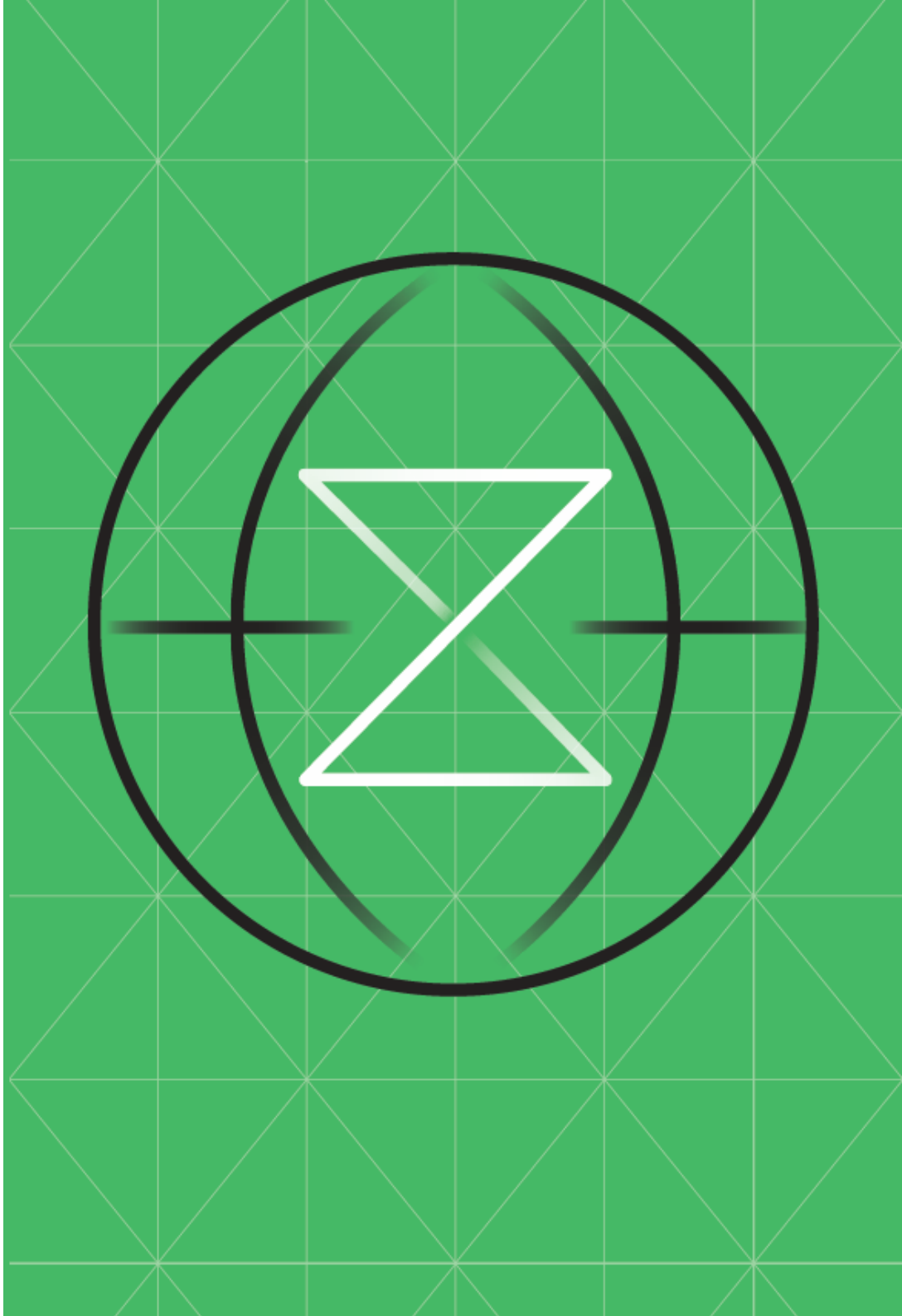
Extended

230222-1425

NODE2

Using node and zoau to access zOS data



NODE.JS AND ZOAU

The ZOAU commands have been developed to enable shell scripting of interations with zOS datasets, jobs, operations, as well as USS filesystems and commands. The `zoautil_py` library is provided with ZOAU to enable similar scripting to be easily built using Python.

This challenge introduces the node.js equivalent module - zoau-node.

The Challenge

During this challenge you will enhance your NODE1 web server application to handle multiple diffrent types of requests - one of which will be retrieve zOS dataset content and present back to the web user.

Before You Begin

You will need the javascript code you created for the NODE1 challenge, and bookmark the zoautil_py documentation link.

Investment

Steps	Duration
6	45 minutes

1 CREATE THE STARTER PROGRAM

When you completed the NODE1 challenge, you had created a javascript program called `node/program1.js` - create copy of that file in the same directory and call it `program2.js`

This program uses a basic http server module, and provides no framework for handling different URLs, to serve different kinds of request/response.

By the the end of this challenge, your application program will be able to serve responses to requests like

URL	Response
/hello/Martin	a greeting for Martin
/hello/Angel	a greeting for Angel
/data/SRCJCL1	the contents of a member of a specific dataset
/data/PLISRCA	the contents of another member of a specific dataset
/	the original “Hello World” greeting from NODE1

NODE2|230222-1425

2 REWORK THE NODE1 PROGRAM

To make your application more flexible, you need to make some minor changes to program2.js to make it easier to extend it later.

A common practice for “listener” application is to allow the listening port to be set from the environment.

Before:

```
const port = xxxxx;              //replace the xxxx with the correct port nr.
```

and after:

```
const port = process.env.PORT || xxxxx;   // remember that xxxxx is your Zid number + 30000
```

Now, you can start the program with something like:

```
PORT=30001 node program2.js
```

and it will use the 30001 port instead of what you have “hardcoded” in the application code.

Since you are accessing environment variables at the beginning of the program, you may as well set the `myname` variable here, also. Remember to remove the “myname” assignment later in the code.

```
const port = process.env.PORT || xxxxx;   // remember that xxxxx is your Zid number + 30000
let myname = process.env.LOGNAME;
```

You may have noticed that to send data back to the user, the program sets 3 kinds or information:

1. status code - 200 for OK, 4xx for errors (full details at IETF RFC9110)
2. response content type - text, image, json, and so on
3. the response content itself - html, text, json, images, …

Your application will be sending content in response to different requests, so it will be easier to make the response action into a function :-

```
const port = process.env.PORT || xxxxx;   // remember that xxxxx is your Zid number + 30000
let myname = process.env.LOGNAME;

function sendResponse(res,content, code, type){
    res.statusCode = code;
    res.setHeader('Content-Type', type || 'text/plain');
    res.end(content)
}
```

Now change the server to set up the response and use the function

Before:

```
res.statusCode = 200;
myname = process.env.LOGNAME
res.setHeader('Content-Type', 'text/plain');
res.end(`Hello World from ${myname} on z/OS`)
server.close()
```

and after:

```
sendResponse(res,`Hello World from ${myname} on z/OS`,200,'text/plain')
server.close()
```

Run the program with `node program2.js` and it should behave exactly the same as program1.js

3 MATCH URLS WITH PATTERNS

Currently, program2.js always responds with the same answer, regardless of how it is invoked.

You can make it more useful if you add functionality to recognise parameters passd to it, and respond with “custom” information.

When your program receives a “request”, the `req` object contains a couple of very useful properties -

- req.url – this tells you how the application was invoked
- req.method – tells you what method was used - GET, POST, HEAD, etc

For the moment, assume your application will receive GET requests from a browser.

There are lots of “frameworks” available for node.js to help you build applications which can handle a variety of different request types; express.js being the most popular, followed by Sails, Loopback, Feather, Meteor and many others.

To understand how these frameworks may work “under the covers”, create a mini framework of your own; use the existing `http` module, and add some “patterns” to help recognise and respond to different URLs.

The patterns can be represented by Regular expressions, used in Python, Node, unix/linux utilities, and many other programming languages

Node.js regular expressions (regex) can be created as a simple variable assignment, and are checked for matches against character strings in the URL using the `.test()` method.

Change the server code to detect and respond to just the “/” request.

```
const basic = /^\/$/;     // between the beginning (^) and end ($) look for "/"
console.log(req.url)
if(basic.test(req.url)){
   sendResponse(res,`Hello World from ${myname} on z/OS`,200,'text/plain')
}
```

When you test this from most browsers, you may see a request URL for "/favicon.ico" - this is the browser asking your server for a picture to place in the browser tab so you can easily identify which tab is running your application.

```
Server running at http://0.0.0.0:30001/
/
/favicon.ico
/favicon.ico
/
```

You need to ignore this request, and make sure you respond properly only to requests you recognise. This will mean relocating the `server.close()` function. Think about what needs to happen to the `.close()` function if you receive the "/favicon.ico" request *before* the URL you actually want to respond to.

4 CUSTOMISE URL RESPONSES

Your application can recognise URLs with patterns, and will soon have the ability to respond with information your requests send in the URL.

This is basically how REST servers) work, used widely for microservices, and for APIs on top of existing systems.

Users can send a request like “/hello/” and expect the response to include the name they provide.

`/hello/Martin` should at least include “Martin” in the response - “Hello Martin!”; to `/hello/Prakash`, your application should respond with “Hello Prakash!” – you get the idea?

Add a pattern, and its corresponding test to your program2.js.

A couple of hints:

- to test for URLs beginning with “/hello/”, use `/^\/hello\//`
- to get the name from the URL, you can use the node.js string replace() method `string.replace(/^\/[^\/]*\/([A-Z0-9a-z]+).*$/,'$1')`

The regex in the replace( ) method is saying:

1. starting at the beginning, look for “/”
2. move along the string until the next “/”
3. store the following set of alphanumeric characters (stop at the first non-alphanumeric character, or the end)

The ‘$1’ tells replace( ) to use the stored value from the last step instead of the whole string.

You will need to add a test in the server section to add support for the new URL.

Your server should look something like:

```
const server = http.createServer((req, res) => {
  const basic = /^\/$/;
  const hello = /^\/hello\//;

  console.log(req.url)
  if (hello.test(req.url)){
    const name = req.url.replace(/^\/[^\/]*\/([A-Z0-9a-z]+).*$/,'$1')
    sendResponse(res,`Hello ${name}!`,200,'text/plain')
  }
  if (basic.test(req.url)){
    ...
```

Once you have this working, you are ready for the next step - making your application into a RESTful API to access zOS datasets!

5 A RESTFUL SERVER

In the Python and USS challenges, you worked with the `zoautil_py` package; this enabled your "members.py" and "datasets.py" code to retrieve information from zOS.

With the node version of this package - `zoau` - your application will be able to something more – return the contents of a dataset directly to the browser.

This will require your application to recognise and respond to another URL format: `/data/<member>`

The idea is to display the contents of a member of a dataset to a requesting user - the user only knows the member name - your application knows which dataset to read the data from.

This will make your application into a simple RESTful service for displaying zOS data. To get access to zOS datasets, update your application to bring in the zoau module:

```
const http = require('http');
const {datasets} = require("zoau")  // import the datasets class from the zoau module
```

Your server section will need a new pattern:

```
const data = /^\/data\/.*/      //  "/data/<member>"
```

and you can use a similar replace( ) pattern to extract the requested "member" name.

To read data from a dataset, your application will use the `datasets.read()` function; what you need to know is that this function does not directly return the data - the response from zOS could take a long time, depending on the size of the dataset.

To process the data when it is ready for your application, you need to add a simple bit of code to handle it; this uses a function called ".then()" which follows the ".read()" function.

```
const dsopen = datasets.read(dsn)
    .then(function(contents){
```

```
        sendResponse(res,contents,200,"text/plain")
    })
```

Now you can see why the sendResponse() function is useful.

Update your server section to construct the dataset name (“dsn” in the example above) to be `ZXP.PUBLIC.SOURCE(<member>)` - where is the name the user requested.

If everything works as required, your application should recognise `/data/NODE2TXT` and respond with the current contents of `ZXP.PUBLIC.SOURCE(NODE2TXT)`

6 HANDLE ERRORS GRACEFULLY AND YOU'RE DONE

By now, your application server is working well for valid members of the dataset, but breaks if the member is not in the dataset.

You might see errors like:

```
        throw new Error(JSON.stringify(response));
              ^
Error: {"stdout":"","stderr":"BGYSC1304E Unable to open dataset ZXP.PUBLIC.SOURCE(BOB) for read.
\n","command":"dtail  -n +1 'ZXP.PUBLIC.SOURCE(BOB)'","rc":2}
    at Object.read (/global/node/lib/node_modules/zoau/lib/dataset.js:163:11)
```

You need a little extra code to add to the datasets.read() function to catch these errors.

```
            .catch(function(error){
                sendResponse(res,`${dsn} not found`,404,"text/plain")
            })
```

This catches the missing member error, and returns a "404" error to the user, indicating that the member could not be found.

Once you have the error-handling working, ***make sure that your application server is running*** and use the CHKNODE2 JCL job to validate the challenge and claim the points.