Lab 02 - Adding Unit Tests

Objectives

In this lab, you will add unit tests for the scheduling features of the project. The current code makes calls to the static `DateTimeOffset.Now` property. This makes unit tests that have to deal with "now" very difficult. First, you will change calls that get the current time so that you can provide your own value for testing. Then you will work with fake objects for the abstractions created in Lab 01. At the end, you will have a set of testing techniques that are transferable to other projects.

Application Overview

Please refer to the "Lab00-ApplicationOverview.md" file for an overview of the application, the projects, and how to run the application.

As a reminder, the "Starter" folder contains the code files for this lab.

> The state of the "Starter" files for Lab 02 are the same as the "Completed" files for Lab 01. So if you completed Lab 01, you can continue coding in the same solution.

Objectives

1. Create a set of unit tests for the Schedule class. The test methods are stubbed out in the `HouseControl.Library.Test` project.
2. Create a set of unit tests for the `ScheduleHelper` class. This class has "Roll" methods to get the next day, weekday, and weekend day. The following methods should have tests:
    - RollForwardToNextDay
    - RollForwardToNextWeekdayDay
    - RollForwardToNextWeekendDay
3. To facilitate the above unit tests, create an abstraction for `DateTimeOffset.Now` using the `TimeProvider` type that is part of .NET 8 and .NET 9.

BONUS

4. Create helper methods for `Today`, `IsInPast`, and `DurationFromNow` along with appropriate tests.

Hints

General

- A unit testing project has been stubbed out using xUnit: `HouseControl.Library.Test`. If you are more comfortable with a different framework (such as NUnit or MSTest), feel free to use that instead.

Unit Testing DateTime.Now

- `DateTimeOffset` is a `DateTime` that also includes timezone information.
- Find all references to `DateTimeOffset.Now` in the code. These should be limited to 2 classes: `Schedule` and `ScheduleHelper`.
- Add a field for the abstract `TimeProvider` type that was added in .NET 8.
- `TimeProvider` has a static `System` property that uses the real time.
- Make the time provider an optional constructor parameter so that it can be changed for a fake time provider unit testing.

Schedule Tests

- A shell class with unit test methods has already been stubbed out: `HouseControl.Library.Test/ScheduleTests.cs`.
- A file with test data is already set up for the tests to use. The test class has a variable with the location (`ScheduleData`).
- You will need a `SunsetProvider`, however, no values from the SunsetProvider will be used in these tests.
- You can create a ScheduleItem with the type "Once" with the following code:

```
new ScheduleItem(
    1, // device number
    DeviceCommand.On, // command
    new ScheduleInfo()
    {
        EventTime = DateTimeOffset.Now.AddMinutes(-2), // time 2 minutes in the past
        Type = ScheduleType.Once, // schedule type "Once"
    },
    true, // enabled
    "" // schedule set
);
```

ScheduleHelper Tests

- The application only deals with the local timezone (meaning, the timezone of the machine the application runs on). The application does not need to support multiple locations or alternate time zones.
- When setting times for tests, you will need to include a time zone offset (which is a time span). You may want to create a field such as the following:

```
TimeSpan timeZoneOffset = new TimeSpan(-2, 0, 0);
```

- To create a `ScheduleInfo` object for testing, only the `EventTime` (`DateTimeOffset`) and `TimeType` (`ScheduleTimeType`) are required. Default values can be used for the other properties.

BONUS "Now" Helper Methods

- Helper methods fit well in the ScheduleHlper class.
- Helper method for `Today` is `Now` with just the date portion. (Note: I would recommend keeping this as a `DateTimeOffset` due to calculations in existing code.)
- Helper method for `IsInPast` compares a DateTimeOffset with `Now` and returns a boolean.
- Helper method for `DurationFromNow` subtracts a `DateTimeOffset` from `Now` and then uses `.Duration()` to return a timespan. Note: `Duration()` returns an absolute value, meaning the result will never be negative.
- Create unit tests for these helper methods.

If this is enough information for you, and you want a bit of a challenge, **stop reading here**. Otherwise, continue reading for step-by-step instructions.

Step-by-Step

Keep scrolling for the step-by-step walkthrough. Or use the links below to jump to a section:

Step-by-Step - Unit Testing DateTime.Now

1. Centralize all calls to `DateTimeOffset.Now`. This will make the abstraction easier.

2. Open the `ScheduleHelper.cs` file in the `HouseControl.Library` project.

3. Create a helper method on the `ScheduleHelper` class for `Now`.

```
public DateTimeOffset Now()
{
    return DateTimeOffset.Now;
}
```

4. In the `ScheduleHelper` class, search for references to `DateTimeOffset.Now` and update them to use `Now()`.

```
public DateTimeOffset Tomorrow()
{
    //return new DateTimeOffset(
    //    DateTimeOffset.Now.Date.AddDays(1),
    //    DateTimeOffset.Now.Offset);
    return new DateTimeOffset(
        Now().Date.AddDays(1),
```

```
        Now().Offset);
}
```

```
public bool IsInFuture(DateTimeOffset checkTime)
{
    // return checkTime > DateTime.Now;
    return checkTime > Now();
}
```

You should have 3 replacements in `ScheduleHelper`.

5. In the `Schedule` class, search for references to `DateTimeOffset.Now` and update them to use `scheduleHelper.Now`.

```
//DateTimeOffset today = DateTimeOffset.Now.Date;
DateTimeOffset today = scheduleHelper.Now().Date;
```

```
//(si.Info.EventTime - DateTimeOffset.Now).Duration() < TimeSpan.FromSeconds(30))
(si.Info.EventTime - scheduleHelper.Now()).Duration() < TimeSpan.FromSeconds(30))
```

```
//while (currentItem.Info.EventTime < DateTimeOffset.Now)
while (currentItem.Info.EventTime < 3cheduleHelper.Now())
```

There are 3 replacements in `Schedule`.

You should be able to buid and run the application at this point.

6. In the `ScheduleHelper` class, create a new private field for a `TimeProvider`.

```
private readonly TimeProvider timeProvider;
```

7. In the `ScheduleHelper` constructor, set the new field to the default system time provider.

```
public ScheduleHelper(ISunsetProvider sunsetProvider)
{
    this.SunsetProvider = sunsetProvider;
```

```
    this.timeProvider = TimeProvider.System;
}
```

8. In the `ScheduleHelper` class, create a new property to hold a `TimeProvider`.

```
private TimeProvider? helperTimeProvider;
public TimeProvider HelperTimeProvider
{
    get { return helperTimeProvider; }
    set { helperTimeProvider = value; }
}
```

9. Update the `get` method so that if the backing field is null, set the backing field to the `CurrentTimeProvider` and return the field.

```
get
{
    if (helperTimeProvider is null)
        helperTimeProvider = TimeProvider.System;
    return helperTimeProvider;
}
```

> The result is that the the System (a.k.a. "real" time provider) is used by default. So if you do not explicitly set the `HelperTimeProvider` property, it uses the current time. This is the behavior you want in production. But for testing, you now have a property to override to provide your own value of `Now`.

10. Using the technique from Lab 01, shorten the code using the null coalescing operator (`??`).

```
get
{
    return helperTimeProvider ??= TimeProvider.System;
}
```

11. The property `get`/`set` can also be updated to use expression-bodied members. Here is the final property:

```
private TimeProvider? helperTimeProvider;
public TimeProvider HelperTimeProvider
{
    get => helperTimeProvider ??= TimeProvider.System;
    set => helperTimeProvider = value;
}
```

12. Update the `Now` method to use the new property.

```
public DateTimeOffset Now()
{
    return HelperTimeProvider.GetLocalNow();
}
```

> Note: The `GetLocalNow` method returns a `DateTimeOffset` with the current time and time zone offset.

With the new property in place for getting the current time, you now have a plug-in point for setting your own "now" time in unit tests.

Step-by-Step - Schedule Tests

There are a variety of ways to organize tests. For this solution, the test projects match the project under test: `HouseControl.Library.Test` => `HouseControl.Library`. In addition, the test classes match the class under test: `Schedule.Test` => `Schedule`. This is just one approach to organizing tests which works well for this solution.

1. Open the `ScheduleTest.cs` file in the `HouseControl.Library.Test` project.

2. This project is already configured to use xUnit as a testing framework. In addition, a data file used for testing is automatically copied into a folder where the tests have access to it. A field is set to the test file location:

```
readonly string fileName = AppDomain.CurrentDomain.BaseDirectory + "\\ScheduleData";
```

3. Run the existing tests.
   Visual Studio 2022: Open the test explorer from the "Test" menu. Then click the "Run all in View" button on the far left.
   Visual Studio Code: From the command line, navigate to the "HouseControl.Library.Test" folder and type `dotnet test`.

The test results will report "Failed". This is because all of the test stubs have `Assert.Fail()` calls as placeholders.

3. Look at the first test in the project:

```
[Fact]
public void ScheduleItems_OnCreation_IsPopulated()
{
    // Arrange / Act
```

```
    // Assert
    Assert.Fail("Test not implemented");
}
```

The tests use a 3 part naming scheme. The first part is the unit under test (`ScheduleItems`). The second is the action run (`OnCreation`). The third is the expected result (`IsPopulated`). So the name of this test tells us that we expect `ScheduleItems` to be populated when the `Schedule` class is created.

> `ScheduleItems` is not a separate property of the `Schedule` class. The `Schedule` class itself is a `List<ScheduleItem>`. So the class itself is a collection of `ScheduleItems`. For example, if you want to know the number of `ScheduleItems`, use `Schedule.Count` (a property on `List<T>`).

In addition, the tests use Arrange / Act / Assert for organization. The Arrange section has any setup code, such as object creation. The Act section is where the action is performed. The Assert section is where the results are checked. In some instances, such as this test, the Arrange and Act sections are combined (since this covers what happens at object creation).

4. Create a new `Schedule` object.

```
// Arrange / Act
var schedule = new Schedule(fileName, );
```

The `Schedule` constructor takes 2 parameters. The first is the data file name (which is in a field). The second parameter is an `ISunsetProvider` (an interface created in Lab 01).

For this set of tests, create a fake sunset provider.

5. In the test project, create a new class called `FakeSunsetProvider`.

```
namespace HouseControl.Library.Test;

public class FakeSunsetProvider
{
}
```

6. Implement in the `ISunsetProvider` interface.

```
public class FakeSunsetProvider : ISunsetProvider
{
    public DateTimeOffset GetSunrise(DateTime date)
    {
        throw new NotImplementedException();
    }
```

```
    public DateTimeOffset GetSunset(DateTime date)
    {
        throw new NotImplementedException();
    }
}
```

> Note: you may need to add a using statement for `HouseControl.Sunset` if your IDE does not add it for you.

7. Hardcode some values to return from the 2 methods:

```
public DateTimeOffset GetSunrise(DateTime date)
{
    DateTime time = DateTime.Parse("06:15:00");
    return new DateTimeOffset(date.Date + time.TimeOfDay);
}

public DateTimeOffset GetSunset(DateTime date)
{
    DateTime time = DateTime.Parse("20:25:00");
    return new DateTimeOffset(date.Date + time.TimeOfDay);
}
```

8. Back in the test class, use the `FakeSunsetProvider` to create the `Schedule`.

```
// Arrange / Act
var schedule = new Schedule(fileName, new FakeSunsetProvider());
```

9. In the Assert section, make sure that the `Schedule` has more than one item.

```
// Assert
Assert.NotEmpty(schedule);
```

10. Review the completed test.

```
[Fact]
public void ScheduleItems_OnCreation_IsPopulated()
{
    // Arrange / Act
    var schedule = new Schedule(fileName, new FakeSunsetProvider());
```

```
    // Assert
    Assert.NotEmpty(schedule);
}
```

11. Re-run the tests. There should now be 1 "Passed" test in addition to the failed ones.

12. Review the next test in the class:

```
[Fact]
public void ScheduleItems_OnCreation_AreInFuture()
{
    // Arrange / Act

    // Assert
    Assert.Fail("Test not implemented");
}
```

When a schedule is loaded from a file, the items are updated to their next future run time. In this test, you will make sure this is true: all `ScheduleItems` have an `EventTime` in the future. The `EventTime` is part of the `ScheduleItem.Info` property.

13. To check that items are in the future, you need the current time. This is part of the arrangement of the test.

```
// Arrange / Act
var schedule = new Schedule(fileName, new FakeSunsetProvider());
var currentTime = DateTimeOffset.Now;
```

The creation of the schedule is the same as the previous test. For the current time, use `DateTimeOffset.Now`. (This test, and other `Schedule` tests, can use the current time; tests for `ScheduleHelper` methods will need a fake time.)

14. For the assert section, loop through the items and check that they are in the future.

```
// Assert
foreach (var item in schedule)
{
    Assert.True(item.Info.EventTime > currentTime);
}
```

15. Review and re-run the tests.

```
[Fact]
public void ScheduleItems_OnCreation_AreInFuture()
{
    // Arrange / Act
    var schedule = new Schedule(fileName, new FakeSunsetProvider());
    var currentTime = DateTimeOffset.Now;

    // Assert
    foreach (var item in schedule)
    {
        Assert.True(item.Info.EventTime > currentTime);
    }
}
```

You should now have 2 passing tests.

16. Review the next test:

```
[Fact]
public void ScheduleItems_AfterRoll_AreInFuture()
```

For this test, the action is `AfterRoll` -- meaning calling `RollSchedule` on the `Schedule` object. To make sure that items are actually rolled forward, in the arrangement you should set the `EventTimes` to a time in the past.

17. Create the schedule (as in the last 2 tests), then subtract 30 days from all of the `EventTimes`.

```
// Arrange
var schedule = new Schedule(fileName, new FakeSunsetProvider());
var currentTime = DateTimeOffset.Now;
foreach (var item in schedule)
{
    item.Info.EventTime = item.Info.EventTime.AddDays(-30);
    Assert.True(item.Info.EventTime < currentTime,
        "Invalid Arrangement");
}
```

The first 2 lines of the Arrange section are the same. Inside the loop, after updating the time, there is a check to make sure it is in the past. This is a "sanity check" to make sure that the arrangement is correct. If this step fails the message "Invalid Arrangement" shows in the test runner so you that this is a setup failure rather than a test failure.

18. For the Act section, call the `RollSchedule` method.

```
// Act
schedule.RollSchedule();
```

19. The Assert section is the same as the previous test: to make sure all items are now in the future:

```
// Assert
foreach (var item in schedule)
{
    Assert.True(item.Info.EventTime > currentTime);
}
```

20. Review and run the tests.

```
[Fact]
public void ScheduleItems_AfterRoll_AreInFuture()
{
    // Arrange
    var schedule = new Schedule(fileName, new FakeSunsetProvider());
    var currentTime = DateTimeOffset.Now;
    foreach (var item in schedule)
    {
        item.Info.EventTime = item.Info.EventTime.AddDays(-30);
        Assert.True(item.Info.EventTime < currentTime,
            "Invalid Arrangement");
    }

    // Act
    schedule.RollSchedule();

    // Assert
    foreach (var item in schedule)
    {
        Assert.True(item.Info.EventTime > currentTime);
    }
}
```

The test runner should now show 3 passing tests.

21. Review the next test:

```
[Test]
public void OneTimeItemInPast_AfterRoll_IsRemoved()
```

This checks an item in the past that is marked as `Once`. See the Hints section for how to create a `ScheduleItem`.

22. Create the schedule (as in previous tests) and add a new item with a past time.

```
// Arrange
var schedule = new Schedule(fileName, new FakeSunsetProvider());

var newItem = new ScheduleItem(
    1,
    DeviceCommand.On,
    new ScheduleInfo()
    {
        EventTime = DateTimeOffset.Now.AddMinutes(-2),
        Type = ScheduleType.Once,
    },
    true,
    "");
schedule.Add(newItem);
```

23. Get the original count of items and the count after the new item is added. Then do a sanity check to make sure the item was added successfully.

```
// Arrange
var schedule = new Schedule(fileName, new FakeSunsetProvider());

var originalCount = schedule.Count;

var newItem = new ScheduleItem(...);
schedule.Add(newItem);

var newCount = schedule.Count;
Assert.True(newCount == (originalCount + 1),
    "Invalid Arrangement");
```

24. Roll the schedule forward.

```
// Act
schedule.RollSchedule();
```

25. Assert that the count is back to the original count (before adding the item).

```
// Assert
Assert.Equal(originalCount, schedule.Count);
```

26. Review and run the tests.

```
[Fact]
public void OneTimeItemInPast_AfterRoll_IsRemoved()
{
    // Arrange
    var schedule = new Schedule(fileName, new FakeSunsetProvider());

    var originalCount = schedule.Count;

    var newItem = new ScheduleItem(
        1,
        DeviceCommand.On,
        new ScheduleInfo()
        {
            EventTime = DateTimeOffset.Now.AddMinutes(-2),
            Type = ScheduleType.Once,
        },
        true,
        "");
    schedule.Add(newItem);

    var newCount = schedule.Count;
    Assert.True(newCount == (originalCount + 1),
        "Invalid Arrangement");

    // Act
    schedule.RollSchedule();

    // Assert
    Assert.Equal(originalCount, schedule.Count);
}
```

27. Look at the last test in the ScheduleTests class.

```
[Fact]
public void OneTimeItemInFuture_AfterRoll_IsStillThere()
```

This test is similar to the last test. The new item is in the future, and the assertion is that it is not removed by the schedule roll.

28. Copy/Paste the previous test. Change the `EventTime` to `+2` instead of `-2`, and change the assertion to check that the schedule count equals the **new count**.

```
[Fact]
public void OneTimeItemInFuture_AfterRoll_IsStillThere()
{
    // Arrange
    var schedule = new Schedule(fileName, new FakeSunsetProvider());

    var originalCount = schedule.Count;

    var newItem = new ScheduleItem(
        1,
        DeviceCommand.On,
        new ScheduleInfo()
        {
            EventTime = DateTimeOffset.Now.AddMinutes(+2),
            Type = ScheduleType.Once,
        },
        true,
        "");
    schedule.Add(newItem);

    var newCount = schedule.Count;
    Assert.True(newCount == (originalCount + 1),
        "Invalid Arrangement");

    // Act
    schedule.RollSchedule();

    // Assert
    Assert.Equal(newCount, schedule.Count);
}
```

29. Run all tests. There should now be 5 passing tests.

Step-by-Step - ScheduleHelper Tests

1. Open the `ScheduleHelper.Tests.cs` file in the `HouseControl.Library.Tests` project. This file does not have any tests stubbed out.

2. There is a comment to show which methods to test:

    - `RollForwardToNextDay`
      Rolls a schedule item to the next date after today.
    - `RollForwardToNextWeekdayDay`
      Rolls a schedule item to the next weekday (Monday - Friday) after today.

- `RollForwardToNextWeekendDay`
  Rolls a schedule item to the next weekend day (Saturday / Sunday) after today.

All methods should have tests for items in the past (which do roll forward) and items in the future (which do not roll forward).

3. For the first test, we will check a Monday date in the past and roll it forward regularly.

```
[Fact]
public void MondayItemInPast_OnRollDay_IsTomorrow()
{
}
```

4. In th Arrange section, create a `ScheduleHelper` instance. This needs a sunset provider. Use the `FakeSunsetProvider` created earlier.

```
// Arrange
ScheduleHelper helper = new(new FakeSunsetProvider());
```

5. Next create a `DateTimeOffset` that happens on a Monday (the date does not matter as long as it is in the past).

```
// Arrange
ScheduleHelper helper = new(new FakeSunsetProvider());
DateTimeOffset mondayInPast = new(2025, 06, 02, 15, 52, 00, );
```

6. To create a `DateTimeOffset`, you need to include a `TimeSpan` that represents the current time difference from UTC. Since this will be used throughout the tests, create a class-level field:

```
private TimeSpan timeZoneOffset = new(-2, 0, 0);
```

The exact value does not matter for our tests. In this case, it represents -2 hour from UTC.

7. Use the `timeZoneOffset` to create the monday variable.

```
DateTimeOffset mondayInPast = new(2025, 06, 02, 15, 52, 00,
    timeZoneOffset);
```

8. Create a variable to hold the expected value. This is tomorrow's date with the time portion from "monday".

```
DateTimeOffset expected =
    new(DateTimeOffset.Now.Date.AddDays(1) + mondayInPast.TimeOfDay,
        timeZoneOffset);
```

9. The `RollForwardToNextDay` method needs a `ScheduleInfo` object. Create a new instance. The only values you must set are `EventTime` and `TimeType`. The `TimeType` should be `Standard`.

```
ScheduleInfo info = new()
{
    EventTime = mondayInPast,
    TimeType = ScheduleTimeType.Standard,
};
```

10. Here is the completed `Arrange` section:

```
// Arrange
ScheduleHelper helper = new(new FakeSunsetProvider());
DateTimeOffset mondayInPast = new(2025, 06, 02, 15, 52, 00,
    timeZoneOffset);
DateTimeOffset expected =
    new(DateTimeOffset.Now.Date.AddDays(1) + mondayInPast.TimeOfDay,
        timeZoneOffset);
ScheduleInfo info = new()
{
    EventTime = mondayInPast,
    TimeType = ScheduleTimeType.Standard,
};
```

11. For the Act and Assert sections, call `RollForwardToNextDay` and compare the result with the `expected` value.

```
// Act
DateTimeOffset actual = helper.RollForwardToNextDay(info);

// Assert
Assert.Equal(expected, actual);
```

12. Run the test.

At this point, the test most likely will fail. And looking at the message, the expected and actual time zone offsets do not match (unless your computer is set for UTC-2).

The reason for the failure is that this test relies on the default time provider that is used by default. You can create your own `TimeProvider` and override certain methods.

But making our own `TimeProvider` is more complicated than I would like it to be. Fortunately, there is a NuGet package to bring in a fake time provider specifically for unit testing.

13. Add the `Microsoft.Extensions.TimeProvider.Testing` NuGet package. You can easily find this by searching for "FakeTimeProvider" in the NuGet Package Manager.

14. At the top of the test, create a new `DateTimeOffset` set to a Thursday after the Monday date that we currently have in the test.

```
DateTimeOffset thursday = new(2025, 06, 05, 15, 52, 00,
    timeZoneOffset);
```

Note: Things look a little complex right now. We'll centralize some of this to make the tests easier to read after we get one or two working.

15. After creating the `ScheduleHelper`, update the `HelperTimeProvider` property to use a fake time provider. The `FakeTimeProvider` comes from the NuGet package, and we can pass in the `DateTimeOffset` value to use as the "current" time.

```
ScheduleHelper helper = new(new FakeSunsetProvider());
helper.HelperTimeProvider = new FakeTimeProvider(thursday);
```

16. Update the `expected` value to use `thursday` rather than `Now`. Here is the updated Arrange section.

```
// Arrange
DateTimeOffset thursday = new(2025, 06, 05, 15, 52, 00,
    timeZoneOffset);
ScheduleHelper helper = new(new FakeSunsetProvider());
helper.HelperTimeProvider = new FakeTimeProvider(thursday);

DateTimeOffset mondayInPast = new(2025, 06, 02, 15, 52, 00,
    timeZoneOffset);
DateTimeOffset expected =
    new(thursday.Date.AddDays(1) + mondayInPast.TimeOfDay,
        timeZoneOffset);
ScheduleInfo info = new()
{
    EventTime = mondayInPast,
```

```
    TimeType = ScheduleTimeType.Standard,
};
```

17. Rerun the test. It should now pass.

We are going to use many of the same variables for testing different scenarios. Rather than duplicating things, let's extract a few items.

18. Create a new method (at the top of the test class) called `GetScheduleHelperForThursday`. This will create a new `ScheduleHelper` and set the `HelperTimeProvider` property.

```
private ScheduleHelper GetScheduleHelperForThursday()
{

}
```

Note: the "ForThursday" is in there because we will set the fake time provider to our thursday date, and we will need other fake dates for different tests.

19. Move the top 3 lines of the `Arrange` section from the test into this method, and return the `ScheduleHelper`.

```
private ScheduleHelper GetScheduleHelperForThursday()
{
    DateTimeOffset thursday = new(2025, 06, 05, 15, 52, 00,
        timeZoneOffset);
    ScheduleHelper helper = new(new FakeSunsetProvider());
    helper.HelperTimeProvider = new FakeTimeProvider(thursday);
    return helper;
}
```

20. Back in the test, update the Arrange section so that the `helper` variable comes from the new method.

```
// Arrange
ScheduleHelper helper = GetScheduleHelperForThursday();

DateTimeOffset mondayInPast = new(2025, 06, 02, 15, 52, 00,
    timeZoneOffset);
DateTimeOffset expected =
    new(thursday.Date.AddDays(1) + mondayInPast.TimeOfDay,
        timeZoneOffset);
ScheduleInfo info = new()
{
```

```
    EventTime = mondayInPast,
    TimeType = ScheduleTimeType.Standard,
};
```

But there's a problem

`thursday` is used to calculate the `expected` value in our test. One way to fix this is to add some class-level fields. As we'll see, these will be helpful in other tests.

21. Create a class level `DateTimeOffset` field to hold "today's" date (meaning, Thursday in this case).

```
private TimeSpan timeZoneOffset = new(-2, 0, 0);
private DateTimeOffset today;
```

22. Update the `GetScheduleHelperForThursday` method to use the new `today` field instead of a local `thursday` variable.

```
private ScheduleHelper GetScheduleHelperForThursday()
{
    //DateTimeOffset thursday = new(2025, 06, 05, 15, 52, 00,
    //    timeZoneOffset);
    today = new(2025, 06, 05, 15, 52, 00, timeZoneOffset);
    ScheduleHelper helper = new(new FakeSunsetProvider());
    //helper.HelperTimeProvider = new FakeTimeProvider(thursday);
    helper.HelperTimeProvider = new FakeTimeProvider(today);
    return helper;
}
```

23. Back in the unit test, change `expected` variable assignment to use `today` instead of `thursday`.

```
DateTimeOffset expected =
    new(today.Date.AddDays(1) + mondayInPast.TimeOfDay,
        timeZoneOffset);
```

24. Re-run the tests. Everything should pass now.

25. The next test is for a Monday in the future. When the `RollForwardToNextDay` method is called, the time should be unchanged.

```
[Fact]
public void MondayItemInFuture_OnRollDay_IsUnchanged()
```

```
{
}
```

26. The internals of this test are almost identical to the previous. Here are the differences.
    - The `mondayInPast` should be `mondayInFuture` and the date should be a date after the `thursday` value.
    - The `expected` value is `mondayInFuture`.

```
[Fact]
public void MondayItemInFuture_OnRollDay_IsUnchanged()
{
    // Arrange
    ScheduleHelper helper = GetScheduleHelperForThursday();

    DateTimeOffset mondayInFuture = new(2025, 06, 09, 15, 52, 00,
        timeZoneOffset);
    DateTimeOffset expected = mondayInFuture;
    ScheduleInfo info = new()
    {
        EventTime = mondayInFuture,
        TimeType = ScheduleTimeType.Standard,
    };

    // Act
    DateTimeOffset actual = helper.RollForwardToNextDay(info);

    // Assert
    Assert.Equal(expected, actual);
}
```

27. Re-run the tests and make sure this passes.

A Bit More Refactoring

At this point it is going to be hard to keep track of `mondayInPast` and `mondayInFuture` in relation to the other dates. To fix this, let's create a few more class-level fields that we can set in the `GetScheduleHelperForThursday` method. Then we will have all of the calculations in one spot.

28. Add fields for `mondayInPast` and `mondayInFuture` to the top of the class.

```
private DateTimeOffset today;
private DateTimeOffset mondayInPast;
private DateTimeOffset mondayInFuture;
```

29. Set the new fields in the `GetScheduleHelperForThursday` method. Instead of hard-coding the relative dates, we'll use date math to set them relative to `today`.

```
private ScheduleHelper GetScheduleHelperForThursday()
{
    today = new(2025, 06, 05, 15, 52, 00, timeZoneOffset);
    mondayInPast = today.AddDays(-3);
    mondayInFuture = today.AddDays(4);

    ScheduleHelper helper = new(new FakeSunsetProvider());
    helper.HelperTimeProvider = new FakeTimeProvider(today);
    return helper;
}
```

30. In the unit tests, remove the lines that create/set `mondayInPast` and `mondayInFuture`.

```
// Arrange
ScheduleHelper helper = GetScheduleHelperForThursday();

//DateTimeOffset mondayInPast = new(2025, 06, 02, 15, 52, 00,
//    timeZoneOffset);
DateTimeOffset expected =
    new(today.Date.AddDays(1) + mondayInPast.TimeOfDay,
        timeZoneOffset);
```

```
// Arrange
ScheduleHelper helper = GetScheduleHelperForThursday();

//DateTimeOffset mondayInFuture = new(2025, 06, 09, 15, 52, 00,
//    timeZoneOffset);
DateTimeOffset expected = mondayInFuture;
```

31. Re-run the tests to make sure everything still passes.

32. The next set of tests are to check the weekday roll. This means that if a schedule item is set for "Weekdays only", we want to check that if today is Friday, it rolls forward to Monday.

33. Before creating a new test, let's create a new method for `GetScheduleHelperForFriday`.

```
private ScheduleHelper GetScheduleHelperForFriday()
{
```

```
}
```

34. We can use the same internals as `GetScheduleHelperForThursday`. The changes include adding 1 to `today` as well as incrementing the math for the `monday` values.

```
private ScheduleHelper GetScheduleHelperForFriday()
{
    today = new(2025, 06, 06, 15, 52, 00, timeZoneOffset);
    mondayInPast = today.AddDays(-4);
    mondayInFuture = today.AddDays(3);

    ScheduleHelper helper = new(new FakeSunsetProvider());
    helper.HelperTimeProvider = new FakeTimeProvider(today);
    return helper;
}
```

34. Create a new test to check the weekday roll.

```
[Fact]
public void MondayItemInPastAndTodayFriday_OnRollWeekdayDay_IsNextMonday()
{

}
```

This will let us use our new method as well as both of our `monday` fields.

35. Copy the body of the prior "MondayItemIsInPast_" method. Here are the changes:

- Change `GetScheduleHelperForThursday` to `GetScheduleHelperForFriday`.
- Change `expected` from `today.Date.AddDays(1)` to `today.Date.AddDays(3)` (Friday + 3 days).
- Change the `Act` section from `RollForwardToNextDay` to `RollForwardToNextWeekdayDay`.

```
// Arrange
ScheduleHelper helper = GetScheduleHelperForFriday();
DateTimeOffset expected =
    new(today.Date.AddDays(3) + mondayInPast.TimeOfDay,
        timeZoneOffset);
ScheduleInfo info = new()
{
    EventTime = mondayInPast,
    TimeType = ScheduleTimeType.Standard,
};
```

```
// Act
DateTimeOffset actual = helper.RollForwardToNextWeekdayDay(info);

// Assert
Assert.Equal(expected, actual);
```

36. The next test is the same as the previous, but with Monday in the future. The `expected` result is that the value is unchanged.

```
[Fact]
public void MondayItemInFutureAndTodayFriday_OnRollWeekdayDay_IsUnchanged()
{
    // Arrange
    ScheduleHelper helper = GetScheduleHelperForFriday();
    DateTimeOffset expected = mondayInFuture;
    ScheduleInfo info = new()
    {
        EventTime = mondayInFuture,
        TimeType = ScheduleTimeType.Standard,
    };

    // Act
    DateTimeOffset actual = helper.RollForwardToNextWeekdayDay(info);

    // Assert
    Assert.Equal(expected, actual);
}
```

37. Create the same pair of tests for `RollForwardToNextWeekendDay`. Use a Friday for the current date, Sunday for the test date (one in the "past", one in the "future").

Start by adding `sundayInPast` and `sundayInFuture` fields.

```
private DateTimeOffset today;
private DateTimeOffset mondayInPast;
private DateTimeOffset mondayInFuture;
private DateTimeOffset sundayInPast;
private DateTimeOffset sundayInFuture;
```

38. Add the calculations to the `GetScheduleHelper` methods.

```
private ScheduleHelper GetScheduleHelperForThursday()
{
    today = new(2025, 06, 05, 15, 52, 00, timeZoneOffset);
    mondayInPast = today.AddDays(-3);
    mondayInFuture = today.AddDays(4);
    sundayInPast = today.AddDays(-4);
    sundayInFuture = today.AddDays(3);

    ScheduleHelper helper = new(new FakeSunsetProvider());
    helper.HelperTimeProvider = new FakeTimeProvider(today);
    return helper;
}

private ScheduleHelper GetScheduleHelperForFriday()
{
    today = new(2025, 06, 06, 15, 52, 00, timeZoneOffset);
    mondayInPast = today.AddDays(-4);
    mondayInFuture = today.AddDays(3);
    sundayInPast = today.AddDays(-5);
    sundayInFuture = today.AddDays(2);

    ScheduleHelper helper = new(new FakeSunsetProvider());
    helper.HelperTimeProvider = new FakeTimeProvider(today);
    return helper;
}
```

39. Create a test with the following setup: Today is Friday, Sunday item is in the past. Action is RollWeekendDay. Expected is that the new date is Saturday in the future.

```
[Fact]
public void SundayItemInPastAndTodayFriday_OnRollWeekendDay_IsSaturday()
{
    // Arrange
    ScheduleHelper helper = GetScheduleHelperForFriday();
    DateTimeOffset expected =
        new(today.Date.AddDays(1) + sundayInPast.TimeOfDay,
            timeZoneOffset);
    ScheduleInfo info = new()
    {
        EventTime = sundayInPast,
        TimeType = ScheduleTimeType.Standard,
    };

    // Act
    DateTimeOffset actual = helper.RollForwardToNextWeekendDay(info);

    // Assert
```

```
    Assert.Equal(expected, actual);
}
```

40. One last test for now: Today is Friday, Sunday item is in the future. Action is RollWeekendDay. Expected is that the date is unchanged.

```
[Fact]
public void SundayItemInFutureAndTodayFriday_OnRollWeekendDay_IsUnchanged()
{
    // Arrange
    ScheduleHelper helper = GetScheduleHelperForFriday();
    DateTimeOffset expected = sundayInFuture;
    ScheduleInfo info = new()
    {
        EventTime = sundayInFuture,
        TimeType = ScheduleTimeType.Standard,
    };

    // Act
    DateTimeOffset actual = helper.RollForwardToNextWeekendDay(info);

    // Assert
    Assert.Equal(expected, actual);
}
```

41. With the initial scenarios laid out, additional tests get easier to add. At this point, we could think about some further refactoring or even parameterizing tests, but I'll leave that exploration up to you.

> For my own tests, my goal is to have tests as readable as possible. This way if something goes wrong, then it will (hopefully) be easy to troubleshoot. The difficult part is finding the balance between tests that are easy to write (which may mean they may have a lot of "magical" setup that runs) and easy to debug (where most of the code is inside the test itself).

Step-by-Step - "Now" Helper Methods

As a bonus, you can do a bit of refactoring and testing of "Now" helper methods. The code currently has 3 helper methods:

- Now()
- Tomorrow()
- IsInFuture(DateTimeOffset)

We will create tests for these items as well as add a few more helpers to make code more readable:

- Today()
- IsInPast(DateTimeOffset)

- DurationFromNow(DateTimeOffset)

For this section, all methods go in the `ScheduleHelper` class. All tests go in the `ScheduleHelper.Tests` class.

1. Create a test for `Now()`.

```
[Fact]
public void Now_ReturnsConfiguredTime()
{
    // Arrange
    ScheduleHelper helper = GetScheduleHelperForThursday();
    DateTimeOffset expected = today;

    // Act
    var actual = helper.Now();

    // Assert
    Assert.Equal(expected, actual);
}
```

By now, you should be able to see how this test works. First, gets the schedule helper for Thursday, which sets the current time and also the `today` field. Then the test calls the `Now()` method and makes sure the result is `today`.

2. Create a test for `Tomorrow`. This will be similar to the test for `Now`.

```
[Fact]
public void Tomorrow_ReturnsConfiguredDatePlusOne()
{
    // Arrange
    ScheduleHelper helper = GetScheduleHelperForThursday();
    DateTimeOffset expected = new DateTimeOffset(today.Date.AddDays(1),
today.Offset);

    // Act
    var actual = helper.Tomorrow();

    // Assert
    Assert.Equal(expected, actual);
}
```

The expected value is just the date portion + 1 day, with the current time zone offset.

3. Create a new method for `Today` which is just the Date portion from `Now` (along with the current time zone offset).

```
public DateTimeOffset Today()
{
    return new DateTimeOffset(
        Now().Date, Now().Offset);
}
```

4. Create a unit test for `Today` (similar to the test for `Now`).

```
[Fact]
public void Today_ReturnsConfiguredDate()
{
    // Arrange
    ScheduleHelper helper = GetScheduleHelperForThursday();
    DateTimeOffset expected = new DateTimeOffset(today.Date, today.Offset);

    // Act
    var actual = helper.Today();

    // Assert
    Assert.Equal(expected, actual);
}
```

5. Refactor calls to `ScheduleHelper.Now().Date`. In the `Schedule` class, `Now().Date` is used in the `LoadSchedule` method.

```
public void LoadSchedule()
{
    this.Clear();
    this.AddRange(Loader.LoadScheduleItems(filename));

    // update loaded schedule dates to today
    //DateTimeOffset today = scheduleHelper.Now().Date;
    DateTimeOffset today = scheduleHelper.Today();
    foreach (var item in this)
    {
        item.Info.EventTime = today + item.Info.EventTime.TimeOfDay;
    }
    RollSchedule();
}
```

6. Run the unit tests.

This tells us 2 things. First, our new method works. Second, that our refactoring did not break any existing functionality. To me, the second factor is really important since it lets me move forward without worrying about

existing code that has tests.

7. Create unit tests for the `IsInFuture` method using both cases: parameter is in the past, parameter is in the future.

```
[Fact]
public void IsInFuture_WithPastDate_ReturnsFalse()
{
    // Arrange
    ScheduleHelper helper = GetScheduleHelperForThursday();

    // Act
    var actual = helper.IsInFuture(mondayInPast);

    // Assert
    Assert.False(actual);
}

[Fact]
public void IsInFuture_WithFutureDate_ReturnsTrue()
{
    // Arrange
    ScheduleHelper helper = GetScheduleHelperForThursday();

    // Act
    var actual = helper.IsInFuture(mondayInFuture);

    // Assert
    Assert.True(actual);
}
```

8. Run the tests.

9. Create a `IsInPast` method that compares a parameter date to `Now()`.

```
public bool IsInPast(DateTimeOffset checkTime)
{
    return checkTime < Now();
}
```

10. Create unit tests for both cases (parameter is in the past, parameter is in the future). These are similar to the `IsInFuture` tests.

```
[Fact]
public void IsInPast_WithPastDate_ReturnsTrue()
```

```
{
    // Arrange
    ScheduleHelper helper = GetScheduleHelperForThursday();

    // Act
    var actual = helper.IsInPast(mondayInPast);

    // Assert
    Assert.True(actual);
}

[Fact]
public void IsInPast_WithFutureDate_ReturnsFalse()
{
    // Arrange
    ScheduleHelper helper = GetScheduleHelperForThursday();

    // Act
    var actual = helper.IsInPast(mondayInFuture);

    // Assert
    Assert.False(actual);
}
```

11. Refactor the `Schedule` class to use `IsInPast`. This is in the `RollSchedule` method.

```
//while (currentItem.Info.EventTime < scheduleHelper.Now())
while(scheduleHelper.IsInPast(currentItem.Info.EventTime))
```

This may seem like a trivial refactoring, but it shows intent. Rather than a comparison, the code asks whether an item is in the past.

12. Rerun the tests. This confirms that we did not change existing functionality.

13. Tie `IsInPast` and `IsInFuture` together, meaning, have one method call the other. This will ensure that the code is always synchronized. Here is an update to `IsInFuture`:

```
public bool IsInFuture(DateTimeOffset checkTime)
{
    return !IsInPast(checkTime);
}
```

14. Rerun the tests.

15. Create a method for `DurationFromNow` that returns a `TimeSpan`. For this, you can subtract the dates which results in a `TimeSpan`. `TimeSpan` has a `Duration` method that will give the absolute value back (meaning, you do not need to worry about negative values).

```
public TimeSpan DurationFromNow(DateTimeOffset checkTime)
{
    return (checkTime - Now()).Duration();
}
```

16. Create a unit test for `DurationFromNow`. This can set the current time, and then set a time to current time + 10 minutes. Duration should be that same 10 minute `TimeSpan`.

```
[Fact]
public void DurationFromNow_Time10MinutesInPast_Returns10Minutes()
{
    // Arrange
    ScheduleHelper helper = GetScheduleHelperForThursday();
    DateTimeOffset testTime = today.AddMinutes(-10);
    TimeSpan expected = new(0, 10, 0);

    // Act
    var actual = helper.DurationFromNow(testTime);

    // Assert
    Assert.Equal(expected, actual);
}
```

17. Refactor the Schedule class to use the new method. This is needed in the `GetCurrentScheduleItems` method. This method looks for items that are within 30 seconds of the current time so that it can execute the commands.

```
// Updated Method
public List<ScheduleItem> GetCurrentScheduleItems()
{
    return this.Where(si => si.IsEnabled &&
        ScheduleHelper.DurationFromNow(si.Info.EventTime) <
TimeSpan.FromSeconds(30))
        .ToList();
}
```

18. Rerun the tests.

Wrap Up

This this lab, you saw some of the quirks and issues with testing `DateTime.Now`. Ideally, it is best to avoid statics in tests. Fortunately, we have the `TimeProvider` type to help us out with that.

You saw how to create an abstraction around `DateTime.Now` (or `DateTimeOffset.Now` in this specific case). By using property injection, you get a good default value (the System time), but there is also an injection point to change the value in unit tests.

For the various dependencies, you saw how you can create a fake class that implements an interface that is needed during testing. By using a fake object, you get full control over how an object responds in a test. These are great for dependencies that are incidental to the methods you are testing.

Hopefully you are more comfortable with unit testing and see the importance of the abstractions that you created in Lab 01.

In the next lab, you will use a dependency injection container and learn some techniques for managing dependencies.

END - Lab 02 - Adding Unit Tests