IRC Data Manager

This development was suggested by our industry partners in hopes of being able to dump Big Data to our labs minimizing the transmission method using e-mail or FTP.

Overview

The IRC Data Manager is a package that contains two programs: an uploader and a retriever. These two programs serve essentially as a wrapper to database API's. The uploader was initially designed to continuously fetch data from OPC servers, particularly OPC-DA used by the *FluidMechatronix*. The retriever was designed to allow non-database users to browse through data in databases and fetch the data they need.

The uploader program was originally written in Java using the open source library JEasyOPC. The library was able to read the 5698 tag output from FluidMechatronix in approximately 2 second intervals. But due to poor type conversions and false tag headings (false negative quality) given by the library, the project was then rewritten in C# which allowed less than 1 second retrievals and no false negative quality readings.

In the lab we have: *Cassandra, mongoDB, and MySQL* servers. More about the findings in using these different databases in the database section in the documentation.

App design

The programs are WPF applications. To decouple components that make up a user interface for easier debugging and more manageable code, it's best to use the MVVM pattern designed for WPF applications. MVVM is some what similar to MVC, but Here's a really great blog where you can learn the design pattern. To summarize:

Component	Function	File type
View	Responsible to display data stored in *models* through a user friendly interface	.xaml
Model	Responsible for templating data that needs to be displayed. Implements INotifyProperty to let view automatically update whenever data in the model is updated.	.cs
ViewModel	Links together the *view* and *model*. Removes application logic or code behind from *views* so that *views* can be changed regardless of the model	.cs

The core component in MVVM is *Data Binding*. Data binding allows users to access and alter the data model through the view. It can be described as two way publish, subscribe design pattern.

In these programs, the **main caller is located in App.xaml and the code behind App.xaml.cs**. These two programs have the DispatcherUnhandledException event handler to handle any global exception.

Material Design

The programs needs to be user friendly as this is the application that our lab members will use when we have the dedicated database server setup. The easiest way to style a WPF app is to implement Google's *Material Design*. They include pack icons and animated buttons. It is worth noting that **using the pop-up box to display a data grid/table that contains many rows is very slow**, it was for this reason that the *AddDataViewDialog* is displayed in a different window to *mimic a pop up*, more on this later.

WPF Shared Library

The WPF Shared Library contains two classes that can be used in any WPF application to implement the MVVM pattern. The RelayCommand is an implementation of ICommand that can be used on buttons or key bindings instead of adding the application logic in the view's code behind (.xaml.cs). These commands are usually found in the view model, and the behavior of the command can be defined as in the example below:

```
new RelayCommand(parameter => {
    // do something with parameter
    // or call a function passing the parameter if needed
});
```

Retriever - IRC Core

The IRC Core resembles the core services offered by inmation. The core is responsible for displaying and retrieving data from these databases. Each of these databases should wrap a common database abstract class so new databases can be supported just by installing the API's package and implementing the same database abstract class.

Code walkthrough

Main interface

In App.xaml.cs the OnStartup override method initializes the main window (should really be called main view) and associates the main view model as the data context for main window's data binding. The main user interface is designed in *MainWindow.xaml*. Here the MainWindow binds to the `DataSources` property in the MainViewModel. And for each `DataSource` in MainViewModel, there will be a view that will be used to display information related to it.

```
<ItemsControl ItemsSource="{Binding DataSources}">
    <ItemsControl.ItemsPanel>
        <ItemsPanelTemplate>
            <StackPanel Orientation="Vertical"/>
        </ItemsPanelTemplate>
    </ItemsControl.ItemsPanel>
    <ItemsControl.ItemTemplate>
        <DataTemplate DataType="{x:Type ds:DatabaseSource}">
            <views:DatabaseView/>
        </DataTemplate>
    </ItemsControl.ItemTemplate>
</ItemsControl>
```

The user control, `ItemsControl` creates a view for each item in the bound item source. Here for each DataSource of type `DatabaseSource` will be assigned a *DatabaseView*. Each DatabaseView User Control will have their DataContext set to the DatabaseSource The ItemsPanelTemplate is used to determine the layout of a collection of these views, here all the views are stacked vertically.

AddDataSourceDialog

To add a database source, the AddDataSourceDialog is created and a new window (acts as a dialog window) is shown. The dialog is shown using the static method `Dialog.Show()` . The dialog results can be retrieved by from the DialogClosingEventHandler passed on to the method. The DataSourceDialog is designed to be modular for future data sources such as data lakes which may or may not require authentication. New data sources can be added by adding it to the AddDataSourceDialog constructor

```
public AddDataSourceDialog() : base()
{
    SupportedTypes = new List<string>
    {
        "Databases",
    };

    SelectedType = SupportedTypes[0];
}
```

and then returning new instances through the DataSourceFactory

```
public static class DataSourceFactory
{
    public static BaseDataSourceDialog CreateDataSource(string type, AddDataSourceDialog mainDialog)
    {
        if (type == "Databases")
        {
            return new AddDatabaseSource(mainDialog);
        }
        throw new NotImplementedException();
    }
}
```

Supporting new databases

To support a new database API, the database API should be split into three sub classes: The database connection, database space and database collection. The database connection is responsible for generating handles to databases, the database space is responsible for generating handles to collections, and database collection is reponsible for retrieving data from those collections.

DatabaseSource

All datasources should inherit the DataSource abstract class. The abstract class simply contains a single public property `Label` used to identify a datasource. A single database server can host multiple databases, and tables or collections in those databases. I have decided to define the hierarchy as follows: DatabaseSource -> Database Collection -> DatabaseSpace. For example:

```
+-- DatabaseSource
|   +-- DatabaseSpace
|   |   +-- DatabaseCollection
|   |   +-- DatabaseCollection
|   +-- DatabaseSpace
|       +-- DatabaseCollection
+-- DatabaseSource
    +-- DatabaseSpace
    |   +-- DatabaseCollection
    +-- DatabaseSpace
        +-- DatabaseCollection
        +-- DatabaseCollection
        +-- DatabaseCollection
```

A DatabaseSource would be a single connection to a server, and each child (space or collection) would have the appropriate handle derived from the connection. Because views can't be derived I decided to just duplicate the view for a database source and renamed the data bindings accordingly.

Most database connections can be established by establishing: the address of the server and the appropriate user credentials that can be used to access that database. For database servers like mongoDB and MySQL where users are uniquely identified by the source (e.g. authentication database or IP address), this information can be passed on in the username column and parsed within the API wrapper. For example in mongoDB, the authentication database associated with the user is required, so I have decided to parse the username input as `<authentication database>/<username>`.

DatabaseSpace

A database space refers to a single database in a database server. It is responsible for generating collection or table handles contained in this database.

DatabaseCollection

A database collection refers to a single collection inside a database. It is reponsible for retrieving data through API's and generating data models to be displayed in the user interface. For an API to implement a DatabaseCollection, the wrapper must implement:

Methods	Function	Return
`ListData()`	Retrieves a table containing the columns: Include(bool), Tag(string), Type(type or string). The type column value should be the .NET equivalent type (e.g. System32.Int32) for an integer which can be found using the `.GetType()` method.	DataTable
`GetDataModel()`	Returns a new DataModel with the specifications given as the argument. E.g. build a plot model for the following tags: FluidMech_Output_TempFlow, FluidMech_Output_TempTank. Ideally the data model should be showing the latest data therefore try to look up how to retrieve the latest data using the API (for mongoDB, it was `.sort({$natural: -1})`).	DataModel
`Update()`	This updates the data held by the data model with the latest data. The `GetDataModel()` doesn't have to call update, as each DataModel in the database collection is automatically updated every second in the code section given below.	void

```
// runs in separate thread to update in the background
new Thread(() =>
{
      while (true)
      {
          foreach (DataModel model in DataModels)
          {
              if (model.IsLive)
              {
                  Update(model);
              }
          }
          Thread.Sleep(1000);
      }
  }) { IsBackground = true }.Start();
```

The IsBackground option for the Thread is to allow the program to terminate and release resources when it is the only living thread, i.e. when the window is closed, the program won't run in the background. These methods should be implemented as asynchronous functions (hence the *Task* return objects). Whenever possible, if the API supports asynchronous queries, then use them as the data size may be big and not fit in memory.