Article
Rob Tweed · Jul 31, 2017 7m read

Creating a QEWD Micro-Service

If you read my previous post that introduced QEWD Micro-Services, you're hopefully eager to learn how to use them. So in this post I'll explain what you need to know in order to get started.

If you look in the QEWD repository, you'll find the folder:

https://github.com/robtweed/qewd/blob/master/example/jwt

In my earlier post on JSON Web Tokens (JWTs) and QEWD I used this example application to explain how to use JWTs. This example application also demonstrates how to set up a simple Micro-Service, in this case a service that handles user authentication. So let me now delve into that aspect of the example application.

If you're wanting to use QEWD Micro-Services, you must also use JWTs - these provide the means by which the user's authentication and session can be cross-communicated and handled by multiple, discrete QEWD servers. So, take a look at the startup file:

https://github.com/robtweed/qewd/blob/master/example/jwt/startupfile/qe...

You'll see that it configures QEWD to use JWTs, and defines the secret string that is used for signing and encrypting the JWT:

```
jwt: {
 secret: 'someSecret123'
 },
```

However, you'll also see that it defines a micro-service:

```
 uservices: [
 {
 application: 'jwt',
 types: {
 login: {
 url: 'http://192.168.1.97:3000',
 application: 'test-app'
 }
 }
 }
 ]
```

You use the "*uservices* " config property to define an array of micro-services. Each array element specifies the incoming request's application name, in this case "*jwt*", and then one or more message types for that application. In the example, we're just specifying a single request message type: "*login*". For each message type, you specify:

- the URL of the QEWD system that you wish to handle that message type
- the QEWD application name on the remote system that will handle the incoming message

So , our example startup file is saying that any incoming QEWD request messages for the application named "*jwt*" and type "*login*" will be handled by a an application named "*test-app*" on a QEWD system at http://192.168.1.97:3000

When you fire up this startup file, one of the first things it will do is to load a built-in QEWD module named "*socketClient*", which establishes a Web-Socket connection to each of the QEWD systems specified in the uservices array. If you monitor the QEWD system log on the remote QEWD system, you'll see it accepting a standard "*ewd-register*" message, as if someone had started the *test-app* application in a browser - to it, there's no difference.

Note that the remote QEWD server's startup file must also configure the use of JWTs, using the same secret string as the master QEWD system, ie:

```
  jwt: {
secret: 'someSecret123'
},
```

The value of the secret string is up to you, but it must be the same on all QEWD master and QEWD Micro-Service systems

So that's both our QEWD systems configured and running. Now let's look at the application. Take a look at:

https://github.com/robtweed/qewd/blob/master/example/jwt/index.html

My previous posting on JWTs explained most of what's going on in this browser-side logic, but look at lines 13-34 of this file: they handle the submission of the login form. As far as this application is concerned, it's just sending a message of type "*login*" to the QEWD system which started it up and on which it was registered - the fact that this will be handled by a Micro-Service is irrelevant to the developer of the browser-side logic - if you've used QEWD before, you'll recognise that it's just a standard QEWD message call.

However by virtue of the *uservices* config mapping we added to the QEWD system's startup file, this request will be automatically forwarded to the second QEWD system that is handling the user authentication micro-service. So now let's look at its handler module, which, if you remember, by virtue of the *uservices* mapping, is handled by a module named *test-app*:

https://github.com/robtweed/qewd/blob/master/example/jwt/microservice/te...

Now, as far as this handler is concerned, an incoming message of type "*login*" might as well have come from a browser, so it's handled in the standard way. By the time this handler function is invoked, QEWD will have already verified and unpacked/decrypted the JWT contents and made them available as an object via the *session* argument:

```
login: function(messageObj, session, send, finished) {
// session contains the decrypted JWT payload
}
```

So now we can handle the *username* and *password* values that came from the user's login form - there's no difference to how you'd have done this in a "conventional" QEWD application. In a proper production application, we'd use some kind of authentication service or database, but here, for simplicity, I'm just doing a simple hard-coded test for a user named "*rob*" whose password is "*secret*":

```
var username = messageObj.params.username;
var password = messageObj.params.password;
if (username === 'rob' && password === 'secret') {
// successful login in
}
else {
// unsuccessful login
}
```

Let's deal with the un-successful login. Just as in a "conventional" QEWD application, we use the *finished()* function to return an error message and release the worker process:

```
return finished({error: 'Invalid login'});
```

If the login is successful, however we want to add a number of things to the session (and therefore the JWT):

- set the authenticated flag (which is invisible within the JWT to the browser user)
- change the session/JWT timeout, probably to a longer timeout such as 3600 seconds

We also might want to convey to the browser some information about the logged in user, eg how to display the user's name, eg:

```
session.userText = 'Rob';
```

We might also want to store the user's identity in the JWT as a secret field (ie invisible to the browser, but visible to a back-end QEWD handler):

```
session.username = username;
session.makeSecret('username');
```

The "logged in successfully" response message is then returned and the worker released in the standard way:

```
return finished({ok: true});
```

So that's it as far as the *login* micro-service developer is concerned - it's just a standard JWT-based QEWD message handler function. You neither know nor care if the response is being returned to a browser user or a QEWD system acting as a web-socket client!

What will happen automatically is that the response + modified JWT will be returned to the QEWD client system, which, in turn, will forward the response + modified JWT to the browser. As far as the browser is concerned, it has received the response + JWT it expected from the QEWD system it communicates with - it's unaware that it was actually handled by a QEWD Micro-Service.

The JWT in the browser now contains the user's identity in a secret claim value. That will be conveyed in any subsequent requests to the back-end QEWD system which, in turn, may forward it to some different QEWD Micro-Service. Provided all the QEWD systems share the same JWT secret string, the JWT and its secret claims are available for use by any QEWD micro-service. The fact that the secret *authenticated* flag is set to true means the user must have been correctly authenticated (via the login micro-service), and the secret *username* JWT claim securely provides the user's identity for use against a database etc.

One cool additional thing to note - we've used the *finished()* function in our example, but remember that QEWD also allows you to optionally have intermediate messages that you send using the *send()* function. You'll find that these are also supported across QEWD micro-services, and are conveyed back to the correct browser client. I believe this is a feature unique to QEWD Micro-Services.

So, from a developer's point of view, that's all there is to defining QEWD micro-services. It's really just business-as-usual, with everything handled automatically for you by QEWD simply by you defining the *uservices* mapping array in your QEWD startup file. All very simple and straightforward, but, as you're probably beginning to realise, allowing all manner of possibilities in terms of defining a complex and highly-scalable mesh of QEWD Micro-Services.

With Cache at the back-end of your QEWD Micro-services, you have the makings of a very powerful, highly scalable, resilient and very high-performance application platform.

#Microservices #JSON #Frontend #JavaScript #Caché

Source URL:https://community.intersystems.com/post/creating-qewd-micro-service