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

# Part IV – Advanced Functionality

Enough boilerplate code and dummy dynamic html generating - in this part [ExpressJS](https://expressjs.com/) and [Pug](http://moderndogmagazine.com/sites/default/files/images/uploads/Pug.jpg) are coming.

## Make the Server-Train "Express"

First, what is **ExpressJS**? **ExpressJS** is a framework built on top of NodeJS which **simplifies** and **accelerates** back-end **development**. Basically, it provides a variety of modules and functionalities making programmers focus more on the application rather than on repeatedly writing same pieces of code.

Like any other module we have to **install** **ExpressJS** first:

|  |
| --- |
| npm install --save --save-exact express |

Since this is going to be the core module we have to **make** some general **configurations** and more importantly **migrate** current **code** base to the state where it is capable of working with **Express**.

We should start with **index.js**:

|  |  |
| --- | --- |
|  | **Notice** that we only said which port is going to be used by the server – any other configuration will come in a **separate file** in the following steps. |

Then let's continue with some basic **ExpressJS** configurations – in create new **"config/express.js"**:

|  |  |
| --- | --- |
|  | In this **file** we will only set up the **public folder** (the folder from which files are accessible to anyone) and setup the middleware for **parsing** form data.  **\*Warning: install 'body-parser' first.** |

Now we have to specify this "**config.rootPath**", go to "**config/config.js**":

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

Route mapping will have the following design in "**config/routes.js**":

|  |  |
| --- | --- |
|  | Routing in **Express** is little more intuitive:  1. We specify **HTTP** method  2. We specify **URL**  3. We specify the **function** responsible for **handling** the **request** |

**In order to** make routing **work** with our current logic, **we have to** **refactor** our **handlers** in a way that they **export** the **functions** which handle request.

**WARNING: For the following parts do not rewrite the whole code – just make it look like it is showed in the pictures.**

Go to **home handler** and update it to look like this:

|  |  |
| --- | --- |
|  | **Remove** any **route** and **http** **checks** and **extract** every single if-case to a **function**. |
|  |  |
|  |  |

One more handler left – the **product handler** (here are given only the changed parts):

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

Here **nothing** is **changed** compared to logic which was in the first "if"- case.

However, the next part is completely different:

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

Here we have file upload, but instead of handling the file upload ourselves we will use middleware for that. It is called [multer](https://www.npmjs.com/package/multer). We will set it up **later** on in "**config/routes**".

One last **handler** to migrate is the **category** one:

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

Change "**handlers/index.js**" to:

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

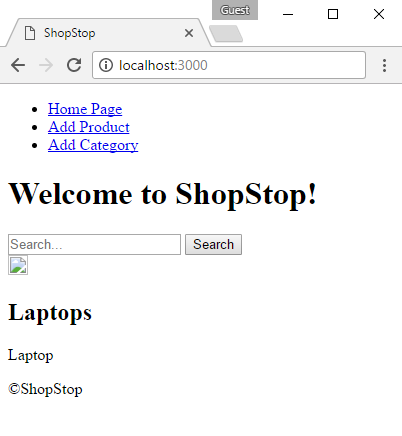
Here is how "**config/routes.js**" should look like:

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

Go to "**index.js**" and **require** them "**config/routes.js**":

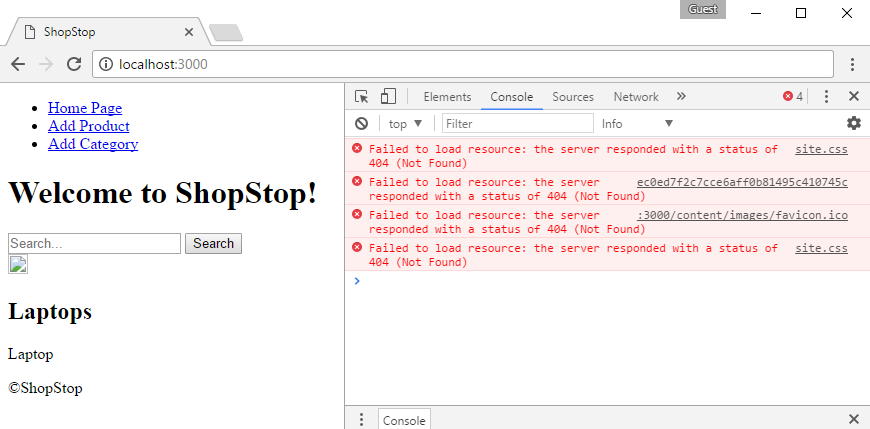
|  |  |
| --- | --- |
|  |  |

Start the application now and if we have followed the steps precisely and installed all needed module something like this should happen:



There is something wrong with our CSS and images – with **F12** go to "**Console**" tab:

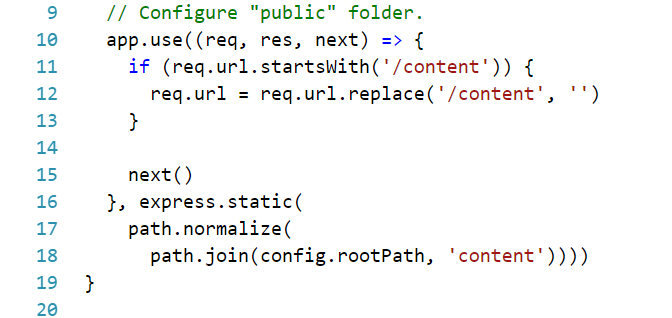
**\*Reminder: make sure you have at least one product with image to see if images are visualized correctly.**



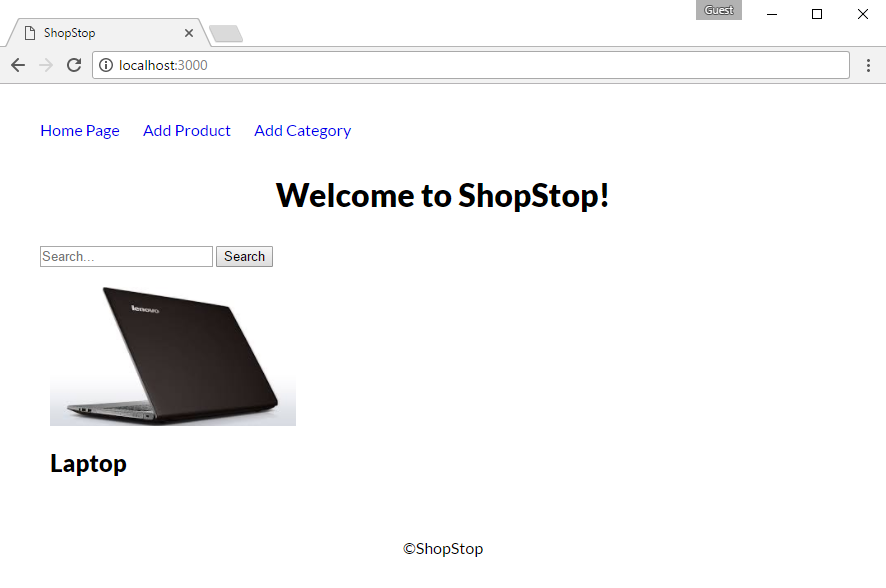
**\*Reminder: make sure you have at least one product with image to see if images are visualized correctly.**

As we can see CSS files and images are not served – our public handler is not working. That's because when we configured in **express.js** the public folder handler – we defined **where the files** are on the server.

In order to get them however we should only specify their path but **without** "**/content**" prefix. We will define our own middleware to handle this case. Go to "**config/express.js**" and **update** the "**express.static**" middleware:



Starting the application again:



## View Engines

View engines are other crucial things when developing with no front-end framework (like Angular or React). View engines help you dynamically create html - based on templates and data coming directly from your back-end. It uses so called **template** **system** where in most of the time you write in **html-based language** but you can include some conditional statements ("**if else**") and repetitions structures ("**for-loops**").

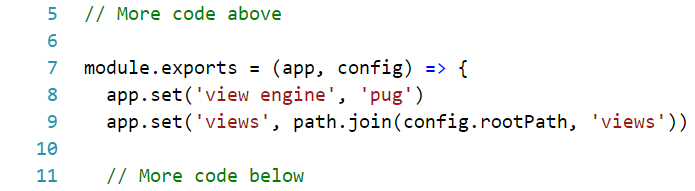
For this lab we will use [Pug](https://pugjs.org/api/getting-started.html) as view engine (this time is the real link), but there are other [options](https://github.com/expressjs/express/wiki?_ga=2.193141869.1793855046.1495192928-256086991.1495190978#template-engines) and you may try them as well.

### Working with Pug

First install pug:

|  |
| --- |
| npm install --save --save-exact pug |

Than go to "**config/express.js**" and add the following:



Now is time to migrate our views to pug, but not simply from html to pug – we are going to use layout (one html file in which we will put dynamically other html pages).

As a starter create "**views/layout.pug**":

|  |
| --- |
| 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  link(rel='icon', type='image/x-icon', href='/content/images/favicon.ico')  link(rel='stylesheet', type='text/css', href='/content/styles/site.css')  body  header  nav.nav  ul  li  a(href='/') Home Page  li  a(href='/product/add') Add Product  li  a(href='/category/add') Add Category  li.delimeter  li.align-right  a(href='/user/register') Register  li.align-right  a(href='/user/login') Login  main  block content  footer  p &copy;ShopStop |

Here we included the following elements: **header** of the page (*navigation bar*) , **main** **content** (*block content* – which is empty) and **footer**. What we will do forward is to **replace** block "**content**" with dynamically created html.

Then **migrate** "**home/index.html**" to "**home/index.pug**":

|  |
| --- |
| extends ../layout.pug  block content  h1.head-title Welcome to ShopStop!  form#search-form  input(type='text', name='query', placeholder='Search...')  input(type='submit', value='Search')  .cards  each product in products  .product-card  img.product-img(src=`${product.image}`)  h2=product.name  p=product.description |

On the first row, we say that we are going to extend the **layout.pug** – this way we enable our current view to be part of another view. By specifying "**block content**" it will look in the **layout.pug** for such block and replace it's html with the one html given in **index.pug**.

Notice the "**each**" **loop** inside the html – we have to **pass** **array of products** and it will know how display every single one – how pleasant.

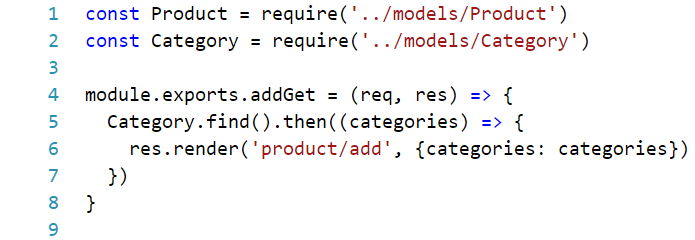
Go to **home handler** and update it to the following:

|  |  |
| --- | --- |
|  | No more boilerplate code: just query to database and rendering the results. |

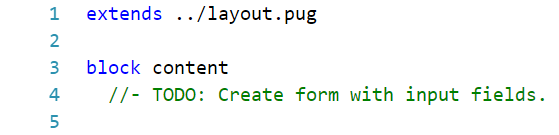
We should also migrate "**product/add.html**" to "**product/add.pug**":

|  |
| --- |
| extends ../layout.pug  block content  form.form.center-form(method='post', enctype='multipart/form-data')  h2 Add Product  .form-group  label(for='name') Name  input#name.input-field(name='name', type='text')  .form-group  label(for='description') Description  textarea#description.input-field(name='description', type='text')  .form-group  label(for='price') Price  input#price.input-field(name='price', type='number', step="0.01" min="0")  .form-group  label(for='image') Image  input#image.input-field(name='image', type='file', accept='.jpg,.jpeg,.png')  .form-group  label(for='description') Group  select.input-field(name='category')  each category in categories  option(value=`${category.\_id}`)=category.name  .form-group  input.form-btn.btn(type='submit', value='Add') |

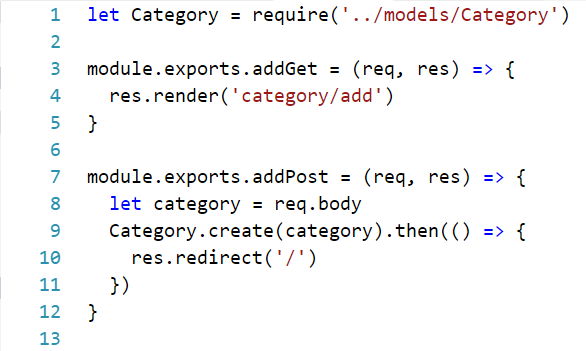
Go to **product** **handler**:



The migration of "**category/add.html**" is left to you:



And don’t forget the **category** **handler**:



Start the application and test if everything was refactored properly.

This is how the folder structure is looking like:

|  |  |
| --- | --- |
|  |  |
|  | Note that the "**products**" folder was renamed to "**product**" in order to match other folders’ signature. |

## Home Page Update

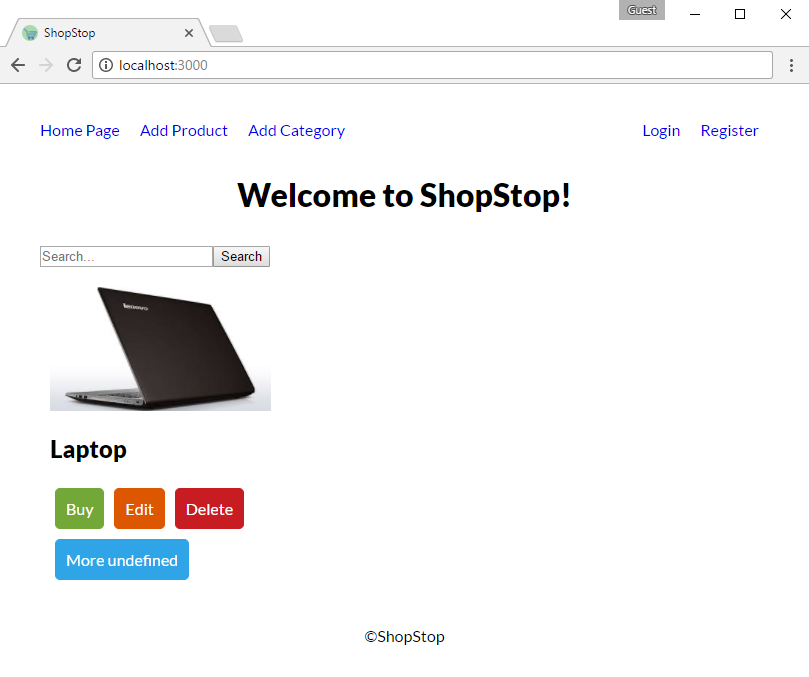
Before we add new functionality go to **"home/index.pug"** and add the following:

|  |
| --- |
| a.btn.btn-small.btn-success(href=`/product/buy/${product.\_id}`)= 'Buy'  a.btn.btn-small.btn-warning(href=`/product/edit/${product.\_id}`)= 'Edit'  a.btn.btn-small.btn-danger(href=`/product/delete/${product.\_id}`)= 'Delete'  a.btn.btn-small.btn-primary(href=`/category/${product.category.name}/products`)  =`More ${product.category.name}` |

These are links for the functionality we are about to add. Keep in mind that these links should be below each product like this:



If everything went well on the **home page** below every product (if any) there should be four new fancy buttons:

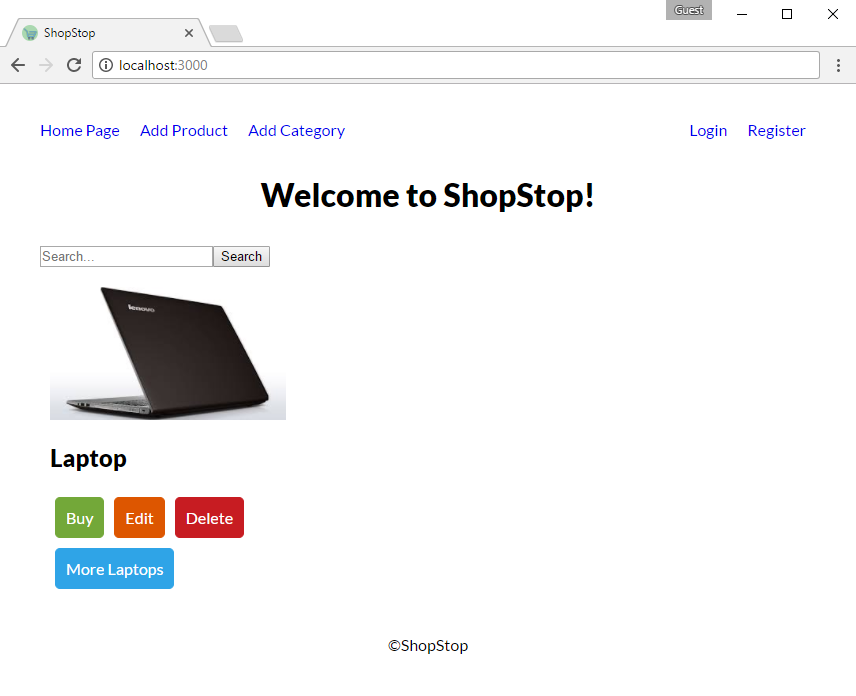


Note that we have "**undefined**" in one of our buttons – so let’s fix it.

The problem is that we want to use product’s category name there, but in order to do that we have to take the name from the database. We are going to use mongoose "[populate](http://mongoosejs.com/docs/populate.html)". Go to **home handler** and modify **index** function to be more like this:



Now, start again and see if it is working properly:

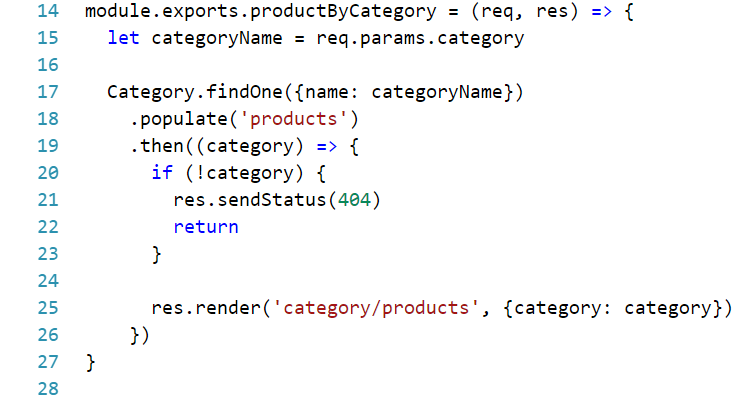


## Show Products by Category

Implementation of showing products by category is next step in developing our application or basically said what will happen when we **click** on "**More Laptops**" button.

At start we should clarify how the above functionality is going to work: first, the **route** will be like this: "/category/:category/products" which means we could get the **category** name **from** the **route**, make **query** to **database** about that specific category and **generate** **view** with all products in given category as response.

So, we should create a new action **category handler**:



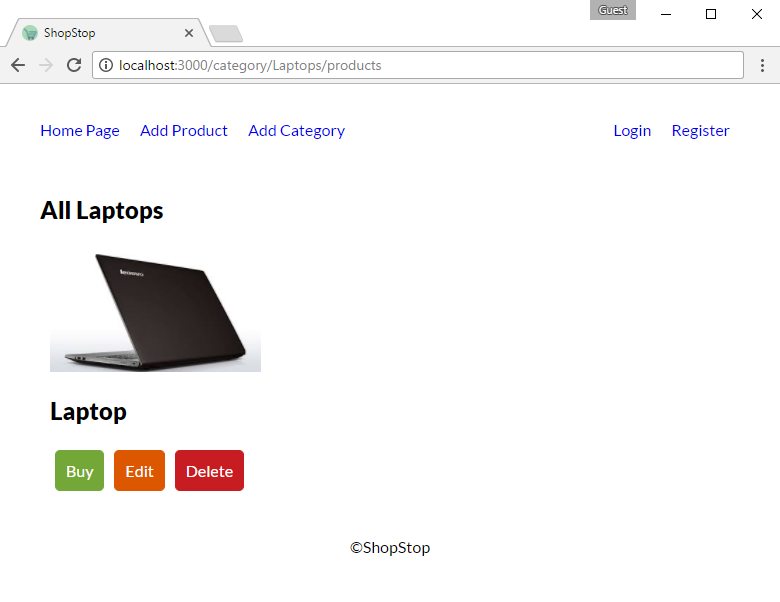
It should **render** the following:

|  |
| --- |
| extends ../layout.pug  block content  h2=`All ${category.name}`  .cards  each product in category.products  .product-card  img.product-img(src=`${product.image}`)  h2=product.name  p=product.description    a.btn.btn-small.btn-success(href=`/product/buy/${product.\_id}`)= 'Buy'  a.btn.btn-small.btn-warning(href=`/product/edit/${product.\_id}`)= 'Edit'  a.btn.btn-small.btn-danger(href=`/product/delete/${product.\_id}`)= 'Delete' |

And here is the **routing** map:



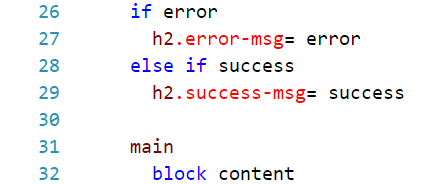
Now start the application and [test](http://imgur.com/gallery/GpWZtCn):



## Message System

In this step we will see how we can send messages to user with simplified **message** **system**.

First go to "**views/layout.pug**" and add the following:



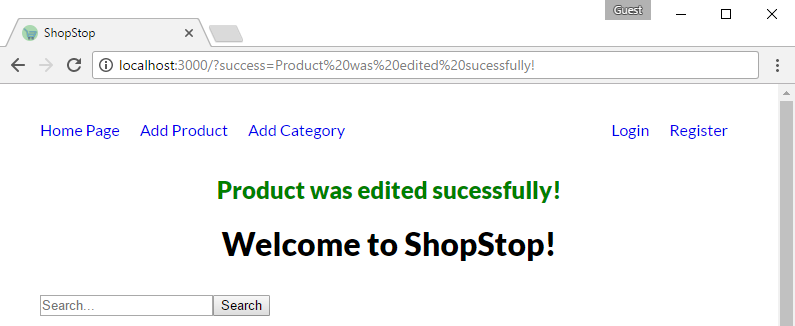
The code above will enable sending "**error**" or "**success**" strings when rendering the view.

When we add the functionality about **editing/deleting/buying** a **product** we will have to make **POST** request which will **redirect** to **home page** with some message. But how the message is going to be passed? The answer is pretty simple – through **query string**: "/?error=This%20is%20error" or "/?success=This%20is%20success"

Go to **home handler** and add the following code:

|  |  |
| --- | --- |
|  | This code will read from the query string for any key-value pair with key: "**error**" or "**success**" and it will take it's value and put it as a message to the layout. |

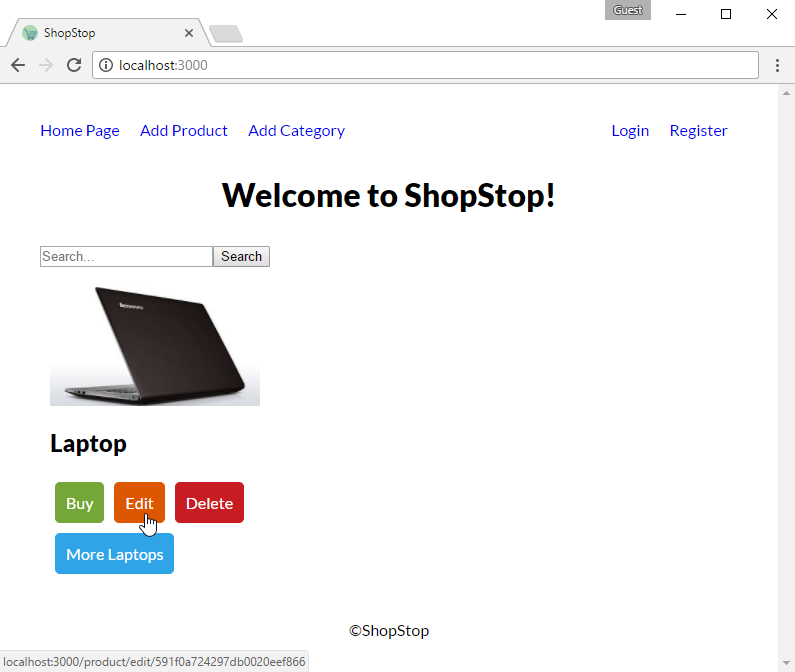
Before moving forward to next part – test if the following message system is working – start the application and use the following url: "/?success=Product%20was%20edited%20sucessfully!". This will lead to:

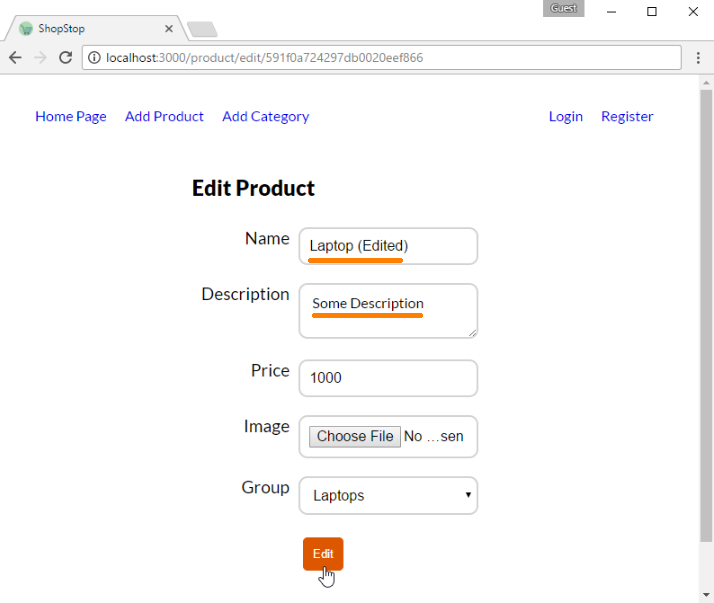


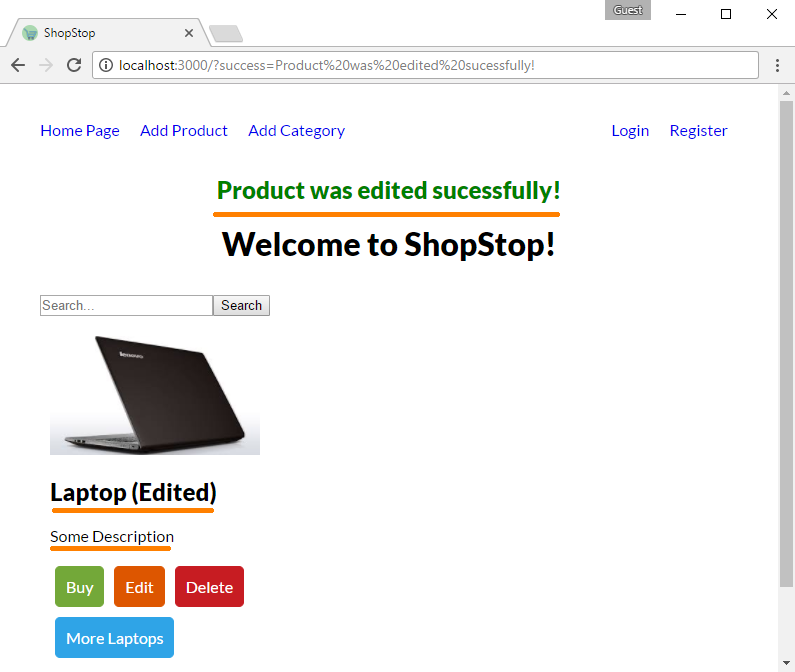
**\*WARNING:** Since the message which is displayed could be edited in the query string – it is not a good practice doing it this way. In the next part of lab (**Part V – User Authentication**) we will see **sessions** and **cookies**. You should **use** **one** of those in order to **improve** current **messaging** **system**.

## Edit Product

Now let’s add the functionality of editing product. Here is what it should look like:







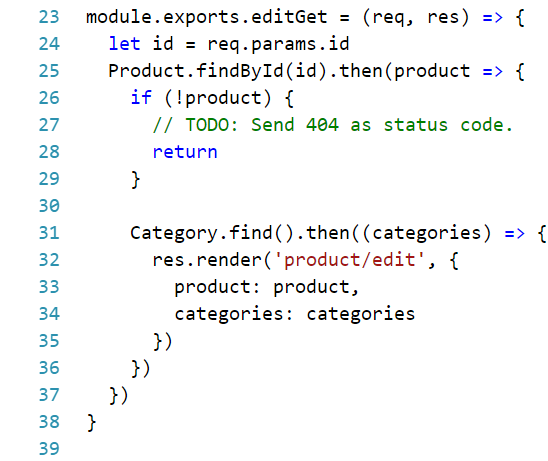
Before diving into writing the logic behind editing product we should consider two key things:

1. Product's picture should be **edited** if **only** a new one is **uploaded** (e.g. if we edit only title there is no need of uploading same picture)
2. With changing product's category we should **remove** **the product's reference** in **old category** and **add** it to the **new** one.

With keeping this arguments in mind we are ready to create our **product/edit** **view**:

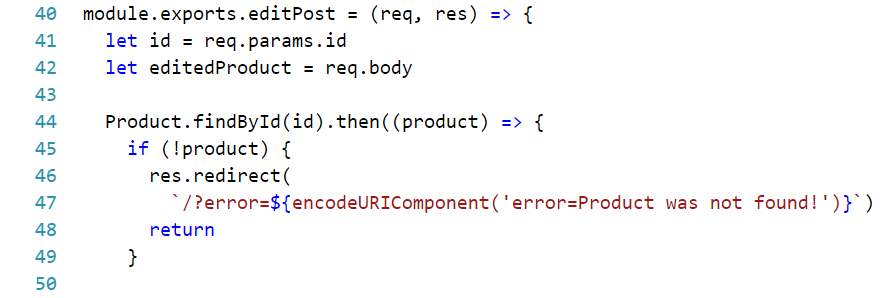
|  |
| --- |
| extends ../layout.pug  block content  form.form.center-form(method='post', enctype='multipart/form-data')  h2 Edit Product  .form-group  label(for='name') Name  input#name.input-field(name='name', type='text', value=product.name )  .form-group  label(for='description') Description  textarea#description.input-field(name='description', type='text')= product.description  .form-group  label(for='price') Price  input#price.input-field(name='price', type='number', step="0.01" min="0" value=product.price)  .form-group  label(for='image') Image  input#image.input-field(name='image', type='file', accept='.jpg,.jpeg,.png')  .form-group  label(for='description') Group  select.input-field(name='category')  each category in categories  if (category.\_id.equals(product.category))  option(value=`${category.\_id}` selected)=category.name  else  option(value=`${category.\_id}`)=category.name  .form-group  input.form-btn.btn.btn-warning(type='submit', value='Edit') |

Now in the **product** **handler** and add action for visualizing the edit form:



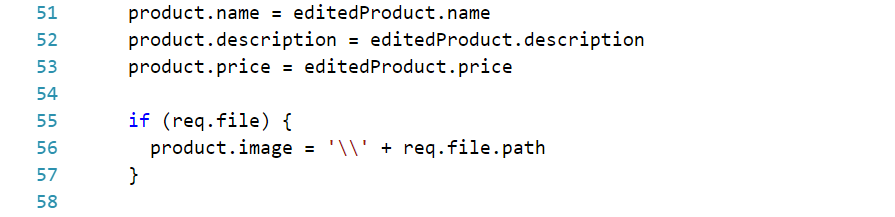
\*Hint: [here](https://stackoverflow.com/a/42370052/5591237) is how you could send status code

This was the easy part, now we have to implement the real logic behind editing a product:



Note the usage of [encodeURIComponent](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent) – it will simply **escape** any **invalid** url **characters** (like "/" or " ")

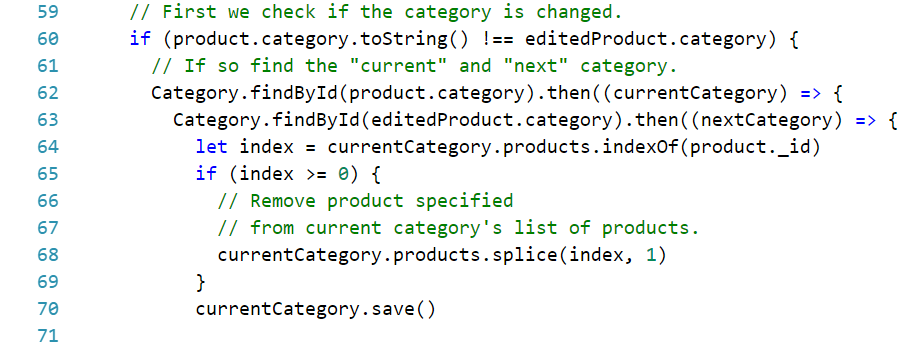
Then we continue with editing the product object:

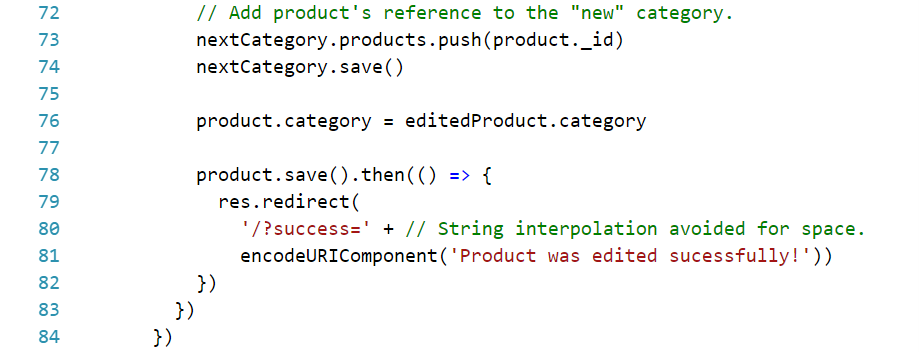


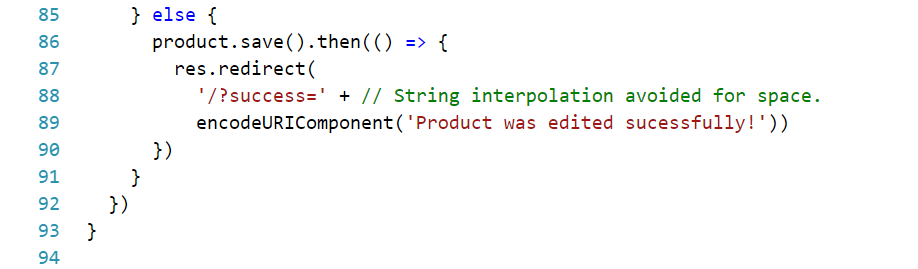
We just **re-assign** **values** passed **from** the **form** in order to **update** the **current** **product** object.

The next part is little messy though. The reason is because we have to deal with the relation between product and category.

Whenever we **change** some **product's** **category** we should go to that **old category** and **remove product's reference** from it and last but least in the **new category** we should **add product's reference**. So here is a suggestion of how it could be done:







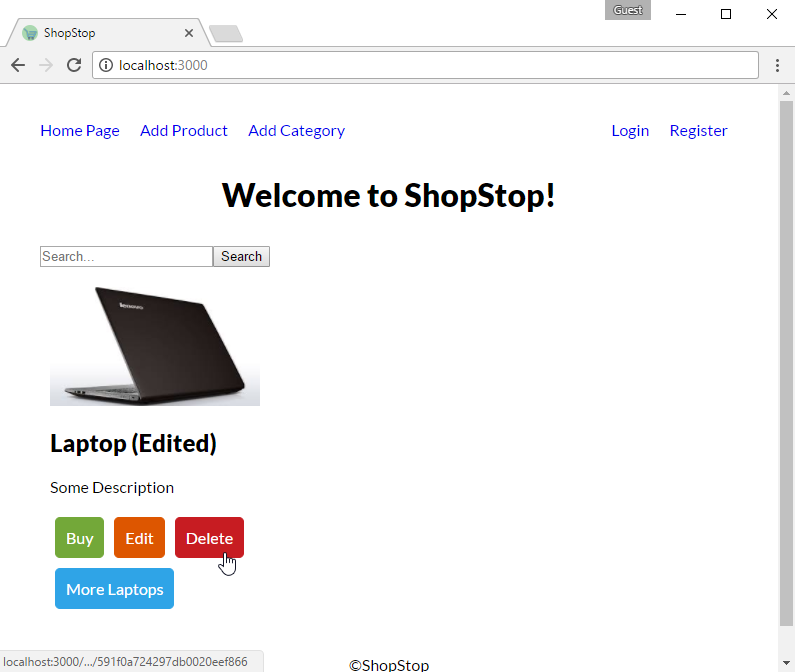
One last thing before we move on – **routing**:

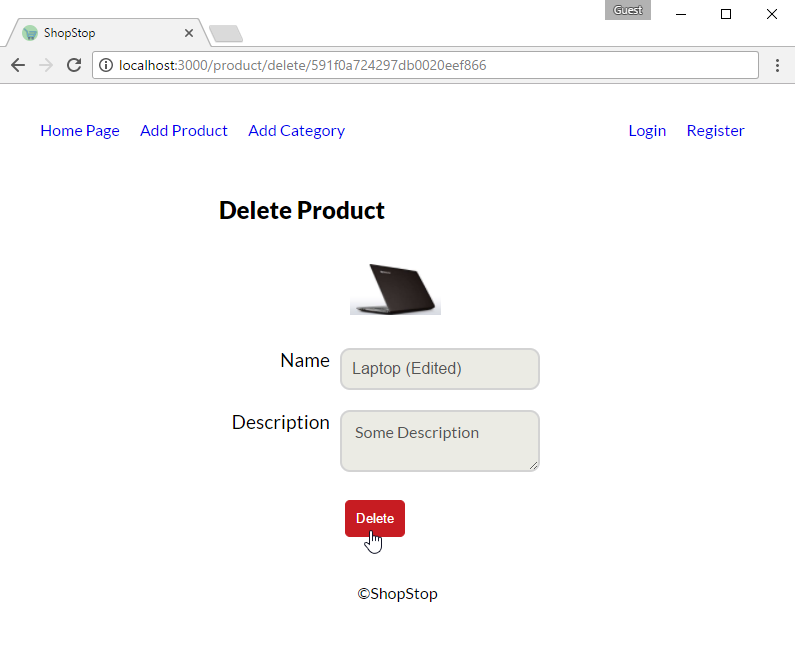


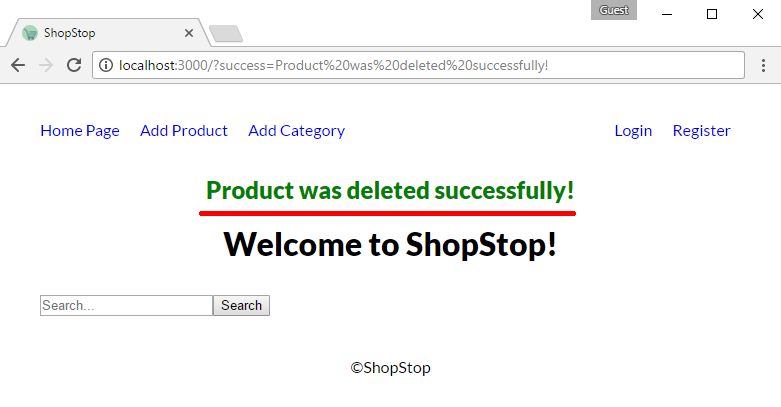
## Delete Product

Moving on next is the deletion of a product. Keep in mind that when you **delete** a **product** you should **delete** it's **reference** **in** the **category** of which is part of.

Here is how it should look like:







Here is the html:

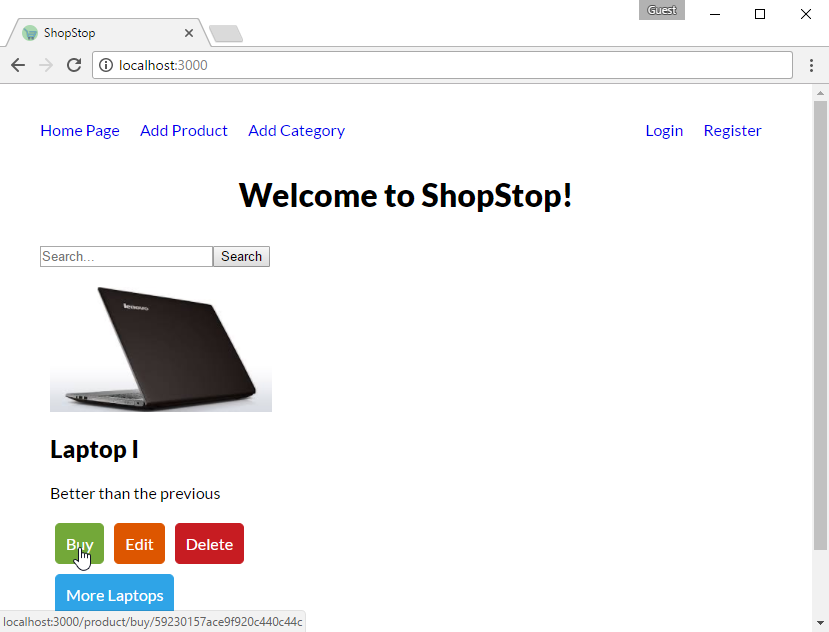
|  |
| --- |
| extends ../layout.pug  block content  form.form.center-form(method='post', enctype='multipart/form-data')  h2 Delete Product  .form-group  label  .product-card  img.product-img(src=`${product.image}`)  .form-group  label(for='name') Name  input#name.input-field(name='name', type='text', value=product.name disabled )  .form-group  label(for='description') Description  textarea#description.input-field(name='description', type='text' disabled )= product.description  .form-group  input.form-btn.btn.btn-danger(type='submit', value='Delete') |

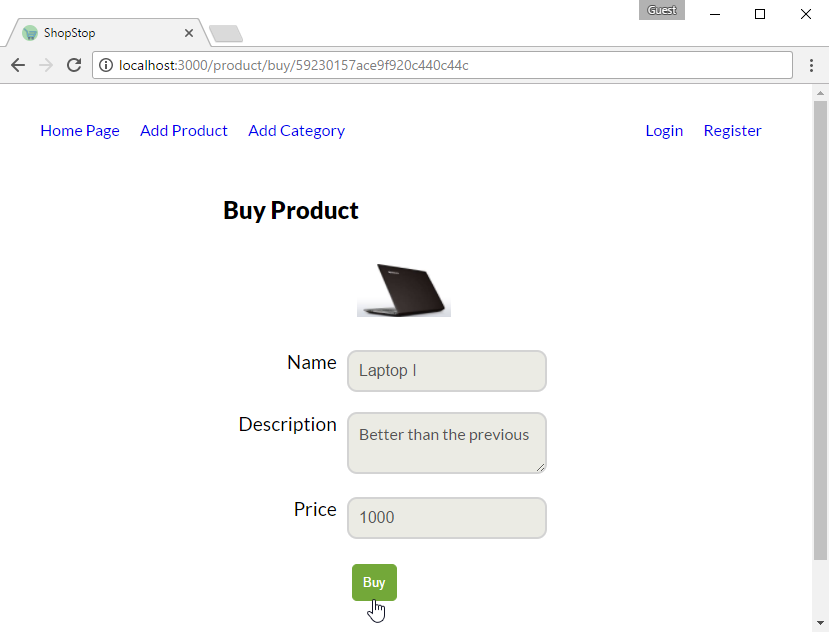
Follow the guidelines from the previous part and you should be able to implement the removal of product on your own.

**\*Hint: when deleting a product, delete product's picture as well.**

## Buy Products

For this part generate only GET request handler displaying the following view:





Use the following html:

|  |
| --- |
| extends ../layout.pug  block content  form.form.center-form(method='post')  h2 Buy Product  .form-group  label  .product-card  img.product-img(src=`${product.image}`)  .form-group  label(for='name') Name  input#name.input-field(name='name', type='text', value=product.name disabled )  .form-group  label(for='description') Description  textarea#description.input-field(name='description', type='text' disabled )= product.description  .form-group  label(for='price') Price  input#price.input-field(name='price', type='number', value=product.price disabled )  .form-group  input.form-btn.btn-success.btn(type='submit', value='Buy') |

**\*Note** that only the **GET** action should be implemented. When we have users we will come back for this one.

**In the next part** we will implement the functionality behind **buying a product** and we will cover up **user authentication**.