Unit Testing

4 Golden Strategies for Unit Testing DateTime.Now in C#

Kristijan Kralj

Who hasn't had a bug in their code that happened once every year, but they couldn't reproduce it?

That bug was probably caused by using *DateTime.Now* in their code, and if you debug it, it won't show up. With the right unit tests, you can make sure your code's assumptions about *DateTime.Now* will hold true.

Although there are other approaches, we will cover 4 golden strategies to help you unit test DateTime.Now in C#.

The most popular strategies for unit testing DateTime.Now in C# are:

- **An interface that wraps the DateTime.Now**
- **SystemTime static class**
- **Ambient context approach**
- **DateTime property on a class**

Let's go over the strategies, see implementations, advantages, and disadvantages.

Table of Contents

What's the problem we are trying to solve?

When writing unit tests (https://methodpoet.com/unit-testing-in-c/), many details need to be considered.

One such detail in unit testing is the *DateTime.Now* property that represents the current system date and time. Now, this is a problem because the value changes all the time.

This means that if you need to unit test code that makes heavy use of *DateTime.Now*, you need to freeze the time somehow. But you have to make sure to reset the time between every single test run. This can be problematic because what would happen if the test fails in the middle of testing? Should the test runner automatically rerun the tests, or should the tests be skipped?

The example we will test in this post is:

```
public class EventChecker
{
    public bool CanScheduleEvent()
    {
        return DateTime.Now.DayOfWeek == DayOfWeek.Saturday
            || DateTime.Now.DayOfWeek == DayOfWeek.Sunday;
    }
}
```

This is a simple class that has a dependency on *DateTime.Now*. It checks if an event can be scheduled or not. The event can be scheduled if the current day of the week is Saturday or Sunday. I want to write 2 tests for it, one that passes, one that fails.

To do that, I need to change the current implementation to control the current DateTime.

IDateTimeProvider

"We can solve any problem by introducing an extra level of indirection."

– ANDREW KOENIG

(...except for the problem of too many levels of indirection.)
(https://en.wikipedia.org/wiki/Fundamental_theorem_of_software_engineering)

One general approach you can use when you need to write unit tests for a static method or class in the code is to replace it with an interface wrapper.

Let me show you one example of this kind of interface.

```
public interface IDateTimeProvider
{
    DateTime GetCurrentTime();
}

public class DateTimeProvider : IDateTimeProvider
{
    public DateTime GetCurrentTime()
    {
        return DateTime.Now;
    }
}
```

Now I can use the dependency injection to pass the interface.

```
public class EventChecker
{
    private IDateTimeProvider _dateTimeProvider;

    public EventChecker(IDateTimeProvider dateTimeProvider)
    {
        _dateTimeProvider = dateTimeProvider;
    }

    public bool CanScheduleEvent()
    {
        return _dateTimeProvider.GetCurrentTime().DayOfWeek == DayOfWeek.Saturday
            || _dateTimeProvider.GetCurrentTime().DayOfWeek == DayOfWeek.Sunday;
    }
}
```

The way it works is where previously the *DateTime.Now* property was called, now the code will call the *GetCurrentTime* method on the *IDateTimeProvider* interface.

And now it's easy to write tests for the code. The only missing thing is a fake implementation of the *IDateTimeProvider*.

```
class FakeDateTimeProvider : IDateTimeProvider
{
    private DateTime _time;

    public FakeDateTimeProvider(DateTime time)
    {
        _time = time;
    }

    public DateTime GetCurrentTime()
    {
        return _time;
    }
}

[Fact]
public void CanScheduleEvent_returns_true_when_day_is_sunday()
{

    var eventChecker = new EventChecker(new FakeDateTimeProvider(new DateTime(202

    var result = eventChecker.CanScheduleEvent();

    Assert.True(result);
}

[Fact]
public void CanScheduleEvent_returns_false_when_day_is_monday()
{
    var eventChecker = new EventChecker(new FakeDateTimeProvider(new DateTime(202

    var result = eventChecker.CanScheduleEvent();

    Assert.False(result);
}
```

FakeDateTimeProvider is an implementation that takes a *DateTime* parameter as a constructor argument. If you use a mocking framework such as moq (https://github.com/moq/moq4)or NSubstitute (https://nsubstitute.github.io/), writing a test should be easier since you don't have to implement the *IDateTimeProvider* manually.

Advantages

The advantage of this approach is that it's easy to set up and write the test. Every test case gets a fresh object instance of the *FakeTimeProvider*. This means no state sharing between test (https://methodpoet.com/state-vs-interaction-based-testing/) methods.

Disadvantages

Every class that you need to test needs to have this additional dependency. If a class already has many dependencies, one more will further complicate the initialization process.

SystemTime static class

The next approach is to change one static property with another. Yes, this doesn't sound like it's a proper solution for this problem. But actually, it is straightforward, and that's why I include it here.

The approach is to define a *SystemTime* class.

```
public static class SystemTime
{
    public static Func<DateTime> Now = () => DateTime.Now;

    public static void SetDateTime(DateTime dateTimeNow)
    {
        Now = () => dateTimeNow;
    }

    public static void ResetDateTime()
    {
        Now = () => DateTime.Now;
    }
}

public class EventChecker
{
    public bool CanScheduleEvent()
    {
        return SystemTime.Now().DayOfWeek == DayOfWeek.Saturday
            || SystemTime.Now().DayOfWeek == DayOfWeek.Sunday;
    }
}
```

The class has three members:

- Now – this function returns by default the current date and time. However, the definition of this function can be changed by the other two methods.
- SetDateTime – this method sets a custom *DateTime* object to be returned by the Now function.
- ResetDateTime – this method resets the Now function back to *DateTime.Now*.

And the tests:

```
[Fact]
public void CanScheduleEvent_returns_true_when_day_is_sunday()
{
    SystemTime.SetDateTime(new DateTime(2021, 4, 4));
    var eventChecker = new EventChecker();

    var result = eventChecker.CanScheduleEvent();

    Assert.True(result);
}

[Fact]
public void CanScheduleEvent_returns_false_when_day_is_monday()
{
    SystemTime.SetDateTime(new DateTime(2021, 4, 5));
    var eventChecker = new EventChecker();

    var result = eventChecker.CanScheduleEvent();

    Assert.False(result);
}
```

One thing you need to have in mind to reset the SystemTime after every test method is completed. You can do that using some teardown mechanism. If you are using NUnit, add a method with the TearDown attribute (https://docs.nunit.org/articles/nunit/writing-tests/setup-teardown/index.html). If you are using the xUnit test framework, then your test class needs to implement IDisposable (https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=net-5.0). Example of the Dispose method:

```
public void Dispose()
{
    SystemTime.ResetDateTime();
}
```

Advantages

The advantage of this approach is its simplicity. The code pretty much stays the same. With a custom class, you get greater flexibility with unit testing.

Disadvantages

The biggest disadvantage with this approach is that you increased the cost of maintainability of your tests. You need to make sure you reset the SystemTime after every test.

The other disadvantage is that your tests are sharing test data. If you run them in parallel, one test can affect the outcome of the other test.

DateTime property

This approach is straightforward, but it requires you to modify the class being tested. You add a new property that you will use instead of DateTime.Now. The other thing you need to add is a method to the current date and time.

```
public class EventChecker
{
    public void SetDateTime(DateTime? newTime)
    {
        _dateTime = newTime;
    }

    private DateTime? _dateTime;
    public DateTime Time
    {
        get
        {
            if (!_dateTime.HasValue)
            {
                return DateTime.Now;
            }
            return _dateTime.Value;
        }
    }

    public bool CanScheduleEvent()
    {
        return Time.DayOfWeek == DayOfWeek.Saturday
            || Time.DayOfWeek == DayOfWeek.Sunday;
    }
}
```

And now the tests.

```
[Fact]
public void CanScheduleEvent_returns_true_when_day_is_sunday()
{
    var eventChecker = new EventChecker();
    eventChecker.SetDateTime(new DateTime(2021, 4, 4));

    var result = eventChecker.CanScheduleEvent();

    Assert.True(result);
}

[Fact]
public void CanScheduleEvent_returns_false_when_day_is_monday()
{
    var eventChecker = new EventChecker();
    eventChecker.SetDateTime(new DateTime(2021, 4, 5));

    var result = eventChecker.CanScheduleEvent();

    Assert.False(result);
}
```

Advantages

This approach doesn't require dependency injection. Nor it requires a custom static object. You exten
original class, and you can test it. In fact, this approach is so simple that I haven't seen it on

StackOverflow.

Disadvantages

The bad thing about this approach is that you need to modify with the same code every class you want to test this way. This leads to code duplication.

Ambient Context approach

The ambient context pattern is a strategy for managing the context of the user's activity across multiple devices. Context is a layer of data that is associated with a user's activity. This data can include location, application, time, device, user, and action being performed.

Let's start with the first piece of the puzzle, DateTimeProvider.

```
public class DateTimeProvider
{
    public static DateTime Now
        => DateTimeProviderContext.Current == null
                ? DateTime.Now
                : DateTimeProviderContext.Current.ContextDateTimeNow;
}
```

The implementation is straightforward. It has one static field, Now, that returns the current time if the DateTimeProviderContext.Current is null. Otherwise, it uses the DateTimeProviderContext.Current.

And what is this DateTimeProviderContext? Well, I'm glad you've asked.

```csharp
public class DateTimeProviderContext : IDisposable
{
    internal DateTime ContextDateTimeNow;
    private static ThreadLocal<Stack> ThreadScopeStack = new ThreadLocal<Stack>((

    public DateTimeProviderContext(DateTime contextDateTimeNow)
    {
        ContextDateTimeNow = contextDateTimeNow;
        ThreadScopeStack.Value.Push(this);
    }

    public static DateTimeProviderContext Current
    {
        get
        {
            if (ThreadScopeStack.Value.Count == 0)
                return null;
            else
                return ThreadScopeStack.Value.Peek() as DateTimeProviderContext;
        }
    }

    public void Dispose()
    {
        ThreadScopeStack.Value.Pop();
    }
}
```

This class uses ThreadLocal to ensure access to it is thread-safe. It also implements IDisposable. So that the current value gets removed once the instance is being disposed.

The production code is only slightly changed.

```csharp
public class EventChecker
{
    public bool CanScheduleEvent()
    {
        return DateTimeProvider.Now.DayOfWeek == DayOfWeek.Saturday
            || DateTimeProvider.Now.DayOfWeek == DayOfWeek.Sunday;
    }
}
```

And the tests use the DateTimeProviderContext to set the custom date.

```csharp
[Fact]
public void CanScheduleEvent_returns_true_when_day_is_sunday()
{
    using var context = new DateTimeProviderContext(new DateTime(2021, 4, 4));
    var eventChecker = new EventChecker();

    var result = eventChecker.CanScheduleEvent();

    Assert.True(result);
}

[Fact]
public void CanScheduleEvent_returns_false_when_day_is_monday()
{
    using var context = new DateTimeProviderContext(new DateTime(2021, 4, 5));
    var eventChecker = new EventChecker();

    var result = eventChecker.CanScheduleEvent();

    Assert.False(result);
}
```

Advantages

The usage is similar to *DateTime.Now*, which is nice. And you don't have to perform additional changes throughout the code. The biggest advantage is that with this solution is possible to run unit tests in parallel.

Disadvantages

One disadvantage is that the implementation is slightly complicated. It's not easy at first to see what's going on in the code.

Conclusion

As you could see, faking *DateTime* is not too hard. You have seen 4 different approaches to test the production code that contains *DateTime.Now*. Which one to use, that's up to you.

Just remember to always use the simplest solution for your current codebase.