The case of the feature flag that didn’t stay on long enough, part 2

devblogs.microsoft.com/oldnewthing/20250418-43/?p=111081

April 18, 2025

Last time, we looked at a crash related to applying a temporary override of a feature flag, but our first attempt to fix it didn’t help.

If you look closely at the call stack for the crash, and read from the bottom up to follow execution in chronological order, we first have the TAEF command execution thread freeing the unit test library, which starts tearing down global variables, and it has reached the `WidgetRouter` singleton:

```
unittests!WidgetRouter::GetInstance'::`2'::`dynamic atexit destructor for
'singleton''
```

The `WidgetRouter` apparently keeps a vector of objects that need closing, and one of them is another Widget, on which it dutifully calls `Close`. The `Widget::Close` method asks for the singleton `WidgetRouter`, and now the assertion fails because the feature override is not in effect.

The feature override was released at the end of the unit test function, and that finished a long time ago. TAEF is trying to clean up after all the tests have completed.

The problem is that the Widget was created while the feature flag was set (due to the override), but it is being closed while the feature flag is clear (because the override has been lifted). This is a “torn state” situation that generally makes nobody happy.

To avoid torn state, you have find everything that changes behavior based on the feature flag and tear it down or at least put it back into a state that is not dependent on the feature flag, and do this before you change the feature flag.

One solution for the purpose of this unit test is to override the `WidgetRouter::Get-Instance` method to return a custom `WidgetRouter` for the state where the feature flag is enabled. Running it down at the end of the function allows everything to clean up while the override is still in place, thereby avoiding the torn state problem.

Here's one possible implementation:

```
/* static method */
std::shared_ptr<WidgetRouter>
    WidgetRouter::GetInstance()
{
    auto result = UnitTest_Override_WidgetRouter_GetInstance();
    if (result) return *result;

    assert(FeatureFlags::IsEnabled(NewFeatureId));

    static std::shared_ptr<WidgetRouter> singleton =
        std::make_shared<WidgetRouter>();

    return singleton;
}
```

where the production code has a stub implementation of the unit test override (that always returns `nullptr`), and the unit test has an implementation that returns a special `WidgetRouter` for unit testing purposes.

```
std::shared_ptr<WidgetRouter> g_unitTestRouter;

std::shared_ptr<WidgetRouter>* UnitTest_Override_WidgetRouter_GetInstance()
{
    return std::addressof(g_unitTestRouter);
}

void WidgetTests::BasicTests()
{
  // Force the feature flag on for the duration of this test
  auto override = std::make_unique<FeatureOverride>(NewFeatureId, true);

  // Create our custom widget router for this unit test.
  // This router's lifetime is enclosed by the lifetime of the
  // feature flag override.
  g_unitTestRouter = std::make_shared<WidgetRouter>();

  auto cleanup = wil::scope_exit([] {
    // Destroy the custom widget router. This will run it down while the
    // feature flag override is still in effect.
    g_unitTestRouter = nullptr;
  });

  auto widget = winrt::Component::Widget();
  ⟦ test the widget in various ways ⟧
}
```

Here, we tell `WidgetRouter::GetInstance()` to use our `g_unitTestRouter` instead of its private one. That way, we can run down the `WidgetRouter` while the feature flag override is still in effect and avoid torn state.

But there's a simpler solution: Give the unit test a way to force the singleton `Widget-Router` to run down.

```
static std::shared_ptr<WidgetRouter> g_singletonWidgetRouter;

/* static method */
std::shared_ptr<WidgetRouter>
    WidgetRouter::GetInstance()
{
    assert(FeatureFlags::IsEnabled(NewFeatureId));

    static bool init = [] {
        g_singletonWidgetRouter = std::make_shared<WidgetRouter>();
    }();

    return g_singletonWidgetRouter;
}

/* static method */
void WidgetRouter::ResetInstance_ForUnitTestOnly()
{
    g_singletonWidgetRouter = nullptr;
}

void WidgetTests::BasicTests()
{
  // Force the feature flag on for the duration of this test
  auto override = std::make_unique<FeatureOverride>(NewFeatureId, true);

  // Clean up the widget router before the override goes
  // out of scope.
  auto cleanup = wil::scope_exit([] {
    WidgetRouter::ResetInstance_ForUnitTestOnly();
  });

  auto widget = winrt::Component::Widget();
  ⟦ test the widget in various ways ⟧
}
```