F28WP Web programming
Lab 3
Node.js and MySQL

IMPORTANT: In this Lab, you will only use online software tools (MySQL, PhpMyAdmin, and Code Sandbox with Nodejs). You do not need to install any software on your computer. You do not need to run any command on the terminal of your computer.

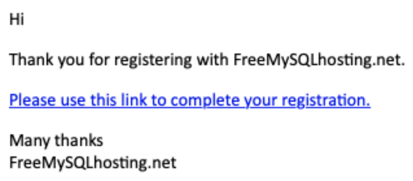
This lab deals with server-side programming using node.js. Using the **express** module, you create a server that delivers dynamic web pages. You develop basic functionalities of an online shopping application. Products and clients are stored in a MySQL database. The server uses EJS module to generate HTML.

1. Tools (0 marks)

To work with this lab, you need to have access to a MySQL server and node.js programming environment. This section explains ways of getting access to these tools.

1.1 MySQL server

For this lab, you need to use MySQL. The easiest (**and recommended**) way is to use an online MySQL server provider. There are several such servers on the internet. For example, sign-up to https://www.freemysqlhosting.net. You will receive a link by email to set up your password.



Hi

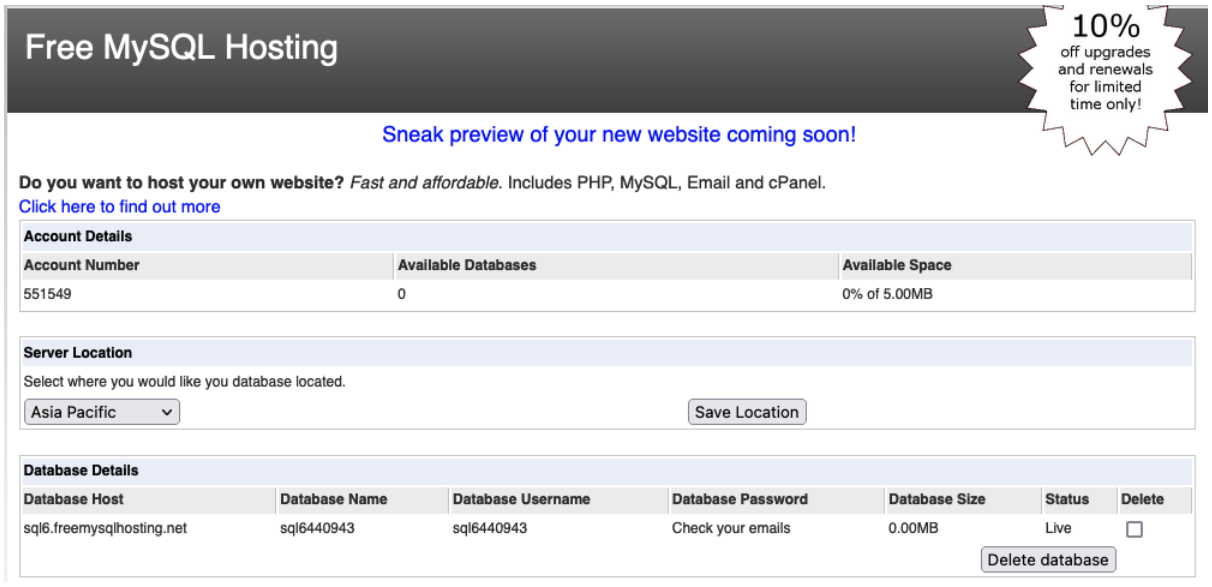
Thank you for registering with FreeMySQLhosting.net.

Please use this link to complete your registration.

Many thanks
FreeMySQLhosting.net

Please note that freemysqlhosting.et might ask for a subscription after a test period. Therefore, you might choose another online MySQL server.

You will have the possibility of creating a database on this server (click create database):



Free MySQL Hosting

10% off upgrades and renewals for limited time only!

Sneak preview of your new website coming soon!

Do you want to host your own website? *Fast and affordable.* Includes PHP, MySQL, Email and cPanel.
Click here to find out more

Account Details

Account Number	Available Databases	Available Space
551549	0	0% of 5.00MB

Server Location

Select where you would like you database located.

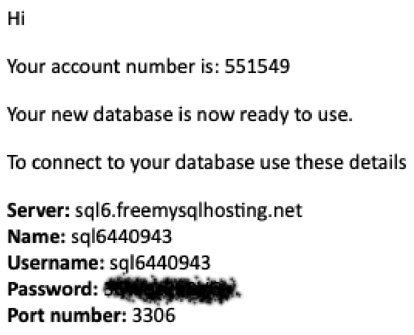
Asia Pacific | Save Location

Database Details

Database Host	Database Name	Database Username	Database Password	Database Size	Status	Delete
sql6.freemysqlhosting.net	sql6440943	sql6440943	Check your emails	0.00MB	Live	☐

Delete database

You will receive an email with details about the database:



Hi

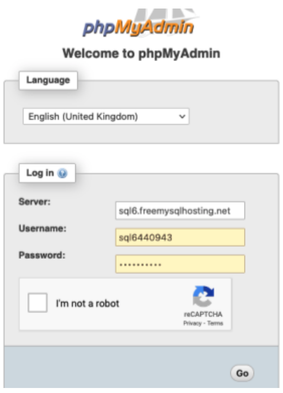
Your account number is: 551549

Your new database is now ready to use.

To connect to your database use these details

Server: sql6.freemysqlhosting.net
Name: sql6440943
Username: sql6440943
Password: [redacted]
Port number: 3306

Now you can administer your database using the online phpMyAdmin (click on “Follow this link for phpMyAdmin”).



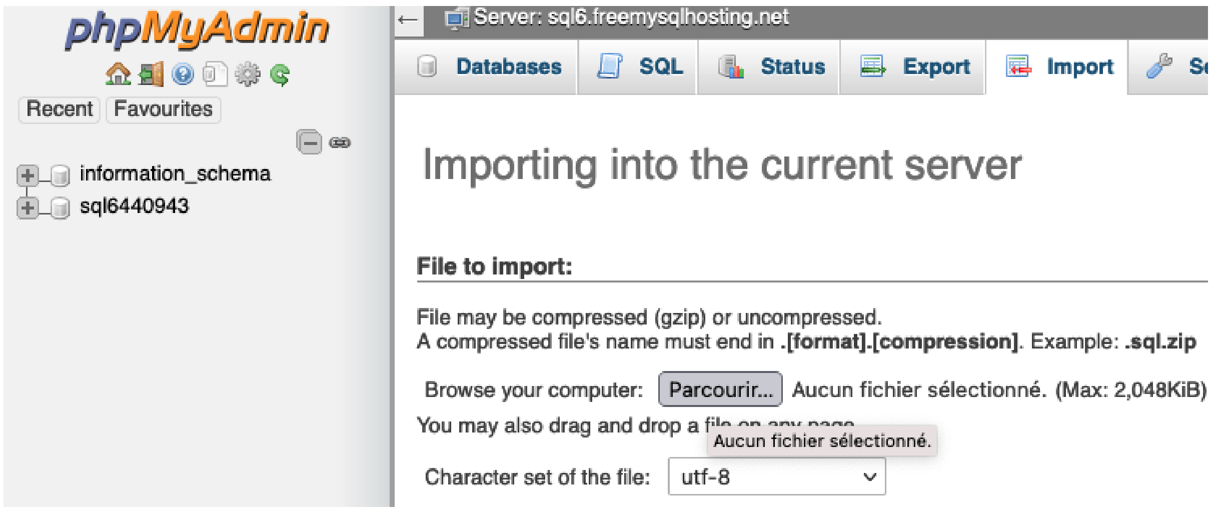
Welcome to phpMyAdmin

Language: English (United Kingdom)

Log in

Server: sql6.freemysqlhosting.net
Username: sql6440943
Password: ••••••••••

Download and edit the file https://github.com/hbatatia/lab03-nodejs/blob/main/sales.sql. Find the line “USE ….”. Replace the database by the name of your database. Import the file using phpMyAdmin. You have now the **sales** database. Explore the tables (click the + sign to the left of the database name) and the data (click the table names).



Importing into the current server

File to import:

File may be compressed (gzip) or uncompressed.
A compressed file's name must end in **.[format].[compression]**. Example: **.sql.zip**

Browse your computer: Parcourir... Aucun fichier sélectionné. (Max: 2,048KiB)

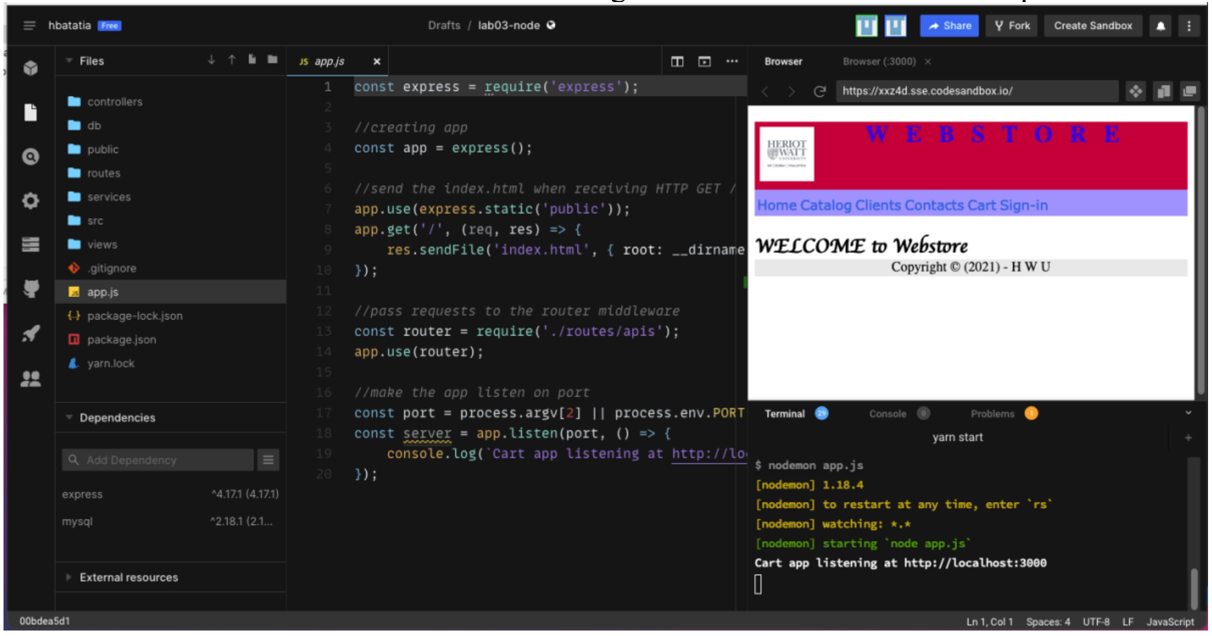
You may also drag and drop a file on any page.

Character set of the file: utf-8

You should not make any change to the database.

1.2 Node.js

The server-side development requires **Node.js**. All students (in lab rooms or on their own computers) must use node.js online. There is no need to install it on your computer. There are several online development environments offering node.js, such as SlackBlitz, Cloud9 IDE… We recommend using **codesandbox** which connects easily to GitHub. Sign-up to **codesandbox.io** and create a new sandbox using the "**node HTTP server**" template.

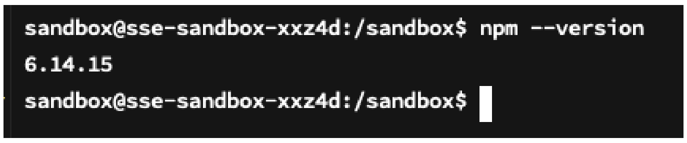


```
const express = require('express');

//creating app
const app = express();

//send the index.html when receiving HTTP GET /
app.use(express.static('public'));
app.get('/', (req, res) => {
    res.sendFile('index.html', { root: __dirname
});

//pass requests to the router middleware
const router = require('./routes/apis');
app.use(router);

//make the app listen on port
const port = process.argv[2] || process.env.PORT
const server = app.listen(port, () => {
    console.log(`Cart app listening at http://lo
});
```

```
$ nodemon app.js
[nodemon] 1.18.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node app.js`
Cart app listening at http://localhost:3000
```

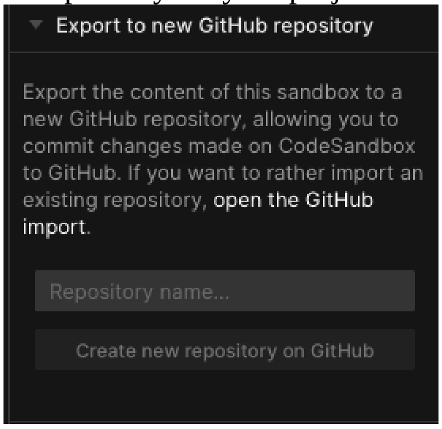
Codesandbox provides a built-in terminal for running Linux commands. Please note the terminal widow in the lower-right part of codesandbox window. Oe of the commands that we are going to run from this same terminal (not the terminal of your computer) is **npm**.

Npm is a package management tool that is shipped with node.js. We will use it to install a few node modules in this lab. From a terminal (online or locally), check the version of your npm (npm –version)



```
sandbox@sse-sandbox-xxz4d:/sandbox$ npm --version
6.14.15
sandbox@sse-sandbox-xxz4d:/sandbox$ 
```

Your application must be stored on GitHub. If you are working with codesandbox.io, click the GitHub icon () to create a repository for your project.

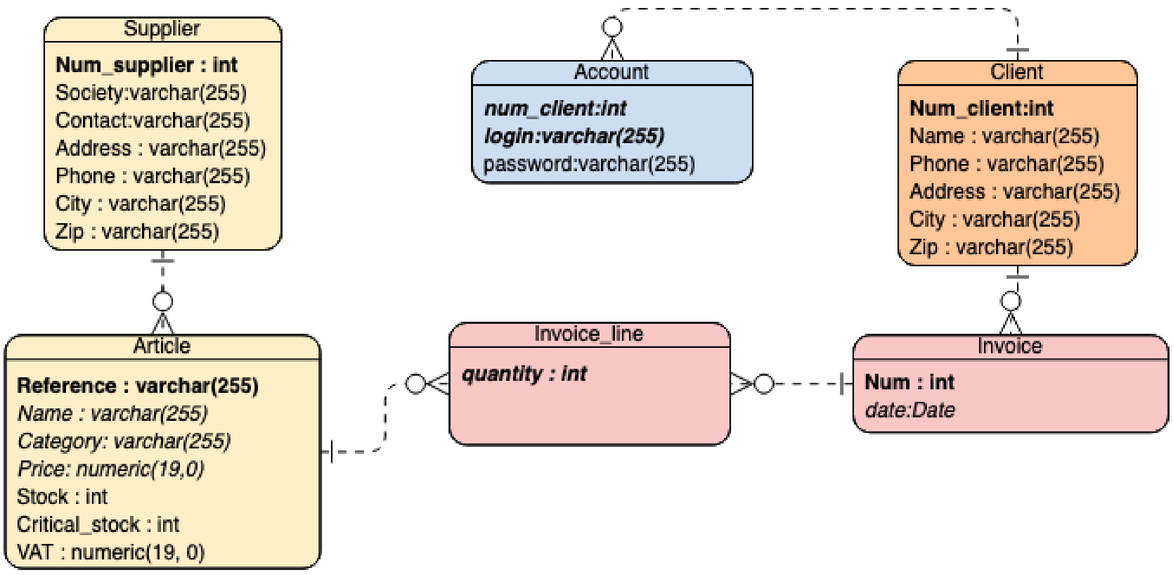


Export to new GitHub repository

Export the content of this sandbox to a new GitHub repository, allowing you to commit changes made on CodeSandbox to GitHub. If you want to rather import an existing repository, open the GitHub import.

2. Application specifications (0 marks)

The objective is to develop a simple online shopping web site. The site
The application allows to browse a catalogue of products (articles) and to visualize the list of clients!
The full database (sales.sql) is given and should be created in a MySQL server (refer to section 2.1). It contains the following tables: article, client, supplier, invoice, invoice_line, guest, draft. The following is a partial entity-relationship model (ERM) of the database (attributes might have different names)



This lab deals mainly with the **client** and **article** tables.
Explore the data using the select query or the table visualization. There are 62 clients and 82 articles. There are also many **invoices** and their corresponding **lines**.

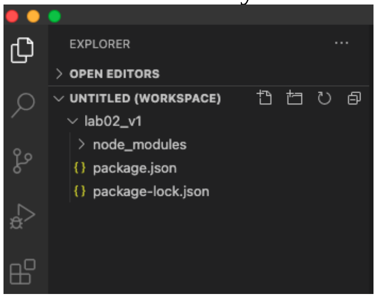
The application must display the catalogue of products; the user should be able to access details of a given product by clicking on its reference or name. The user can add products to their shopping cart. They can modify the quantity of a given product in the cart or delete it. They can also clear the cart. In addition, they can access information about their account.

3 Create the first web server (2 marks)

Using your code editor, create a file named **package.json** with the following content:

```
{
   "name": "lab02_cart",
   "version": "1.0.0",
   "description": "",
   "main": "app.js",
   "keywords": [],
   "author": "",
   "license": "ISC"
}
```

This file describes the application (name, version, main script…).
We need to install the **express** node module, which will allow us to create the web server. In the terminal, change to the directory containing your project and run the following command:
npm install express
In the project folder, note that this has created a new folder called **node_modules**. The file **package.json** has also changed to show the decency on the installed module.

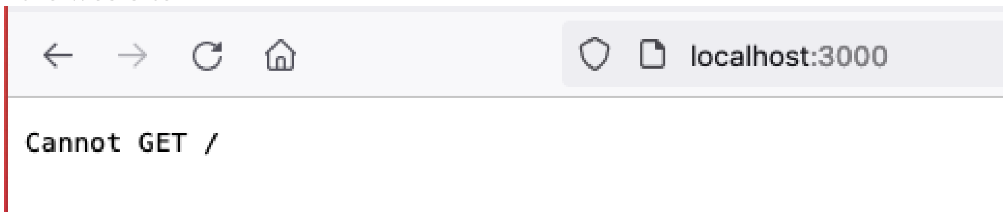


Create a file called app.js with the following content, that loads the express module, create an app and a http server, make the server listen on port 3000.

```
const express = require('express');

//creating app
const app = express();

//make the app listen on port
const port = process.argv[2] || process.env.PORT || 3000;
const server = app.listen(port, () => {
   console.log(`Cart app listening at http://localhost:${port}`);
});
```

Run the server from the terminal by typing:
node app.js

From a web browser, type http://localhost:3000/. The server replies but cannot find the root folder of the web site

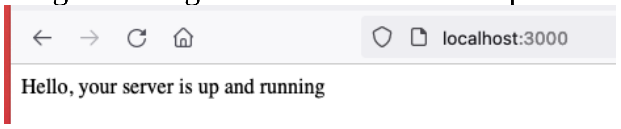


```
Cannot GET /
```

Add the following to app.js (just after const app = …)

```
//send an HTTP response when receiving HTTP GET /
app.get('/', (req, res) => {
   res.send("Hello, your server is up and running");
});
```

Stop the server and run it again. Using a browser check the response:



Hello, your server is up and running

Notice that every time you make a change to your code, you need to stop and re-run the server. A way to automate restaring the server after every change is to use nodemon module. Install nodemon (**npm i nodemon**). Modify package.json to contain the following:

```
{
   "name": "lab02_cart",
   "version": "1.0.0",
   "description": "",
   "main": "app.js",
   "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1",
      "dev": "nodemon app.js",
      "debug": "nodemon --inspect app.js"
   },
   "keywords": [],
   "author": "",
   "license": "ISC",
   "dependencies": {
      …
   }
}
```

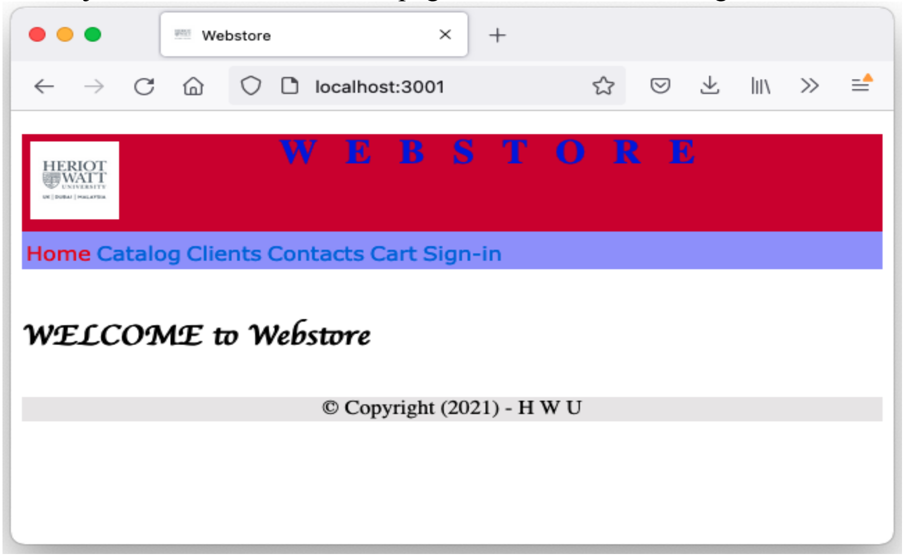
Now, you can run you server once from the terminal using

npm run dev
It will restart the server whenever a change is made.

4. Send a web page (1 marks)

We will now modify our server to send a web page (HTML, css and image files) to the client.
In a subfolder named **public**, create the files index.html, hwu.css, hwu.png (see resources in the appendix). This defines your web site. The page consists of a banner (for the logo and the title), a navigation bar, a content area, and a footer. Each part is a different div with its identifier.

The objective is to make the web page look like the following



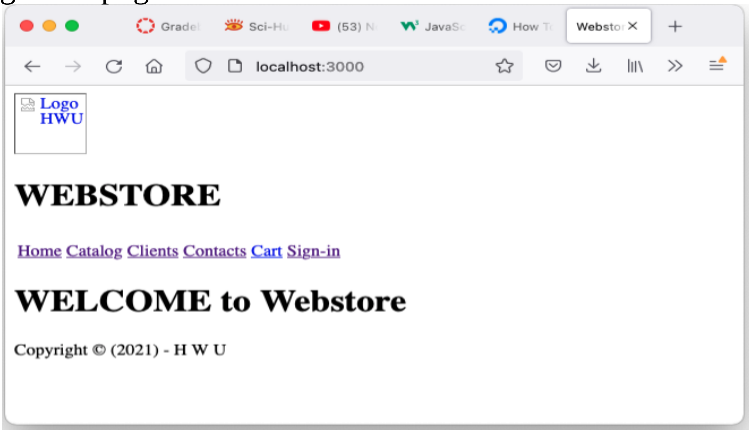
WELCOME to Webstore

© Copyright (2021) - H W U

In the app.js file, change the statement that handles the "HTTP request /" to be as follows:

```
//send the index.html when receiving HTTP GET /
app.get('/', (req, res) => {
   res.sendFile('public/index.html', { root: __dirname });
});
```

Run the server and connect. The page has only the content and no CSS styles have been applied. This is because, there is no code in the app.js to send the css file. The same remark applies to the image hwu.png.



WEBSTORE

Home Catalog Clients Contacts Cart Sign-in

WELCOME to Webstore

Copyright © (2021) - H W U

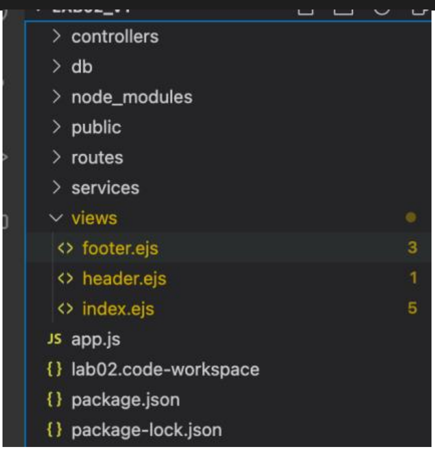
In order to avoid adding an app.get(…) statement for each file to send to client, we will use the static module to send automatically ay static file (html, css, images, JavaScript…).
Modify the previous code bloc to become

```
app.use(express.static('public'));
app.get('/', (req, res) => {
  res.sendFile('index.html', { root: __dirname });
});
```
Now, the server should send the html, the CSS and image files in the public directory.

5. Using ejs templates (1 mark)

Usually, HTML files making a web site repeat the same layout. Most often, the header and footer sections are identical in all pages. To avoid rewriting the same content multiple times, we can split the HTML into different files and use EJS to include them.
EJS uses the folder views to search for .ejs files. Split your index.html file into three ejs files under views:
- header.ejs: contains HTML content until the content div (excluded). It does not have to close the tags (such as html, body…)
- footer.ejs: contains the footer div and the closing tags for body and html
- index.ejs: includes header.ejs, contains the content div, and includes footer.ejs (note the include directive surrounded by specific tag symbols (<%- … -%>)

```
<%- include('header'); -%>
  <div class="content">
    <h1 id="welcome">WELCOME to Webstore</h1>
  </div>
<%- include('footer'); -%>
```



```
> controllers
> db
> node_modules
> public
> routes
> services
v views
  <> footer.ejs      3
  <> header.ejs      1
  <> index.ejs       5
JS app.js
{} lab02.code-workspace
{} package.json
{} package-lock.json
```

To process ejs in our server, we need to load ejs and to render the ejs file. Install ejs package (npm i ejs). In app.js, include the following:

```
//handling static HTML and EJS templates
app.use(express.static('public'));
app.set('view engine', 'ejs');
app.get('/', (req, res) => {
  res.render('index'); //no need for ejs extension
});
```
Test your server. It should show the same content. Actually, ejs processed the .ejs files and generated HTML that has been sent to the client.
Create a contacts.ejs file with the same structure as index.ejs. Add in names, address, phone, and emails of contacts.

To be able to send this ejs file to the client on request, add the following to app.js

```
//route for contacts
app.get('/contacts', (req, res) => {
   res.render('contacts');
});
```

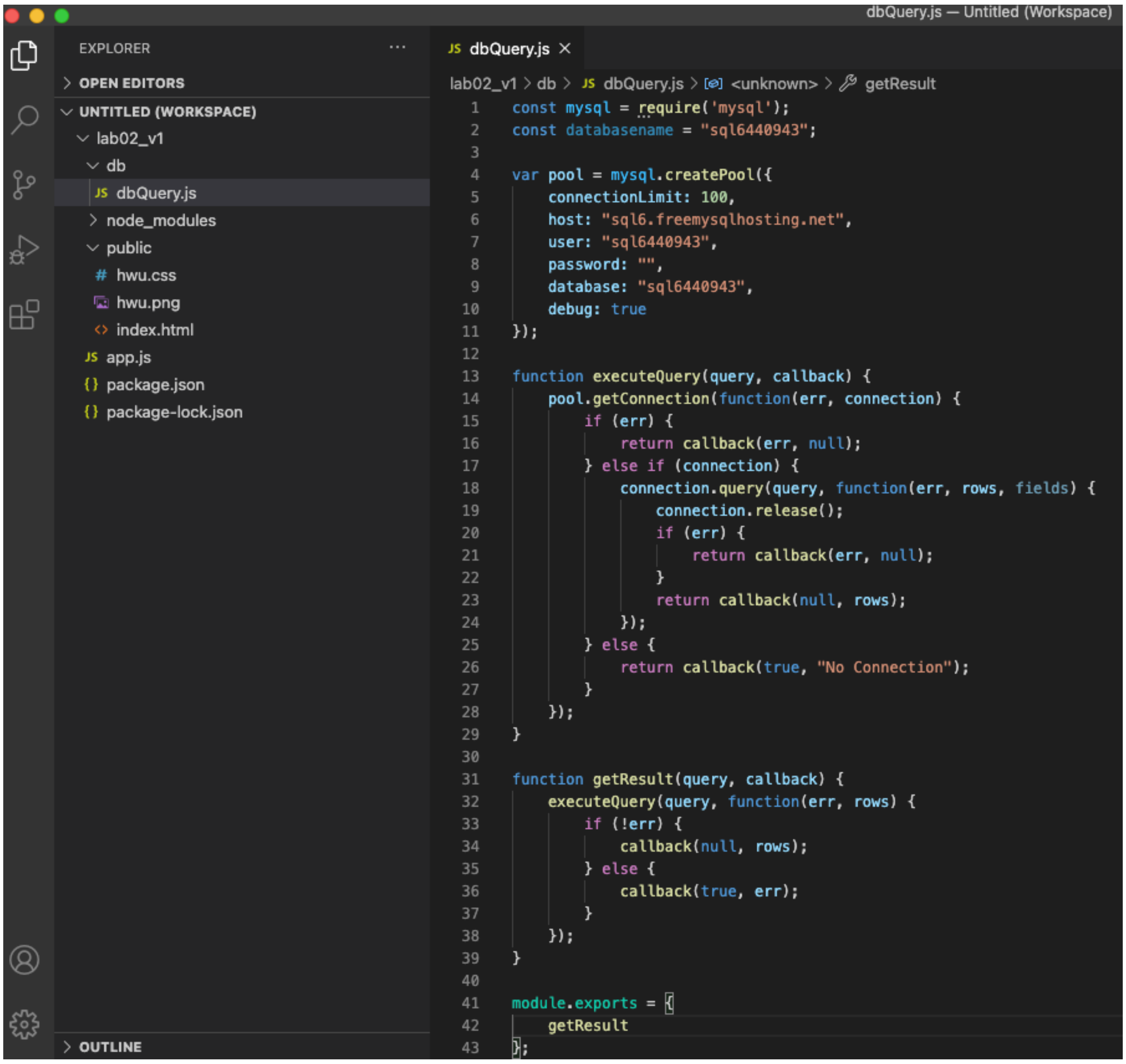
In a similar way, create:
- register.ejs: contains information for a client to sign-up (usrname, password, name, phone, address…). The form should contain all attributes from the Client table. The form should generate a HTTP POST requests to the route /api/register.
- login.ejs: has a login and password fields and sends a HTTP GET query to the route (/api/login).

You do not need to implement the server-side code for /api/register and /api/login.

6. Creating catalogue (2 marks)

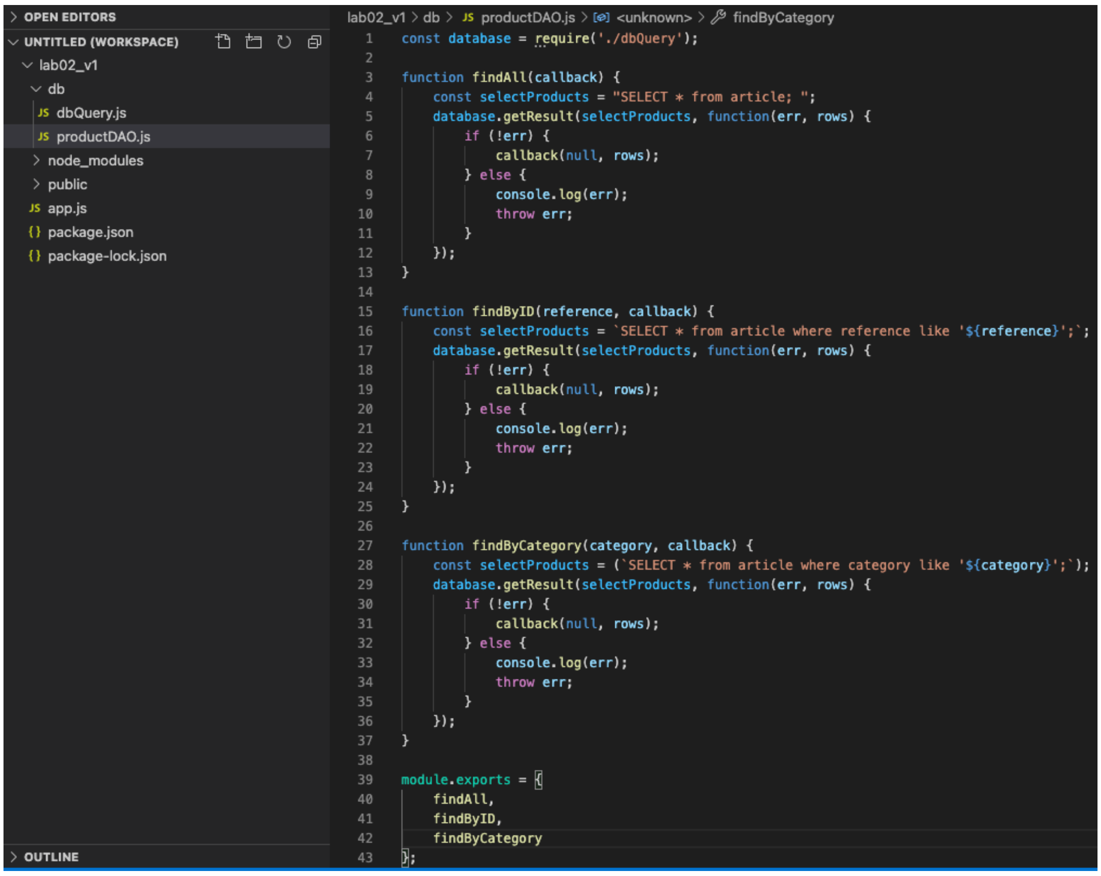
At this stage, we will create node.js code to display the list of products when the user clicks **Catalogue**. The Catalogue link in the web page must be associated to the hyperlink "**GET /api/catalogue**."
On the server-side, we first need to create a layer responsible for querying the database.
We start by installing the node module mysql (npm i mysql) in the project folder.

In app.js, we add the module dbQuery.js under the folder db (see resource 4):



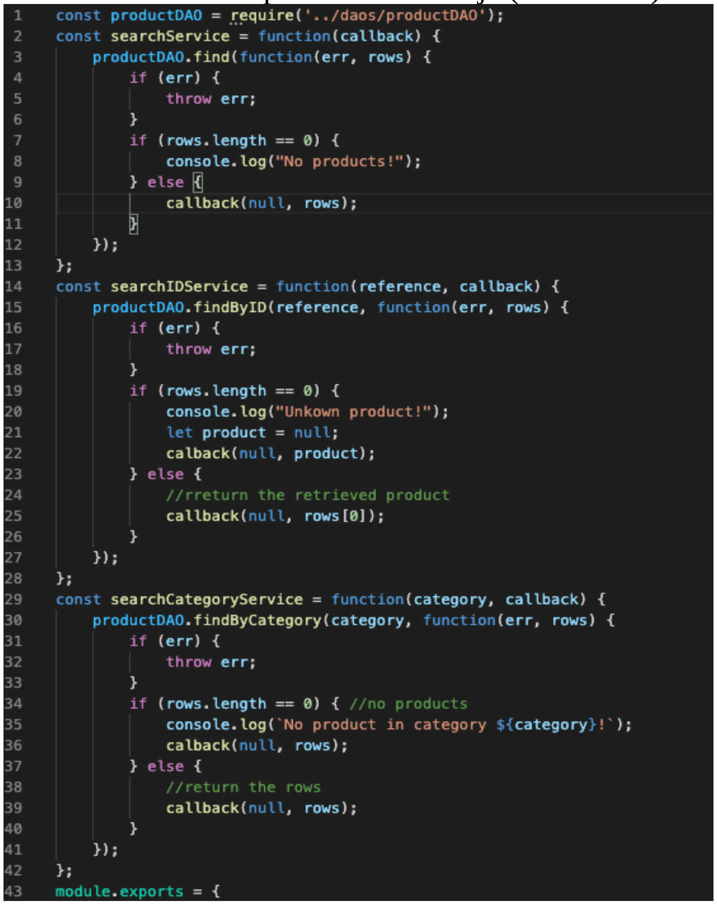
```javascript
const mysql = require('mysql');
const databasename = "sql6440943";

var pool = mysql.createPool({
    connectionLimit: 100,
    host: "sql6.freemysqlhosting.net",
    user: "sql6440943",
    password: "",
    database: "sql6440943",
    debug: true
});

function executeQuery(query, callback) {
    pool.getConnection(function(err, connection) {
        if (err) {
            return callback(err, null);
        } else if (connection) {
            connection.query(query, function(err, rows, fields) {
                connection.release();
                if (err) {
                    return callback(err, null);
                }
                return callback(null, rows);
            });
        } else {
            return callback(true, "No Connection");
        }
    });
}

function getResult(query, callback) {
    executeQuery(query, function(err, rows) {
        if (!err) {
            callback(null, rows);
        } else {
            callback(true, err);
        }
    });
}

module.exports = {
    getResult
};
```

This module is responsible for creating a connection to the database server, submit any SQL query and get the results. These are returned through the callback mechanism.
Next, we create the productDAO.js module that implement the queries related to products we need in our application (resource 5).



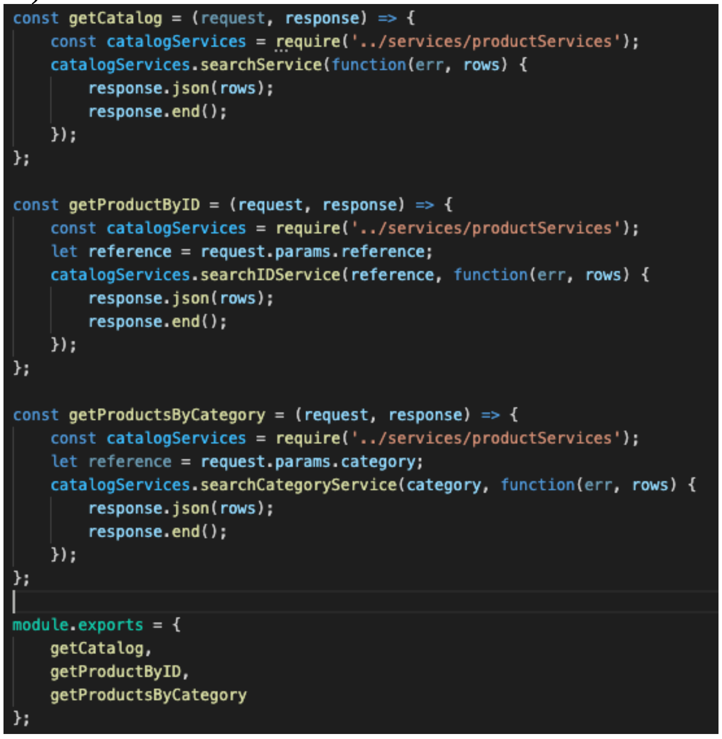
```javascript
const database = require('./dbQuery');

function findAll(callback) {
    const selectProducts = "SELECT * from article; ";
    database.getResult(selectProducts, function(err, rows) {
        if (!err) {
            callback(null, rows);
        } else {
            console.log(err);
            throw err;
        }
    });
}

function findByID(reference, callback) {
    const selectProducts = `SELECT * from article where reference like '${reference}';`;
    database.getResult(selectProducts, function(err, rows) {
        if (!err) {
            callback(null, rows);
        } else {
            console.log(err);
            throw err;
        }
    });
}

function findByCategory(category, callback) {
    const selectProducts = (`SELECT * from article where category like '${category}';`);
    database.getResult(selectProducts, function(err, rows) {
        if (!err) {
            callback(null, rows);
        } else {
            console.log(err);
            throw err;
        }
    });
}

module.exports = {
    findAll,
    findByID,
    findByCategory
};
```

This module needs to be used by a higher-level module that searches for the products using different criteria. This module is called productServices.js (resource 6):



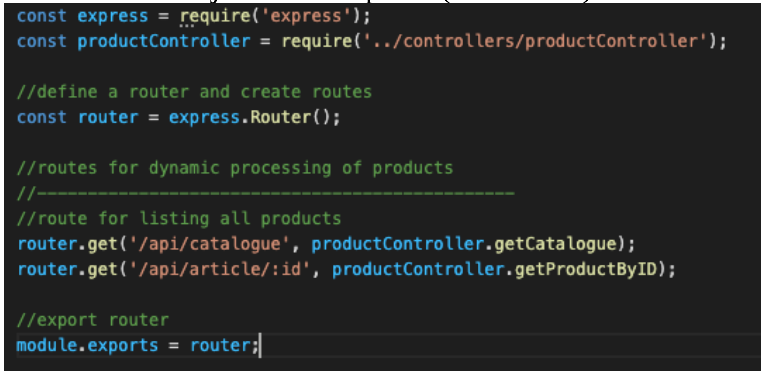
```javascript
const productDAO = require('../daos/productDAO');
const searchService = function(callback) {
    productDAO.find(function(err, rows) {
        if (err) {
            throw err;
        }
        if (rows.length == 0) {
            console.log("No products!");
        } else {
            callback(null, rows);
        }
    });
};
const searchIDService = function(reference, callback) {
    productDAO.findByID(reference, function(err, rows) {
        if (err) {
            throw err;
        }
        if (rows.length == 0) {
            console.log("Unkown product!");
            let product = null;
            calback(null, product);
        } else {
            //rreturn the retrieved product
            callback(null, rows[0]);
        }
    });
};
const searchCategoryService = function(category, callback) {
    productDAO.findByCategory(category, function(err, rows) {
        if (err) {
            throw err;
        }
        if (rows.length == 0) { //no products
            console.log(`No product in category ${category}!`);
            calback(null, rows);
        } else {
            //return the rows
            callback(null, rows);
        }
    });
};
module.exports = {
```

In a web application, the control layer is responsible for handling the HTTP request to extract parameters (possibly check their validity), and sending back the response to the client. I our case, the module productController.js is responsible for processing HTTP requests related to products (Resource 7).



```
const getCatalog = (request, response) => {
    const catalogServices = require('../services/productServices');
    catalogServices.searchService(function(err, rows) {
        response.json(rows);
        response.end();
    });
};

const getProductByID = (request, response) => {
    const catalogServices = require('../services/productServices');
    let reference = request.params.reference;
    catalogServices.searchIDService(reference, function(err, rows) {
        response.json(rows);
        response.end();
    });
};

const getProductsByCategory = (request, response) => {
    const catalogServices = require('../services/productServices');
    let reference = request.params.category;
    catalogServices.searchCategoryService(category, function(err, rows) {
        response.json(rows);
        response.end();
    });
};

module.exports = {
    getCatalog,
    getProductByID,
    getProductsByCategory
};
```

Finally, the apis.js module defines all the routes for which our server provides a response. For this purpose, we use the router object from express (resource 8):

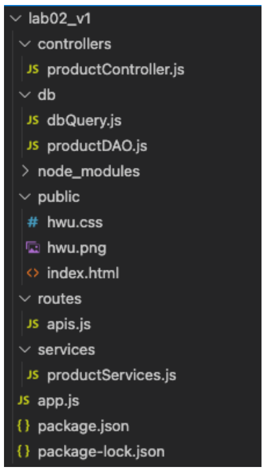


```
const express = require('express');
const productController = require('../controllers/productController');

//define a router and create routes
const router = express.Router();

//routes for dynamic processing of products
//-----------------------------------------------
//route for listing all products
router.get('/api/catalogue', productController.getCatalogue);
router.get('/api/article/:id', productController.getProductByID);

//export router
module.exports = router;
```

For the server to pass the HTTP requests to the APIs, we need to add the **router** to app.js:

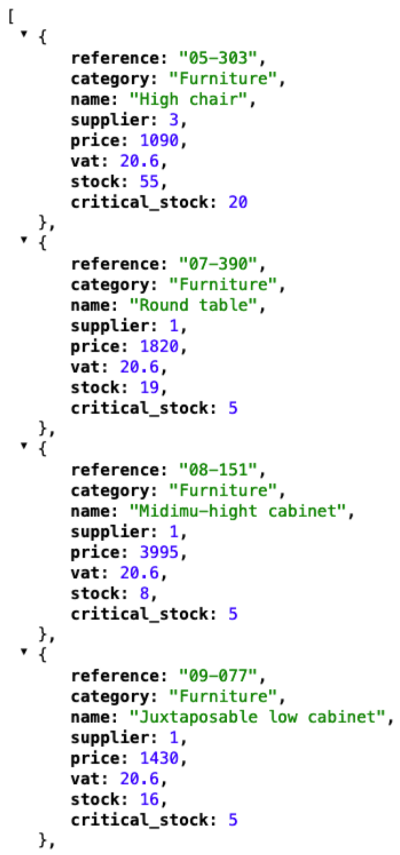
```
//pass requests to the router middleware

const router = require('./routes/post');

app.use(router);
```

The resulting application has the following structure



```
lab02_v1
  controllers
    productController.js
  db
    dbQuery.js
    productDAO.js
  node_modules
  public
    hwu.css
    hwu.png
    index.html
  routes
    apis.js
  services
    productServices.js
  app.js
  package.json
  package-lock.json
```

Running this application and clicking on Catalogue will bring the list of products as a JSON array



```
[
 ▾ {
      reference: "05-303",
      category: "Furniture",
      name: "High chair",
      supplier: 3,
      price: 1090,
      vat: 20.6,
      stock: 55,
      critical_stock: 20
   },
 ▾ {
      reference: "07-390",
      category: "Furniture",
      name: "Round table",
      supplier: 1,
      price: 1820,
      vat: 20.6,
      stock: 19,
      critical_stock: 5
   },
 ▾ {
      reference: "08-151",
      category: "Furniture",
      name: "Midimu-hight cabinet",
      supplier: 1,
      price: 3995,
      vat: 20.6,
      stock: 8,
      critical_stock: 5
   },
 ▾ {
      reference: "09-077",
      category: "Furniture",
      name: "Juxtaposable low cabinet",
      supplier: 1,
      price: 1430,
      vat: 20.6,
      stock: 16,
      critical_stock: 5
   },
```

7. Using ejs to display dynamic content (2 marks)

The previous result lacks the structure of our web page. We need to pass the JSON object to ejs to generate proper HTML to send to the server. Create **catalogue.ejs** with the content from resource 9. Note the foreach loop iterates through all items in the array products (sent by the controller, see next) to create one table row for each product, with one column for each attribute:

```
<% products.forEach(product => { %>
            <tr>
              <th scope="row"><%= product.name %></th>
              <td>
                <a href="/api/article/<%= product.reference %>">
                  <%= product.reference %>
                </a>
              </td>
              <td><%= product.category %></td>
              <td><%= product.price %>AED</td>
            </tr>
<% }) %>
```

Also note the tags for the foreach line, and the attributes. These are different. <% … %> is used for code that executes and does not display content. <%= %> is for displaying some content.
We now need to send the JSON table of products to the ejs file. For that, we need to replace sending JSON by rendering the ejs file. In productControl.js, in the function getCatalogue, find

```
    response.json(rows);
    response.end();
```

Replace it with (note we are passing the rows array of products as an object called **products**):

```
response.render('catalogue', { products: rows });
```

Test your server and check that the web page shows the list of products using the css style defined before.
Similarly, create **article.js** that displays the full details of any article selected by the user (by clicking its reference). Develop the code for the route /api/article/:id to select the article having the id given by the user and pass it o to article.ejs to generate the appropriate html.

8. Client login (1 mark)

Resource 10 provides a module to run CRUD queries on clients, including searching with different criteria.
Resource 11 consists of a module providing login and registration services. Both services rely on the module **bcryptjs** (that you should install) to crypt password and compare hashes.
Add the two routes to the router and test the server.

```
//routes for dynamic processing of clients
//-----------------------------------------------
//route for registration
router.post('/api/register', clientController.registerControl);
//route for login
router.post('/api/login', clientController.loginControl);
```

The response provided by these two apis are displayed in a new page, without respecting the page layout defined before. Change the response by using ejs to create a web page that respects the same layout.

9. Additional features (1 mark if any feature is developed)

Using the same approach adopted in this lab, add the following features:

- List all clients when the user is logged in and has admin privilege.
- Display details of a client when the user clicks their name
- Allow the user to add a selected product with a chosen quantity to a shopping cart. The cart should be a list of objects stored in the session (like the user login). Each line of the cart will have the reference of the product and the selected quantity.
- Develop the functionalities to modify the quantity of an item in the shopping cart, to delete an item from the shopping cart and to clear the shopping cart.
- Display the cart (when the user clicks on cart) by showing the price of each item including vat and the total.
- When the user checks out, they are asked to log in and the cart is displayed with the total amount. When the user confirms, the cart is inserted in the database. You need to add two new tables: one (named cart) with the cart data, id, and the client id. The second (named cartLine) will have product reference, quantity and price including vat.

Resources

Resource 1: initial index.html
https://github.com/hbatatia/lab03-nodejs/blob/main/public/index.html

Resource 2: CSS file – hwu.css

https://github.com/hbatatia/lab03-nodejs/blob/main/public/hwu.css

Resource 3: HWU logo – hwu.png
https://github.com/hbatatia/lab03-nodejs/blob/main/public/hwu.png

Resource 4: database query module
https://github.com/hbatatia/lab03-nodejs/blob/main/db/dbQuery.js

Resource 5: module to query product table
https://github.com/hbatatia/lab03-nodejs/blob/main/db/productDAO.js

Resource 6: module to search products
https://github.com/hbatatia/lab03-nodejs/blob/main/services/productServices.js

Resource 7: HTTP controller module for products
https://github.com/hbatatia/lab03-nodejs/blob/main/controllers/productController.js

Resource 8: module that handle routes to the APIs of our server
https://github.com/hbatatia/lab03-nodejs/blob/main/apis/routes.js

Resource 9: catalogue.ejs
https://github.com/hbatatia/lab03-nodejs/blob/main/views/catalogue.ejs

Resource 10: module to query for clients
https://github.com/hbatatia/lab03-nodejs/blob/main/db/clientDAO.js

Resource 11: Module providing login and registration services
https://github.com/hbatatia/lab03-nodejs/blob/main/services/clientServices.js

Resource 12: Module for controlling the login and registration
https://github.com/hbatatia/lab03-nodejs/blob/main/controllers/clientController.js

IMPORTANT: Each lab must be demonstrated in one of the allocated lab sessions (either virtually or face-to-face). After you have demonstrated your lab work you should submit a copy of your work on CANVAS as evidence (your mark is allocated/given for the demonstration session). The submitted copy, should be a short report and include, your name, date, which lab, who you demonstrated to, and any screenshots/code/links.