The error code you get might not be the one you want

devblogs.microsoft.com/oldnewthing/20220119-00

January 19, 2022

Raymond Chen

A customer was following the instructions on how to create a register a background task from a Win32 desktop app, but they found that the sample code failed with an exception:

```
bool taskRegistered = false;
std::wstring sampleTaskName = L"SampleTask";
auto allTasks = BackgroundTaskRegistration::AllTasks();
//                                          ^^^^^^^^^ exception

for (auto const& task : allTasks)
{
    if (task.Value().Name() == sampleTaskName)
    {
        taskRegistered = true;
        break;
    }
}

// The code in the next step goes here.
```

The code for the exception was `ERROR_NOT_FOUND` (“Element not found”).

Which is weird. I mean, I’m asking for all of the tasks, and you’re telling me “I couldn’t find any” and throwing an exception? Shouldn’t it just be returning an empty collection if there aren’t any existing tasks?

Indeed, it does return an empty collection if there aren’t any existing tasks. The error code is not telling you that there are no existing tasks. The error code is telling you that your app doesn’t have a packaged app identity (probably because it’s not packaged at all). The thing that couldn’t be find was the package identity.

Okay, sure, but shouldn’t the error code have been `APPMODEL_ERROR_NO_PACKAGE` (“The process has no package identity”)?

Error codes are one of the places where abstractions are leakiest, because errors merely propagate out from their point of origin, and the context in which the error is generated may not be anything the application developer is aware of.

In this case, what's happening is that the request is going down through the background task infrastructure. The request goes out to a server, and the server tries to find out who is calling. There's an internal function for identifying the caller, and it works by looking for information inside the caller's token.

If the caller does not have package identity, then the "Find the thing in a token" function returns `ERROR_NOT_FOUND` because "I couldn't find the thing you asked about." That low-level function doesn't know *why* the caller is asking for the thing. It is just reporting the simple fact that the thing the caller wants is not present.

As the error propagates out of the system, nobody thinks to say "Let me transform that generic `ERROR_NOT_FOUND` into this other more specific-sounding error code," so what comes out is a generic `ERROR_NOT_FOUND`.

Which is technically correct, but in the context of something you the application developer are probably unaware of.

As a general rule, `HRESULT`s and Win32 error codes are like that. The actual numeric value is often descriptive of a low-level situation far removed from the high-level operation the application requested. In the absence of specific documentation to the contrary, the only stable thing about `HRESULT` and Win32 error codes is whether or not they succeeded. If they report failure, you might be able to use the specific value to help guide debugging, but it's not usually expected that they be specific with pinpoint accuracy.

This is one reason why the Windows Runtime design guidelines recommend that failures that application are expected to reason over should be reported with a specific enumeration, rather than using `HRESULT`s. This makes the stability boundary explicit: The system will make sure that the errors map to one of the values of the enumeration, even if under the covers they are aggregated from multiple sources. It is the system's responsibility to take those failures and convert them to something that the application can rely on not changing.

Bonus chatter: As far as I can tell, the `APPMODEL_ERROR_NO_PACKAGE` error code is used only by a handful of functions in `appmodel.h`.

Raymond Chen

Follow