Test a Bank Account Web App.

Step 8. Validate User Input

Run your project to test your code. Think of various edge cases for the values and see what happens when you input the following cases:

- Balance as a character or string ‘a’
- Amount as a character or string ‘z’
- Negative balance as input
- Negative amount as input
- Both balance and amount are negative
- Resulting negative balance (amount would overdraw the account)

Add Range and possibly Required statements in your Model.

How can you handle bad input in the Controller?

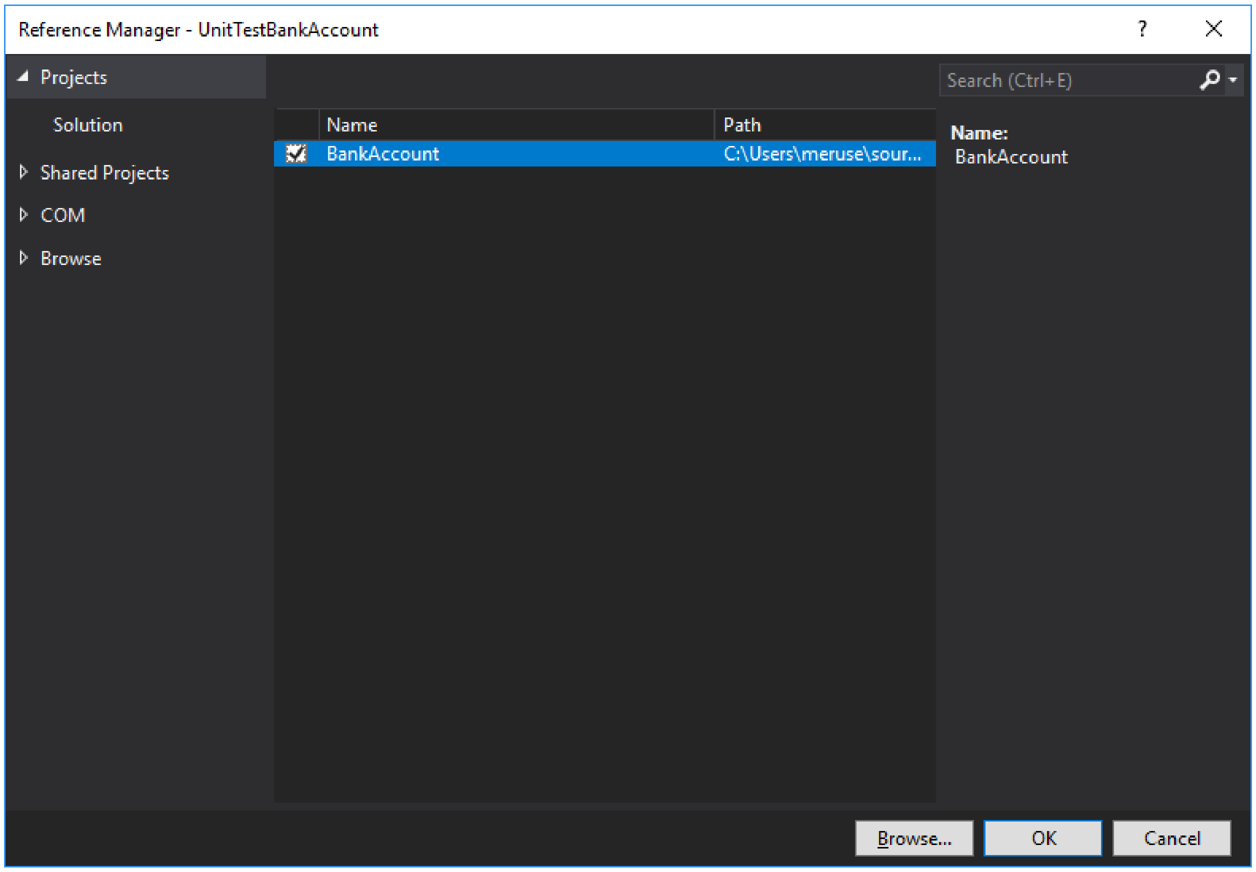
Add ways to parse the Balance and Amount.

How about testing the class itself?

Write Unit Tests for the above.

To the solution, Add New Project, xUnitTest Project, name it UnitTestBankAccount.

Under Dependencies, Add Reference, Check BankAccount and click OK.


Reference Manager - UnitTestBankAccount

Name	Path
✓ BankAccount	C:\Users\meruse\sour...

Name:
BankAccount

Add the following Unit Tests, notice the final one is for Withdraw().

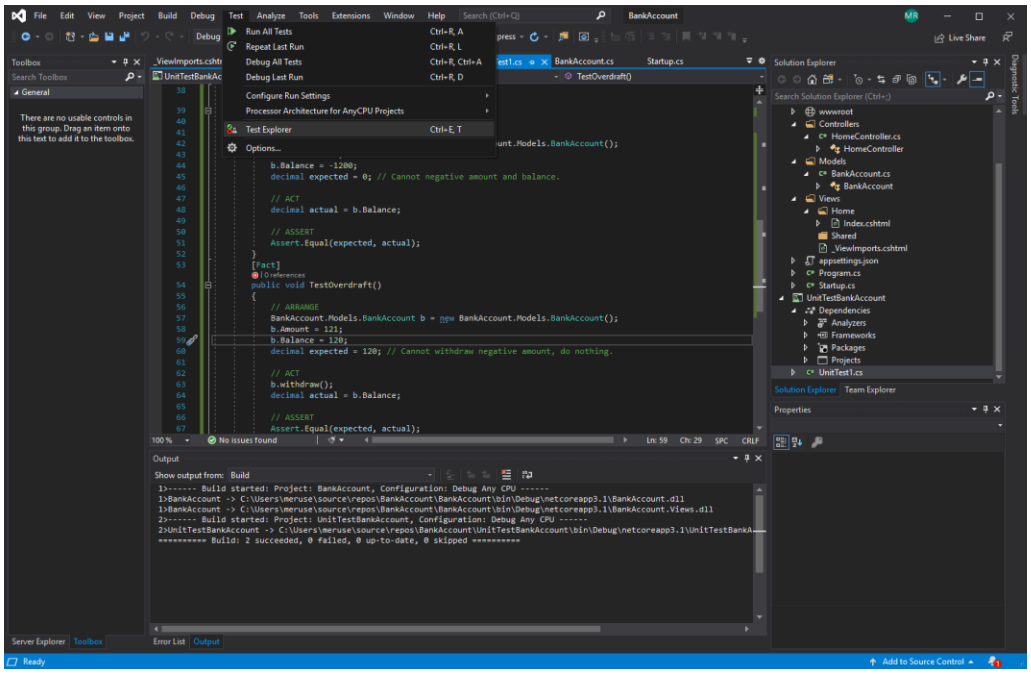
We did not test Deposit() manually.

Run the tests, then change the code so the so the tests pass. DO NOT CHANGE THE TESTS. They represent the expected behavior. The ideal would be an error message throw, but you can learn that later.

```
public class UnitTest1
{
    [Fact]
    public void TestNegativeAmount()
    {
        // ARRANGE
        BankAccount.Models.BankAccount b = new BankAccount.Models.BankAccount();
        decimal negativeValue = -1;

        // ACT
        // ASSERT
        Assert.Throws<ArgumentOutOfRangeException>(() => b.Amount = negativeValue);
    }
    [Fact]
    public void TestNegativeBalance()
    {
        // ARRANGE
        BankAccount.Models.BankAccount b = new BankAccount.Models.BankAccount();
        decimal negativeValue = -1;

        // ACT
        // ASSERT
        Assert.Throws<ArgumentOutOfRangeException>(() => b.Balance = negativeValue);
    }
    [Fact]
    public void TestNegativeBalanceAndAmount()
    {
        // ARRANGE
        BankAccount.Models.BankAccount b = new BankAccount.Models.BankAccount();
        decimal negativeValue = -12;

        // ACT
        // ASSERT
        Assert.Throws<ArgumentOutOfRangeException>(() => b.Balance = negativeValue);
        Assert.Throws<ArgumentOutOfRangeException>(() => b.Amount = negativeValue);

    }
    [Fact]
    public void TestOverdraft()
    {
        // ARRANGE
        BankAccount.Models.BankAccount b = new BankAccount.Models.BankAccount();
        b.Amount = 103;
        b.Balance = 102;

        // ACT
        // ASSERT
```

```
            Assert.Throws<ArgumentOutOfRangeException>(() => b.Withdraw());
        }
    }
```

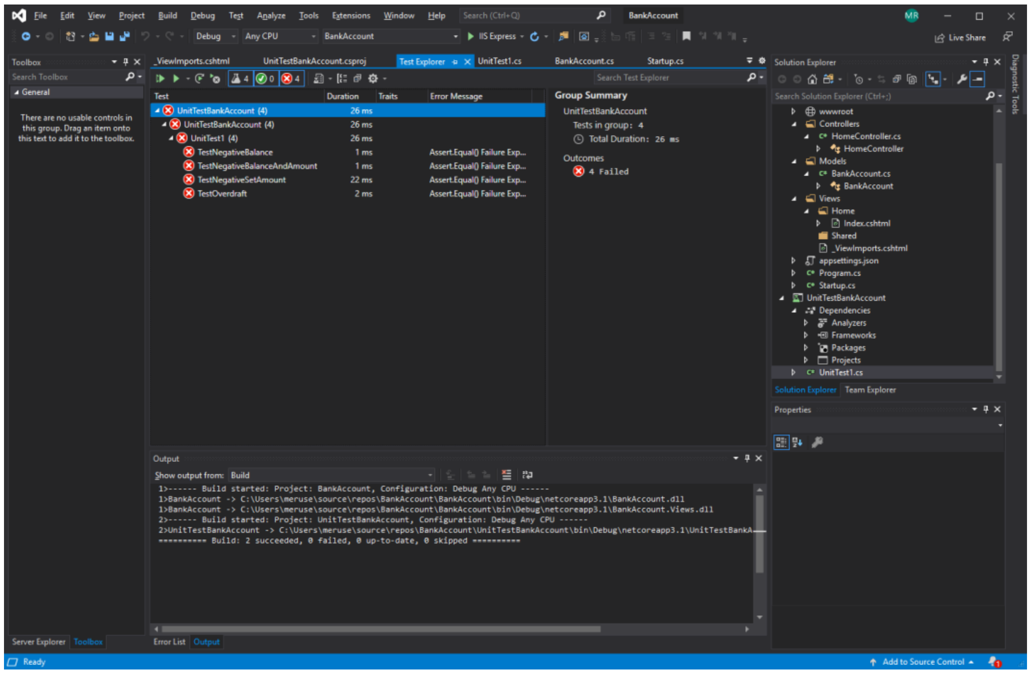
Open the Test Explorer:



```
unt.Models.BankAccount();
            b.Balance = -1200;
            decimal expected = 0; // Cannot negative amount and balance.

            // ACT
            decimal actual = b.Balance;

            // ASSERT
            Assert.Equal(expected, actual);
        }
        [Fact]
        public void TestOverdraft()
        {
            // ARRANGE
            BankAccount.Models.BankAccount b = new BankAccount.Models.BankAccount();
            b.Amount = 121;
            b.Balance = 120;
            decimal expected = 120; // Cannot withdraw negative amount, do nothing.

            // ACT
            b.withdraw();
            decimal actual = b.Balance;

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

```
1>------ Build started: Project: BankAccount, Configuration: Debug Any CPU ------
1>BankAccount -> C:\Users\meruse\source\repos\BankAccount\BankAccount\bin\Debug\netcoreapp3.1\BankAccount.dll
1>BankAccount -> C:\Users\meruse\source\repos\BankAccount\BankAccount\bin\Debug\netcoreapp3.1\BankAccount.Views.dll
2>------ Build started: Project: UnitTestBankAccount, Configuration: Debug Any CPU ------
2>UnitTestBankAccount -> C:\Users\meruse\source\repos\BankAccount\UnitTestBankAccount\bin\Debug\netcoreapp3.1\UnitTestBankA
========== Build: 2 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Expand to see all tests failing:



Test	Duration	Traits	Error Message
UnitTestBankAccount (4)	26 ms		
UnitTestBankAccount (4)	26 ms		
UnitTest1 (4)	26 ms		
TestNegativeBalance	1 ms		Assert.Equal() Failure Exp...
TestNegativeBalanceAndAmount	1 ms		Assert.Equal() Failure Exp...
TestNegativeSetAmount	22 ms		Assert.Equal() Failure Exp...
TestOverdraft	2 ms		Assert.Equal() Failure Exp...

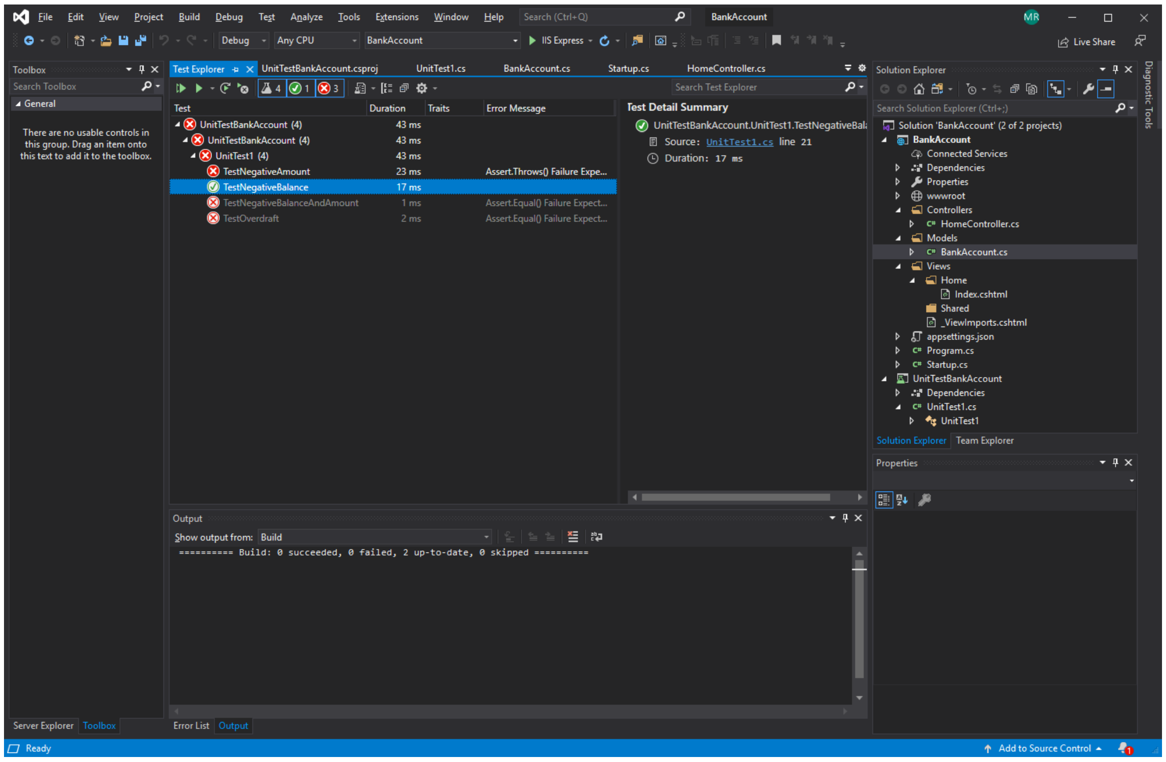
Group Summary
UnitTestBankAccount
Tests in group: 4
Total Duration: 26 ms
Outcomes
4 Failed

```
1>------ Build started: Project: BankAccount, Configuration: Debug Any CPU ------
1>BankAccount -> C:\Users\meruse\source\repos\BankAccount\BankAccount\bin\Debug\netcoreapp3.1\BankAccount.dll
1>BankAccount -> C:\Users\meruse\source\repos\BankAccount\BankAccount\bin\Debug\netcoreapp3.1\BankAccount.Views.dll
2>------ Build started: Project: UnitTestBankAccount, Configuration: Debug Any CPU ------
2>UnitTestBankAccount -> C:\Users\meruse\source\repos\BankAccount\UnitTestBankAccount\bin\Debug\netcoreapp3.1\UnitTestBankA
========== Build: 2 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Start with TestNegativeBalance, in BankAccount.cs, change the Property:

```
        public decimal Balance
        {
            get { return _balance; }
            set
            {
                if (value < 0)
                {
                    throw new ArgumentOutOfRangeException($"{nameof(value)} must be
positive.");
                }
                _balance = value;
            }
        }
```

Run all tests again, see now that test is passing.



```
Test                                  Duration  Traits  Error Message
UnitTestBankAccount (4)               43 ms
  UnitTestBankAccount (4)             43 ms
    UnitTest1 (4)                     43 ms
      TestNegativeAmount              23 ms             Assert.Throws() Failure Expe...
      TestNegativeBalance             17 ms
      TestNegativeBalanceAndAmount    1 ms              Assert.Equal() Failure Expect...
      TestOverdraft                   2 ms              Assert.Equal() Failure Expect...
```

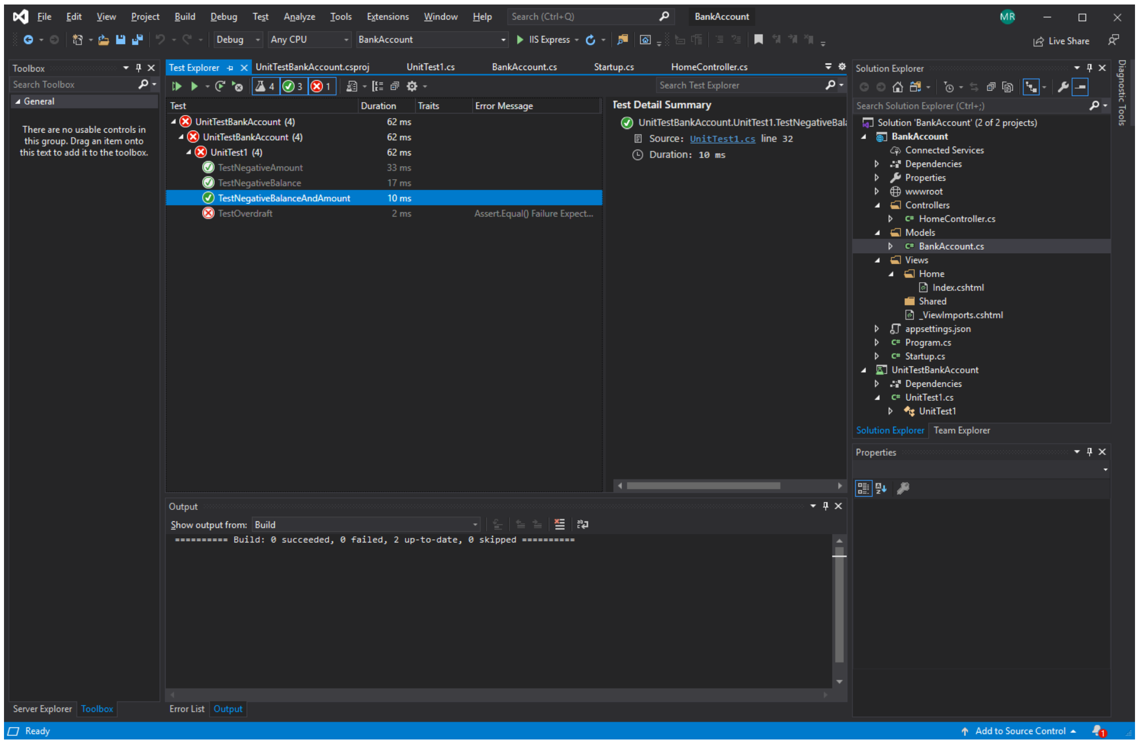
```
Test Detail Summary
UnitTestBankAccount.UnitTest1.TestNegativeBal
Source: UnitTest1.cs line 21
Duration: 17 ms
```

```
Solution 'BankAccount' (2 of 2 projects)
  BankAccount
    Connected Services
    Dependencies
    Properties
    wwwroot
    Controllers
      HomeController.cs
    Models
      BankAccount.cs
    Views
      Home
        Index.cshtml
      Shared
      _ViewImports.cshtml
    appsettings.json
    Program.cs
    Startup.cs
  UnitTestBankAccount
    Dependencies
    UnitTest1.cs
      UnitTest1
```

```
========== Build: 0 succeeded, 0 failed, 2 up-to-date, 0 skipped ==========
```

Next, make TestNegativeAmount pass by changing its Property similar to Balance above.

Are they any redundant test (meaning any test that test the same thing)? Remove one of the redundant tests. Do not comment it out, fully remove it.



Test	Duration	Traits	Error Message
UnitTestBankAccount (4)	62 ms		
UnitTestBankAccount (4)	62 ms		
UnitTest1 (4)	62 ms		
TestNegativeAmount	33 ms		
TestNegativeBalance	17 ms		
TestNegativeBalanceAndAmount	10 ms		
TestOverdraft	2 ms		Assert.Equal() Failure Expect...

Test Detail Summary

UnitTestBankAccount.UnitTest1.TestNegativeBal

Source: UnitTest1.cs line 32

Duration: 10 ms

Output

Show output from: Build

```
========== Build: 0 succeeded, 0 failed, 2 up-to-date, 0 skipped ==========
```

Next, add the logic to Withdraw() for TestOverdraft, but have the message say “Amount of $$ will overdraft your account.” Make sure $$ is the value.

All three tests should be passing!

Take a screen shot of your passing tests, upload your BankAccount.cs file, UnitTest1.cs file, and your screen shot.