async-js-promise-hw

September 8, 2019

1 Read

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

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

1.1 Preliminaries

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

[1]:
```
const https = require('https');
const debug = false; // Set to true for debugging output
```

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).

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

1.2 Main Function

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

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

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

1. Sets up to make an HTTPS request using the contents of the `requestOptions`.
2. Asynchronously sends the request.
3. When response data are received, invokes the callback function. Note that HTTP and HTTPS allow responses to be sent in smaller "chunks" so as not to overwhelm the receiver. We have to hang on all chunks and assemble them together when we've received them all.

[3]:
```
function randomOrgApiCallback (requestOptions, postData) {

  const request = https.request(requestOptions, response => {
    // Start of callback code, invoked when HTTPS request completes.

    const chunks = []; // Hold intermediate pieces of overall response.

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

    response.on('data', chunk => {
      // Got some data - hold on to it until we're all done.
      if (debug) {
        console.log(`CHUNK: ${prettyPrintJson(JSON.parse(chunk))}`);
      }
      chunks.push(chunk);
    });

    response.on('end', () => {
      // That's it for new data; here, just print out all the chunks.
      const content = chunks.join('');
      console.log(`CONTENT (Callback):\n${prettyPrintJson(JSON.
↪parse(content))}`);
    });

    // End of callback code.
  });

  // Back to the main thread of control. F
  request.on('error', err => {
    console.error(`Request error: ${err}`);
  });

  // Send the request data.
  request.write(JSON.stringify(postData));
  // Can also SEND multiple chunks, but we're done for now.
  request.end();
}
```

1.3 Events and Event Emitters

Many of the modules in (including `http` and `https`) use events to trigger functions asynchronously.

For example, the `request` object (returned by `https.request` near the top of the code above)

can *emit* an `error` event when it detects a problem. Not surprisingly, JavaScript objects that emit events are called *event emitters*. The `request` object is an event emitter.

1.4 Registering for Events

Code can *register* to listen for an event by calling the `on` method of an event emitter. This code (near the end of the listing above)

```
request.on('error', err => { ... });
```

registers to receive any `error` event emitted by the `request` object while sending the HTTPS request itself.

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.

1.5 Sending a Request

Finally, at the very end of the listing, we send the request itself by invoking `request.write`. This transmits the `postData` to the server. For a large transmission, we may elect to call `write` multiple times, sending a portion of the data each time Here, we only call it once. Regardless of the number of `write` calls, we indicate that we're done sending data by invoking `request.end`.

1.6 Callback Code

Our callback takes up most of the code in the body of the listing above. It sets up event listeners that allow the `request` object to inform your code of the progress of receiving a response to your request.

HTTP responses can be received by 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).

Our callback registers two event handlers:

1. The `data` event indicates that 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.

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

1.7 Configuration

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

[4]:
```
const randomOrgRequestOptions = {
  hostname: 'api.random.org',
  method: 'POST',
  path: '/json-rpc/2/invoke',
```

```
  headers: { 'Content-Type': 'application/json' }
};
```

The `randomOrgRequestOptions` parameter gives details about the server and the URL for our request. We're connecting to the API of https://www.random.org/, a service that returns truly random values (not just pseudo-random values, which is what most "random" functionsreturn). We make an HTTP `POST` request to the `path` defined in this object. The `Content-Type` header that we send tells Random.Org that we want a JSON response.

[5]:
```
const generateIntegersPostData = {
  "jsonrpc": "2.0",
  "method": "generateIntegers",
  "params": {
    "apiKey": "ccba1f8a-21ab-48c6-b62d-e8d34bb19c36",
    "n": 5,
    "min": 1,
    "max": 100
  },
  "id": 1
};
```

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

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

1.8 Invocation

Invoking the function is simple: pass the two parameters just discussed to the main function.

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

[6]:
```
randomOrgApiCallback(randomOrgRequestOptions, generateIntegersPostData);
```

```
CONTENT (Callback):
{
  "jsonrpc": "2.0",
  "result": {
    "random": {
      "data": [
        10,
        68,
        76,
        79,
        36
```

```
      ],
      "completionTime": "2019-09-09 02:59:02Z"
    },
    "bitsUsed": 33,
    "bitsLeft": 249670,
    "requestsLeft": 990,
    "advisoryDelay": 990
  },
  "id": 1
}
```

2 Assignment

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 exactly as follows:

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

You may use the code in this write-up as the basis for your solution.

3 Submit

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