

Programación Web .NET Core

Módulo 9

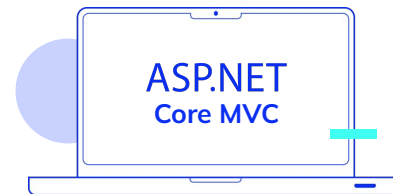
Prueba de aplicaciones MVC

Prueba de aplicaciones MVC

Una **prueba unitaria** es un procedimiento que instancia un componente que has escrito, llama a un aspecto de sus funcionalidades, y comprueba que responde correctamente a las pruebas. En la *programación orientada a objetos*, las pruebas unitarias suelen crear una **instancia de una clase** y llamar a uno de sus métodos.

En una aplicación web **ASP.NET Core MVC**, las pruebas unitarias pueden crear instancias de clases de modelo o controladores, y llamar a sus métodos y acciones. Como desarrollador web, necesitas saber cómo crear un proyecto de

prueba, agregarle pruebas y ejecutarlas. Las pruebas verifican aspectos individuales de tu código sin depender de servidores de bases de datos, conexiones de red u otra infraestructura. Esto es porque las pruebas unitarias se centran en el código que escribes.



¿Por qué realizar pruebas unitarias?

Las pruebas unitarias **verifican pequeños aspectos de funcionalidad**. Por ejemplo, pueden verificar el tipo de retorno de un solo método, o si el método devuelve los resultados esperados. Al definir muchas pruebas unitarias para tu proyecto, puedes asegurarte de que cubran todos los aspectos funcionales de la aplicación.

¿Qué es una prueba unitaria?

Es un procedimiento que **verifica un aspecto específico de la funcionalidad**. Las pruebas unitarias múltiples pueden ser realizadas para una sola clase e incluso para un solo método en una clase.

Por ejemplo, puedes escribir una prueba que comprueba que el método `Validate` siempre devuelve un valor booleano.

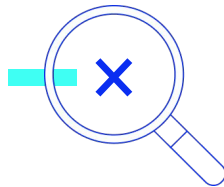


¿Cómo ayuda la prueba unitaria a diagnosticar errores?

Debido a que las pruebas unitarias verifican un aspecto pequeño y específico del código, es fácil diagnosticar el problema cuando las pruebas fallan. **Las pruebas unitarias generalmente funcionan con una sola clase y aíslan la clase de otras clases donde sea posible.**

Por ejemplo, las pruebas unitarias deben ejecutarse sin conectarse a una base de datos real o servicio web porque los problemas de la red o las interrupciones del servicio pueden provocar una falla.

Se puede utilizar la inyección de dependencias, crear clases falsas e implementar varios métodos como sean necesarios con los resultados esperados. Esto permite realizar pruebas completas para una clase, sin depender de otras entidades.

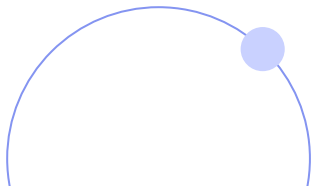


Escribir pruebas unitarias para componentes MVC

Los **modelos** son simples de probar porque son clases independientes que puedes instanciar y configurar durante la fase de organización de una prueba. Los **controladores** son clases simples que puedes probar, pero es complejo probar controladores con datos *inmemory*, en lugar de utilizar una base de datos. En los controladores de prueba con datos en memoria, se debe crear una clase doble de prueba, también conocida como *repositorio falso*. Los **objetos** de esta clase se pueden completar probando datos en la memoria sin consultar una base de datos.

Debes comprender cómo escribir dobles de prueba y cómo escribir una prueba típica para clases MVC.

MSTest es un marco que está integrado en **Visual Studio** y es mantenido por Microsoft. Otras populares opciones son **xUnit** y **NUnit**, las cuales también se pueden usar con bastante facilidad en Visual Studio.



Agregar y configurar un proyecto de prueba en Visual Studio

Se puede probar un proyecto de aplicación web **ASP.NET Core MVC** agregando un nuevo proyecto a la solución, basado en la plantilla **MSTest Test Project (.NET Core)**. Debes agregar una referencia del proyecto de prueba al proyecto de la aplicación **web MVC** para que el código de prueba pueda acceder a las clases en el proyecto de aplicación web MVC. También debes instalar el paquete **Microsoft.AspNetCore.Mvc** en el proyecto de prueba. En un proyecto de prueba de **Visual Studio MSTest**, las clases se marcan con el atributo **TestClass**.

Las pruebas son métodos marcados con el atributo **TestMethod**. Las pruebas unitarias generalmente llaman a un método de la clase **Assert**, como **Assert.AreEqual** para comprobar los resultados en la fase de prueba **Assert**.

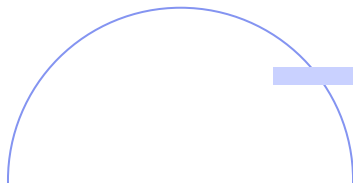
Nota: Hay muchos otros marcos de prueba disponibles para aplicaciones web MVC, y puedes elegir el que prefieras. Muchos de estos marcos se pueden agregar mediante el uso del *administrador de paquetes NuGet* en el **Visual Studio**.

Clases modelo de prueba y lógica empresarial

En MVC, las clases de modelo no dependen de otros componentes o infraestructura. Puedes instanciar fácilmente y ponerlos en memoria, ordenar sus valores de propiedad, actuar sobre ellos llamando a un método y afirmar el resultado esperado. El siguiente código es un ejemplo de un modelo de producto, que se probará:

Modelo para prueba

```
public class Product
{
    public string Name { get; set; }
    public int Id { get; set; }
    public float BasePrice { get; set; }
    public float GetPriceWithTax(float taxPercent)
    {
        return BasePrice + (BasePrice * (taxPercent / 100));
    }
}
```



El siguiente código es un ejemplo de una prueba para el modelo de **Producto** creado anteriormente:

Prueba de una clase de modelo

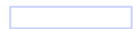
```
[TestClass]
public class ProductTest
{
    [TestMethod]
    public void TaxShouldCalculateCorrectly()
    {
        // Arrange
        Product product = new Product();
        product.BasePrice = 10;
        // Act
        float result = product.GetPriceWithTax(10);
        // Assert
        Assert.AreEqual(11, result);
    }
}
```



Esta prueba está diseñada para comprobar que el método **GetPriceWithTax** funciona correctamente.

Prueba de clases de controladores

A diferencia de los modelos, que son clases simples diseñadas para ser principalmente auto-suficientes sin depender de otras clases, **los controladores son más complejos de probar**. Para probar los controladores, necesitarás trabajo adicional, porque un controlador individual puede, potencialmente, confiar en modelos, repositorios y servicios y más.




El código que veremos en la siguiente pantalla, es un ejemplo de una clase **ProductController** que debe probarse.



Controlador para pruebas

```
public class ProductController : Controller
{
    IProductRepository _productRepository;
    public ProductController(IProductRepository productRepository)
    {
        _productRepository = productRepository;
    }
    public IActionResult Index()
    {
        var products = _productRepository.Products.ToList();
        return View(products);
    }
}
```



Crear un servicio de repositorio

Un **repositorio** es un patrón de diseño común que se utiliza para separar el código de lógica empresarial de la recuperación de datos. Esto se hace comúnmente creando una **interfaz de repositorio** que define propiedades y métodos que MVC puede utilizar para recuperar datos. Normalmente, el repositorio maneja las distintas **CRUD** (crear, leer, actualizar y borrar), operaciones sobre los datos.

En las aplicaciones ASP.NET Core MVC, el repositorio se maneja creando un **Servicio**. En general, querrás **un repositorio por entidad**.

Los repositorios son muy útiles para realizar **pruebas de los controladores** porque los modelos generalmente no usan datos externos.

El siguiente código es un ejemplo de una interfaz de repositorio:

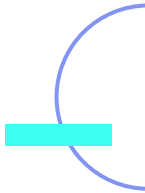
Una interfaz de repositorio

```
public interface IProductRepository
{
    IQueryable<Product> Products { get; }
    ProductAdd(Product product);
    ProductFindProductById(int id);
    ProductDelete(Product product);
}
```

Implementación y uso de un repositorio en la aplicación

La interfaz del repositorio define métodos para interactuar con los datos, pero no determina cómo esos datos serán configurados y almacenados. Debes proporcionar dos **implementaciones del repositorio**:

- **Una implementación de servicio del repositorio que se utilizará dentro de la aplicación.** Esta implementación utilizará datos de la base de datos o algún otro mecanismo de almacenamiento.
- **Una implementación del repositorio para su uso en pruebas.** Esta implementación utilizará datos en memoria para cada prueba.



El siguiente ejemplo de código muestra una implementación de servicio de un repositorio:

Implementación de un repositorio en el proyecto ASP.NET Core MVC

```
public class ProductRepository : IProductRepository
{
    StoreContext _store;
    public ProductRepository(StoreContext store)
    {
        _store = store;
    }
    public IQueryable<Product> Products
    {
        get { return _store.Products; }
    }
}
```



...

```
public Product Add(Product product)
{
    _store.Products.Add(product);
    _store.SaveChanges();
    return _store.Products.Find(product);
}

public Product Delete(Product product)
{
    _store.Products.Remove(product);
    _store.SaveChanges();
    return product;
}

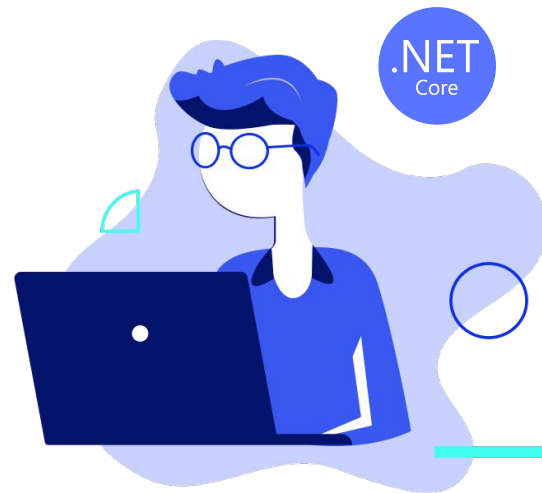
public Product FindProductById(int id)
{
    return _store.Products.First(product => product.Id == id);
}
}
```

Se debe tener en cuenta que **StoreContext** es una clase que hereda de la clase llamada **DbContext de EntityFramework**, que es inyectado mediante inyección de dependencia y declarado dentro del método denominado **ConfigureService**. De hecho, los métodos de interfaz como **Add**, **Delete** y **FindProductById** simplemente envuelven los métodos del **DbContext** clase como **Delete**.

Implementación de un doble de prueba de repositorio

La segunda implementación de la interfaz del repositorio es la implementación que usará en la **unidad pruebas**.

Esta implementación utiliza datos en memoria y una colección de objetos con clave para funcionar como un Entity Framework, pero evita trabajar con una base de datos.



El siguiente ejemplo de código muestra cómo implementar una clase de repositorio para pruebas:

Implementar un repositorio falso

```
public class FakeProductRepository : IProductRepository
{
    private IQueryable<Product> _products;
    public FakeProductRepository()
    {
        List<Product> products = new List<Product>();
        _products = products.AsQueryable();
    }
    public IQueryable<Product> Products
    {
        get { return _products.AsQueryable(); }
        set { _products = value; }
    }
}
```



...

```
public Product Add(Product product)
{
    List<Product> products = _products.ToList();
    products.Add(product);
    _products = products.AsQueryable();
    return product;
}

public Product Delete(Product product)
{
    List<Product> products = _products.ToList();
    products.Remove(product);
    _products = products.AsQueryable();
    return product;
}

public Product FindProductById(int id)
{
    return _products.First(product => product.Id == id);
}
}
```



**¡Sigamos
trabajando!**