Simple API server with file-based store

You will build a simple API server that serves a list of products and allows you to add a product.

Find the products.json file in the exercise/01-api-server folder. This has a JSON array of products. Every product has a unique id and more details.

Create a simple HTTP server that serves the following

1. GET /products
   This returns a JSON array of all products

2. POST /products with JSON data of a new product sent in HTTP request
   This extracts the new product's details (use anybody to extract JSON from request body), generates a unique id for the product, and adds it to the products.json file (it is recommended you also maintain an in-memory copy of the products.json file – so you need to update the in-memory copy as well as the products.json file).

3. Any other endpoint returns an error message saying "Cannot <METHOD> <ENDPOINT>". For example, GET /abc returns as response "Cannot GET /abc".

Make sure to handle all error conditions and send HTTP response with appropriate error code.

The following functions will help. Please Google for details of their usage.

1. JSON.parse()
2. JSON.stringify()
3. res.writeHead() where res is the server response object of Node.js HTTP server

Suggested implementation

You are free to implement the above requirement in whatever way you deem fit. One design is shown below. This implementation will also serve as a gentle introduction to Express.js framework as the approach is somewhat similar to Express.js way of approaching the problem of routing (though much simplified).

1. ./db.js
   Define an exported object (say DB) with
   - getProducts(cb) – Gets the products and passes them to callback function cb
   - postProduct(product, cb) – Gets the products details (a JS object), generates a unique number id, and adds product to in-memory copy of products array and the products.json file. The callback is called and passed error details on error (if any – for example product could not be added to the products.json file), and product object (with id) otherwise.
2. ./router.js
   - Maintains an array with details of routes that are handled by the application. For example, it would maintain a routes array like so for this application.

     [

       {

```
        method: 'get',
        route: '/products',
        handler: function( req, res ) {
                // does something when HTTP request for GET /products is received
            }
    },
      {
        method: 'post',
        route: '/products',
        handler: function( req, res ) {
                // does something when HTTP request for POST /products is received
            }
    }
]
```

This array may be hard-coded, or generated via a method defined on a Router object (which is exported from this module). For example, you may define a function like the following to add routes to the above array

```
const add = function( method, route, handler ) {
    routes.push({
        method: method.toLowerCase(),
        route: route,
        handler: handler
    });
};
```

The Router object (which is exported), can have a method like Router.handle(req, res) which goes through the routes array calls the handler function of the route matching the requested route.

For example, if POST /products is received, the handler for that route (handler in second item of the routes array above) is executed (and passed the req, res objects). The handlers need to send appropriate responses after getting products array / passing new product detail and getting confirmation of product addition (via callbacks of DB module methods)

Note: You can extract the method from req.method, and route from req.url.

3. ./routes/products.js
   This adds the routes for GET /products and POST /products by calling the add method of the Router module above. The handlers for each endpoint make use of DB module to get products, and add a new product. Use anybody to extract JSON data passed in request body (in case of POST /products).

4. ./server.js
   This script requires the ./routes/products script (so that routes are added to Router). It then creates and starts an HTTP server – the request handler simply calls Router.handle() passing it request and response objects.