Making MQ Application Development Easier

Mark Taylor
marke_taylor@uk.ibm.com
IBM Hursley
Presentation includes source code
Agenda

• Overview

• Learn-MQ

• Runtime access

• New Languages
  – Go
  – JavaScript
Developer engagement

- MQ can be hard to get going with for new developers

- How do we make it easier for new application teams to pick up MQ and integrate with their development practices?

- Not always been a product focus – spent more time on administration enablement
>50% coders have fewer than 5 years’ professional experience*

*https://insights.stackoverflow.com/survey/2018
To enable a user instructed to use MQ for the first time, to go from zero understanding to running a sample application in a sandbox environment with a fundamental understanding of MQ concepts in 2 hours.

To enable an application developer, instructed to use MQ for the first time, to go from zero understanding to writing their first MQ application in the language and environment of their choice within an afternoon.
LearnMQ

Finding it hard to get developers started with MQ?

Point them to:

developer.ibm.com/messaging/learn-mq

Totally new to MQ? Learn the basics

Step-by-step guide to getting up and running with MQ

Tutorials on building your applications
Candidate tutorials, samples & assets

- Connect Securely
- Write a production JMS app
- Write a production C# (.NET) app
- Write a publish Subscribe app
- Write a C app
- Deliver a microclimate app
- Deliver a Spring App

- Transactions
- Languages
- JEE
- Recipes & scripts
- REST
- Deployment
- Patterns

- Protocols
- Monitoring
- Advanced samples & templates
- Troubleshooting
- Testing
- Delivery Support
- ...

© Copyright IBM Corporation 2018
Not just about the language or education

• Developers also need easy access to interfaces – no matter their experience
• Do not want to have to install full products

• Many IDEs and build tools integrate with public repositories
• MQ Java interfaces now available from Central Repository (Maven)
  – No need to explicitly install or download
  – Just reference MQ jars in application configuration

```gradle
dependencies {
  compile("com.ibm.mq:com.ibm.mq.allclient:9.1.0.0")
}
```

```xml
Maven: pom.xml
<dependency>
  <groupId>com.ibm.mq</groupId>
  <artifactId>com.ibm.mq.allclient</artifactId>
  <version>9.1.0.0</version>
</dependency>
```
Easier ways to get started – Java

• Many Java developers use Spring
  – Can reduce amount of code needed
  – MQ JMS used with Spring for many years

• Spring Boot & Auto-configure give further code reduction
  – You can get a program running quickly
  – With default capabilities

• MQ now has Spring Boot starter
  – Versions for both Boot 1 and Boot 2
  – Source on github; jar on Maven Central

```groovy
build.gradle
dependencies {
  compile("com.ibm.mq:mq-jms-spring-boot-starter:+")
}
```

Easier ways to get started – runtime availability

• Docker container with full MQ server function
  – Creates sample objects for developers
  – Queues, topics, userids etc
• See github.com/ibm-messaging/container with several build options

• Redistributable Client packages now easier to access
  – To make it easy for developers to package standalone applications
  – Perhaps in a container
  – No login required to download
• See https://public.dhe.ibm.com/ibmdl/export/pub/software/websphere/messaging/mqdev/redist/
Language Interfaces to MQ
Multiple APIs and Protocols

IBM MQ supports multiple APIs and multiple client protocols. Both proprietary and open.

APIs: MQI, JMS, MQ Light, REST …

Protocols: MQ, AMQP, MQTT, HTTP

These support a wide range of application styles, from the simplest of messaging needs through to the most sophisticated

MQ is simply the broker of messages produced from any API, protocol or language

Messages from one application can be received by any other application, independent of API or protocol.
MQ Light APIs

- MQ Light interfaces in a variety of languages
  - Client source in GitHub
- And integrated with natural public repository
  - JavaScript: NPM
  - Java: Maven
  - Ruby: gem install mqlight
  - Python: pip install mqlight
- .Net information at
The classic MQI

• The full-function MQI has primarily been used from C, Java and COBOL

• Other languages for the MQI have been in the product but used less
  – PL/1
  – RPG
  – C++
  – etc

• On Distributed platforms many of these are built on top of the C library
  – The COBOL bindings, for example, are a small mapping layer
• The "pure" bindings (.Net, Java) have to reimplement all the protocols
Example C code

```c
char QMName[MQ_Q_MGR_NAME_LENGTH] = {0};
strncpy(QMName, "QM1", (size_t)MQ_Q_MGR_NAME_LENGTH);
MQCONNX(QMName,    /* queue manager */
     &cno,       /* connection options */
     &Hcon,      /* connection handle */
     &CompCode,  /* completion code */
     &CReason);  /* reason code */
```

Buffers for character strings

Fixed length – ensure no overflow

Multiple values returned via pointers

Function with no direct return code

None of these hard to work with, but not always natural

Designed this way for performance, commonality and extensibility
New interfaces - convergence of requirements

• Demonstrate monitoring capabilities of MQ with newer technologies

• Developers writing applications in a broader variety of languages
  – What do they know? What do they prefer to use?
  – Often do not have a choice as other aspects – toolkits, standards etc drive decisions

• Need to integrate with package managers for ease of access
Full MQI capability

- MQ Light APIs not rich enough for all the things I needed to do

- Simplified APIs are always prone to needing to expose just one more thing …

- Building on C MQI runtime libraries means not reimplementing protocol flows
  - Makes the language bindings thin

- Still allowing higher level, pattern-based components to be built
  - For example, a single API call to do request-and-wait-for-reply, or wait-then-send-reply which incorporates best practices for handling CorrelId and MsgId, or poison messages
New bindings released to GitHub

- Interfaces for Go and Node.js designed around full MQI function
- Provided as as-is open source
- Pull requests can be accepted
3rd party language interfaces

- Varying maintenance, capabilities, license
  
  Perl: search.cpan.org/dist/MQSeries/
  PHP: pecl.php.net/package/mqseries
  Python: pythonhosted.org/pymqi/
  Ruby: github.com/reidmorrison/rubywmq
  
- Also built on top of C MQI library
Monitoring with Prometheus

• Given a requirement to work with a monitoring solution

• Prometheus was one of the most popular metric stores (time-series DB)

• Standard toolkit for getting data to Prometheus was in Go
  – Collector program needed to be written in Go while also processing MQ messages
  – But we had no Go API for MQ …
  – There is now a Java Prometheus client, but too late for this requirement
What is Go

• Language from Google
• Often called golang (easier for searching!)
  • Removes some of the dangerous features of C like pointers
  • Fast compilation times
  – Distinctive approaches to particular problems:
    • Built-in concurrency primitives (channels, go-routines)
    • Implicit object class inheritance
    • Toolchain produces statically linked native binaries without external dependencies.
  – Make the common things for common patterns easy
• Standard external repository managers – particularly github
  – You can easily exploit these packages in your own programs
• Good set of standard packages
Where is it used

• Many projects in infrastructure and systems

• For example,
  – Docker
  – Hyperledger
  – Prometheus

• And a large set of toolkits to help build other projects
The usual starter

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World")
}
```

"import" refers to standard libraries or to automatically-installed/downloaded packages

Indentation not critical but formatter applies it consistently
Getting to MQ from Go

- Go includes a mechanism – "cgo" – for calling C libraries from Go programs

- Can refer to C headers and structures in Go programs
  - Making use of them, though, is a bit more tricky
  - Similar to writing a JNI layer for Java/C interaction
  - Explicitly state when code goes outside Go's safety boundaries

- The cgo system does not include standard C pre-processing.
  - No #ifdef for conditional compilation, though there are whole-file optional inclusions
  - It is possible to do some limited #ifdef at the top of the file for the C code
  - Makes it harder to share code between platforms and versions
  - Could be done via intermediate build step
The design

• Go deals with "packages"
  – Analagous to libraries in C, package in Java

• Created an ibmmq package that maps the MQI into and out of Go
  – Layered on top of the MQ C client interface
  – To build it requires the C client and SDK

• Functions (MQI verbs) and structures

• Constants made available in Go-native format

• Making the MQI look more natural to a Go programmer
  – Structures use Go strings
A Go function can return multiple results, one of which usually indicates errors

```go
MQOPEN(hConn, &hObj, &mqOD, options, &cc, &rc);
```

becomes

```go
hObj, err := qmgr.Open(mqOD, options)
```

where qmgr encapsulates the hConn
- and err encapsulates both cc and rc values
- "if err != nil …" for error checking (and note no parens on the if)

Based on MQI verb names rather than Java's `qMgr.accessQueue()` style
import (  
  "fmt"
  "os"
  "strings"

  "github.com/ibm-messaging/mq-golang/ibmmq"
)
Opening a queue

```go
mqod := ibmmq.NewMQOD()
openOptions = ibmmq.MQOO_OUTPUT |
               ibmmq.MQOO_FAIL_IF_QUIESCING |
               ibmmq.MQOO_INPUT_AS_Q_DEF
mqod.ObjectType = ibmmq.MQOT_Q
mqod.ObjectName = "SYSTEM.DEFAULT.LOCAL.QUEUE"
qObject, err = qMgr.Open(mqod, openOptions)
if err != nil {
    fmt.Println(err)
}
```

Initialise default values

Refer to constants

Simple string

Call MQ and get hObj
Opening a queue

```go
mqod := ibmmq.NewMQOD()
openOptions = ibmmq.MQOO_OUTPUT |
    ibmmq.MQOO_FAIL_IF_QUIESCING |
    ibmmq.MQOO_INPUT_AS_Q_DEF
mqod.ObjectType = ibmmq.MQOT_Q
mqod.ObjectName = "SYSTEM.DEFAULT.LOCAL.QUEUE"
qObject, err = qMgr.Open(mqod, openOptions)
if err != nil {
    fmt.Println(err)
}
```
Putting a message

```go
mqod := ibmmq.NewMQOD()
openOptions = ibmmq.MQOO_OUTPUT |
    ibmmq.MQOO_FAIL_IF_QUIESCING |
    ibmmq.MQOO_INPUT_AS_Q_DEF
mqod.ObjectType = ibmmq.MQOT_Q
mqod.ObjectName = "SYSTEM.DEFAULT.LOCAL.QUEUE"
qObject, err = qMgr.Open(mqod, openOptions)
if err != nil {
    fmt.Println(err)
}

mqmd := ibmmq.NewMQMD()
pmo  := ibmmq.NewMQPMO()
mqmd.Format = "MQSTR"
msg  := "Hello from Go"
buffer := []byte(msg)
err = qObject.Put(mqmd, pmo, buffer)
if err != nil {
    fmt.Println(err)
}
```

Go has `=` and `:=`

Some more strings

No need to say how long message is
Putting a message

```go
mqod := ibmmq.NewMQOD()
openOptions = ibmmq.MQOO_OUTPUT |
    ibmmq.MQOO_FAIL_IF_QUIESCING |
    ibmmq.MQOO_INPUT_AS_Q_DEF
mqod.ObjectType = ibmmq.MQOT_Q
mqod.ObjectName = "SYSTEM.DEFAULT.LOCAL.QUEUE"
qObject, err = qMgr.Open(mqod, openOptions)
if err != nil {
    fmt.Println(err)
}

mqmd := ibmmq.NewMQMD()
pmo := ibmmq.NewMQPMO()
mqmd.Format = "MQSTR"
msg := "Hello from Go"
buffer := []byte(msg)
err = qObject.Put(mqmd, pmo, buffer)
if err != nil {
    fmt.Println(err)
}
```

Some more strings

No need to say how long message is
Getting a message

```go
var datalen int
getmqmd := ibmmq.NewMQMD()
gmo := ibmmq.NewMQGMO()
gmo.Options = ibmmq.MQGMO_NO_SYNCPOINT |
    ibmmq.MQGMO_FAIL_IF_QUIESCING | ibmmq.MQGMO_WAIT
gmo.WaitInterval = 3000

buffer := make([]byte, 32768)
datalen, err = qObject.Get(getmqmd, gmo, buffer)
if err != nil {
    fmt.Println(err)
    mqret := err.(*ibmmq.MQReturn)
    if mqret.MQRC == ibmmq.MQRC_NO_MSG_AVAILABLE {
        err = nil
    }
} else {
    fmt.Printf("Got message of length %d: ", datalen)
    fmt.Println(strings.TrimSpace(string(buffer[:datalen])))
}
```

Where will message data go

Look at MQRC value
Getting a message

Where will message data go

Look at MQRC value

```go
var datalen int
getmqmd := ibmmq.NewMQMD()
gmo := ibmmq.NewMQGMO()
gmo.Options = ibmmq.MQGMO_NO_SYNCPOINT |
ibmmq.MQGMO_FAIL_IF_QUIESCING | ibmmq.MQGMO_WAIT
gmo.WaitInterval = 3000

buffer := make([]byte, 32768)
datalen, err = qObject.Get(getmqmd, gmo, buffer)
if err != nil {
    fmt.Println(err)
    mqret := err.(*ibmmq.MQReturn)
    if mqret.MQRC == ibmmq.MQRC_NO_MSG_AVAILABLE {
        err = nil
    }
} else {
    fmt.Printf("Got message of length %d: ", datalen)
    fmt.Println(strings.TrimSpace(string(buffer[:datalen])))
}
```
The mqmetric package

- Also in the repository is the *mqmetric* package to handle some PCF operations
- Comparable to the Java PCF classes

- Created to allow execution of basic queries and parse responses
  - Not all PCF element structures, just the ones needed for the monitoring agents
Grafana dashboard

MQ PROMETHEUS CAPTURE

Queue Activity

CPU

Log write latency

File system

© Copyright IBM Corporation 2018
What is missing

• Not all the MQI has been implemented

• Missing verbs: MQSET, MQxxxMP, MQCB/MQCTL, MQSTAT, MQBEGIN

• Not all the structure fields have been implemented (eg Distribution Lists)

• No message header generation or parsing except for some PCF
  – Structures like the RFH2, DLH headers
Current status

• Original repository now split to make it easier to get just the pieces you need

• **mq-golang** has the core MQI and PCF packages
  – Some sample code to demonstrate use of most functions
  – Assumes you already know the MQI principles from another language

• **mq-metric-samples** has Prometheus, Cloudwatch etc monitor programs
  – Along with a "vendor" tree
    • How dependencies can be managed within the Go ecosystem

• Currently being managed and enhanced by the MQ Cloud team
Future thoughts

• Further simplify build and distribution processes

• Add more focussed samples to demonstrate specific features
  – Equivalents of amqsput etc
  – Instead of one or two all-embracing demonstrations

• Reduce number of unimplemented verbs and fields

• (Internal) See what can be automatically generated to reduce effort of hand-crafted layers
https://github.com/ibm-messaging/mq-mqi-nodejs
What is Node.js

• Run-time environment to execute standalone JavaScript programs
  – Rather than embedded in HTML pages, executed by browser

• Main technical distinction: highly asynchronous, for event-driven programming
  – Lots of tasks handled via callback functions
  – All user code runs on a single main thread (no parallel execution)

• No real relation to Java apart from the name and some syntax

• Standard external repository manager – npm
  – You can (all too) easily reference and use these packages in your own programs
Where is it used

• Many users of "server" applications written in Node.js
  – https://www.netguru.co/blog/top-companies-used-nodejs-production

• Organisations listed include Netflix, ebay, Wal-mart, Uber, NASA
The usual starter

```
console.log("Hello, World")
```

Rather simple!
The usual starter and an example of callback

```javascript
console.log("Hello, World")
```

Rather simple!

```javascript
const fs = require('fs');

fs.copyFile('source.txt', 'destination.txt', function (err) {
    if (err) throw err;
    console.log('source.txt was copied to destination.txt');
});
```

Defines a function that is invoked after the copy has completed
The design

• One package, ibmmq, that exports the MQI
  – Structures, functions and constants

• Async model with exceptions as optional alternative
  – If no callback provided, then most functions indicate errors via exception
  – Some functions always require callback

• Most MQI calls are really synchronous
  – They run immediately to completion on the main execution thread
  – MQGET is different
Getting to MQ from Node.js

• Packages on npm enable access to C interfaces
  – "ffi" (Foreign Function Interface) is the equivalent of dlopen/dlsym
  – "ref" converts between JavaScript datatypes and raw byte buffers

• No direct use of C interface elements was used
  – All structure mappings created by hand
  – There are ways to import/convert C headers but those still need lots of manual fixup
  – Similar to a Java/JNI layer

• Did consider a C++ "Addon" which might permit more asynchronous MQI calls
  – But would probably not help with MQGET callbacks
  – And rather complex to write
MQGET

• Package provides Get() and GetSync() verbs, with unique GetDone()
  – GetSync() is common JS pattern

• GetSync() is a blocking wait
  – Not recommended in Node.js programs as it stops any other work being done
  – OK for when WaitInterval is zero

• Get() is an asynchronous operation
  – Callback returns data or failure
  – Keeps returning more messages until GetDone() is called – similar to MQCB model
  – Implemented as a polling MQGET because real MQCB/MQCTL could not be used
  – Poll intervals can be tuned

• Looking at a native async operation for some application patterns
Opening a queue

```javascript
mq = require('ibmmq');
MQC = mq.MQC;
...
var od = new mq.MQOD();
var qName = "SYSTEM.DEFAULT.LOCAL.QUEUE";
od.ObjectName = qName;
od.ObjectType = MQC.MQOT_Q;
var openOptions = MQC.MQOO_INPUT_AS_Q_DEF;
mq.Open(hConn, od, openOptions, function(err, hObj) {
    if (err) {
        console.log("MQ call failed: " + err.message);
    } else {
        console.log("MQOPEN of %s successful", qName);
        getMessages(hObj);
    }
});
```

- Access the package
- Access the constants within the package
- Work with strings and use the constants
- Defines a function that is invoked after the Open has completed
- Convenient pre-formatted error message
function getMessage(hObj) {
    var buf = Buffer.alloc(1024);
    var mqmd = new mq.MQMD();
    var gmo = new mq.MQGMO();
    gmo.Options = MQC.MQGMO_NO_SYNCPOINT | MQC.MQGMO_NO_WAIT |
                  MQC.MQGMO_CONVERT | MQC.MQGMO_FAIL_IF_QUIESCING;
    mq.GetSync(hObj,mqmd,gmo,buf,function(err,len) {
        if (err) {
            if (err.mqrc == MQC.MQRC_NO_MSG_AVAILABLE)
                console.log("no more messages");
            else
                console.log("MQ call failed: "+ err.message);
        } else {
            if (mqmd.Format=="MQSTR")
                console.log("message <%s>", decoder.write(buf.slice(0,len)));
            else
                console.log("binary message: "+ buf.slice(0,len));
        }
    });
}
function getMessages(hObj) {
    var md = new mq.MQMD();
    var gmo = new mq.MQGMO();

    gmo.Options = MQC.MQGMO_NO_SYNCPOINT | MQC.MQGMO_WAIT |
                 MQC.MQGMO_CONVERT | MQC.MQGMO_FAIL_IF_QUIESCING;
    gmo.MatchOptions = MQC.MQMO_NONE;
    gmo.WaitInterval = waitInterval * 1000; // from seconds

    // Tune down poll interval to 0.5s for demo
    mq.setPollTime(500);

    mq.Get(hObj, md, gmo, getCB);
}

function getCB(err, hObj, gmo, md, buf) {
    if (err) {
        if (err.mqrc == MQC.MQRC_NO_MSG_AVAILABLE)
            console.log("No more messages available.");
        else
            console.log("MQ call failed: " + err.message);
        mq.GetDone(hObj);
    } else {
        if (md.Format == "MQSTR")
            console.log("message <%s>", decoder.write(buf));
        else
            console.log("binary message: " + buf);
    }
}
# MQMD

This is a class containing the fields needed for the MQMD (MQ Message Descriptor) structure. See the [MQ Knowledge Center](https://www.ibm.com/support/knowledgecenter/SSSRTW) for more details on the usage of each field. Not all of the underlying fields may be exposed in this object.

## Constructor

```javascript
new MQMD()
```

This constructor sets default values for the object.

## Members

**AccountingToken : Buffer**

**Type:**

- `Buffer`

**ApplIdentityData : String**

**Type:**

- `String`

**ApplOriginData : String**

**Type:**

- `String`
Automatic installation during deployment

- Refer to ibmmq in your package and C runtime will be automatically installed

```
$ cat package.json
{
  "name": "amqsput",
  "version": "0.0.1",
  "description": "Demo MQ API",
  "main": "amqsput.js",
  "dependencies": {
    "ibmmq": ">=0.7.0"
  }
}
```

```
$ npm install
... (messages show install of other pieces) ...
> ibmmq@0.7.0 postinstall
/tmp/node_modules/ibmmq
> node postinstall.js

Downloading IBM MQ Redistributable C Client runtime libraries - version 9.0.5.0
Unpacking libraries...
Removing 9.0.5.0-IBM-MQC-Redist-LinuxX64.tar.gz
amqsput@0.0.1 /tmp
  └─┬ ibmmq@0.7.0
     ├─┬ ffi@2.2.0
     │ └─┬ debug@2.6.9
     │     └─┬ bindings@1.2.1
     │         └─┬ debug@2.6.9
     │             ... (more about the dependency tree)
```
Containers

- The samples include a Dockerfile showing how to build a container with just your Node.js program in it
- And how to configure client connectivity to a queue manager

```
$ cd node_modules/ibmmq/samples
$ ./run.docker
Sending build context to Docker daemon  87.04kB
Step 1/13 : FROM debian:jessie-slim
  ---> f1ff1c889d54
Step 2/13 : ENV NODE_USER app
...
Step 13/13 : CMD node amqsput ${DOCKER_Q} ${DOCKER_QMGR}
  ---> Using cache
  ---> 0cd6e7086633
Successfully built 0cd6e7086633
Successfully tagged mq-node-demo:latest
Sample AMQSPUT.JS start
```
Current status

• All MQI verbs implemented except for MQCTL/MQCB

• Helper for building and parsing DLH recently made available
  – Along with a sample program to show how to use it

• No helpers for other headers such as RFH2
• No PCF classes in this library
Summary

• This presentation has described efforts to improve the developer experience

• To learn about MQ faster
• To make it easier to use MQ from modern development environments

• Feedback will inform future development in these areas
Any questions?