Asynchronous JavaScript

In this assignment, you will convert asynchronous node.js code implemented using callbacks to use promises instead.

1 Read

For this lab, you will want to have handy the node.js documentation for HTTP and HTTPS.

2 Understand

In this section, we walk through some node.js code that makes asynchronous HTTPS requests using callbacks. This is the code you will convert to use promises.

2.1 Literate Programming

We'll explore this code using *Literate Programming*. Rather than a simple listing with a few comments littered about, Literate Programming aims to turn the description of code into an act of literature (seriously!). We intersperse code and prose to explain a program's behavior in bite-sized chunks that are better suited to literary exposition than computational execution.

From a single file that contains both the description and the code, we can:

- *weave* the file into a human-readable document
- *tangle* the file into a program that can be complied or executed

I created this assignment using Literate Programming techniques.

2.2 Preliminaries

Because we'll be making *secure* (encrypted) HTTP requests, we import the node.js `https` module.

To help with debugging, we define `prettyPrintJson`, a function that takes a JavaScript object and "pretty prints" it as JSON. *Pretty printing* is a term of art for generating nicely formatted output that's easier to understand (and use for debugging).

```
const https = require('https');
const debug = false;            // Set true for debug output

function prettyPrintJson (jsonObject) {
  return JSON.stringify(jsonObject, null, 2);
}
```

To enable debugging output, set the value of `debug` to `true`.

2.3 Main Function

The `randomOrgApiCallback` function is our main function. As parameters, it takes two configuration objects:

- `requestOptions` contains the details of the HTTPS request (Section 2.5.1).
- `postData` is the information we're going to send in the body of our request (Section 2.5.2).

Node's `https.request` function takes a `requestOptions` object and does the following:

1. Set up to make an HTTPS request using the contents of the `requestOptions`.
2. Asynchronously send the request.
3. When response data are received, invoke the callback function; if the response data are large, the callback may be triggered multiple times. (We won't see that happen here, but it's good to be ready for the possibility.)

```
function randomOrgApiCallback (requestOptions, postData) {
  const request = https.request(requestOptions, response => {
      «response-callback»
  });

  request.on('error', err => {
    console.error(`Request error: ${err}`);
  });

  request.write(JSON.stringify(postData));
  request.end();
}
```

The code refers to code discussed elsewhere by bracketing a reference to that code with « and ». We'll consider the details of the code designated by `«response-callback»` in Section 2.4.

Many of the modules in NODE.JS (including `http` and `https`) use events to trigger functions asynchronously. For example, the `request` object can *emit* an `error` event when it detects a problem. Not surprisingly, objects that emit events are called *event emitters*. The `request` object returned by `https.request` is an event emitter.

Code can *register* to listen for an event by calling the `on` method of an event emitter. That's what the main function does: registers to receive any `error` event emitted by the `request` object. If the `request` object detects a problem, it will emit the `error` event, passing with it details of the error (usually a JavaScript `Error` object) in the `err` parameter. Our code can then deal with the error.

Finally, we send the request itself by invoking `request.write`, which transmits the `postData` to the server. We can call `write` multiple times to send additional data; here, we only call it once. We indicate that we're done sending data by invoking `request.end`.

2.4 Response Callback

Let's take a look at the `response-callback` code mentioned in Section 2.3. The callback is invoked when your code receives response data from the server. HTTP responses can be received by NODE.JS in multiple "chunks," which we must assemble into a single response message. We use the `chunks` array for this purpose.

If the `debug` flag is `true`, first we print out information for debugging. In particular, we print the HTTP status code (e.g., `200` for "OK") and the headers sent by the server (for which we use our pretty printing function).

Like the event handler for the `error` event on the `request` object (Section 2.3), the response callback registers two event handlers, here on the `response` object.

- The `data` event indicates that NODE.JS has received a chunk of the response. After printing the content of the chunk (if debugging is enabled), this event handler pushes the received data onto the `chunks` array for later processing.

- The `end` event signals that we've received the entire response. The handler responds by concatenating all the chunks into a single response.

```
if (debug) {
  console.log(`STATUS: ${response.statusCode}`);
  console.log(`HEADERS: ${prettyPrintJson(response.headers)}`);
}
response.setEncoding('utf8');
const chunks = [];

response.on('data', chunk => {
  if (debug) {
    console.log(`CHUNK: ${prettyPrintJson(JSON.parse(chunk))}`);
  }
  chunks.push(chunk);
});

response.on('end', () => {
  const content = chunks.join('');
  console.log(`CONTENT (Callback):
  ↪  ${prettyPrintJson(JSON.parse(content))}`);
});
```

2.5 Configuration

The `randomOrgApiCallback` function takes two objects that configure the request.

2.5.1 Node.JS

The `requestOptions` parameter to `randomOrgApiCallback` gives details about the server and the URL for our request. We're connecting to the RESTful API of RANDOM.ORG, a service that returns truly random values (not just pseudo-random values, which is what most "random" functions return). We make an HTTP `POST` request to the `path` defined in this object.

```
const randomOrgRequestOptions = {
  hostname: 'api.random.org',
  method: 'POST',
  path: '/json-rpc/1/invoke',
  headers: { 'Content-Type': 'application/json-rpc' }
};
```

2.5.2 Random.Org

The `postData` parameter of `randomOrgApiCallback` supplies the payload to be sent in the HTTP `POST` request. The RANDOM.ORG API expects a specific JSON object that contains the following fields:

- `method` selects the type of random data; we're asking for integers
- `params` gives details about the type of data; we're requesting five (`n`) integers in the range [1 . . . 100] (`min` . . . `max`).
- `apiKey` is a key supplied by RANDOM.ORG to authorize access to their API. I registered this one for this assignment; please don't abuse it.

```
const generateIntegersPostData = {
  "jsonrpc": "2.0",
  "method": "generateIntegers",
  "params": {
    "apiKey": "d8e8fe5f-4d3c-4d2a-8410-04ced4ec17aa",
    "n": 5,
    "min": 1,
    "max": 100
  },
  "id": 1
};
```

2.6 Invocation

Invoking the function is simple: pass the two parameters discussed in Section 2.5 to the main function (Section 2.3).

```
randomOrgApiCallback(randomOrgRequestOptions, generateIntegersPostData);
```

2.7 Complete Listing

Here is a listing of the entire program. As described in Section 2.1, this code was created automatically by *tangling* the code snippets from throughout this document.

```
const https = require('https');
const debug = false;            // Set true for debug output

function prettyPrintJson (jsonObject) {
  return JSON.stringify(jsonObject, null, 2);
}
function randomOrgApiCallback (requestOptions, postData) {
  const request = https.request(requestOptions, response => {
      «response-callback»
  });

  request.on('error', err => {
    console.error(`Request error: ${err}`);
  });

  request.write(JSON.stringify(postData));
  request.end();
}
const randomOrgRequestOptions = {
  hostname: 'api.random.org',
  method: 'POST',
  path: '/json-rpc/1/invoke',
  headers: { 'Content-Type': 'application/json-rpc' }
};
const generateIntegersPostData = {
  "jsonrpc": "2.0",
  "method": "generateIntegers",
  "params": {
    "apiKey": "d8e8fe5f-4d3c-4d2a-8410-04ced4ec17aa",
    "n": 5,
    "min": 1,
    "max": 100
  },
  "id": 1
};
randomOrgApiCallback(randomOrgRequestOptions, generateIntegersPostData);
```

2.8 Sample Output

Following is the output from one run of the program. The five random integers we requested are buried in the `data` array inside the response object.

```
CONTENT (Callback): {
  "jsonrpc": "2.0",
  "result": {
    "random": {
      "data": [
        54,
        6,
        94,
        18,
        89
      ],
      "completionTime": "2018-09-16 15:51:29Z"
    },
    "bitsUsed": 33,
    "bitsLeft": 249637,
    "requestsLeft": 989,
    "advisoryDelay": 1030
  },
  "id": 1
}
```

3 Code

You may use the code from Section 2.7 to implement the following. The file containing your program *must* be called `async-js-solution.js`.

3.1 Promises

Wrap the callback-based code in a function that implements a promise-based interface. Observe the following requirements:

1. Your implementation should handle both the *fulfilled* and *rejected* states of the promise object.
2. Your promise-based function *must* be called `randomOrgApiPromise`. It should take the same parameters as `randomOrgApiCallback`
3. At the end of your source code, invoke your function as follows:

```
randomOrgApiPromise(randomOrgRequestOptions,
↪  generateIntegersPostData)
  .then(content => console.log(`CONTENT
  ↪  (Promise):\n${prettyPrintJson(JSON.parse(content))}`))
  .catch(error => console.log(`BUMMER: ${error}`));
```

3.2 Asynchronous Functions

Write a function called `randomOrgApiFunction` that uses the `async` and `await` keywords to invoke your promise-based implementation. Your code should:

1. Store the value returned from the async function in a variable called `result`

2. Invoke the following line of code to print the results:

```
console.log(`CONTENT (Async
↪   Function):\n${prettyPrintJson(JSON.parse(result))}`);
```

3. Catch exceptions using the standard JavaScript `try` / `catch` mechanism. When an exception is detected, invoke the following line of code to report the error (stored in the variable `err`):

```
console.log(`Something went wrong: ${err}`);
```

4 Submit

Submit your source file, which *must* be called `async-js-solution.js`, to the course web site.