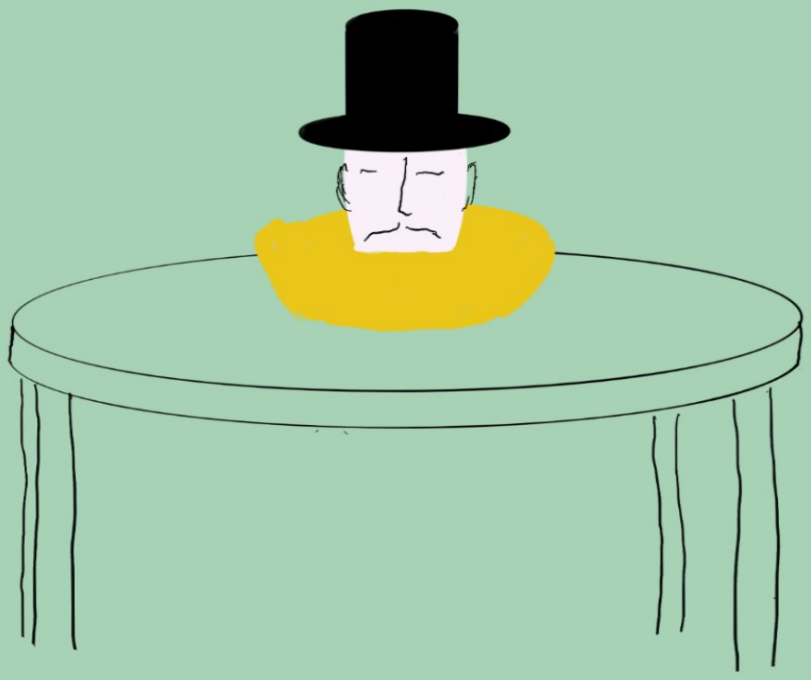
The Case of the Vanishing Bodies



An Introduction to Swift Macros

Daniel H Steinberg

The Case Of The Vanishing Bodies

An Introduction To

Swift Macros

by Daniel H Steinberg

Editors Cut

Copyright

ISBN-13: 978-1-944994-06-8

Book Version

This is the initial release of this book for Swift 5.10, Xcode 15.3, macOS Sonoma 14.4, and iOS 17.4 released April 2024. All code has been tested on Apple Silicon.

Code Download

Visit https://github.com/editorscut/ec015swiftmacros for all of the code for this book.

Run it in Xcode 15 or higher (15.3 if possible). All code is written in Swift.

To avoid long lines and code that wraps, I've split some lines in code listings in ways that you might not in an IDE. Please feel free to not break the lines where I have.

Recommended Settings

The ePub is best viewed in scrolling mode on an iPad. On smaller devices I also choose landscape. For some reason that I don't

understand, scrolling mode is supported by Apple's Books app on the iPad but not on the Mac. If you view this book in Apple's Books app, choose "Let lines break naturally" in Preferences > General. Finally, I've gone to great pains to make this look good in light and dark mode but Apple has foiled me yet again. I'm told that not all of the syntax coloring works in dark mode.

Submit Errata

Submit your errata here for the book or for the source code by selecting New Issue. Please provide the book version listed above, chapter, section, and page number in your issue so that I can find it and, if possible, resolve it quickly.

Official Links

Please check http://developer.apple.com for additional resources including videos, sample code, documentation, and forums. You'll also find information on what is required to take advantage of these resources.

Apple has posted videos, slides, and sample code from the Worldwide Developers Conference.

Legal

The sample code is intended to be used to illustrate points made in the text. It is not intended to be used in production code.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks or service marks. Where those designations appear in this book, and Dim Sum Thinking, Inc. was aware of the trademark claim, the designations have been printed with initial capital letters or in all capitals.

This book uses terms that are registered trademarks of Apple Inc. for which the terms of use don't permit rendering them in all caps or initial caps. You can view a complete list of the trademarks and registered trademarks of Apple Inc at http://www.apple.com/legal/trademark/appletmlist.html.

The Editor's Cut name and logo are registered trademarks of Dim Sum Thinking, Inc.

Table Of Contents

CHAPTER 3

Freestanding Macros

The freestanding macro we created in Chapter 2 was an expression macro. The macro produced an expression.

You saw that our expression macro `unwrapOrDie` was implemented in `UnwrapOrDieMacro` which conformed to `ExpressionMacro`.

Although we didn't make a big fuss about what this conformance means, it mainly means that the `expansion()` method that `UnwrapOrDieMacro` implements returns `ExprSyntax`. Casually, we would say that `expansion()` returns an expression.

We begin this chapter by creating another expression macro. In fact, you're going to do all the work. Then we'll replace the `fatalError()`s with Swift `Error`s. We can test for these errors using a feature of Swift macros called diagnostics.

We'll pause briefly in creating new macros to use the macro. We use `unwrapOrDie` in a project and we use it in the implementation of another macro.

The reason we began with examples of a freestanding expression macro, is that what the Xcode template stubs out for us. The final macro we create in this chapter is a freestanding declaration macro. It is implemented in a struct that conforms to `DeclarationMacro`. The

`expansion()` method returns `[DeclSyntax]`. As you might guess from the name, it returns an array of declarations. Our example will return an enum.

We come full circle at the end of this chapter by exploring more useful error messages we can generate using diagnostics.

Throwing Errors

"Swiftly," said Edges, pausing to enjoy the coffee and croissant I'd delivered. "Yesterday, when the man you were following boarded the bus, you just quit."

"I didn't know what to do," I said.

"Was there nothing else you could have done?"

I looked puzzled. There hadn't seemed to be anything possible at the time.

"For instance," said Edges, "you might have sent me a message alerting me that something had gone wrong. Perhaps, I could have adjusted the assignment or provided you with an alternative."

I nodded. Quitting without any chance to recover was an extreme choice. It would have indeed been better to send Edges a text and allowed them to give me further instructions.

Because a macro is valid Swift code, the compiler can check to see if we've called it correctly.

When things go wrong in the actual implementation of the macro, we've been quitting using `fatalError()`. There are times when it would be more helpful to the caller if we issued a Swift `Error` instead.

In this section we create a custom `Error` type and throw errors from our code if the input passes the compiler but is not acceptable either because

it does not correspond to a valid URL or it is not a single `String` literal.

In order to check for these errors, we'll modify our tests to use `Diagnostic`s.

Continue with our current package or start with the `URL` package in *Chapter03/01/* by double-clicking *Package.swift*.

Compile time checks on Macro calls

There are conditions that could arise if someone doesn't call our macro correctly.

Suppose they call `URL` with no parameters, too many parameters, the wrong type of parameter, or the wrong label for the parameter.

In any of those cases, the compiler will flag an error so we don't need to handle these potential problems.

To see this, add the following line to *main.swift*.

*URL/Sources/URLClient/**main.swift***
```
let temporaryURL = #URL()
```

You may have to build the code before you see the error:

```
Missing argument for parameter #1 in macro expansion
```

Good. We have an error because `#URL` requires a parameter.

Suppose we mistakenly add a label.

*URL/Sources/URLClient/**main.swift***
```
let temporaryURL = #URL(string: "http://example.com")
```

Now we see this helpful error message in Xcode.

```
Extraneous argument label 'string:' in macro expansion
```

Finally, what if we pass the wrong type into `URL`.

*URL/Sources/URLClient/**main.swift***
```
let temporaryURL = #URL(27)
```

Again, the compiler has our back.

```
Cannot convert value of type 'Int' to expected argument type 'String'
```

One of the many nice features of Swift Macros is that they must take valid Swift Code as their input and it can be typechecked against the definition.

The definition for `URL` is

```
@freestanding(expression)
public macro URL(_ string: String) -> URL
```

so we know it takes exactly one parameter, this parameter must be of type `String`, and this parameter has no label.

We get this checking for free and don't need to write guards against it. The template includes a guard against this macro being called with no arguments and we've left this check in, but it isn't required as the macro can't be called with no arguments.

There are, however, issues the compiler can't check for. We should guard against these.

Adding an Error Type

There are two possible errors I'd like to handle. The first is that I'd prefer not to use a `fatalError()` for an invalid `URL`. Let's throw our own error so we have the option of handling it instead of crashing the app that uses it.

Just so we have an example of a second error, let's test that the `String` is a single `String` literal and throw an error if it isn't.

Add a new Swift file to *URLMacros* named *URLMacroError.swift*. Let's implement it as an enum with the two cases described above.

*URL/Sources/URLMacros/**URLMacroError.swift***

```
enum URLMacroError: Error {
  case argumentMustBeASingleStringLiteral
  case invalidURL
}
```

Let's use this error in our macro implementation.

First, `expansion()` must be marked with `throws` because it may now throw an error. I've removed the body of the method for now. We'll build it up together.

*URL/Sources/URLMacros/**URLMacro.swift***

```
public static func expansion(
  of node: some FreestandingMacroExpansionSyntax,
  in context: some MacroExpansionContext
  ) throws -> ExprSyntax { //    not completed yet
}
```

Next, we know `node.arguments` contains a single element. That's guaranteed to us by the macro definition. Let's confirm that this element is a `StringLiteralExprSyntax`. While we're at it, let's confirm that it has exactly one segment. If it doesn't, we'll throw an error.

*URL/Sources/URLMacros/**URLMacro.swift***

```
public static func expansion(
  of node: some FreestandingMacroExpansionSyntax,
  in context: some MacroExpansionContext
  ) throws -> ExprSyntax {
  guard let stringLiteral
          = node.arguments.first?
    .expression
    .as(StringLiteralExprSyntax.self)?
    .representedLiteralValue
  else {
    throw URLMacroError.argumentMustBeASingleStringLiteral
  }

//    not completed yet
}
```

If we survive the `guard` clause, we know that `stringLiteral` is an instance of `StringLiteralExprSyntax` with one segment.

Next, let's check that `stringLiteral` represents a valid `URL`. If it doesn't, we throw an `invalidURL` error.

*URL/Sources/URLMacros/**URLMacro.swift***

```
public static func expansion(
  of node: some FreestandingMacroExpansionSyntax,
  in context: some MacroExpansionContext
  ) throws -> ExprSyntax {
  guard let stringLiteral
          = node.arguments.first?
    .expression
    .as(StringLiteralExprSyntax.self)?
    .representedLiteralValue
  else {
    throw URLMacroError.argumentMustBeASingleStringLiteral
  }
  guard URL(string: stringLiteral) != nil else {
    throw URLMacroError.invalidURL
  }
//    not completed yet
}
```

If we survive this second `guard` clause then `stringLiteral` produces a valid `URL`. Return a force-unwrapped `URL` created from `stringLiteral` from `expansion()`.

*URL/Sources/URLMacros/**URLMacro.swift***

```
public static func expansion(
  of node: some FreestandingMacroExpansionSyntax,
  in context: some MacroExpansionContext
  ) throws -> ExprSyntax {
  guard let stringLiteral
          = node.arguments.first?
    .expression
    .as(StringLiteralExprSyntax.self)?
    .representedLiteralValue
  else {
    throw URLMacroError.argumentMustBeASingleStringLiteral
  }

  guard URL(string: stringLiteral) != nil else {
    throw URLMacroError.invalidURL
  }

  return "URL(string: \(literal: stringLiteral))!"
}
```

Let's use this implementation in `main` and our tests.

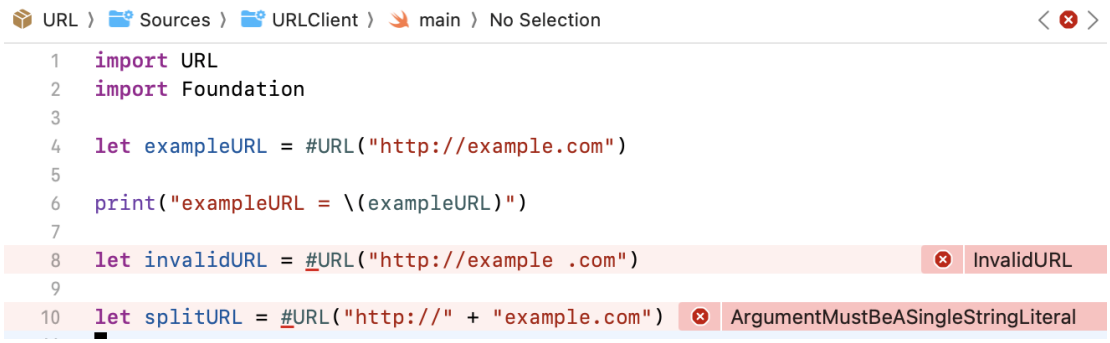
Build Errors

Add an invalid `URL` and a `URL` specified with two `String` literals to *main.swift*.

*URL/Sources/URLMacros/**URLMacro.swift***

```
import URL
import Foundation

let exampleURL = #URL("http://example.com")

print("exampleURL = \(exampleURL)")

let invalidURL = #URL("http://example .com")

let splitURL = #URL("http://" + "example.com")
```

Build and we see more specific and helpful errors.



The `invalidURL` leads to a `URLMacroError.invalidURL` error and the `splitURL` produces a `URLMacroError.argumentMustBeASingleStringLiteral` error.

Delete the lines with errors.

*URL/Sources/URLMacros/**URLMacro.swift***

```
import Foundation

let exampleURL = #URL("http://example.com")

print("exampleURL = \(exampleURL)")

~~let invalidURL = #URL("http://example .com")~~

~~let splitURL = #URL("http://" + "example.com")~~
```

Next, let's work on our tests. One is failing.

Testing errors

Run the unit tests. The first one passes but the second one, the one where we expand an invalid `URL`, fails.

There are two errors reported from running `testInvalidStringInMacro`.

The first error is that our expanded code is not what we expected to see.

```
error: -[URLTests.URLTests testInvalidStringInMacro] : failed -
Actual output (+) differed from expected output (-):

-URL(string: "http://example .com")!
+#URL("http://example .com")
```

The actual output is just the same as our input. The macro isn't expanded at all when `expansion()` throws an error. We'll update the value of `expandedSource` in our test.

The second error is that the failing test is issuing a diagnostic that isn't being handled.

```
error: -[URLTests.URLTests testInvalidStringInMacro] :
failed - Expected 0 diagnostics but received 1:
```

Add a parameter for `diagnostics` that confirms we are receiving a `URLMacroError.invalidURL` error.

*URL/Tests/URLTests/**URLTests.swift***

```
  func testInvalidStringInMacro() throws {
#if canImport(URLMacros)
    assertMacroExpansion(
          """
          #URL("http://example .com")
          """,
          expandedSource: """
          #URL("http://example .com")
          """,
          diagnostics: [DiagnosticSpec(message: "invalidURL",
                                       line: 1,
                                       column: 1,
                                       severity: .error)],
          macros: testMacros
    )
#else
    throw XCTSkip("macros are only supported on the host platform")
#endif
  }
```

Run the tests and both tests pass.

If you want a nicer error message we could conform `URLMacroError` to `CustomStringConvertible` and provide a nice `description`. We could even use an associated value for each of our cases to display the actual `String` in our error message.

We'll leave `URLMacroError` as it is.

Take a moment and write a test for a split `URL` with a test named `testMultiSegmentStringLiteral()`.

A possible solution

*URL/Tests/URLTests/**URLTests.swift***

```
  func testMultiSegmentStringLiteral() throws {
#if canImport(URLMacros)
    assertMacroExpansion(
          """
          #URL("http://" + "example.com")
          """,
          expandedSource: """
            #URL("http://" + "example.com")
            """,
          diagnostics:
            [DiagnosticSpec(message: "argumentMustBeASingleStringLiteral",
                            line: 1,
                            column: 1,
                            severity: .error)],
          macros: testMacros
    )
#else
    throw XCTSkip("macros are only supported on the host platform")
#endif
  }
```

Run the tests and all three tests pass. We now have a macro that throws errors for conditions that can't be enforced by the compiler. We also have tests that can ensure that these errors are thrown.

We'll return to diagnostics later in this chapter to see how we can explicitly create and use them.

Let's look at using the macros we create. In the next section we use `UnwrapOrDie` in a project and in the section after that we use it in the implementation of another macro.