Article
Rob Tweed · Jun 22, 2016 14m read

Creating Node.js Web and REST services on a Caché system using EWD 3

In this article I'll describe how to set up web services and/or REST services using EWD 3.

Since EWD 3 is designed to be modular, you can construct the environment that exactly meets your needs, but for much of the time you'll probably find that the pre-built EWD 3 ewd-xpress super-module does most of what you need because it hooks together all the core EWD 3 and other building-blocks you'll need:

- the Node.js web server that pretty much everyone now uses: Express
- Caché (via the ewd-qoper8-cache module, which, in turn, relies on the ***cache.node*** interface file)
- ewd-qoper8 - the master/worker architecture that underpins EWD 3
- ewd-qoper8-express which adds Express middleware for ***ewd-qoper8***
- the EWD 3 Document Store abstraction of Caché
- EWD 3 Sessions

ewd-xpress allows you to further customise its behaviour, and that's what we'll do in order to define web or REST services.

As you'll see in the ***ewd-xpress*** documentation, you basically have to define two things:

- a startup file that defines how you want to configure ewd-xpress and then starts it up using that configuration
- one or more Node.js module files that define the back-end processing that will take place in an ***ewd-qoper8*** worker process.

So let's start with the startup file. This is just a JavaScript text file that you create in your EWD 3 run-time directory (eg ~/ewd3 or c: \ ewd3). Here's a simple example defining a web service URL path prefix of ***/myWebService***:

```
var config = {
  managementPassword: 'keepThisSecret!',
  serverName: 'My EWD 3 WebService Server',
  port: 8080,
  poolSize: 1,
  database: {
    type: 'cache',
    params: {
     path: '/opt/cache/mgr', // path to Cache Mgr directory
     namespace: 'USER'
    }
  }
};

var ewdXpress = require('ewd-xpress').master;

// add your custom middleware by first intercepting:
```

```
var xp = ewdXpress.intercept();
xp.app.use('/myWebService', xp.qx.router());

// now start up ewd-xpress
ewdXpress.start(config);
```

So the trick is to invoke the ***ewd-xpress intercept()*** function which allows you to define your own Express middleware. In fact we'll use the ***ewd-qoper8-express*** router and apply it to any incoming URL starting ***/myWebService***. You can specify whatever URL prefix you like, and, in fact, you could specify as many as you like - just add more ***xp.app.use()*** lines, eg:

```
xp.app.use('/myWebService', xp.qx.router());
xp.app.use('/anotherService', xp.qx.router());
xp.app.use('/myRESTStuff', xp.qx.router());
```

In all cases, the ewd-express-router will repackage the incoming HTTP request information into a JSON message that is dispatched to an available ***ewd-qoper8*** worker. That packaging is as follows:

```
{
  type: 'ewd-qoper8-express',
  path: // the full URL path that was received,
  method: // the HTTP method, eg GET, POST,
  headers: {
    // all the received HTTP headers, eg content-type: 'application/json'
  },
 params: {
   // the Express route parameters, as parsed by Express from the URL
 }
 query: {
   // any name/value pairs that were added to the end of the URL
 },
 body: {
  // the HTTP request's payload.  For application/json requests, this will be pre-
parsed into a JS object
 },
 application: // the first piece in the URL eg if the URL is /myWebService/login, the
n application is myWebService
 expressType: // the second piece in the URL, if defined.  eg if URL is /myWebService
/login then expressType is login
}
```

So this is the message structure that is dispatched to a worker process.

To handle these messages, you define a Node.js module with the same name as the application, eg for ***/myWebService*** URLs, you'll need to create a file in your EWD 3 ***node_modules*** directory named ***myWebService.js*** eg:

~/ewd3/node_modules/myWebService.js:

```
module.exports = {
  restModule: true,
  handlers: {
    // message handler functions for each type
```

```
  }
};
```

Note above how you must instruct the ***ewd-qoper8*** worker process that this application's messages are web service ones by exporting the property: ***restModule: true***

You now define handler functions for each ***expressType*** value (which is based on the 2nd piece of the incoming URL path). So, if you sent a request using the URL ***/myWebService/login***, you'd be able to handle this by defining a handler for the ***expressType login,*** eg:

```
module.exports = {
  restModule: true,
  handlers: {
    login: function(messageObj, finished) {
      // handle /myWebService/login requests
    }
  }
};
```

messageObj is the incoming re-mapped HTTP request message object, as shown earlier above (except that the type and expressType properties will now have the same value)

So typically you'll determine what to do with the incoming message based on one or more of the following JSON objects:

- **messageObj.body**: the POST'ed request payload
- **messageObj.query:** name/value pairs in the query string at the end of the URL (following a ?)
- **messageObj.headers:** HTTP request headers

Your handler function must finish by defining the JSON response for the incoming request - use the function provided by the ***finished*** argument for this, eg:

```
finished({
  // response object
});
```

The response object is up to you to define and can be as simple or as complex as you want.

If you want to signal an error, just return an error object, eg:

```
finished({
  error: 'some error message'
});
```

ewd-xpress will automatically return this as an ***application/json*** response with a status of 400.

The ***finished()*** function also instructs the worker to release itself back to the ***ewd-qoper8*** available pool, so you should only invoke this function once, and when all message handling processing is completed. Note that the ***finished()*** function can be invoked from within a call-back function if your message handler needs to use asynchronous logic, eg to access a remote resource. Only once your asynchronous logic is completed and you invoke the ***finished()*** function will your worker process be released back to ***ewd-qoper8***.

Handling REST

From the point of view of EWD 3 / ewd-xpress, there's really no difference when creating REST services. The main difference is that you'll write specific handling logic for a URL path (aka resource) depending on the value of messageObj.method (GET, POST, PUT, DELETE etc) which will define what you want to do with that resource.

Accessing the Caché Database

Your message handler functions have access to the Caché database:

- - via the low-level APIs provided by Caché's Node.js interface file - ***cache.node;*** and
- - via the ***ewd-document-store*** abstraction

It's up to you which you use. Access them respectively via:

- **this.db**, eg:

```
var node = {global: 'myGlobal', subscripts: ['a']);
var value = this.db.get(node).data;
```

- **this.documentStore**, eg:

```
var glo = new this.documentStore.DocumentNode('myGlobal', []);
var value = glo.$('a').value;
```

For details on using the ewd-document-store APIs (which abstracts Globals as persistent JavaScript objects and a Document Database), see:

http://gradvs1.mgateway.com/download/ewd-document-store.pdf

Via the ***cache.node*** APIs you can also invoke Extrinsic Functions (***this.db.function***) and access Cache Objects / Classes.

Using EWD 3 Sessions

In many situations where you're handling web service or REST requests, you'll want to use EWD 3 Sessions. To do this, you must first load the ***ewd-session*** module into your application's handler module, eg:

```
var sessions = require('ewd-session');

module.exports = {
  restModule: true,
  handlers: {
  login: function(messageObj, finished) {
  // handle /myWebService/login requests
  }
  }
};
```

Now all you need to do is use the two ***ewd-session*** APIs:

- **sessions.create()** to create a new EWD 3 session
- **sessions.authenticate()** to check an incoming session token and give access to the associated EWD 3 Session

So, for example, you might have a login function which first authenticates the user's username and password, and if valid, creates a new EWD 3 session and returns its token, eg:

```
var sessions = require('ewd-session');

module.exports = {
  restModule: true,
  handlers: {
     login: function(messageObj, finished) {
      if (messageObj.method !== 'POST') {
        finished({error: 'Only POST requests are accepted'});
        return;
      }
      var username = messageObj.body.username;
      if (username === '') {
        finished({error: 'You must enter a username'});
        return;
      }
      var password = messageObj.body.password;
      if (password === '') {
        finished({error: 'You must enter a password'});
        return;
      }

      // login credentials in global ^credentials('byUsername','rob','password')='sec
ret'

      var credentials = new this.documentStore.DocumentNode('credentials', ['byUserna
me', username]);
      if (!credentials.exists) {
        finished({error: 'Invalid username'});
        return;
      }
      if (password !== credentials.$('password').value) {
        finished({error: 'Invalid password'});
        return;
      }
      // login credentials OK - create EWD session
      var session = sessions.create(messageObj.application, 1200);
      session.authenticated = true;
      session.data.$('username').value = username;
      finished({token: session.token});
    }
  }
};
```

and you might have a handler for a message of type ***doSomething*** which can only be invoked if the client is logged in with a currently active EWD 3 session. The trick here is that the ***/myWebService/doSomething*** message is expected to contain a valid EWD 3 session token in its Authorization HTTP request header:

```
    doSomething: function(messageObj, finished) {
```

```
      if (messageObj.method !== 'POST') {
        finished({error: 'Only POST requests are accepted'});
        return;
      }
      if (messageObj.headers.authorization) {
        // authenticate against EWD Session token
        var result = sessions.authenticate(messageObj.headers.authorization, 'noCheck
');
        if (result.error) {
          finished({error: result.error});
          return;
        }
        // token is OK, but make sure it's for this Web Service and that the user's c
redentials have been authenticated

        var session = result.session;
        if (!session.authenticated || session.application !== 'myWebService') {
          finished({
            error: 'Invalid session'
          });
          return;
        }

        // OK to do something!

        // process the incoming message object

        // the Cache database is accessible via this.documentStore
        //   eg var glo = new this.documentStore.DocumentNode('myGlobal', ['a']);

        //  and session data via session.data, eg session.data.$('someName').value

        //  .. then return results:

        finished({
          // your response object
        });

      }
      else {
        finished({error: 'Missing authorization token'});
      }
    }
```

Remember that your web service handler MUST return a response, so return errors for every failed eventuality, otherwise the worker will never get released back to the ***ewd-qoper8*** available pool and your client will sit waiting for a response until it times out.

Note that you don't have to worry about managing EWD 3 sessions - ***ewd-session*** will garbage-collect timed-out sessions automatically for you.

Starting and Running your EWD 3 Web Service Platform

Assuming you named your ewd-xpress startup file ~/ewd3/ewd-xpress.js, you'd start the service up on a Linux system using:

```
cd ~/ewd3
sudo node ewd-xpress
```

sudo is only needed if the permissions for Cache restrict its access to the root user.

On a Windows machine, eg:

```
cd  c:\ewd3
node ewd-xpress
```

You can test your web service using a browser for simple GET requests, or use a REST client such as Chrome Advanced REST Client (ARC).

Set the content type to ***application/json.***

Installing ewd-xpress

Of course the previous instructions assume you've installed ***ewd-xpress***. So let's look at how that's done.

It's pretty straightforward. The following instructions are for a Caché system running on Linux, but for Windows, it's pretty much identical except for the different directory naming. At the Node.js-level it's all the same.

I'll assume that you already have Caché installed and running and that you've installed a version of Node.js supported by the version of cache.node on your Caché system.

First, we'll create a directory for all our EWD 3 work - I'm going to use ~/ewd3

```
cd ~
mkdir ewd3
cd ewd3
```

Now you just need to let NPM do the work:

```
npm install ewd-xpress ewd-xpress-monitor
```

You'll probably see a whole bunch of warnings about peer dependencies. Just ignore them. ***exd-xpress*** will install a whole bunch of other EWD 3 modules along with others such as ***socket.io*** that can be used by your ***ewd-xpress*** applications. Express will also be automatically installed as a peer dependency.

In order to use Node.js with Caché , you'll need to have obtained a compatible version of the ***cache.node*** file. I'll be using the Node.js 4.2-compatible version (for 64-bit Linux) - ***cache421.node***. So this needs to be moved into place and rename it. EWD 3 modules expect to find it in your bottom-level ***node_modules*** directory, ie ~/***ewd3/node_modules*** in our case. This ***node_modules*** directory will have been created when you installed ***ewd-xpress*** in the command above. So, for example:

```
cd ~/ewd3/node_modules
mv cache421.node cache.node
```

Finally, you'll need to copy a few files from directories within the various EWD 3 modules into the directories that EWD 3 and ***ewd-xpress*** will expect them to reside. First:

```
mv ~/ewd3/node_modules/ewd-xpress/example/ewd-xpress.js ~/ewd3/ewd-xpress.js
```

This command above moves the pre-built example script for running ***ewd-xpress***. EWD 3 will expect you to invoke this from your ~***/ewd3*** directory. If you've been using all the steps above to install on a previously new, virgin Ubuntu system, the example ***ewd-xpress.js*** script will run "out of the box". However, if you are using an existing Caché system, you'll need to modify the access parameters that allow the ***cache.node*** interface to connect to Caché. If so, find the following lines in ***exd-xpress.js***

```
var config = {
  managementPassword: 'keepThisSecret!',
  serverName: 'New EWD Server',
  port: 8080,
  poolSize: 1,
  database: {
    type: 'cache'
  }
};
var ewdXpress = require('ewd-xpress').master;
ewdXpress(config);
```

There are 4 key Caché access parameters:

- **path:** the mgr path for your Caché system
- **username**: the username you'll use for accessing Caché
- **password**: the corresponding password
- **namespace**: the Caché namespace to which you want to connect

The defaults are:

- path: /opt/cache/mgr
- username: _SYSTEM
- password: SYS
- namespace: USER

If your Caché system requires different values, you specify them using the database.params object. You only need to specify those that are different from the defaults above. So, for example:

```
var config = {
  managementPassword: 'keepThisSecret!',
  serverName: 'New EWD Server',
  port: 8080,
  poolSize: 1,
  database: {
    type: 'cache',
    params: {
      path: '/usr/lib/cache2015/mgr',
      namespace: 'VISTA'
    }
  }
};

var ewdXpress = require('ewd-xpress').master;
ewdXpress(config);
```

The other config parameters are:

- **managementPassword:** the password you'll need to use with the ***ewd-xpress-monitor*** application in order to gain access
- **port**: the port on which Express is listening for input
- **poolSize**: the maximum number of worker processes that ewd-xpress will start. Note that each worker will consume a Caché license

OK, just one last step - move the monitor application code into place. ***ewd-xpress*** assumes that the physical path that will represent Express's root URL is ~***/ewd3/www***. The browser-side resources for each of your ***ewd-xpress*** applications should be saved under this www directory. So:

```
cd ~/ewd3
mkdir www
cd www
mkdir ewd-xpress-monitor
cp ~/ewd3/node_modules/ewd-xpress-monitor/www/bundle.js ~/ewd3/www/ewd-xpress-
monitor
cp ~/ewd3/node_modules/ewd-xpress-monitor/www/*.html ~/ewd3/www/ewd-xpress-
monitor
cp ~/ewd3/node_modules/ewd-xpress-monitor/www/*.css ~/ewd3/www/ewd-xpress-
monitor
```

That's it, you should now be ready to try it out.

First, start ***ewd-xpress:***

```
cd ~/ewd3
sudo node ewd-xpress
```

You should see the following or similar:

```
ubuntu@ip-172-30-1-247:~/ewd3$ sudo node ewd-xpress
webServerRootPath = /home/ubuntu/ewd3/www/
Worker Bootstrap Module file written to node_modules/ewd-qoper8-worker.js
=======================================================
ewd-qoper8 is up and running.  Max worker pool size: 1
=======================================================
```

EWD 3 will not start up worker processes until they are actually needed. It's actually the ***ewd-qoper8*** module that is responsible for the worker pool mechanics.

Now go to your browser and invoke the following URL (adjust the IP address appropriately to the one allocated to your Ubuntu system:

```
http://192.168.1.100:8080/ewd-xpress-monitor/index.html
```

The monitor application should start and ask for your management password. Enter **keepThisSecret!**

You should now see the overview panel for your EWD 3 / e***wd-xpress*** environment. Currently that's all there is in the monitor application, but you'll find that it provides a very important function - the ability to stop worker

processes. You'll need to do this when you begin to develop your own e***wd-xpress*** applications. Try clicking the worker process stop button and see what happens: immediately you'll see a new one starts up.

Meanwhile, in the Node.js / ***ewd-xpress*** process console window, you'll see lots of activity now going on.

That's it, you now have EWD 3 / e***wd-xpress*** fully working with Caché. What you're seeing in the browser is a React.js application. if you're interested in React.js development, you can see its source code by looking in the directory:

```
~/ewd3/node_modules/ewd-xpress-monitor/www/
```

What you're actually running in the ~***/ewd3/www/ewd-xpress-monitor*** directory is a bundled "compiled" version of all these React.js components (and many more client-side modules) - hence you'll just see the following files in the working ***/www/ewd-xpress-monitor*** directory:

- index.html
- bundle.js
- Select.css

One last thing - provided you have a suitable Caché license, you can take a peek at where and how EWD 3 creates and stores its Session information. Use your favourite tool for examining Globals and look in your configured namespace for the Global ***^CacheTempEWDSession***

Now that you have ***ewd-xpress*** working, go back to the top of this document and make the modifications to the startup file to get it serving up Web Services.

#JSON #Node.js #React #REST API #SOAP #Caché

Source URL: https://community.intersystems.com/post/creating-nodejs-web-and-rest-services-cach%C3%A9-system-using-ewd-3