Live Unit Testing and Live Dependency Validation in Visual Studio 2017

Lab version: 15.0.26228.0

Last updated: 3/6/2017

1. **TABLE OF CONTENTS**

[Overview 1](#_Toc475302485)

[Related Resources 1](#_Toc475302486)

[Prerequisites 2](#_Toc475302487)

[Exercise 1: Live Unit Testing in Visual Studio 2017 2](#_Toc475302488)

[Task 1: Live unit testing 2](#_Toc475302489)

[Exercise 2: Real Time Dependency Validation in Visual Studio 2017 4](#_Toc475302490)

[Task 1: Creating a dependency diagram 4](#_Toc475302491)

[Task 2: Live dependency validation 7](#_Toc475302492)

# Overview

Live Unit Testing is a new feature introduced in Visual Studio 2017. It visualizes unit test results and code coverage live in the editor alongside your code and updates while you are coding. It works with C# and VB projects for the .NET framework and supports three test frameworks: MSTest, xUnit, and NUnit. Live Unit Testing makes it easy to maintain quality and test coverage during rapid development and takes your productivity to a whole new level. Imagine fixing a bug in a code base in which you may not be completely familiar with. With Live Unit Testing, you can know right away—as you are making edits to fix the bug—that you did not break another part of the system. Getting this feedback as you type will help you feel confident that your effort is delivering the results you expect.

Visual Studio also enables teams to validate architectural dependencies in their solutions using the [Layer Designer](https://msdn.microsoft.com/en-us/library/dd465141.aspx), which first shipped in Visual Studio 2010. This feature helps ensure that developers respect the architectural constraints of the application as they edit their code. It has been re-written in Visual Studio 2017 to use Roslyn analyzers for a more robust, reliable experience. Developers now get real-time design feedback in the editor so they can immediately stop introducing architectural flaws. “Layer” diagrams have also been renamed “dependency validation” diagrams to make it clearer what they are for.

## Related Resources

See this [blog post](https://blogs.msdn.microsoft.com/visualstudio/2016/11/18/live-unit-testing-visual-studio-2017-rc/) for more information on live unit testing.

See this [blog post](https://blogs.msdn.microsoft.com/visualstudioalm/2016/11/30/live-dependency-validation-in-visual-studio-2017/) for a detailed discussion comparing the new live dependency validation experience with the past.

## Prerequisites

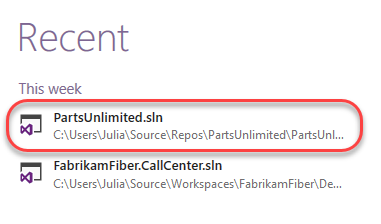
In order to complete this lab you will need the Visual Studio 2017 virtual machine provided by Microsoft. For more information on acquiring and using this virtual machine, please see [this blog post](http://aka.ms/almvm).

# Exercise 1: Live Unit Testing in Visual Studio 2017

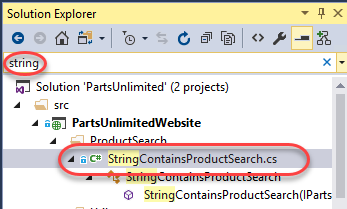
Live Unit Testing is a service that watches your codebase for changes. As you edit files, it automatically detects which tests are impacted and runs those tests in the background. The results are provided immediately so that you can get feedback on how your changes affect the overall codebase quality. It also indicates which lines of code have no test coverage so that you can think about testability as you develop.

## Task 1: Live unit testing

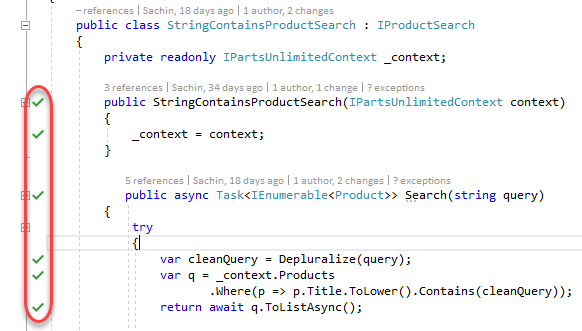
1. Log in as **Sachin Raj (VSALM\Sachin)**. All user passwords are **P2ssw0rd**.
2. Launch **Visual Studio 2017** from the taskbar.
3. Open the **PartsUnlimited** solution from the **Start Page**.



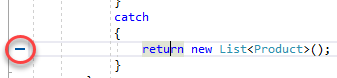
1. In **Solution Explorer**, search for **“string”** and double click **StringContainsProductSearch.cs** to open it.



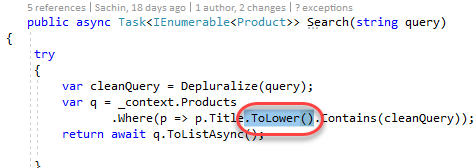
1. This class already has some tests written and is currently in pretty good shape. However, suppose you were planning to make some changes and wanted to get an understanding of the quality of the code before you begin. Rather than running all tests, you can now simply enable live unit testing and let it illustrate the coverage.
2. Select **Test | Live Unit Testing | Start** to start live unit testing. After a moment you will see glyphs appear in the left margin of the editor. There are three types of glyphs that indicate that all tests for a line pass (a green check), at least one test fails (a red X), or that the line is not tested (a blue line). In this case, you can see that the code in the constructor and beginning of **Search** pass their tests.



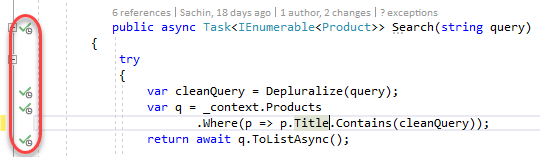
1. Looking further down, it’s clear that the exception case for this method is not tested, which is something you should think about adding later on.



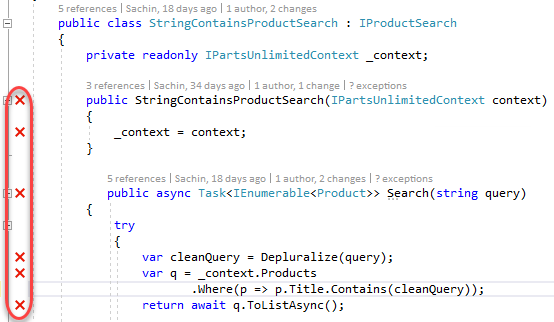
1. Now let’s experiment with a change to the code. Delete the .**ToLower()** call in **Search**.



1. The live unit testing service will immediately detect the change in code and re-run the tests (note the clock element is added to the glyphs). You don’t even need to save the file.



1. Since the change above broke some tests, the glyphs are changed to reflect which lines of code are now part of failing tests. It’s important to note that these glyphs don’t indicate that the error occurred on their respective lines, but rather that those lines are executed by tests that ultimately fail.

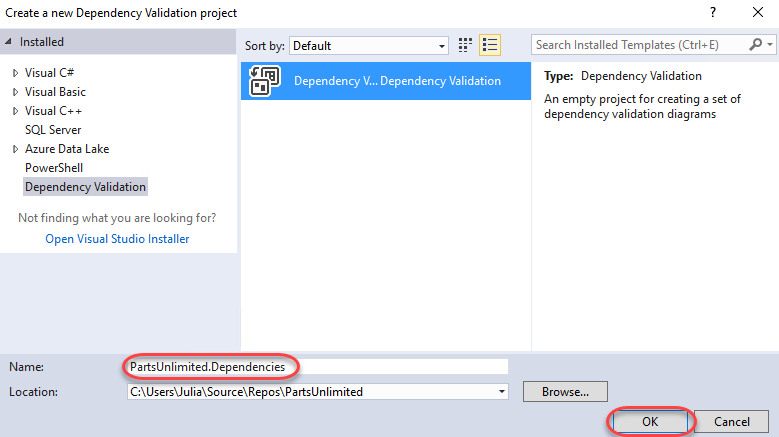


1. Pres **Ctrl+Z** to undo the changes made earlier. You may need to press it multiple times to restore the **.ToLower()** from above. The glyphs should all return to passing.

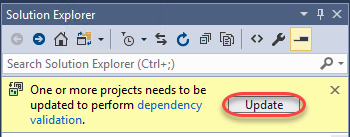
# Exercise 2: Real Time Dependency Validation in Visual Studio 2017

## Task 1: Creating a dependency diagram

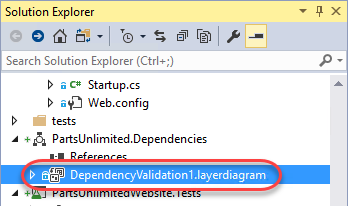
1. Press **Ctrl+Shift+B** to rebuild the solution.
2. Select **Architecture | New Dependency Validation Diagram** from the main menu.
3. Enter a **Name** of **“PartsUnlimited.Dependencies”** and click **OK**.



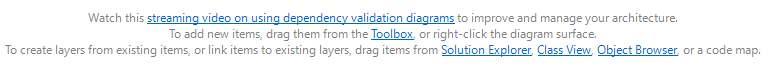
1. If asked to update projects to support dependency validation, click **Update**.



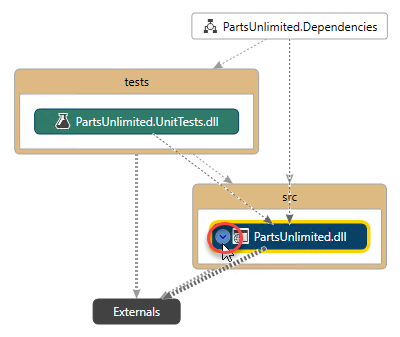
1. Note that the new project has been added to the solution. Double-click **DependencyValidation1.layerdiagram** to open it.



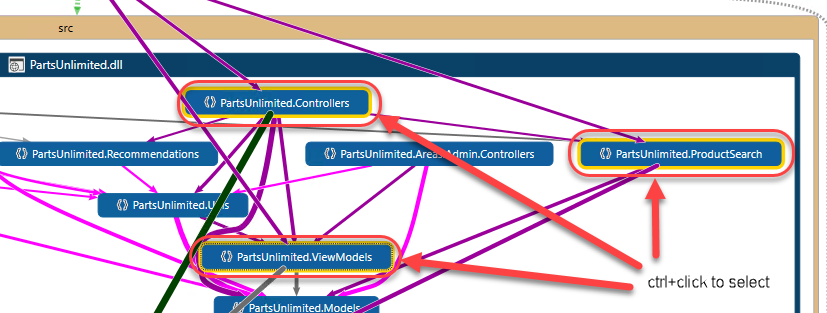
1. There are a lot of ways to add layers to the diagram, including from the **Solution Explorer**, **Class View**, and **Object Browser**.



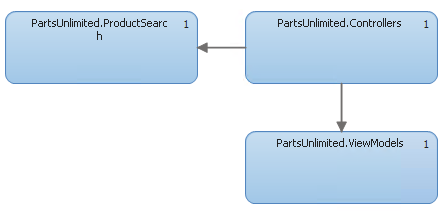
1. In this case, we’re going to automatically generate a code map and then use that to codify our architecture via the diagram. Select **Architecture | Generate Code Map for Solution**. This code map will pull in everything it can about the project and represent it in a diagram so we can easily walk through the de facto dependencies and relationships each component (class, etc) has.
2. Expand the **PartsUnlimited.dll** component to see the namespaces it contains.

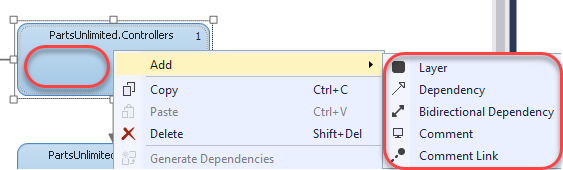


1. It gets a little messy at this point because we have lots of relationships and dependencies mapped across and within the project and its tests. Use **Ctrl+Click** to select **PartsUnlimited.Controllers**, **PartsUnlimited.ViewModels**, and **PartsUnlimited.ProductSearch**. Press **Ctrl+C** to copy them.



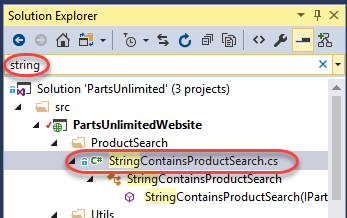
1. Switch back to the layer diagram and press **Ctrl+V** to paste the three components from the last step. You may want to rearrange them as shown below to make their relationships easier to understand. Put simply, the controllers namespace depends on the product search and viewmodel namespaces. While these relationships existed before and may have been unofficial while the project was being developed, putting them into the layer diagram makes them official architectural law moving forward.



1. It’s important to understand that the rules only exist for layers that exist on the diagram. For example, the **Controllers** namespace can reference the **ProductSearch** or **ViewModels** namespaces because their relationships are explicitly defined. At the same time, those two namespaces may not reference anything from the **Controllers** namespace because the dependency is directional (there are bidirectional dependencies available if two-way references are desired). **ProductSearch** and **ViewModels** may also not reference each other because they do not have an explicit dependency in the diagram. However, a namespace like **PartsUnlimited.Utils** is not used in the layer diagram and therefore is not enforced by the dependency validation engine. It may reference anything and anything may reference it. However, if it were added to the diagram, then all of its dependencies with every other layer in the diagram would be enforced.
2. In addition to the methods discussed earlier, you can also easily add new layers and dependencies by right-clicking the diagram or layers and selecting them. The layers can be at any level, whether it’s a namespace, class, or even a method.
3. Press **Ctrl+S** to save the diagram.

## Task 2: Live dependency validation

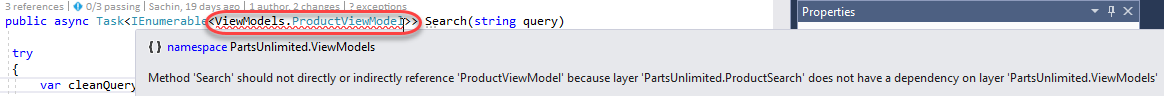
1. Now it’s time to break some rules. Press **Ctrl+Shift+B** to rebuild the solution. It should build as expected because we haven’t violated any policy…yet.
2. In **Solution Explorer**, search for **“string”** and open **StringContainsProductSearch**. This class is a member of the **PartsUnlimited.ProductSearch** namespace, and therefore is not allowed to reference anything from **PartsUnlimited.ViewModels**.



1. Locate the **Search** method. It currently returns a **Task<IEnumerable<Product>>**. Change it to return a **Task<IEnumerable<ViewModels.ProductViewModel>>**.



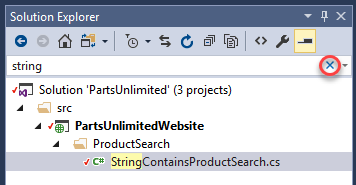
1. While this change introduces other errors as well, the key thing to focus on here is that the editor immediately provides a red squiggle indicating that this reference is not allowed based on its defined relationships. This live dependency validation provides immediate feedback to enforce architectural practices so that developers can feel confident that they’re following the project rules.



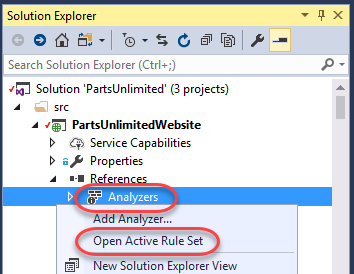
1. And even if the developer were to ignore the red squiggle, they would still be unable to build the solution due to error-level enforcement of the policy.



1. However, you can customize exactly how dependency rule violations are treated. In **Solution Explorer**, clear the search box.



1. Under **References**, right-click **Analyzers** and select **Open Active Rule Set**.



1. Expand the **Microsoft.DependencyValidation.Analyzers** rule set and note how you could change the violation behavior from an **Error** to a **Warning** (or whatever you prefer). There is great flexibility in managing dependency validation.

