Microsoft® “Roslyn”

How To: Write a Visual Basic Quick Fix

September 2012

# Prerequisites

* [Roslyn Project Overview](http://go.microsoft.com/fwlink/?LinkId=230702)

# Introduction

In previous releases of Visual Studio, it has always been difficult to create custom refactorings that target C# or Visual Basic. With the Project Roslyn Services APIs, this once difficult task has become easy! All that is needed is to perform a bit of analysis to identify an issue, and optionally provide a tree transformation as a code fix. The heavy lifting of running your analysis on a background thread, showing squiggly underlines in the editor, populating the Visual Studio Error List, creating “light bulb” suggestions and showing rich previews is all done for you automatically.

In this walkthrough, we’ll explore the creation of a Code Issue and an accompanying Code Action using the Roslyn Services APIs. A Code Issue is a way to perform source code analysis and report problems to the user. Optionally, a Code Issue can also provide a Code Action which represents a modification to the user’s source code. You can think of the two together as finding a source code problem and providing a fix. For example, a Code Issue could be created to detect and report any local variable names that begin with an uppercase letter, and provide a Code Action that corrects them.

# Writing the Code Issue

Suppose that you wanted to report to the user any local variable declarations that can be converted to local constants. For example, consider the following code:

Dim x As Integer = 0

Console.WriteLine(x)

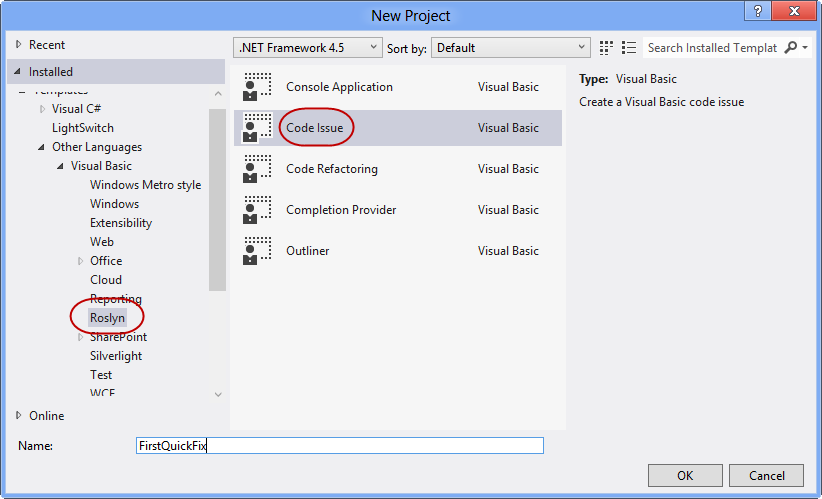
In the code above, x is assigned a constant value and is never written to. Thus, it can be declared using the Const modifier:

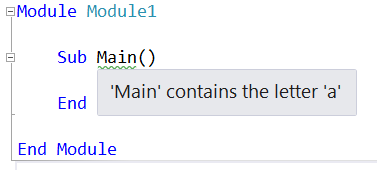
Const x As Integer = 0

Console.WriteLine(x)

The analysis to determine whether a variable can be made constant is actually fairly involved, requiring syntactic analysis, constant analysis of the initializer expression and dataflow analysis to ensure that the variable is never written to. However, performing this analysis with the Roslyn Compiler APIs and exposing it as a Code Issue is pretty easy.

1. First you’ll create a new Visual Basic Code Issue project.
   * In Visual Studio, choose File -> New -> Project… to display the New Project dialog.
   * Under Visual Basic -> Roslyn, choose “Code Issue.”
   * Name your project “FirstQuickFix” and click OK.



1. Press F5 to run the newly created Code Issue project in a new instance of Visual Studio with the Roslyn Language Service loaded. This new instance of Visual Studio is called the debuggee.
   * In the debuggee Visual Studio instance, create a new Visual Basic Console Application project. Hover over one of the tokens in the code with a wavy underline and the warning text provided by a code issue appears. These code issues are provided by the GetIssues method in the debugger project. So initially, the debugger project contains enough code to create a Code Issue for every token in a Visual Basic file that contains the letter ‘a’.  
       
     
   * Now that you’ve seen the initial Code Issue in action, close the debuggee Visual Studio instance and return to your Code Issue project.
2. Take a moment to familiarize yourself with the Code Issue Provider in the CodeIssueProvider.vb file of your project. There are two important aspects to draw your attention to:
   * Every ICodeIssueProvider implementer must provide an ExportCodeIssueProvider attribute that describes important details, such as the name of the provider and the language it operates on.
   * Every Code Issue Provider must implement the ICodeIssueProvider interface. This interface contains two overloaded methods called GetIssues, one for operating on syntax nodes and one for operating on syntax tokens. In addition, ICodeIssueProvider contains two properties, SyntaxNodeTypes and SyntaxTokenKinds, which indicate what sort of syntax nodes or tokens the ICodeIssueProvider implementer supports. Because this ICodeIssueProvider only operates on syntax nodes, it returns null from SyntaxTokenKinds and implements the GetIssues method for syntax nodes. The other GetIssues method throws a NotImplementedException.

**Note**: For a full treatment of syntax nodes and tokens please refer to the [Roslyn Project Overview](http://go.microsoft.com/fwlink/?LinkId=230702).

1. To restrict the Code Issue Provider to only operate on local variables, update the existing Code Issue Provider to operate on LocalDeclarationStatementSyntax nodes:
   * In SyntaxNodeTypes, change GetType(SyntaxNode) to GetType(LocalDeclarationStatementSyntax).
   * Remove the source code from inside the GetIssues method that operates on syntax nodes.
   * When you’re finished, the code in CodeIssueProvider.vb should look like the following code.

Imports System.Threading

Imports Roslyn.Compilers

Imports Roslyn.Compilers.Common

Imports Roslyn.Compilers.VisualBasic

Imports Roslyn.Services

Imports Roslyn.Services.Editor

<ExportCodeIssueProvider("FirstQuickFix", LanguageNames.VisualBasic)>

Public Class CodeIssueProvider

Implements ICodeIssueProvider

Public Function GetIssues(document As IDocument, node As CommonSyntaxNode,

Optional cancellationToken As CancellationToken = Nothing) \_

As IEnumerable(Of CodeIssue) Implements ICodeIssueProvider.GetIssues

End Function

Public ReadOnly Iterator Property SyntaxNodeTypes As IEnumerable(Of Type) Implements \_

ISyntaxMetadata.SyntaxNodeTypes

Get

Yield GetType(LocalDeclarationStatementSyntax)

End Get

End Property

#Region "Unimplemented ICodeIssueProvider members"

Public Function GetIssues(document As IDocument, token As CommonSyntaxToken,

Optional cancellationToken As CancellationToken = Nothing) \_

As IEnumerable(Of CodeIssue) Implements ICodeIssueProvider.GetIssues

Throw New NotImplementedException()

End Function

Public ReadOnly Property SyntaxTokenKinds As IEnumerable(Of Integer) Implements \_

ISyntaxMetadata.SyntaxTokenKinds

Get

Return Nothing

End Get

End Property

#End Region

End Class

* + Now you’re ready to write the logic to determine whether a local variable can be declared as a Const in the GetIssues method.

1. First, you’ll need to perform the necessary syntactic analysis.
   * In the GetIssues method that operates on syntax nodes, cast the node passed in to a LocalDeclarationStatementSyntax type. You can safely assume this cast will succeed because the SyntaxNodeTypes property declares that your Code Issue Provider only operates on syntax nodes of that type.

Dim localDeclaration = CType(node, LocalDeclarationStatementSyntax)

* + Ensure that the local variable declaration only has a Dim modifier.

' Only consider local variable declarations that are Dim (no Static or Const).

If Not localDeclaration.Modifiers.All(Function(m) m.Kind = SyntaxKind.DimKeyword) Then

Return Nothing

End If

1. Next, you’ll perform the semantic analysis necessary to determine whether the local variable declaration can be made Const.
   * Retrieve an ISemanticModel for the IDocument that the Code Issue Provider is operating on.

**Note**: An IDocument is conceptually equivalent to a file in Visual Studio, and an ISemanticModel is a representation of all semantic information in a single source file. Please see the [Roslyn Project Overview](http://go.microsoft.com/fwlink/?LinkId=230702) for a more detailed treatment of these concepts.

Dim semanticModel = document.GetSemanticModel(cancellationToken)

* + Next, ensure that every variable in the declaration has an initializer. This is necessary to match the Visual Basic specification which states that all Const variables must be initialized. For example, Dim x As Integer = 0, y As Integer = 1 can be made Const, but Dim x As Integer, y As Integer = 1 cannot. Additionally, use the ISemanticModel to ensure that each variable’s initializer is a compile-time constant. You’ll do this by calling ISemanticModel.GetConstantValue() for each variable’s initializer and checking that the returned Optional(Of Object) contains a value.

' Ensure that all variable declarators in the local declaration have

' initializers and a single variable name. Additionally, ensure that

' each variable is assigned with a constant value.

For Each declarator In localDeclaration.Declarators

If declarator.Initializer Is Nothing OrElse declarator.Names.Count <> 1 Then

Return Nothing

End If

If Not semanticModel.GetConstantValue(declarator.Initializer.Value).HasValue Then

Return Nothing

End If

Next

* + Use the ISemanticModel to perform data flow analysis on the local declaration statement. Then, use the results of this data flow analysis to ensure that none of the local variables are written with a new value anywhere else. You’ll do this by calling ISemanticModel.GetDeclaredSymbol to retrieve the ILocalSymbol for each variable and checking that it isn’t contained with the WrittenOutside collection of the data flow analysis.

' Perform data flow analysis on the local declaration.

Dim dataFlowAnalysis = semanticModel.AnalyzeDataFlow(localDeclaration)

' Retrieve the local symbol for each variable in the local declaration

' and ensure that it is not written outside of the data flow analysis region.

For Each declarator In localDeclaration.Declarators

Dim variable = declarator.Names.Single()

Dim variableSymbol = semanticModel.GetDeclaredSymbol(variable)

If dataFlowAnalysis.WrittenOutside.Contains(variableSymbol) Then

Return Nothing

End If

Next

1. With all of the necessary analysis performed, you can return a new CodeIssue that reports a warning for the variable declaration.

Return {New CodeIssue(CodeIssueKind.Warning, localDeclaration.Span,

"Can be made constant")}

At this point, your GetIssues method should look like so:

Public Function GetIssues(document As IDocument, node As CommonSyntaxNode,

Optional cancellationToken As CancellationToken = Nothing) \_

As IEnumerable(Of CodeIssue) Implements ICodeIssueProvider.GetIssues

Dim localDeclaration = CType(node, LocalDeclarationStatementSyntax)

' Only consider local variable declarations that are Dim (no Static or Const).

If Not localDeclaration.Modifiers.All(Function(m) m.Kind = SyntaxKind.DimKeyword) Then

Return Nothing

End If

Dim semanticModel = document.GetSemanticModel(cancellationToken)

' Ensure that all variable declarators in the local declaration have

' initializers and a single variable name. Additionally, ensure that

' each variable is assigned with a constant value.

For Each declarator In localDeclaration.Declarators

If declarator.Initializer Is Nothing OrElse declarator.Names.Count <> 1 Then

Return Nothing

End If

If Not semanticModel.GetConstantValue(declarator.Initializer.Value).HasValue Then

Return Nothing

End If

Next

' Perform data flow analysis on the local declaration.

Dim dataFlowAnalysis = semanticModel.AnalyzeDataFlow(localDeclaration)

' Retrieve the local symbol for each variable in the local declaration

' and ensure that it is not written outside of the data flow analysis region.

For Each declarator In localDeclaration.Declarators

Dim variable = declarator.Names.Single()

Dim variableSymbol = semanticModel.GetDeclaredSymbol(variable)

If dataFlowAnalysis.WrittenOutside.Contains(variableSymbol) Then

Return Nothing

End If

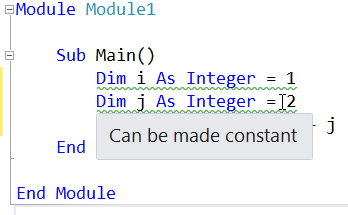
Next

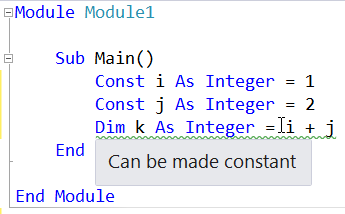
Return {New CodeIssue(CodeIssueKind.Warning, localDeclaration.Span,

"Can be made constant")}

End Function

1. Press F5 to run the Code Issue project in a new instance of Visual Studio with the Roslyn Language Service loaded.
   * In the debuggee Visual Studio instance create a new Visual Basic Console Application project and add a few local variable declarations assigned to constant values in the Main method.  
     You’ll see that the local variables are reported as warnings, as pictured below.



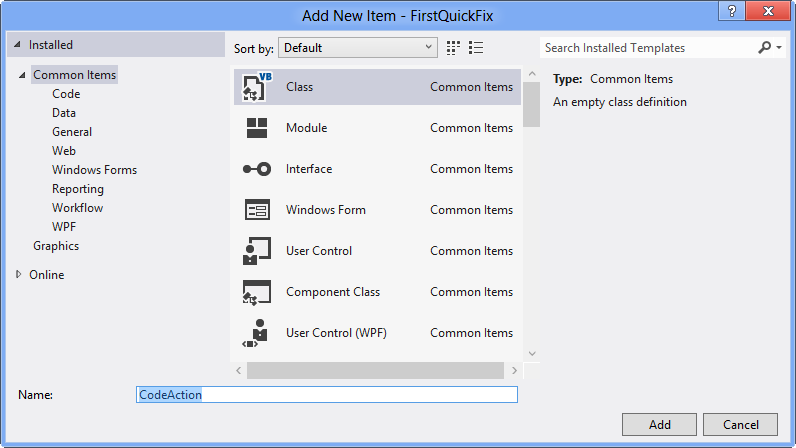
* + Notice that if you type Const before each variable, the warnings are automatically removed. Additionally, changing a variable to Const can affect the reporting of other variables.  
      
    

1. Congratulations! You’ve created your first code issue using the Roslyn APIs to perform non-trivial syntactic and semantic analysis.

# Writing the Code Action

Any Code Issue can provide one or more Code Actions which define an edit that can be performed to the source code to address the reported issue. For the Code Issue that you just created, you can provide a Code Action that inserts the Const keyword when the user chooses it from the suggestion UI in the editor. To do so, follow the steps below.

1. First, add a new Class implementing ICodeAction to your Code Issue project.
   * In the Solution Explorer, right-click on your Code Issue project and choose Add -> Class… from the context menu to display the Add New Item dialog.
   * Name the new file CodeAction and click Add.



1. At the top of the file, add the following using directives:

Imports System.Threading

Imports Roslyn.Compilers

Imports Roslyn.Compilers.VisualBasic

Imports Roslyn.Services

Imports Roslyn.Services.Editor

1. Type “Implements ICodeAction” to implement the ICodeAction interface on your newly created class. After typing “ICodeAction” press ENTER to generate a default implementation of ICodeAction.
2. Change the Description property to return "Make Constant".
3. At this point, your code should look like so:

Imports System.Threading

Imports Roslyn.Compilers

Imports Roslyn.Compilers.VisualBasic

Imports Roslyn.Services

Imports Roslyn.Services.Editor

Public Class CodeAction

Implements ICodeAction

Public ReadOnly Property Description As String Implements ICodeAction.Description

Get

Return "Make Constant"

End Get

End Property

Public Function GetEdit(Optional cancellationToken As CancellationToken = Nothing) \_

As CodeActionEdit Implements ICodeAction.GetEdit

End Function

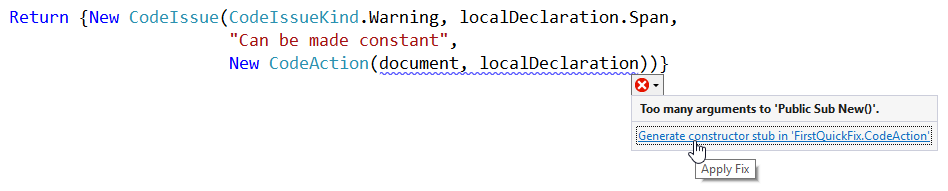
End Class

1. In order, to implement the GetEdit method, you’ll need a couple of other objects.
   * Return to your Code Issue Provider and update the code at the bottom of GetIssues that returns a new CodeIssue to also include a new instance of your CodeAction class, passing document and localDeclaration.

Return {New CodeIssue(CodeIssueKind.Warning, localDeclaration.Span,

"Can be made constant",

New CodeAction(document, localDeclaration))}

* + Invoke the smart tag using CTRL+. on CodeAction to generate a new constructor and fields in the CodeAction class.  
      
    

1. Now, return to your Code Action and implement the GetEdit method.
   * First, create a new Const keyword token that will replace the first token of the local declaration. Be careful to first remove any trivia from the first token of the local declaration and attach it to the Const token.

' Create a const token with the leading trivia from the local declaration.

Dim firstToken = \_localDeclaration.GetFirstToken()

Dim constToken = Syntax.Token(

firstToken.LeadingTrivia, SyntaxKind.ConstKeyword, firstToken.TrailingTrivia)

* + Next, create a new SyntaxTokenList containing the Const token and the existing modifiers of the declaration statement.

' Create a new modifier list with the const token.

Dim newModifiers = Syntax.TokenList(constToken)

* + Create a new local declaration containing the new list of modifiers.

' Produce new local declaration.

Dim newLocalDeclaration = \_localDeclaration.WithModifiers(newModifiers)

* + Add a FormattingAnnotation to the new local declaration, which is an indicator to the code action engine to format any whitespace using the Visual Basic formatting rules.

' Add an annotation to format the new local declaration.

Dim formattedLocalDeclaration =

CodeAnnotations.Formatting.AddAnnotationTo(newLocalDeclaration)

* + Retrieve the root CommonSyntaxNode from the IDocument and use it to replace the old declaration statement with the new one.

' Replace the old local declaration with the new local declaration.

Dim oldRoot = \_document.GetSyntaxRoot(cancellationToken)

Dim newRoot = oldRoot.ReplaceNode(\_localDeclaration, formattedLocalDeclaration)

* + Finally, create a new CodeActionEdit passing a document updated with the tree transformation that you just performed.

' Create and return a new CodeActionEdit for the transformed tree.

Return New CodeActionEdit(\_document.UpdateSyntaxRoot(newRoot))

At this point, your GetEdit method should like so:

Public Function GetEdit(Optional cancellationToken As CancellationToken = Nothing) \_

As ICodeActionEdit Implements ICodeAction.GetEdit

' Create a const token with the leading trivia from the local declaration.

Dim firstToken = \_localDeclaration.GetFirstToken()

Dim constToken = Syntax.Token(

firstToken.LeadingTrivia, SyntaxKind.ConstKeyword, firstToken.TrailingTrivia)

' Create a new modifier list with the const token.

Dim newModifiers = Syntax.TokenList(constToken)

' Produce new local declaration.

Dim newLocalDeclaration = \_localDeclaration.WithModifiers(newModifiers)

' Add an annotation to format the new local declaration.

Dim formattedLocalDeclaration =

CodeAnnotations.Formatting.AddAnnotationTo(newLocalDeclaration)

' Replace the old local declaration with the new local declaration.

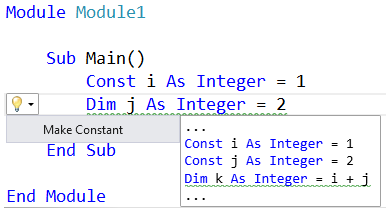
Dim oldRoot = \_document.GetSyntaxRoot(cancellationToken)

Dim newRoot = oldRoot.ReplaceNode(\_localDeclaration, formattedLocalDeclaration)

' Create and return a new CodeActionEdit for the transformed tree.

Return New CodeActionEdit(\_document.UpdateSyntaxRoot(newRoot))

End Function

1. Press F5 to run the Code Issue project in a new instance of Visual Studio with the Roslyn Language Service loaded.
   * In the debuggee Visual Studio instance create a new Visual Basic Console Application project and add a few local variable declarations assigned to constant values in the Main method. You’ll see that they are reported as warnings and “light bulb” suggestions appear next to them when the editor caret is on the same line.
   * Move the editor caret to one of the squiggly underlines and press CTRL+. to display the suggestion. Notice that a preview window appears next to the suggestion menu showing what the code will look like after the Code Action is invoked. Your Code Issue is even run in the preview window, which shows that a different variable can be made Const once the Code Action is applied.  
       
     
2. Congratulations! You’ve created your first Roslyn extension that performs on-the-fly code analysis to detect an issue and provides a code fix to correct it.