Using Abstraction to Test the User Interface

Benjamin Day
TRAINER | COACH | DEVELOPER

@benday www.benday.com

Overview

Unit testing an ASP.NET MVC user interface

Model-View-Controller

Dependency Injection

Mocks, Stubs, and Fakes

First up:
Thinking about UI testing

Testing a real, running
user interface is hard.

UI Automation Tests

Selenium, Coded UI, ...

Why is it hard?

You have to deploy your app.

Deploy your app = integration test

Unit tests →
Test in isolation
with no dependencies

UI Automation Tests

Selenium, Coded UI

Simulating a browser using your app

Can be valuable...

...but it's not a unit test

Layer of indirection

Not testing at the API level

Unit tests will test the
UI at the API level.

No simulated browsers.

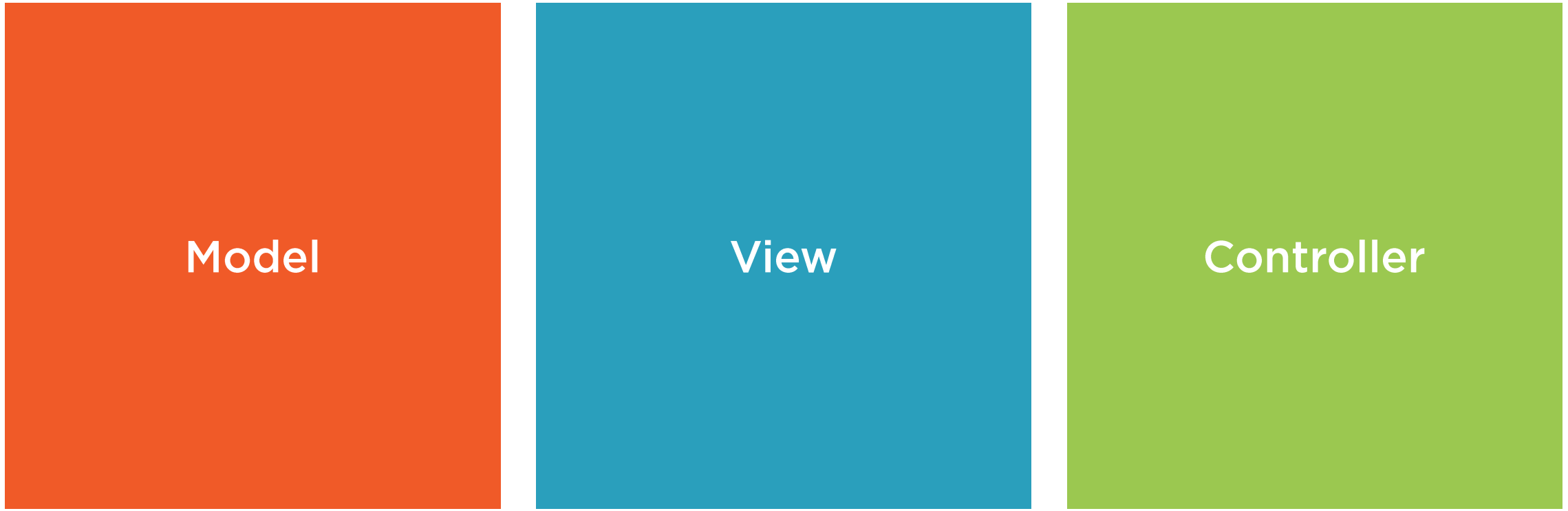
Design for Testability?

You'll be unit testing an abstraction of the user interface.

Model-View-Controller (MVC)

ASP.NET MVC

Three Pieces



Model

Data that gets displayed on the UI

State of the UI

- Is a control visible?
- What values are displayed?

View

The visible part of the UI

Turns the model into HTML

Controller

Glue between the View and Model

Has actions

Performs actions requested by the user or app

Reads data from the View via the Model

Talks to the rest of the application

Design for Testability

Design for Doing Yourself a Favor

Model Tips

Think "ViewModel"

- Popular in WPF
- Data + State of the UI
- Bind to the ViewModel

Model is inextricably linked to a View

View is usually inextricably linked to a Model

Your Entity Framework Entities are NOT your Models

EF Entities != Models

Model / ViewModel → UI

EF Entities → Database

Don't weld your UI to your database

Probably violates
Single Responsibility Principle

Changes in one implies changes in the other

Next up:
ASP.NET Calculator Demos

Web-based Calculator with ASP.NET MVC Core

Add, Subtract, Multiply, Divide

Getting Started

Focus your unit tests on the Controller

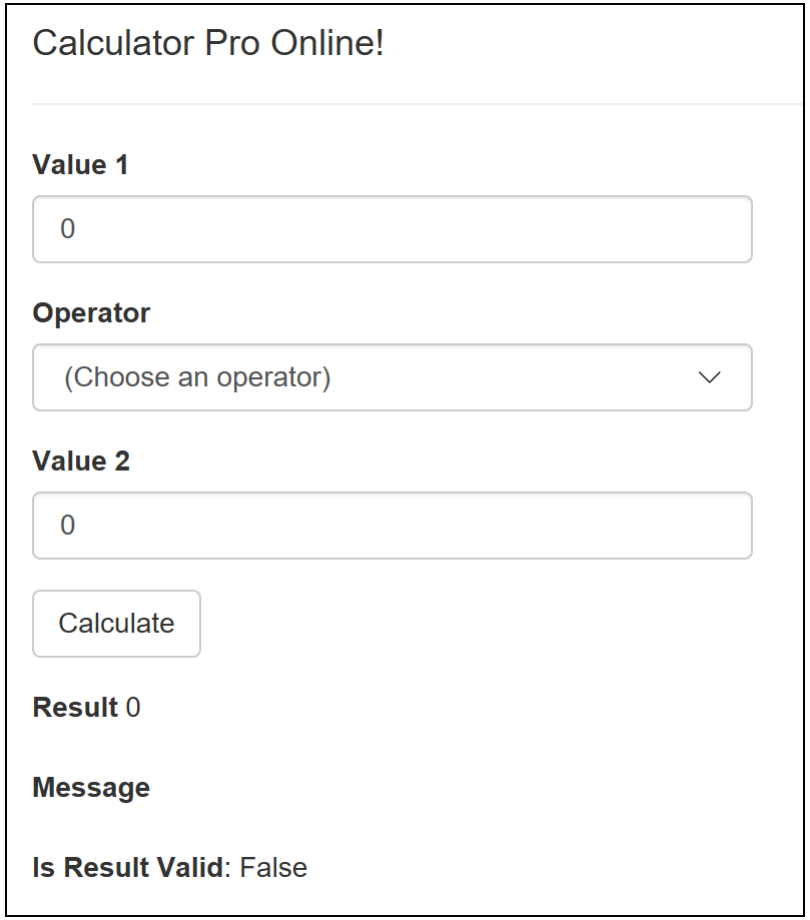
Controller:

- CalculatorController

Unit Test Class:

- CalculatorControllerFixture

The User Interface



What Are Your Basic Test Cases?

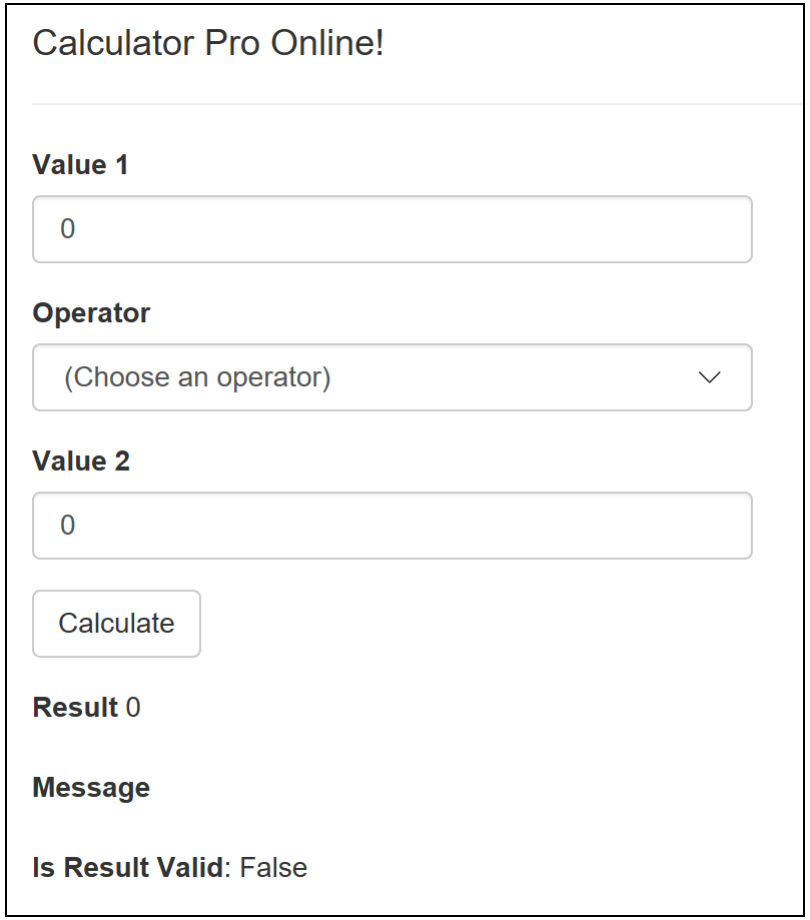
Draw the blank calculator page

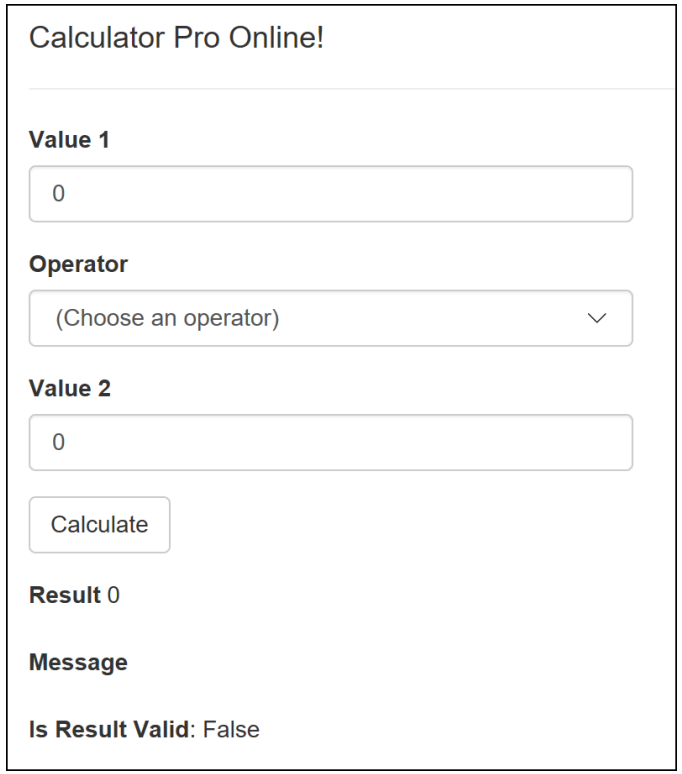
Do a calculation

- Add, Subtract, Multiply, Divide

Divide by zero

What Else Should You Test?





Initial Calculator Screen

Value 1 = 0

List of Operators is populated

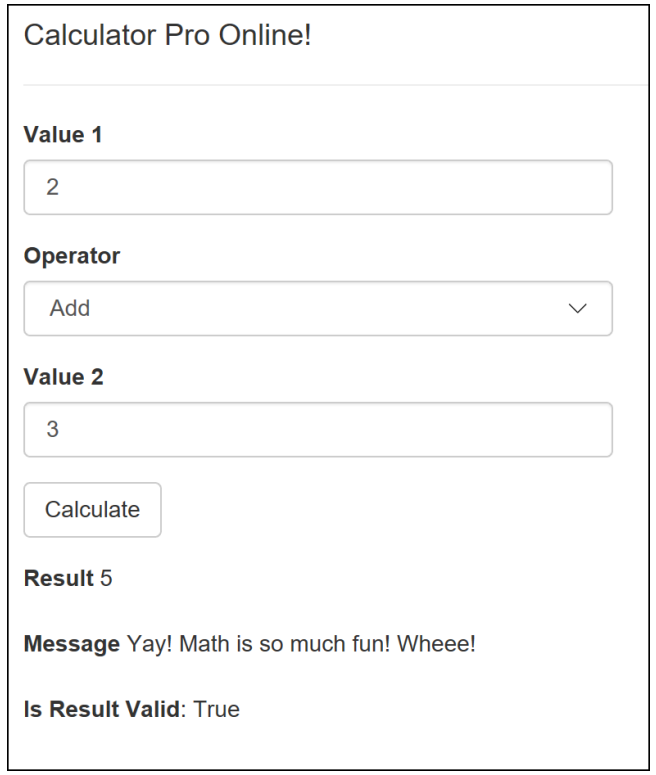
Default operator is “(Choose an operator)”

Value 2 = 0

Result = 0

Message is blank

Is Result Valid is False



Run a Calculation – Add

Set Value 1 = 2

Choose an operator

- Add

Set Value 2 = 3

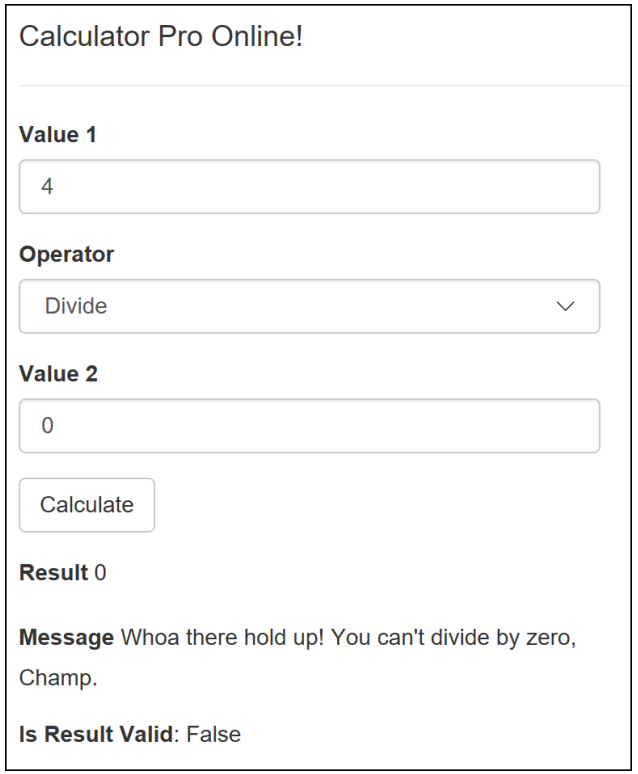
Run the calculation

Result = 5

Message is
“Yay! Math is so much fun! Wheee!”

Is Result Valid is True

Value 1, Value 2, Operator are the expected values



Divide By Zero

Set Value 1 = 4

Choose Divide

Set Value 2 = 0

Run the calculation

Result = 0

Message is
“Whoa there hold up! You can’t divide by zero, Champ.”

Is Result Valid is False

Value 1, Value 2, Operator are the expected values

Next up:
Demos

Demo

Web-based Calculator

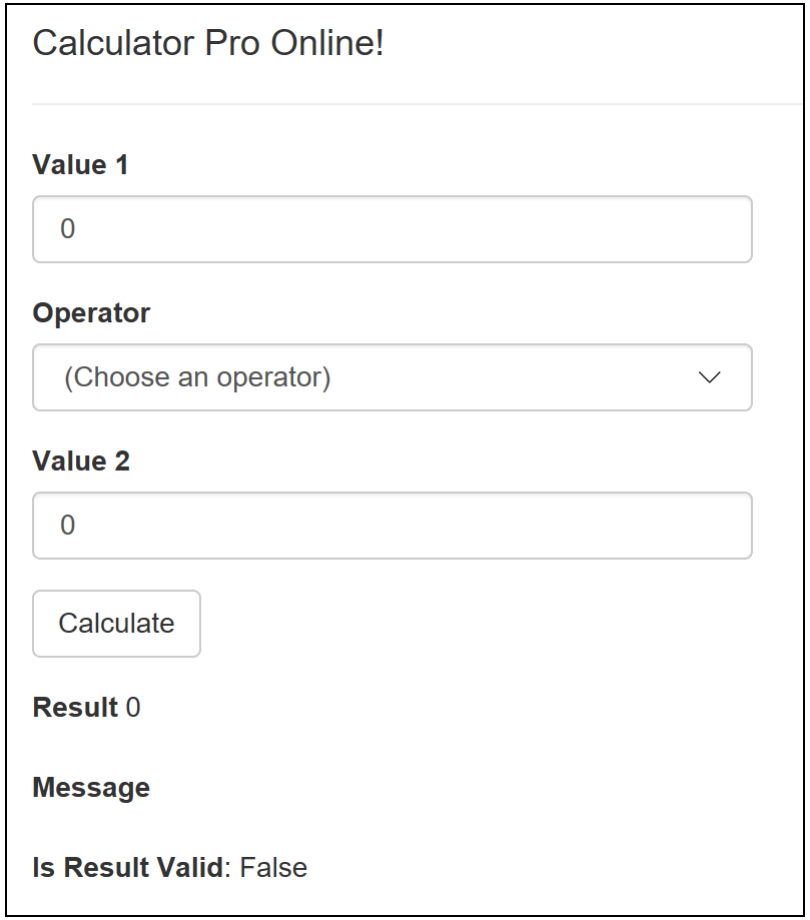
Multi-part demo

Unit test for initial load of the page

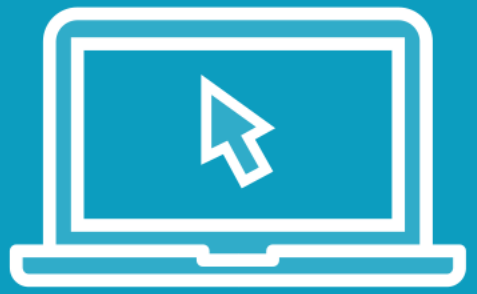
Part 1:

- Setup the UI unit test code
- Create the Controller
- Focus on the *.cshtml view

The User Interface



Demo



Web-based Calculator

Multi-part demo

Unit test for initial load of the page

Part 2:

- Set up the Model
- Start implementing unit tests
- Basic implementation of the Controller
- Accessing Model from a unit test

Demo

Web-based Calculator

Multi-part demo

Unit test for initial load of the page

Part 3:

- Implement unit tests to check content of the Operators drop down list
- Continue implementing the Controller

Demo

Web-based Calculator

Add two numbers

Demo

Web-based Calculator

Unit testing the error case

Divide by zero

Have you noticed that I'm doing anything wrong?

I'm violating the
"no dependencies" rule.

CalculatorController
depends on
Calculator

Dependency Injection

Dependency Injection

If a class depends on some other resource...

...pass it in on the constructor

Relying classes shouldn't create instances of dependencies

Constructing Instances vs. Dependency Injection

```
public class CalculatorController : Controller
{
    private Calculator _Calculator;

    1 reference | 0 changes | 0 authors, 0 changes | 0 exceptions
    public CalculatorController()
    {
        _Calculator = new Calculator();
    }
```

Dependencies

```
public class CalculatorController : Controller
{
    private ICalculatorService _CalculatorService;

    1 reference | Benjamin Day, 151 days ago | 1 author, 2 changes | 0 exceptions
    public CalculatorController(ICalculatorService calculator)
    {
        if (calculator == null)
        {
            throw new ArgumentNullException(nameof(calculator),
                $"{nameof(calculator)} is null.");
        }

        _CalculatorService = calculator;
    }
```

Dependency Injection

Dependency Injection with ASP.NET Core

Turn your dependencies into interfaces

- C# interfaces

Pass them in on the constructor

Register the dependencies in Startup.cs

- ConfigureServices(IServiceCollection)
- Services.AddTransient<>()

Next up:
Refactor To Use Dependency Injection

Demo

Web-based Calculator

Refactor CalculatorController to use Dependency Injection (DI)

Refactor unit tests

In the last demo...

Dependency Injection (DI)

Refactored Calculator to implement ICalculatorService

Refactored CalculatorController to use DI

- Takes instance of ICalculatorService on constructor

Unit tests still use Calculator

Unit tests should be focused.

No dependencies.

Goal:
Unit test the MVC implementation of the web-based calculator

In our unit tests,
do we still need to pass an instance of
Calculator to CalculatorController?

No.

We can pass anything that implements ICalculatorService.

Fakes, Stubs, Mocks – The ultimate break-er of dependencies

Fakes, Stubs, Mocks – Who cares?

Fakes, Stubs, Mocks – Distinctions without a difference

Fakes, Stubs, Mocks – Implement an interface and pretend to do real work

Fakes, Stubs, Mocks –
They all basically provide
canned answers and
never go to production

Behavior Verification

Verify how the SystemUnderTest is interacting with the dependency's interface

Examples:

- Did a method get called?
- How many times was the method called?
- What were the parameters?

Next up:
Refactor to Use Mocks

Demo

- Web-based Calculator
- Break the dependency on Calculator
- Add MockCalculatorService
- Modify the test to use a mock object
- Add behavior verification to tests

On with the demo...

Summary

Unit testing an ASP.NET MVC user interface

Model-View-Controller

Dependency Injection

Mocks, Stubs, and Fakes

Behavior verification

Next up:
Unit Testing Database Stuff

Next up:
Finding creative ways to not test database stuff