# Processing a ValueSet or PropertySet even in the face of possible mutation, part 4

devblogs.microsoft.com/oldnewthing/20220715-00

July 15, 2022

Raymond Chen

The customer that was trying to apply the conclusions of this series had a scenario where the PropertySet had both an auto-save feature, as well as an explicit-save feature. So they tried refactoring:

```
std::optional<SomeKindOfDataBuffer> Widget::SaveToBuffer()
{
    try {
        SomeKindOfDataBuffer buffer;
        auto it = m_propertySet.First();
        if (it.HasCurrent()) {
            do {
                auto current = it.Current();
                buffer.AddKeyAndValue(current.Key(), current.Value());
            } while (it.MoveNext());
        }

        auto guard = m_lock.lock();

        // verify that the collection is still unchanged before returning
        std::ignore = it.HasCurrent();

        return buffer;
    } catch (winrt::hresult_changed_state const&) {
        // Abandon the operation.
        return std::nullopt;
    }
}

void Widget::OnPropertySetChanged()
{
    auto buffer = SaveToBuffer();
    if (buffer) {
        SaveToAutoSaveLocation(*buffer);
    }
}

void Widget::Save()
{
    auto buffer = SaveToBuffer();
    if (!buffer) throw winrt::hresult_changed_state();
    SaveBuffer(*buffer);
    return true;
}
```

The code to save the property set to a buffer is factored into a helper function, which we then use in the two code paths.

Unfortunately, the refactoring reintroduced the bug.

The purpose of the lock is to prevent a competing auto-save from racing ahead of the current auto-save. But the `SaveToBuffer()` releases the lock before returning. The new auto-save code reopens the race condition:

| Thread 1 | Thread 2 |
|---|---|
| Insert begins | |
| OnPropertySetChanged begins | |
| SaveToBuffer begins | |
| Create the buffer | |
| Enter the lock | |
| Check for concurrent mutation | |
| Release the lock | |
| Return from SaveToBuffer | |
| | Insert begins |
| | OnPropertySetChanged begins |
| | SaveToBuffer begins |
| | Create the buffer |
| | Enter the lock |
| | Check for concurrent mutation |
| | Release the lock |
| | Return from SaveToBuffer |
| | SaveToAutoSaveLocation |
| | OnPropertySetChanged returns |
| | Insert returns |
| SaveToAutoSaveLocation | |
| OnPropertySetChanged returns | |
| Insert returns | |

Once you release the lock, another thread can come in and mutate the collection and save the changes. And then you get around to saving the changes, and you overwrote the good changes with out-of-date ones.

Fortunately, we can solve this problem by returning the iterator, too!

```
std::pair<std::optional<SomeKindOfDataBuffer>,
          winrt::Iterator<winrt::IKeyValuePair<winrt::hstring,
                                               winrt::IInspectable>>>
Widget::SaveToBuffer()
{
    SomeKindOfDataBuffer buffer;
    auto it = m_propertySet.First();
    try {
        if (it.HasCurrent()) {
            do {
                auto current = it.Current();
                buffer.AddKeyAndValue(current.Key(), current.Value());
            } while (it.MoveNext());
        }

        // auto guard = m_lock.lock();

        // verify that the collection is still unchanged before returning
        // std::ignore = it.HasCurrent();

        return { buffer, it };
    } catch (winrt::hresult_changed_state const&) {
        // Abandon the operation.
        return { std::nullopt, it };
    }
}
```

The two callers can then check the validity of the iterator to determine whether or not to proceed. In the case of auto-save, the check can be performed under the lock.

```
void Widget::OnPropertySetChanged()
{
    auto [buffer, it] = SaveToBuffer();
    if (buffer) {
        auto guard = m_lock.lock();
        try {
            // final check under the lock
            std::ignore = it.HasCurrent();
            SaveToAutoSaveLocation(*buffer);
        } catch (winrt::hresult_changed_state const&) {
            // Abandon the operation.
        }
    }
}

void Widget::Save()
{
    auto [buffer, it] = SaveToBuffer();

    // Verify that the collection wasn't changed
    // while we were trying to build the buffer.
    std::ignore = it.HasCurrent();

    SaveBuffer(*buffer);
}
```

An alternative is to make the caller provide the iterator:

```
std::optional<SomeKindOfDataBuffer>
Widget::SaveToBuffer(
    winrt::Iterator<winrt::IKeyValuePair<winrt::hstring,
                                         winrt::IInspectable>> it)
{
    try {
        SomeKindOfDataBuffer buffer;
        // auto it = m_propertySet.First();
        if (it.HasCurrent()) {
            do {
                auto current = it.Current();
                buffer.AddKeyAndValue(current.Key(), current.Value());
            } while (it.MoveNext());
        }

        // auto guard = m_lock.lock();

        // verify that the collection is still unchanged before returning
        // std::ignore = it.HasCurrent();

        return buffer;
    } catch (winrt::hresult_changed_state const&) {
        // Abandon the operation.
        return std::nullopt;
    }
}

void Widget::OnPropertySetChanged()
{
    auto it = m_propertySet.First();
    auto buffer = SaveToBuffer(it);
    if (buffer) {
        auto guard = m_lock.lock();
        try {
            // final check under the lock
            std::ignore = it.HasCurrent();
            SaveToAutoSaveLocation(*buffer);
        } catch (winrt::hresult_changed_state const&) {
            // Abandon the operation.
        }
    }
}

void Widget::Save()
{
    auto it = m_propertySet.First();
    auto buffer = SaveToBuffer(it);

    // Verify that the collection wasn't changed
    // while we were trying to build the buffer.
    std::ignore = it.HasCurrent();
```

```
    SaveBuffer(*buffer);
}
```

But the pattern I am currently a fan of is to make the caller do the exception handling, too:

```
SomeKindOfDataBuffer Widget::SaveToBuffer()
{
    SomeKindOfDataBuffer buffer;
    for (auto [name, value] : m_propertySet)
        buffer.AddKeyAndValue(name, value);
    }
    return buffer;
}

void Widget::OnPropertySetChanged() try
{
    auto it = m_propertySet.First();
    auto buffer = SaveToBuffer();

    auto guard = m_lock.lock();
    // final check under the lock
    std::ignore = it.HasCurrent();

    SaveToAutoSaveLocation(buffer);
} catch (winrt::hresult_changed_state const&) {
    // Abandon the operation.
}

void Widget::Save()
{
    auto buffer = SaveToBuffer();
    SaveBuffer(buffer);
}
```

Raymond Chen

**Follow**