# The need.

How many times do we need to import from a delimited or a fixed length file. Each time invariably we do a custom implementation for each file format that we wish to import or process.

We decided to try and build a generic file parser that would make it easy to import from any delimited or fixed length format, and be flexible enough to allow custom processing of imported rows, custom validation etc.

# The solution.

We created an AbstractFileParser from which it is easy to derive and implement a custom file parser. The parser uses concrete classes and attributes to define the structure of the data. For example, you can define a header row using the following

|  |
| --- |
| [ImportType(ImportTypes.Delimited)]  public class DelimitedTestHeader  {  [ColumnIndex(1)]  public int NumberOfRecords { get; set; }  [ColumnIndex(0)]  public DateTime Date { get; set; }  } |

Each class is painted with an ImportType attribute. This defines the class as a Delimited or FixedLength class. (Not sure why one would do this, but this approach also gives you flexibility to have a header and footer that is delimited but data rows that are fixed length.)

Each field must be painted with the ColumnIndex attribute. This defines the order of the columns. Optionally you can also decorate each field with the IsRequired and the FieldLength attributes. The FieldLength attribute is used for fixed length files and will be ignored by the parser if the class is marked as delimited.

When deriving from the abstract class all you have to implement are the following

|  |
| --- |
| protected override void ProcessHeader<T>(T header)  {  DelimitedTestHeader hdr = header as DelimitedTestHeader;  *…… your logic here…*  }  protected override void ProcessData<T>(T data)  {  DelimitedTestData record = data as DelimitedTestData;  *…… your logic here…*  }  protected override void ProcessFooter<T>(T footer)  {  DelimitedTestFooter record = footer as DelimitedTestFooter;  *…… your logic here…*  } |

The neat thing about this is that in your processing logic you are dealing with a strongly typed class. The parser will parse the row and return the data in the strongly typed class.

You can also have custom validation in your derived implementation. The parser will validate things like type errors, missing required fields. Too few fields etc. Any custom business validation you wish to implement you do by overriding one of the following...

protected virtual List<string> ValidateHeader<T>(T header)

protected virtual List<string> ValidateData<T>(T data)

protected virtual List<string> ValidateFooter<T>(T footer)

For example:

|  |
| --- |
| protected override List<string> ValidateData<T>(T data)  {  var errors = new List<string>();  DelimitedTestData record = data as DelimitedTestData;  if (DateTime.Now.CompareTo(record.ActionDate) > 0)  {  errors.Add("Action date may not be in the past.");  }  return errors;  } |

# Notes / Comments

* Enumerations can be parsed from the string or integer value. For an enumeration   
  public enum EnumTest

{

Male = 0,

Female = 1,  
}  
The data in the import file could be either “Male” or “1”.

* The delimiter can be specified and if not the parser will figure out what delimiter has been used.
* The parser takes care of delimiter characters inside single or double quotes. So if you delimiter if a comma and a row contains the following data  
  1,1975-09-10,Frank,”Field Heights, 12”,some more data  
  It will ignore the comma in column 4.
* The test project contains various derived implementations showcasing and testing the various features.
* You can opt to process the whole file and have all the data returned to you as one large graph. However sometimes the import file is too large for this and you want to process a row at a time and not hold all the data in memory. Use shouldReturnDataCollection=true in the constructor to return all the data.
* You can specify to stop on error or to continue and import all the rows that are error free and get a validation report on the rows that had errors.
* Please let us know of any cool extensions, improvements (refactoring) as well as any bugs. We’ve used this in two projects to import two different data types and so far, it’s worked great. But there’s invariably a gremlin somewhere

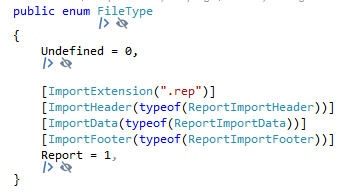
# Auto Documentation

The Data Contracts can easily be managed through a centralized File Enum class that is decorated with descriptive attributes. One enum class can drive your allowed files and be used to generate auto documentation.

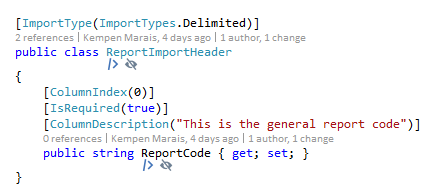
**ENUM that contains all the supported import types**

With one command, you can get a comprehensive overview of all your import files (Use for auto documentation, valid file whitelisting on import, etc.)

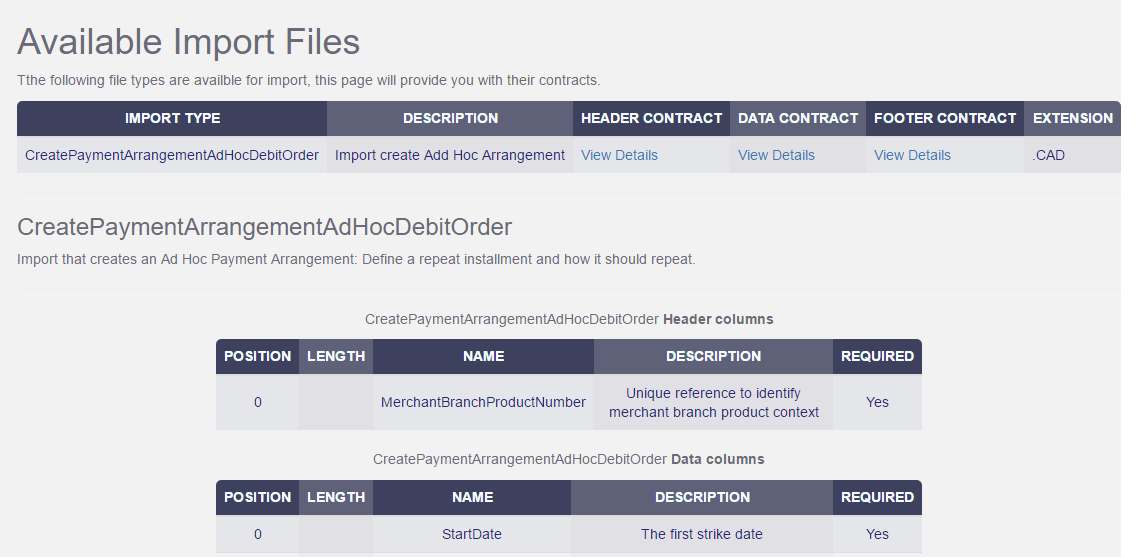




All these attributes that describes the import process is also available on request in the *GetAllFileDefinitions* call so it is easy to get an real-time overview of all your import details.



After calling *GetAllFileDefinitions* it is a simple task to create a basic razor view page provides auto documentation.



# Future extensions.

* Exporting.
* Master detail rows. For example if the data definition contains the following  
   [ChildData]  
   public IEnumerable<MyChildDataDefinition> ChildRows { get; set; }

(or just one Child row if you omit the IEnumerable)