryptoId);
            var priceEur = rate.Value["eur"];
            allPrices.Add(cryptoCode, priceEur);
        }

        _priceDatabase.SetAllPrices(allPrices);
    }
}
```

Por simplicidad, también, se ha optado por soportar solamente HBAR, BTC, ETH. La siguiente clase estática contiene el mapeo entre el código de cada ciptomoneda y el identificador que utiliza la API de CoinGecko.

```
internal static class SupportedCrypto
{
    private static readonly IDictionary<string, string> SupportedCoins
        = new Dictionary<string, string>
        {
            { "HBAR", "hedera-hashgraph" },
            { "BTC", "bitcoin" },
            { "ETH", "ethereum" },
```

```
        };

    public static bool IsSupported(string cryptoCode)
    {
        return SupportedCoins.ContainsKey(cryptoCode);
    }

    public static string GetCryptoCode(string id)
    {
        var code = SupportedCoins.First(x => x.Value == id).Key;
        return code;
    }

    public static IEnumerable<string> GetAllIds()
    {
        return SupportedCoins.Values.ToArray();
    }
}
```

Finalmente, la clase Converter contiene la funcionalidad que queremos exponer al mundo exterior con nuestra librería.

```
public class Converter
{
    private readonly IPriceDatabase _priceDatabase;

    public Converter(IPriceDatabase priceDatabase)
    {
        _priceDatabase = priceDatabase;
    }

    public ConversionResult ConvertToEur(string cryptoCode, decimal amount)
    {
        var isSupported = SupportedCrypto.IsSupported(cryptoCode);
        if (!isSupported)
        {
            throw new ArgumentException($"Crypto code {cryptoCode} is not
supported");
        }

        var currentPrice = _priceDatabase.GetPrice(cryptoCode);

        var total = amount * currentPrice.PriceInEur;
        var result = new ConversionResult(currentPrice.CryptoCode,
currentPrice.LastUpdatedOn, total);
        return result;
    }
}
```

Como se aprecia, esta clase simplemente depende de una base de datos, de donde obtiene los tipos de cambio cacheados, y tiene una simple función para convertir la cantidad de la criptomoneda indicada en EUR.

El resultado está modelado por una clase inmutable que contiene toda la información que le interesa al usuario

```
public sealed class ConversionResult
{
    public string CryptoCode { get; }
    public DateTime RateUpdatedOn { get; }
    public decimal TotalEur { get; }

    public ConversionResult(string cryptoCode, DateTime rateUpdatedOn, decimal
totalEur)
    {
        CryptoCode = cryptoCode;
        RateUpdatedOn = rateUpdatedOn;
        TotalEur = totalEur;
    }
}
```

Vamos a comenzar con las pruebas unitarias y a explorar los diferentes escenarios, tanto funcionales como técnicos, que nos iremos encontrando.

Un test muy sencillo podría ser el de testear el constructor (que también es un método) de la clase `ConversionResult` y analizar si los valores con los que instanciamos objetos de esta clase son los valores que esperamos poder leer después.

```
public class CtorTest
{
    [Fact]
    public void Given_An_Instance_The_Property_Values_Are_As_Expected()
    {
        // Given
        var cryptoCode = "BTC";
        var rateUpdatedOn = new DateTime(2021, 11, 3, 0, 0, 0);
        var totalEur = 123.456m;

        // When
        var sut = new ConversionResult(cryptoCode, rateUpdatedOn, totalEur);

        // Then
        Assert.Equal(sut.CryptoCode, cryptoCode);
        Assert.Equal(sut.RateUpdatedOn, rateUpdatedOn);
        Assert.Equal(sut.TotalEur, totalEur);
    }
}
```

Si ejecutamos este test, vemos que las 3 comprobaciones son correctas y que los valores que estamos pasando al constructor son los mismos que estamos obteniendo cuando leemos las propiedades.

Esta forma de comprobar resultados, es la que xUnit nos facilita, pero hay muchas librerías adicionales que podemos utilizar para mejorar la legibilidad de nuestros tests. Probemos a añadir el paquete nuget llamado `FluentAssertions` y modifiquemos el test.

```
[Fact]
public void Given_An_Instance_The_Property_Values_Are_The_Expected()
{
    // Given
    var cryptoCode = "BTC";
    var rateUpdatedOn = new DateTime(2021, 11, 3, 0, 0, 0);
    var totalEur = 123.456m;

    // When
    var sut = new ConversionResult(cryptoCode, rateUpdatedOn, totalEur);

    // Then
    sut.CryptoCode.Should().Be(cryptoCode);
    sut.RateUpdatedOn.Should().Be(rateUpdatedOn);
    sut.TotalEur.Should().Be(totalEur);
}
```

Veamos cómo se puede testear el método `GetPrice(string cryptoCode)` de `InMemoryPriceDatabase`. Queremos comprobar dos escenarios. El primero es que si no hay información sobre el código de la criptomoneda, debe lanzar una excepción `KeyNotFoundException`. El segundo es que si hay información, debe devolver el `CurrentPrice` esperado.

```
public class GetPriceTests
{
    [Fact]
    public void
Given_Non_Existent_Crypto_Code_When_Getting_Price_Throws_KeyNotFoundException()
    {
        // Given
        var sut = new InMemoryPriceDatabase();
        KeyNotFoundException exception = null;

        // When
        try
        {
            sut.GetPrice("BTC");
        }
        catch (KeyNotFoundException keyNotFoundException)
        {
            exception = keyNotFoundException;
        }

        // Then
```

```
            exception.Should().NotBeNull();
        }
    }
```

En el segundo test, podemos comprobar que el código y el precio son los esperados

```
[Fact]
public void
Given_Existent_Crypto_Code_When_Getting_Price_Returns_Expected_Current_Price_Detai
ls()
{
    // Given
    var sut = new InMemoryPriceDatabase();
    sut.SetAllPrices(
        new Dictionary<string, decimal>
        {
            {"BTC", 0.1m}
        });

    // When
    var result = sut.GetPrice("BTC");

    // Then
    result.CryptoCode.Should().Be("BTC");
    result.PriceInEur.Should().Be(0.1m);
    // result.LastUpdatedOn.Should().BeAfter(new DateTime(2021, 09, 1, 0, 0, 0));
// ¡esto no es determinista!
}
```

Pero, ¿qué ocurre con la última fecha de actualización de `CurrentPrice`? ¿Podemos comprobar que el valor que devuelve es el esperado? No. No podemos, porque esa fecha va a cambiar constantemente cada vez que ejecutemos la funcionalidad, ya que hay una instanciación en el código, un `new` implícito que hace

```
_lastUpdatedOn = DateTime.UtcNow;
```

cada vez que se establecen los precios.

Los test unitarios deben ser deterministas. Deben producir el mismo resultado cada vez que se ejecuten con los mismos pre-requisitos. Por lo tanto, cualquier instanciación en el código, o cualquier variable como una fecha que puede ser diferente cada vez que se ejecuta un test, supondrá un problema para la automatización de pruebas. Además, los test unitarios nos dan *feedback* de nuestro código, y el *feedback* ahora mismo nos informa de que hay cosas mejorables.

Si abrazamos la inyección de dependencias, podremos inyectar una factoría cuya única responsabilidad sea la de recuperar la fecha actual. Al ser un contrato (interfaz o clase abstracta), podremos crear un doble (i.e: crear un *mock* o *mockear*) y configurarlo para que devuelva exactamente lo que le digamos.

Vamos a crear un `IDateTimeFactory`

```
public interface IDateTimeFactory
{
    DateTime GetCurrentUtc();
}
```

Y una implementación concreta

```
public class DateTimeFactory
    : IDateTimeFactory
{
    public DateTime GetCurrentUtc()
    {
        return DateTime.UtcNow;
    }
}
```

Ahora, el `InMemoryPriceDatabase` ya no necesita instanciar una fecha, sino que puede utilizar esta factoría/servicio externo, del cual depende, para obtenerlo. Vamos a inyectarlo.

```
public class InMemoryPriceDatabase
    : IPriceDatabase
{
    private readonly IDateTimeFactory _dateTimeFactory;
    private readonly IDictionary<string, decimal> _prices = new Dictionary<string,
decimal>();
    private DateTime _lastUpdatedOn = DateTime.MinValue;

    public InMemoryPriceDatabase(IDateTimeFactory dateTimeFactory)
    {
        _dateTimeFactory = dateTimeFactory;
    }


    public CurrentPrice GetPrice(string cryptoCode)
    {
        var isStored = _prices.TryGetValue(cryptoCode, out var price);
        if(!isStored)
        {
            throw new KeyNotFoundException($"Key {cryptoCode} does not exist");
        }

        var currentPrice = new CurrentPrice(cryptoCode, _lastUpdatedOn, price);
        return currentPrice;
    }

    public void SetAllPrices(IDictionary<string, decimal> prices)
```

```
    {
        foreach (var newPrice in prices)
        {
            _prices[newPrice.Key] = newPrice.Value;
        }

        _lastUpdatedOn = _dateTimeFactory.GetCurrentUtc();
    }
}
```

Ya podemos tener control sobre esa factoría en nuestros tests. Pero, un momento, antes de continuar con ese problema debemos atajar otro problema que acabamos de introducir. Hemos modificado la firma del constructor de `InMemoryPriceDatabase` para que ahora acepte un `IDateTimeFactory`, pero ahora todas las instanciaciones que teníamos de `InMemoryPriceDatabase` se han roto y no compilan.

Por suerte solamente tenemos un par de tests por arreglar, pero imaginemos que tenemos varias decenas. ¿Se nos ocurre alguna forma de desacoplarnos de la instanciación de algo y centralizarlo en un punto, de modo que para instanciar `InMemoryPriceDatabase` podamos delegar en alguna especie de servicio que podamos cambiar sin afectar a cada uno de los tests? Aunque no lo vamos a tratar aquí, una pista sería: el patrón builder

```
public class InMemoryPriceDatabaseBuilder
{
    private IDateTimeFactory _dateTimeFactory;

    public InMemoryPriceDatabaseBuilder()
    {
        _dateTimeFactory = new DateTimeFactory();
    }

    public InMemoryPriceDatabaseBuilder WithDateTimeFactory(IDateTimeFactory
dateTimeFactory)
    {
        _dateTimeFactory = dateTimeFactory;
        return this;
    }

    public InMemoryPriceDatabase Build()
    {
        var result = new InMemoryPriceDatabase(_dateTimeFactory);
        return result;
    }
}
```

Volviendo al problema principal, ya tenemos abstraída la funcionalidad para obtener la fecha actual. Podemos hacer un mock que devuelva siempre la misma fecha, para conseguir determinismo en nuestros tests.

```
public class DateTimeManualMock
    : IDateTimeFactory
{
    public DateTime GetCurrentUtc()
    {
        return new DateTime(2021, 09, 1, 0, 0, 0, DateTimeKind.Utc);
    }
}
```

Es más, lo podemos modificar para no hardcodear una fecha en concreto sino para que utilice la fecha que le indiquemos al instanciarlo, haciéndolo mucho más útil y respetando la interfaz que implementa para que siga comportándose como algo que devuelve fechas en utc.

```
public class DateTimeManualMock
    : IDateTimeFactory
{
    private readonly DateTime _dateTimeToReturn;

    public DateTimeManualMock(DateTime dateTimeToReturn)
    {
        if (dateTimeToReturn.Kind != DateTimeKind.Utc)
        {
            throw new ArgumentException("Only UTC DateTime allowed on this mock");
        }
        _dateTimeToReturn = dateTimeToReturn;
    }

    public DateTime GetCurrentUtc()
    {
        return _dateTimeToReturn;
    }
}
```

Y ahora podemos arreglar las instanciaciones de `InMemoryPriceDatabase` para inyectar un `IDateTimeFactory`. La clave está en que no utilizaremos la implementación real de `IDateTimeFactory` (e.g: `DateTimeFactory`), sino un doble, un mock, un `DateTimeManualMock`. Como ya tenemos control sobre la fecha y hora "actual", podemos programar el mock para que nos devuelva una fecha en concreto y evaluar que el `InMemoryPriceDatabase` está asignando esa misma fecha como la del tipo de cambio. Problema resuelto.

```
public class GetPriceTests
{
    [Fact]
    public void
Given_Non_Existent_Crypto_Code_When_Getting_Price_Throws_KeyNotFoundException()
    {
        // Given
```

```
        var sampleDateTime = new DateTime(2021, 10, 1, 0, 0, 0, DateTimeKind.Utc);
        var sut = new InMemoryPriceDatabase(new
DateTimeManualMock(sampleDateTime));
        KeyNotFoundException exception = null;

        // When
        try
        {
            sut.GetPrice("BTC");
        }
        catch (KeyNotFoundException keyNotFoundException)
        {
            exception = keyNotFoundException;
        }

        // Then
        exception.Should().NotBeNull();
    }

    [Fact]
    public void
Given_Existent_Crypto_Code_When_Getting_Price_Returns_Expected_Current_Price_Detai
ls()
    {
        // Given
        var sampleDateTime = new DateTime(2021, 10, 1, 0, 0, 0, DateTimeKind.Utc);
        var sut = new InMemoryPriceDatabase(new
DateTimeManualMock(sampleDateTime));
        sut.SetAllPrices(
            new Dictionary<string, decimal>
            {
                {"BTC", 0.1m}
            });

        // When
        var result = sut.GetPrice("BTC");

        // Then
        result.CryptoCode.Should().Be("BTC");
        result.PriceInEur.Should().Be(0.1m);
        result.LastUpdatedOn.Should().BeSameDateAs(sampleDateTime);
    }
}
```

Este es un doble muy sencillo, pero a pesar de ello hemos tenido que crear una clase con un par de métodos y demasiado *boilerplate*. Existen librerías que podemos utilizar precisamente para generar dobles o *mocks*. La más utilizada en .NET es `Moq`. Vamos a verla.

Instalamos la dependencia `Moq`.

El mismo test nos quedaría de la siguiente forma utilizando esta librería.

```
[Fact]
public void
Given_Existent_Crypto_Code_When_Getting_Price_Returns_Expected_Current_Price_Detai
ls_With_Moq()
{
    // Given
    var sampleDateTime = new DateTime(2021, 10, 1, 0, 0, 0, DateTimeKind.Utc);

    var dateTimeFactoryMock = new Mock<IDateTimeFactory>();
    dateTimeFactoryMock
        .Setup(x => x.GetCurrentUtc())
        .Returns(sampleDateTime);
    var dateTimeFactory = dateTimeFactoryMock.Object;

    var sut = new InMemoryPriceDatabase(dateTimeFactory);
    sut.SetAllPrices(
        new Dictionary<string, decimal>
        {
            {"BTC", 0.1m}
        });

    // When
    var result = sut.GetPrice("BTC");

    // Then
    result.CryptoCode.Should().Be("BTC");
    result.PriceInEur.Should().Be(0.1m);
    result.LastUpdatedOn.Should().BeSameDateAs(sampleDateTime);
}
```

Por cierto, estamos evaluando cada una de las propiedades del resultado, pero podríamos evaluar todo el resultado de una sola vez. Por ejemplo comprobando que el resultado es equivalente a una instancia en concreto.

```
// Assert the result at once
var expectedResult = new CurrentPrice("BTC", sampleDateTime, 0.1m);
result.Should().BeEquivalentTo(expectedResult);
```

Vamos ahora a ampliar el tamaño de la unidad en nuestros "unit" tests. Algunos hablarían de test de integración, o de componente. El nombre no es importante. Lo importante es que queremos ampliar el test de caja negra para abarcar una unidad más amplia y acercarnos un poco más a un punto de vista de caso de uso de negocio.

Queremos probar que el siguiente escenario funciona como se espera. Dadas dos actualizaciones del tipo de cambio, en las que la criptomoneda HBAR está, en un primer momento, a 0.30 EUR y, en la siguiente actualización, a 0.50 EUR cuando convertimos 10 HBAR a Euro nos deberá devolver la cantidad de 5 EUR y la fecha de actualización esperada.

¿Sabríamos comprobar ese escenario? ¿Utilizaríamos algún Mock? ¿Cuáles? Sería interesante ver qué desarrolladores optarían por utilizar algún mock y discutir las ventajas y desventajas de cada propuesta diferente.

Si supiéramos que cada vez que llamamos a la API de CoinGecko para obtener los tipos de cambio más actuales, CoinGecko nos cobra dinero, ¿decidiríamos utilizar la API real de CoinGecko o la mockearíamos? ¿Nos importa realmente para nuestras pruebas de funcionalidad que el tipo de cambio sea realista? Y, si la API de CoinGecko estuviese caída justo en el momento en el que lanzamos nuestras pruebas, ¿significaría que nuestra librería no funciona correctamente y evitaríamos, por ejemplo, lanzar una nueva versión al mercado solamente por ese hecho?

Todas estas preguntas son muy relevantes y son dilemas a los que nos enfrentamos continuamente como programadores. Es importante tener siempre muy presente los sacrificios que hacemos y las ventajas de una solución sobre otra.

Una solución sensata es la de buscar siempre el determinismo en los test y, por lo tanto, evitar depender de terceros y *mockear* esa funcionalidad si no la podemos controlar.

```
public class ConvertToEurTests
{
    [Fact]
    public void
Given_Two_Exchange_Updates_When_Converting_Hbar_To_Eur_It_Uses_The_Second_Update_A
nd_Returns_Expected_Values()
    {
        // Given
        var firstUpdatedOn = new DateTime(2021, 10, 1, 0, 0, 0, DateTimeKind.Utc);
        var secondUpdatedOn = firstUpdatedOn.AddMinutes(1);

        var dateTimeFactoryMock = new Mock<IDateTimeFactory>();
        dateTimeFactoryMock
            .SetupSequence(x => x.GetCurrentUtc())
            .Returns(firstUpdatedOn)
            .Returns(secondUpdatedOn);
        var dateTimeFactory = dateTimeFactoryMock.Object;

        var priceDatabase = new InMemoryPriceDatabase(dateTimeFactory);

        priceDatabase.SetAllPrices(
            new Dictionary<string, decimal>
            {
                { "BTC", 40000.5m },
                { "ETH", 3000.5m },
                { "HBAR", 0.30m }
            });

        priceDatabase.SetAllPrices(
            new Dictionary<string, decimal>
            {
                { "BTC", 28000.0m },
                { "ETH", 2500.5m },
```

```
                { "HBAR", 0.50m }
            });

        var sut = new Converter(priceDatabase);

        var expectedResult = new ConversionResult("HBAR", secondUpdatedOn, 5);

        // When
        var result = sut.ConvertToEur("HBAR", 10);

        // Then
        result.Should().BeEquivalentTo(expectedResult);
    }
}
```

Como observación final, ¿qué validación de las dos siguientes nos aporta mayor valor en un test?

Opción 1, validar las propiedades de un resultado una a una:

```
result.CryptoCode.Should().Be("HBAR");
result.LastUpdatedOn.Should().BeSameDateAs(secondUpdatedOn);
result.TotalEur.Should().Be(5);
```

Opción 2, validar todo el resultado de una vez comparando equivalencia o igualdad con el objeto esperado.

```
result.Should().BeEquivalentTo(expectedResult);
```

La segunda opción es más determinista. Pensemos en un escenario donde tengamos que añadir en el resultado una propiedad `TotalUsd` y, por alguna razón, se nos olvidara setearlo en la funcionalidad y estuviese devolviendo siempre el valor por defecto `0.0`. ¿Cual de las dos opciones pensamos que nos ayudaría más a detectar ese error/bug?