**[8- Utilizando Moq](https://github.com/ingsoft3ucc/TPs/blob/master/trabajos/09-pruebas-unitarias.md" \l "8--utilizando-moq)**

*Intro*

Moq es un marco de pruebas y simulación (mocking framework) para el lenguaje de programación C# en el entorno de desarrollo de .NET. Permite a los desarrolladores crear objetos simulados, llamados "mocks" o "stubs", para simular el comportamiento de componentes del sistema durante las pruebas unitarias.

Algunas de las características y ventajas clave de Moq incluyen:

Sintaxis Fluent: Moq utiliza una sintaxis fluent y expresiva que facilita la creación y configuración de objetos simulados. Esto hace que las pruebas sean más legibles y mantenibles.

Generación Dinámica: Moq genera objetos simulados en tiempo de ejecución, lo que significa que no es necesario escribir clases separadas para implementar mocks. Esto ahorra tiempo y reduce la complejidad del código de prueba.

Configuración de Comportamiento: Puedes configurar cómo debe comportarse un objeto simulado cuando se llama a sus métodos o propiedades. Esto incluye especificar los valores de retorno, establecer acciones personalizadas y verificar si se han llamado métodos específicos.

Verificación de Llamadas: Moq permite verificar si se han llamado los métodos simulados y cuántas veces se han llamado. Esto es útil para asegurarse de que el código bajo prueba interactúa correctamente con sus dependencias simuladas.

Soporte para Pruebas Parametrizadas: Moq es compatible con pruebas parametrizadas, lo que significa que puedes ejecutar la misma prueba con múltiples conjuntos de datos o escenarios, cambiando la configuración de los mocks según sea necesario.

Integración con Marcos de Pruebas: Moq se integra bien con marcos de pruebas populares como NUnit y xUnit.NET, lo que facilita la incorporación de mocks en tus pruebas unitarias.

Ligero y de Código Abierto: Moq es una biblioteca de código abierto y liviana que no agrega una sobrecarga significativa a tu proyecto.

En resumen, Moq es una herramienta valiosa para escribir pruebas unitarias efectivas en C#. Permite a los desarrolladores crear mocks de manera rápida y sencilla para simular el comportamiento de las dependencias y componentes externos, lo que facilita la prueba aislada de unidades de código y la identificación de problemas en el código durante el desarrollo.

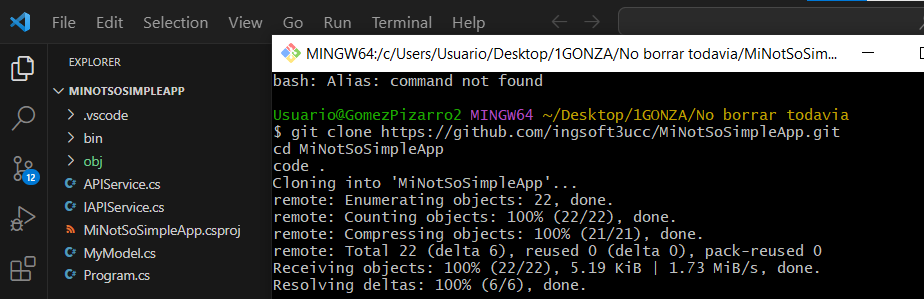
8.1 Preparamos el entorno.

* Clonamos una app de consola en .NET Core que hace uso de un servicio externo (una llamada a una API Rest) y la abrimos en VS.Code

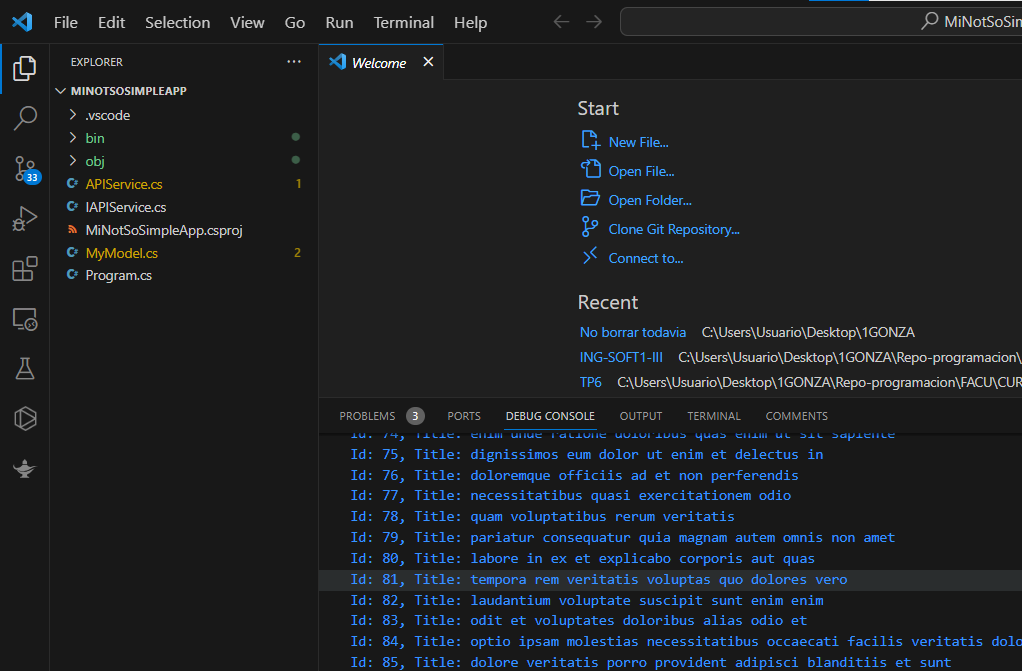
git clone https://github.com/ingsoft3ucc/MiNotSoSimpleApp.git

cd MiNotSoSimpleApp

code .



* Ejecutamos la app y vemos como nos devuelve 100 items desde la API:



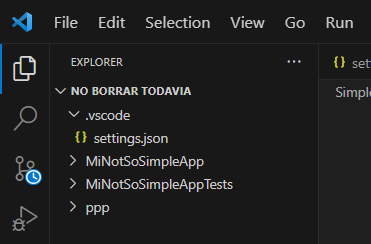
* Analizamos el código

Este Codigo realiza lo siguiente:

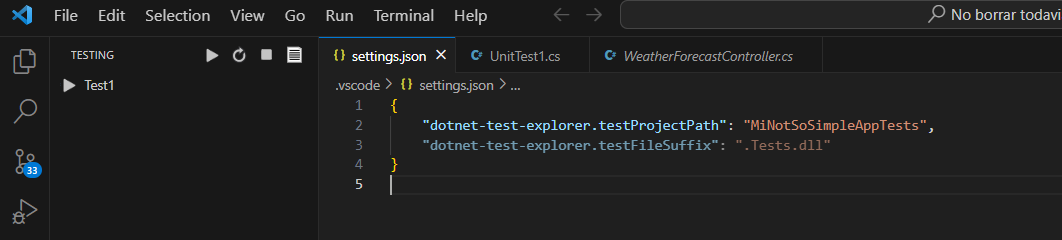
1. Importa las bibliotecas necesarias para trabajar con solicitudes HTTP y servicios de inyección de dependencias.
2. Define una clase llamada Program.
3. En el método Main, crea un proveedor de servicios (serviceProvider) que se utiliza para administrar las dependencias del programa.
4. Registra un cliente HTTP en el proveedor de servicios mediante AddHttpClient. Esto permite realizar solicitudes HTTP.
5. Registra una clase llamada ApiService como un servicio transitorio en el proveedor de servicios. Esto sugiere que ApiService se utilizará para realizar tareas específicas en la aplicación.
6. Obtiene una instancia de ApiService del proveedor de servicios.
7. Llama al método GetMyModelsAsync de ApiService para obtener una colección de modelos llamada myModels.
8. Itera a través de los elementos en myModels y muestra la información de cada modelo en la consola.

En resumen, este programa utiliza inyección de dependencias para gestionar las dependencias de servicios, incluido un cliente HTTP para realizar solicitudes web. Luego, utiliza el servicio ApiService para obtener y mostrar información de modelos desde una fuente externa, probablemente una API web.

* Identificamos el servicio externo y su interfaz a mockear. **Incluir en la entrega una explicación del código**
* Cerramos VS.Code y creamos el proyecto de NUnit



* Abrimos VS.Code y configuramos settings:
* Seleccionamos el icono de Test y OpenSettings Se abre el archivo settings.json y escribimos el nombre del proyecto de pruebas y el nombre de la dll resultante de la compilación del proyecto de pruebas:
* Guardamos el archivo.



8.2 Escribimos Test:

Reemplazamos el codigo de UnitTest1.cs por

8.3 Incluir en la entrega una explicación del código, centrado en explicar cómo se reemplaza el servicio real por el mock, que líneas de nuestro código de test se encargan de hacerlo y cómo funciona el mecanismo.

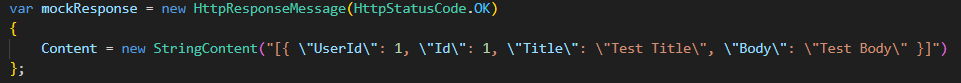
La clave de esta prueba radica en el uso de un objeto "mock" para reemplazar el servicio real de **HttpClient** y simular las respuestas HTTP sin realizar llamadas reales a una API web. A continuación, se explica cómo se reemplaza el servicio real por el mock y cómo funciona este mecanismo:

1. **Mocking el servicio HttpClient**:



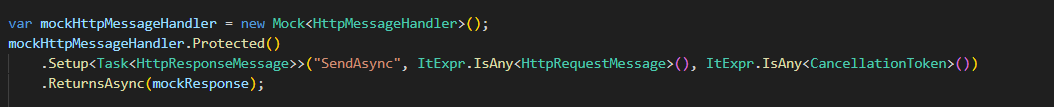
* + Se crea una instancia de **ServiceCollection** para configurar la inyección de dependencias en la prueba.

1. **Simulando la Respuesta HTTP**:



* + Se crea un objeto **mockResponse** de **HttpResponseMessage** que simula una respuesta HTTP exitosa. Esta respuesta incluye un cuerpo JSON que contiene un modelo de datos de ejemplo.

1. **Configurando el Mock del Manejador de Mensajes HTTP**:



* + Se crea un mock del manejador de mensajes HTTP utilizando **Mock<HttpMessageHandler>**. Esto permite interceptar las llamadas HTTP realizadas por el cliente HttpClient.
  + Se configura el mock del manejador de mensajes utilizando el método **Protected().Setup<Task<HttpResponseMessage>>("SendAsync", ...)**. Esto establece que cuando se llame al método **SendAsync** del cliente HttpClient (que se utiliza internamente en **ApiService**), en lugar de realizar una solicitud real, se devolverá la respuesta simulada **mockResponse**.

1. **Configurando la Inyección de Dependencias**:



* + Se agrega **ApiService** como un servicio transitorio en **serviceCollection**, pero en lugar de usar una instancia real de **HttpClient**, se le pasa una instancia que utiliza el mock del manejador de mensajes HTTP creado anteriormente.

1. **Creando el Proveedor de Servicios**:
   * Se crea un proveedor de servicios (**serviceProvider**) a partir de la colección de servicios configurada. Esto es esencial para obtener una instancia de **ApiService** más adelante.
2. **Obteniendo una Instancia de ApiService**:



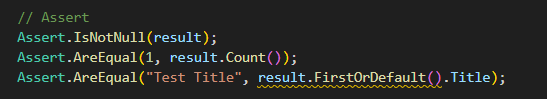
* + Se obtiene una instancia de **IApiService** del proveedor de servicios. Dado que la configuración ahora utiliza el mock del cliente HttpClient, cuando se realicen solicitudes HTTP dentro de **ApiService**, se responderán con la respuesta simulada en lugar de realizar solicitudes reales a una API web.

1. **Realizando la Prueba**:



* + Dentro del bloque **Act**, se llama al método **GetMyModelsAsync** de **ApiService** para obtener los modelos simulados.

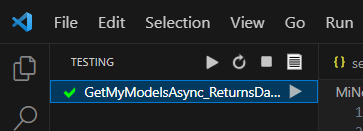
1. **Assert (Afirmaciones)**:

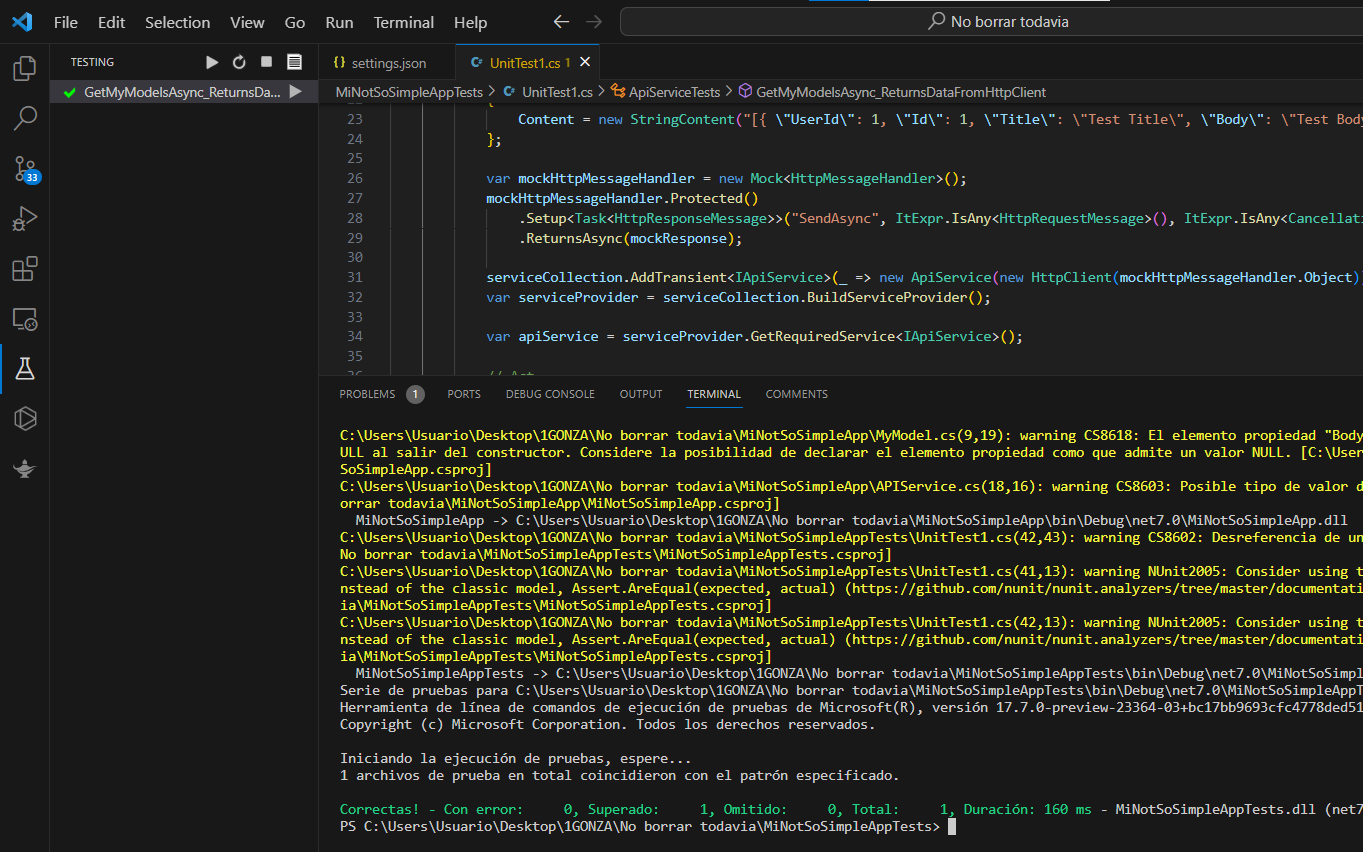


* + Se realizan afirmaciones para verificar si la respuesta de **GetMyModelsAsync** coincide con las expectativas definidas en **mockResponse**. En este caso, se verifica que la respuesta no sea nula, que contenga un elemento y que el título del primer elemento sea "Test Title".

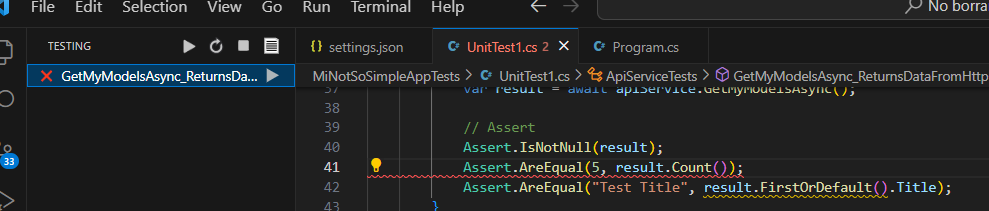
En resumen, este código utiliza un objeto mock para interceptar y simular las respuestas HTTP que se utilizan en la prueba de **ApiService**. Esto permite probar el método **GetMyModelsAsync** de **ApiService** de manera aislada y sin depender de una API web real.

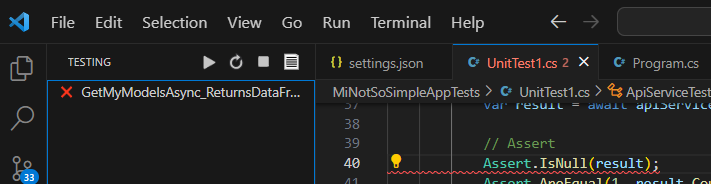
8.4 Ejecutamos Test en VSCode y en linea de consola.





8.5 Hacerlo falllar, arreglarlo y volverlo a correr.





8.6 Modificar el mock para que devuelva una colección de Posts y en un nuevo test verificar que la cantidad devuelta por el mock coincida con la esperada en el nuevo test.

8.7 Detallar todo lo realizado en el entregable.