# 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 (current)**
* Part IV – Advanced Functionality
* Part V – User Authentication

# Part III – Defining Database Models and Relations

This part will cover on using MongoDB as database using Mongoose as a plugin which works on top of MongoDB. We will define schemas and reference relations in our models.

**Reminder**: visit the first [part](file:///C:\Users\PC1\AppData\Local\02.%20ExpressJS-Fundamentals-Introduction-to-Node-JS\02.%20ExpressJS-Fundamentals-Introduction-to-Node-JS-Lab.docx) of this lab and more specifically - the "Models" section to see more details about each model which is going to be implemented.

## Set up MongoDB and Mongoose

Initially make sure to have installed [MongoDB](https://www.mongodb.com/download-center?jmp=homepage#community) on your machine. Next, cool UI program for visualizing data and schema would be pretty nice. For this lab is used [RoboMongo](https://robomongo.org/) – you can choose any other alternative you like or none at all.

To make sure that everything is up and ready run this command in command prompt:

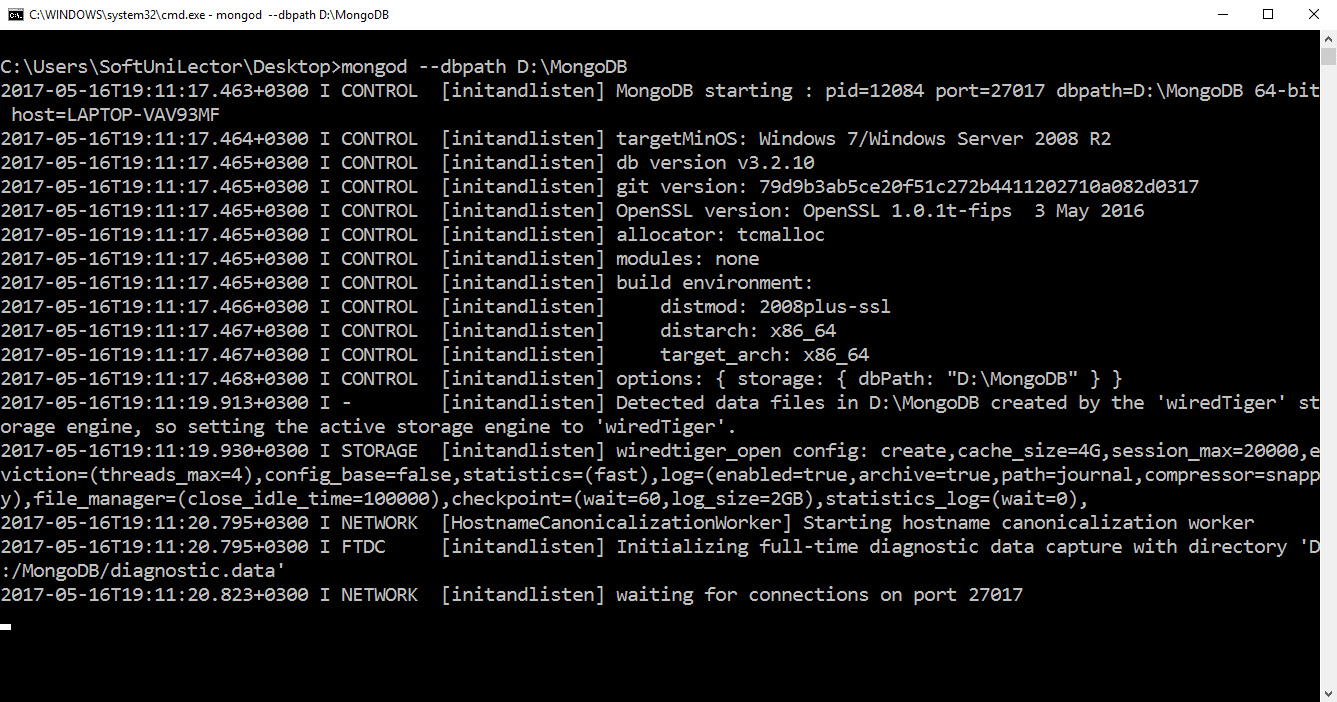
|  |
| --- |
| mongod --dbpath {yourCustomPath} |

yourCustomPath – place where database files generated from MongoDB will be stored

**\*Note** that: "--dbpath {yourCustomPath}" is optional by default and it will store information in "**C:\data\db**" – if such directory exists

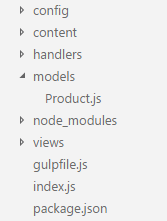
After the command is executed some output (or simply called "noise") should be printed. If everything goes well the program will continue working. Do not close the window because it is our connection to database.

**Warning**: Keep in mind that when you are working with any application which connects MongoDB - this window should always be open!



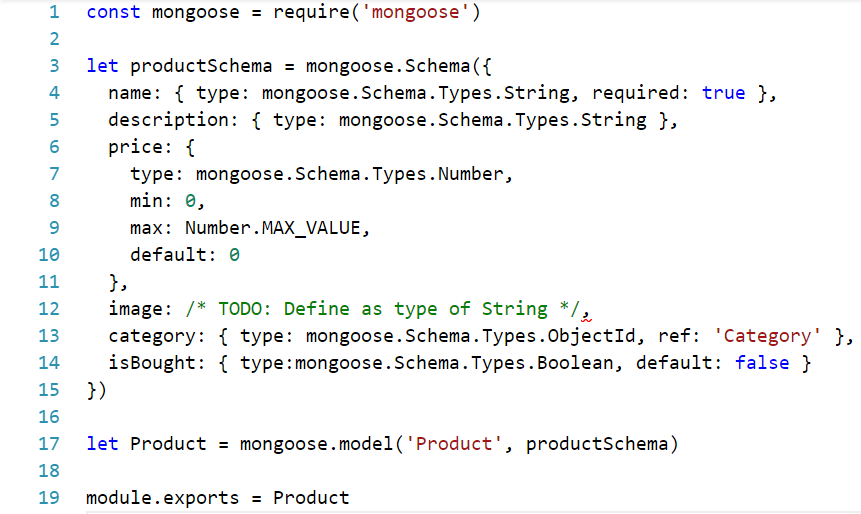
## Define Product Model

Create new folder called "**models**" inside the application. In it add new file named "**Product.js**":



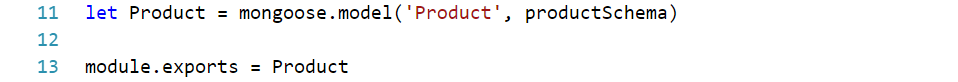
Defining database models is straight forward: first you define schema then you name that schema and export it as a model. Schema is an object containing key-value pairs - "propertyName: propertyType|{propSettings}". See below for clarity.

First reference "mongoose" then create product's schema:



**\*Note** that we have not declared all needed properties – they will be added later on (when the other models are added)

Finally create the model and export it:



## Set up Database Connection

Next, connection with database have to be established in order to persist data in it. For more clarity of our application structure some **refactoring** should be made.

First **delete** **database.js** and **database.json**. Then **create** in "**config**" folder new file called "**database.config.js**". This file (or so called module) will export function which will connect to **MongoDB** based on **config** parameter which is passed in:

|  |  |
| --- | --- |
|  | Make sure to npm **install** **mongoose** in current project. |
|  | The "config" parameter will be simple **object** containing some configuration data which may differ when we change our application environment (e.g. let’s say from "**production**" to "**development**"). In other words we make the application more durable because these setting we are passing may be change when we deploy the app on any server. |
|  | Requiring the model is **crucial** here because this is how we "**load**" it in our application and tell mongoose to use this as a database model. |

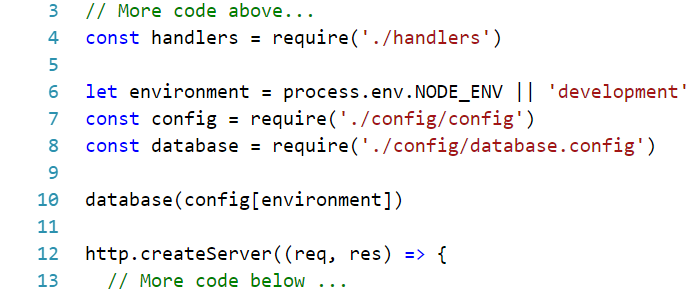
**Notice** that is defined event when we initially connect to our database (**once** the connection to database is **open**). The "console.log('Connected!')" will inform us when the **first** **connection** to database is successfully **made**.

Next create "**config.js**" file inside "**config**" folder:

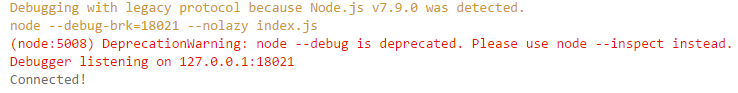


For now let's leave it like that – later on we may go back to add some additional settings and configurations.

Go to the **entry point** of the application – "**index.js**". Require the "**config**" and "**database.config**" modules, check if the application environment is set – if not make it to be "**development**":

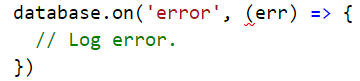


Now **start** the **application** and see if everything goes well. In the console: "**Connected!**" should be printed:



**\*Warning**: in "**home**" and "**product**" **handlers** we use **old** **database** module that we have deleted – just comment out those lines – in the next step we will configure how to get the data from [real](http://s2.quickmeme.com/img/6a/6acbf45c7fde7f40efb6f1561cd52d0eb62f70dff22602e3e096d215a7e79e84.jpg) database.

**\*Recommended**: if you want to handle any errors that the database is throwing you can specify on **'error'** callback (in "**database.config**"):



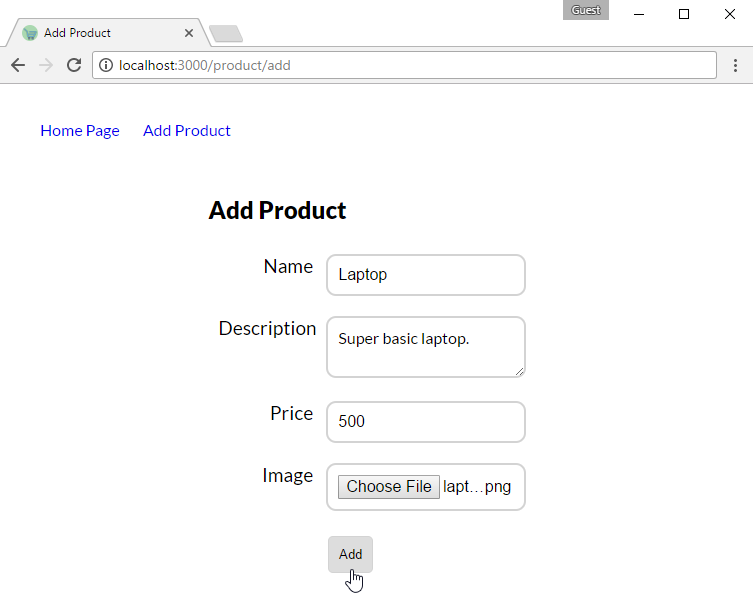
## Working with Models

We are now connected to the database – let's utilize this functionality and start making queries.

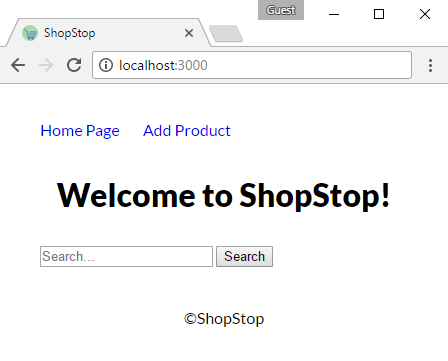
Let's start with the addition of products – go to "**product**" handler, require the "**Product**" model and insert the logic of saving products:

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

Let's test what we have done – start the application and rush in creating new product:

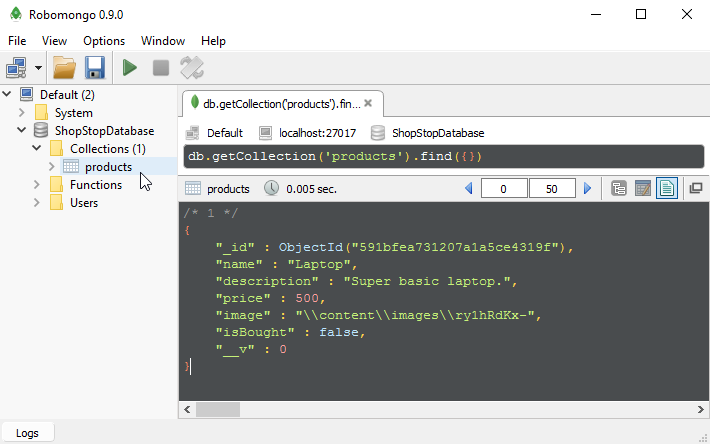


The **first** **signal** that everything went well is the immediate **redirect** to **home page**:



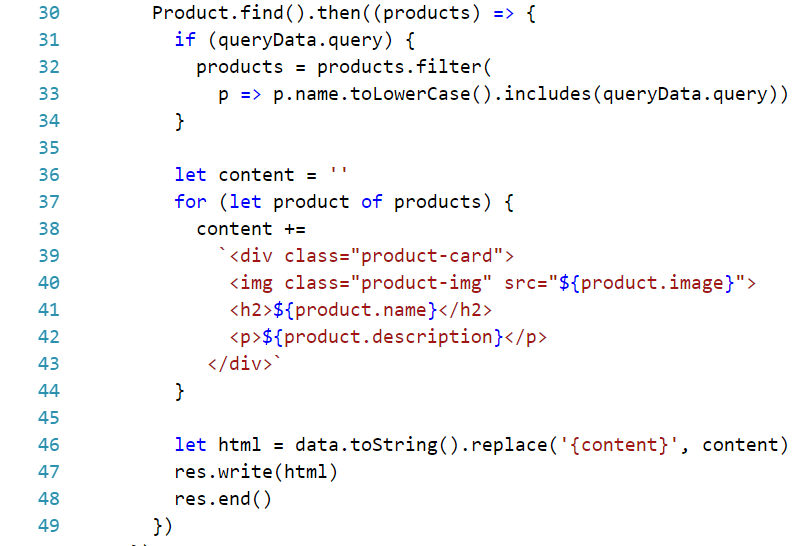
**\*Note** that the product is not displayed because we have not configured our **home** **handler**… yet.

Open **RoboMongo** or any other GUI (or console) client you are using and check if the database is created with proper collection name and data in it (see this [section](#_SetUp_RoboMongo) if you have not **set up** **RoboMongo** already):



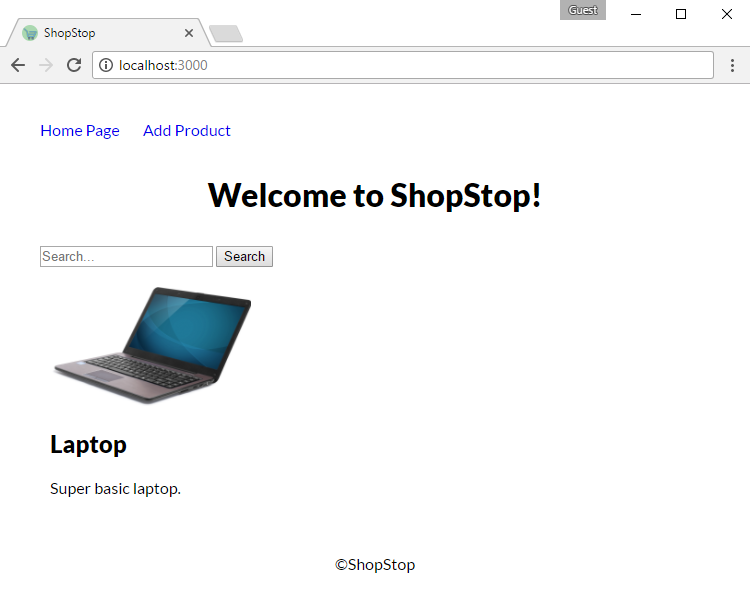
**\*Note** that your database name may differ depending on what you have configured in the connection string.

If there are no problems we can continue with **home** **handler** – again **require** the "**Product**" model’s module (bad pun sorry ☹) and use it to **load** products:



Basically you make call to Product.find() than in the promise you will **receive** the whole collection of **products** and you just have to **move** all the **logic** sending the html **inside** that **promise**.

Again, **re-run** the application and see if anything shows on home page:



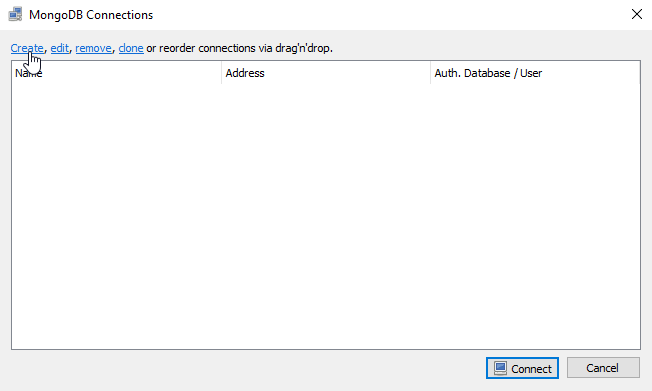
### Set up RoboMongo

If you are using **RoboMongo** but you have not configured it already do the following steps otherwise feel free to skip this part:

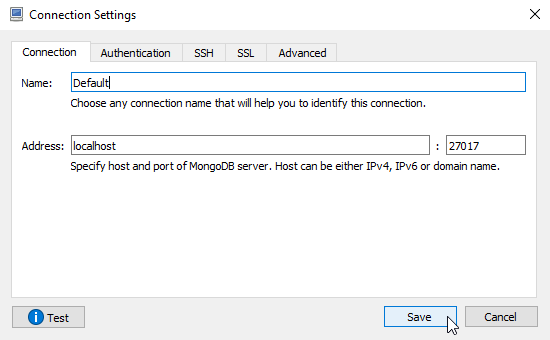
1. Open RoboMongo:



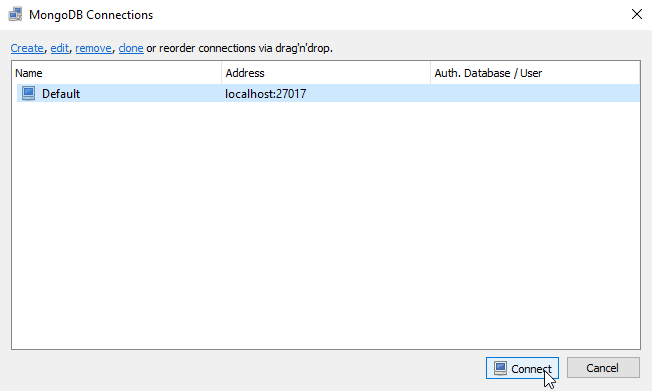
1. Create new connection (in upper left corner):



Window should pop out after clicking "**Create**", in it just give name to the connection and hit "**Save**":

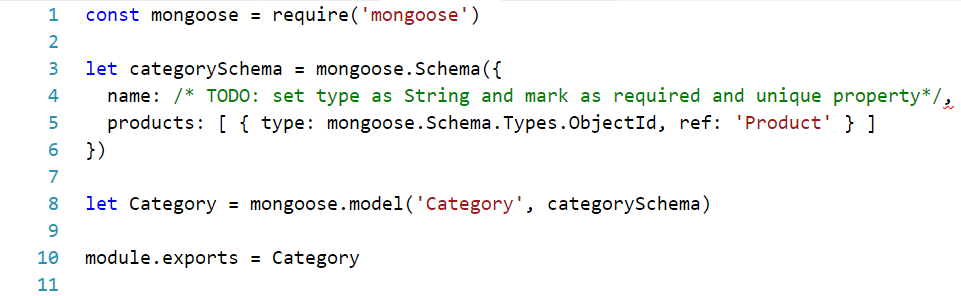


1. Connect to newly created connection:

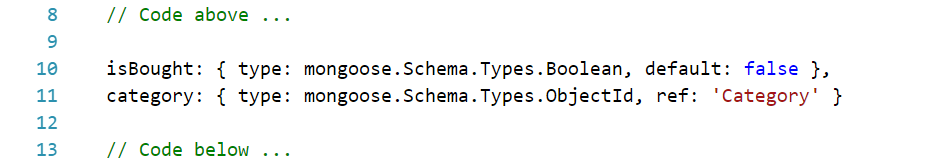


## Define Category Model

Let’s continue with adding the "**Category**" model. In a separate file inside the "**models**" folder create new module named "**Category.js**":



We are doing something different here – we define **products** as array of object of type **ObjectId** and we state that these Ids will reference **Products.** This way we define **one-to-many** relationship, which means that **one** category can have **multiple** products**.** Now we need to change the **Product** model in such a way, that every product can have **only one** category:

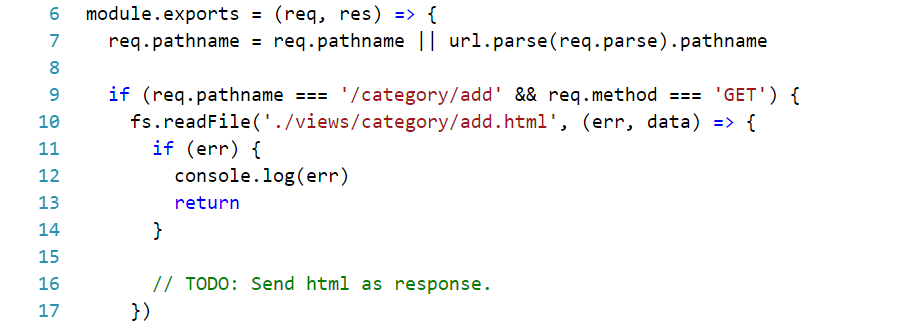


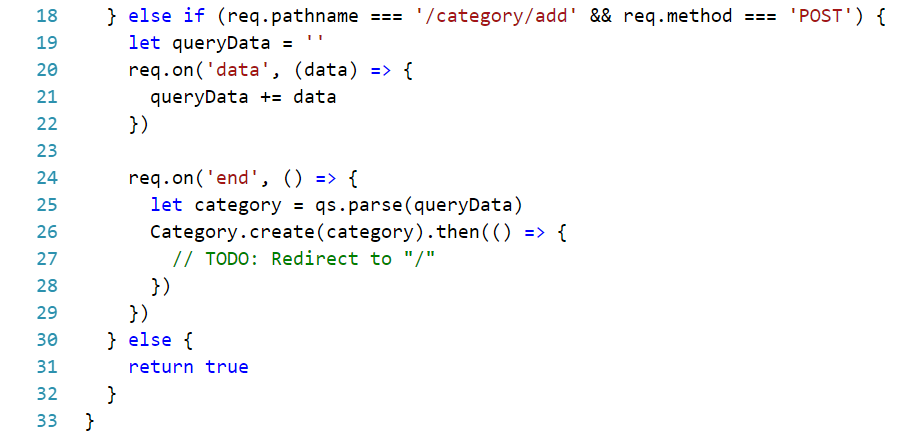
Next, move on creating categories dinamically inside our application.

## Category Create

In this step new handler should be added – the "category" handler. This handler will be responsible for creating a new category (for now – later on more functionality will be covered).

Go in "**handlers**" folder add new "**category.js**" file and implement the following:





The logic in this **handler** is **identical** with **product** handler: we use the "**GET**" method to **return** **html** form as a result and we use "**POST**" to **retrieve** **data** from that form and insert it in our database.

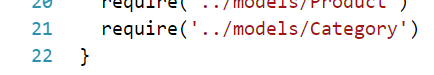
**\*Note** that you should require all needed modules

Let's create that html file that we want to display when someone wants to add new category:

|  |
| --- |
| <!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">  <link rel="stylesheet" type="text/css" href="/content/styles/site.css">  <link rel="icon" type="image/x-icon" href="/content/images/favicon.ico">  <title>Add Category</title>  </head>  <body>  <header>  <nav class="nav">  <ul>  <li><a href="/">Home Page</a></li>  <li><a href="/product/add">Add Product</a></li>  **<li><a href="/category/add">Add Category</a></li>**  </ul>  </nav>  </header>  <main>  <form id="add-category-form" class="center-form" method="post">  <h2>Add Category</h2>  <div class="form-group">  <label for="name">Name</label>  <input id="name" name="name" type="text" class="input-field" />  </div>  <div class="form-group">  <input id="add-category-btn" type="submit" class="btn" value="Add">  </div>  </form>  </main>  </body>  </html> |

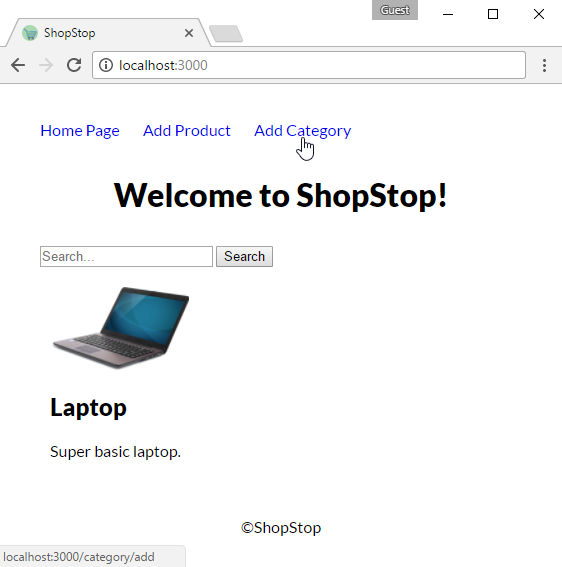
In the **navigation bar** however is added **new link** (compared to what we have in "**home/index.html**") – this is the link for **creating category** (it is bolded in the html above). Go back to " **home/index.html**" and "**product/add.html**" and **add** it to both views.

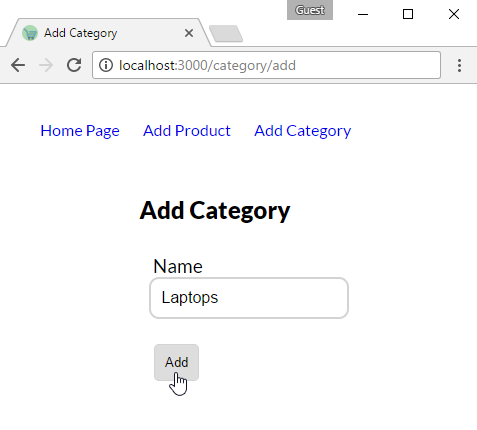
Go to **database.config** and in the end add the following:



The handler is implemented, the view is created – we are ready to go … well, not yet. **Before** **testing** the above code there is **one thing left** to do – is up to you to **configure** it.

Now, that we completed the above steps let’s add new category:



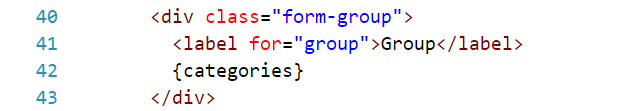


After clicking the button "**Add**" you should be redirected to home page. Then check in database (using **RoboMongo** etc.) if everything passed as expected.

## Using Relations

We can add products and also categories, but we haven’t connected them in our application. Yes, we have added so called foreign keys – fields that reference from current model to other model, but how we decide each product in which group is going? This is what we are going to do in this section.

The easiest way to say which product to which category is **related** is by making the creator **choose** one **category** **when** initially **creating** the **product**. This means that we should to "**product/add.html**" and add the following:



**\*Warning**: The above snippet should be right above the "Add" button.

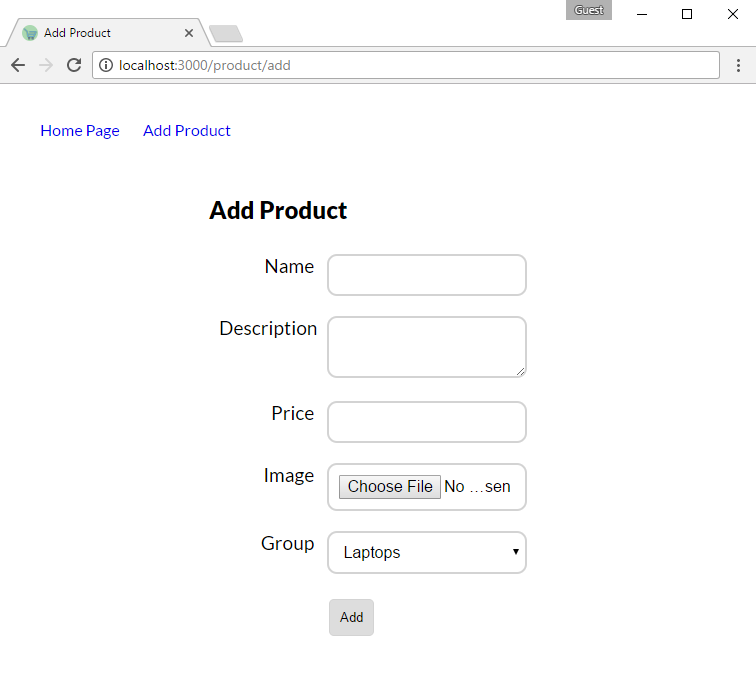
We have placeholder to fill – it's purpose is to generate [dropdown](https://www.w3schools.com/tags/tag_select.asp) menu displaying each group name individually. In the placeholder we will put all categories that we have in database.

Now let's get to the **handler** (**product**) and instead of sending directly the html – make query to **take all categories**, **place them in the html** file and finally **return the result** (the result of replacing) as response:



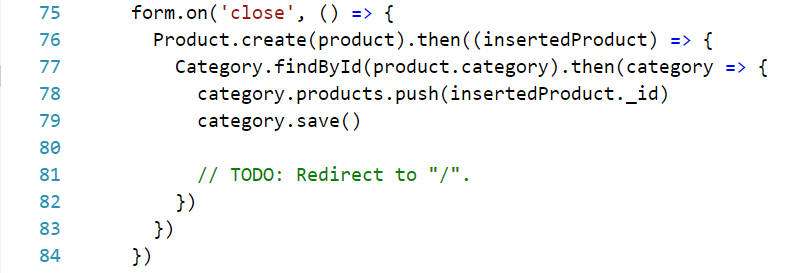
Note that in the <option> tag we specify value attribute to be equal to group's id. This will be the **value** which will be **send** to the **server** instead of it's name which is pretty awesome because it save us one additional query to retrieve the category's id by it's name.

Now try to create a new product**, new dropdown** field should be **displayed** and inside of it **all categories** **inside** the database:



We are almost ready to go. If we create a new product now the product will know to which category is pointing to because we choose it from the dropdown however we should update that specific category and put the product inside it's collection. In other words – the **product** **knows** in which category it is, but the **category** **does not**.

In order to make the relation **two-sided** when we **add** the **product** we should find the category which the product is pointing to and then add the product's id into category’s collection of products:



This being implemented we are ready to go! 😊

Stay tuned for the next part because Express.js is coming alongside view engines!