Article
Rob Tweed · Jul 31, 2017 5m read

Introduction to QEWD Micro-Services

In my previous posting about the new support in QEWD for JSON Web Token (JWT) support, I mentioned that it was a key step in enabling Micro-Service support in QEWD. In this post I'll give some background to how they work and the thinking behind them.

If you haven't heard about Micro-Services and/or want to learn more, there's lots of information available if you do a Google Search. Here's a good starting point:

https://smartbear.com/learn/api-design/what-are-microservices/

In a nutshell, they are all about breaking down, into separate self-contained units, the processes and functionality that you want an application to provide. So instead of you having a single, monolithic back-end that handles all the incoming requests, they are separated out into separate systems - usually separate servers (often using containers such as provided by Docker and either Docker Swarm or Kubernates). Incoming requests are routed to the system that will handle that particular business function or operation, usually using an inter-networking "fabric" based around HTTP(S).

Like everything, there are pros and cons to Micro-Services:

- the big pro is being able to separate out business functionality to sub-systems that are designed solely for one dedicated activity, often the responsibility of a specialist team dedicated to that one piece of business functionality
- the second pro is scalability and resilience - you can scale out your application over numerous different systems, with no single point of failure
- a significant con is the increased potential complexity of the resulting mesh of sub-systems, each of which needs managing and maintaining, in addition to some kind of oversight management and maintenance that monitors and ensures the availability of all the individual sub-systems
- a second significant con can relate to the use of HTTPS to provide a secure micro-service inter-networking fabric. The handshaking required to establish an SSL connection involves multiple round-trips between the client and server system which can take a significant number of milliseconds. If a request involves the invocation of multiple micro-services across this fabric, or a chain of microservice invocations, this latency can start to add up. Once completed, the HTTPS connection(s) is/are torn down, and need to be re-established anew when the next request comes in. Companies such as nginx provide specialist Micro-Service support technologies to work around such issues, which is great, but such solutions introduce more moving parts into an already complex inter-networked fabric.

As it turns out, QEWD was in a pretty good place, once JWT support was added, to achieve the pros of a Micro-Service architecture, whilst going a long way to addressing those two worrying cons.

The key lies in the web-socket support that is built-in to QEWD. Normally you make use of this by using the ewd-client module in your browser-side logic. It creates and maintains a persistent web-socket connection to a QEWD server, making use of the socket.io library. If your browser is connecting over SSL to QEWD, then the web-socket connection is secured over that connection also. Once the SSL handshaking has completed and the socket connection is established, there is no further overhead involved in using and re-using the web-socket connection, and it's a bi-directional channel. QEWD provides a simple set of APIs for communicating over the web-socket connection between your browser and back-end logic.

Since QEWD's master process is a Node.js/JavaScript process, there's no real difference between its run-time environment and that of the browser. So what if the equivalent of ewd-client was made available to a QEWD

master process, so that it could connect to another QEWD server over a socket.io-based web-socket connection? This would allow two or more QEWD servers to establish a secure, high-performance inter-networking fabric. The cool part of it is that a "server" QEWD system in such a set-up wouldn't actually know the difference between a browser client and a QEWD server acting as a web-socket client. As a result, a QEWD micro-service could be implemented in just the same way as a normal QEWD back-end application!

That's what I've done and how QEWD can now support micro-services. Provided all your QEWD micro-service systems share the same JWT secret string, a master QEWD server (or farm of servers) can establish a web-socket-based inter-networking fabric to as complex a mesh of other QEWD-based micro-service servers as you like.

You simply define on a QEWD server the micro-service mappings you want for handling incoming requests, in terms of the message application name and type. If a matching incoming request is received, the QEWD server's master process forwards it to the mapped micro-service QEWD system, complete with the incoming JWT, rather than handling it locally, via its queue, on one of its worker processes. The micro-service QEWD system to which the request is forwarded could, itself, have mappings defined that further forward the request to other QEWD systems. On each "client" QEWD system, a callback is automatically set up to handle the response it gets back from the "server" QEWD system, so the response is automatically returned to the client that originally sent the incoming request.

How you stratify your applications and break them into micro-services is up to you. It can be as simple or as complex as you like. Each micro-service is simply implemented as standard QEWD back-end message handler functions.

There's no set-up and tear-down costs for the SSL fabric - all the traffic is over persistent web-socket connections between QEWD servers that are set up at start-up time. There's no additional moving parts or technologies - it's just a set of QEWD systems, each of which is just 100% Node.js.

So that's the background to QEWD MicroServices. If that's whetted your appetite, in the next post I'll explain how to set up a simple MicroService.

#Microservices #JSON #Node.js #Frontend #JavaScript #Caché