In order to build a webhook from scratch, you need an HTTPS server which is running on a machine that is accessible via the public Internet, and which has a certificate from a recognized certificate authority. The attached server code is a very basic HTTPS server which can receive POST requests like the ones that TradeLens uses to send subscription messages. To make it work in real life, we'll also need infrastructure around it:

1. It has to run on a machine that's accessible via the public Internet. We can create an IBM Cloud virtual machine for that purpose, using your account. (We could do this on any other cloud as well, but IBM Cloud is the only one where I've done this before.)  Cost is like $15 per month.

2. It has to have a valid security certificate -- we can get one for free from Let's Encrypt. (The certificate will have to be renewed every 90 days, which is pretty easy to do.)

3. Let's Encrypt requires that the host machine be accessible via a DNS name (something like foo.bar.com), not just via a numeric IP address. So, we have to create a DNS domain and add our server to that domain. Again there are a number of ways to do this, but IBM Cloud is the only one I've used. Cost of a DNS domain from IBM Cloud is like $20 per year.

4. Once we have these three things, we can run the attached server.js on our public machine, and it should be able to receive subscription messages from TradeLens. I am confident that this will work because we did this same thing last spring using an Apache server. The infrastructure requirements were exactly the same as what we need for the node.js server -- the only difference is that last spring Apache was accepting the incoming requests, whereas now we'll use node.js for that.

5. But both of our samples -- Apache and node.js -- just accept incoming subscription messages and display them on the console. To make this useful in real life, we'll also want to add a database, so we have persistent storage for the subscription messages. We could use an IBM Cloud database for that, or we could install Postgres or some other free database on the server machine; then we'd modify the server code to write to the database. Also need to think about how people will want to pull data out of the DB -- you mentioned Power BI for example.

So -- most of the remaining work is infrastructure, mostly in the cloud. In our call today, I fumbled around trying to find the menu options that I needed, but I went back in after the call and figured out what I was missing.

But just as a sanity check before we get all cloudy, you can test the node.js server on your local machine if you want to. I used a Red Hat Linux machine, but it should be basically the same on other systems. You need to install npm and node.js; also curl and openssl. Once you have those things, you can exercise the server as follows, and as shown in the attached recording:

A. Store the two attached .pem files someplace on your local machine. Those two files form the self-signed certificate, in the name of /C=US/ST=Pennsylvania/L=Erie/O=MyCo/OU=PDH/CN=Test2 . Modify the two fs.readFileSync() lines at the top of server.js to specify the full path names to the .pem files.

B. Run the server in one window, via this command:   node server.js      . You may have to do the following first (I did) to install a few optional packages:

npm install body-parser
npm install express
npm install lodash

When successful, the server will be listening for HTTP requests on port 8080, and will listen for HTTPS requests on port 8443.

C. Confirm that the server is listening and that the certificate works, via this command in a separate window:

openssl s\_client -connect localhost:8443

You should get a page or two of output, with the crucial lines being these (showing the self-signed certificate issuer and subject):

Certificate chain
 0 s:/C=US/ST=Pennsylvania/L=Erie/O=MyCo/OU=PDH/CN=Test2
   i:/C=US/ST=Pennsylvania/L=Erie/O=MyCo/OU=PDH/CN=Test2

Hit control-C to break out of the openssl connection.

D. Still in the second window, send a couple of JSON messages to the server via curl as follows (two long lines):

curl --insecure -s -X POST --header 'Content-type: application/json' -d '[{ "firstField": "Test message from TradeLens Support", "sender": "XXXXXXXXXXXXX" }]' <https://localhost:8443>

curl --insecure -s -X POST --header 'Content-type: application/json' -d '[{ "firstField": "Another message from TradeLens Support", "sender": "Same as the first one: XXXXXXXXXXXX" }]' <https://localhost:8443>

This emulates TradeLens sending a JSON subscription message. You should see "Message received" in the second window, and you should see the JSON in the server window.

E. The server stores the messages it receives in an in-memory array. To see the contents of that array, do this in the second window:

curl --insecure -s -X GET <https://localhost:8443>

As noted above, in real life the server would save the messages in a database, and you'd retrieve them via some other tool. The curl command here just shows that the basic server did receive the messages and still has them in memory.

By the way, we have to say --insecure on the curl commands because we're using a self-signed certificate, which is not very secure :-) Saying --insecure tells curl that we're OK with that insecurity.

So -- next steps:

I. Make sure you can run the sample server as-is, as a baseline.

II. To set up the IBM Cloud infrastructure, we have two options: since I'm now a member of your account, you could grant appropriate permissions to me and I could do the setup; or I could lead you though it. We'd want to record it in either case.

III. Run the sample server on the new IBM Cloud VM, using the self-signed certificate. If we send TradeLens subscription messages to it, we would expect to get certificate errors.

IV. Get a real, trustworthy certificate from Let's Encrypt -- I know how to do that. Once we've done that, we should be able to successfully send bona fide subscription messages from TradeLens to our server. They won't be stored in a database yet, but the server should accept the messages, store them in memory, and display them if we ask it to.

V. Database setup --