# Lab: Shop Stop

This lab is part of [“ExpressJSFundamentals” course @ SoftUni](https://softuni.bg/trainings/1642/expressjs-fundamentals-may-2017). The lab itself will be distributed into several parts each containing more concrete information and guide steps on how to develop the functionality specified below.

“*Shop Stop*” is very simple **product** **catalog** website (like [OLX](https://www.olx.bg/), [Amazon](https://www.amazon.com/) – but simpler 😊). The application will consist of **users**, **products** and **categories**. Each **user** can **register**, **login** and **logout**. **Users** also can **create**, **buy**, **edit** or **delete** a **product**. Each **product** has **a** **category** in which it is specified. Site will implement of **searching** for a **product** by the product’s **name** or **category**.

### Table of Contents

* **Part I - Laying the Project Fundamentals (current)**
* Part II – Using Third-Party Modules
* Part III – Defining Database Models and Relations
* Part IV – Advanced Functionality
* Part V – User Authentication

## Project Specification

Design and implement a **“Shop Stop” web application** (containing routing and multiple web pages) using HTML5, CSS3 and Node.js. It must contain the following functionality:

### Functionality

* **User Login**
  + Login in current application using username and password of already registered user.
* **User Register**
  + Register a new user by providing username and password.
* **User Logout**
  + Logouts from the application.
* **Product Create**
  + Creates a new product, makes currently logged in user as it’s publisher.
  + Data must be saved in some sort of database.
* **Product Buy**
  + Products may be bought by any user. One product could not be bought more than one time.
* **Product Edit**
  + Edits product's information. Changes must be persisted in database.
* **Product Delete**
  + Deletes specific product from database.
* **Category Create**
  + Create a category which later on can be picked when a new product is created (uploaded).
* **Products by Name**
  + List all products which name contains given text
* **Products by Category**
  + List all products are in particular category (use the name of the category)

This is how your app may look like (before user login):

|  |  |  |
| --- | --- | --- |
|  | | |
|  | |  |
|  | | |
|  |  | |

This is how your app may look like (after user login):

|  |  |  |
| --- | --- | --- |
|  | | |
|  | | |
|  | |  |
|  | | |
|  | | |
|  | | |
|  | | |
|  |  | |
|  |  | |
|  |  | |

### Routing

There will be different views which are displayed based on the routing (the URL). There are no strict rules of how the current application routing should look like but there are some common set of rules to follow:

1. Use short and clear URL's
   1. Good examples
      * "/user/login"
      * "/category/{categoryName}/products"
   2. Bad examples
      * "/loginPage.html"
      * "showAllProductsByCategoryName/{categoryName}"
2. Follow the HTTP standards
   1. Proper usage of HTTP methods (GET, POST, PUT etc.)
   2. Follow the GET – POST – Redirect pattern

### Models

Here will be described the mandatory information that has to be provided for each model

**User**

* **Username** – **string** which is **required** and **unique**
* **Password** – **string** which is **required** and represents the **hashed** value of the user's **password**
* **Created Products** – **collection** of **products** created by given user

**Product**

* **Name** – **string** which is **required** and **unique**
* **Description** – **string** containing some additional information about the product
* **Price** – **decimal** number which is **non-negative**
* **Creator** – **reference** to the **user** who created the product
* **Image Url** – **string** containing **reference** to an **image** which displays the given product
* **Is Bought** – **bool** flag which is set to true whenever the user buys the product
* **Category** – **reference** to the **category** where the current product is placed in.

**Category**

* **Name** – **string** which is **required** and **unique**
* **Products** – **collection** of all **products** in given category

## Prerequisites

All of the pictures containing any sort of programming code (HTML and CSS too) are taken within **Visual Studio Code**.

Keep in mind that this guide is divided into parts – each part covers a specific (single) lecture from [“ExpressJSFundamentals”](https://softuni.bg/trainings/1642/expressjs-fundamentals-may-2017) course. Thus said the first part (lecture) won’t cover the implementation of all the specified above functionalities and also some of the code may be later on refactored or improved based on contents of the given lecture (example: first we will use in-memory array as database and later on we will use MongoDB for that particular job).

# Part I – Laying the Project Fundamentals

## Setup IDE

For starter configure your IDE or text editor. As mentioned above the following steps will include screenshots from [Visual Studio Code](https://code.visualstudio.com/).

Other popular choices are [Atom](https://atom.io/) and [WebStorm](https://www.jetbrains.com/webstorm/).

Before you continue with the next step make sure everything about your IDE is configured and you are up and ready to go.

## Initial Setup

First let’s create our project

Go to the folder where you want the project to be, then press “*Shift*” + “*Right mouse click*” and use “*Open command window here*”, then type “npm init” and fill the project's data as you see fit.

|  |  |
| --- | --- |
|  |  |

If you are using **Visual Studio Code** then in the same console (after you have inserted all the needed information) type: “code .”. This will open the editor for you.

|  |  |
| --- | --- |
|  |  |

If you are using **different** **IDE** the above steps **might** **be** a little bit **different** so find in **internet** how to create a new **NodeJS/ExpressJS** application for your IDE of choice.

Now let's create the initial project folder structure. It may look like this:

|  |
| --- |
|  |

Download the resources [zip](https://softuni.bg/downloads/svn/js-web/May-2017/Express/02.%20ExpressJS-Fundamentals-Introduction-to-Node-JS/02.%20ExpressJS-Fundamentals-Introduction-to-Node-JS-Lab-Resources.zip) and put the **site.css** and **favicon.ico** in the "***content***" folder like this:

|  |
| --- |
|  |

## Start Server

Go to index.js file and start implementing the server:

|  |  |
| --- | --- |
|  |  |

After you are done – start the application in debug mode (F5 or any similar shortcut), go to your browser at "localhost:{yourPortHere}/":

|  |  |
| --- | --- |
|  |  |

## Display Home Page

We will start implementing the logic for application in the following steps. For now there will be **no logged-in** users or **guest users**: everyone may buy/add/edit/delete products and so on.

First whenever we access our site we want to display the "default" page or so called "home" page. In order to do that we have to:

1. Implement back-end logic of what to be displayed
2. Write our view (the HTML and CSS)
3. Make the server execute the logic we implement in the first step

Now go the "***handlers***" folder and add new "***home.js***" file. In the beginning add the modules that we are going to use:

|  |  |
| --- | --- |
|  |  |

Then let's start with exporting the logic as a function which is accepting both request and response. Then we can parse the requested URL and attach it to the request object:

|  |  |
| --- | --- |
|  |  |

Now we should tell the server when the home handler will handle request (when the requested URL is: "/" and the request method is "GET"):

|  |  |
| --- | --- |
|  |  |

If we could not handle the current request we will notify the server of that by returning true (is request not handled - true).

What is left is to find the HTML5 page read it and send it as a response – here is how it could be done:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

On row 11 it is specified that there will be "***/views/home/index.html***" file which will be sent to client since we don't have it is time to create it.

Go to the "**views**" folder and add new folder "**home**" in it add new file "***index.html****"*:

|  |  |
| --- | --- |
|  |  |

Use the following html:

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>ShopStop</title>  <!--  TODO: Link favicon.  TODO: Link css. -->  </head>  <body>  <header>  <nav class="nav">  <ul>  <li><a href="/">Home Page</a></li>  <li><a href="/product/add">Add Product</a></li>  </ul>  </nav>  </header>  <main>  <h1 class="head-title">Welcome to ShopStop!</h1>  </main>  <footer>  <p>&copy; ShopStop</p>  </footer>  </body>  </html> |

Hour html page is almost ready (later on products will be added). We are almost ready to test if everything mentioned above is about to work or nah.

Go to the "**handler**" folder and add "***index.js***" file:

|  |  |
| --- | --- |
|  |  |

Finally go back to **root** **folder** and in "***index.js***" and replace the old logic with this one:

|  |
| --- |
|  |

\*Note that we did not specified to search in "***./handlers/index***"? It will search for "**index.js**" file by default.

Now **start** the web application:

|  |  |
| --- | --- |
|  |  |

## Serve Static Files

In this step logic about distributing public files will be implemented – in other words how load different files (.css or simply image) from our server.

Let's begin with adding the back-end logic. In "**handler**" folder add new "***static-files.js***". It will behave like a normal handler but instead of returning html it will return file(s). Our public folder will be the "**content**":

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

The getContentType(url) function will check what the content type of the resource should be (e.g – if the url ends with *"***.css**" the content type should be "**text/css**"). For the full list of content types click [here](https://www.sitepoint.com/web-foundations/mime-types-complete-list/).

Go back to "***handlers/index.js***" and add the static file handler:

|  |  |
| --- | --- |
|  |  |

If you have not [referenced](https://www.w3schools.com/tags/tag_link.asp) the "**site.css**" file and the "**favicon.ico**" in "***home/index.html***" go back and do it.

\*Note make sure that the **href** starts with "**/content/…**"

After that start the web application, now the home page should be slightly different:

|  |  |
| --- | --- |
|  |  |

## Implement Database

In order to **add product** and list all products we should implement some sort of database. In the "**config**" folder add new file called: "***database.js***".

In it there will be in-memory array used as a database. There will be three methods that the "database" should give: get all products, add product and find product by name:

|  |  |
| --- | --- |
|  |  |
|  |  |

Now that we have some sort of database let's continue with adding the functionality of uploading a product to our website.

## Product Create Page

New handler should be created in order to handle all the needed logic for addition of products.

Create "***product.js***" inside the "**handlers**" folder. The handler will be responsible for displaying the (html) form **or** parsing the data from it and put new a product in the database:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

Now that the logic about sending a html form is completed the html itself should be created. Go to "**views**" and add new folder "***products***" and add "***add.html***" inside it:

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>ShopStop</title>  <!--  TODO: Link favicon.  TODO: Link css. -->  </head>  <body>  <header>  <nav class="nav">  <ul>  <li><a href="/">Home Page</a></li>  <li><a href="/product/add">Add Product</a></li>  </ul>  </nav>  </header>  <main>  <form class="form center-form" method="post">  <div class="form-group">  <label for="name">Name</label>  <input id="name" name="name" type="text" class="input-field" />  </div>  <div class="form-group">  <label for="description">Description</label>  <textarea id="description" name="description" type="text"   class="input-field"></textarea>  </div>  <div class="form-group">  <label for="price">Price</label>  <input id="price" name="price" type="number" step="0.01"   class="input-field" />  </div>  <div class="form-group">  <label for="image">Image</label>  <input id="image" name="image" type="text" class="input-field" />  </div>  <div class="form-group">  <input class="btn form-btn" type="submit" class="btn" value="Add">  </div>  </form>  </main>  <footer>  <p>&copy; ShopStop</p>  </footer>  </body>  </html> |

One more thing before we test our application – go to "**handlers/index.js**" and add the new handler. The following view should be displayed whenever we try to add new product:

|  |
| --- |
|  |

Now let's go back to the **product** **handler** and implement the POST request – when the data is sent to the server.

If everything went as expected whenever we click the "**Add**" button we should be redirected to home page.

|  |  |
| --- | --- |
|  |  |
|  |  |

## Home Page

In order to validate previous section it would be nice to have a place where all products are displayed. This is why we have to go back to home handler and change things a little.

But before that go the "**home/index.html**" and put a placeholder inside the **<main>** tag:

|  |  |
| --- | --- |
|  |  |

Now let's go the home handler. We should get a reference to our database and from it get all **\*available** products:

|  |
| --- |
|  |

\*Note that we do not filter the products which are already bought – it will be implemented in the next parts.

If you are not fan of replacing strings you can try any lightweight view engine (like [EJS](https://www.npmjs.com/package/ejs)).

Start the application to see if it works as expected:

|  |
| --- |
|  |
|  |

## Search Products by Name

Having products is fun but sometimes you want to filter them by some criteria. This is why we can create a simple form in our home page which will have only one text field (product's name or part of it). After submitting the form all the products containing such text or having same name should be displayed (case-insensitive):

|  |
| --- |
|  |
|  |

#### Hints:

In "**home/index.html**" below the heading add simple form with one input field and one submit button.

Since the form is displayed on the home page the "**home**" handler should process the request. Go to the "**home**" handler and check if there is any filter argument and if there is filter the collection on the output:

|  |  |
| --- | --- |
|  |  |
|  | \*qs comes from querystring module |