Developer Productivity in Visual Studio 2015

# Setup

* Start Windows Magnifier
* Open Expenses.
  + Put breakpoint in GetEmployeeAsync method in RepositoryService
  + Put breakpoint in ShowChargesAsync method in MainWindowViewModel

# Developer Productivity

Start with WebApplication1 open.

## Sign in with multiple accounts

* Account settings
* Add an account. Add Comcast account
* Select filter for your MSA. You can filter to a particular VSO project you belong to.

## Touch

* Open AccountController.cs.
* Zoom in and out. Scroll up and down.

## Windows Layouts

* Modify default.
* Create work, home, code.

## Enhanced Scrollbar

1. Open AccountController.cs in the code editor
2. Right-click the scrollbar and select Scroll Bar Options.
3. Show each of the options.
4. Type some code in any method. Notice yellow and green show up on the scrollbar.
5. Scroll to the top and comment out using System.Web.MVC. Notice red shows up on the scrollbar.

## Code-focused IDE Experience

1. Note that some of the using statements are faded out. This means that they are currently unused.
2. There are now Light Bulbs that expose quick actions that you can take depending up the selected context within the editor window. These Light Bulbs are the new home for all quick actions you take in the editor, including fixes to common code issues and refactoring code.
3. Place the cursor on one of the faded out using statements and then click on the Light Bulb to remove unnecessary using statements from the file.
4. Navigate to the ExternalLogin method within the AccountController.cs code file.
5. Highlight the entirety of the second parameter to the ChallengeResult constructor and then press Ctrl + <period>.
6. Note that the Light Bulb on the left-hand side expands and shows options to introduce a local and extract a method. Also note that there is a preview window showing the diff.
7. Select the first option to introduce a local for this code block.
8. The new Rename window is shown with the new local variable highlighted. Type “redirectUri” to rename the local variable.
9. Click the Apply button (or press Enter) in the Rename window.
10. Highlight the “ExternalLoginCallback” string including the quotes and then bring up the Quick Actions once again (press Ctrl + <period>).
11. Select the option to introduce a constant for all occurrences.
12. Rename the new constant something meaningful such as “ExternalLoginCallbackConstant”. Imagine that after refactoring this code file, you plan to move any new string constants to a different location, but we will not do this right now.

Note: All refactoring types available in C# are now also available in Visual Basic for the first time in 2015.

1. Highlight the second parameter to the ExternalLogin method, right-click within the editor, and then select Rename.
2. Attempt to rename the parameter “provider”, and note that a conflict with the first parameter was detected, saving us from introducing an error.
3. Press Escape to exit the rename operation.
4. Hold the mouse cursor over the call to Url.Action within the ExternalLogin method. Note that the parameter help is now colorized.
5. Collapse the ExternalLogin method in the code editor window and hover the mouse cursor over the collapsed code block (shown as a small box with ellipses) and note that the tooltip now displays the code in full color and even shows the full method declaration.

## Solution Navigation

1. Load Expenses.sln solution file from the Samples folder on the Desktop.
2. Expand ExpenseService.svc.cs.
3. Expand ExpenseService class to see methods.
4. Right-click and view options.
5. Double-click on method and go to it.
6. Type charge in search block.
7. Type charges in search block.
8. Type getcharge in search block.
9. Point out icons for methods, classes, interfaces, etc.
10. Right-click WCF project and select Scope to This.
11. Click Back and Forward to move back and forth.
12. Right-click WCF project and select New Solution Explorer View.
13. Click Back in first Solution Explorer.
14. Open any code file.
15. Click each method and the code editor goes to that method.
16. Click anywhere else in the Solution Explorer.
17. Click Sync with Active Document.
18. Select CreateNewCharge method in ExpenseService.svs.cs.
19. Click Show on Code Map.

## C# Interactive Window

1. Select View | Other Windows | C# Interactive.
2. Close the window.
3. Close the solution.
4. Type C# Inter in the Quick Launch window.
5. Enter **2 + 2**
6. Enter **Console.Write(“I <3 C#”)**
7. Use window buttons to scroll through statements, clear and reset window.



1. Show the C# Interactive Walkthrough.
2. Enter **using System.IO;**
3. Enter **using System.Net;**
4. Enter **var url = "http://ichart.finance.yahoo.com/table.csv?s=MSFT&d=10&e=10&f=2015&g=d&a=2&b=13&c=1986&ignore=.csv";**
5. Enter **Console.Write("url: " + url)**
6. Enter

**var request = WebRequest.Create(url);**

**var response = request.GetResponse();**

**var dataStream = response.GetResponseStream();**

**var reader = new StreamReader(dataStream);**

**var csv = await reader.ReadToEndAsync();**

**reader.Close();**

**dataStream.Close();**

**response.Close();**

1. Enter **csv.Length**
2. Enter **csv.Split('\n').Length**
3. Enter **Console.Write(csv.Substring(0,200))**
4. Enter

**var prices = csv.Split('\n').Skip(1)**

**.Select(line => line.Split(','))**

**.Where(values => values.Length == 7)**

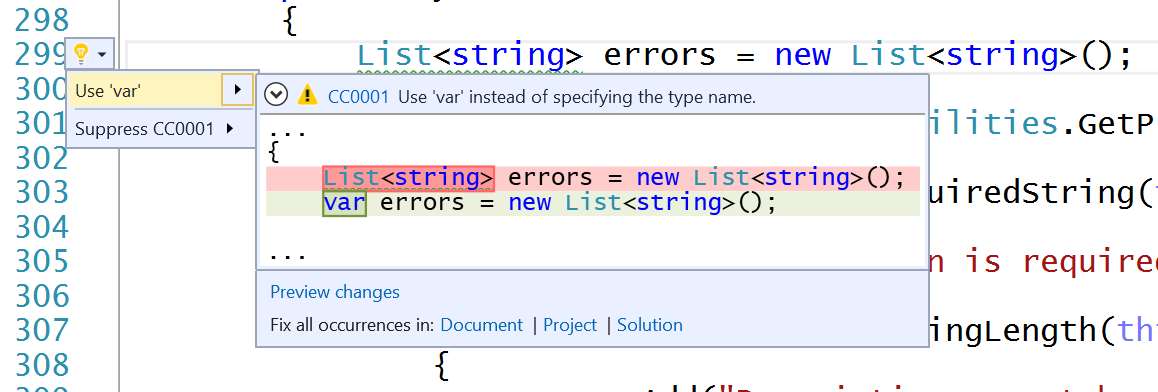
**.Select(values => new { date = DateTime.Parse(values[0]),**

**price = float.Parse(values[6]) });**

1. Enter **prices.Where(s=>s.price > 50).Count()**
2. Load MyAmazingCalculator solution.
3. Right-click solution and select Open Solution in Windows Explorer. Copy the path.
4. Enter **#r "C:\Users\rogreen\Documents\visual studio 2015\Projects\MyAmazingCalculator\MyAmazingCalculator\bin\Debug\MyAmazingCalculator.dll"**
5. Enter **var calc = new MyAmazingCalculator.Calculator();**
6. Enter **Console.Write(calc.Add(4,5))**
7. Go to Developer Command Prompt.
8. Enter csi.
9. Enter **#r "C:\Users\rogreen\Documents\visual studio 2015\Projects\MyAmazingCalculator\MyAmazingCalculator\bin\Debug\MyAmazingCalculator.dll"**
10. Enter **var calc = new MyAmazingCalculator.Calculator();**
11. Enter **Console.Write(calc.Add(4,5))**

## Live Code Analysis and Error List for .NET

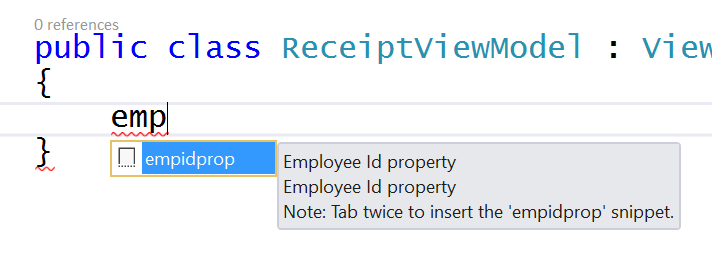
1. Load Expenses.sln solution file from the Samples folder on the Desktop.
2. Select Build | Run Code Analysis on Solution (or press Alt + F11).
3. Open the Error List window and note that there are a few Warnings shown. These represent the normal code analysis messages that you may be used to seeing, although the Error List window has been improved in Visual Studio 2015.
4. Try clicking on the code link to view documentation in a browser window. Close the browser window when done.
5. Some columns such as Code, Project, and File can be sorted and filtered by hovering the mouse cursor over them.
6. Right-click on the top of the grid that contains the column names and note that you can also show or hide columns as you see fit.
7. You can now also get live code analysis and automatic fixes as you type, with specific code-aware guidance for the Microsoft platforms and NuGet packages that you're targeting.
8. Tools | NuGet Package Manager | Manage NuGet Packages for Solution.
9. Search for codecracker.
10. Install codecracker.CSharp.
11. Expand the Analyzers node in the Expenses.WPF project to see the CodeCracker analyzers that were added to the project.
12. Expand the CodeCracker.CSharp node to see all of the individual rules that are defined, for example, use ‘var’ whenever possible and use autoproperty.
13. This is a live code analyzer, so these things will show up as errors or warnings the same way that the lightbulbs did earlier. So you will see errors and warnings as you type, but also if you build. Build the solution.
14. Look at the Error List.
15. Double click use ‘var’. You go to that line of code in that file. Notice a green squiggly and a lightbulb.



1. Change the line of code to use var. The squiggly goes away.
2. Change it back and the squiggly and lightbulb reappear.
3. Notice that all the rules have an icon next to them indicating the severity.
4. The var rule has a severity of warning.
5. Change the severity to error. The icon changes.
6. After a short time, a red squiggly appears.
7. Change it back.
8. You can right click on any rule and View Help.

## Code Snippets

1. Open ChargeViewMode.cs.
2. Locate the EmployeeId property. I write a lot of apps that have classes with EmployeeIds. I would like to make this code reusable. I would like to make a code snippet out of it. There are a number of code snippet editing tools out there, but there is also a very easy way to create your own snippets.
3. Create a solution level folder named Code Snippets.
4. Add an XML file to it named empidprop.snippet.
5. What goes in here? Let’s look at one of the existing snippets that ships with Visual Studio.
6. Select Tools | Code Snippet Manager.
7. Set the Language to CSharp. Copy the Location.
8. Open Windows Explorer and paste the Location.
9. Open if.snippet in Notepad.
10. Copy the XML.
11. Return to Visual Studio. Paste the XML into empidprop.snippet.
12. Change the Title and Description to **Employee Id property**.
13. Change the Shortcut to **empidprop**.
14. Delete the SurroundsWith SnippetType.
15. Return to ChargeViewMode.cs.
16. Copy the EmployeeId property.
17. Return to the code snippet.
18. Paste the property between the CDATA brackets.
19. Save the snippet.
20. Right-click the solution and select Open Folder in Windows Explorer.
21. Copy the path.
22. Select Tools | Code Snippet Manager.
23. Set the Language to CSharp.
24. Select Import.
25. Paste the folder path.
26. Select empidprop.snippet and click Open.
27. Select My Code Snippets and click Finish.
28. In the Code Snippet Manager, expand My Code Snippets and notice VS copied the snippet to this folder on your disk. It is now available in all projects, not just this one.
29. Create a new view model named ReceiptViewModel.
30. Modify the class definition so it is public and inherits ViewModelBase.
31. Add a folder named Model.
32. Create a class named Employee.
33. Type **emp** and the snippet appears.



1. Press tab twice to create the property.
2. Optional: fix the errors.

# Debugging, Diagnostics, Testing

## Diagnostic Tools Window and PerfTips

1. Press F5 to launch a debugging session. This will launch the WCF service and WPF client application.
2. Note the new Diagnostic Tools window. In Visual Studio 2015, diagnostic tools such as the Output window, IntelliTrace, CPU Usage, and Memory Usage are being brought closer together into this single tool window. This new experience allows you to monitor performance while debugging and to correlate that performance data with debugging activity. You will now see this Diagnostic Tools window appear by default when debugging a supported project type.

Note: If the Diagnostic Tools window is not visible, select Debug | Show Diagnostic Tools.

1. The diagnostic tools in this window surface information in two complementary ways: by adding graphs to the timeline in the upper half of the window, and by providing detailed information in the tabs in the bottom half of the window. Diagnostic tools use graphs to surface at-a-glance information, and when you see something of interest you can even select a range of time to filter the detailed information in the tabs below. Note that the there are multiple tracks shown in the Debugger Events section representing Break events, IntelliTrace events, and Output window events.
2. The Debugger break event track at the top of the timeline provides a visual history of events that stopped or resumed the execution of the application. This includes breakpoints being hit, stepping through code, clicking the Break All (pause) button, breaking exceptions, and Debugger.Break() statements in code. The rectangles indicate the amount of time the app was running in between break events, and color of the rectangles indicates whether the app was paused due to a breakpoint being hit (indicated by red), for a step running to completion (indicated by yellow), or for Break All (indicated by blue).
3. So far, we just have a single red rectangle that corresponds to the breakpoint that we just hit in the GetEmployeeAsync method. Hold the mouse cursor over the red rectangle and note that the elapsed duration since the start of the WPF application is shown.
4. Click on the red rectangle break event and note that it is selected along with the event from the Debugger tab in the lower half of the window.
5. Press F10 to take a step in the debugger and note that the start time of the selection in the timeline stays the same, and the end of the selection will expand to include the last step. In this case, a yellow rectangle should be shown.

Note: If you can’t see the yellow rectangle, it may be because the previous step didn’t take very long to execute and is therefore insignificant with respect to the rest of the timeline so far.

1. In the code editor, place a breakpoint on the opening brace of the “using” code block, before the line of code that returns the Employee object that is retrieved from the WCF service.
2. Press F5 to continue execution to the next breakpoint.
3. In the Diagnostic Tools window, note that the time selection automatically updated to start at the time you pressed F5 and end at the current breakpoint. This helps keep you in sync with changes since your last debugging actions by filtering out older data. However, keep in mind that you can change this time selection at any time.
4. Also note that the code editor shows a performance tip just to the right of the current breakpoint. This PerfTip shows the elapsed time since the previous breakpoint and allows you to keep tabs on code performance without the need to instrument the code with StopWatch or other utilities.

## Lambda debugging

1. Open MainWindowViewModel
2. In ShowChargesAsync method, put breakpoint after inside .ContinueWith
3. Run app
4. Want to see what is in charges. Can use datatip, but not easy to use. Can use Locals window. Better but not flexible enough
5. In Immediate Window enter **charges.Result**
6. Enter **charges.Result.Count()**
7. Enter **charges.Result.Select(c=>c.Description)**
8. Enter **charges.Result.Where(c=>c.BilledAmount>100).Count()**
9. Enter **charges.Result.Where(c=>c.BilledAmount>100).Select(c=>new {c.Merchant, c.BilledAmount})**
10. Copy any of these into Watch window

## Live Visual Tree & Live Properties

1. Run Expenses app
2. Debug | Windows | Live Visual Tree
3. Debug | Windows | Live Properties
4. Explore visual tree
5. Click Enable selection in the running application
6. Modify some text, eg button labels and title above grid
7. Select first row of data
8. What happens if charges are large? Modify charge to be 20000 and then 200000
9. What happens if there is a lot of text? Modify merchant name
10. Click Display layout adorners in the running application
11. Select Border in visual tree and set Background to Red
12. To get app working again unselect Enable selection and Display layout
13. Click New Charge

## IntelliTest

1. Open ExpenseService.svc.cs
2. Locate CreateNewCharge method
3. Right-click | Create Unit Tests. Just creates scaffolding. No actual tests.
4. Delete the test project
5. Right-click | Run IntelliTest. 2 tests run. 1 fails because I don’t check for null.
6. Couple of warnings.
7. Click Save in the IntelliTest Exploration Results window. New test project is created
8. ExpenseServiceTest.cs is the PUT (parameterized unit test)
9. ExpenseServiceTest.CreateNewCharge.g.cs is the actual unit test
10. Test | Windows | Test Explorer. No tests appear
11. Build
12. Test | Windows | Test Explorer. 1 test appears.
13. Run. It fails.
14. Select CreateNewChargeThrowsNullReferenceEx. See test.
15. Modify CreateNewCharge to return -1 if charge is null
16. Build
17. Run test in Test Explorer. Passes.
18. Run tests in IntelliTest Exploration Results. They pass.
19. Expand ExpenseServiceTest in Solution Explorer. Notice two versions of CreateNewCharge, 1 when charge is null and one when it isn’t.
20. In IntelliTest Exploration Results, click Warnings.
21. Suppress Uninstrumented Method.
22. Fix Boundary. Show MaxBranches in ExpenseServiceTest.cs
23. Run tests.
24. Set MaxBounds to 80,000.
25. Run tests
26. Allow No connection string
27. Run tests. All tests pass.
28. Right click inside the class but outside any method ad select Run IntelliTest.
29. See the new test project in the Solution Explorer. Expand ExpenseServiceTest and see that VS created unit tests for each method inside the class.
30. Note: You need to save each of them one at a time. If you run any, the unsaved ones are tossed away. Hint: Close the project and VS prompts you to save all.

## Timeline Tool

### Profile Application with the Timeline Tool

1. Load the BlankXamlApp.sln solution file from the Samples folder on the Desktop and then start a debugging session using the Simulator (select the drop down with the green play arrow in the toolbar if necessary).
2. Take a look at the Diagnostic Tools window.
3. Stop the debugging session.
4. Select Debug | Start Diagnostic Tools Without Debugging from the main menu.
5. Select the Timeline tool from the list and then click Start.
6. Exercise the application in the same way that you previously did, by scrolling to the right a few times.
7. Switch back to Visual Studio and stop the debugging session.

### Analyzing the Diagnostic Session Report

1. After the diagnostic session report is shown, take a look at the UI thread utilization (%) graph. This graph details the time spent by the UI thread in parsing, layout, rendering, I/O, application code, and so on. In this particular case, it shows that it spent quite a while on layout work (in orange color).
2. Application lifecycle marks are shown above the UI thread utilization graph, and you should see one representing application launch and another representing application activation. It is possible to add user marks also, but we will not do so in this lab.
3. The Timeline Details view allows you to perform some more detailed analysis of the application CPU utilization categorized by the UI Framework subsystem or system component that consumed the CPU. Most of the events occur on the UI thread and are marked as such by the purple bar on the left the event in the details view. In addition, the timeline details view also shows the amount of time taken for each event.
4. Use the mouse to zoom into a section (click and drag) that shows significant Layout activity on the UI thread utilization graph. This will allow you to further drill down and filter to the events in the selected time range. Some of the events have additional descriptors in their names that provide more information, for example the Layout event shows the number of elements that have taken part in that layout pass.
5. By default, the timeline is sorted chronologically. Select the Sort By drop down and select the Duration option to see the most expensive items at the top of the list. Note that they are mostly Layout related.
6. Drill into one of the Layout nodes a couple of steps to see what XAML elements are involved that are taking so long.
7. Leave the diagnostic session window open, as you will be able to compare this baseline to future runs.

### Fixing the Performance Issue

1. Open MainPage.xaml in the editor and take a look at the GridView declaration. The GridView is ultimately bound to a list of data items in code, and it uses a data template definition to display each item. In an attempt to improve UI responsiveness, the developer of the application tried to take advantage of the ContainerContentChanging event to incrementally update pieces of each tile, but unfortunately it appears that the code added to the handler is quite expensive.
2. Place the cursor anywhere on the event handler assignment for the ContainerContentChanging event (MyGridView\_ContainerContentChanging) and then press F12 to go to the definition in the code behind.
3. The first line of the event handler makes a call to a long running operation, and is likely to be a major contributor to the poor responsiveness that you just experienced since it executes on the UI thread.
4. Offload the intensive operation from the UI thread by using Task.Run() as shown below. For the purposes of this lab, assume that this long running operation needs to be performed here and can be performed asynchronously without side effect.

**Task.Run(() =>**

**{**

**Utility.LongOperation();**

**});**

### Verifying the Performance Issue Fix

1. Run the XAML UI Responsiveness tool once again, performing a similar test that you previously did including scrolling through the items. You should notice an improvement in responsiveness as you scroll.
2. Stop the profiler to view the report.
3. Note that the UI thread utilization percentage is much less than it was before. You can switch between the before and after reports to verify this.

# Language Enhancements

## C# Language Enhancements

No big new concepts. Mostly small useful features that help clean up code and get rid of boilerplate.

* Open CSharp6Demos project.
* Create Point class. Copy code from StartingPoint.

## Getter only auto properties

Currently auto properties need to have setters. Means you can’t really have immutable data types. Now allow getter only auto properties. They have a backing field that is read-only.

1. Modify auto properties to be getter only:

|  |  |
| --- | --- |
| Old | New |
| public int X { get; set; }  public int Y { get; set; } | public int X { get; };  public int Y { get; }; |

## Initialize auto properties

We now allow auto properties to be initialized.

1. Initialize the auto properties

|  |  |
| --- | --- |
| Old | New |
| public int X { get; };  public int Y { get; }; | public int X { get; } = 5;  public int Y { get; } = 7; |

## Reference static classes

Can now reference static classes in your using clauses, rather than only to namespaces. Static members of the class are now directly in cope. Note this only works for static classes. Can’t use System.Console and then call WriteLine.

1. Add using System.Math and change Math.Sqrt to Sqrt.

|  |  |
| --- | --- |
| Old | New |
| get { return Math.Sqrt(X \* X + Y \* Y); } | using static System.Math;  public double Dist => Sqrt(X \* X + Y \* Y); |

## String interpolation

String.Format is hard to use and is very error prone. String interpolation makes it easier to format strings. You can put the expressions right in their place.

1. Change the String.Format call

|  |  |
| --- | --- |
| Old | New |
| return String.Format("{0},{1})", X, Y); | return $"{X}, {Y}"; |

## Expression bodies on method-like members

There are lots of methods that consists just of a return statement. You can now use the lambda arrow to write this in only one line of code.

1. Change ToString method

|  |  |
| --- | --- |
| Old | New |
| public override string ToString()  {  return $"{X}, {Y}";  } | public override string ToString() => $"{X}, {Y}"; |

## Expression bodies on property-like function members

This also applies to computed properties. Here we have a getter that just returns a value. We can reduce this to one line of code.

1. Change Dist method

|  |  |
| --- | --- |
| Old | New |
| public double Dist  {  get { return Math.Sqrt(X \* X + Y \* Y); }  } | public double Dist => Sqrt(X \* X + Y \* Y); |

## Index initializers

Currently, you can initialize properties in an object initializers, but not indices. Now you can.

1. Change ToJson method

|  |  |
| --- | --- |
| Old | New |
| public JObject ToJson()  {  var result = new JObject();  result["x"] = X;  result["y"] = Y;  return result;  } | public JObject ToJson() => new JObject() {["x"] = X, ["y"] = Y}; |

## Null-conditional operators

Look at this code to turn Json objects back into Point objects. The return is the line of code that does something. Everything else is checking for nulls and types. Null-conditional operators make it much easier to check for nulls. If the left hand side is null, the whole thing is null. Only if it isn’t null do we call the dot operator.

1. In FromJson method, add ? after json[“x”] and json[“y”] and remove lines that check if those are null.

|  |  |
| --- | --- |
| Old | New |
| public Point FromJson(JObject json)  {  if (json != null &&  json["x"] != null &&  json["x"].Type == JTokenType.Integer &&  json["y"] != null &&  json["y"].Type == JTokenType.Integer)  {  return new Point((int)json["x"], (int)json["y"]);  }  return null;  } | public Point FromJson(JObject json)  {  if ((json != null &&  json["x"] ?.Type == JTokenType.Integer &&  json["y"]?.Type == JTokenType.Integer)  {  return new Point((int)json["x"], (int)json["y"]);  }  return null;  } |

1. Add ? after json in both of those and remove first null check.

|  |  |
| --- | --- |
| Old | New |
| public Point FromJson(JObject json)  {  if ((json != null &&  json["x"] ?.Type == JTokenType.Integer &&  json["y"]?.Type == JTokenType.Integer)  {  return new Point((int)json["x"], (int)json["y"]);  }  return null;  } | public Point FromJson(JObject json)  {  if (json?["x"] ?.Type == JTokenType.Integer &&  json["y"]?.Type == JTokenType.Integer)  {  return new Point((int)json["x"], (int)json["y"]);  }  return null;  } |

## Nameof operator

Often you need to get the name of a program element as a string. The problem is that if you rename the element, your code is broken. The nameof operator solves this problem. If you rename the element, you stay in sync.

1. In Add method, change call to ArgNullEx

|  |  |
| --- | --- |
| Old | New |
| throw new ArgumentNullException("json"); | ArgumentNullException(nameof(json)); |

## Exception filters

VB and F# contain exception filters. C# now has them too.

1. Add an exception filter to the try..catch

|  |  |
| --- | --- |
| Old | New |
| catch (HttpRequestException e)  {  if (DateTime.Today.DayOfWeek == DayOfWeek.Thursday)  {  Log(e);  }  }  } | catch (HttpRequestException e) when (DateTime.Today.DayOfWeek == DayOfWeek.Thursday)  {  Log(e);  } |

## Async calls in catch and finally block

1. Add async calls to catch and finally block.

|  |  |
| --- | --- |
| Old | New |
| catch (HttpRequestException e) when (DateTime.Today.DayOfWeek == DayOfWeek.Thursday)  {  Log(e);  }  finally  {  LogGenericError();  } | catch (HttpRequestException e) when (DateTime.Today.DayOfWeek == DayOfWeek.Thursday)  {  await LogAsync(e);  }  finally  {  await LogGenericErrorAsync();  } |

## C# Essentials extension

1. Navigate to <https://visualstudiogallery.msdn.microsoft.com/a4445ad0-f97c-41f9-a148-eae225dcc8a5> in browser.
2. Select Tools | Extensions and Updates.
3. Search for and download c# essentials.
4. Build the solution.
5. Open StartingPoint.cs.
6. Click on String.Format. Lightbulb makes a suggestion. Select Convert to interpolated string.
7. Click in Dist method. Lightbulb makes a suggestion. Select Use expression bodied member.
8. Click on last line of FromJson. Lightbulb makes a suggestion. Select Use NameOf.