Organización y Estructura de Proyectos en Soluciones .NET

La base para entender cuál es la forma que más nos conviene para estructurar una solución y la dependencia entre los proyectos (i.e: assemblies) de esa solución es conocer en detalle la diferencia entre los siguientes conceptos:

- Inversión de Control (IoC): se resume con el principio de Hollywood, "no me llames, ya te llamaremos nosotros".
- Inyección de Dependencias (DI): se resume en no instanciar una dependencia dentro de una clase o función que la necesite, sino inyectarla.
- Principio de Inversión de Dependencias (DIP): la D de SOLID. Se resume en favorecer la dependencia con abstracciones y no con implementaciones concretas. Las abstracciones no deben depender de detalles, los detalles deben depender de abstracciones.

Inversión de control (IoC)

Vamos a comenzar por una nueva solución y un nuevo proyecto de consola.

La inversión de control se puede evidenciar de muchas formas. Una función callback, un programa que lanza eventos al que se le puede conectar manejador de eventos, una factoría en la que se delega el control para que instancie algo, etc.

Si creamos una clase con un método capaz de obtener la fecha actual, basta con invocar esa funcionalidad. Si queremos que ese método, además, nos llame de vuelta para hacer algo con esa fecha, estaremos ante un clásico ejemplo de callback.

```
public class SampleDelegate
{
    public void RunForCurrentDate(Action<DateTime> callback)
    {
        var currentDate = DateTime.UtcNow;
        callback.Invoke(currentDate);
    }
}
```

Este método obtiene la fecha actual y después ejecuta el método que le estamos pasando y que acepta una fecha como argumento. El que llama, es el que decide qué es lo que se hace con ese parámetro, es decir, el que le pasa un método *callback* específico. La funcionalidad del llamado se limita a obtener la fecha actual y decidir cuándo y cómo llama a la función callback.

Como se puede apreciar, el control se invierte. El que llama indica lo que hay que hacer, pero es el llamado el que controla cuándo y cómo se ejecuta ese método.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Inversion of Control with a C# delegate (callback)");
        var sampleDelegate = new SampleDelegate();
        sampleDelegate.RunForCurrentDate(date =>
Console.WriteLine(date.ToLongDateString()));
    }
}
```

Podemos observar cómo utilizar una expresión lambda para crear un método anónimo, aunque también podríamos no utilizar un método anónimo y darle un nombre a ese método para despues pasar su referencia. No se ejecuta hasta que el que recibe la referencia a la función así lo decida.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Inversion of Control with a C# delegate (callback)");
        var sampleDelegate = new SampleDelegate();
        sampleDelegate.RunForCurrentDate(MyCallback); //se pasa la referencia a la
función, su nombre
    }

    private static void MyCallback(DateTime date)
    {
        Console.WriteLine(date.ToLongDateString());
    }
}
```

Otro ejemplo de inversión de control puede ser el de una factoría en la que delegamos para que haga una instanciación más o menos complicada.

```
public static class SampleFactory
{
    public static DateTime CreateDateTime()
    {
        return new DateTime();
    }
}
```

La cual ejecutamos, pero es la factoría quien decide cómo instanciar una fecha, invirtiendo así el control de la instanciación.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Inversion of Control with a C# delegate (callback)");
        var sampleDelegate = new SampleDelegate();
        sampleDelegate.RunForCurrentDate(date =>
Console.WriteLine(date.ToLongDateString()));

        Console.WriteLine("Inversion of Control with a factory");
        var date = SampleFactory.CreateDateTime();
        Console.WriteLine(date);
    }
}
```

Inyección de dependencias (DI)

Una funcionalidad puede instanciar a otra o utilizar una clase estática, pero en el momento en el que eso ocurre, se crea una fuerte dependencia entre la primera funcionalidad y la segunda. Por ejemplo, podemos tener una calculadora que suma números y los imprime en consola utilizando una impresora.

La impresora tiene un simple método que, dado un texto, lo imprime en consola.

```
public class Printer
{
    public void Print(string text)
    {
        Console.WriteLine(text);
    }
}
```

La calculadora tiene un método que dados dos números, los suma y los imprime, y para ello crea una instancia de la impresora

```
public class Calculator
{
    public void Sum(int a, int b)
    {
        var result = a + b;
        var printer = new Printer(); // calculadora depende de una implementación
concreta de printer
        printer.Print($"Result: {result}");
    }
}
```

Podemos iniciar este ejemplo desde nuestro método main para verlo en acción mostrando el resultado de la suma en consola.

```
class Program
{
    static void Main(string[] args)
    {
        var calculator = new Calculator();
        calculator.Sum(10, 5);
    }
}
```

Una decisión interesante podría ser la de hacer explícita esa dependencia entre calculadora e impresora, e inyectar la instancia de la impresora en el método de suma de la calculadora (i.e: inyección de dependencia por método). De esta manera, calculadora ya no se debe preocupar de instanciar una calculadora.

```
public class Calculator
{
    public void Sum(int a, int b, Printer printer)
    {
        var result = a + b;
        printer.Print($"Result: {result}");
    }
}
```

Ahora la instanciación de la impresora sigue ocurriendo, alguien la debe hacer, pero a la calculadora ya no le importa ni quién lo haga ni cómo lo haga.

```
class Program
{
    static void Main(string[] args)
    {
        var calculator = new Calculator();
        var printer = new Printer();
        calculator.Sum(10, 5, printer);
    }
}
```

¿Qué ventajas tiene esto?

- Se potencia el principio de responsabilidad única (i.e: la S de SOLID)
- La impresora puede evolucionar, e incluso cambiar la forma en la que se instancia, sin que esto afecte a la calculadora
- Se hace explícita esta dependencia en el método de suma, por lo que el que utilice esta funcionalidad sabe que existe esa dependencia y debe satisfacerla.
- Más adelante veremos cómo podemos meter el polimorfismo en todo esto y conseguir abstracción y mayor testeabilidad.

Las dependencias se pueden inyectar a través de un método, o de un constructor. Si tuviésemos un método que resta dos números, podría ser interesante hacer la dependencia a la impresora todavía más explícita a nivel de clase.

```
public class Calculator
{
    private readonly Printer _printer;

    public Calculator(Printer printer)
    {
        _printer = printer;
    }

    public void Sum(int a, int b)
    {
        var result = a + b;
        _printer.Print($"Result: {result}");
    }

    public void Sub(int a, int b)
    {
        var result = a - b;
        _printer.Print($"Result: {result}");
    }
}
```

Ahora la calculadora entera depende de una impresora y los métodos simplemente la utilizan. Hemos conseguido hacer esa dependencia más explícita.

```
class Program
{
    static void Main(string[] args)
    {
        var printer = new Printer();
        var calculator = new Calculator(printer);

        calculator.Sum(10, 5);
        calculator.Sub(30, 20);
    }
}
```

Principio de Inversión de Dependencias

Ya conocemos lo que significa invertir el control e inyectar dependencias. Todas estas técnicas nos van a permitir escribir mejor código, más mantenible, más modular y explícito. Pero aún podemos ir más allá y aprovecharnos del poder que el polimorfismo nos da en la programación orientada a objetos.

El polimorfismo dice que algo puede tener una o varias formas. Si algo se puede manifestar de diferentes formas, podemos abstraernos y depender de un contrato, mejor que de una implementación concreta.

Vamos a ver cómo podemos abstraer la impresora. Sabemos que algo es una impresora si puede imprimir. A la calculadora seguramente no le importe si imprimir significa mostrarlo por consola, plasmarlo físicamente en un papel o enviarlo a otra máquina para que aparezca como una notificación en pantalla. A la calculadora le interesa que se imprima. La forma concreta en la que así ocurra es responsabilidad de otro (el que le inyecta una impresora concreta).

En C# podemos utilizar abstracciones con clases abstractas o con interfaces. La relación en ambos casos es de tipo "Is A". Si una impresora que imprime en consola hereda de una abstracción `Printer`, se dice que esa impresora concreta es una `Printer`. Podemos tener otra impresora concreta que imprime en papel y que también sea una `Printer`.

El contrato puede ser, por ejemplo, una clase abstracta (no se instancia directamente, sus métodos abstractos los debe implementar quien herede de ella)

```
public abstract class Printer
{
    public abstract void Print(string text);
}
```

Ahora podemos tener una impresora por consola que **es una** `Printer`

```
public class ConsolePrinter
    : Printer
{
    public override void Print(string text)
    {
        Console.WriteLine(text);
    }
}
```

y una impresora por papel que también **es una** `Printer` y, por lo tanto, deben sobreescribir los métodos de `Printer` de una forma concreta.

```
public class PaperPrinter
    : Printer
{
    public override void Print(string text)
    {
        // I use some OS printing drivers to connect to a device and send
documents
        Console.WriteLine($"A4 paper says: {text}");
    }
}
```

La calculadora puede, ahora, depender de la abstracción, y no de una implementación concreta.

```
public class Calculator
{
    private Printer _printer;

    public Calculator(Printer printer)
    {
        _printer = printer;
    }

    public void Sum(int a, int b)
    {
        var result = a + b;
        _printer.Print($"Result: {result}");
    }

    public void Sub(int a, int b)
    {
        var result = a - b;
        _printer.Print($"Result: {result}");
    }

    /// <summary>
    /// Printer can be replaced at runtime
    /// </summary>
    public void ReplacePrinter(Printer printer)
    {
        _printer = printer;
    }
}
```

¿Qué implementación concreta de `Printer` utiliza la calculadora? Ni lo sabe, ni le importa. Solo quiere imprimir. Lo decide otro, el que le inyecte la dependencia. Se invierte el control, se inyecta la dependencia y se invierte la dependencia porque a través de la abstracción se está invocando una funcionalidad concreta sin ligarse a ella. La dependencia sigue existiendo, pero ahora se depende de una abstracción, no de los detalles.

Este concepto es fundamental para dominar la programación orientada a objetos, porque la mayoría de los patrones de diseño se basan en este uso del polimorfismo, herencia y composición.

`ConsolePrinter` y `PaperPrinter` heredan de `Printer`. `Calculator` se compone de `Printer`. El que instancia, decide qué forma concreta adquiere la `Printer` en ese momento. Lo poderoso, si se puede apreciar, es que la calculadora puede usar una u otra impresora indiferentemente, y esa decisión se puede tomar en tiempo de ejecución sin "molestar" a la calculadora.

```
class Program
{
    static void Main(string[] args)
    {
```

```
        Printer printer = new ConsolePrinter(); // soy una impresora de consola
        var calculator = new Calculator(printer);
        calculator.Sum(10, 5);
        calculator.Sub(30, 20);

        Console.WriteLine("Now I replace printer with a paper one at runtime");
        calculator.ReplacePrinter(new PaperPrinter());
        calculator.Sum(10, 5);
        calculator.Sub(30, 20);
    }
}
```

Este ejemplo tan simple, nos está mostrando un patrón de diseño muy importante, el de estrategia. Y lo hemos alcanzado con inyección de dependencia, composición y con inversión de dependencia.

En lugar de clase abstracta, podemos utilizar también interfaces. En ambos casos la relación de herencia se mantiene.

```
public interface IPrinter
{
     void Print(string text);
}

public class ConsolePrinter
    : IPrinter
{
    public void Print(string text)
    {
        Console.WriteLine(text);
    }
}

public class PaperPrinter
    : IPrinter
{
    public void Print(string text)
    {
        // I use some OS printing drivers to connect to a device and send
documents
        Console.WriteLine($"A4 paper says: {text}");
    }
}

public class Calculator
{
    private IPrinter _printer;

    public Calculator(IPrinter printer)
    {
        _printer = printer;
    }
```

```
    public void Sum(int a, int b)
    {
        var result = a + b;
        _printer.Print($"Result: {result}");
    }

    public void Sub(int a, int b)
    {
        var result = a - b;
        _printer.Print($"Result: {result}");
    }

    /// <summary>
    /// Printer can be replaced at runtime
    /// </summary>
    public void ReplacePrinter(IPrinter printer)
    {
        _printer = printer;
    }
}
```

Y el programa principal sería prácticamente idéntico, pero con una interfaz como abstracción

```
class Program
{
    static void Main(string[] args)
    {
        IPrinter printer = new ConsolePrinter(); // soy una impresora de consola
        var calculator = new Calculator(printer);
        calculator.Sum(10, 5);
        calculator.Sub(30, 20);

        Console.WriteLine("Now I replace printer with a paper one");
        calculator.ReplacePrinter(new PaperPrinter());
        calculator.Sum(10, 5);
        calculator.Sub(30, 20);
    }
}
```

Esta inversión de dependencia es la base para comprender la arquitectura limpia, como demuestra la solución de `clean-architecture-soccer` que es una aplicación web completa, con una capa de dominio, aplicación y varias de infraestructura, y con tests unitarios y funcionales.

Nuget

En un tema anterior de pruebas unitarias vimos cómo crear una librería con una funcionalidad para convertir cryptomonedas a EUR.

Si quisiéramos compartir esta funcionalidad con otros programadores, podríamos dejar que copiaran y pegaran el código, pero esto sería algo absurdo. Una mejor opción es empaquetar esta funcionalidad, como código ya compilado, en un paquete o librería nuget, y compartir ese nuget.

El nuget se puede compartir directamente, o hacer que esté disponible en un repositorio online para que otros programadores puedan, desde sus proyectos, añadir una referencia a esta librería, e incluso actualizar la versión cuando se actualice.

Vamos a transformar la librería `crypto-fiat-converter` en un paquete nuget. Pero antes, hagamos unos pequeños cambios.

Toda la funcionalidad que, como usuarios de la librería, queríamos tener disponible estaba en la clase `Converter`, y más concretamente en un método con firma `public ConversionResult ConvertToEur(string cryptoCode, decimal amount)`.

Ahora que dominamos la inversión de dependencias, sabemos que podemos programar hacia una abstracción mejor que hacia una implementación concreta. Vamos a abstraer esta funcionalidad creando un nuevo proyecto de tipo class library `LemonCode.CryptoFiatConverter.Abstractions`

Todo lo que a un usuario de nuestra librería le interesa conocer a la hora de utilizar la funcionalidad es el método

```
public interface IConverter
{
    ConversionResult ConvertToEur(string cryptoCode, decimal amount);
}
```

y el tipo especial `ConversionResult`, así que lo movemos a este nuevo proyecto.

```
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;
    }
}
```

Debemos ahora arreglar el proyecto anterior. Por un lado crearemos una dependencia de `Lemoncode.CryptoFiatConverter` a `Lemoncode.CryptoFiatConverter.Abstractions`

Después adaptaremos la clase `Converter` para que herede (i.e: implemente) la interfaz `IConverter` y para que sepa dónde está el tipo `ConversionResult`

```
public class Converter
    : IConverter
{
    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;
    }
}
```

Después hay que arreglar los proyectos de test para que sepan dónde encontrar `ConversionResult`, que se ha movido de namespace.

Nos aseguramos de que la solución compila y que los tests pasan correctamente.

Vamos, ahora, a asegurarnos de que la metainformación del proyecto que contiene las abstracciones públicas y el que contiene la implementación concreta de `IConverter`, son capaces de producir paquetes nuget y publicarse.

Ambos deben tener esto

```
<PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <IsPackable>true</IsPackable>
    <IsPublishable>true</IsPublishable>
</PropertyGroup>
```

Por el contrario, no queremos que el proyecto de test se convierta en un paquete nuget ni que publique artefactos dll cuando se despliegue en un servidor (solo interesa para ejecutar tests), por lo que debe tener lo siguiente.

```
<PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <IsPublishable>false</IsPublishable>
    <IsPackable>false</IsPackable>
</PropertyGroup>
```

Nuget en local

Lo primero que debemos comprobar es que los proyectos

Para crear un nuget en local

```
dotnet pack *.sln --configuration Release --output ../misNugets
```

Se puede observar que los paquetes `Lemoncode.CryptoFiatConverter.1.0.0.nupkg` y `Lemoncode.CryptoFiatConverter.Abstractions.1.0.0.nupkg` se han generado dentro del directorio `misNugets`

Si quisiéramos versionar los paquetes, para que se genere una versión diferente siguiendo **semVer** con `Major.Minor.Patch` cuando modifiquemos algo, podemos añadir más metadatos a los proyectos. Por ejemplo en el `Lemoncode.CryptoFiatConverter.csproj`

```
<PropertyGroup Label="Package information">
    <Authors>Diego Martin</Authors>
    <Company>Lemoncode</Company>
    <PackageTags>bootcamp</PackageTags>
    <Website>https://lemoncode.net/</Website>
    <PackageWebsite>https://lemoncode.net</PackageWebsite>
    <NeutralLanguage>en</NeutralLanguage>
    <Title>$(AssemblyName)</Title>
    <VersionPrefix>1.1.0</VersionPrefix>
</PropertyGroup>
```

Si volvemos a generar nugets

```
dotnet pack *.sln --configuration Release --output ../misNugets
```

podemos ver cómo hay una nueva versión, la que indique `VersionPrefix`, en uno de los paquetes. Las abstracciones, como no han cambiado, podemos dejarlas con la misma versión.

Ahora creamos un nuevo proyecto de consola en una nueva solución.

En Visual Studio ir a Tools > Nuget Package Manager > Package Manager Settings. Vamos a añadir una ruta al gestor de paquetes nuget de Visual Studio que apunte a nuestra carpeta donde tengamos paquetes nuget.

Ahora buscamos paquetes en este repositorio nuget que hemos añadido. Podemos ver que si seleccionamos el feed indicado, hay dos paquetes disponibles para ser utilizados. Uno de ellos tiene dos versiones disponibles.

Si observamos las dependencias, se aprecia que `Lemoncode.CryptoFiatConverter` depende de `Lemoncode.CryptoFiatConverter.Abstractions`, así que si instalamos el primero tendremos implícitamente la dependencia al segundo también.

Ahora podemos utilizar la librería gracias al paquete nuget.

```
class Program
{
    static async Task Main(string[] args)
    {
        var dateTimeFactory = new DateTimeFactory();
        var priceDatabase = new InMemoryPriceDatabase(dateTimeFactory);


        // admin part
        IExchangeRateUpdater updater = new
CoinGeckoExchangeRateUpdater(priceDatabase);
        await updater.Update();

        // user part
        IConverter converter = new Converter(priceDatabase);
        var amount = 15000;
        var result = converter.ConvertToEur("HBAR", amount);
        Console.WriteLine($"Result of {amount} HBAR: {result.TotalEur} EUR, (last
updated on {result.RateUpdatedOn})");
    }
}
```

Nuget en un feed remoto en GitLab

Compartir nugets directamente está bien, pero nos hace depender de un directorio concreto en nuestro PC que, además, otros desarrolladores del proyecto podrían no tener.

Hoy en día hay muchos productos online que ofrecen cosas como repositorio git, espacio para guardar paquetes nuget, imágenes docker, etc. como GitHub, GitLab, Azure DevOps, etc.

GitLab, por ejemplo, permite la creación de proyectos y repositorios nuget privados de forma gratuita.

Creamos un nuevo proyecto privado en GitLab. Solo usuarios autorizados podrán acceder a él y a lo que en él se guarde.

En la solución de nuestra librería vamos a crear un repositorio git.

```
git init
```

y lo asociamos al repositorio git de GitLab recientemente creado

```
git remote add origin git@gitlab.com:sunnyatticsoftware/lemoncode/crypto-fiat-converter-library.git
```

Añadimos y hacemos commit de nuestro código en el nuevo repositorio git local

```
git add .
git commit -m "Initial commit"
```

Setea el upstream branch para la rama main

```
git push --set-upstream origin main
```

Y sincronizamos los cambios locales con el repositorio remoto.

```
git push
```

Ahora creamos un archivo `.gitlab-ci.yml` en el repositorio de GitLab para la pipeline CI/CD. Usa este:

```
variables:
  PACKAGE_OUTPUT_DIR: nuget

image: mcr.microsoft.com/dotnet/sdk:5.0

stages:
  - build
  - test
  - package
  - delivery

build:
  stage: build
  script:
    - dotnet restore --no-cache --force
    - dotnet build --configuration Release --no-restore
  artifacts:
    paths:
    - test
    expire_in: 8 hour
```

```
test:
  stage: test
  script: dotnet test --blame --configuration Release
  rules:
    - exists:
      - test/**/*Tests.csproj

package_beta:
  stage: package
  variables:
    PACKAGE_UNSTABLE_SUFFIX: beta
  script:
    - dotnet pack *.sln --configuration Release --output $PACKAGE_OUTPUT_DIR --
version-suffix $PACKAGE_UNSTABLE_SUFFIX --include-source --include-symbols
  rules:
    - if: $CI_COMMIT_BRANCH != "main"
  artifacts:
    paths:
    - $PACKAGE_OUTPUT_DIR/
    expire_in: 8 hour

package_stable:
  stage: package
  script:
    - dotnet pack *.sln --configuration Release --output $PACKAGE_OUTPUT_DIR --
include-symbols
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
  artifacts:
    paths:
      - $PACKAGE_OUTPUT_DIR/
    expire_in: 8 hour

package_registry:
  stage: delivery
  script:
    - dotnet nuget remove source gitlab || true
    - dotnet nuget add source
"$CI_SERVER_URL/api/v4/projects/$CI_PROJECT_ID/packages/nuget/index.json" --name
gitlab --username gitlab-ci-token --password $CI_JOB_TOKEN --store-password-in-
clear-text
    - dotnet nuget push $PACKAGE_OUTPUT_DIR/**/*.nupkg --source gitlab --skip-
duplicate
```

Añade cambios, comitea y haz push.

Observa la pipeline en ejecución y lo siguiente:

- Generación de paquete nuget
- Guardado del nuget en el package registry del proyecto... y del grupo (que se puede usar como nuget feed)

Bonus - compartir acceso a paquetes nuget de repositorio privado

Ahora imagina que en algún proyecto necesitas acceder a repositorios nuget privados como el de GitLab, ¿cómo conseguirlo?

En el proyecto aislado de consola, elimina los nuget añadidos desde local y ciérra la solución.

En GitLab, en el proyecto, grupo o subgrupo que contenga al repositorio, ve a Settings > Repository > Deploy Tokens y genera un nuevo token con permiso `read_package_registry`

Name: `nuget-read-only` Scopes: `read_package_registry`

Esto produce el token username (e.g: `gitlab+deploy-token-XXXXXX`) y una password (e.g: `123456789abcdefgh`)

Crea, en el directorio principal de la solución independiente, un `NuGet.Config` con el siguiente contenido sustituyendo el {{group_id}} por el grupo o subgrupo de GitLab y sustituyendo el {{username}} y el {{password}} por el que corresponda según el deploy token:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <packageSources>
        <clear />
        <add key="nuget.org" value="https://api.nuget.org/v3/index.json"
protocolVersion="3" />
        <add key="gitlab"
value="https://gitlab.com/api/v4/groups/{{group_id}}/-/packages/nuget/index.json"
/>
    </packageSources>
    <packageSourceCredentials>
        <gitlab>
            <add key="Username" value="{{username}}" />
            <add key="ClearTextPassword" value="{{password}}" />
        </gitlab>
    </packageSourceCredentials>
</configuration>
```

Abre la solución. Observa que:

- Nuget package manager tiene automáticamente un nuevo package source, el definido en Nuget.Config, y tienes permisos de lectura en ese repositorio.

Sustituye la dependencia a un nuget local (elimina el package source local) por un nuget del repositorio privado. Prueba que la aplicación sigue funcionando.