FREE eBook

LEARNING

moq

Free unaffiliated eBook created from
Stack Overflow contributors.

#moq

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: moq

It is an unofficial and free moq ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official moq.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with moq

Remarks

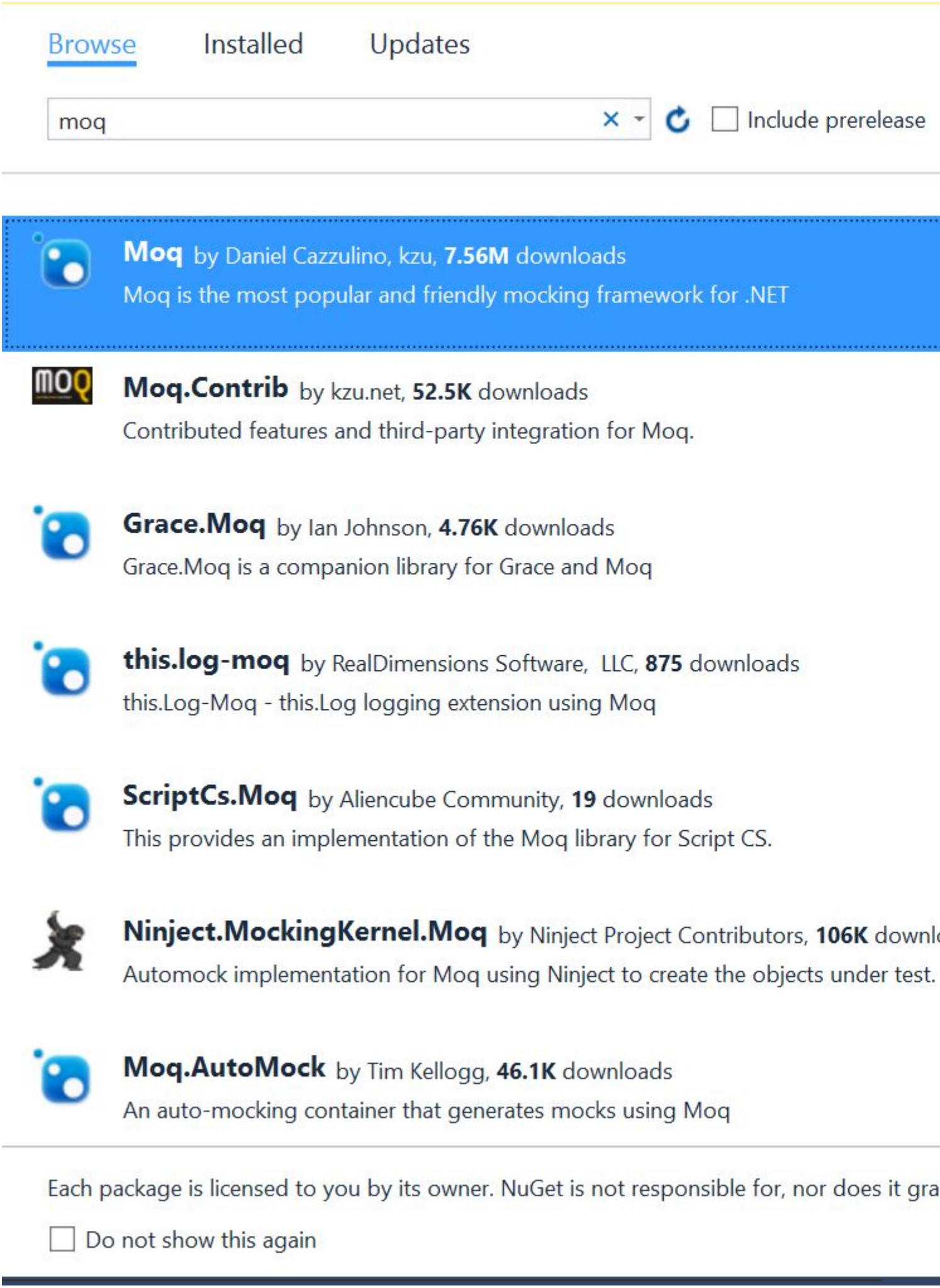
Moq is a mocking library for .Net. It allows interactions with dependencies to be simulated and verified in order to facilitate unit testing.

Release notes for different version of Moq can be found here.

Examples

Installation or Setup

1. Select the project you want to add the reference to Moq.
2. Open Nuget for this project.
3. Select "Browse" than type "moq" at the search box.
4. Select "Moq" and than click on Install.



Following these steps will install the Moq package and will add a reference to it in the selected pro

```
Using Moq;
```

https://riptutorial.com/moq/topic/5717/getting-started-with-moq

Chapter 2: Mocking Behavior

Syntax

- mock.Setup(expression).Returns(value) //Whenever called the method in the expression will return value

Parameters

Parameter	Details
expression	Lambda expression that specifies the method invocation.

Examples

No-Argument method mocking

```
interface Mockable {
    bool DoSomething();
}

var mock = new Mock<Mockable>();
mock.Setup(x => x.DoSomething()).Returns(true);

var result = mock.Object.DoSomething(); //true
```

Mocking void methods to confirm what they return

```
var logRepository = new Mock<ILogRepository>();
logRepository.Setup(x => x.Write(It.IsAny<Exception>(), It.IsAny<string>(),
It.IsAny<string>(), It.IsAny<string>(), It.IsAny<string>(), It.IsAny<string>()))
  .Verifiable();
```

In this case, we are using the Verifiable to ensure that it runs.

We could also use a callback here:

```
logRepository.Setup(x => x.Write(It.IsAny<Exception>(), It.IsAny<string>(),
It.IsAny<string>(), It.IsAny<string>(), It.IsAny<string>(), It.IsAny<string>()))
     .Callback<Exception, string, string, string, string, string>((ex, caller, user, machine,
source, message) => { Console.WriteLine(message); });
```

this would log the output from the method to standard output on the console (many testing frameworks let you capture that output into their runner)

Mocking protected members

To mock a protected member you must first include the following at the top of your test fixture:

```
using Moq.Protected;
```

You then call `Protected()` on your mock, after which you can use the generic `Setup<>` with the return type of your method.

```
var mock = new Mock<MyClass>();
mock.Protected()
     .Setup<int>("MyProtectedGetIntMethod")
     .Returns(1);
```

If the method returns void then use the non-generic `Setup()`.

Read Mocking Behavior online: https://riptutorial.com/moq/topic/8156/mocking-behavior

Chapter 3: Mocking common interfaces

Examples

Mocking IEnumerable

Mocking an interface that inherits from `IEnumerable` to return canned data is quite straightforward. Assuming the following classes:

```
public class DataClass
{
    public int Id { get; set; }
}

public interface IEnumerableClass : IEnumerable<DataClass>
{
}
```

The following approach can be taken. First, create a list containing the information that needs to be returned by the mock:

```
IList<DataClass> list = new List<DataClass>();
for (int i = 0; i < 10; i++)
{
    list.Add(new DataClass { Id = 20 + i });
}
```

Then create a mock of the `IEnumerable` class and setup its `GetEnumerator` method to return the list's enumerator instead:

```
var mock = new Mock<IEnumerableClass>();
mock.Setup(x => x.GetEnumerator()).Returns(list.GetEnumerator());
```

This can be validated as follows:

```
int expected = 20;
foreach (var i in mock.Object)
{
    Assert.AreEqual(expected++, i.Id);
}
```

Read Mocking common interfaces online: https://riptutorial.com/moq/topic/6776/mocking-common-interfaces

Chapter 4: Mocking properties

Examples

Auto stubbing properties

Sometimes you want to mock a class or an interface and have its properties behave as if they were simple getters and setters. As this is a common requirement, Moq provides a short cut method to setup all properties of a mock to store and retrieve values:

```
// SetupAllProperties tells mock to implement setter/getter funcationality
var userMock = new Mock<IUser>().SetupAllProperties();

// Invoke the code to test
SetPropertiesOfUser(userMock.Object);

// Validate properties have been set
Assert.AreEqual(5, userMock.Object.Id);
Assert.AreEqual("SomeName", userMock.Object.Name);
```

For completeness, the code being tested is below

```
void SetPropertiesOfUser(IUser user)
{
    user.Id = 5;
    user.Name = "SomeName";
}
```

Properties with private setters

Sometimes you want to create a mock of a class that has a private setter:

```
public class MockTarget
{
    public virtual string PropertyToMock { get; private set; }
}
```

Or an interface that only defines a getter:

```
public interface MockTarget
{
    string PropertyToMock { get; }
}
```

In both cases, you can ignore the setter and simply Setup the property getter to return a desired value:

```
var mock = new Mock<MockTarget>();
mock.SetupGet(x => x.PropertyToMock).Returns("ExpectedValue");
```

```
Assert.AreEqual("ExpectedValue", mock.Object.PropertyToMock);
```

Read Mocking properties online: https://riptutorial.com/moq/topic/6774/mocking-properties

Chapter 5: Validating call order

Examples

Validating call order implicitly

Where a method to be tested uses information from one call to pass on to subsequent calls, one approach that can be used to ensure the methods are called in the expected order is to setup the expectations to reflect this flow of data.

Given the method to test:

```
public void MethodToTest()
{
    var str = _utility.GetInitialValue();

    str = _utility.PrefixString(str);
    str = _utility.ReverseString(str);

    _target.DoStuff(str);
}
```

Expectations can be set to pass data from `GetInitialValue` through `PrefixString` and `ReverseString` to `DoStuff`, where the information is verified. If any of the methods are called out of order, the end data will be wrong and the test will fail.

```
// Create mocks
var utilityMock = new Mock<IUtility>();
var targetMock = new Mock<ITarget>();

// Setup expectations, note that the returns value from one call matches the expected
// parameter for the next call in the sequence of calls we're interested in.
utilityMock.Setup(x => x.GetInitialValue()).Returns("string");
utilityMock.Setup(x => x.PrefixString("string")).Returns("Prefix:string");
utilityMock.Setup(x => x.ReverseString("Prefix:string")).Returns("gnirts:xiferP");

string expectedFinalInput = "gnirts:xiferP";

// Invoke the method to test
var sut = new SystemUnderTest(utilityMock.Object, targetMock.Object);
sut.MethodToTest();

// Validate that the final call was passed the expected value.
targetMock.Verify(x => x.DoStuff(expectedFinalInput));
```

Validating call order with callbacks

When you can't / don't want to use Strict Mocks, you can't use `MockSequence` to validate call order. An alternate approach is to use callbacks to validate that the `Setup` expectations are being invoked in the expected order. Given the following method to test:

```
public void MethodToTest()
{
    _utility.Operation1("1111");
    _utility.Operation3("3333");
    _utility.Operation2("2222");
}
```

It can be tested as follows:

```
// Create the mock (doesn't have to be in strict mode)
var utilityMock = new Mock<IUtility>();

// Create a variable to track the current call number
int callOrder = 1;

// Setup each call in the sequence to be tested.  Note that the callback validates that
// that callOrder has the expected value, then increments it in preparation for the next
// call.
utilityMock.Setup(x => x.Operation1(It.IsAny<string>()))
           .Callback(() => Assert.AreEqual(1, callOrder++, "Method called out of order") );
utilityMock.Setup(x => x.Operation2(It.IsAny<string>()))
           .Callback(() => Assert.AreEqual(2, callOrder++, "Method called out of order") );
utilityMock.Setup(x => x.Operation3(It.IsAny<string>()))
           .Callback(() => Assert.AreEqual(3, callOrder++, "Method called out of order") );


// Invoke the method to be tested
var sut = new SystemUnderTest(utilityMock.Object);
sut.MethodToTest();


// Validate any parameters that are important, note these Verifications can occur in any
// order.
utilityMock.Verify(x => x.Operation2("2222"));
utilityMock.Verify(x => x.Operation1("1111"));
utilityMock.Verify(x => x.Operation3("3333"));
```

Validating call order with MockSequence

Moq provides support for validating call order using `MockSequence`, however it only works when using Strict mocks. So, Given the following method to test:

```
public void MethodToTest()
{
    _utility.Operation1("1111");
    _utility.Operation2("2222");
    _utility.Operation3("3333");
}
```

It can be tested as follows:

```
// Create the mock, not MockBehavior.Strict tells the mock how to behave
var utilityMock = new Mock<IUtility>(MockBehavior.Strict);

// Create the MockSequence to validate the call order
var sequence = new MockSequence();
```

```
// Create the expectations, notice that the Setup is called via InSequence
utilityMock.InSequence(sequence).Setup(x => x.Operation1(It.IsAny<string>()));
utilityMock.InSequence(sequence).Setup(x => x.Operation2(It.IsAny<string>()));
utilityMock.InSequence(sequence).Setup(x => x.Operation3(It.IsAny<string>()));

// Run the method to be tested
var sut = new SystemUnderTest(utilityMock.Object);
sut.MethodToTest();

// Verify any parameters that are cared about to the operation being orchestrated.
// Note that the Verify calls can be in any order
utilityMock.Verify(x => x.Operation2("2222"));
utilityMock.Verify(x => x.Operation1("1111"));
utilityMock.Verify(x => x.Operation3("3333"));
```

The above example uses `It.IsAny<string>` when setting up the expectations. These could have used relevant strings ("1111", "2222", "3333") if more exact matches were required.

The error reported when calls are made out of sequence can be a bit misleading.

> invocation failed with mock behavior Strict. All invocations on the mock must have a corresponding setup.

This is because, each `Setup` expectation is treated as if it doesn't exist until the previous expectation in the sequence has been satisfied.

Read Validating call order online: https://riptutorial.com/moq/topic/6775/validating-call-order

Credits

S. No	Chapters	Contributors
1	Getting started with moq	Community, forsvarir, meJustAndrew, Old Fox
2	Mocking Behavior	J. Pichardo, jcolebrand, Owen Pauling
3	Mocking common interfaces	forsvarir
4	Mocking properties	forsvarir, Owen Pauling
5	Validating call order	forsvarir