# 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
* **Part V – User Authentication (current)**

# Part V – User Authentication

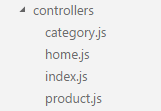
This part will cover user authentication and will provide knowledge needed for creating an application which provides user register/login/logout. This is one of the major functionalities we use in everyday apps.

|  |
| --- |
| mongod --dbpath {yourCustomPath} |

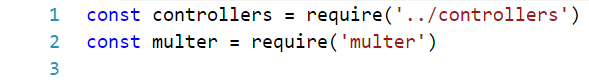
## MVC in a Nutshell

Let's migrate our application project structure to **MVC** – we just have to rename all of our handlers to controllers. Why? – the reason is we used **MVC** model pattern long ago before this part. The concept was slowly integrating within our application – from creating the handlers to distributing the views into different folders.

Rename "handlers" folder to "controllers":



Go to **routes.js**, fix require call and rename "**handlers**" variable to "**controllers**":



## Implementing the Concepts of Authentication

Before we start we should add **utilities** folder. In it we can put all non-core logic.

Add new **encryption.js** file:

|  |
| --- |
| const crypto = require('crypto')  module.exports = {  generateSalt: () => {  return crypto.randomBytes(128).toString('base64')  },  generateHashedPassword: (salt, pwd) => {  return crypto.createHmac('sha256', salt).update(pwd).digest('hex')  }  } |

In the next section we will look into User model.

### User Model

Inside our "**models"** folder create new **User.js**:

|  |
| --- |
| const mongoose = require('mongoose')  const encryption = require('../utilities/encryption')  const propertyIsRequired = '{0} is required.'  let userSchema = mongoose.Schema({  username: {  type: mongoose.Schema.Types.String,  required: propertyIsRequired.replace('{0}', 'Username'),  unique: true  },  password: {  type: mongoose.Schema.Types.String,  required: propertyIsRequired.replace('{0}', 'Password')  },  salt: {  type: mongoose.Schema.Types.String,  required: true  },  firstName: {  type: mongoose.Schema.Types.String,  required: propertyIsRequired.replace('{0}', 'First name')  },  lastName: {  type: mongoose.Schema.Types.String,  required: propertyIsRequired.replace('{0}', 'Last name')  },  age: {  type: mongoose.Schema.Types.Number,  min: [0, 'Age must be between 0 and 120'],  max: [120, 'Age must be between 0 and 120']  },  gender: {  type: mongoose.Schema.Types.String,  enum: {  values: ['Male', 'Female'],  message: 'Gender should be either "Male" or "Female".'  }  },  roles: [{ type: mongoose.Schema.Types.String }],  boughtProducts: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Product' }],  createdProducts: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Product' }],  createdCategories: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Category' }]  }) |

Our **User** model is validation rich – it has validation message for almost every property. Check the official [documentation](http://mongoosejs.com/docs/validation.html) about validation.

The reason why we want so much detailed validation is because we will use it later on when **registering** user.

Below is defined method for authenticating user:

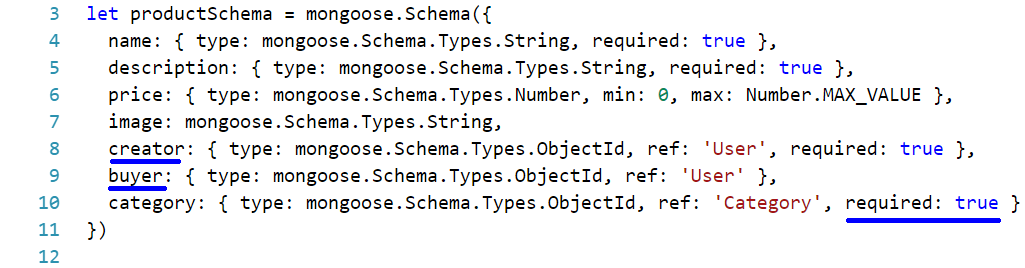
|  |
| --- |
| userSchema.method({  authenticate: function (password) {  let hashedPassword = encryption.generateHashedPassword(this.salt, password)  if (hashedPassword === this.password) {  return true  }  return false  }  })  const User = mongoose.model('User', userSchema)  module.exports = User |

Let's implement the logic about seeding an admin user:

|  |
| --- |
| module.exports.seedAdminUser = () => {  User.find({username: 'admin'}).then(users => {  if (users.length === 0) {  let salt = encryption.generateSalt()  let hashedPass = encryption.generatedHashedPassword(salt, 'Admin12')  User.create({  username: 'admin',  firstName: 'Chuck',  lastName: 'Test',  salt: salt,  password: hashedPass,  age: 33,  gender: 'Male',  roles: ['Admin']  })  }  })  } |

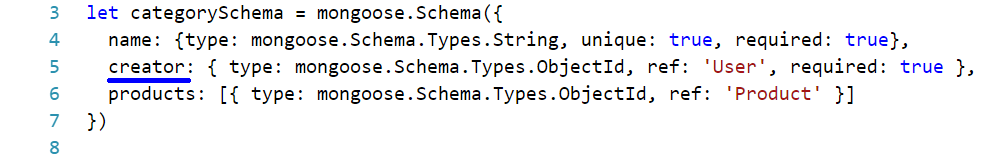
We have defined User relations: **one** **user** may have **many** **uploaded** **products** and **one user** may have **many** **bought products**.

We have to go back to **Product** model and the following properties in the schema:

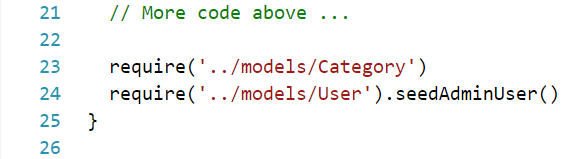


Notice that isBought property is replaced with buyer property.

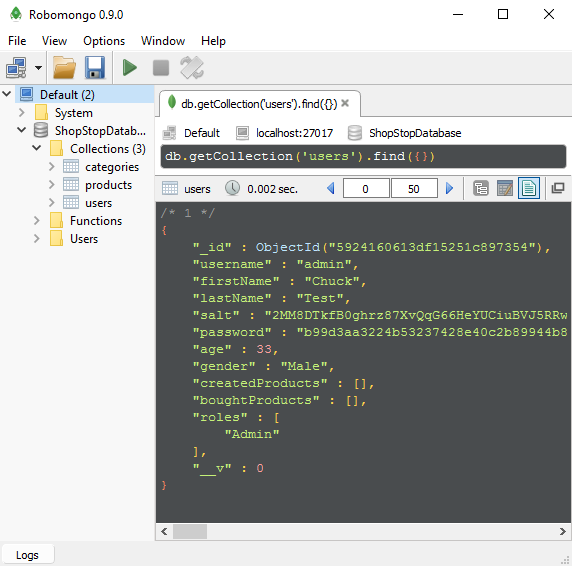
We have to add reference in the **Category** **model**:



Now, let's register **User** schema in our application. Go to **dabase.config.js**:



Start the application and see if there is seeded admin user:



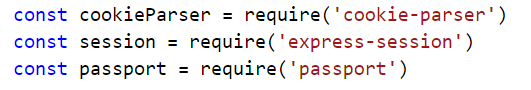
### Authentication Middlewares

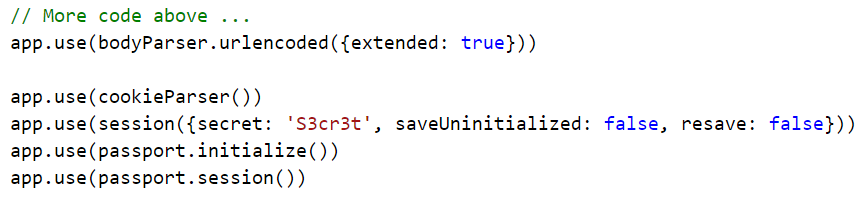
So far, so good but this is not the end. We are going to use third-party **modules** for user authentication and authorization.

Let's begin by **downloading** them:

|  |
| --- |
| npm install --save --save-exact express-session cookie-parser passport passport-local |

Now, to **config/express.js** some configurations should be made:



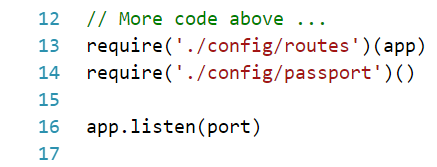


With the above code all the modules are set as **middlewares** which means they will somehow change our request or response (e.g. passport module will **attach** **user** to the **request** every time we have **logged** one).

We have defined our **middlewares** and our model – now we will have to create the link between these two. In other words – how the authorization modules to recognize users and more importantly how to authorize them. In order this to happen create **passport.js** file in our **config** folder:

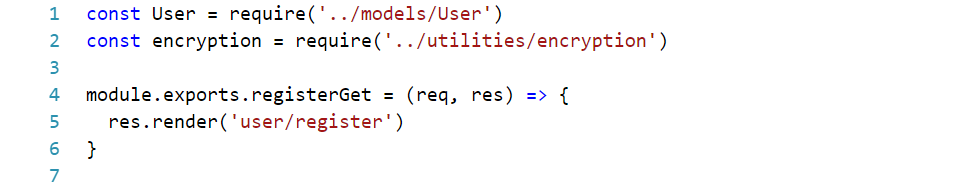
|  |
| --- |
| const passport = require('passport')  const LocalPassport = require('passport-local')  const User = require('mongoose').model('User')  module.exports = () => {  passport.use(new LocalPassport((username, password, done) => {  User.findOne({ username: username }).then(user => {  if (!user) { return done(null, false) }  if (!user.authenticate(password)) { return done(null, false) }  return done(null, user)  })  }))  passport.serializeUser((user, done) => {  if (user) { return done(null, user.\_id) }  })  passport.deserializeUser((id, done) => {  User.findById(id).then(user => {  if (!user) { return done(null, false) }  return done(null, user)  })  })  } |

One more thing go index.js and require the module we have just created:



## Register User

Moving on, we should be able to register users. New controller should be added – the **user controller**:

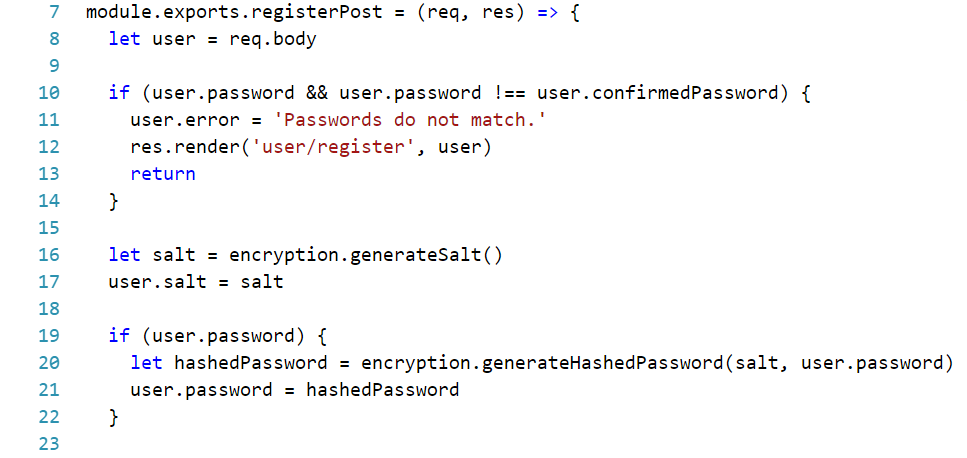


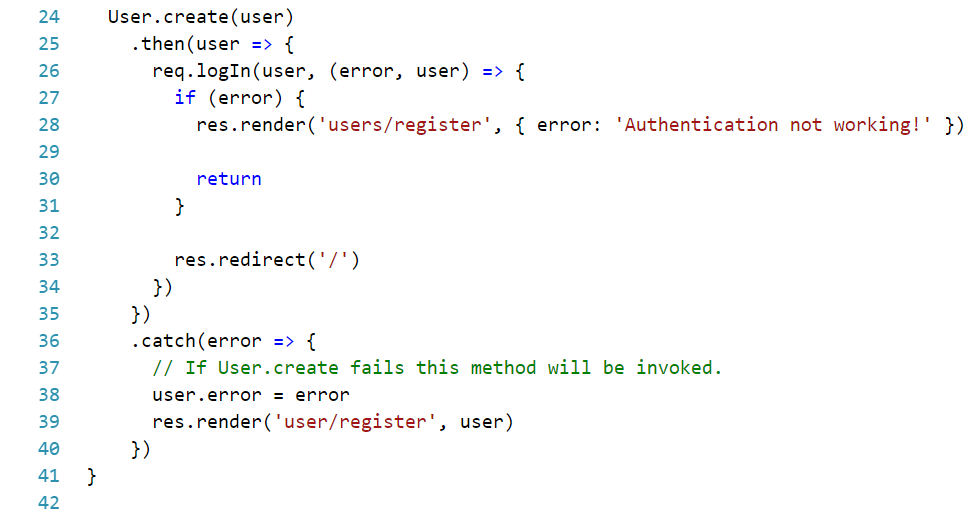
We should render the following view:

|  |
| --- |
| extends ../layout.pug  block content  form.form.center-form(method='post')  h2 Register User  .form-group  label(for='username') Username  input#username.input-field(name='username', type='text' value=username)  .form-group  label(for='password') Password  input#password.input-field(name='password', type='password')  .form-group  label(for='confirmedPassword') Confirm Password  input#confirmedPassword.input-field(name='confirmedPassword', type='password')  .form-group  label(for='firstName') First Name  input#firstName.input-field(name='firstName', type='text' value=firstName)  .form-group  label(for='lastName') Last Name  input#lastName.input-field(name='lastName', type='text' value=lastName)  .form-group  label(for='age') Age  input#age.input-field(name='age', type='number' value=age)  .form-group  label(for='gender') Gender  select#gender.input-field(name='gender')  option Male  option Female  .form-group  input.form-btn.btn(type='submit', value='Register') |

Note that the "**username**" **input field** has "**value"** **attribute** set to "**username**". This will allow us to pass some username to that field (e.g. user enters invalid password we re-render the same view but we do not delete all the data he has previously typed).

Now that we can display the form the next step is to parse the data from it and create new user. Here is what we should add to **user controller**:

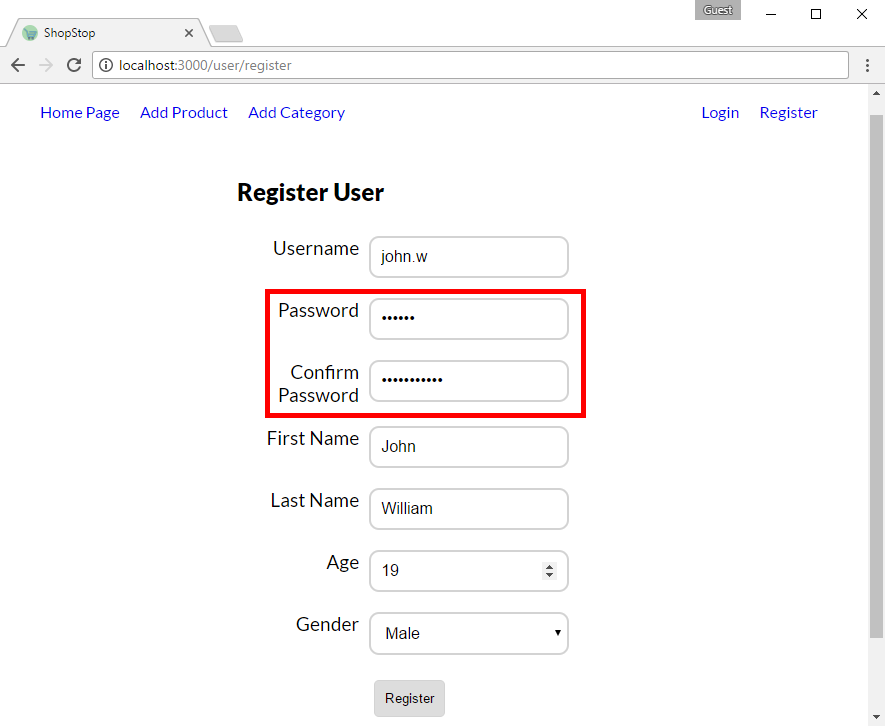


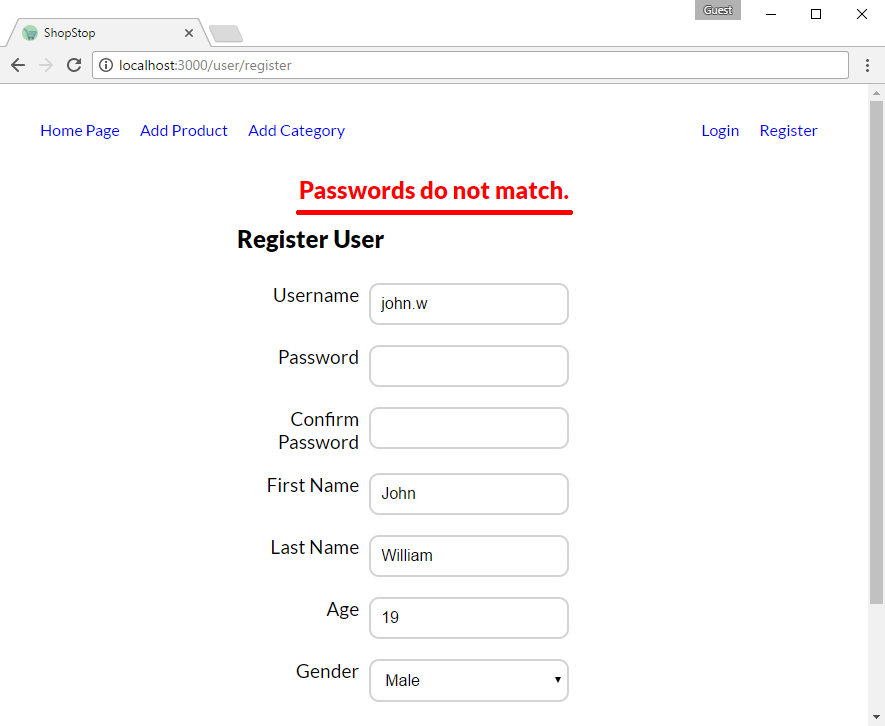


We have previously worked with promises – but this time we will define "**catch**" handler which will be invoked if our promise **throws** **exception**. This is helpful in current case because if the user profile violates any of the rules specified in the User schema – it will throw exception with some **message** (and we **specified** many of those messages). We can use those messages to display them to the user.

Last but not least **routing** should be **configured**. This step is left to you.

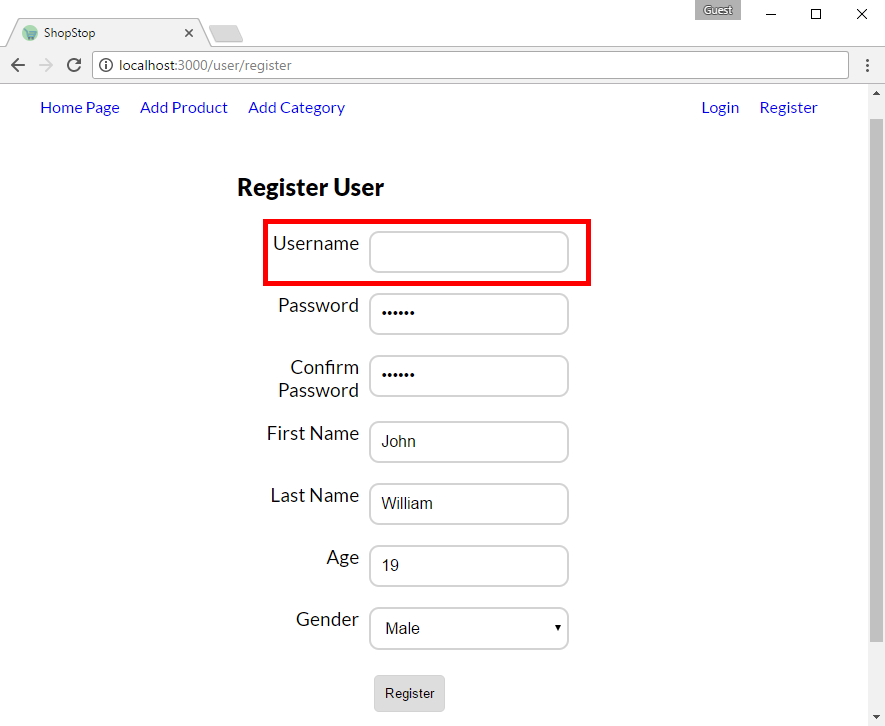
Let’s test what happens if passwords are not matching:

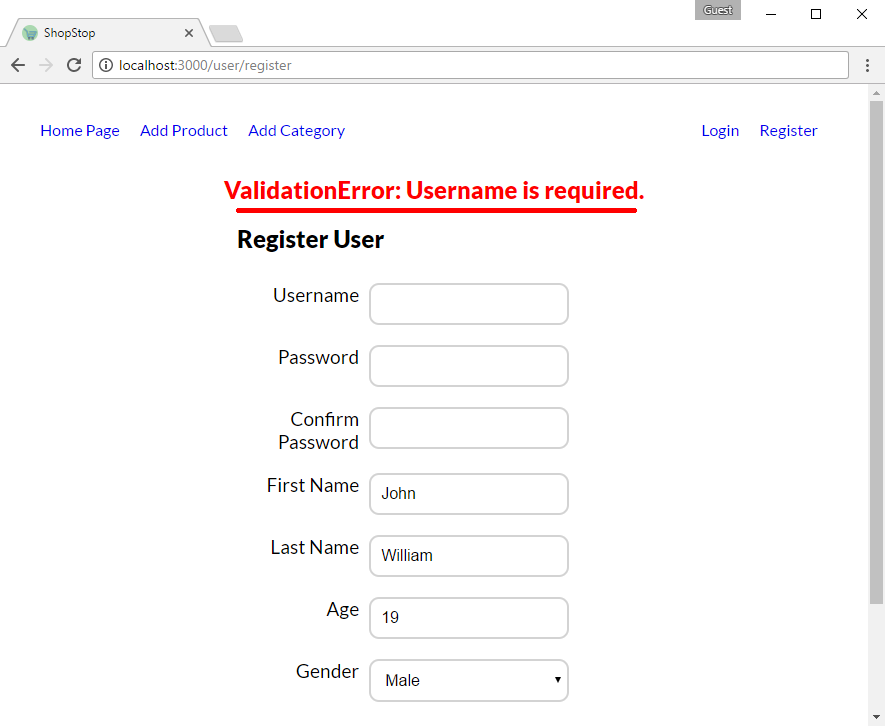




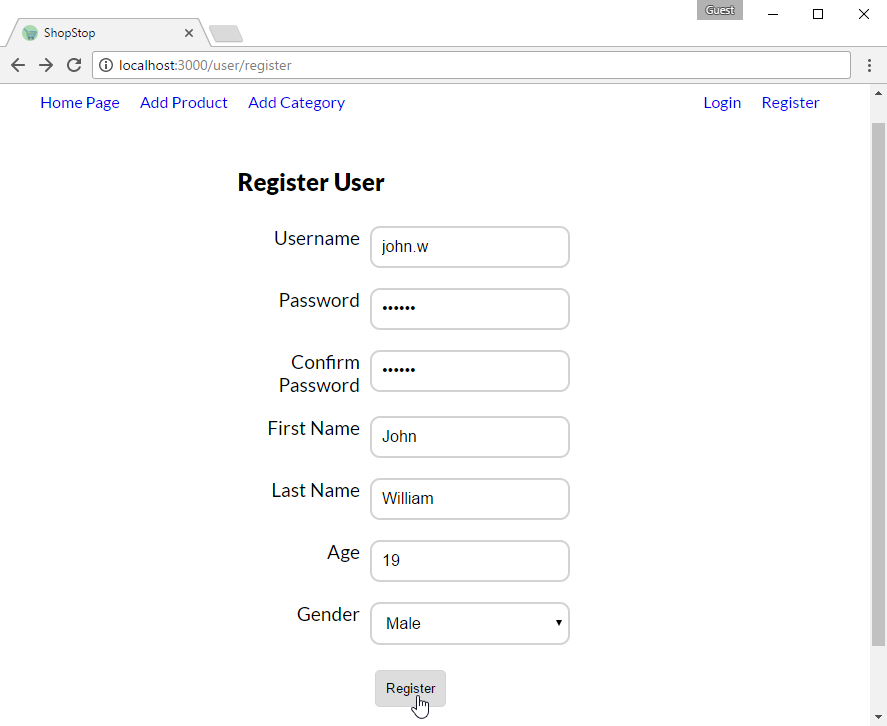
**\*Note that the red line under error message is not part of the html.**

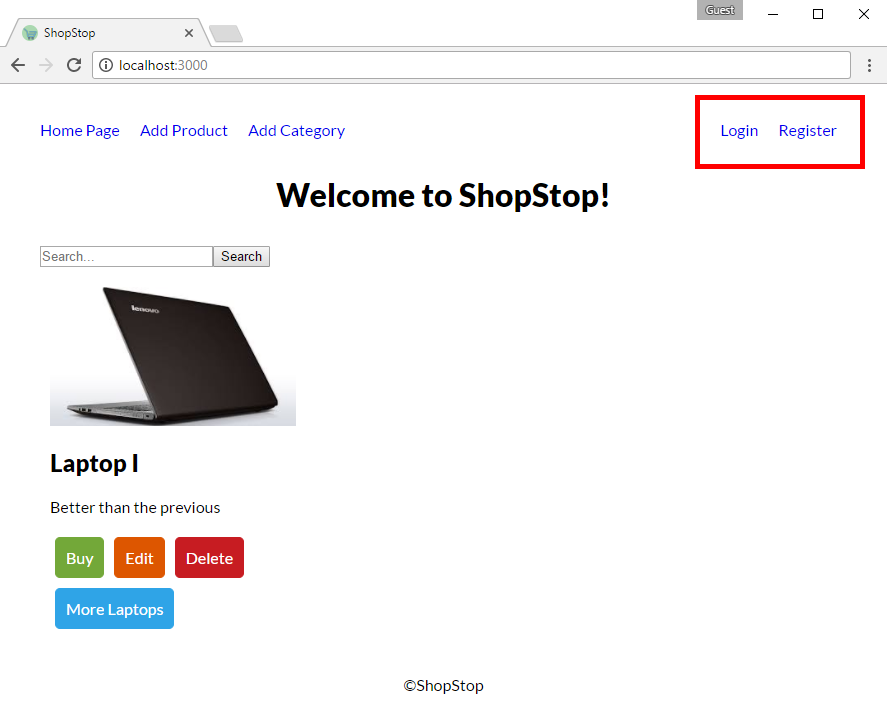
Let’s see what happens if we do not put username:



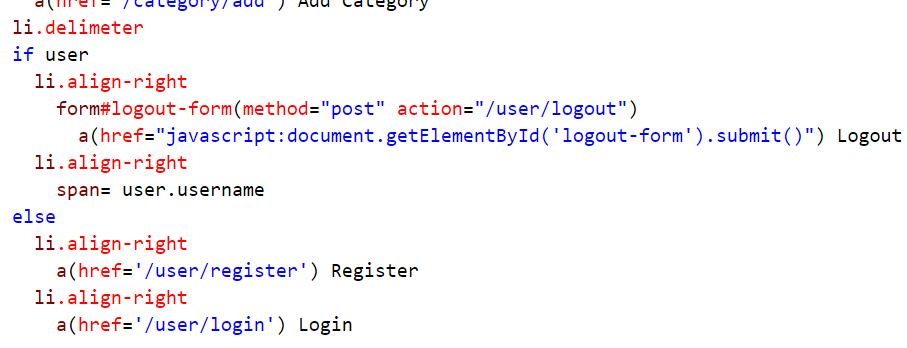


Let's register normal user:

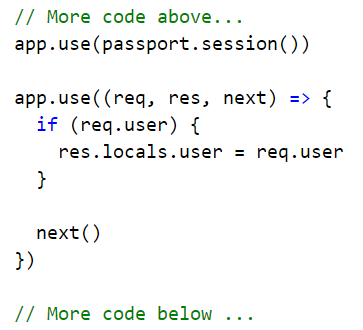




We were redirected but as you can see the navigation bar did not change - let's fix it. Go to "**layout.pug**" and add this in the end of the navigation bar:

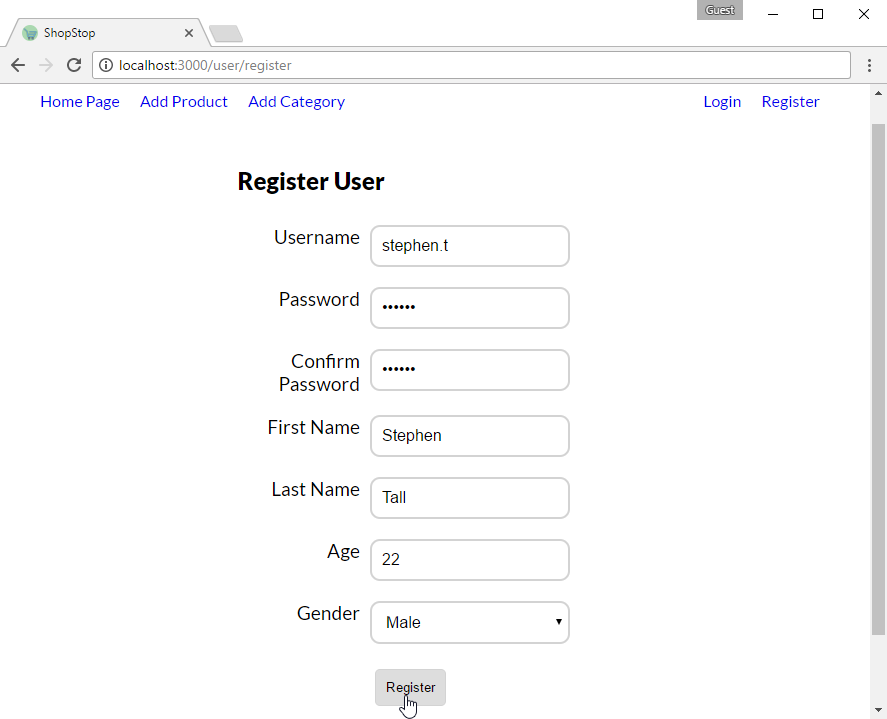


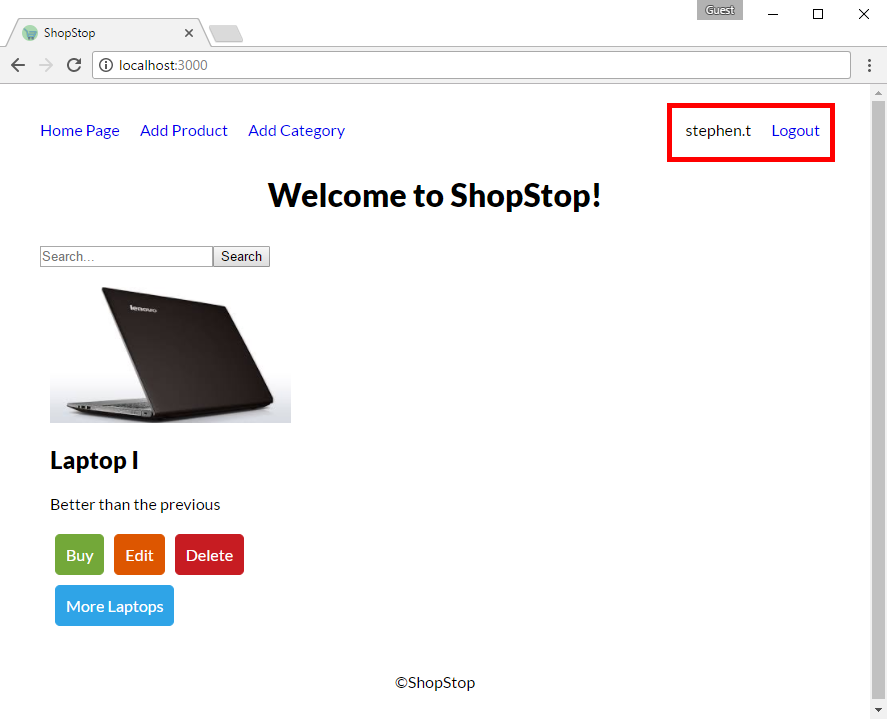
Our view render user's username if there is a currently logged-in user. That's awesome but are we going to pass it every time we render views? Hell, no. We are going to use some pure magic ([res.locals](https://expressjs.com/en/api.html#res.locals) + middlewares). Go to **express.js** config and add the following:



Since we use **passport** (**module**) for authorizing users we are aware that when there is logged-in on he/she will be attached to the request through **req.user** ([source](http://passportjs.org/docs#authenticate)).

Now register again some user and see if everything is going according to the plan:





We can see that our template is successfully working! In next part we will see how to login.

## Login User

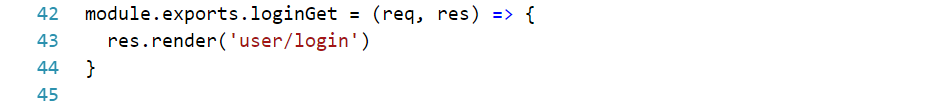
In current implementation of the application the only way to login is through registration – in this step we will implement how log in already registered users.

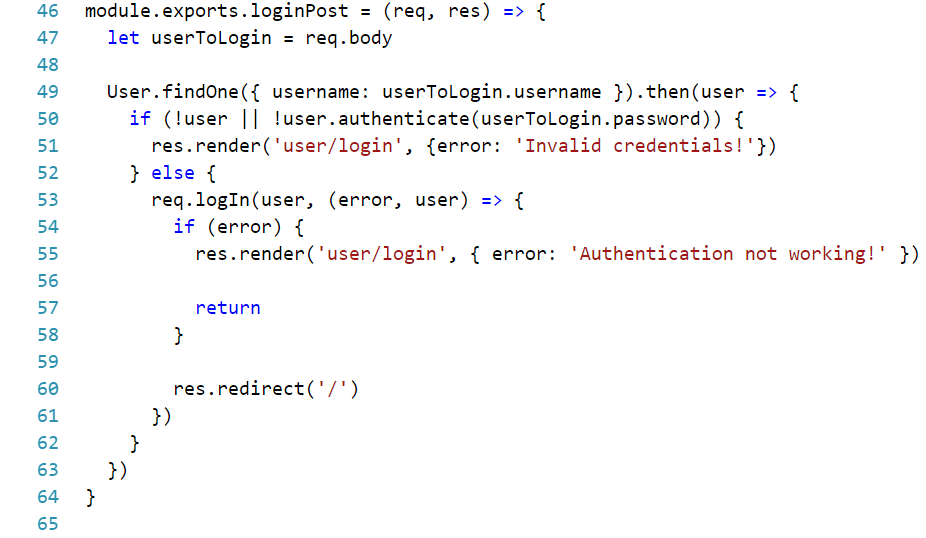
Like many of the steps we took already this one is not different. Why? When we add new **core** functionality we simply did this: 1) create view (**html and css**), 2) implement handler (logic for **displaying** **form** and **parsing** **form** data) and 3) **routing**.

Let's start with our html:

|  |
| --- |
| extends ../layout.pug  block content  form.form.center-form(method='post')  h2 Login  .form-group  label(for='username') Username  input#username.input-field(name='username', type='text')  .form-group  label(for='password') Password  input#password.input-field(name='password', type='password')    .form-group  input.form-btn.btn(type='submit', value='Login') |

Again, we should add some logic to our user handler:

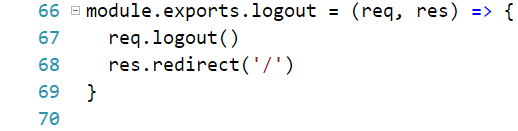




Lastly you should configure **routing** – again is left to you. 😊

## Logout User

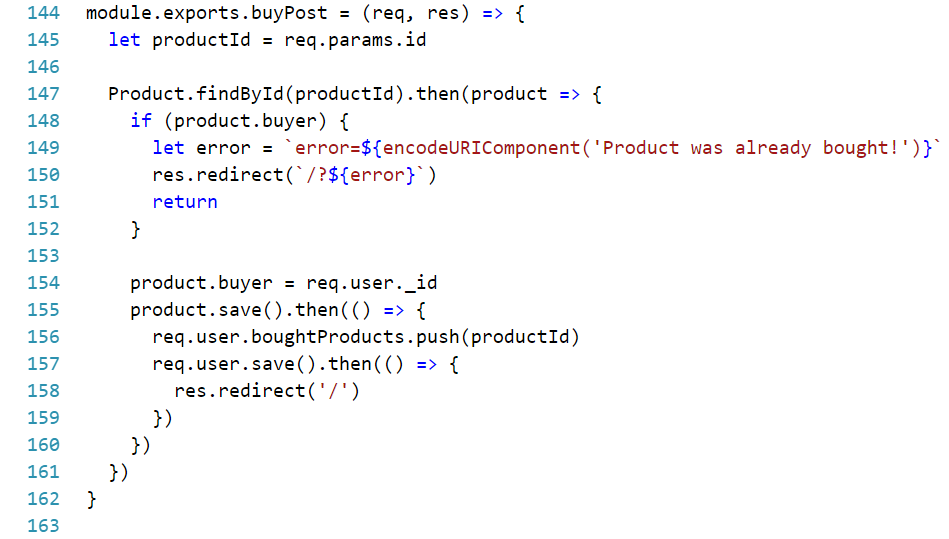
One last thing until we are ready with whole user functionality – implementing logout. [PassportJS](http://passportjs.org/docs) made it really easy to logout in our application – all you have to do is to get the request object and call: "logout()". Logout of user should be done with **POST** **request** and if successful it should redirect to home page – something like this:



**\*Hint: Use "/user/logout" as route URL.**

## Buy Products

In the last part (Part IV – Advanced Functionality) we included the view for buying a product. Now that we have users ready we can finish that one up. Create new function in **product** **handler**:



It will enable us to buy a particular product.

Make sure to configure routing and move on the next step – do not test anything out yet.

## Protect and Serve

Let's implement user authentication and admin authorization.

In **config** folder create **auth.js**:

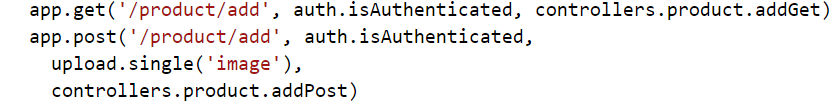
|  |
| --- |
| module.exports = {  isAuthenticated: (req, res, next) => {  if (req.isAuthenticated()) {  next()  } else {  // If not authenticated - login.  res.redirect('/user/login')  }  },  isInRole: (role) => {  return (req, res, next) => {  if (req.user && req.user.roles.indexOf(role) > -1) {  next()  } else {  // If not authorized - login with proper account.  res.redirect('/user/login')  }  }  }  } |

We wrote two **middlewares** which will simply authorize our routes. Why? We want only logged in users to create/buy products and only admins to add categories. These **middlewares** will help us with that:

First require the module



Then use it in routes like that:



Or

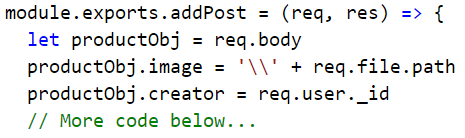


Make sure all other actions (buy/edit/delete of a product and logout) to **require** logged-in user.

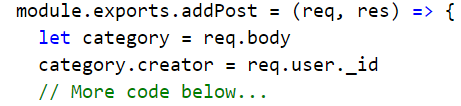
## Going Backward

We are almost ready to finish our application. However, there same major issues we should clean up before reaching final point. Because we added the users' functionality last there are some things to patch up:

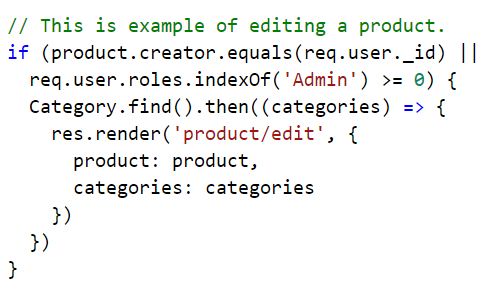
1. When **adding** new **product** we have to make the **currently** logged-in **user** it's **creator** (we should specify that the **product** has a **creator** and that the **user** has new **created** **product**):



1. **Same** goes when **adding** new **category**:

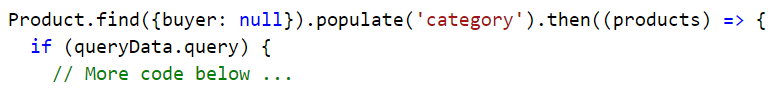


1. We have to check when we **edit**/**delete** product if the current user is creator of the product or Admin:



**\*Warning: Make same check when deleting a product (both in GET and POST requests)**

1. When product has **buyer** it should be not listed in home page/edited/deleted:



**\*Note: make same check when editing/deleting specific product.**

**One last thing – delete all categories and products – they have outdated schemas/properties. Create new ones and test the above logic above.**

## Going Forward

Well done – you have completed the lab! 😊

Here are some useful libraries/frameworks that you might use in future:

**CSS:**

[LittleSnippets](http://littlesnippets.net/) **- CSS snippets**

[Bootstrap](http://getbootstrap.com/) **– framework for front-end development**

[Bootswatch](https://bootswatch.com/) **– templates for bootstrap**

[Bootsnipp](https://bootsnipp.com/) **– small code snippets (login forms etc.) with some HTML and CSS**

**JS**:

[JQuery](https://jquery.com/) – **making asynchronous calls and DOM manipulation**

**Design**:

[Dribble](https://dribbble.com/) **– custom design ideas**

**More:**

[WebAppIdeas](https://trello.com/b/j5HTvKjl/web-app-ideas) – **ideas for web projects**

[FontAwesome](http://fontawesome.io/icons/) **– cool icons for your project**

[SweetAlert2](https://limonte.github.io/sweetalert2/) **– alerter for warnings and messages**

[NotifyJS](https://notifyjs.com/) **– simpler alerter**

[CSSTemplates](http://www.free-css.com/free-css-templates) **– free CSS templates**