

# Datatype Channel

Exercise Notes

# Introduction

- In this exercise you will create a Datatype channel
- Previously we passed a string, but now we want to pass data
  - The message body is a data structure that we want to share
  - We now have distinguish Producer and Consumer
    - One serializes the other deserializes
  - We know what the schema is, because of the channel we received it on
  - In our case it is JSON, but it could be XML, Protobuf, Avro
  - The definition of the schema is communicated out-of-band
    - Via documentation such as AsyncAPI
    - Not show here

# Notes

- Is it a good idea to share a type when serializing code?
  - This is a form of coupling
    - Platform Coupling – both sides need so use the same language and framework
  - This may be fine if we ship both components as part of the same CI-boundary
    - For example, if we use the Task Queue pattern (covered in part 2) for background processing
    - Or, for example, if the two components are part of the same microservice boundary
      - Such as a web process receiving HTTP and a worker process receiving AMQP

C#

You'll need to serialize as well

We pass in a  
deserialization function

```
1 usage 2 iancooper
public DataTypeChannelConsumer(Func<string, T> messageDeserializer, string hostName = "localhost")
{
    messageDeserializer = messageDeserializer;
    // sr: guest pwd: guest port:5672 virtual host: /
    ConnectionFactory() { HostName = hostName };
    QueryEnabled = true;
    CreateConnection();
    .CreateModel();

    /* We choose to base the key off the type name, because we want to publish to folks interested in this type
       We name the queue after that routing key as we are point-to-point and only expect one queue to receive
       this type of message */
    var routingKey :string = typeof(T).Name;

    _channel.ExchangeDeclare(ExchangeName, ExchangeType.Direct, durable: false);
    _channel.QueueDeclare(queue: _queueName, durable: false, exclusive: false, autoDelete: false, arguments: null);
    _channel.QueueBind(queue: _queueName, exchange: ExchangeName, routingKey: routingKey);
}

/// <summary>
/// Receive a message from the queue
/// The queue should have received all message published because we create it in the constructor, so the
/// producer will create as well as the consumer making the ordering unimportant
/// </summary>
/// <returns></returns>
1 usage 2 iancooper
public T Receive()
{
    var result = _channel.BasicGet(_queueName, autoAck: true);
    if (result != null)
        return _messageDeserializer(Encoding.UTF8.GetString(result.Body));
    else
        return default(T) ;
}
```

We deserialize the  
message body

Python

You'll need to serialize as well

```
class Consumer:

    def __init__(self, request_class: Type[Request], mapper_func: Callable[[str], Request], host_name: str='localhost') -> None:
        """
        We assume a number of defaults: usr:guest pwd:guest port:5672 vhost: /
        """
        self._queue_name = request_class.__name__
        self._routing_key = self._queue_name
        self._mapper_func = mapper_func

        self._connection_parameters = pika.ConnectionParameters(host=host_name)

    def __enter__(self) -> 'Consumer':
        """
        text manager as resources like connections need to be closed
        f as the channel is also the send/receive point in this point-to-point scenario
        point-to-point channel

        connection = pika.BlockingConnection(parameters=self._connection_parameters)
        channel = self._connection.channel()
        channel.exchange_declare(exchange=exchange_name, exchange_type='direct', durable=False, auto_delete=False)

        self._channel.queue_declare(queue=self._queue_name, durable=False, exclusive=False, auto_delete=False)
        self._channel.queue_bind(exchange=exchange_name, routing_key=self._routing_key, queue=self._queue_name)

        return self

    def __exit__(self, exc_type, exc_val, exc_tb) -> None:
        """
        We must kill the connection, we chose to kill the channel too
        """
        self._channel.close()
        self._connection.close()

    def receive(self) -> Request:
        """
        We just use a basic get on the channel to retrieve the message,
        But what we get back is a byte array really, and we need to convert that to a string
        We ignored this in prior exercises, because 'it just worked' but now we care about it
        :return: The message or None if we could not read from the queue
        """
        method_frame, header_frame, body = self._channel.basic_get(queue=self._queue_name, no_ack=True)
        if method_frame is not None:
            self._channel.basic_ack(delivery_tag=method_frame.delivery_tag)
            body_text = body.decode("unicode_escape")
            request = self._mapper_func(body_text)
            return request
        else:
            return None
```

We pass in a  
deserialization function

We deserialize the  
message body

JavaScript



```
function Consumer(queueName, url, deserialize) {  
  this.queueName = queueName;  
  this.brokerUrl = url;  
  this.deserialize = deserialize;  
}
```

We pass in a  
deserialization function

You'll need to serialize as well

```
module.exports.Consumer = Consumer;  
  
//cb - the callback to send or receive  
Consumer.prototype.receive = function(channel, cb){
```

```
  var me = this;  
  channel.get(this.queueName, {noAck:true}, function(err, msgOrFalse){  
    if(err){  
      console.error("AMQP", err.message);  
    }  
    else if (msgOrFalse === false){  
      cb({});  
    }  
    else {  
      const request = me.deserialize(msgOrFalse.content);  
      cb(request);  
    }  
  });  
};
```

We deserialize the  
message body