# Power Apps Testing – Extending and Contributing to the PowerApps Test Engine

## Overview

**Microsoft Power Apps** provide all users with the ability to rapidly develop business custom applications. Power Apps democratizes the development experience, enabling the creation of apps without writing code. When code is needed, both makers and professional developers are empowered with low code (**Power Fx**) and pro code options. The platform is completely extensible, allowing interaction with cross cloud and on-premise data. **Microsoft Power Apps** also provides custom connectors and integrations to connect to virtual any data source.

In this series, we are focusing on how organizations can incorporate **Power Apps** into their suite of business-critical applications. All organizations require some level of testing, from unit to stress, all applications needed by the enterprise must be resilient. In this series, we will attempt to lay out the necessary tools and design to empower organizations to implement a testing strategy for **Power Apps**.

For Model Driven Applications, I highly encourage you to check out [my other series on EasyRepro and Test Automation.](https://community.dynamics.com/365/b/crminthefield/posts/test-automation-and-easyrepro-01---overview-and-getting-started)

This specific section will discuss how to extend the PowerApps Test Engine. We will look to describe the architecture of the source code to understand how the components interact with each other. This will set the foundation allowing us to extend the tooling to suit our business needs. We will take an example, walk through the steps to do implement and how to contribute back to the PowerApps Test Engine.

This section covers professional development topics and will require a basic understanding of the C# language. I’ll attempt to make this topic as approachable as possible but want to set the expectation of the skillset involved.

## What is the Power Fx Engine?

[Power Fx](https://learn.microsoft.com/en-us/power-platform/power-fx/overview) is the low-code language that is used across **Microsoft Power Platform**. It's a general-purpose, strong-typed, declarative, and functional programming language. The anticipation is it will continue to grow as the preferred language of the platform, for apps, flow, bots, etc. **Power Fx** even has the ability to [transform natural language](https://learn.microsoft.com/en-us/power-apps/maker/canvas-apps/power-apps-ideas-train-examples) to low-code as well as [provide recommendations](https://learn.microsoft.com/en-us/power-apps/maker/canvas-apps/power-apps-ideas-transform?source=recommendations).

In this section, **Power Fx** is used to define what actions the tests are performing.

To learn more about **Power Fx** including a curated learning path, [start here](https://learn.microsoft.com/en-us/training/paths/use-basic-formulas-powerapps-canvas-app/?ns-enrollment-type=Collection&ns-enrollment-id=m0js310oz3r5z).

## Working with the PowerApps Test Engine source code

**The source code is available on GitHub and is open for replication and contribution. This allows the community to contribute while maintaining a specific version of the test engine.**

**The source code included the assembly (Microsoft.PowerApps.TestEngine) used by the test engine that tests will be submitted (PowerAppsTestEngine).**

## **The PowerApps Test Engine Architecture**

## **The Microsoft.PowerApps.TestEngine Architecture**

This includes the following:

|  |  |
| --- | --- |
| Property | Description |
| [Test Reporter](https://github.com/microsoft/PowerApps-TestEngine/blob/main/src/Microsoft.PowerApps.TestEngine/Reporting/TestReporter.cs) | Defines the test suites, cases and runs and generates a test report. Depends on the File System and will write to file by default. |
| [Power Fx Engine](https://github.com/microsoft/PowerApps-TestEngine/blob/main/src/Microsoft.PowerApps.TestEngine/PowerFx/PowerFxEngine.cs) | Works directly with the Power Fx Engine and is responsible for defining which functions can be used by PowerApps Test Engine. **We will explore how to update this in the following section.** |
| [Test Infrastructure Functions](https://github.com/microsoft/PowerApps-TestEngine/blob/main/src/Microsoft.PowerApps.TestEngine/TestInfra/PlaywrightTestInfraFunctions.cs) | Creates the browser configuration, any mock responses and navigates to the Url. Relies on the test state and file system. |
| [User Manager](https://github.com/microsoft/PowerApps-TestEngine/blob/main/src/Microsoft.PowerApps.TestEngine/Users/UserManager.cs) | Manages user log-in including working with environment variables, routing to the log-in url and submitting credentials. Defines the input types for the Microsoft log-in page (e.g. idBtn\_Back). |
| Test State |  |
| Url Mapper |  |
| File System |  |
| Logger factory |  |

## Configuring a new function for PowerApps Test Engine

The test settings section allows testers to define which devices and browsers will be used to run the tests. Test settings also provide additional properties to define how the tests are run such as interactive mode and parallelism. Also included are default settings to record a video of the tests or how long to wait until the test should end if it doesn’t receive a response from an action.

## Test Best Practices

The documentation for Power Apps Test Studio does a really good job of laying out fundamental best practices including:

* Keeping test cases small.
* Keeping expressions to a single action in a test.
* Building deterministic tests.
* Managing multiple tests with test suites.

A few points I want to call out about tests come from the Test Desiderata. Building upon the deterministic attribute above, we must consider the fact we are running tests against an ever-changing variable, the browser. Fundamentally our tests should always provide the same result and I expect they should.

However, it is key that we are testing against the build version (vNext) understanding this is not what the current user base may see. I suggest running the same tests across not only the vNext but the vCurrent of the browser. This also applies to devices. Consider the landscape of devices available that users interact with. Tablets, phones, watches, of all different builds and versions. Attempting to test across this can provide cumbersome. It’s important that as we define our test strategy, we make every good faith attempt to provide coverage but clearly set expectations to the user base and decision makers. If you are interested, I highly recommend finding a used copy of [Dino Esposito’s “Architecting Mobile Solutions for the Enterprise”.](https://www.microsoftpressstore.com/store/architecting-mobile-solutions-for-the-enterprise-9780735663022) The section covering WURFL is especially appealing.

## Dynamically Setting Variables

Not to look too far ahead, but by now we should be planning how to run multiple test cases and suites at scale. Most likely this will take form in an automated fashion using test runners. Each test runner engine has various ways to configure variables that can be passed into and used within each test run.

In the case of **PowerApps-TestEngine**, the **filePath** property is extremely helpful as we look to dynamically set variables. Referencing a singular file for configuration can make our lives easier by simply modifying the single file which replicates to all tests.

Using variables to modify the single file makes this process even more agile.

## Next Steps

By now, you should have a firm understanding of test tools available for Canvas Apps. You should also be able to articulate and define test suites and cases. You should be able to show how to configure tests within a specific suite and globally across all tests.