Agenda: UI Test using Selenium

- Overview about Functional Tests
- Kind of Functional Tests
- UI Test with Selenium on Local System
- UI Tests in Build and Release Pipeline
- Capture Video

Overview about Functional Tests

What are functional and nonfunctional tests?

- ***Functional tests*** verify that each function of the software does what it should. How the software implements each function isn't important in these tests. What's important is that the software behaves correctly. You provide an input and check that the output is what you expect.
- ***Nonfunctional tests*** check characteristics like performance and reliability. An example of a nonfunctional test is checking to see how many people can sign in to the app simultaneously. Load testing is another example of a nonfunctional test.

Kind of functional tests:

1. Smoke testing

Smoke testing verifies the most basic functionality of your application or service. These tests are often run before more complete and exhaustive tests. Smoke tests should run quickly.
For example, say you're developing a website. Your smoke test might use curl to verify that the site is reachable and that fetching the home page produces a 200 (OK) HTTP status. If fetching the home page produces another status code, such as 404 (Not Found) or 500 (Internal Server Error), you know that the website isn't working. You also know that there's no reason to run other tests. Instead, you diagnose the error, fix it, and restart your tests.

2. Unit testing

unit testing verifies the most fundamental components of your program or library, such as an individual function or method. You specify one or more inputs along with the expected results. The test runner performs each test and checks to see whether the actual results match the expected results.

3. Integration testing

Integration testing verifies that multiple software components work together to form a complete system. For example, an e-commerce system might include a website, a products database, and a payment system. You might write an integration test that adds items to the shopping cart and then purchases the items. The test verifies that the web application can connect to the products database and then fulfill the order.

4. Regression testing

A *regression* occurs when existing behavior either changes or breaks after you add or change a feature. *Regression testing* helps determine whether code, configuration, or other changes affect the software's overall behavior.

Regression testing is important because a change in one component can affect the behavior of another component. For example, say you optimize a database for write performance. The read performance of that database, which is handled by another component, might unexpectedly drop. The drop in read performance is a regression.

5. Sanity testing

Sanity testing involves testing each major component of a piece of software to verify that the software appears to be working and can undergo more thorough testing. You can think of sanity tests as being less thorough than regression tests or unit tests. But sanity tests are broader than smoke tests.

Although sanity testing can be automated, it's often done manually in response to a feature change or a bug fix. For example, a software tester who is validating a bug fix might also verify that other features are working by entering some typical values. If the software appears to be working as expected, it can then go through a more thorough test pass.

6. User Interface Testing

User interface (UI) testing verifies the behavior of an application's user interface. UI tests help verify that the sequence, or order, of user interactions leads to the expected result. These tests also help verify that input devices, such as the keyboard or mouse, affect the user interface properly. You can run UI tests to verify the behavior of a native Windows, macOS, or Linux application. Or you can use UI tests to verify that the UI behaves as expected across web browsers.

A unit test or integration test might verify that the UI *receives* data correctly. But UI testing helps verify that the user interface ***displays*** correctly and that the result functions as expected for the user.

For example, a UI test might verify that the correct animation appears in response to a button click. A second test might verify that the same animation appears correctly when the window is resized.

What tools can I use to write UI tests?

- **Windows Application Driver** (WinAppDriver): WinAppDriver helps you automate UI tests on Windows apps. These apps can be written in Universal Windows Platform (UWP) or Windows Forms (WinForms).
- **Selenium**: Selenium is a portable open-source software-testing framework for web applications. It runs on most operating systems, and it supports all modern browsers. You can write Selenium tests in several programming languages, including C#. In fact, there are NuGet packages that make it easy to run Selenium as NUnit tests. We already use NUnit for our unit tests.

- **SpecFlow**: SpecFlow is for .NET projects. It's inspired by a tool called Cucumber. Both SpecFlow and Cucumber support behavior-driven development (BDD). BDD uses a natural-language parser called Gherkin to help both technical teams and nontechnical teams define business rules and requirements. You can combine either SpecFlow or Cucumber with Selenium to build UI tests.

UI Test with Selenium

The Windows agent that's hosted by Microsoft is already set up to run Selenium tests.

What is Selenium

- Selenium Is an **open source tool** which provides support for automating web applications.
- Selenium supports the following:
  - **Languages**: Java, C#, Python, ruby, Perl, PHP.
  - **Operating Systems**: Windows, Linux, Mac etc.
  - **Browsers**: Chrome, Firefox, Safari, Edge, IE etc.

Components of Selenium:

Selenium is an API having group of components

1. Selenium IDE
2. Selenium RC
3. Selenium Web driver
4. Selenium Grid

What are locators in Selenium?

In a Selenium test, a *locator* selects an HTML element from the DOM (Document Object Model) to act on. Think of the DOM as a tree or graph representation of an HTML document. Each node in the DOM represents a part of the document.

In a Selenium test, you can locate an HTML element by its:

- id attribute.
- name attribute.
- XPath expression.
- Link text or partial link text.
- Tag name, such as body or h1.
- CSS class name.
- CSS selector.

UI Testing with Selenium on Local System

SeleniumTestSolution

    MyWebApp

NUnitTestProject1

Create the Main Web Application

1. Create a New **ASP.NET Core Web Application: MyMathWebApp** in Solution **SeleniumDemoSolution**
2. Add the following MyMath class to the Models Folder **(/Models/MyMath.cs)**

```
public class MyMath
{
    public int N1 { get; set; }
    public int N2 { get; set; }
    public int Result { get; set; }
}
```

3. Add the following to **/Controllers/HomeController.cs**

```
public IActionResult Index()
{
    MyMath m = new MyMath() { N1 = 10, N2 = 5, Result = 15 };
    return View(m);
}
[HttpPost]
public IActionResult Index(MyMath m)
{
    ModelState.Clear();
    m.Result = m.N1 + m.N2;
    return View(m);
}
```

4. Replace the content of **/Views/Home/Index.cshtml** with the following

```
@model MyMath
<div>
    <form method="post">
        N1: <input type="text" id="n1" asp-for="N1" /><br />
        N2: <input type="text" id="n2" asp-for="N2" /><br />
        Result: <input type="text" id="result" asp-for="Result"/><br />
        <input type="submit" name="submit" value="Add" /><br />
    </form>
</div>
```

Add a Unit Test Project

5. Right Click on Solution → Add → New Project
6. Select **NUnit Test Project** (.NET Core) → Next
7. Project Name = "**MyMathWebApp.UITests**" → Create
8. Add the following NuGet Packages.
    a. Selenium.WebDriver
    b. Selenium. Support
    c. Selenium.WebDriver.ChromeDriver
    d. Selenium.Firefox.WebDriver
9. Add the following class to the project

```
using NUnit.Framework;
using OpenQA.Selenium;
using OpenQA.Selenium.Chrome;
using OpenQA.Selenium.Firefox;
using OpenQA.Selenium.IE;
using System;
using System.Threading;

[TestFixture("Chrome")]
[TestFixture("Edge")]
//[TestFixture("Firefox")]
public class IndexPageTest
{
  private string browser;
  private IWebDriver driver;
  public IndexPageTest(string browser)
  {
    this.browser = browser;
  }

  [SetUp]
  public void Setup()
  {
    var cwd = Environment.CurrentDirectory;
    // Create the driver for the current browser.
    switch (browser)
    {
      case "Chrome":
            driver = new ChromeDriver(cwd);
            break;
          case "Edge":
            driver = new Microsoft.Edge.SeleniumTools.EdgeDriver(cwd);
            break;
        default:
            throw new ArgumentException($"'{browser}': Unknown browser");
    }
}

private void ClickElement(IWebElement element)
{
    // We expect the driver to implement IJavaScriptExecutor.
    // IJavaScriptExecutor enables us to execute JavaScript code during the tests.
    IJavaScriptExecutor js = driver as IJavaScriptExecutor;
    // Through JavaScript, run the click() method on the underlying HTML object.
    js.ExecuteScript("arguments[0].click();", element);
}
[Test]
public void AddTest()
{
    if (driver == null)
    {
        Assert.Ignore();
        return;
    }
    driver.Manage().Window.Maximize();
    driver.Url = Environment.GetEnvironmentVariable("SITE_URL");
    IWebElement n1 = driver.FindElement(By.Name("N1"));
    IWebElement n2 = driver.FindElement(By.Name("N2"));
    n1.Clear();
    n2.Clear();
    n1.SendKeys("10");
    n2.SendKeys("3");

    ClickElement(driver.FindElement(By.Name("submit")));
```

```
        IWebElement result = driver.FindElement(By.Name("Result"));

        Assert.AreEqual(result.GetAttribute("value"),"13");

        Thread.Sleep(3000);

        driver.Quit();

    }


}
```

Run the Test

10. Run the Web Application
11. Using Visual Studio - Run the Unit Test using Test Explorer

    OR

    dotnet test --configuration Release MyMathUITest

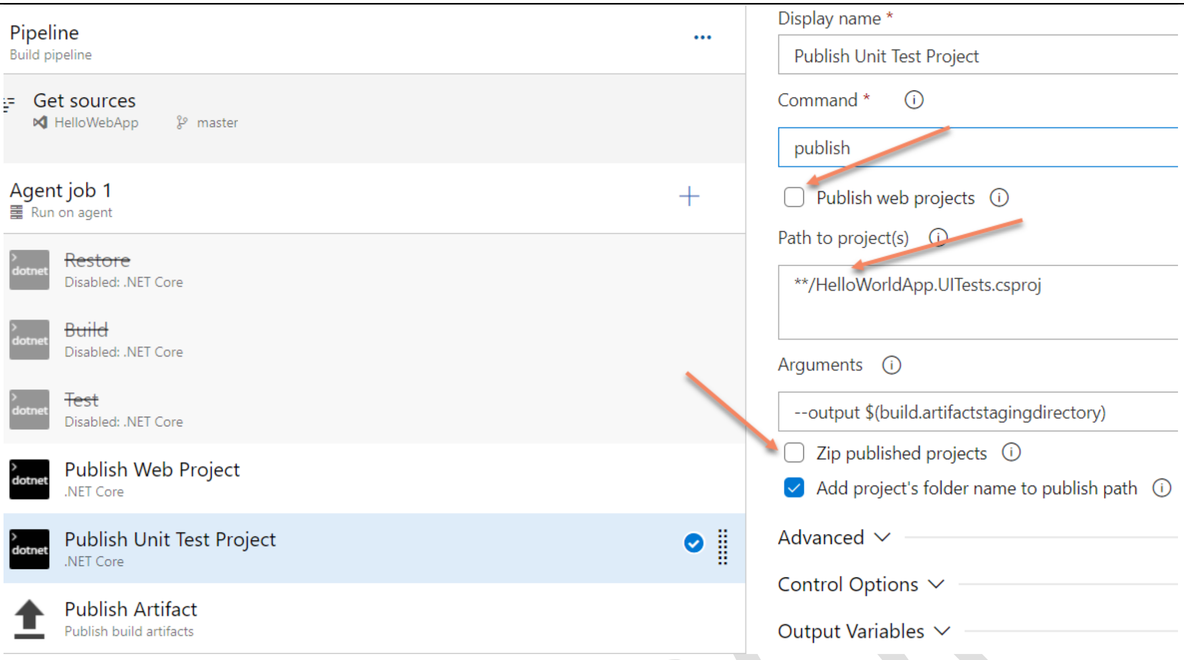
UI Tests in Pipeline

Move the Project to Azure Repos

1. Add **VisualStudio.gitignore** NuGet Package to SeleniumDemoSolution
    a. Right Click on Solution → Manage NuGet Package for Solution…
    b. Browse Tab → Search **gitignore**
    c. Select both project and Install
2. In Solution Folder execute the following commands
    a. git init
    b. git add .
    c. git commit -m "Initial Commit"
3. Azure DevOps → Create a New Project **SeleniumDemoSolution**
    a. Azure Repos → Files → Copy Command to Set Origin and Add
4. Execute the command from local folder to push the project to Repository

Create a Build Pipeline

Use Windows Agent.


Pipeline
Build pipeline

Get sources
HelloWebApp master

Agent job 1
Run on agent

~~Restore~~
Disabled: .NET Core

~~Build~~
Disabled: .NET Core

~~Test~~
Disabled: .NET Core

Publish Web Project
.NET Core

Publish Unit Test Project
.NET Core

Publish Artifact
Publish build artifacts

Display name *
Publish Unit Test Project

Command *
publish

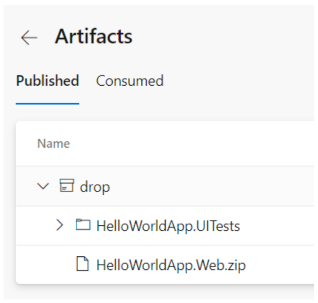
☐ Publish web projects

Path to project(s)
**/HelloWorldApp.UITests.csproj

Arguments
--output $(build.artifactstagingdirectory)

☐ Zip published projects
☑ Add project's folder name to publish path

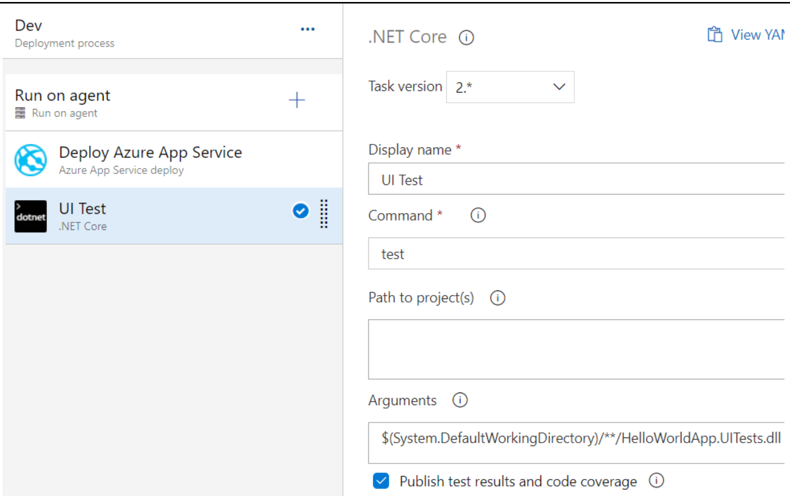
Advanced
Control Options
Output Variables


Artifacts
Published Consumed

Name
drop
HelloWorldApp.UITests
HelloWorldApp.Web.zip

Create a Release Pipeline

1. In Azure Portal, Create an App Service – Web App, Name=MyMathWebApp
2. In DevOps Portal Create a New Service Connection to Azure DevOps
3. Create a New Release Pipeline
   a. Change the Agent to "**windows-2019**"
   b. For Deploy Azure App Service Task: Set Package or folder =
      $(System.DefaultWorkingDirectory)/*/*/MyMathWebApp)
      (Don't mention ** in the path above)
   c. Add Steps **Visual Studio Test Platform Installer** and **Visual Studio Test**



.NET Core

Task version: 2.*

Display name *: UI Test

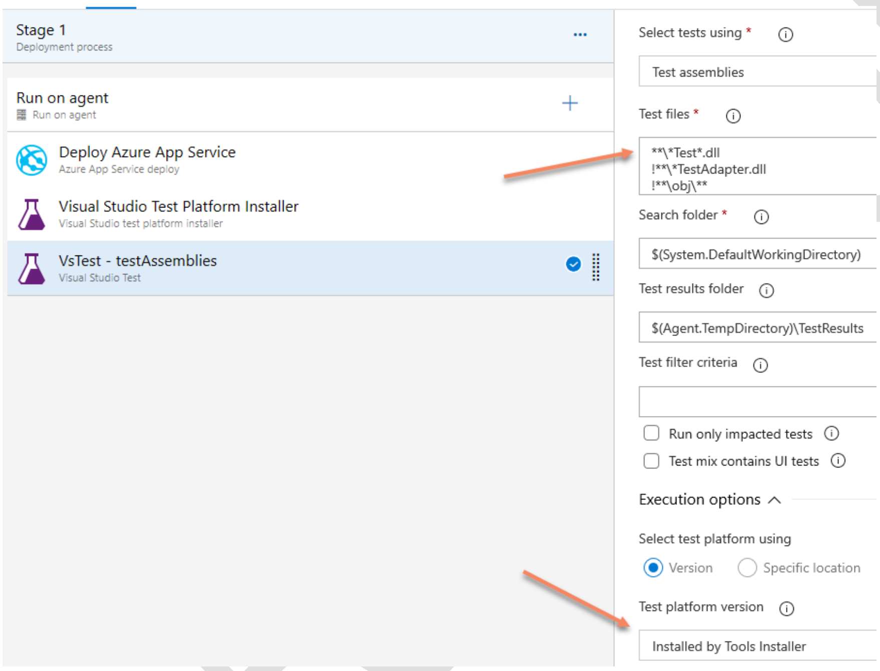
Command *: test

Path to project(s):

Arguments: $(System.DefaultWorkingDirectory)/**/HelloWorldApp.UITests.dll

☑ Publish test results and code coverage

OR



Select tests using *: Test assemblies

Test files *:
```
**\*Test*.dll
!**\*TestAdapter.dll
!**\obj\**
```

Search folder *: $(System.DefaultWorkingDirectory)

Test results folder: $(Agent.TempDirectory)\TestResults

Test filter criteria:

☐ Run only impacted tests

☐ Test mix contains UI tests

Execution options

Select test platform using: [● Version] ○ Specific location

Test platform version: Installed by Tools Installer

Note: We can also use **.NET Core Task** with command as Test.

4. Variables Tab → Add Variable SITE_URL = https://<demosite>.azurewebsites.net

5. Create a Release

Note: You will now get an error - The file D:\a\r1\a_HelloWorldApp-ASP.NET Core-CI\drop\HelloWorldApp.UITests\chromedriver.exe does not exist…

6. In Vistual Studio UITest Project → Copy chromedriver.exe (from its output directory)

7. In Solution Explorer → Select chromedriver.exe → F4 (properties) → **Copy to output directory = Copy Always**

8. Push the changes to Azure Repo.

9. Run - Build and Release Pipelines

10. Look at the Test Results in Test Tab (takes couple of minutes to populate)

Capture Video

If you use the Visual Studio test task to run tests, video of the test can be captured and is automatically available as an attachment to the test result. For this, you must configure the video data collector in a **.runsettings** file and this file must be specified in the task settings.

1. To test project add the file **test.runsettings**

```
<?xml version="1.0" encoding="utf-8" ?>
<RunSettings>
 <DataCollectionRunSettings>
  <DataCollectors>
   <DataCollector uri="datacollector://microsoft/VideoRecorder/1.0"
assemblyQualifiedName="Microsoft.VisualStudio.TestTools.DataCollection.VideoRecorder.VideoRecorderData
Collector, Microsoft.VisualStudio.TestTools.DataCollection.VideoRecorder, Version=15.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" friendlyName="Screen and Voice Recorder">
    <!--Video data collector was introduced in Visual Studio 2017 version 15.5 -->
    <Configuration>
     <!-- Set "sendRecordedMediaForPassedTestCase" to "false" to add video attachments to failed tests only
-->
     <MediaRecorder sendRecordedMediaForPassedTestCase="true"  xmlns="">
      <ScreenCaptureVideo bitRate="512" frameRate="2" quality="20" />
     </MediaRecorder>
    </Configuration>
   </DataCollector>
  </DataCollectors>
 </DataCollectionRunSettings>
</RunSettings>
```

2. Select **test.runsettings** → Properties → **Copy to output directory = "Copy always"**
3. Visual Studio → Test → Select Settings File → Browse to **test.runsettings**
4. Run the Test and View the Video of UI Test.

5. Wait for Build Pipeline to complete.
6. Edit Release Pipeline → VsTest – testAssemblies → **Settings file** = (browse and visit the file)
   $(System.DefaultWorkingDirectory/**/HelloWorldApp.UITests/test.runsettings
7. Create Release

8. Goto **Tests Tab** → Expant Outcomes and Check Passed → Select Passed Test → Attachments Tab → Select **ScreenCapture.wmv** file → Download and Play

Automating Selenium Tests in Azure Pipelines

https://azuredevopslabs.com/labs/vstsextend/selenium/