Pipes and Filters

Exercise Notes

Introduction

- In this exercise you will create a Content Enricher

- Previously we forwarded a message from a Producer to a Consumer
  - The message travelled by an Exchange
  - An RMQ Exchange is a Dynamic Router
    - Or perhaps a direct/default exchange is a Recipient List and a topic exchange is a dynamic router

Notes

- We create an explicit Filter type, but that is actually unnecessary
    - A Handler can take a dependency on a Producer and send in response to actioning a message
    - Often understood as receive Command and raise Event/Document in response
    - Our use here is mainly to highlight the role being played, which is often elided when we think about a Hander raising an event in turn.
- Ask Ian to tell you about Clarissa, when you reach this point
    - We'll explain how it relates to messaging

C#

```csharp
public Task Run(CancellationToken ct)
{
    var task = Task.Factory.StartNew( action: () =>
    {
        ct.ThrowIfCancellationRequested();
        using (var inPipe = new DataTypeChannelConsumer<TIn>(_messageDeserializ
        {
            while (true)
            {
                var inMessage = inPipe.Receive();
                if (inMessage != null)
                {
                    TOut outMessage = _operation.Execute(inMessage);
                    using (var outPipe = new DataTypeChannelProducer<TOut>(_messasgeSerializer, _hostName))
                    {
                        outPipe.Send(outMessage);
                    }
                }
                else
                {
                    Task.Delay(1000, ct).Wait(ct); //yield
                }
                ct.ThrowIfCancellationRequested();
            }
        }
    }, ct
    );
    return task;
}
```

We are running a message pump

We receive on the in port

A handler transforms the code

We forward on the out port

Python

```python
def filter(cancellation_queue: Queue, input_class: Type[Request], deserializer_func: Callable[[str], Request],
           output_class: Type[Request], operation_func: Callable[[Request], Request], serializer_func: Callable[[Request], str],
           host_name: str= 'localhost') -> None:
    """
    Intended to be called from a thread, we consumer messages in a loop, with a delay between reads from the queue in order
    to allow the CPU to service other requests, including the supervisor which may want to signal that we should quit
    We use a queue to signal cancellation - the cancellation token is put into the queue and a consumer checks for it
    after every loop
    :param cancellation_queue: Used for inter-process communication, push a cancellation token to this to terminate
    :param input_class: What is the type of message we expect to receive on this channel
    :param deserializer_func: How do we serialize messages from the wire into a python object
    :param host_name: Where is the RMQ exchange
    :return:
    """

    with Consumer(input_class, deserializer_func, host_name) as in_channel:
        while True:
            in_message = in_channel.receive()
            if in_message is not None:
                with Producer(output_class, serializer_func) as out_channel:
                    out_message = operation_func(in_message)
                    out_channel.send(out_message)
                    print("Sent Message: ", json.dumps(vars(out_message)))
            else:
                print("Did not receive message")

            # This will block whilst it waits for a cancellation token; we don't want to wait long
            try:
                token = cancellation_queue.get(block=True, timeout=0.1)
                if token is cancellation_token:
                    print("Stop instruction received")
                    break
            except Empty:
                time.sleep(0.5)  # yield between messages
                continue
```

We are running a message pump

We receive on the in port

A handler transforms the code

We forward on the out port

JavaScript

```
Filter.prototype.filter = function(channel, inCb, outCb){
    var me = this;
    channel.prefetch(1);
    channel.consume(me.inputQueueName, function(msg){
        try {
            const request = me.deserialize(msg.content);
            const output = inCb(null, request);
            channel.ack(msg);
            channel.publish(exchangeName, me.outputRoutingKey, Buffer.from(me.serialize(output)), {persistent:true}, function(err,ok){
                if (err) {
                    console.error("AMQP", err.message);
                    throw err;
                }
                outCb(output);
            });
        }
        catch(e){
            channel.nack(msg, false, false);
            outCb(e, null);
        }
    }, {noAck:false});
};
```

We receive on the in port

A handler transforms the code

We forward on the out port