Introducing: an MQ Light client for Java

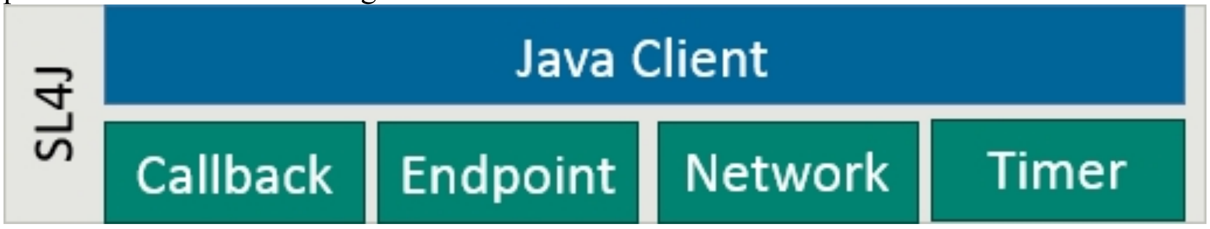
Adrian Preston
Published on 06/02/2015 / *Updated on 06/02/2015*

Today we're excited to make available an early version of a Java-based client API for MQ Light. While this API shares similarities with our Node.js client we've also set our sights on creating a truly asynchronous API that's ideal for wiring together the components comprising a reactive system. Right now, you can download a .zip from this page that contains the client, documentation and also sample code. Or if you're just browsing: the Javadoc is also available here.

The MQ Light messaging model underpinning the Java client emphasises simplicity and the speed with which it can be used to solve common problems. The same model is used by all of the other MQ Light clients. Another point of commonality is support for a carefully selected set of message payloads. All of the MQ Light language clients support exchanging: text, binary, and JSON data – making it straight forward to wire together components written in a variety of different languages.

Another key design point for the Java API – is that *nothing* should block. The client does all of its work either on the thread calling into client code – or on a small number of pooled threads used to carry out specific functions (such as network I/O – more on this later!). This fits well with the asynchronous nature of messaging and also has the benefit of reducing the number of thread context switches required to send a message. We've also been keeping one eye on how frameworks like Akka are taking an actor-based approach to concurrency and have built the Java client to be straightforward to encapsulate as an actor.

While on the subject of framework integration: isn't it annoying when a library looks great, but doesn't quite integrate with the other components you've chosen to use? That annoys us too, which is why we've built the Java client in a modular way. Here's how all the pluggable parts of the Java client fit together:



By default the client uses Logback to implement the SL4J logging interfaces, Netty to implement network I/O, with standard Java Executor classes backing the other components. But say you want to use a different network library? No problem: just write a few lines of shim code implementing the relevant Java interfaces, and away you go!

As always – we're keen to hear from you. So if you have any question, or have tried the Java client and want to give us feedback – head on over to the forum.