Microsoft® “Roslyn”

Getting Started: Workspace

June 2012

# Prerequisites

* [Roslyn Project Overview](http://go.microsoft.com/fwlink/?LinkID=230702)
* Getting Started: Syntax Analysis
* Getting Started: Semantic Analysis

# Introduction

This walkthrough builds on concepts and techniques introduced in the **Roslyn Project Overview** document, Section 5 – Working with a Workspace. Additionally, it is advised to complete the **Getting Started: Syntax Analysis** and **Getting Started: Semantic Analysis** walkthroughs to familiarize yourself with the concepts of syntax trees, compilations and immutability.

In this walkthrough we’ll explore the **Roslyn Workspace APIs**. The purpose of the workspace is to:

* Provide language-agnostic and immutable data structures to represent solutions, projects and documents.
* Connect to the outside environment called a Host to populate and edit these data structures.
* Manage and expose Compilations for projects.
* Manage and expose SyntaxTrees and SemanticModels for documents.
* Enable and connect the Roslyn services and editor features.

# Understanding Workspaces

A compilation is an in-memory abstraction of a single compiled project. With it you can access the compiler’s semantic understanding of the source code; which identifiers are bound to which symbols, the static type of expressions and all the errors the compiler has reported.

However, you may need to work with multiple projects at once. For instance, many Visual Studio IDE features such as Find All References need to be aware of all the C# and VB projects in the solution. The Roslyn.Services namespace offers an API that models entire solutions. It also may be a lot more convenient to use the workspace/solution API even if you have only one project, since you can use the workspace to access a project defined in a VS project or solution file.

The types ISolution, IProject and IDocument together represent all the pieces necessary to produce compilations for all related projects. ISolution has a set of projects and IProject has a set of documents. From an IProject you can also directly access the project’s corresponding compilation, and from an IDocument you can get directly at the text, syntax tree and semantic model corresponding to a source file. All Roslyn Services features operate in terms of solutions, projects and documents.

Each of these types is an immutable data structure, meaning they are read-only, and can never be modified by anyone. Once constructed, an ISolution is a snapshot of a single point in time of the state of your source code. This gives you the ability to operate over the model without the fear of having the rug pulled out from under you when something changes. Yet sometimes, you do want to make changes and observe when changes are made by others.

A workspace is a component that maintains an always up-to-date model of the entire solution, signaling you via an event every time it is changed. You can either get at the most recent solution from the event itself or directly from the workspace’s CurrentSolution property.

To get back to the same project or document you were examining before a change occurred, each project or document has a unique ID that will let you find the new version within the new solution instance. You can use ISolution.GetProject(ProjectId) and ISolution.GetDocument(DocumentId) to retrieve the new IProject or IDocument instances.

Within a host environment like Visual Studio, a workspace is provided corresponding to the open solution. You can access this workspace from the Workspace.PrimaryWorkspace property. Changes in the environment caused by typing in an open document window, adding or removing documents in the Solution Explorer or side effects of other IDE features are automatically reflected in changes to the workspace’s current solution. Services listening to workspace events can then react to these changes, communicating results and changes back to the host environment, colorizing text, triggering IntelliSense, changing text in the editor, or adding files to projects.

Outside of a host like Visual Studio, a workspace can be constructed directly by loading a solution or project file. This is useful for building command line tools, such as static analysis (e.g. a StyleCop-like tool), or a task that formats all documents in a solution without opening Visual Studio.

## Example – Sorting all usings in a solution

We’re going to build a console application that populates a workspace from a solution on disk, and changes every \*.cs and \*.vb file in the solution to have their using directives and Imports statements sorted alphabetically. The changes will be written back to disk.

First, we’ll need a test solution that we’re going to operate on. If you have a solution that you can experiment on, you can skip this step. Be careful, changes made by this sample cannot be undone, as it overwrites files on disk.

### To create a sample solution:

1. Create a new C# Console Application (you can call it SampleCode1)
2. In Program.cs, change the order of the using directives to be not alphabetical:

using System.Linq;

using System.Text;

using System;

using System.Threading.Tasks;

using System.Collections.Generic;

1. Add a new Visual Basic Console Application to the solution
2. In Module1.vb, add some unsorted Imports statements, such as:

Imports System.Xml.Linq

Imports System.Collections.Generic

Imports Microsoft.VisualBasic

Imports System

### Creating the actual application that will sort usings

Create a new C# Roslyn Console Application project:

1. In Visual Studio, choose File -> New -> Project… to display the New Project dialog.
2. Under Visual C# -> Roslyn, choose “Console Application”.
3. Name your project “**OrganizeUsingsCS**” and click OK.

Most of the Workspace API is in the Roslyn.Services namespace:

using Roslyn.Services;

We’ll add the rest of the code inside the Main method created by the template.

First, we create a workspace instance by calling the static LoadSolution() method on the Roslyn.Services.Workspace class:

var workspace = Workspace.LoadSolution(@"C:\Users\USERNAME\My Documents\Visual Studio 2010\Projects\SampleCode1\SampleCode1.sln");

As an argument we pass the full file path to the .sln file. The workspace then reads the source code from disk and creates an instance of ISolution available through the CurrentSolution property. Substitute the path to your test solution here.

Further work consists of two parts: iterating over all the documents in the solution and accumulating changes to each document in a new intermediate ISolution snapshot to apply them in one fell swoop at the end. We’ll need a reference to the original ISolution snapshot to iterate over, as well as a reference to the forked ISolution with the changes applied at each step.

// Take a snapshot of the original solution.

var originalSolution = workspace.CurrentSolution;

Initially, we set the newSolution to point to the originalSolution:

// Declare a variable to store the intermediate solution snapshot at each step.

ISolution newSolution = originalSolution;

The outer foreach loop will iterate over all projects in the original solution:

foreach (var project in originalSolution.Projects)

{

The inner foreach loop will iterate over each document in the current project. There is one fine detail here: during each iteration of the loop we update the newSolution to contain the most up-to-date accumulated snapshot. Therefore, we can’t iterate over the Documents collection of the current project because that would give us instances of IDocument from the originalSolution, not the newSolution. However, we can rely on the fact that the DocumentId of a document is preserved across solution snapshots and iterate over project.DocumentIds instead:

// Note how we can't simply iterate over project.Documents because it will return

// IDocument objects from the originalSolution, not from the newSolution. We need to

// use the DocumentId (that doesn't change) to look up the corresponding snapshot of

// the document in the newSolution.

foreach (var documentId in project.DocumentIds)

{

Given the DocumentId we look up the IDocument in the current newSolution snapshot.

// Look up the snapshot for the original document in the latest forked solution.

var document = newSolution.GetDocument(documentId);

**Note:** there is a small discrepancy here. The inner foreach loop iterates over the DocumentIds in each project, but the outer foreach loop doesn’t iterate over ProjectIds in the originalSolution – it iterates directly over Projects. Every project we inspect comes from the originalSolution, not from the newSolution. There is a different way:

foreach (var projectId in originalSolution.ProjectIds)

{  
 // Look up the snapshot for the original project in the latest forked solution.  
 var project = newSolution.GetProject(projectId);

foreach (var documentId in project.DocumentIds)

{

This way, every project we inspect comes from the most up-to-date solution snapshot available. However since our sample doesn’t really change the list of documents in a project, it doesn’t matter which project snapshot to use to retrieve a list of documentIds – both the original project snapshot and the modified project snapshot contains the same list of document Ids.

### Modifying the workspace by creating new ISolution, IProject and IDocument objects

Now that we have a document, we’d like to modify it by sorting the using directives (if it’s a C# document) or Imports statements (if it’s a Visual Basic document). The Workspace API is usually language agnostic and is the same for C# and VB projects and documents.

Every modification of the workspace data structures implies the creation of a new ISolution object (“forked” solution). You can’t change an IDocument and keep it in the same IProject, because that would break the immutability of IProject – once an ISolution, IProject or IDocument object is created, it or any of the children can never change again. Hence we have to create a new IProject snapshot to host the modified IDocument, and a new ISolution snapshot to host the modified IProject - all the way up.

**Note:** When a new solution snapshot is created (a solution is “forked”), the operation is very efficient – it reuses as much data as possible from the old solution. We rely on the fact that the old solution is immutable so we can be sure it never changes again. In fact, when the Roslyn workspace is hosted inside Visual Studio, every time you type a character or do any other change, a new ISolution is created, and this operation is optimized to be frugal and efficient.

There are various APIs to simplify modifying code and creating new ISolution snapshots. In our case the IDocument class has the convenient OrganizeImports() extension method. This method returns an instance of the DocumentTransformation class, which contains the description of the changes that need to be applied to the document to sort its usings.

// Get the object that describes the changes (nothing is modified yet).

var transformation = document.OrganizeImports();

DocumentTransformation also contains the GetUpdatedDocument() method that returns a new IDocument instance (with the usings sorted).

// Get a transformed version of the document (a new solution snapshot is created

// under the covers to contain it - none of the existing objects are modified).

var newDocument = transformation.GetUpdatedDocument();

Under the covers, GetUpdatedDocument() creates a new ISolution snapshot that reuses most of the information from the original ISolution, but contains a new IProject snapshot and a new IDocument snapshot. The ProjectId and DocumentId of these instances remain the same however. From the new IDocument snapshot we reference its IProject and ISolution. We then store this new ISolution as the latest state in the newSolution variable.

// Store the solution implicitly constructed in the previous step as the latest

// one so we can continue building it up in the next iteration.

newSolution = newDocument.Project.Solution;

}

}

Note that because of the immutability of ISolution, you can still access the document variable which contains the original text of the document (if you need to) – it didn’t change when we applied the transformation.

Finally, although we did create the new snapshot, the workspace still doesn’t know anything about it. The snapshot is only known to us and doesn’t interfere with any other state because it doesn’t have side effects. We call workspace.ApplyChanges to apply the changes back to the Workspace. Insert this code after the foreach loops (we only want to call this once at the very end):

// Actually apply the accumulated changes and save them to disk. At this point

// workspace.CurrentSolution is updated to point to the new solution.

workspace.ApplyChanges(originalSolution, newSolution);

This call updates the Workspace.CurrentSolution property to point to the new ISolution and physically saves the changes to disk. The ApplyChanges method is optimized to find minimal differences between the old and the new ISolution snapshots, so it is efficient to do call it.

Again, because of the immutability of the data structures involved, we can still access the originalSolution and inspect its state if we need to – it didn’t change when we called ApplyChanges.

The final text of the program should look like this:

using Roslyn.Services;

// This program will sort the global using directives in C# source files or

// the global import statements in Visual Basic source files for an entire solution.

internal class Program

{

private static void Main(string[] args)

{

// The test solution is copied to the output directory when you build this sample.

var workspace = Workspace.LoadSolution(@"C:\Users\USERNAME\My Documents\Visual Studio 2010\Projects\SampleCode1\SampleCode1.sln");

// Take a snapshot of the original solution.

var originalSolution = workspace.CurrentSolution;

// Declare a variable to store the intermediate solution snapshot at each step.

ISolution newSolution = originalSolution;

foreach (var project in originalSolution.Projects)

{

// Note how we can't simply iterate over project.Documents because it will return

// IDocument objects from the originalSolution, not from the newSolution. We need to

// use the DocumentId (that doesn't change) to look up the corresponding snapshot of

// the document in the newSolution.

foreach (var documentId in project.DocumentIds)

{

// Look up the snapshot for the original document in the latest forked solution.

var document = newSolution.GetDocument(documentId);

// Get the object that describes the changes (nothing is modified yet).

var transformation = document.OrganizeImports();

// Get a transformed version of the document (a new solution snapshot is created

// under the covers to contain it - none of the existing objects are modified).

var newDocument = transformation.GetUpdatedDocument();

// Store the solution implicitly constructed in the previous step as the latest

// one so we can continue building it up in the next iteration.

newSolution = newDocument.Project.Solution;

}

}

// Actually apply the accumulated changes and save them to disk. At this point

// workspace.CurrentSolution is updated to point to the new solution.

workspace.ApplyChanges(originalSolution, newSolution);

}

}

You can now press F5 to run the program. After it is finished, you can open your sample solution (or reload it if you still had it open in another Visual Studio instance). You will see that the using directives and Imports statements are now sorted.

## Implementing your own Document-to-Document transformations

If we didn’t already have the OrganizeImports() extension method on IDocument, we’d need to implement our own document-to-document transformation, that does all the work of creating a new forked ISolution with the required changes.

Here is how the body of the inner foreach loop would look like if we had to implement the transformation ourselves:

// Look up the snapshot for the original document in the latest forked solution.

IDocument document = newSolution.GetDocument(documentId);

// Transform the syntax tree of the document and get the root of the new tree

CommonSyntaxNode newRoot = TransformSyntaxRoot(document);

// Get a forked solution snapshot where the document's syntax tree is replaced

// with a new tree rooted at newRoot node

newSolution = newSolution.UpdateDocument(document.Id, newRoot);

We can call the ISolution.UpdateDocument overload that accepts a DocumentId and a CommonSyntaxNode. This method returns a new forked solution with the syntax tree (and text) for the document replaced.

Note that nowhere above so far we’ve concerned ourselves with the language of the code (C# or Visual Basic). However the actual work to do the syntax transformation is highly language-dependent.

This is how the TransformSyntaxRoot method might look:

private static CommonSyntaxNode TransformSyntaxRoot(IDocument document)

{

var originalRoot = document.GetSyntaxRoot();

switch (document.LanguageServices.Language)

{

case LanguageNames.CSharp:

return TransformRootCSharp(

(Roslyn.Compilers.CSharp.SyntaxNode)originalRoot);

case LanguageNames.VisualBasic:

return TransformRootVisualBasic(

(Roslyn.Compilers.VisualBasic.SyntaxNode)originalRoot);

}

return originalRoot;

}

We determine the language of the document using the Language property of type string on the ILanguageServiceProvider interface. Before we pass the original root node to the language-specific transformation code, we cast it to the language-specific interface SyntaxNode (either in Roslyn.Compilers.CSharp or Roslyn.Compilers.VisualBasic namespace).