# Power Apps Testing – Troubleshooting and Monitoring 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.

## Troubleshooting Basics

Its recommend, when troubleshooting, to try to reproduce the issue. Once the issue is reliably reproducible, troubleshooting can be as easy as walking through the steps, identifying a failure and correcting.

Start with reviewing the logs from Power Apps Test Engine and the test runner if running in an automated fashion. For test runner details, refer to the infrastructure documentation. Most likely, the issue will come from the steps needed to build and execute Power Apps Test Engine.

Once Power Apps Test Engine has been built and is executing, look to the information provided in the logs. Power Apps Test Engine has a mechanism built in, covered in Section 03, allowing for different log levels to be output. Review the current configuration and if needed, increase the level of logging down to Verbose.

### Error: : System.ArgumentException: locale

lll

Error: : Timed out during login attempt.

In order to determine why, it may be beneficial to view the output recording. Make sure that your login credentials are correct.

Error: : System.TimeoutException: The operation has timed out.

### Error: : Something went wrong when Test Engine tried to get App status.

Error: : System.TimeoutException: Something went wrong when Test Engine tried to get App status.

## Extending Logging Information

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 code for the test engine is very straightforward. It consists of a TestEngine project that interacts directly with [PowerFx code found here](https://github.com/microsoft/Power-Fx). This TestEngine project has an accompanying test project providing a quick way to test various areas of the engine. This includes testing working with user personas and different configurations. It also test underlying browser automation frameworks such as ***playwright*** or ***selenium*** if desired.

Diagram

Description automatically generated

The above image shows the design of the **PowerApps Test Engine** and how it interacts with tests and the Power App. Within the red box is the code located in the **PowerApps-TestEngine** source.

Power Apps tests go in, actions are executed against the app and results are output. For most cases, interaction only with the *PowerAppsTestEngine* is needed.

That said, there might come a time where the *TestEngine* itself needs to be extended.

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

The **Microsoft.PowerApps.TestEngine** is the primary component of the **PowerApps-TestEngine** project. It creates and maintains the test infrastructure needed to run our Power Fx tests. The project uses the test supplied and can execute the tests as instructed by the *PowerAppsTestEngine* application. Let’s look at the architecture.

Diagram

Description automatically generated

The **Microsoft.PowerApps.TestEngine** is responsible for all test components, including delivering Power Fx statements to the **Microsoft.PowerFx** framework. The entry point is the Single Test Runner and 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](https://github.com/microsoft/PowerApps-TestEngine/blob/main/src/Microsoft.PowerApps.TestEngine/Config/TestState.cs) | Defines the test plan definition and settings used to run tests. |
| [Url Mapper](https://github.com/microsoft/PowerApps-TestEngine/blob/main/src/Microsoft.PowerApps.TestEngine/PowerApps/PowerAppsUrlMapper.cs) | Used to generate the Power App url and pass test parameters. |
| [File System](https://github.com/microsoft/PowerApps-TestEngine/blob/main/src/Microsoft.PowerApps.TestEngine/System/FileSystem.cs) | Used for reading and writing to disk. Uses System.IO. |
| Logger factory | Derived from Microsoft.Extensions.Logging. Could be used to inject |

## Configuring a new function for PowerApps Test Engine

Start by reviewing the [PowerFx Engine](https://github.com/microsoft/PowerApps-TestEngine/blob/main/src/Microsoft.PowerApps.TestEngine/PowerFx/PowerFxEngine.cs). This class creates a Power Fx configuration used by the [Recalc Engine](https://github.com/microsoft/Power-Fx/blob/main/src/libraries/Microsoft.PowerFx.Interpreter/RecalcEngine.cs). The image below is a snapshot, review the ***Setup*** function for existing functionality.

Text

Description automatically generated

Highlighted above is the declaration of the *PowerFxConfig* object, the addition of functions used by the Test Engine and the creation of the *RecalcEngine*.

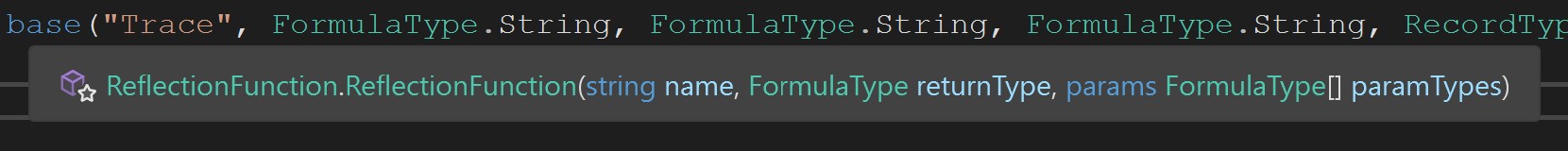
## Defining the Trace Function

Defining a new function requires the use of the *ReflectionFunction* class. Start with implementing this base class as shown below.

Text

Description automatically generated

Define the constructor with the *ILogger* interface as a dependent parameter. Further define with the base constructor properties.



The name parameter will be what the Power Fx test will call. The return type property can help respond with an object of string. Finally, the param types will allow us to define one or more properties for the Power Fx function.

## Defining the Execute function

Using a new function called Execute, further define the actual implementation of the Power Fx test action.

In this case, the code will create the *Trace* function and will use the Power Fx documentation located here. The documentation calls out three properties: the message, the severity level and a custom object.

Text

Description automatically generated

The image above shows how to loop through the properties of the *RecordValue*. Use the PrimitiveValue<T> to get the value of the object property.

In this example, the following code is used within the Power Fx test:

**Trace(“Sample message”, “Information”, {sampleKey: “sampleValue”});**

## Adding the command to the Test Engine command list

Navigating to the *PowerFxEngine*, add the following to the setup. By doing this, we are now adding our Trace command to the list of available commands executable in a Power Fx test.

Text

Description automatically generated

## Debugging and Testing Locally

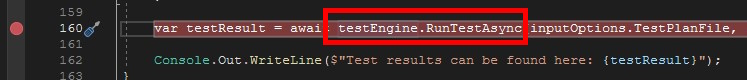
Building and Running tests against the source code will help pause the execution allowing for deep insights into the runtime. You can use multiple IDE’s such as **Visual Studio Code** or **Visual Studio**. I chose to go with Visual Studio 2022.

Look at the image below, highlighting the three projects and the default project used for debugging.

Graphical user interface, text, application

Description automatically generated

Using the *PowerAppsTestEngine* project and starting the debugger will run the code in real time. The benefit to this is we can add stops or breakpoints in the code to better understand how the code is working. In the example below, a breakpoint has been set on the before entering into the [***RunTestAsync***](https://github.com/microsoft/PowerApps-TestEngine/blob/522401e9c03a049e28bce3f48e951ec6b04f2a0b/src/PowerAppsTestEngine/Program.cs#L160) function. This allows developers to review and modify each input.



## Configuring Debug Properties

*PowerAppsTestEngine* is an executable that requires environment variables for the user persona. Optionally, arguments can be passed in further defining what the engine can do.

**To set environment variables during debug, open the project properties**. Navigate to the debug properties and define the variables. Depending on the IDE you use this process may change slightly.

Graphical user interface, text

Description automatically generated

Command line arguments can be provided during the debug session. Review the argument list in *PowerAppsTestEngine* for the latest options. As with the trace function, if an argument doesn’t exist but is needed, it can be added and contributed.

I’ve found that adding verbose tracing is essential when stepping into each test case step. Using the   
“-l 0” we can enable verbose tracing during debugging.

## Contributing to the source

The source code for the **PowerApps Test Engine** is located on GitHub. Contributions can be made using standard GitHub procedures. In this case, we will Fork the project, make changes and make a pull request.

## Fork Power Apps Test Engine

Begin by navigating to the source code and clicking the Fork button. This will present an option to fork the project into a specific account.

Graphical user interface, text

Description automatically generated

## Clone local and commit changes

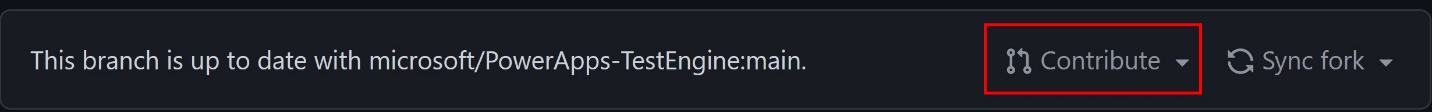
Clone the fork to a workstation and commit the changes to a new branch. Ideally the fork and branch would have been done before creating the code, but changes can be moved into the forked project as well.

Its key to point out that once the project has been forked, you are free to make any changes to the forked project. No changes will be applied to the source code until a Pull Request has been made.

Once all the changes are ready, commit the changes and push remotely.

## Creating a Pull Request and Contributing back to Power Apps Test Engine

Once the changes have been pushed to your GitHub forked project, you can continue working on the forked project. If you have a contribution, use the “Contribute” button to create a Pull Request. A Pull Request is an attempt to merge your changes into the source. This will require validation and authorization.



Once the Pull Request has been merged, you have **successfully contributed to Power Apps Test Engine**.

## 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.

You have now learned how to modify the source code for the Power Apps Test Engine. Additionally, you can now articulate the steps needed to contribute back to the open source project.

Continue evaluating any gaps in the **Power Apps Test Engine** for your business needs. If a more complex command is needed review other commands such as the Select, which perform an action and retrieve updates to the app.