* 1. 

Hands-On Lab

Module 07: Web Services

Lab version: 1.0.0

Last updated: 10/14/2011

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user.

Microsoft may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, or other intellectual property.

© 2011 Microsoft Corporation. All rights reserved.

Microsoft, MS, Windows, MSDN, Visual Basic, Visual C++, Visual C#, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

[Creative Commons License](http://creativecommons.org/licenses/by/3.0/)Microsoft Biology Initiative Training by [Microsoft Corporation](http://research.microsoft.com/bio) is licensed under a [Creative Commons Attribution 3.0 Unported License](http://creativecommons.org/licenses/by/3.0/). Based on a work at [research.microsoft.com/bio](http://research.microsoft.com/bio).

[Module 07: Web Services 1](#_Toc293908690)

[Web Services 4](#_Toc293908691)

[Objectives 4](#_Toc293908692)

[System Requirements 4](#_Toc293908693)

[Task 1: Executing Web Services in .NET Bio 5](#_Toc293908694)

[Task 2: Adding Async service support 13](#_Toc293908695)

[Task 3: Customizing service settings 19](#_Toc293908696)

Web Services

This Hands-On Lab walks you through building an application that executes the BLAST algorithm using web services against sequences loaded with .NET Bio. You will extend an existing WPF application to call various implementations of BLAST using web service calls.

Objectives

In this Hands-On Lab, you will get some experience building an application that uses .NET Bio

* + Loading and identifying web service handlers for BLAST.
  + Passing sequences and sequence fragments to BLAST.
  + Changing the BLAST parameters.
  + Displaying the results from a BLAST run.

# System Requirements

You must have the following items to complete this lab:

* + Microsoft Visual Studio 2010
  + .NET Bio 1.0 or later
  + Windows XP SP3 or better

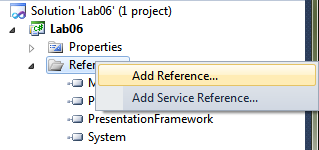
Task 1: Executing Web Services in .NET Bio

In this task you will examine an existing application and see how it is put together so you can add to and extend it. The first extension we will add is to identify all the BLAST algorithms and synchronously invoke one.

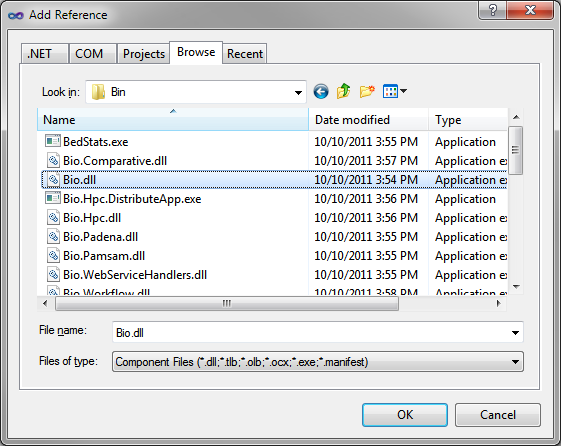
1. Open the starter project located at [Task1\before\Lab06.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp5_MBF.V2.zip\MBF.V2\Module%2006\Lab\Task1\before\Lab06.sln).
2. Examine the project structure. It has the following folders and files:

|  |  |
| --- | --- |
|  | * **Infrastructure** – contains some helper classes and that allow the application to separate behavior and UI with the Model-View-ViewModel (a variation of MVC) design pattern. * **ViewModels –** logic that drives the UI – this is where you will make all of your changes. * **App.xaml** – the primary Application object. * **MainWindow.xaml** – main window UI, this is connected to the ViewModel using WPF data binding. You will not make any changes to this file, but you can examine it to see how the UI is composed if you like. |

1. Expand the ViewModels folder. Here you will find three view models. Remember from our previous WPF labs that a ViewModel is simply a model wrapper for the view: it provides data-bindable properties that WPF can then visualize. In this application, the .NET Bio interfaces and structures (such as **ISequence**) represent the underlying *model* and these classes wrap the data structures to make them easier to display in WPF.
   1. **BaseViewModel.cs** is the base class for all view models; it provides the **INotifyPropertyChanged** support required for any VM to communicate property changes with the View.
   2. **SequenceViewModel** is a VM that wraps an **ISequence** interface.
   3. **MainViewModel** is the primary VM for the application; if you open it you will find it holds a collection of **SequenceViewModel** objects, and also manages a single “selected” sequence. The code to load sequences has already been provided, and in addition, two stub functions are provided – one to run the BLAST algorithm and a second to cancel a running operation. Our work in this lab will revolve around implementing these two features.
   4. **Note:** if you want to explore the Model-View-ViewModel (MVVM) design pattern that is being used here more fully to understand the code, please read: <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>
2. As a first step, we must add a reference to .NET Bio – right click on the **References** folder and select “**Add Reference**”.

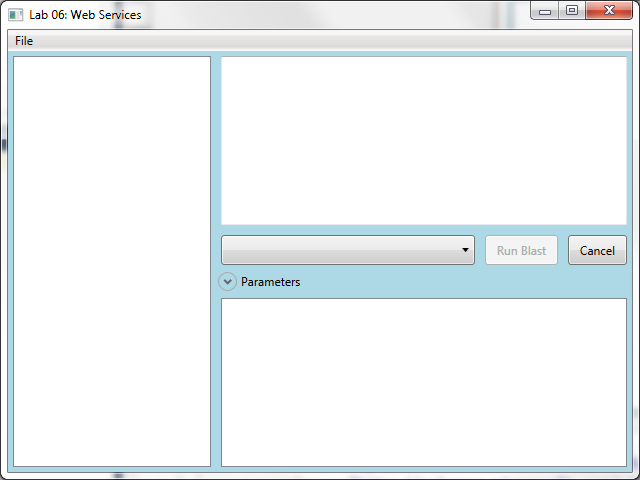


1. Change the directory to the “**.NET Bio**” folder. On a 32-bit machine this will normally be located in the “**C:\Program Files**” and on a 64-bit machine it will be in “**C:\Program Files (x86)**”. Select the version of .NET Bio you want to use by navigating to the next directory (as of this writing, the current version is “1.0”), and then finally select the assemblies in the **Tools\Bin** directory. Below, the full path is **“C:\Program Files\.NET Bio\1.0\Tools\Bin”**

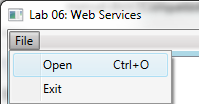


* 1. Select both Bio.dll and Bio.WebServiceHandlers.dll. We will use both in this lab.
  2. Click the **OK** button to add the references.

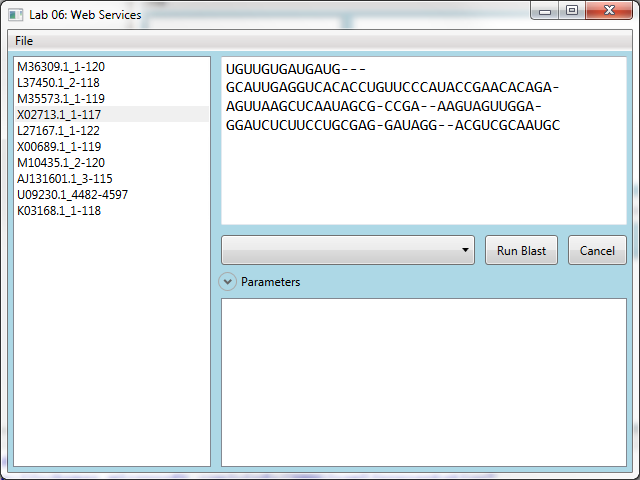
1. Build and run the application. It will present a UI that looks like:



1. You can load sequences from all the supported parser formats using the **File | Open** menu option.



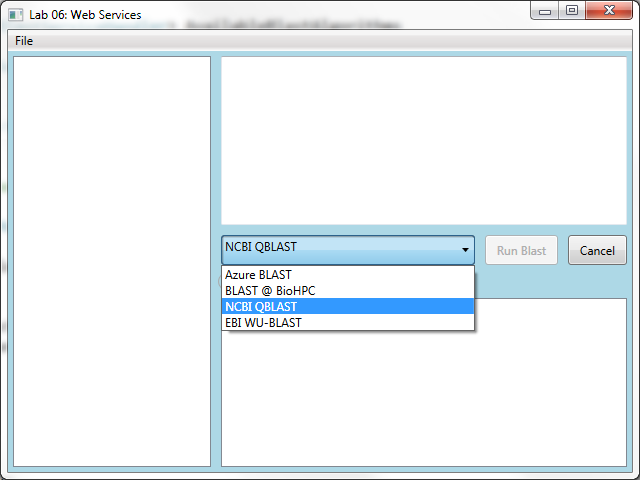
1. There are several sample files in the **Data** folder at the root of this lab and you can import as many files as you like, even of different molecule types and formats. Once you load a data file, you can click on a sequence loaded to select it. This will display the entire sequence in the **TextBox** on the right side of the window.



1. Notice that there are already UI elements present for selecting the algorithm, running the BLAST algorithm and canceling it; but none of them are functional at the moment.

We will start by implementing support to select the BLAST algorithm. The UI requires us to have a property **AvailableBlastAlgorithms** that returns an enumerable sequence of the **IBlastServiceHandler** objects that are registered with .NET Bio.

1. Open the **MainViewModel.cs** file in the **ViewModels** folder.
2. You will need to add namespace references for **.NET Bio.Web** and **.NET Bio.Web.Blast** at the top of the file. These are where the base web service support and specific-BLAST support classes are located.
3. Add a new public property called **AvailableBlastAlgorithms** that returns an **IEnumerable<IBlastServiceHandler>**. The implementation should use the **WebServices** class and return all **IBlastServiceHandler** implementations. Here is an example implementation that uses the LINQ **OfType<T>** extension method, but you can also iterate the collection and locate each implementation of the proper type as well.
   1. public IEnumerable<IBlastServiceHandler> AvailableBlastAlgorithms
   2. {
   3. get { return WebServices.All.OfType<IBlastServiceHandler>(); }
   4. }
4. Add a second public property to hold the selected BLAST implementation. It should be called **SelectedBlastAlgorithm** and return a single **IBlastServiceHandler**. Make sure to call **OnPropertyChanged** to notify the UI that the property has changed values in the setter.
   1. private IBlastServiceHandler \_selectedBlastAlgorithm;
   2. public IBlastServiceHandler SelectedBlastAlgorithm
   3. {
   4. get { return \_selectedBlastAlgorithm; }
   5. set
   6. {
   7. \_selectedBlastAlgorithm = value;
   8. OnPropertyChanged("SelectedBlastAlgorithm");
   9. }
   10. }
5. Build and run the application. The dropdown list should now contain all the located BLAST handlers:



If you load and select a sequence, you may notice that the “**Run Blast**” button is enabled. This happens even if you do not have a specific BLAST algorithm selected in the ComboBox. We only want the “**Run BLAST**” button available when we have a selected algorithm. It currently just requires a loaded sequence. To fix this:

1. Locate the constructor for MainViewModel. This is where the command for RunBlast is created. The second lambda is the “conditional” that determines whether the command is valid (and therefore whether the button should be selected).
   1. public MainViewModel()
   2. {
   3. LoadedSequences = new ObservableCollection<SequenceViewModel>();
   4. ImportFile = new DelegatingCommand(OnLoadFile);
   5. RunBlast = new DelegatingCommand(OnBlastSequence,
   6. **() => SelectedSequence != null**);
   7. CancelBlast = new DelegatingCommand(OnCancelBlast);
   8. }
2. Add a second conditional to test the **SelectedBlastAlgorithm** property to ensure it is not null:
   1. RunBlast = new DelegatingCommand(OnBlastSequence,
   2. () => SelectedSequence != null **&& SelectedBlastAlgorithm != null**);

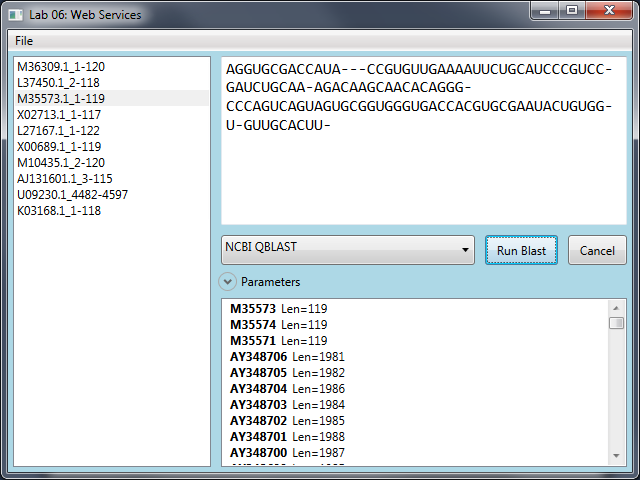
Next, we will call the implementation.

Locate the **OnBlastSequence** method at the end of the **MainViewModel** class. It currently has some code to create or assign the selected **ISequence** we want to BLAST. It uses either the selection, or the entire selected sequence if no selection is performed in the TextBox.

1. Execute the selected BLAST algorithm – there are several setup steps:
   1. Create a **BlastParameters** structure. For now we will assume you are calling the NCBI service which requires the **Program**, **Database** and **Expect** range. Set each of these values to “blastn”, “nr” and “10.0” respectively. Later we will add options for other versions of the service.
   2. Invoke the **SubmitRequest** method and check the return value to ensure it is non-null (null indicates a failure). Make sure to wrap the call in a try/catch and output a failure using the **ShowError** method that’s part of the class.
   3. Fetch the results using the **FetchResultsSync** method. Again, test for errors here.
   4. Here is an example implementation if you need help with this:
   5. private void OnBlastSequence()
   6. {
   7. // Get the full sequence, or the selected fragment
   8. ISequence sequence = !string.IsNullOrEmpty(SelectedSequenceFragment)
   9. ? new Sequence(SelectedSequence.RawSequence.Alphabet,
   10. SelectedSequenceFragment)
   11. : SelectedSequence.RawSequence;
   12. IBlastServiceHandler blastServiceHandler = SelectedBlastAlgorithm;
   13. BlastParameters bp = new BlastParameters();
   14. bp.Add("Program", "blastn");
   15. bp.Add("Database", "nr");
   16. bp.Add("Expect", "10.0");
   17. try
   18. {
   19. string resultKey = blastServiceHandler.SubmitRequest(sequence, bp);
   20. if (string.IsNullOrEmpty(resultKey))
   21. ShowError("Failed to execute Blast", "No results returned", null);
   22. else
   23. {
   24. IList<BlastResult> results =
   25. blastServiceHandler.FetchResultsSync(resultKey, bp);
   26. if (results == null || results.Count == 0)
   27. {
   28. MessageBox.Show("No Results returned.");
   29. ResultHits = null;
   30. }
   31. }
   32. }
   33. catch (Exception ex)
   34. {
   35. ShowError("Failed to execute Blast", "Error occurred", ex);
   36. ResultHits = null;
   37. }
   38. }

Recall that the **BlastResult** contains a collection of **BlastSearchRecords** detailing the results. Each search record contains a set of Hits that detail how close the fragment matched the given record. The UI is already created to display these results and expects a collection of **Hit** objects to be stored in a public property called **ResultHits.** It will display these results in the bottom-right list of the UI.

1. Create the **ResultHits** property and assign the **Hit** collection from the first search record. We will assume a single return record for now.
   1. Use **IList<Hit>** as the return type
   2. Make sure to call **OnPropertyChanged** when the setter is called.
   3. Assign the value in your **OnBlastSequence** method when valid results are detected.
   4. public IList<Hit> ResultHits
   5. {
   6. get { return \_resultHits; }
   7. private set { \_resultHits = value; OnPropertyChanged("ResultHits"); }
   8. }
   9. private void OnBlastSequence()
   10. {
   11. ...
   12. try
   13. {
   14. string resultKey = blastServiceHandler.SubmitRequest(sequence, bp);
   15. if (string.IsNullOrEmpty(resultKey))
   16. ShowError("Failed to execute Blast", "No results returned", null);
   17. else
   18. {
   19. IList<BlastResult> results =
   20. blastServiceHandler.FetchResultsSync(resultKey, bp);
   21. if (results == null || results.Count == 0)
   22. {
   23. MessageBox.Show("No Results returned.");
   24. ResultHits = null;
   25. }
   26. else
   27. ResultHits = results[0].Records[0].Hits;
   28. }
   29. }
   30. catch (Exception ex)
   31. ...
2. Run the program, load some sequences, select one and run the BLAST algorithm by selecting the NCBI version and clicking the “**Run Blast**” button. It will take some time, but eventually return results:

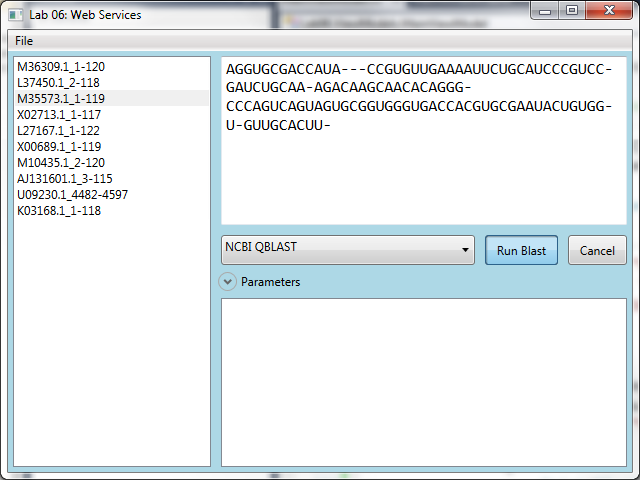


In the next task we will add some async support so the application continues to respond while waiting for the results.

Task 2: Adding Async service support

In this task you will modify your code that calls the BLAST service to be performed asynchronously so it does not block the application from responding to the user. As part of that, we will add support to cancel an existing operation now that it is running in the background.

If you recall, in the previous task we added support to call NCBI’s BLAST service. When you run the BLAST algorithm, the UI “locks up” waiting for the response – which can take several seconds or even minutes to respond:

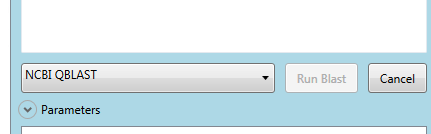


The key to fixing this issue is to execute the call asynchronously on a background thread which leaves the primary UI thread free to process UI events. We could do this a variety of ways –

* Use a .NET 4.0 Task – this is similar to a thread, but a bit more intelligent in scheduling. It also supports cancelation which is a nice feature.
* Execute the call on a dedicated thread, or using Background Worker (a Windows Forms and WPF class intended for background operations). This also supports cancelation, although it may require some code to implement it.
* Use an asynchronous delegate or Threadpool call to dispatch the work onto a thread pool owned thread. This is not ideal for this operation because we cannot tell how long the operation will take, it cannot be canceled once it is executed and there are a limited number of thread pool threads available.
* Use any built-in asynchronous support in the class. This is the ideal solution because it leaves the implementation up to the component in question: in some cases this may even not require a thread for the entire time (such as I/O bound operations which can be dispatched to the kernel).

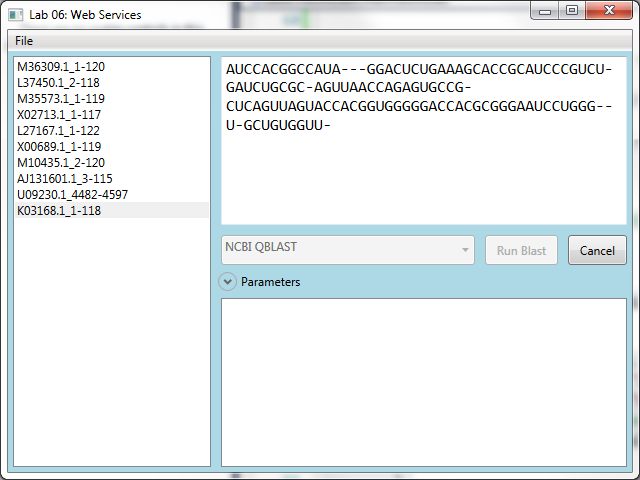
Since the **IBlastServiceHandler** interface already supports native async support, we will take advantage of that here. If it did not, we would have to use one of the other three options.

1. You can continue from the prior task, or open [Task2\before\Lab06.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp5_MBF.V2.zip\MBF.V2\Module%2006\Lab\Task2\before\Lab06.sln).
2. Open the **MainViewModel** class (if you closed it). Locate your **OnBlastSequence** handler.
3. Just before you submit the request, wire up an event handler to the **RequestComplete** event of the **IBlastServiceHandler** interface.
   * 1. **Hint**: you can get Visual Studio to generate the handler by typing “blastServiceHandler.RequestCompleted +=” and then hitting TAB twice. This assumes that your instance is called “blastServiceHandler”.
4. Move the results code into the new handler – instead of calling **FetchResultsSync** to get the results, just use the supplied EventArgs **SearchResult** property (**e.SearchResult**).
5. You also need to check for errors. The **ErrorMessage** and/or the **Error** property of the passed EventArgs will be set if there are errors. Output an error message if that happens and just return from the method. Make sure to do this before attempting to process the results!
6. Here is some example code if you need some help:
   1. private void OnBlastSequence()
   2. {
   3. ...
   4. blastServiceHandler.RequestCompleted += BlastServiceHandlerRequestCompleted;
   5. try
   6. {
   7. string resultKey = blastServiceHandler.SubmitRequest(sequence, bp);
   8. if (string.IsNullOrEmpty(resultKey))
   9. ShowError("Failed to execute Blast", "No results returned", null);
   10. }
   11. catch (Exception ex)
   12. {
   13. ShowError("Failed to execute Blast", "Error occurred", ex);
   14. ResultHits = null;
   15. }
   16. }
   17. void BlastServiceHandlerRequestCompleted(object sender, RequestCompletedEventArgs e)
   18. {
   19. if (e.Error != null || !string.IsNullOrEmpty(e.ErrorMessage))
   20. {
   21. ShowError("An error occurred", e.ErrorMessage, e.Error);
   22. return;
   23. }
   24. IList<BlastResult> results = e.SearchResult;
   25. if (results == null || results.Count == 0)
   26. {
   27. MessageBox.Show("No Results returned.");
   28. ResultHits = null;
   29. }
   30. else
   31. ResultHits = results[0].Records[0].Hits;
   32. }
7. Try the application now. It should submit the request and then allow the UI to continue to be interactive. When the results are available, they should just appear automatically. However, it now has a different problem – you can submit *multiple requests*! To fix this, we will capture the result key in a property and check that property before allowing the **RunBlast** command to execute.
   1. The property type should be a string since that is the key returned.
   2. It should be set to a non-null value when a request is running, null otherwise.
   3. Add a call to **CommandManager.InvalidateRequerySuggested()** in the setter for the property. This will ensure that all commands are re-evaluated when this property changes so WPF knows the command state has changed.
   4. Here is the property implementation:
   5. private string \_isRunning;
   6. public string IsRunning
   7. {
   8. get { return \_isRunning; }
   9. set
   10. {
   11. \_isRunning = value;
   12. OnPropertyChanged("IsRunning");
   13. // Force WPF to reevaluate commands
   14. CommandManager.InvalidateRequerySuggested();
   15. }
   16. }
8. Set the value when you start a request, and set it to **null** when the request completes.
9. Add a check in then **RunBlast** command delegate in the constructor to check the new parameter. It should be added to the second lambda expression:
   1. RunBlast = new DelegatingCommand(OnBlastSequence,
   2. () => SelectedSequence != null && SelectedBlastAlgorithm != null
   3. **&& IsRunning == null**);
10. Run the application again and test the results. The button should be disabled while a request is pending:

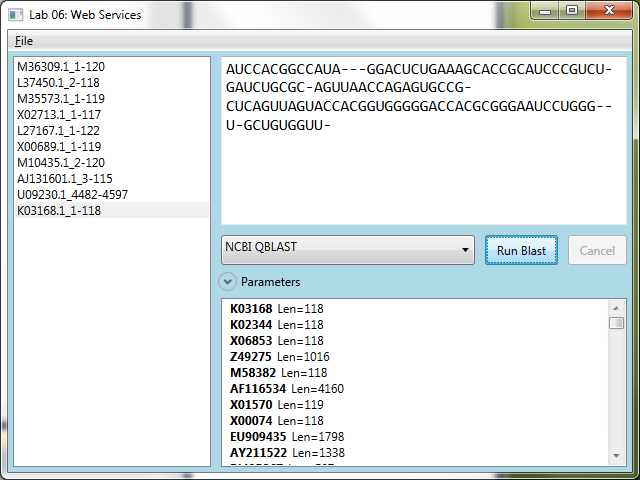


1. Next, we will add support to cancel the pending request. This turns out to be fairly easy. We just need to use the selected algorithm and call **CancelRequest** passing it the request key which we have stored in the **IsRunning** property. The code can be added to the existing **OnCancelBlast** stub at the bottom of the **MainViewModel** class. The following book keeping elements also need to be added:
   1. **CancelRequest** returns a Boolean. It should be tested to ensure it was canceled. If not, report a message to the user indicating the operation cannot be canceled. If it returns **true**, then make sure to set the **IsRunning** property to **null**. It is also helpful but not necessary to output a message and set **ResultHits** to **null** to clear any prior request.
   2. Depending on when the cancel request occurs, it is possible that a notification is being generated on the background thread. If that happens, the **IsCanceled** property will be **true** on the **EventArgs** passed to the completion method. Test for that case and ignore the results if that happens.
   3. The **Cancel** button should only be available when the **IsRunning** property is non-null – this can be ensured by adding a lambda to the **CancelBlast** delegate command in the constructor.
   4. public MainViewModel()
   5. {
   6. ...
   7. CancelBlast = new DelegatingCommand(OnCancelBlast, **() => IsRunning != null**);
   8. }
   9. ...
   10. void BlastServiceHandlerRequestCompleted(object sender, RequestCompletedEventArgs e)
   11. {
   12. IsRunning = null;
   13. **if (e.IsCanceled)**
   14. **return;**
   15. if (e.Error != null || !string.IsNullOrEmpty(e.ErrorMessage))
   16. {
   17. ...
   18. }
   19. private void OnCancelBlast()
   20. {
   21. **if (!SelectedBlastAlgorithm.CancelRequest(IsRunning))**
   22. **MessageBox.Show("Operation cannot be canceled.");**
   23. **else**
   24. **{**
   25. **IsRunning = null;**
   26. **MessageBox.Show("Pending request has been canceled!");**
   27. **ResultHits = null;**
   28. **}**
   29. }
2. Run the application and observe the results. It should cancel the request. However, we have introduced a bug. If you change the active service prior to hitting cancel, it will execute the cancel request on the wrong object!
3. To fix this, we will disable the selection while a request is running. The UI has already been setup to do this if you supply a public **Boolean** property called **CanChangeAlgorithm**. In reality, the state is driven off the **IsRunning** property; but that is a string, not a Boolean as is required by the **ComboBox.IsEnabled** property. So, we will generate a new property that is driven off our **IsRunning** state.
   1. Create the public Boolean property – the getter can simply check the **IsRunning** property for null. Since it’s driven off the **IsRunning** property, we can call **OnPropertyChanged** for this property in the **IsRunning** property setter:
   2. **public bool CanChangeAlgorithm**
   3. **{**
   4. **get { return IsRunning == null; }**
   5. **}**
   6. public string IsRunning
   7. {
   8. get { return \_isRunning; }
   9. set
   10. {
   11. \_isRunning = value;
   12. OnPropertyChanged("IsRunning");
   13. **OnPropertyChanged("CanChangeAlgorithm");**
   14. // Force WPF to reevaluate commands
   15. CommandManager.InvalidateRequerySuggested();
   16. }
   17. }

Run the application again and make sure the algorithm cannot be changed while we are pending a result.



If it is canceled or finishes, the combo box should allow selection again.



In the next task we will finish the lab by allowing custom properties and URLs so you can utilize the other BLAST algorithms.

Task 3: Customizing service settings

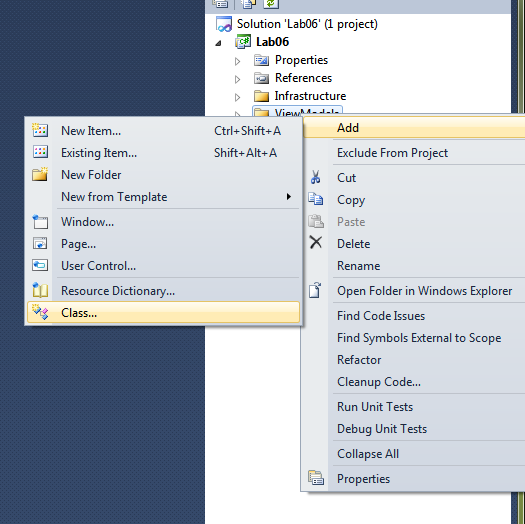
In this final task, we will finish out the **IBlastServiceHandler** support by allowing you to customize the specific parameters required for the algorithm being called. In addition, we will add support to change the URL being invoked and support using the browser proxy for connections.

Parameters to the BLAST service are supplied using the **BlastParameters** structure. This is passed to the **SubmitRequest** method when you initiate a BLAST request. Up to this point, we have been hardcoding the parameters for the NCBI variant of BLAST. To call other variations we need to change these parameters.

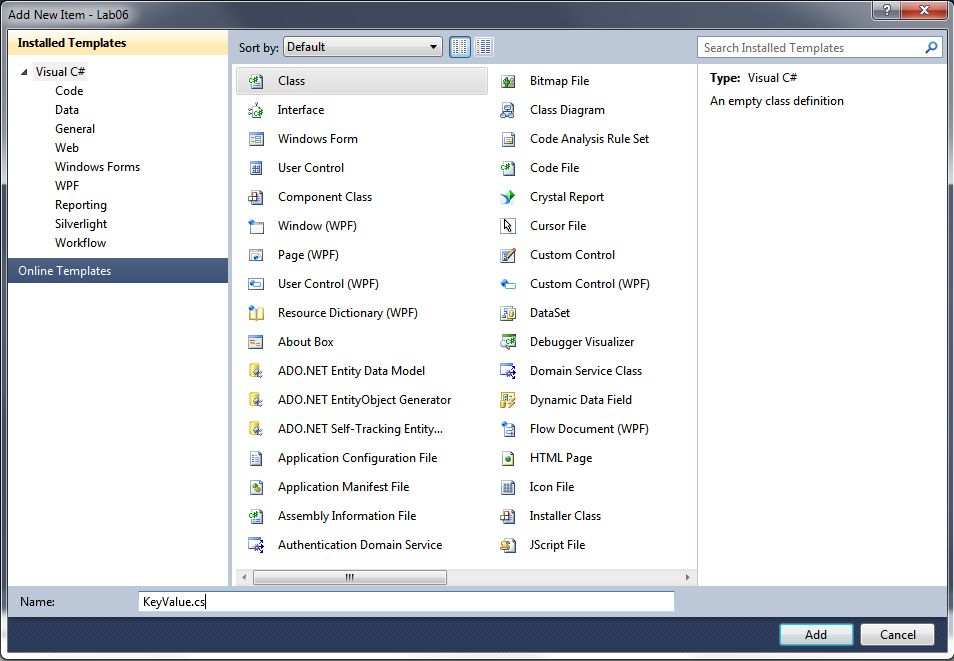
1. You can continue on from Task 2, or start with the starter solution – [Task3/before/Lab06.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp5_MBF.V2.zip\MBF.V2\Module%2006\Lab\Task3\before\Lab06.sln).

First, we will need a class to encapsulate the properties – there is a new **Tuple<T1,T2>** class in .NET 4 which would be ideal except it does not allow the values to be altered. The properties are read-only and only settable during creation. So, let’s create our own WPF-friendly version.

1. Right click on the **ViewModels** folder in the solution and select **Add | New Class**.

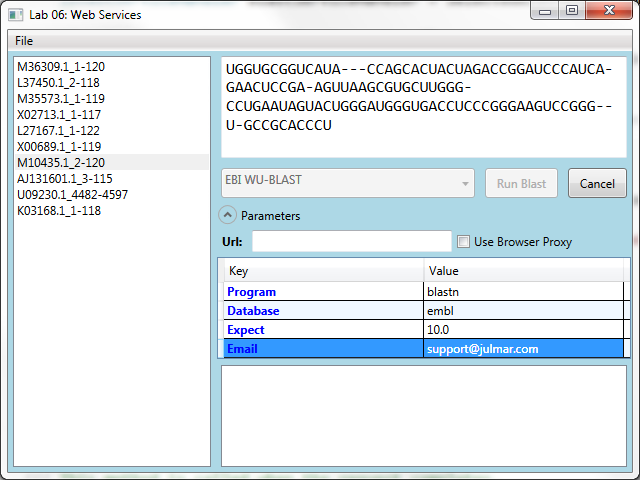


1. Name the class KeyValue.cs and click **Add** to add it to your solution.

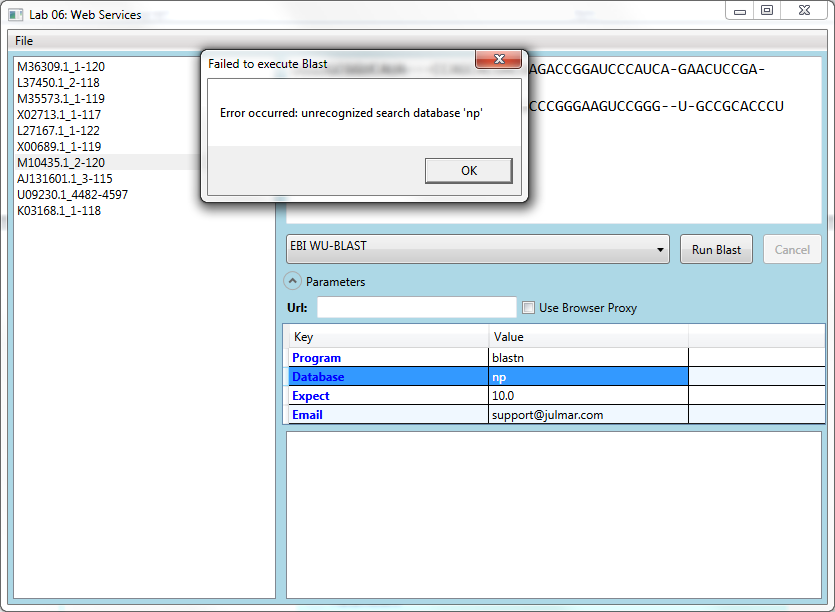


1. Change the base class to be **BaseViewModel** so you inherit property change notification.
   1. Add two new public properties to the class: **Key** and **Value**
   2. Make sure to call **OnPropertyChanged** for each one when it changes.
   3. Add a public constructor that takes a string key and optional value string as shown below:
   4. namespace Lab06.ViewModels
   5. {
   6. public class KeyValue : BaseViewModel
   7. {
   8. private string \_key;
   9. private string \_value;
   10. public string Key
   11. {
   12. get { return \_key; }
   13. set { \_key = value; OnPropertyChanged("Key"); }
   14. }
   15. public string Value
   16. {
   17. get { return \_value; }
   18. set { \_value = value; OnPropertyChanged("Value"); }
   19. }
   20. public KeyValue(string key, string value = "")
   21. {
   22. Key = key;
   23. Value = value;
   24. }
   25. }
   26. }
2. Switch to your **MainViewModel** class.
3. Add a new public property called **BlastParameters**. It should be an **IList<KeyValue>**.
4. Initialize the property in the constructor by setting it to an **ObservableCollection<KeyValue>**. Go ahead and add default values for **Program**, **Database**, **Expects** and a blank entry for **Email**. These are the parameters that almost every BLAST algorithm expects to see.
   1. public IList<KeyValue> BlastParameters { get; private set; }
   2. public MainViewModel()
   3. {
   4. BlastParameters = new ObservableCollection<KeyValue>
   5. {
   6. new KeyValue("Program", "blastn"),
   7. new KeyValue("Database", "nr"),
   8. new KeyValue("Expect", "10.0"),
   9. new KeyValue("Email"),
   10. };
   11. ...
   12. }
5. Next, populate the **IBlastServiceHandler** properties in your **OnBlastSequence** method using the collection you have defined and replace your current hard coded values. Only populate items that have values:
   1. IBlastServiceHandler blastServiceHandler = SelectedBlastAlgorithm;
   2. BlastParameters bp = new BlastParameters();
   3. **foreach (var item in BlastParameters.Where(kv => !string.IsNullOrEmpty(kv.Value)))**
   4. **bp.Add(item.Key, item.Value);**
   5. blastServiceHandler.RequestCompleted += BlastServiceHandlerRequestCompleted;

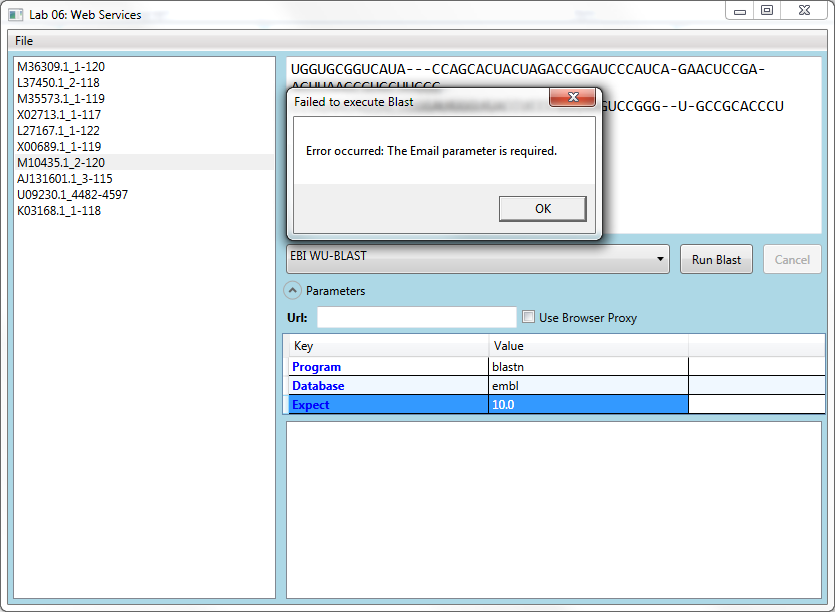
The UI is already hooked up to the **BlastParameters** property. It has a DataGrid that allows you to edit the options. You can change existing values, add new items, or delete existing ones by clicking and hitting the DEL key. Go ahead and run the application – then expand the **Parameters** section:



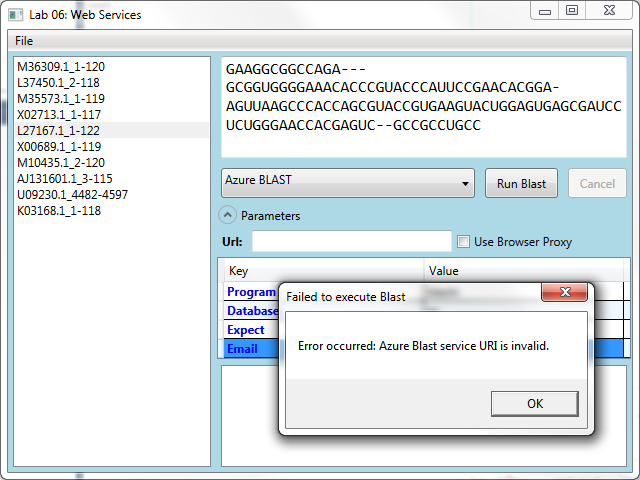
We can switch to the EBI version, by simply changing the database. If you type an invalid value, the error result should indicate what has happened:



If you omit required keys, it will also generate an error. For example, deleting the **EMAIL** key results in the following:



Last but not least, the Azure algorithm does not have a known URL. It requires that the URL be supplied as part of the call:



To support this, we will add the final bit of code. Notice the **Url** and **Use Browser Proxy UI** elements just above the grid. These expect to find the following two properties in our MainViewModel:

* 1. **Url** – a string with the full URL for the service.
  2. **UseBrowserProxy** – a true/false Boolean.

1. Go ahead and add the public properties to the MainViewModel and in the setter make sure to call **OnPropertyChanged**.
   1. private string \_url;
   2. public string Url
   3. {
   4. get { return \_url; }
   5. set { \_url = value; OnPropertyChanged("Url"); }
   6. }
   7. private bool \_useBrowserProxy;
   8. public bool UseBrowserProxy
   9. {
   10. get { return \_useBrowserProxy; }
   11. set { \_useBrowserProxy = value; OnPropertyChanged("UseBrowserProxy"); }
   12. }
2. In the **OnBlastSequence** handler, fill in the **IBlastServiceHandler.Configuration** properties. We want to set the Uri and **UseBrowserProxy** properties if we have valid data.
   1. BlastParameters bp = new BlastParameters();
   2. foreach (var item in BlastParameters.Where(kv => !string.IsNullOrEmpty(kv.Value)))
   3. bp.Add(item.Key, item.Value);
   4. **if (!string.IsNullOrEmpty(Url))**
   5. **{**
   6. **blastServiceHandler.Configuration.Connection = new Uri(Url);**
   7. **if (Url.StartsWith("https"))**
   8. **blastServiceHandler.Configuration.UseHttps = true;**
   9. **}**
   10. **blastServiceHandler.Configuration.UseBrowserProxy = UseBrowserProxy;**
3. With this in place, run the application and try out the Azure Blast service. You will need to get the final URL from the instructor as it is a temporary service setup specifically for the class. You can also change the invocation for the other (NCBI or EBI) services as well to try this out.

That completes the Web Services lab – you can get a finished solution from [Task3/after/Lab06.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp5_MBF.V2.zip\MBF.V2\Module%2006\Lab\Task3\after\Lab06.sln).