# INTRODUCTION

This example was inspired by a recent job interview question. This was the original question:

*Write a simple Email app that uses Dependency Injection.*

*Use the following struct and interfaces in your solution:*

*public struct ParsedName*

*{*

*public string First { get; set; }*

*public string Last { get; set; }*

*}*

*public interface INameParser*

*{*

*ParsedName ParseName(string input);*

*}*

*public interface IEmailSender*

*{*

*void SendEmail(string to, string from, string subject, string body);*

*}*

I provided a simplified answer and moved on. After the interview, I returned to the question and created an application that is more thorough.

# SUMMARY

The purpose of this application is to demonstrate Dependency Injection. This is accomplished by creating a hypothetical SMTP, mass mailing application.

The application creates a custom message body based on a collection of name strings. The custom message body also contains the names of the classes used to generate the message and send the message. Displaying class names, demonstrates .Net reflection and confirms that the correct classes were injected into the application interfaces.

Once the custom message body is generated, the application passes the custom message body to the appropriate email sender. The email sender then send’s the message.

# CONDITIONS

1. This is a simple C# console application. All of the code is included in a single file. Running the application is as simple as creating a console application in Visual Studio and replacing the Program.cs file with the example code.
2. The structure of this application is intentionally simple in order to not obfuscate the DI example.
3. There is no exception handling in this application, such as TRY/CATCH/FINALLY or USE(…){…}. This is intentional because it would obfuscate the DI example.
4. This application does not send a real Email message. The message is ‘sent’ to the Console. It is reasonably simple to upgrade this app to send a real message. At least two email services would be required, such as Gmail and Hotmail.
5. The ‘To’, ‘From’, and ‘Subject’ text is static. DI could easily be used to automate these values.
6. The EmailSender is arbitrarily selected based on the length of the last name. See ‘Final Thoughts’ for enhancements.

# APPLICATION FLOW

The main application flow is contained in Main( ). It is easy to see what is happening. Main( ) contains four code blocks which perform specific tasks.

**Block One**

This code block builds a collection of name strings that will be parsed. This is the data that will be used to generate the body of the email message.

**Block Two**

This code block decides which name parser class will be used and initializes the INameParser interface. The correct parser is selected based on the first and last name form. See the GetNameForm( ) description below. At the end of the block the correct version of the method ParseName( ) is called. The results are returned to a ParseName struct.

**Block Three**

This code block decides which mail sender class will be used and initializes the IEmailSender interface. The choice is based, arbitrarily, on the length of the last name. See ‘Final Thoughts’ for enhancements to selecting the mail sender.

**Block Four**

This code block builds the message and sends it. The BuildMessage( ) method creates the body of the message. See BuildMessage( ) description below. At the end of the code block, the message is sent from the correct mail sender by the SendMail( ) method.

# INTERFACES & CLASSES

**FirstLastParser : INameParser**

* The implementation of ParseName( ), in the FirstLastParser class, parses a name string with the following pattern: “John Smith”.

**LastFirstParser : INameParser**

* The implementation of ParseName( ), in the LastFirstParser class, parses a name string with the following pattern: “Smith, John”.

**MyEmailSender : IEmailSender**

**YourEmailSender : IEmailSender**

The implementation of SendEmail( ) is identical for both classes. The method constructs the message to be sent. The message contains:

* + To
  + From
  + Subject
  + Body. The body of the message is created by the BuildMessage( ) helper method. See below for a description.

# HELPER METHODS

**GetNameForm( )**

* This method inspects each input name and returns the ‘form’ of the name as a string: “LastFirst” or “FirstLast”. It’s sole purpose is to help choose the correct name parser class to instantiate.

**BuildMessage()**

* This method creates the message body with this data:
  + Original input name
  + Parsed first name
  + Parsed last name
  + Name of the class that was used to parse the input name
  + Name of the Email Sender that will be used

# FINAL THOUGHTS

This application arbitrarily selects the EmailSender based on the length of the last name.

An enhancement could select the EmailSender based on geographic or demographic properties. These properties would have to be added to the input name collection.

An advanced enhancement could send the messages in parallel. A task would be started for each EmailSender instance. Each name in the name list would be passed to an available EmailSender task. When the task completes, another name in the name list would to assigned to the task. This would continue until all of the names in the list had been sent.

My goal of publishing this document and the code is two fold:

* Demonstrate my understanding of Dependency Injection by applying it in a working application
* Help people trying to understand Dependency Injection with a simple working example.