Dist. Sys. Pipes and Filters

In this lab you'll be looking at the concept of Pipes and Filters. This is an architectural pattern extremely common in many types of distributed system.

There will be no networking involved in this lab. Instead we are considering the concept and producing a custom pipe and filter mechanism using C# to investigate the pattern.

We will start off by following this flow:

Pipes and Filters is a pattern used when you want to systematically pass data to one or more sequential functions which each perform some operation on the data before it is output.

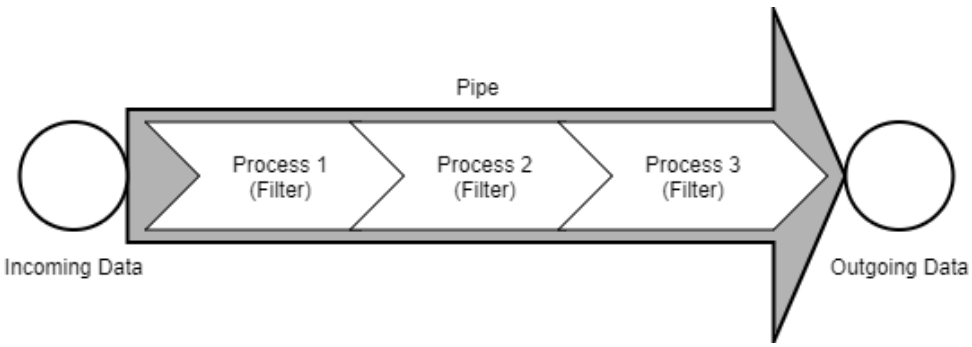
Pipes and filters exist at many layers in networking and are an essential option for separation of concerns. This is to say that operations which are not directly connected should not be bundled together as to do so would make it difficult to change or replace parts of the system in isolation and problematic to distribute the processes across different systems for, for example, the purposes of scaling up.

If we consider a basic pipe and filter (option 1):

In this example, data is put into the first pipe at one end, passing through a filter and into a subsequent pipe before being output.

We can therefore think of this as two pipes with one filter. This follows the concepts that we might come to expect from the real world, e.g. 'Waste water flows through a pipe to a water treatment plant, where it is processed into clean water, which flows through a pipe to a reservoir". This same theory can be used for networked systems where the pipe is a mechanism for data transfer between components. This type of pipeline makes it much easier to logically or physically separate filters so that they run in different operating system processes or across a physical boundary.

However, in software, pipelining may be less rigid. Rather than a transport mechanism we could instead consider our pipe to be a collection of sequential processes and a means to call each of these processes in turn (option 2):



Pipes and filters give us benefits, no matter which option we choose:

- The filters are independent so processes can be easily added, replaced, removed or altered in isolation from one another.
- We can choose different pipelines depending on the message. A routing component can choose a pipeline specific to the type of data being sent.

However, there are also some drawbacks:

- Filters can only be applied sequentially. There is no scope for asynchronous operation or to skip a filter once data is in a pipeline.
- Filters cannot (usually) talk to one another. Typically, the only way to do this is to add data to the message. This means that the data size may grow as you add more filters.
- Some filters may have dependencies on preceding filters. E.g. Filter A must have converted binary to hex before filter B can encrypt the data supplied as hex. This can mean that modifications to the pipeline ordering or the filter functionality may have knock-on effects for other filters.

Option 2 gives us some benefits over option 1:

- Pipe boundaries tend to cause slowdowns and maybe even data loss. Removing boundaries by treating the collection of operations as a single pipe can increase throughput.
- The pipe acts as the caller for each process, meaning that additional functionality can be added to the pipe itself. Rather than simply a transport mechanism it could theoretically handle errors, perform routing (to skip certain filters) or offer common data storage/access/sharing mechanisms rather than requiring all data to be passed with the message.
- We can perform actions which may expose data that we do not want to be intercepted. For example, a filter could perform decryption but if it did so using option 1 then theoretically it could be intercepted as it moves to the next filter. With option 2, the decrypted data is all contained within a single environment.

But there are also negatives to using option 2 over option 1:

- Each filter is likely more tied to the other filters, especially if they all exist within a common, shared environment that they all use. This reduces independence and may make it more difficult to swap and change filters.
- Without unique pipes with specific boundaries it becomes more difficult to distribute each component to improve scalability of the overall system. This may necessitate a cap on the number of concurrent users in order to prevent slow-down of the system due to overloading. This is especially important for large or resource-consuming filters.

Our problem is that we have a sequence of incoming messages (requests), which have come from users and are to be handled by our service, and outgoing messages (responses), which have come from our service and are going to our users.

We will use a basic (and insecure) user authentication method (as this is just an example) so that we know who the user is.

We also want our service to be able to receive and respond to ASCII messages sent as bytes, hex or binary. We also want it to be able to translate the response to return the message using the same formatting.

We could build this functionality into the service endpoints but then we may need to replicate this functionality many, many times for different endpoints. We also don't want to mix our business logic with message processing.

Before the request message gets to our service we need to perform some actions:

1. Check if a header is included which indicated a specific user ID. If so, update the server environment to indicate which user is interacting with the service
2. Check another header to translate the body of the message from either a byte array, hex or binary to a string

Once the response message leaves our service, but before it is dispatched to the user we are going to perform some other actions:

1. Translate the message body from a string to either a byte array, hex or binary as stipulated in the headers
2. Add a timestamp header to the outgoing message

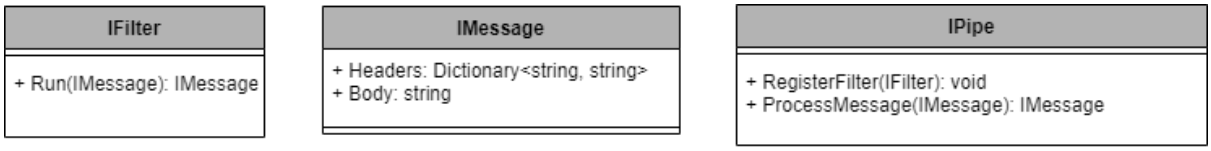
None of these filters will be particularly large or resource-consuming, and we want the data to stay within the same environment, so it seems appropriate to choose option 2 as defined in the previous section.

This technique is similar to the middleware and filter concepts used in ASP.NET.

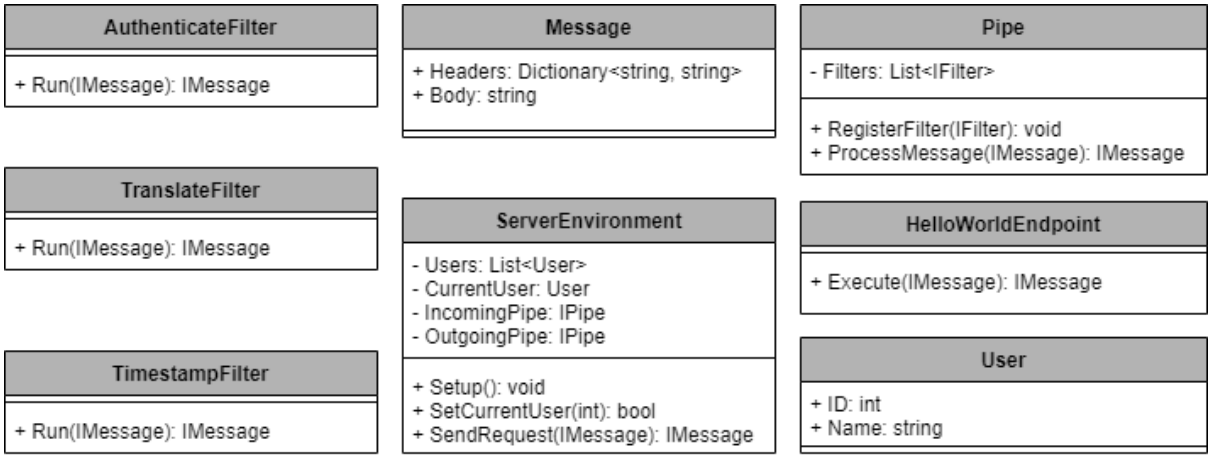
Alongside the pipes and filters, we will also use a basic 'client' and 'service' to act as the components on either side of the pipe.

Now we have an idea of what we want, let's consider an architectural design. We'll start by defining our interfaces. The main, crucial thing is that all of our filters have the same public method with which to call them. This means that we can add filters to the pipeline whenever we like.

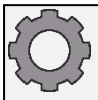
Here are the interfaces we will be implementing…



And the classes we will start off with…



At the moment the ServerEnvironment will only call a single endpoint. The 'Final Tasks' section identifies the modification of the solution to support multiple endpoints. If you feel confident in doing so, you may like to add an IEndpoint interface, and a Dictionary of IEndpoints to the ServerEnvironment class in preparation for that.

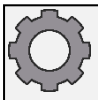


Download the lab file skeleton solution from Canvas and open it in Visual Studio.

Have a look through the files that are included. At the moment there are a number of errors. We'll be creating code to fix these. You should find a User and HelloWorldEndpoint class and a barebones ServerEnvironment class. Verify them based on the UML designs. Note that ServerEndpoint is static.

Have a look at Program.cs, it contains the entry point and a Client class. Investigate what you expect it to do once run.

First, we shall create the interfaces…



Right-click the Messages folder in the Solution Explorer and choose Add -> New Item. Select **Interface** and name it IMessage.cs.

- Add a 'Headers' public property of type Dictionary<string, string> with accessors:

```
public Dictionary<string, string> Headers { get; set; }
```

- and a 'Body' public property of type string with both get and set accessors too.

Add another interface to your project into the Filters folder. Call this one IFilter.

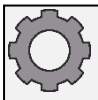
- Add the public 'Run' method signature from the interface definition above:

```
public IMessage Run(IMessage message);
```

Finally, add the IPipe interface to the Pipes folder.

- Follow the interface definition to create public RegisterFilter and ProcessMessage method signatures with the given parameters and return types.

Now let's make a concrete Pipe class that conforms to the IPipe interface…



Right-click the Pipes folder, choose Add -> **Class** and name it Pipe.cs.

- Modify the class definition to identify that it implements IPipe:

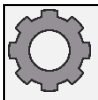
```
class Pipe : IPipe
```

You should notice that IPipe is now underlined in red. If you check the error you can see that this is because Pipe does not currently correctly implement the IPipe interface.

- There is a quick way to rectify this. Click on the red-underlined IPipe and press CTRL+. Choose **Implement Interface**.

Two methods will be created for you that throw NotImplementedExceptions at the moment.

- Add the private 'Filters' property to Pipe as defined in the UML. Then, add a constructor to Pipe and instantiate Filter to a new List<IFilter>();
- Modify the code inside RegisterFilter to add the supplied IFilter parameter to the end of the Filters List.
- Modify the code inside ProcessMessage, creating a loop to sequentially pass the supplied IMessage through each filter in the Filters List and return the message passed back by the final filter.



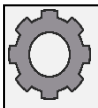
Add a new **Class** to the Messages folder and name it Message.cs.

- Modify the class definition to identify that it implements IMessage
- Implement the IMessage Interface.

If you did this automatically, you will need to remove the automatically generated get and set implementations to leave them with default behaviours.

- remove all instances of `=> throw new NotImplementedException()`
- add a constructor to Message and instantiate Headers to a new Dictionary<string, string>();

And for our final bit of class creation, let's make our three concrete filters…



Add a new **Class** to the Filters folder and name it AuthenticateFilter.cs.

- Modify the class definition to identify that it implements IFilter
- Implement the IFilter Interface.

The Run method in this class should look for a header in the message with the key 'User' and, if one exists, call the ServerEnvironment.SetCurrentUser method, passing the integer parsed from the header value. Look at the behaviour in the RequestHello method of the Client class inside Program.cs. On line 23 you can see this header is added before the message is sent.

- Add the behaviour described above to the AuthenticateFilter Run method.

Add a new **Class** to the Filters folder and name it TranslateFilter.cs.

- Modify the class definition to identify that it implements IFilter and implement the IFilter Interface.

The Run method in this class should look for a header in the message with the key 'RequestFormat' and, if one exists, translate the body of the message from the given format to an ASCII string. Look at the behaviour in the RequestHello method of the Client class inside Program.cs. On lines 26-33 you can this being set for the request.

The Run method should also look for a header in the message with the key 'ResponseFormat' and, if one exists, translate the body of the message from an ASCII string to the given format.

At the moment the client only sends and expects a byte string so we'll implement only this option.

- Using the code in the RequestHello method of the Client class inside Program.cs for reference, add the behaviour described above to the TranslateFilter Run method to allow it to convert to and from a byte string.

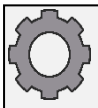
Add a new **Class** to the Filters folder and name it TimestampFilter.cs.

- Modify the class definition to identify that it implements IFilter and implement the IFilter Interface.

The Run method should add a header to the IMessage containing the key: "Timestamp" and the current DateTime.Now.ToString() value.

- Add the behaviour described above to the TimestampFilter Run method.

Now that we have all of the interfaces and classes we need, let's add code to the barebones methods in ServerEnvironment so it does what it is supposed to…



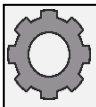
Open ServerEnvironment.cs

- Modify the code in the SetCurrentUser method to loop through the Users List looking for a user with the given id. If one is found, set CurrentUser to the User and return true. Otherwise return false.
- Add code to the botom of the Setup method:
    - Create a new instance of Pipe and set it as IncomingPipe
    - Create another new instance of Pipe and set it as OutgoingPipe
    - Create a new instance of AuthenticateFilter and add it to the **IncomingPipe** by calling RegisterFilter
    - Create a new instance of TranslateFilter and add it to the **IncomingPipe** by calling RegisterFilter
    - Create a new instance of TranslateFilter and add it to the **OutgoingPipe** by calling RegisterFilter
    - Create a new instance of TimestampFilter and add it to the **OutgoingPipe** by calling RegisterFilter

Review the code in the SendRequest method. At this point, all of this code should be error-free (no squiggly red lines).

Hopefully once you have completed the last step your solution is free from errors. If you are seeing any compilation errors, check the error message and, if that doesn't give away the answer immediately, refer back to the steps above.

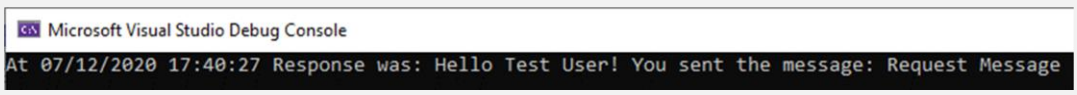
As a last resort I have also provided a link on Canvas to a working version of the solution for you to investigate. If you do use this example, make sure you look through it all so you fully understand what is happening at each step. Use this to inform your solution – don't just copy/paste.



Ensure there is a breakpoint on the first line of the Main Method inside Program.cs.

Run your solution and step through to track the progress of your message.

If all goes well you should be treated to a message similar to this:



If there are any errors, refer to the advice above.

Final Tasks

1. Modify your TranslationFilter to translate to and from binary and hex. Add new clients which use binary and hex values.

2. At the moment there is only one endpoint available. Add functionality to allow multiple endpoints:
    a. Add an IEndpoint interface with a method `public IMessage Execute(IMessage message)`. Ensure that HelloWorldEndpoint implements IEndpoint.
    b. Add a Dictionary of <string, IEndpoints> to the ServerEnvironment class.
    c. Ensure your clients add an "Endpoint" header that defines the endpoint to use.
    d. Add a check in part 2 of the `SendRequest` method inside the ServerEnvironment class to call the specific endpoint in the endpoint dictionary based on the header value.
    e. Add some new endpoints with new functionality.

3. Add some of your own filters.
    a. How about a filter that checks a 'status code' in the outgoing pipeline response headers? If the code is 1 (success) do nothing, if the code is 2 (format error) modify the response body to say "Error – the request format caused an error"... etc. Use the status code to decide whether to continue a pipeline/call the endpoint. You may also need to think about the order that you register your filters…

Outcome

By the end of this workshop you should have a good understanding of the Pipes and Filters pattern.

You should also have a better understanding of:

- some of the core concepts that are relevant for comprehension of the functionality in many existing web frameworks such as ASP.NET Core,
- some key ideas related to the coursework, such as converting strings based on Encoding.