Easier use of the Node.js interface for MQ

Mark Taylor
Published on 05/06/2018

In November, I delivered a new language binding to MQ, making it possible to use Javascript in a Node.JS environment to access the MQ API. I published the first release of that code only through GitHub which required that you clone it locally before you could use it. Once the initial public exposure proved some stability to the interface, I added the **ibmmq** package to the npm repository meaning that you could simply refer to it in the descriptors (*package.json*) for your programs and npm would automatically download and install it.

But there was still one drawback. The package builds on, and requires, the MQ C client runtime libraries. That needed a separate operation to complete the installation, as those libraries came from product installation media or required you to login to an authorised location before they could be accessed.

No longer though.

Recent updates to the MQ Redistributable Client package and then to the MQ Node.js component have made it much simpler to get an MQ application running, and to create standalone containers for those applications.

The MQ Redistributable Client

This package was designed to make it possible to embed the essential MQ runtime files alongside an application, making it possible to distribute a standalone resource.

The two primary application platforms – Linux x64 and Windows – are supported by this component.

Earlier this year, the latest versions of the Redistributable Client were put into the same public repository as the full MQ Advanced Server for Developers product. That makes it possible to download the file directly without needing any login credentials.

Automatic installation with the Node.js package

The MQ Node.js package now exploits that easy access to the C runtime to download it as part of its own installation.

When you write a Node.js program, the description of the program in *package.json* lists which other packages are required. Running `npm install` for your application tries to download those prerequisite packages (and their dependencies) from the npm repository. Part of the ibmmq package includes directives (also written in JavaScript) for a postinstall step that pulls down that tar or zip file during the npm installation process. The postinstall code then unpacks the image and deletes many of the files, leaving just the pieces needed for actually running a C application.

For example, using one of the sample programs included with this package, and creating a basic package file:

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

Then running the installation step, which shows the download of the C libraries:

```
$ npm install
... (various messages showing installation of other pieces) ...
> ibmmq@0.7.0 postinstall /tmp/nodetest/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/nodetest
└─┬ ibmmq@0.7.0
  ├─┬ ffi@2.2.0
  │ ├── bindings@1.2.1
  │ └─┬ debug@2.6.9
... (more about the dependency tree)
```

Now the amqsput program is ready to run with no further intervention

```
$ node amqsput
Sample AMQSPUT.JS start
MQCONN to QM1 successful
MQOPEN of SYSTEM.DEFAULT.LOCAL.QUEUE successful
MQPUT successful
MQCLOSE successful
MQDISC successful
```

Disabling the automatic installation

It is possible to disable the automatic download of the MQ C runtime. You may have a preferred version of the MQ client libraries, or you may already have installed the client for other reasons and want to reuse the same installation.

Set the environment variable MQIJS_NOREDIST to any value and the file is not downloaded:

```
$ export MQIJS_NOREDIST=true
$ npm install
... (various messages showing installation of other pieces) ...
> ibmmq@0.7.0 postinstall /tmp/nodetest/node_modules/ibmmq
```

```
> node postinstall.js

Environment variable set to not install IBM MQ Redistributable C Client
amqsput@0.0.1 /tmp/nodetest
└── ibmmq@0.7.0
```

Platform support

The redistributable clients are only available for Linux x64 and Windows. The automatic installation attempt is bypassed if you are running on a platform that does not have the Redistributable Client available. In those circumstances, you will have to get the libraries from a more traditional route. And of course, there are platforms where Node.js runs but there is no MQ client available at all. For that situation, you will have to find a completely different mechanism such as running in a VM or container.

What parts are kept

The full Redistributable Client packages contain much more than the libraries needed to run a C program. Most of the rest of that package gets deleted by the unpacking process, but one program is left alone – **runmqsc**. That is done so that it is possible to use this downloaded image to create Client Channel Definition Tables (CCDT) from just within the Node directory by using runmqsc with the `-n` flag. And that in turn is done because you may want to use this solution within a container for simpler deployment. Which is the subject of the next section.

Docker container

Included with the ibmmq package is an example for how to build a Docker container that runs an MQ Node.js program.

The Dockerfile shows one way to build from a base image, and add a program along with its prereqs. A script runs the build and then executes the program. The script sets some environment variables including MQSERVER to point at a queue manager.

Unsurprisingly in this case, the program did not have credentials for a successful client connection. But it demonstrates the principle is correct after getting as far as making the initial connection to the remote 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
```

```
MQ call failed in CONNX: MQCC = MQCC_FAILED [2] MQRC = MQRC_NOT_AUTHORIZED
[2035]
```

Client connections

For simplicity, the sample script for building the image takes the most basic approach for defining the client connection. In the `docker run` command it sets the MQSERVER environment variable. But I have played with a couple of other mechanisms.

- As part of the build process, I run `runmqsc` on the build machine (because I've got a full MQ installation) and use it to create a CCDT. Then a step in a Dockerfile copies that CCDT to the container and at runtime I set the environment variables MQCHLLIB and MQCHLTAB to point at where I've copied the CCDT.
- For a truly self-contained approach where the Docker build machine does not have MQ installed, I can use the fact that runmqsc will exist in the container to put an MQSC script into the build and have a program execute runmqsc there before executing the real program.

Something similar would work for TLS certificate stores. Either create the certificate and the store locally and copy as part of the `docker build`, or build the image and run a command (the gskit libraries and commands are also left inside the unpacked redistributable directory) to populate a store.

Minimising the image

The Dockerfile is structured to try to minimise the number of layers in the image, as part of making it as small as possible. It starts with a reasonably small base image, a build of debian called **jessie-slim**. Then the build combines many operations (in particular, installation of temporary components and their eventual uninstallation) into a single RUN command. It does mean that the mostly likely variation between two containers – the program itself – is established about half-way through the build process. That might slow down generation of multiple containers that only vary by the program code as all the identical subsequent steps need to exeucte for each container build. You could probably tweak Dockerfiles based on better knowledge of differences between your programs and their prereqs so that one of the final steps installs the real program but you would need to be careful not to miss out on pieces of the dependency chain that `npm install` handles for you.

Summary

Making it easier to build and deploy MQ applications is important. I hope the new features described here help with that.