

# Упражнение: Софтуерно тестване. Непрекъснатата интеграция

**Проектът:** Ще ви бъде предоставен проект - .NET Core библиотека, върху която ще трябва да пишете тестовете си. Преди да започнете работата по упражнението се запознайте добре със структурата и функционалността на проекта. (Проектът може да бъде изтеглен и от: <https://github.com/it-career/SoftwareTestingExercise>)

## 1. Какво ще постигнем

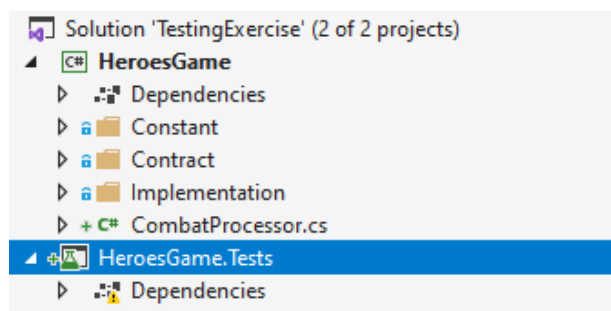
Следвайки упражнението, ще упражните практиките за писане и провеждане на unit тестове с NUnit + mocking с Moq.

## 2. Проект за тестване

Ще започнем като към дадения ни проект добавим нов, който ще служи за тестване. Ще кръстим проекта HeroesGame.Tests (.NET Core library). След това ще инсталираме нужните пакети за unit тестване:

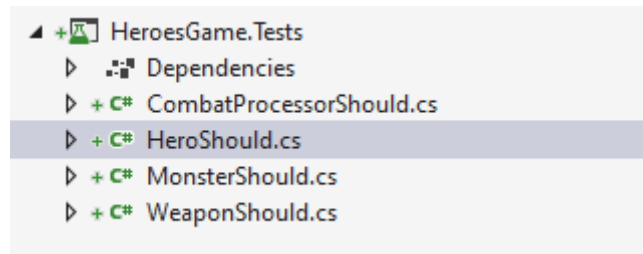
- NUnit
- NUnit3TestAdapter
- Microsoft.NET.Test.Sdk

Това може да бъде направено през NuGet package manager-а. След инсталацията build-нете проекта, за да се сигурни, че всичко е наред.



## 3. Първи стъпки в unit тестването

След като имаме проект за тестове, нека разделим тестовете за Weapon класовете, Hero класовете и Monster класовете в отделни класове. Аналогично CombatProcessor ще бъде тестван в отделен клас. Вече можем да добавим референция към тествания проект и да започнем писането на нашите тестове. Unit тестовете не са нищо по-различно от публични методи на класове, като всеки метод трябва да бъде аотиран с атрибута [Test], който идва от NUnit.Framework библиотеката. Нека първо тестваме класовете за герои.



Ще започнем като тестваме дали при създаване на нов герой му се задават правилните начални стойности.

```
0 references
public class HeroShould
{
    [Test]
    0 references
    public void HaveCorrectInitialValues()
    {
    }
}
```

Следвайки принципа за три логически фази на тестване първо ще подготвим тествания обект като го инициализираме, като в този случай тестваме конструктора т.е. тази фаза съвпада със втората фаза за извикване на тествания метод. Накрая – в третата фаза ще сравним получения резултат с очаквания от нас такъв. Сравняването на резултатите става чрез статичния клас на NUnit.Framework Assert. С класът Is пък се достъпват различни опции за сравнение. Всеки тест метод може да съдържа много Assert твърдения, като тестът минава тогава и само тогава, когато всички Assert връщат true. Може да прочетете повече за класовете Assert и Is като ги маркирате и натиснете F1 (ще бъдете препратени към онлайн документацията на NUnit.Framework).

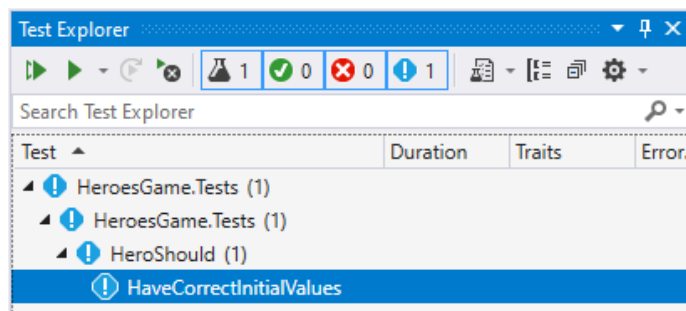
В случая, за да тестваме дали правилно са зададени първоначалните стойности трябва да проверим една по една стойностите на свойствата на току-що създаден от нас обект от тип герой:

```
[Test]
0 references
public void HaveCorrectInitialValues()
{
    //Arrange = Act
    var hero = new Mage();

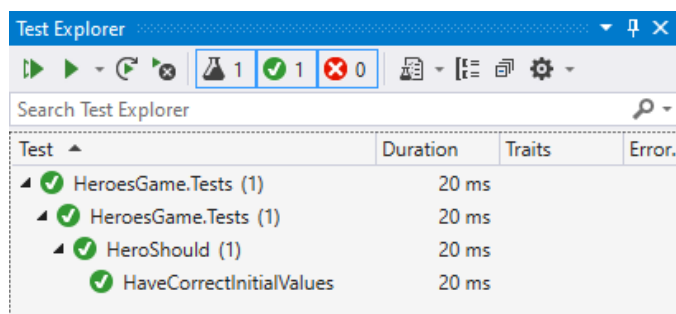
    //Assert
    Assert.That(hero.Level, expression: Is.EqualTo( expected: HeroConstants.InitialLevel));
    Assert.That(hero.Experience, expression: Is.EqualTo( expected: HeroConstants.InitialExperience));
    Assert.That(hero.MaxHealth, expression: Is.EqualTo( expected: HeroConstants.InitialMaxHealth));
    Assert.That(hero.Health, expression: Is.EqualTo( expected: HeroConstants.InitialMaxHealth));
    Assert.That(hero.Armor, expression: Is.EqualTo( expected: HeroConstants.InitialArmor));
    Assert.That(hero.Weapon, Is.NotNull);
}
```

## 4. Провеждане на тестове

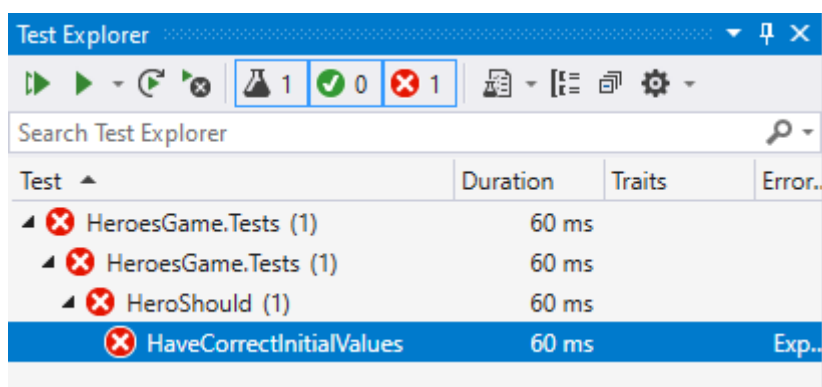
Visual Studio предлага лесен начин да проведем нашите тестове – чрез Test Explorer. За целта изкарайте Test Explorer-а от Test -> Test Explorer. Visual Studio автоматично ще открие методите маркирани с атрибута Test и ще ги разпознае като тестове. Ако всичко е наред трябва теста да се покаже.



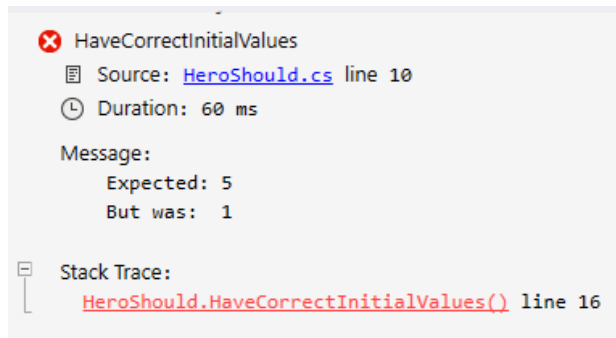
Провеждането на теста става или от менюто на Test Explorer-а или десен бутон върху теста -> Run. След провеждането му трябва да получим положителен резултат.



Нека нарочно добавим грешка в теста, която след това ще премахнем, за да видим какво става, когато теста се проваля. Нека обозначим, че очакваме нивото на новосъздадения ни герой да е 5-то, след което пуснем теста отново.



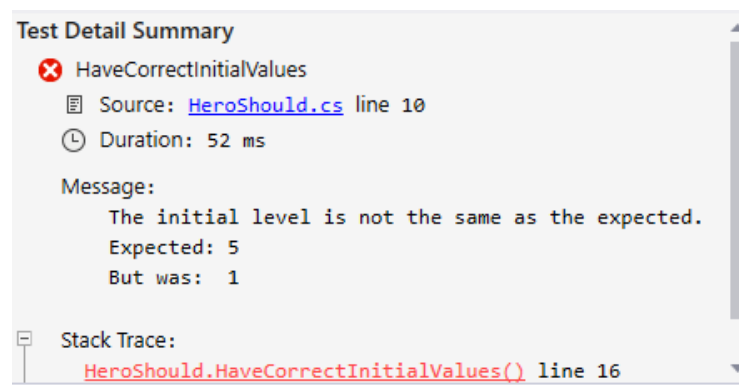
Този път теста се провали. В TestExplorer-а можем да видим съобщение, което описва защо тестът се е провалил. Ако не харесваме това съобщение можем и ръчно да създадем наше такова, трябва само да го добавим като символен низ, който е трети аргумент на That() метода.



Нека добавим наше съобщение и пуснем теста отново, за да видим как изглежда.

```
//Assert
Assert.That(hero.Level, Is.EqualTo(5), "The initial level is not the same as the expected.");
Assert.That(hero.Experience, Is.EqualTo(HeroConstants.InitialExperience)).
```

Тестът отново се проваля, но този път получаваме по-подробна информация относно причината.



Нека върнем теста в предишното му, работещо, състояние.

## 5. Тестване класа за герои

В тази част ще напишем тестове за всички методи на героя.

- `double` TakeHit (`double` damage)

Очаквано е за нов герой след поемане на удар жизнените му точки му да е равна на първоначалните жизнени точки минус стойността на удара. В уравнението добавяме и стойността на първоначалната защита за герой.

```
[Test]
[0 references]
public void TakeHitCorrectly()
{
    //Arrange
    var hero = new Warrior();

    //Act
    var damage = 50;
    hero.TakeHit(damage);

    //Assert
    Assert.That(hero.Health, expression: Is.EqualTo(expected: HeroConstants.InitialMaxHealth - damage +
        HeroConstants.InitialArmor));
}
```

Забележете как и в двата теста, които имаме досега се повтаря един и същи код – инициализираме нов герой (няма значение от какъв вид, тъй като тестваме базовия клас). Тази дубликация на код лесно може да бъде избегната като се използва метод, който се изпълнява само веднъж преди всички тестове и в който се инициализира герой или такъв в който наново се инициализира герой преди всеки един тест. Това може да бъде постигнато като се използват съответно атрибутите `OneTimeSetUp` и `SetUp`.

```
private IHero _hero;

[SetUp]
0 references
public void Setup()
{
    this._hero = new Mage();
}
```

След като добавихме поле, което пази нашия герой и метод, който го реинициализира преди всеки тест можем спокойно да изтрием повтарящия се код от тестовете си и да направим проверките да сочат към това поле.

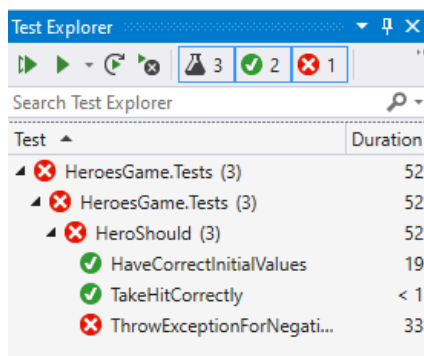
Аналогично ако се наложи можем да използваме атрибутите `OneTimeTearDown` и `TearDown`, за да аотираме методи, които да се изпълняват след тестовете.

Освен за правилно калкулиране на щетите и отнемането от жизнените точки на даден герой трябва да проверим дали `TakeHit` методът хвърля грешка при подаване на невалидна стойност (отрицателно число за щетите). Това може да се постигне, като се използва метода `Throws<T>()` на класа `Assert`.

```
[Test]
0 references
public void ThrowExceptionForNegativeTakeHitValue()
{
    //Act
    var damage = -50;

    //Assert
    Assert.Throws<ArgumentException>(() => this._hero.TakeHit(damage), "Damage value cannot be negative");
}
```

Нека проведем всички тестове и видим резултатите от тях.



Последният ни тест се проваля – по този начин открихме първата си грешка в кода чрез unit тестване. Нека оправим метода TakeHit() – при отрицателна стойност за щетите да хвърля грешка от тип ArgumentException със съобщение „Damage value cannot be negative“ и пуснем теста отново. Ако този път всички тестове минат, грешката е правилно отстранена.

Забележете, че в момента тестваме метода само с една отрицателна стойност и с една положителна такава. Това не е много показно дали метода работи коректно. Какво ако искаме да тестваме за повече стойности? Да напишем 100 теста, за да тестваме метода с различни стойности е грешен подход, който ще отнеме много време. За целта NUnit предлага няколко решения на този проблем.

Нека първо разгледаме атрибута TestCase. Той позволява няколко пъти да се изпълни един и същи метод като различен тест с различни стойности. За целта сигнатурата на метода трябва да се промени по такъв начин, че стойностите, които искаме да се различават в тестовете да се получават като параметри за метода.

```
[Test]
[TestCase(arg: 10)]
[TestCase(arg: 20)]
[TestCase(arg: 30)]
0 references
public void TakeHitCorrectly_TestCase(int damage)
{
    this._hero.TakeHit(damage);

    //Assert
    Assert.That(this._hero.Health, expression: Is.EqualTo(expected: HeroConstants.InitialMaxHealth - damage + HeroConstants.InitialArmor));
}
```

След провеждането на тестовете, в Test Explorer-а ще откриете, че с новият ни метод са проведени 3 теста съответно със стойности 10, 20 и 30 за щетите.

Аналогично напишете TestCase-ове и за отрицателни стойности, които се подават на метода TakeHit().

Друго решение на същия проблем е да се аотира параметрите на метода с атрибута Values, като се подадат всички стойности, които искаме да се тестват за даден параметър. Обърнете внимание, че по този начин ще се извъртят всички възможни опции – ако имаме 3 параметъра с по три стойности зададени чрез Values ще бъдат проведени 27 теста.

```
[Test]
0 references
public void TakeHitCorrectly_Combinatorial([Values(40, 50, 60)] int damage)
{
    this._hero.TakeHit(damage);

    //Assert
    Assert.That(this._hero.Health, expression: Is.EqualTo(expected: HeroConstants.InitialMaxHealth - damage + HeroConstants.InitialArmor));
}
```

Отново можем да видим, че са проведени 3 нови теста.

Последното решение на този проблем, което ще разгледаме, е чрез атрибута Range. Той работи по подобен начин на Values, но приема начална стойност, крайна стойност и стъпка, като започва от началната и увеличава със стъпката докато не достигне до крайната стойност.

```
[Test]
0 references
public void TakeHitCorrectly_Range([Range(from: 70, to: 100, step: 10)] int damage)
{
    this._hero.TakeHit(damage);

    //Assert
    Assert.That(this._hero.Health, expression: Is.EqualTo(expected: HeroConstants.InitialMaxHealth - damage + HeroConstants.InitialArmor));
}
```

Отново, ако проверим ще видим, че за всяка стъпка се провежда отделен тест.

Напишете тестове, проверяващи коректността на методите:

- `void GainExperience(double xp)`
  - Забележете, че според имплементацията на методи дори при получаване на огромно количество точки опит не може да се вдигне повече от 1 ниво наведнъж (оставете това така).
- `void Heal ()`
- `bool IsDead ()`

### Примерна имплементация (отвори само при нужда):

```
[Test]
0 references
public void GainExperienceCorrectly([Range(from: 25, to: 500, step: 25)]double xp)
{
    //Act
    this._hero.GainExperience(xp);

    //Assert
    if (xp >= HeroConstants.MaximumExperience)
    {
        var expectedXp = (HeroConstants.InitialExperience + xp) % HeroConstants.MaximumExperience;
        Assert.That(this._hero.Experience, expression: Is.EqualTo(expectedXp));
        Assert.That(this._hero.Level, expression: Is.EqualTo(expected: HeroConstants.InitialLevel + 1));
    }
    else
    {
        Assert.That(this._hero.Experience, expression: Is.EqualTo(expected: HeroConstants.InitialExperience + xp));
    }
}
```

```

[Test]
✓ | 0 references
public void HealCorrectly([Range(from: 5, to: 25, step: 1)]int level, [Range(from: 25, to: 500, step: 25)]int damage)
{
    // Act
    // level up our hero
    this.LevelUp(level);
    // then take a hit
    double totalDamage = HeroConstants.InitialMaxHealth + damage;
    totalDamage = this._hero.TakeHit(totalDamage);
    this._hero.Heal();

    //Assert
    var healValue = this._hero.Level * HeroConstants.HealPerLevel;
    var expectedHealth = (this._hero.MaxHealth - totalDamage) + healValue;

    if (expectedHealth > this._hero.MaxHealth)
        expectedHealth = this._hero.MaxHealth;

    Assert.That(this._hero.Health, expression: Is.EqualTo(expectedHealth));
}

1 reference | ✓ 420/420 passing
private void LevelUp(int levels)
{
    for (int i = 0; i < levels; i++)
    {
        this._hero.GainExperience(xp: HeroConstants.MaximumExperience);
    }
}

[Test]
✓ | 0 references
public void NotBeBornDead()
{
    //Act
    var isDead = this._hero.IsDead();

    //Assert
    Assert.That(this._hero.IsDead, Is.False);
}

[Test]
✓ | 0 references
public void BeDeadWhenCriticallyHit([Range(from: 50, to: 150, step: 25)] double damage)
{
    //Act
    damage = this._hero.TakeHit(damage);

    //Assert
    if (damage >= this._hero.MaxHealth)
    {
        Assert.That(this._hero.IsDead);
    }
    else
    {
        Assert.That(this._hero.IsDead, Is.False);
    }
}

```



## Тестване в специфика

След като сме написали нашите тестове за базовата функционалност на всеки герой е време да напишем и тестове в специфика. Например, когато герой от тип ловец вдигне ниво, неговите атрибути (жизнени точки, броня) трябва да се увеличават според класа му. Опитайте да напишете тестовете самостоятелно с придобитите вече знания - това няма е фокус на упражнението, а самостоятелна задача, за да упражните наученото.

## 6. Тестване на класовете за оръжие и чудовище

По аналогичен начин ще тестваме всеки метод и свойство на класовете за оръжия и чудовища, като започнем от базовата функционалност и преминем към специфика. Опитайте да напишете тестовете сами и не забравяйте, че това което прави даден тест добър е количеството код, изпълнявано заедно с теста. Колкото повече код покриват тестовете ни толкова повече може да се разчита на тях. Ние ще преминем директно към по-специфичната част – тестването на процесора за битки.

## 7. Тестване процеса на битка

В тази секция ще тестваме класа `CombatProcessor`. Нека започнем от конструктора, като проверим дали при създаването на нов процесор на битка правилно се задават първоначални стойности за протокола и героя. Очаквано е при създаване на нов процесор, той да има герой, както и протокол и проткола да е празен.

```
private CombatProcessor _cp;

[SetUp]
0 references
public void Setup()
{
    this._cp = new CombatProcessor(new Hunter());
}

[Test]
✓ | 0 references
public void InitializeCorrectly()
{
    //Assert
    Assert.That(this._cp.Hero, Is.Not.Null);
    Assert.That(this._cp.Logger, Is.Not.Null);
    Assert.That(this._cp.Logger, Is.Empty);
}
```

След това ще продължим с тестването на основния метод, който класа предлага - `void Fight(BaseMonster monster)`. Старайте се да обхванете колкото се може повече гранични случаи. Използвайте и други класове на NUnit като например `Does` и `Has`.

### Примерна имплементация (покажи само при затруднение)

```
[Test]
0 references
public void FightCorrectly_WeakerEnemy()
{
    //Arrange
    IMonster monster = new MedusaTheGorgon(level:1);
    this.LevelUpHero(50);

    //Act
    this._cp.Fight(monster);
    var logger = this._cp.Logger;

    //Assert - expected monster to die in 1 hit
    Assert.That(logger.Count, expression: Is.EqualTo(expected:2));
    Assert.That(logger, expression: Does.Contain(expected:"The Hunter hits the MedusaTheGorgon dealing 510 damage to it.").And.Contains(expected:"The monster dies. (4 XP gained.)"));
}
```

```
[Test]
0 references
public void FightCorrectlyAndRepeatedly_StrongerEnemy()
{
    //Arrange
    IMonster monster = new MedusaTheGorgon(level:50);

    //Act
    this._cp.Fight(monster);
    var logger = this._cp.Logger;

    //Assert - expected hero to die in 1 hit but heal 3 times which gives him an extra turn
    Assert.That(logger, expression: Has.Count.EqualTo(expected:12));
    Assert.That(logger, expression: Does.Contain(expected:"The hero dies on level 1 after 4 fights."));
}
```

### Какво следва?

След като минахме през първите стъпки в писането на unit тестове с NUnit е време да продължим с `mocking` като за целта ще използваме `Moq`.

## 8. Mocking с Moq

За да започнем, на първо място е нужно да инсталираме пакета `Moq`, което може да бъде направено през `NuGet Package Manager`.

### Защо е нужен mocking?

Най-просто казано `mocking`-ът ни е нужен, за да не работим с истински данни повреме на тестване. Освен това с тази техника можем да създадем копия на различни класове, от които нашият тестван клас зависи, но в момента не са обект на нашето тестване. Вместо да създаваме реални такива

класове можем да създадем Mock клас, който да замести тези зависимости. В случай, че се тества репозитори е препоръчително да се използва mocking, за да няма реален досег с базата от данни.

## Как да започнем

Нека инсталираме пакета Moq и се върнем при нашите тестове за герои. Като използваме конструктора на класа Mock, създаваме инстанция от класа, който ще ни трябва. Нека заменим кода в сегашния ни Setup метод.

```
[Setup]
0 references
public void Setup()
{
    this._hero = new Mock<Mage>();
    this._hero.Protected()
        .Setup( voidMethodName: "LevelUp")
        .CallBase();
}
```

Moq позволява да се зададе и функционалност за всеки метод. За да работят нашите тестове върху метода LevelUp, трябва да окажем, че искаме той да се изпълнява. Това става с помощта на Moq.Protected, в метода Setup подаваме като символен низ името на метода ни (понеже е защитен) а с метода CallBase оказваме, че желаем реалния метод да се извика.

Това е всичко, което ни трябва за сега. Ще забележите, че тестовете вече не могат да бъдат проведени. За да достъпим инстанцията на нашият Mock обект използваме свойството му Object.

```
[Test]
✓ | 0 references
public void TakeHitCorrectly()
{
    //Act
    var damage = 50;
    this._hero.Object.TakeHit(damage);

    //Assert
    Assert.That(this._hero.Object.Health, expression: Is.EqualTo(expected: HeroConstants.InitialMaxHealth
        - damage + HeroConstants.InitialArmor));
}
```

По аналогичен начин използвайте mocking при тестването и на другите класове.

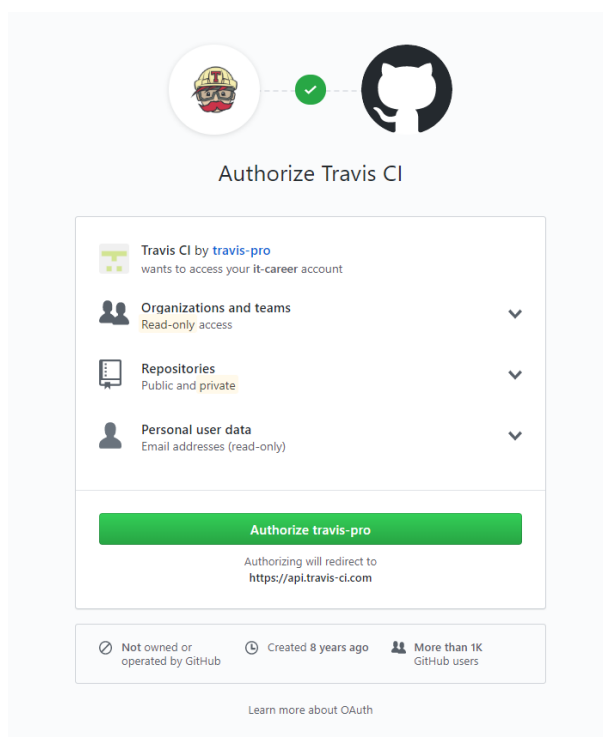
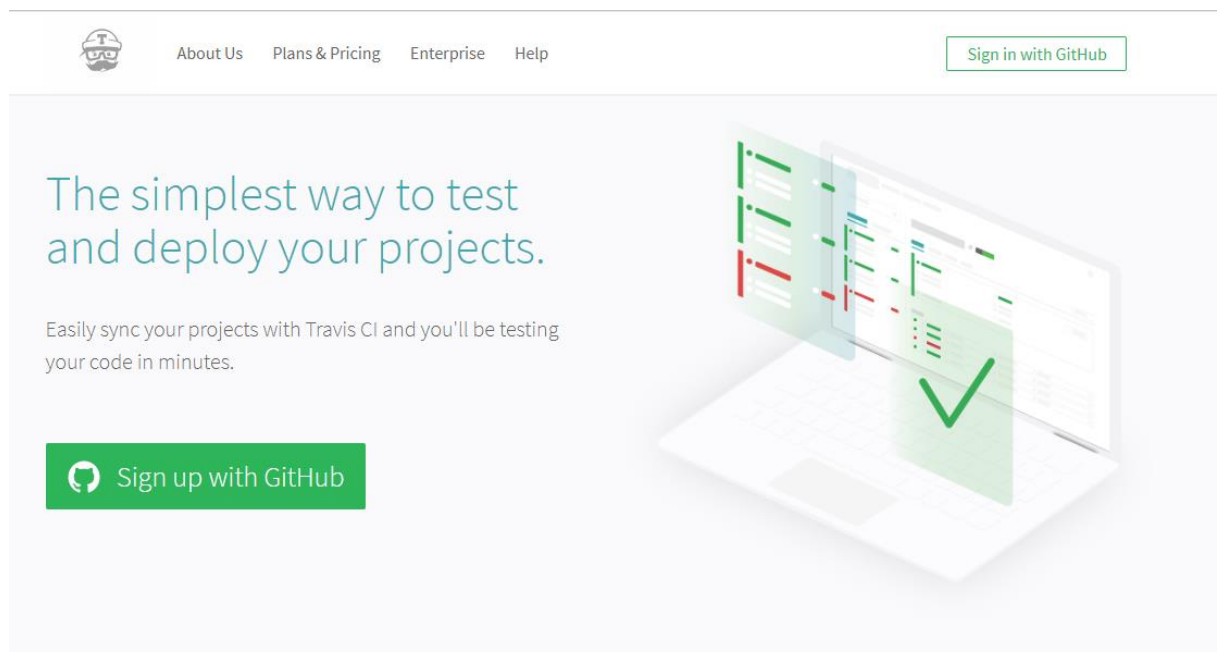
Информация относно възможностите на Moq можете да откриете и в GitHub репозиторието на библиотеката - <https://github.com/Moq/moq4/wiki/Quickstart>

## 9. Непрекъснатата интеграция

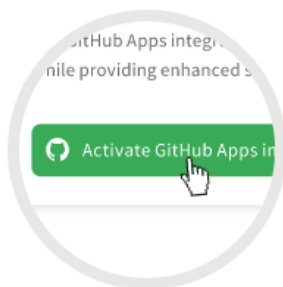
След като накратко се запознахме с unit тестването със C# и mocking с Moq, сега е време да преминем към следващата стъпка – непрекъснатата интеграция. За целта ще използваме Travis CI и GitHub.

### Как да започнем

Влезте чрез GitHub акаунта си в <https://travis-ci.com/>, след което синхронизирайте репозиторията, върху което искате да имплементирате непрекъснатата интеграция.



При поискване, дайте права на Travis CI да достъпи вашия GitHub акаунт. Ще бъдете пренасочени към страница, съдържаща инструкции как да започнете първата си непрекъсната интеграция. Насочете се към “Activate all repositories using GitHub Apps”.



### 1 Activate your GitHub repositories

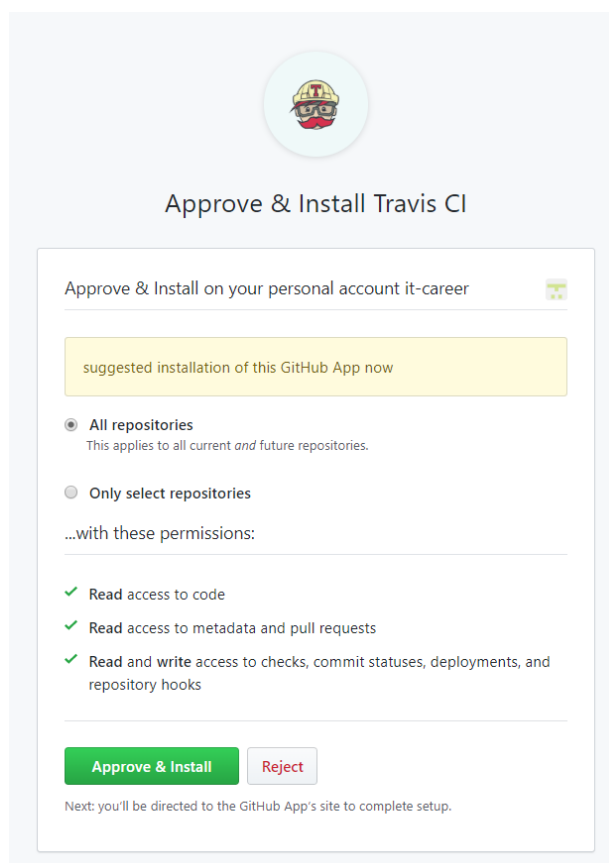
Once you're signed in, go to your profile page where you'll see all the organizations you're a member of.

You can install the GitHub App integration by clicking the Activate button for each organization you would like to use with Travis CI.

Please note: You need to be an admin for any repositories you want to setup on Travis CI.

 [Activate all repositories using GitHub Apps](#)

Отново ще бъдете питани за разрешение, което трябва да дадете за да продължи процеса.



След това ще бъдете пренасочени към страница с всичките ви репозитория от GitHub акаунтът ви. Изберете репозитория върху което искате да работите. След това добавете нов файл в основата (root) на репозитория – файлът трябва е наименован .travis.yml и да съдържа инструкции какво да прави за нас Travis CI. Съдържанието на файла варира според използвания език, библиотеката за тестване, какви скриптове искаме да се изпълняват и др. Файлът, който ние ще използваме за нашето .NET Core приложение с тестове на NUnit е следния:

```
language: csharp
mono: none
sudo: false
dist: bionic
dotnet: 3.0
before_install:
- sudo apt-get install -y dotnet-runtime-2.2
script:
- dotnet --version
- cd ./TestingExercise
- dotnet restore
- dotnet build
- dotnet test ./HeroesGame.Tests/HeroesGame.Tests.csproj
```

Конфигурациите, които задаваме са за .NET Core, като след като инсталираме dotnet-runtime проверяваме дали инсталацията е била успешна с командата dotnet --version и ако е била такава навигираме до папката на проекта си, след което го build-ваме и пускаме тестовите с командата dotnet test. Ако всички тестове минат ще получим одобрение в travis-ci интерфейса.

За да започне Travis CI да следи вашето репозитори, трябва първо да добавите .travis.yml файлът (т.е. да го качите в репозиторието – най-външната/git директория).

it-career / HeroesGame Private

Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Actions Projects 0 Security Insights Settings

No description, website, or topics provided. Edit

Manage topics


18 commits 1 branch 0 packages 0 releases


Branch: master New pull request Create new file Upload files Find file Clone or download


User	Commit	Time
User Bug fix 11	Latest commit bd7a99b	12 minutes ago
TestingExercise	Final	17 hours ago
.gitignore	Initial commit	5 days ago
.travis.yml	Bug fix 11	12 minutes ago
README.md	Initial commit	5 days ago


След тази стъпка Travis CI вече следи проекта ни – нека се върнем в клиента му на адрес [https://travis-ci.com/име\\_на\\_GitHub\\_акаунт/<име\\_на\\_проекта>](https://travis-ci.com/име_на_GitHub_акаунт/<име_на_проекта>)


След определено време, ако всичко е наред ще видим следния екран оказващ, че commit-а, който току що сме направили за да добавим .travis.yml файла не носи никакви грешки в проекта:


 master Bug fix 11


 #12 passed


 Restart build


 Commit bd7a99b


 Compare def3693...bd7a99b


 Branch master


 Ran for 1 min 45 sec

 14 minutes ago

 Debug build


 it-career


 Mono: none C#


 AMD64


От сега нататък Travis ще следи всеки последващ commit и ще проверява дали пречи по някакъв начин на build-а на проекта или пък проваля някой от unit тестовите ни. Нека направим минимална промяна в проекта – например да добави коментар в някой от файловете, за да сме сигурни, че това няма да провали тестовите или build-а, след което ще видим как Travis CI действа.


След като сме направили промени и сме ги качили в репозиторието на проекта ни можем да се върнем към Travis CI.


 master Introduced comments


 #13 started


 Cancel build


 Commit 57a07c6


 Compare bd7a99b...57a07c6

 Branch master


 Running for 13 sec


 it-career


 Mono: none C#


 AMD64


Новият commit е разпознат и скрипта ни е в действие. Обърнете внимание, че това време в зависимост от големината на скрипта (т.е. командите, които трябва да бъдат изпълнени) и големината на проекта може да е няколко минути.


 master Introduced comments


 #13 passed


 Restart build


 Commit 57a07c6


 Compare bd7a99b...57a07c6


 Branch master


 Ran for 1 min 51 sec

 less than a minute ago

 Debug build

 it-career

 Mono: none C#

 AMD64

В нашият случай отнема около 1 минута и 51 секунди, след което получаваме информация, че всичко е успешно и commit-а по никакъв начин не ощетява проекта ни.

Това е краят на упражнението. Вече знаете как да пишете unit тестове, използвайки нереални обекти (mocking) и да имплементирате в някаква форма продължителна интеграция.