# Blog: Javascript and MongoDB

This document defines a complete walkthrough of creating a **Blog** application with the [Express.js](http://expressjs.com/) Framework, from setting up the framework through [authentication](http://passportjs.org/) module, ending up with creating a **CRUD** around [MongoDB](https://www.mongodb.com/) entities using [Mongoose](http://mongoosejs.com/) object-document model module.

Make sure you have already gone through the [Getting Started: Javascript](https://softuni.bg/downloads/svn/soft-tech/Sep-2016/Software-Technologies-Oct-2016/06.%20JS-Blog-Basic-Functionality/06.%20Software-Technologies-JS-Blog-Getting-Started.docx) guide. In this guide we will be using: [WebStorm](https://www.jetbrains.com/webstorm/) and [RoboMongo](https://robomongo.org/) GUI. The rest of the needed non-optional software is described in the guide above.

**Chapters from I to III are for advanced users, but is recommended to be read. There’s a** [**skeleton**](https://softuni.bg/downloads/svn/soft-tech/Sep-2016/Software-Technologies-Oct-2016/06.%20JS-Blog-Basic-Functionality/Blog-Basic-Functionality-Skeleton.zip) **which you can use and start from chapter IV.**

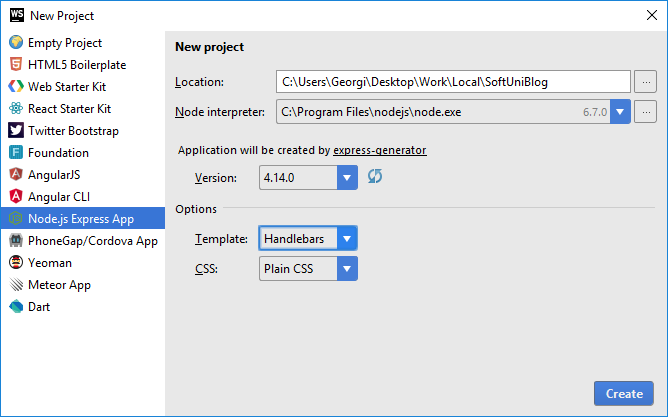
# Set Up Node.js Express Project

**WebStorm** comes directly with project structure plus we don’t need to download any plugins in order to develop our Node.js/Express.js application

## Create the Project from IDE

Once you have installed the plugins and started the **IDE**, you will have in the **Create Project** context menu either a “Node.js and NPM**”** -> “Node.js Express app” (**IntelliJ** with Node.js plugin) or directly a “Node.js Express app**”** one (**WebStorm**)



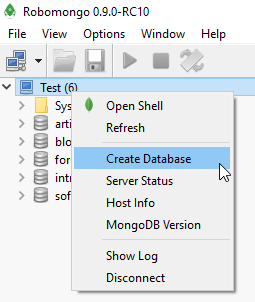


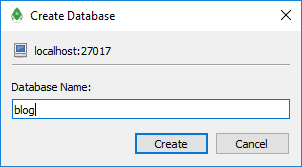
Make sure that you have [Node interpreter](https://nodejs.org/en/) installed and the chosen directory is the right one.

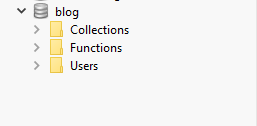
* Also choose **Template** to be [Handlebars](http://handlebarsjs.com/).
* Express recommended **versions** are any above: 4.0.0

## Create Database

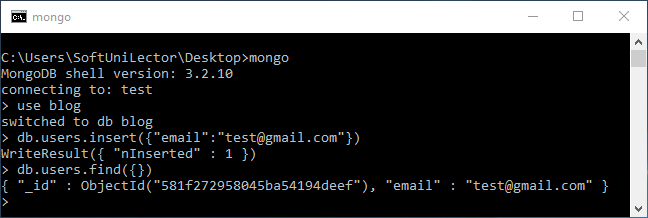
Open **RoboMongo**, connect to the default instance (with port: 27017) and create a database named “blog”.







Or if you want to do it using the **command line** use the following commands:

****

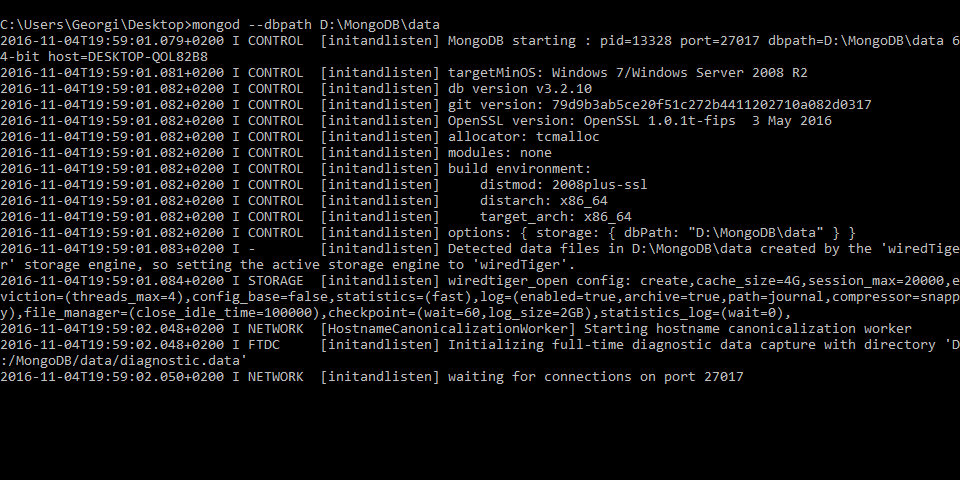
mongo

use blog

db.users.insert({"email": "[test@gmail.com](mailto:test@gmail.com)"})

db.users.find({})

**Note** that in order to use command line you should have all **environment variables** set or if not, you should run the command line from the place where **mongod.exe** is (“[C:\Program](file:///C:/Program) Files\MongoDB\Server\3.0\bin” - the version after server **might** be different – instead of 3.0 to 3.2, but the path is relatively the same). Also you should your **MongoDB** connection **open** (“**mongod –dbpath D:\example\path**” command).

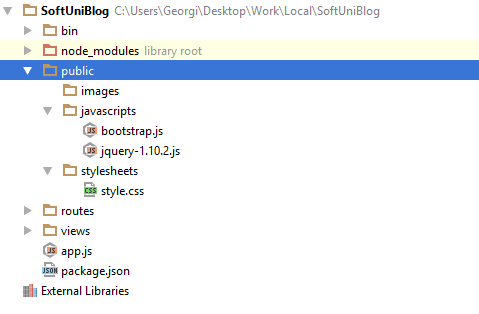


## Setup Layout

We will need a base layout for all of our templates. As we are using **Bootstrap**, we will need its css included in all pages, and the related scripts too. We can download the sample **blog design skeleton** from [here](https://softuni.bg/downloads/svn/soft-tech/Sep-2016/Software-Technologies-Oct-2016/03.%20PHP-Blog-Basic-Functionality/blog%20design.zip), where part of our **JavaScript** and **CSS** is included. In addition, we will need

1. [Bootstrap Date Time picker](http://www.malot.fr/bootstrap-datetimepicker/) for choosing dates in our forms
2. [Moment JS](http://momentjs.com/) for validating dates

All of our styles and scripts we need to include to our project. We should add stylesheets into the **public/stylesheets** and our public scripts in **public/javascript**. We will add the above two libraries when we need them:



Then we need to use this styles and script setting up a base layout in views/layout.hbs.

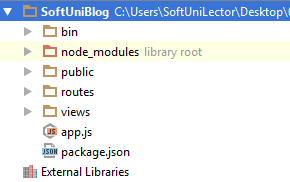
Setup a base layout as you wish or use the following one:

|  |
| --- |
| <!DOCTYPE **html**>  <**html**>  <**head**>  <**title**>SoftUni Blog</**title**>  <**link rel='stylesheet' href='/stylesheets/style.css'** />  <**script src="/javascripts/jquery-1.10.2.js"**></**script**>  <**script src="/javascripts/bootstrap.js"**></**script**>  </**head**>  <**body**>  <**header**>  <**div class="navbar navbar-default navbar-fixed-top text-uppercase"**>  <**div class="container"**>  <**div class="navbar-header"**>  <**a href="/" class="navbar-brand"**>SoftUni Blog</**a**>  <**button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse"**>  <**span class="icon-bar"**></**span**>  <**span class="icon-bar"**></**span**>  <**span class="icon-bar"**></**span**>  </**button**>  </**div**>  {{#**if user**}}  <**div class="navbar-collapse collapse"**>  <**ul class="nav navbar-nav navbar-right"**>  <**li**><**a href="/user/details"**>Welcome({{**user**.**email**}})</**a**></**li**>  <**li**><**a href="/article/create"**>New Article</**a**></**li**>  **<li>**<**a href="/user/logout"**>Logout</**a**></**li**>  </**ul**>  </**div**>  {{/**if**}}  {{#**unless user**}}  <**div class="navbar-collapse collapse"**>  <**ul class="nav navbar-nav navbar-right"**>  <**li**><**a href="/user/register"**>Register</**a**></**li**>  <**li**><**a href="/user/login"**>Login</**a**></**li**>  </**ul**>  </**div**>  {{/**unless**}}  </**div**>  </**div**>  </**header**>  {{{**body**}}}  </**body**>  <**footer**>  <**div class="container modal-footer"**>  <**p**>**&copy;** 2016 - Software University Foundation</**p**>  </**div**>  </**footer**>  </**html**> |

# Node.js Express app Base Project Overview

Node.js is a **platform** written in **Javascript** and provides **back-end** functionality. Express is a **module** (for now we can associate module as a **class** which provides some functionality) which wraps Node.js in way that makes coding faster and easier and it is suitable for **MVC** architecture.

Initially the project comes with the following structure:



We can see several folders here. Let look at them one by one and see what they are for:

* **bin –** contains single file named **www**, which is the starting point of our program. The file itself contains some configuration logic needed in order to run the server **locally**.
* **node\_modules** (library root) – as far as the name tells in this folder we put every library (**module**) that our project depends on.
* **public** – here comes the interesting part. Everything that is in our public folder (files, images etc.) will be accessible by every user. We cover on that later.
* **routes –** folder in which we will put our routes configurations. To make things clear: routes are responsible for distributing the work in our project (e.g. when user tries to get on "[www.oursite.com/user/login](http://www.oursite.com/user/login)" to call the specific controller or module that is responsible for displaying login information)
* **views –** like in the previous blog (PHP) we again have folder named **views**. There we will store the views for our model. Again we will use templates with the help of the **Handlebars** view engine.
* **app.js** – the script containing our server logic.
* **package.json** – file containing project information (like project's **name**, **license** etc.). The most important thing is that there is a "**dependencies**" part in which are all names and versions of every module that our projects uses.

# User Authentication

We have to implement the user’s authentication by ourselves. Hopefully we will use some security modules to help us with that. But first let's start with our User entity.

## Creating User Entity

Our users should be stored in the database (**MongoDB**). This means we need **U**sers collection. **Collections** are represented as an array [JSON](http://www.w3schools.com/js/js_json_intro.asp) objects. In Mongo, these objects are called **Documents**.

Let’s define rules for a user:

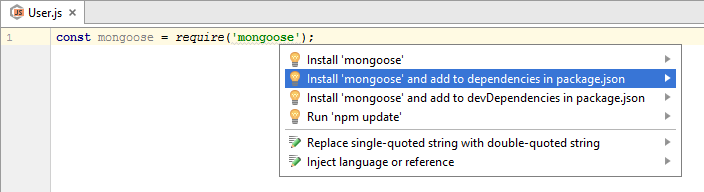
* Should have a unique login name, let’s say email
* Should have a passwordHash (which we will won't save in it's pure view)
* Should have a full name, let’s say fullName

We won’t user pure MongoDB. We will use [Mongoose](http://mongoosejs.com/). [Mongoose](http://mongoosejs.com/) is a module that will make creating and manipulating collections easier.

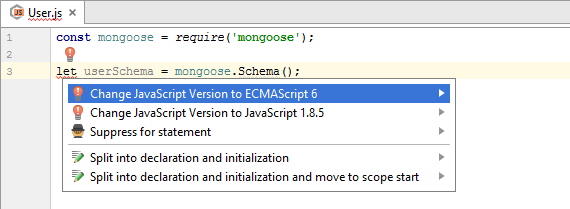
As a starter, create folder named "models". There create "User.js" file. In this file we will put our logic for the User collection (entity).

First we are going to require the "mongoose" module. Then we will create a [schema](http://mongoosejs.com/docs/guide.html) (look on the schema as a class in which we say what our objects will have). The schema will contain information about what the user will have (properties, functions and so on...).

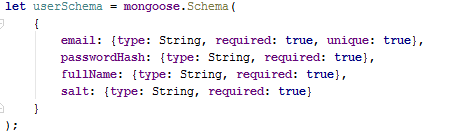
Javascript is dynamically typed language. The type of our variables is defined when the project is run. It’s called JIT (or Just In Time compilation). This is why this language is slow compared to C++ and even C#/Java. We have several keywords to declare and initialize a variable (var, let and const – and do not use var – just don’t). Use const when you create a constant value and let for any other uses.

The above command “require” will look into our libraries and will try to find a module with name: “mongoose” (it’s like calling using System in C# but instead of typing it on top of the file, we just assign it as a variable, in order to use the functionality in the module). Whenever we add new module it is a must to add it as a dependency in our package.json file. The IDE is smart and can do it automatically. “Alt + Enter“ is like calling “Ctrl + .” in Visual Studio.

Let’s create our user schema:



Unfortunately when we use “let” it is highlighted in red. This is because we have to switch our Javascript version to ECMAscript 6. ”Alt + Enter” to popup the helper, then click “Enter” and everything should be fine.



Here is how our Schema should look. We create schema by using that mongoose module we already imported. The Schema function accepts a Javascript [object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures). In plain words the above means: we will create a schema where every entity will have: email, passwordHash, fullName and salt (will explain it later). They are all type of String and they are all required. More info on types in Javascript read this [article](http://www.w3schools.com/js/js_datatypes.asp).

To finalize creating the User collection there are two things left to do: create and export a model. Model is just a wrapper of our schema. It let’s us to make queries to the database directly and even create, update and delete documents from our collection. Should look like this:



Creating the model is easy: just call “mongoose.model” and pass as first argument the model’s name and then the schema, that the model will be using. In order to export that model as a module, simply write that “module.exports” assignment. This means that everytime someone requires our “User.js” file he will get the User model.

## Create connection with MongoDB

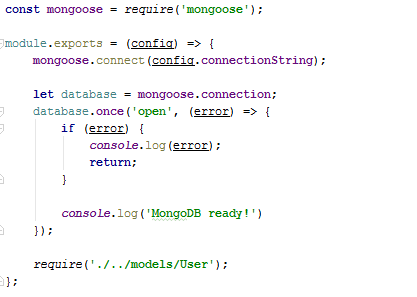
Before we start setting up our connection with database let’s create config.js file in our config folder (configception). There we will store information about our project root folder and a connection string, which is needed to connect with our database (MongoDB).



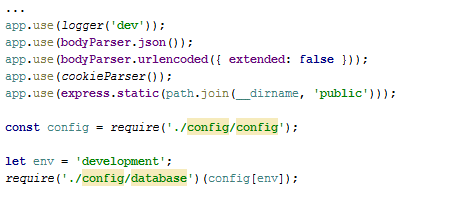
The idea behind creating a config file is to get our configuration variables from a separate place where they can easily be changed. Let's say that we will have two different configuration environments: production and development.

The two things that we will need for now are: rootFolder and connectionString. The rootFolder can be used when we need to declare path to some of the project's dependencies. As for the connection string the mongoose module will require it so it can save the changes we made to our documents.

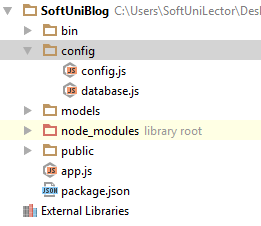
Let’s move onto creating the connection itself. We need to create a “**database.js**” file in our **config** folder. It should look something like this:



Now go back in app.js file and require that config module. Also, make sure that the code in database.js is also called:



The project structure should something like this:

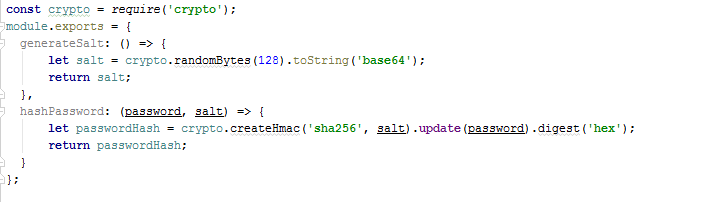


## Setting Up Security Configuration

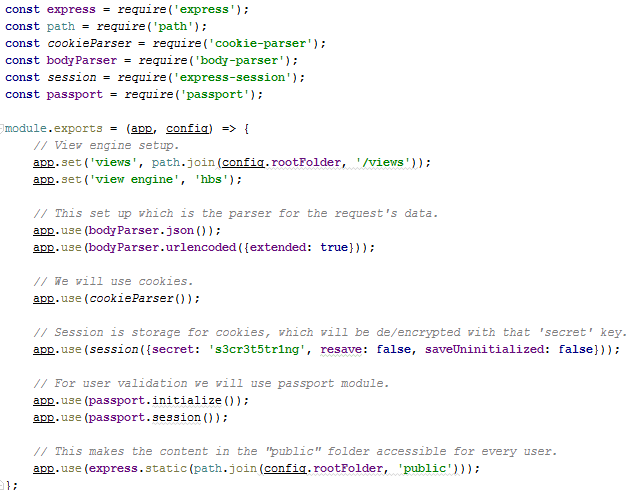
We have our model ready. Now we have to create some security configuration. First, create folder named “utilities”. Inside of it create file named: “encryption.js“. There will be our logic for generating [salt](https://www.addedbytes.com/blog/why-you-should-always-salt-your-hashes/) and hashing our password. So we have to create two functions in order to do that and also make them public so they can be useful.

First we will need some helper module (“crypto”). And in order to make functionality visible to outer world we will export an object which will have two properties which are functions (It’s [Javascript](https://cdn.meme.am/cache/instances/folder522/500x/63119522.jpg)):



Salt will be generated by firstly creating array of 128 random bytes, which are later going to be converted to their [base64](https://en.wikipedia.org/wiki/Base64) presentation. For our hashing logic, it is used the [SHA256](http://www.xorbin.com/tools/sha256-hash-calculator) hashing algorithm:

Create “express.js” file in the “config” folder. In it we will put some setup logic. Simply copy the ”app.js” file and remove the some of the code there and add the authentication modules – it should look like this:



Let’s talk about the modules we are using:

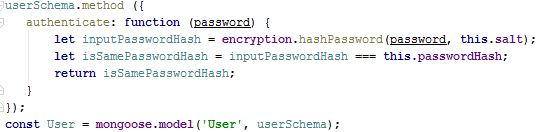
* express – wraps functionality that Node.js platform provides while making coding easier and faster. Look at the example with “express.static”. What it does is to take the provided file path (which is resulted by using the module below) static. This means that absolutely every file in that path is visible to anybody on our server (no-restrictions).
* path – supply utility functions for joining file paths (relative or absolute – doesn’t matter) or any tools needed around when using file paths.
* cookie-parser – cookies contain crypted data about current user and they are sent on every request. With this module we enable working with cookies.
* body-parser – parses data from the request’s body and making it accessible by simply mapping that data as a object with different properties. See [documentation](https://github.com/expressjs/body-parser).
* express-session – server-side storage. With that “secret” string differ cookies (sets every cookie an ID). Keeps information about current’s user connection. Only for development uses.
* passport – security module that uses session in order to save information about the user. It requires saving strategy (“Facebook”, “Google”, “Local” etc.) and also tells which data from the user to be put in the cookie. It binds two functions to our requirest: logIn and logOut.

Now let’s create “passport.js” in the “config” folder in and choose authentication strategy for our login:



As you see we have declared a function to authenticate user by it’s username and password. This means: first, the username should be existing in database and second - the given password to be equal to the one in database (hashed of course). Additional to that our function recieves third argument called “done” – another function which will be invoked inside the current function. The logic behind that is to pass error (if any have occurred) as the first argument and as second argument false – if you can’t authenticate user or the user itself whenever authentication is successful. This logic is needed to implement Passport Login strategy. In this project we will use “Local Passport” strategy. This means that the current user will be authorized only in the borders of our application(you can have a Facebook passport strategy where you will use Facebook credentials in order to log in).

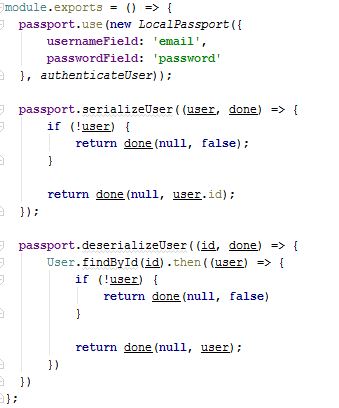
Here we use authentication method from the **User’s** model. It’s job will be to see if the currently given password is matching the original one. Here is the logic in the User’s **schema**:



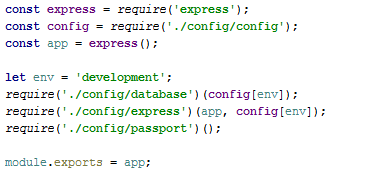
The passport module will provide us with two functions (as said above) which means that it automatically takes care of logging in/out the user. However the input data may be called differently than “email” and “password” (aka in our html form the input fields can be named differently) and this is why we can pass some configuration object in which we can set these names (usernameField: username). And to make that strategy complete we should pass it to the passport module using the keyword: “use”.

Next we will need to implement two functions for our passport module. They are called: serializeUser and deserializeUser. Passport is responsible for distinguishing users (as the passport in real life) so in order to do that we should tell him how to differentiate users.

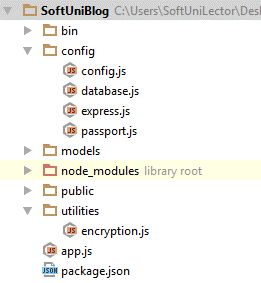
* serializeUser – given the whole user object return the unique identifier(in our case pass it to the “done” function).
* deserializeUser – given that unique identifier from the serialize function return the whole user object (again passed to “done” function).



Since we moved a lot of our logic in the “express.js” module we can safely remove it from “app.js”. Here is how the “app.js” should look:



Here is how the project structure should look like after the addition of these three modules:

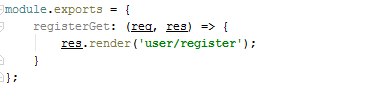


## Register user

Now that we have our authentication strategy and entity model, let’s start creating some views in order to register our first user! So, in our views folder simply create “user” folder. Put a “register.hbs” file in it and copy the following html:

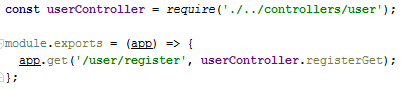
|  |
| --- |
| <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" method="post" action="/user/register"**>  <**fieldset**>  <**legend**>Register</**legend**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="inputEmail"**>Email</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="inputEmail" placeholder="Email" name="email" required value=**{{**email**}} >  </**div**>  </**div**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="inputFullName"**>Full Name</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="inputFullName" placeholder="Full Name" required name="fullName" value=**{{**fullName**}}>  </**div**>  </**div**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="inputPassword"**>Password</**label**>  <**div class="col-sm-4"**>  <**input type="password" class="form-control" id="inputPassword" placeholder="Password" required name="password"**>  </**div**>  </**div**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label"**>Confirm Password</**label**>  <**div class="col-sm-4"**>  <**input type="password" class="form-control" id="inputPassword" placeholder="Confirm Password" required name="repeatedPassword"**>  </**div**>  </**div**>  <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**button type="reset" class="btn btn-default"**>Cancel</**button**>  <**button type="submit" class="btn btn-primary"**>Submit</**button**>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**> </**div**> |

Now, after we have our user registration view, let’s create a controller to render it. For this purpose create a folder “controllers”. Then a file “user.js”. We will put there everything we need about our User model. Add a method which will render the html passed above:

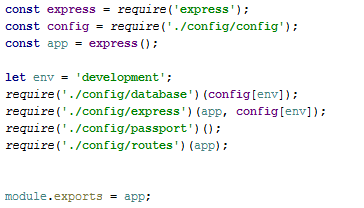


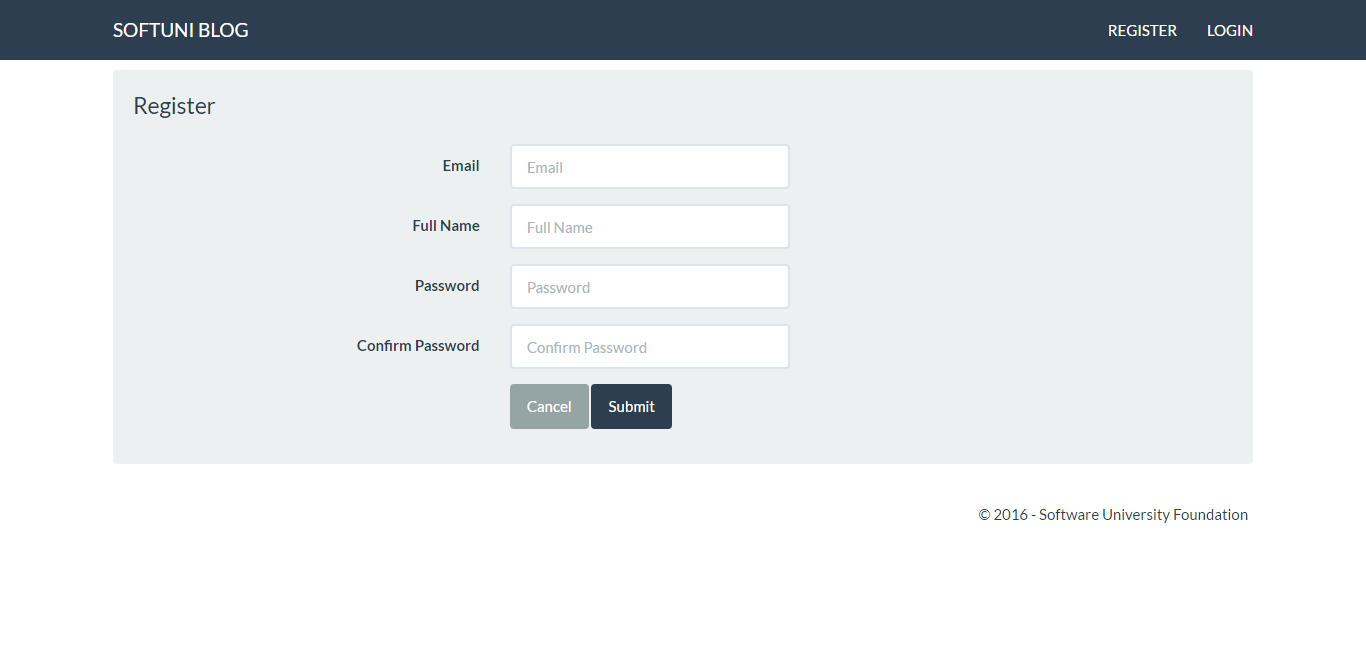
Our function in the controller will receive request and response as parameters…

What we need now is to define routes (routes will say which controller when to be called). The logic of routes is simple and lay on [REST](http://www.andrewhavens.com/posts/20/beginners-guide-to-creating-a-rest-api/) API definition. Let’s delete that routes folder that we have and create a “routes.js” file in the “config” folder where we can handle all requests:



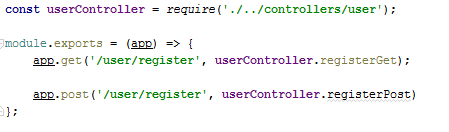
Now, require it in our app.js file:



If everything is ok and we run the program, when we go on localhost:3000/user/register the following should be displayed:

We have our form displayed (using GET request). Let’s dive deeper in “user/register.hbs”. If we look into the (the form tag) we will see that the form is having two attributes: “method” which is equal to “post” and an “action” equal to “/user/register”. This simply means that whenever this form is submitted (aka the button of type “submit” is clicked). It will create a POST request towards the URL described above:

This means that we need to create new route with same URL, but different HTTP method:

1. First add the route (in “routes.js” file):
2. Second create a new action in the User’s controller. That action should do the following:

Parse the input data. We can find it in the request’s body. You can access concrete arguments from it by passing the name of the input field (taken from the html). Take a look into “user/register.hbs” and you can see that every input field has a name attribute (name=”email” and so on):

 So if we want to take the “email” value we can do it with: “registerArgs.email”. For more clarity look at the pictures below.

Second, validate two things: is the email given already registered and are both passwords matching.

We have to connect to our database and check if there is any user with that email. Mongoose gives us functionality to do it by just requiring the Model. This means that we require the User model and search in all of it’s documents(entities). It can be done by using the command findOne(). This command accepts object which we can use as a filter:



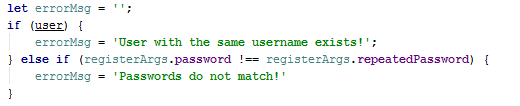
However here is something very important: this function is asynchronous (like the most query functions) and it will not directly return the user. This means that we cannot do something like this:



Instead we have to use promises. You can use promise with the keyword then(). If we want to print a user with specific email like in the code above we should do the following syntax:



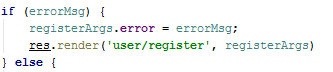
Now, validate if user is not existing and are passwords matching:



For every error case we will create a string variable in which we will save error message. Note that Javascript is weird when speaking about truthy and falsy values. Read this [article](http://james.padolsey.com/javascript/truthy-falsey/) for further clarity.

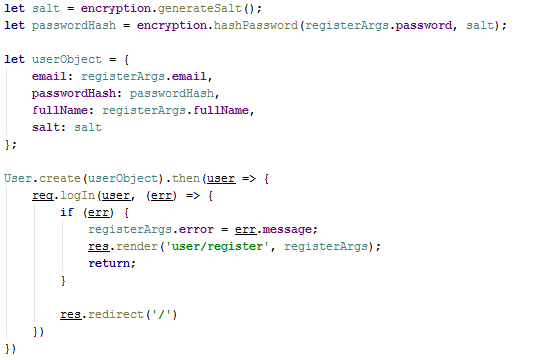
If any user with passed email is found it will return an object with some properties in it (properties from the User’s schema) and that will be considered true when converted to bool, else he will return undefined which is considered to be false when converted to bool.

After we have our validations, we should check for any violations. And we will simply do it with the following:



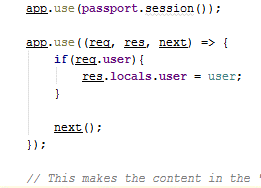
If any errors occurred we will simple reload the page. The key thing here is that we will reload it with the previous values and also with error message. Error message will be displayed in the layout(“layout.hbs”).

On the other side if our registration is successful we should insert new entity in database and log the current user:



\*Do not forget to require the “User” model and the “encryption” utility module.

One last thing before we move on the Login form. Go to the “express.js” and add the following:



We have just declared a [middleware](http://expressjs.com/en/guide/using-middleware.html), which will simply make our current user visible for both the views and the controllers.

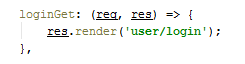
## Login Form

We will create our login functionallity in the same fashion we created the register one. In the previous step we did the following: register form **view** -> **controller** -> **route** -> **controller**.

Create “login.hbs” in “views/user” folders:

|  |
| --- |
| <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" method="post" action="/user/login"**>  <**fieldset**>  <**legend**>Login</**legend**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label"**>Email</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="inputEmail" placeholder="Email" name="email"**>  </**div**>  </**div**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label"**>Password</**label**>  <**div class="col-sm-4"**>  <**input type="password" class="form-control" id="inputPassword" placeholder="Password" name="password"**>  </**div**>  </**div**>  <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" href="/"**>Cancel</**a**>  <**button type="submit" class="btn btn-primary"**>Login</**button**>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**> </**div**> |

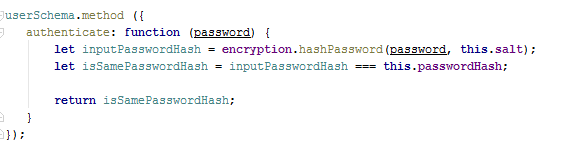
Then add action in the controller:



After that extend the “routes.js”:

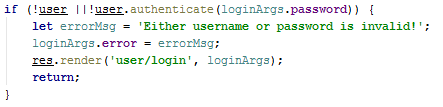


Go back to the controller and create login logic. This time we have we will have to validate not only if the user is existing but also if the hashed password of the user is the same as the hashed password given in the input. The easiest way to do that is to give every User a validation function. This is the easiest way because the users have all the needed information (**salt** and **passwordHash**). Go to the **User.js** in “**models**” folder and add this block of code:



Make sure that this **method** appending is **before** creating the User’s **model** (“mongoose.model” thing)**.**

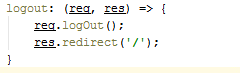
Again, on the **controller**. Write a search query (aka User.findOne) and **validate** user’s input:



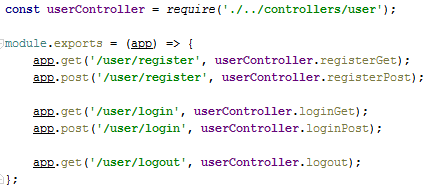
So, we have some **validation** on the input, what left is to actually **log** the **user**. You may use the **same** logic **as** we used in the **registration** section.

## Logout

Logging out is very simple:



Add the logout route. Here is how “**routes.js**” should look:



# Creating Articles

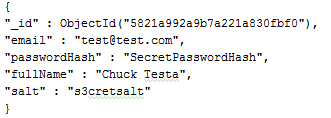
## Start MongoDB (Only if you are here from the start)

Before going ham on MongoDB let’s clarify some standings. MongoDB is a (NoSQL) database. But what is database? **Database** is just a **storage** for information. For now, we can assume that database is just a bunch of several tables in which we save information (SQL). Here is how our User table looks like from previous steps:



So, we have a couple of **tables**, each have some **columns** which gives us the opportunity to **store** **data**. This is example of SQL database.

**MongoDB** selects different approach. Instead of saving the data into table-columns format, it parses every object to **JSON** **string** and saves it. That’s all! Here is an example of user **object** saved in MongoDB:

****

One more thing: concrete **objects** are named **documents** – a list of **grouped** **documents** – **collection**.

Enough talk, let’s do some action:

1. Open MongoDB connection – open Command Prompt and type **“mongod --dbpath "D:\test\example"**”. What this will do is: create server (locally) on some of the computer’s port (default is **27017**) and will wait for any contact (command).
2. Connect to the newly-created server: depends on whether you are using console or GUI client

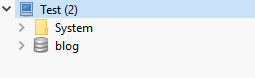
* For console client simply run “**mongo**” command. But in **different** window.
* If you are using **RoboMongo** just simply **start the application** and **connect** to the Mongo server.

Now you can communicate with the database and execute [commands](https://docs.mongodb.com/v3.2/reference/command/).

You can create a database named “blog”. Look in the previous step №2 “**Create database**”.

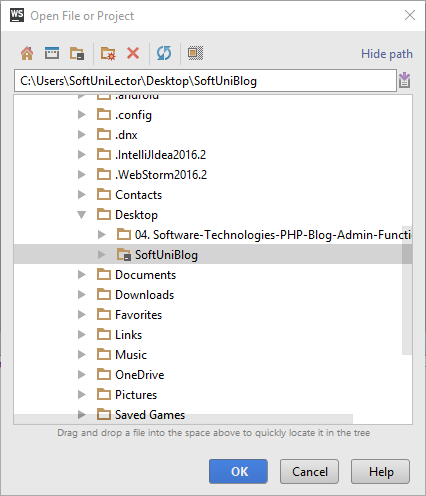
Summary: Now we know simple definition of a database. We saw different ideas behind implementing a database. Also how to start a MongoDB server from which we can create and manipulate different databases.

Here is how your connection **might** look like:

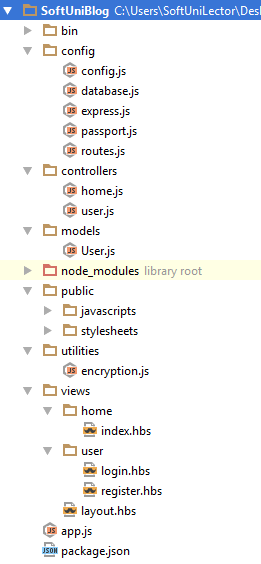


## Open/Create project

We have our database ready. Let’s go ahead and load the skeleton. Click open and find the downloaded and unzipped skeleton project: 



Note that the skeleton project has also one more controller named “**home**” and one more folder in “**views**” also named “**home**”. Don’t worry if you don’t have them in the moment, we will talk about them later. Here is how the project structure would be:



This is our Node.js project. In the previous steps, we described on how we got here. Now let’s talk about **Node**:

As we know it’s a **platform** written in **Javascript**, providing **back-end** functionality. This gives a lot flexibility because our **front-end** (html, using [jQuery](https://jquery.com/), [Ajax](http://www.w3schools.com/xml/ajax_intro.asp) etc.) also uses **Javascript**. This makes mutual **communication** easier. It is fast because it uses C++ behind the scenes and also because is capable of making asynchronous calls. It uses event loop [system](https://www.tutorialspoint.com/nodejs/nodejs_event_loop.htm).

Summary: we have downloaded the project and we are ready for further action!

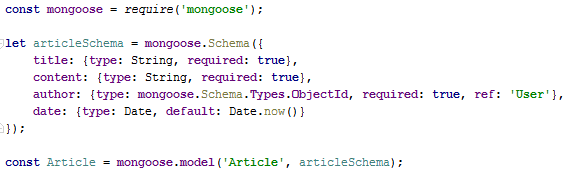
## Create Article Model

It is time to design our main entity – the **Article**. It will contain the following properties:

* title
* content
* author
* date

The interesting one here is the **author** property, because it is **already** a **model** in our database. Imagine that every time when we create an **article** we **bind** that author **information**. What if one **author** creates **50** **articles** and for every single one there is **separate** **property** containing the very same author information, wouldn’t be a waste of **memory**? Yes, so how to resolve that problem. We will simply put a **reference** key (something unique for the author – like **ID** or **name**) and instead of binding the whole information, just **save** that **key** in the article. Whenever we need more **detailed** information about the author we will just **query** our **database** one more time to give us information about an author with specified **ID/name**. This is called (database) **relations**. One author – has zero or many articles. We will cover on that in the next chapter.

Let’s create our model in the **Mongoose** way. **In** our “**models**” **folder** **create** a **file** named “**Article**.**js**”:

1. **Define** article **schema**:  
   
2. **Declare** **properties** with their types and any other **constraints** (such as **default** values, is current property **unique**, is it **required** and so on…):  
   

See also how we created the user schema (if you have skipped first 3 steps).

**Two** key **things** to notice: **author** is of **type** “mongoose.Schema.Types.**ObjectId**”. That “**ObjectId**” is the type of the **unique** **identifier** that our database puts on every document in order to differ two documents. This is done **when** you **initially** **save** a document (aka **when** you **create** an article – the database will put an “**id**” **property** – may simply be just a string with a **unique** content) and “ref” is telling that this “**id**” will be in “User” collection.

After we have defined our schema with all of it’s relations and constraints we will **wrap** it in a model. **Model** gives us the functionallity to perform **CRUD** operations. This means that if I **create** an **article** which has the same structure like the article’s schema (aka object **with** **title**, **content** etc.) by using the Model wrapper we can **save** it **to** the **database**. See this [guide](http://mongoosejs.com/docs/guide.html) for more explanation.

Almost done: export our model so it can be visible for the outer world:



One last thing: we need to add a reference to the **Article model** in our **database.js** file, so our database can know articles exist and can use them:

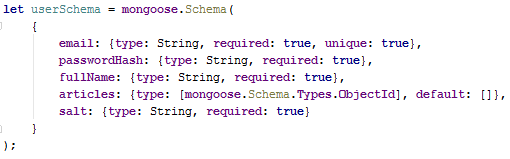


Summary: we now know how to create a user schema, wrap it in a model and define a relation with another model.

## Create Author - Article Relationship

Our **program** is like our real world – it is **based** on **connections** and interactions between it’s elements. We have a **user** which has **zero** or **more** **articles**. This relation is called **one** to **many**. Tomorrow we will want the **articles** to have **tags**. Many articles with many tags. Again relation – this one is called **many** to **many**. Our articles may have categories. **One** **article** – **one** **category**, from this side it looks like a **one** to **one** relation. Well, this is true **but** keep in mind that **one** **category** may have **many** **articles**. Here is the conclusion: relations can be: One to One, One to Many, Many to Many. There is one more called One to Few.

Let’s go back to the **author** - **article** relation. One article will have one author. We defined it with property in the article model. In order to complete the relation we have to change current user’s schema. In database world this is called **Migration**. Let’s do the migration in the **user’s schema**:



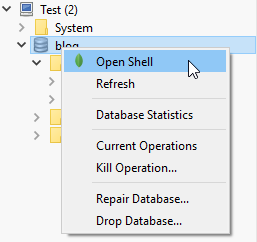
Just **add** **property** articles of type **ObjectId** **array**, with **default** value – empty array. This is our **migration**.

Summary: a database **relation** defines **connection** between two entities. The **relation** type depends on the point of **view**. In MongoDB **migrations** are as free as **changing** the **model**.

## Migrations

In MongoDB world, where we don’t have tables and columns, relations between models are more loose. The whole [responsibility](https://media.tenor.co/images/4c3b395bb7e3b40b780ac97f287b6ab3/raw) of handling a migration is given to the programmer. There are some [frameworks](https://github.com/emirotin/mongodb-migrations) that might help us with that, but for the scope of our project it would be simply overkill to use one.

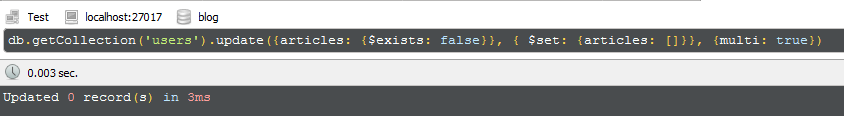
Let’s continue with our logic. In order to keep our data up to date we have to find all users who do not have ”articles” property and set the default value of it. This simply means that every change that we made to the schema will affect every next document inserted and will not make any update on existing documents.



This migration problem has two solutions: either delete all the old data and start over or run an update query on the not-updated entities. This can be done with the following command (which can be executed both on the console or GUI):

db.getCollection('users').update({articles: {$exists: false}}, { $set: {articles: []}}, {multi: true})

We can execute this command pretty easy on the console – just copy the update statement. When it comes to the RoboMongo – just select the “**blog**” database, right-click on it and choose “Open Shell”. From there the logic is the same as the console client (we will be using a console client in our graphical one):



Then run the command in the new command line window (Click **F5** to execute it).

Let’s look closer on this query:

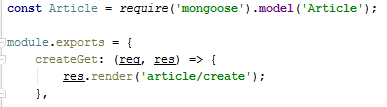
* db.getCollection(‘users’) – find all users.
* update() – update the first match found (by default)
* {articles: {$exists: false}} – for every user where “articles” is not existing property.
* {$set: {articles: []}} – sets “articles” value to empty array.
* {multi: true} – update all matches, not just the first one.

[Source](http://mongoosejs.com/docs/api.html#model_Model.update).

Summary: When having a **migration** we are the ones to **update**/delete the already **existing** data. Updating can be done with the “**update**” command and we can **pass** some **filter** **arguments** to it.

## Create Article Controller

The next part will be creating the article controller where we will put every logic connected directly with the Article model. Create “**article.js**” file in “**controllers**” folder. As a starter, we want to create a method which will render the form for creating an article. The controller might look like this:

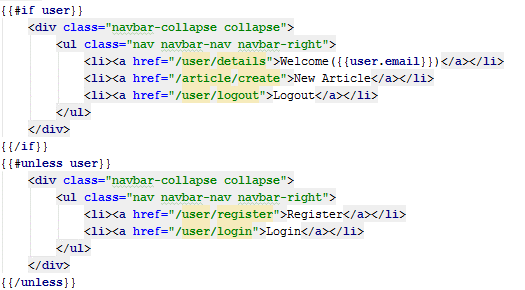


Note that we can require a mongoose **model** through the mongoose module just by passing the model’s **name**. Important thing about this way is that the code, initializing the Article model must be compiled before we try to access the model.

With the above code are in need to create a view which will render the form for creating article.

## Templating Article Form

In the beginning of the project creation we said that we will use the **Handlebars** view engine. So this time, instead of copying the html and directly moving forward, let’s see how **templating** is done. As an example we will take on **layout.hbs:**

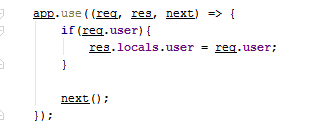


We can see that there is a lot of html but also there are multiple blocks of code which are not. These parts are for the view engine. Let’s explain what does to code below. With double curly brackets “**{{**” we say that the next part will not be html but a command for our view engine. This scope for the command ends with next closing curly brackets – “**}}**”. In current example, we can see that we have an **if** statement (#if). If the **variable** passed next to the “if” is **truthy** all the html until the **{{/if}}** will be displayed.

Okey, but what if the variable is **falsy** and we want to display something different? We will use “**unless**” in the same fashion that we did with “if”.

In the end the result will be: if there is any user logged in, display the first blog html (with “Welcome”, “Logout” etc.), else display the other blog html (“**Register**”, “**Login**” etc.)

But how does the view know about the current user? Look at “**express.js**” – there is a middleware that binds user in way that allows to be visible to the view (if you don’t have that middleware – write it somewhere after **passport.session()**):



Another **thing** to mention: look at the first **<a> tag** – there is a block “{{**user.email**}}”. This simply means that we can not only use “**user**” as a boolean but to actually **take** **data** from it! There are more commands to use (like “each”), but we’ll cover that later. For now, let’s go back to the article. We need a view which will display an **html** **form**. In this form, **data** (title, content etc.) will be **inserted** and we’ll have to take that data **into** our logic (for example, a **controller**). Create a “**views/article/create.hbs**” file:

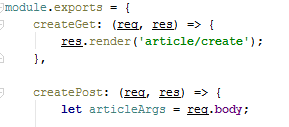
Our **form** html tag contains two important attributes: **method** – which defines what the HTTP [method](https://www.tutorialspoint.com/http/http_methods.htm) of the request will be, and **action** – the actual link where we want the data to go. So, wherever this form is submitted the request will go where the **action** attribute points.

<**div class="container body-content span=8 offset=2"**>  
 <**div class="well"**>  
 <**form class="form-horizontal" method="POST" action="/article/create"**>  
 <**fieldset**>  
 <**legend**>New Article</**legend**>  
  
 <**div class="form-group"**>  
 <**label class="col-sm-4 control-label" for="articleTitle"**>Article Title</**label**>  
 <**div class="col-sm-4 "**>  
 <**input type="text" class="form-control" id="articleTitle" placeholder="Article Title" name="title" required** >  
 </**div**>  
 </**div**>  
 <**div class="form-group"**>  
 <**label class="col-sm-4 control-label" for="articleContent"**>Content</**label**>  
 <**div class="col-sm-6"**>  
 <**textarea class="form-control" id="articleContent" rows="5" name="content" required**></**textarea**>  
 </**div**>  
 </**div**>  
  
 <**div class="form-group"**>  
 <**div class="col-sm-4 col-sm-offset-4"**>  
  
 <**button type="reset" class="btn btn-default"**>Cancel</**button**>  
  
 <**button type="submit" class="btn btn-primary"**>Submit</**button**>  
 </**div**>  
 </**div**>  
 </**fieldset**>  
 </**form**>  
 </**div**>  
</**div**>

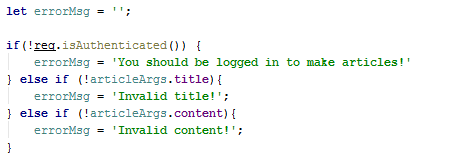
Summary: View engine helps us **put** **logic** in our views and also helps us **display** even **more** information. The best thing here is that that logic can be put **directly** **into** our **html** code. ☺

## Finalize Article creation

After we have our form displayed, it’s time to parse its data and complete our article creation. Go back to “**controllers/article.js**” and create another function to handle that logic:

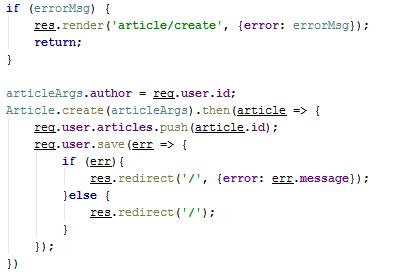


This is how the article controller should look for now. We have our article data parsed so start making some **validations**:



**req.isAuthenticated()** comes from the passport module and it checks if there is currently logged in user. This validation is optional for now. Other checks validate if the title/content is empty/undefined/null. If they are error message is created.

After all validations there are two things we can do: either if error occurred to inform the user or create article and put it in database:



**If** there are any **errors**, we will **re-render** the same page, but this time we pass object **with** an “**error**” which will be displayed in the “layout.hbs”. **Else** we will do the following: **assign** to the **article** object an **author** id. **Then** **save** it to database. **After** it’s saved, MongoDB will attach to the **article** an “**id**” which later we will **add** to the **author’s** **articles**.

Here is our **redirect**. We just say **where** to redirect (in our case will be just the home page – ‘’**/**”) and pass any additional info (object) to the view engine (if needed).

If you are coming with the skeleton skip the following step:

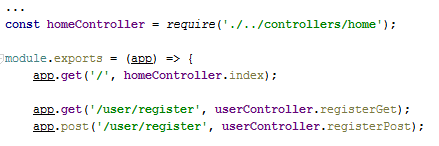
**Create** a **folder** named “**home**” **in** the “**views**” folder. Then create an empty “**index.hbs**” file. Go to “controllers” folder and add new controller named – “**home.js**”. Inside of it just simply type:

module.exports = {  
 index: (req, res) => {  
 res.render(**'home/index'**);   
 }  
};

And don’t forget to **require** the **Article.js** in the **database.js** :



Then add the home controller into the “**routes.js**” and the “home” routing:



If you had problems with this setup (or any other) feel free to look from the skeleton. ☺

Summary: We have completed our logic for creation an article. We have performed **validations** and based on them we can **inform** our **user** for any errors. After **saving** the **article** in database we **update** our user’s **articles**.

# Read, Update and Delete Articles

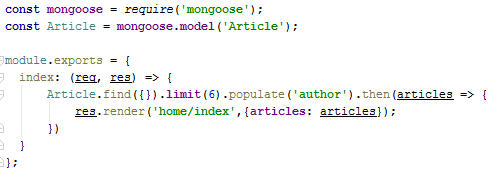
In this part, we will focus on manipulating the article entity.

## List Articles

What we will try to do now is to display 6 articles with information about every one of them. We want to do it on our home, so let’s go the “home/index.hbs” view and type the following:

Here, we use the Handlebars’ full strength. We are using an “**each**” construction (which works the same like foreach). In simple words we go through every article which was passed to us. For every single one we will display: it’s title (using **this** means that we are iterating over the **current** **article**), its content and author. The interesting part here is that we pass this statement: “**this.author.fullName**”. Remember when we created the Article **model**? The “**author**” property **was** of type “**ObjectId**”, right? Yes, here comes the cruicial point in getting the whole information (from our relation). Let’s see how we will get that information from the “**home**” controller:

<**div class="container body-content"**>  
 <**div class="row"**>  
 {{#**each articles**}}  
 <**div class="col-md-6"**>  
 <**article**>  
 <**header**>  
 <**h2**>{{**this**.**title**}}</**h2**>  
 </**header**>  
  
 <**p**>{{**this**.**content**}}  
 </**p**>  
  
 <**small class="author"**>  
 {{**this**.**author**.**fullName**}}  
 </**small**>  
  
 <**footer**>  
 <**div class="pull-right"**>  
 <**a class="btn btn-default btn-xs" href="/article/details/**{{**this**.**id**}}**"**>Read more **&raquo;**</**a**>  
 </**div**>  
 </**footer**>  
 </**article**>  
 </**div**>  
 {{/**each**}}  
 </**div**>  
</**div**>

What this will do is: get all articles, give me **6** of them, **populate** their “author” property. And after that send them to the “home/index” view. Populating a property means that MongoDB will attach addition object information based on the provided key.

Example: If we have an **article** with “**author**” **property** = “**a3fvce4GtT**” (which is the author’s ID) and we say that we want to populate that property, **MongoDB** will search in the **User** model for a user with the **same ID** and simply attach **all the information** it has for that user.

Also, notice the **link** for the “**Read more**”: it is “article/details/**this.id**”. This means that every article we want to display – we have unique route (URL), based on the article’s id. This is how our controller can get information about the article we want to see. We will go deeper in the next chapter.

Here is how the article should appear in our homepage:

Summary: We now know how we can **iterate** over an **object** in our **view** engine. Also, we saw the basics of “**populating**” a **relation** property.

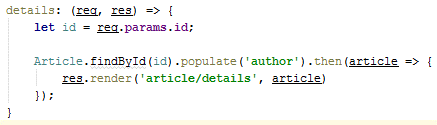
## Details Articles

Have you noticed the “**Read more**” button? Let’s implement it. We want to display more detailed information about the specific article when we click on it. Maybe some administration tools (like “Edit“ or “Delete”), too...

Again, our first step is to generate the view. This means that we have to create in our “views/article” folder another file named “**details.hbs**”:

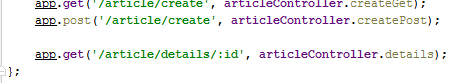
<**div class="container body-content"**>  
 <**div class="row"**>  
 <**div class="col-md-12"**>  
 <**article**>  
 <**header**>  
 <**h2**>{{**title**}}</**h2**>  
 </**header**>  
  
 <**p**>  
 {{**content**}}  
 </**p**>  
  
 <**small class="author"**>  
 {{**author**.**fullName**}}  
 </**small**>  
  
 <**footer**>  
 <**div class="pull-right"**>  
 <**a class="btn btn-success btn-xs" href="/article/edit/**{{**id**}}**"**>Edit **&raquo;**</**a**>  
 <**a class="btn btn-danger btn-xs" href="/article/delete/**{{**id**}}**"**>Delete **&raquo;**</**a**>  
 <**a class="btn btn-default btn-xs" href="/"**>Back **&raquo;**</**a**>  
 </**div**>  
 </**footer**>  
 </**article**>  
 </**div**>  
 </**div**>  
</**div**>

We have the view, now let’s use it in our controller:



Whenever we want to so see the specifics of a concrete article, we should inform the server which one. This information will be sent through the URL link. In the URL, we will pass the article’s “id” (we already did in the “index.hbs”). Once we get that “id” on the server side we can find the specific article and then pass it on the view engine.

How to get the information from the link? We will use **req.params**. But first let’s look how our routing will look like in “**routes.js**”**:**



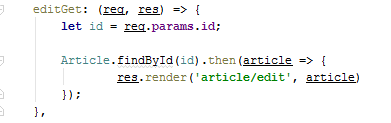
Just add the “/article/details/:id” part. This means that in the end of our link we are expecting a parameter named “id”. Later on while using req.params we can access that parameter by just getting it’s name as property of the req.params object. So, if we want to get a parameter with name “id” we will do the following: req.params.**id**. This is how we get parameters from our URL.

**Summary**: We saw how to **display** more **detailed** **information** about an **article**. We **passed** the needed **parameters** in our **URL** **link** which we can easily from the server side. Providing the **flexibility** to **display** **information** **for** **every** **article** in the database.

## Edit Articles

Let’s continue with article editing. In order to add that functionality we will follow again the same pattern as we did: **controller** (create get action - display the view), **view** (our html form), **route** (connect action with url route), **controller** (create post action – parsing input data). Note that the routing will be almost the same as the last step: instead of: ”**/article/details/1eazWre1rea**” it will be - ”**/article/edit/1eazWre1rea**”.

**First**, define action in our “**article.js**” **controller**:

****

**Second,** with previous action we say that we want to render the “**article/edit**” **view**. But if you look in our “**views/article**” **folder** there is no such view so let’s **create** “**edit.hbs**”:

<**div class="container body-content span=8 offset=2"**>  
 <**div class="well"**>  
 <**form class="form-horizontal" method="POST" action="/article/edit/**{{**id**}}**"**>  
 <**fieldset**>  
 <**legend**>Edit Article</**legend**>  
  
 <**div class="form-group"**>  
 <**label class="col-sm-4 control-label" for="articleTitle"**>Article Title</**label**>  
 <**div class="col-sm-4 "**>  
 <**input type="text" class="form-control" id="articleTitle" placeholder="Article Title" value="**{{**title**}}**" name="title" required** >  
 </**div**>  
 </**div**>  
 <**div class="form-group"**>  
 <**label class="col-sm-4 control-label" for="articleContent"**>Content</**label**>  
 <**div class="col-sm-6"**>  
 <**textarea class="form-control" id="articleContent" rows="5" name="content" required**>{{**content**}}</**textarea**>  
 </**div**>  
 </**div**>  
  
 <**div class="form-group"**>  
 <**div class="col-sm-4 col-sm-offset-4"**>  
 <**a href="/" class="btn btn-default"**>Cancel</**a**>  
  
 <**button type="submit" href="/article/edit/**{{**id**}}**" class="btn btn-success"**>Edit</**button**>  
 </**div**>  
 </**div**>  
 </**fieldset**>  
 </**form**>  
 </**div**>  
</**div**>

**Third**, we have the action – which will display the view we already created. Now is the part of putting things together. Go to the “**routes.js**” file and add:



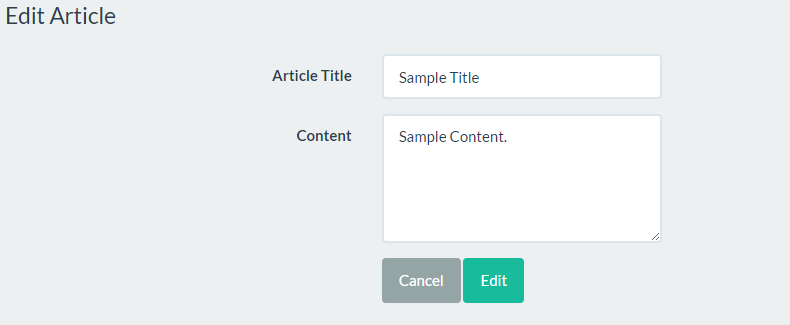
Cool, so if we start the server go on our home page and click on the “Read more” on one the articles (if you don’t have any – create one):



We will be rerouted to its details page:



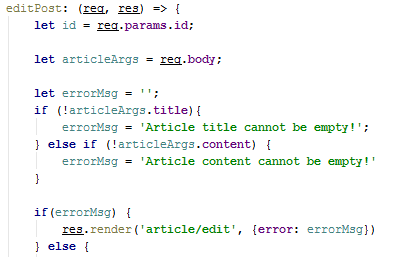
Click “Edit” and the following form should be displayed (Not that the title and content might be different):



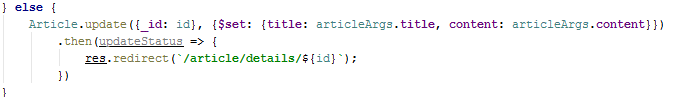
**Fourth**, we have our edit article html displayed, now it’s time to make the logic behind that “**Edit**” **button** (in the previous picture). Go back to the “**routes.js**” and add:



Get back to “**article.js**” in the “**controllers**” folder as our **fifth** step. And create the action we have just linked with the route:



This part is pretty much the **same** **as** the **creation** of an **article**. What differs now is in the “**else”** clause:



What this code will do is: find an article with specified **id** (the article we want to edit) and we will change its title and content. Then we will receive an update information object which is not important for now. After the update is done we just redirect to its details page. And **voila**, we are done with editing our article, so let’s test it:

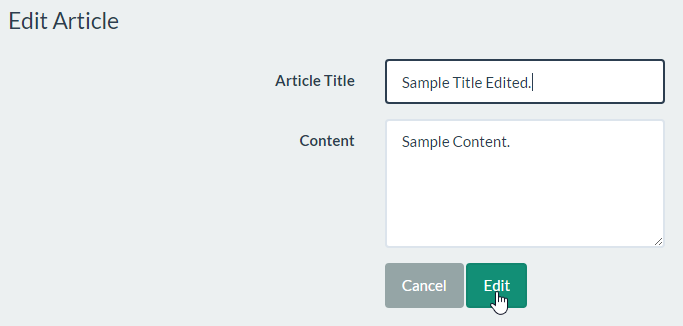


Go to the home page and select one of your articles.

Then click on the edit button:



Somehow change your article’s content or title:



Hit “Edit” and it should be displayed updated article information:

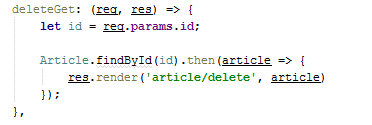


**Summary**: **Articles** can be **updated** and we saw we can **populate** some (html) **form** **on** **displaying** and using that in out update statement.

## Delete Articles

One last action of our CRUD remains: the deletion of an article. Again we will follow the pattern: **controller** – **view** – **route** – **controller**.

As you may have predicted, let’s go in our “**article.js**” in “**controllers”** folder and create another action:



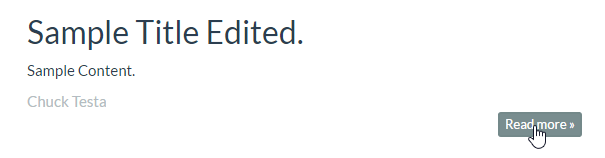
Good, now let’s create the “**delete.hbs**” file in our “views/article” folder:

<**div class="container body-content span=8 offset=2"**>  
 <**div class="well"**>  
 <**form class="form-horizontal" method="POST" action="/article/delete/**{{**id**}}**"**>  
 <**fieldset**>  
 <**legend**>Delete Article</**legend**>  
  
 <**div class="form-group"**>  
 <**label class="col-sm-4 control-label" for="articleTitle"**>Article Title</**label**>  
 <**div class="col-sm-4 "**>  
 <**input type="text" class="form-control" id="articleTitle" placeholder="Article Title" value="**{{**title**}}**" name="title" disabled** >  
 </**div**>  
 </**div**>  
 <**div class="form-group"**>  
 <**label class="col-sm-4 control-label" for="articleContent"**>Content</**label**>  
 <**div class="col-sm-6"**>  
 <**textarea class="form-control" id="articleContent" rows="5" name="content" disabled**>{{**content**}}</**textarea**>  
 </**div**>  
 </**div**>  
  
 <**div class="form-group"**>  
 <**div class="col-sm-4 col-sm-offset-4"**>  
  
 <**a href="/" class="btn btn-default"**>Cancel</**a**>  
  
 <**button type="submit" class="btn btn-danger"**>Delete</**button**>  
 </**div**>  
 </**div**>  
 </**fieldset**>  
 </**form**>  
 </**div**>  
</**div**>

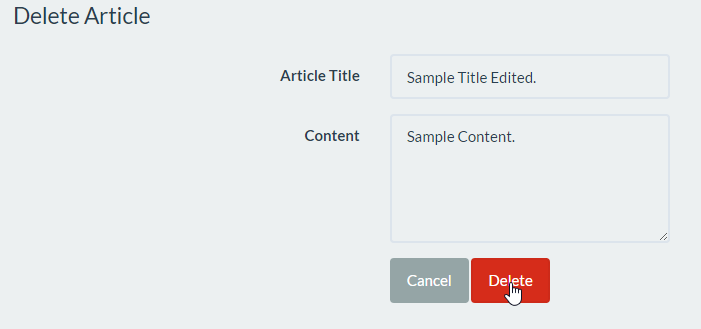
So, we have our action and view, go back to “**routes.js**”, so we can add it our application:



If we click on “**Delete**” **button** once we are on some of our **articles’** **details** something like this should appear:



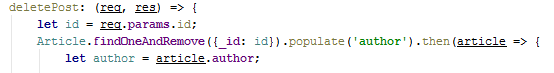




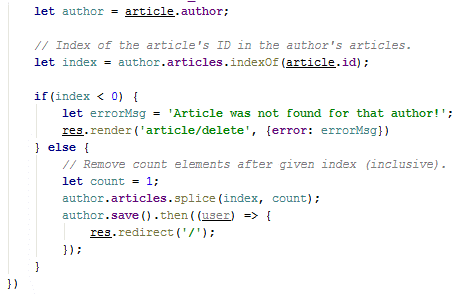
So, now we what is going to happen when we click delete button. If we look at our “**delete.hbs**” and see in the **form** **tag(<form>)**: a **POST** request will be sent to – “article/delete/articleId”. This means that we should create a routing for that. Go back to “**routes.js**” and **add** the following:



From now on we have to create the action we referenced above. Go into the article’s controller and add another function:



Here, we will use the command: “**findOneAndRemove**”. We will need also the author property since we are deleting the article we **must** **delete** it’s **reference** in **author’s** **articles**. Our code **continues**:



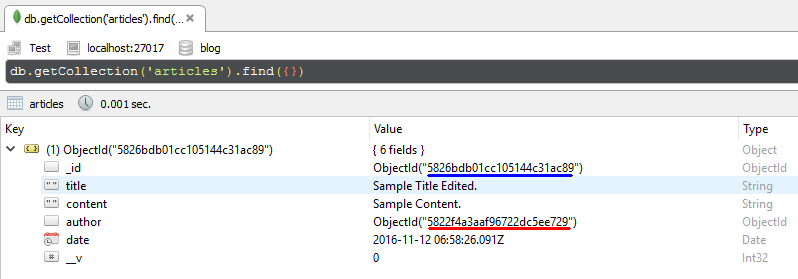
Here are the specifics of how to delete a specific element in array. For more information visit this [thread](http://stackoverflow.com/questions/5767325/how-to-remove-a-particular-element-from-an-array-in-javascript.). But the logic is simple: **first** find the **index** **of** the specified **element** and **then** **remove** the value **from** the resulted **index** (if index is valid). The **remove** is done with the “**splice**” command, which **receives** an **index** to **start** from and **length** of **elements** to **remove** (something like substring, but for an array).

Also, it’s crucial to run “**.save()**” **command** on article’s author in order **changes** to be **saved** **in** our **database**.

Our code works on paper and everything should be fine. It’s time for the second part of our development – testing, so let’s test if our article deletion is correct.

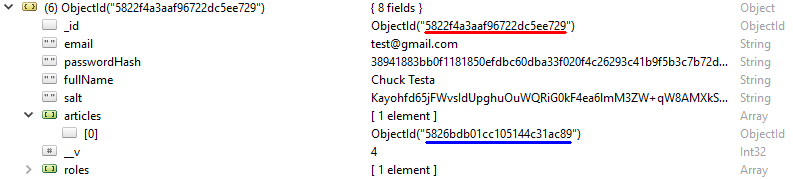
Let’s check our database to see how the data is looking before deletion. Note that the current data is sample for this example (it might look differently on your machine ☺):

In “**articles**” collection:



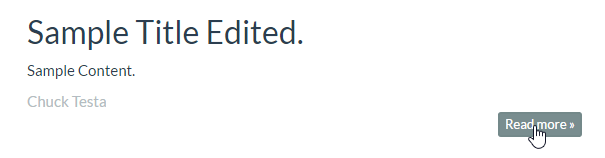
The blue highlighted **ObjectId** is **article’s** and the red one is for the **author’s**.

And in “**users**” collection to see our **author** (the red **ObjectId**):



We clearly have our documents properly saved in our database. Now let run the delete operation:

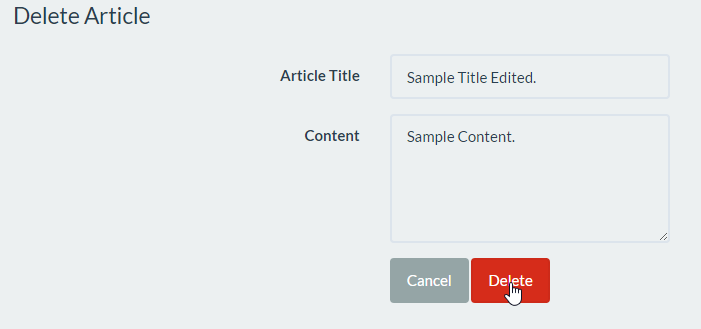
Go to **home** **page** and click “**Read more**” on some of the articles:



Then click the “**Delete**” button:

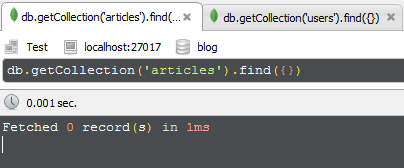


Then, confirm the deletion:



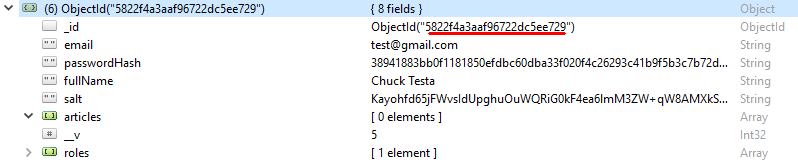
**After** we hit the "**Delete**" button we should be **redirected** to the **home** **page** where the **deleted** **article** **should** **not** be **displayed**.

Turn back to our “**articles**” collection:



In the current collection, the article was only one so after the delete operation the “articles” collection should be empty as is in the example.

Let’s see if the author’s is probably updated as well:



As you see there are **no** **articles** **in** “**articles**” property.

**Summary**: We have done it! We made **all** **CRUD** operations available **for** our **articles**. In this section in particular, we touched on how **relation** **should** **be** properly **updated** when **removing** **one** of the **related** **components**. In the next section, we will take more about **authentication** and **authorization**, **why** they are needed, what is the **difference** between them.

# Authentication and Authorization (Roles)

In the previous steps, we saw how to edit and delete articles. But the whole time we were ignoring one single question valid for our project. Who can create, read, update etc. who cannot! In the following steps, we will cover how to:

* **Authenticate** – simply the process of defining if there is logged in user or not.
* **Authorize** – is currently logged user (if any)has the credentials to do certain operation.

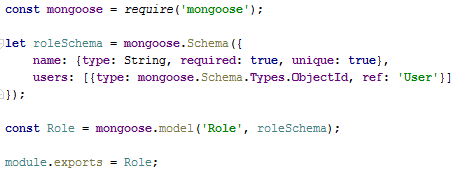
The most common way to do this is through using **roles**.

## Create Role Model

The finest way to connect an “**User**” with “**Role**” is to define **relation**. In order to do that we have to create “**Role**” **model** **first**. Our model should have:

* **name** – **unique** and **required**.
* **users** – **one** **role** may have **many** **users**, again we will reference just the **ObjectId’s** of the users.

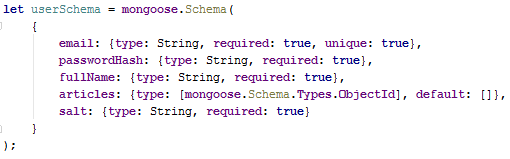
Go in “**models**” folder and create “**Role.js**”:



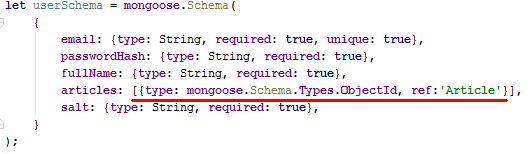
**Summary**: we have **defined** our “**Role” model** and “Role – User” relation but we have to create “User – Role” relation.

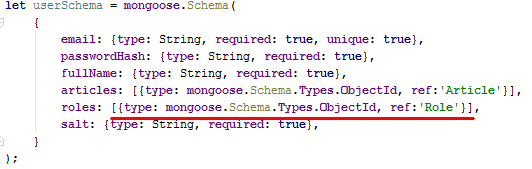
## Create User – Role relation

Go to “User.js” in “models” folder and take a look onto our current schema:



Everything looks fine, but **we** **have** a **problem** **with** our “**User** - **Article**” **relation** – we **don’t** **have** the “**ref**” property. And when we want to load all the information about the author’s articles - we won’t be able to do so since we did not tell **MongoDB** from which **collection** to get data from. Let’s fix this one:

  
Then add the “roles” property:

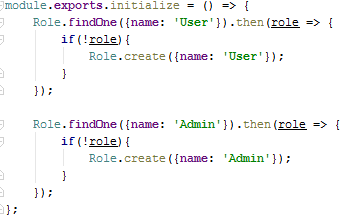
****

**Summary**: we have created our “**User – Role**” **relation**. As we can see **one** **user** may have **many** **roles**. What is left is to **update** **all** the **existing** **users** to have a roles property (**or** to **delete** **them** **all**) – aka to **migrate** the “**User**” model.

## Initialize Roles

We have **defined** our **relation** between “User - Role” and vice versa. But we have to think how about **what** **role(s)** **users** should **have** by **default**. But in order to have default roles these **roles** **should** **be** **existing** in the database (in order to make relation successful and complete). So in order to do that we have to do **two** **things**: **initialize** **two** main **roles** every time they are not existing and **adding** a **default** **role** to **user** when **registration**. In this section we will cover on the first part – initializing **roles**.

What this code will do is: **create** **two** **roles** with names “**User**” and “**Admin**”, when they are not existing in database. And this function can be called when we are connecting to the **database**. This means we just go in “**database.js**” and **add** the following:



A good way to it is in the “**Role.js**” module:



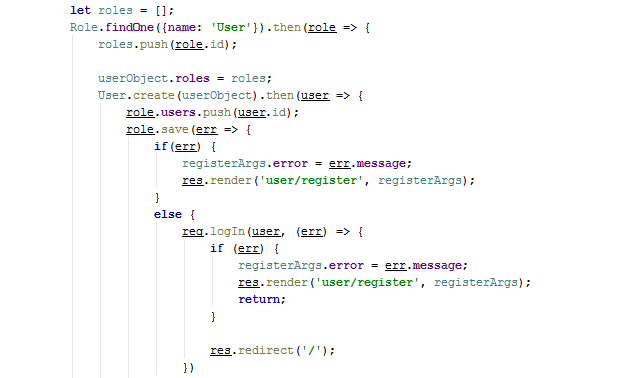
## Edit User Registration

Now let’s continue from the previous step and add the default (“**User**”) role for every new user. This means that we should go in our “**user.js**” **controller**.



On top of the file add the “**Role**” model:

Then in our “**registerPost**” function **update** the **code** **below** the “**userObject**” **variable**:



Let’s explain what is the above code doing:

* 1. Create array “**roles**”.
  2. **Search** in our “**Role**” model for any document **with** **name**: “**User**”.
  3. When that “**Role**” document is found insert it’s **Id** **into** the **array** (from step №1)
  4. Creates a **user** **object** (assigns “**roles**” **property** on it) and **then** creates a “**User**” **entity**.
  5. To the **role** (found in №2) **adds** the **user’s** **Id**.
  6. **Saves** the **role** entity with the newly inserted Id.
  7. **If** any **errors** occurred just **re-render** the view while displaying a message.
  8. **Logs** the **user**.

Twisted, isn’t it?

Summary: With the two steps above, we have completed making actual **relation** between **users** and **roles** **when** a **user** is initially **created**.

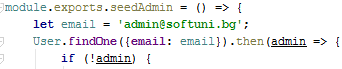
## Seed Admin

With now having an “Admin” role, would be nice to seed an admin user? Just for test uses, yeah, you [know](http://ekladata.com/I-mrdvkopRdfr5BZsIslBvjBMQk.png).. Let’s **create** a **“seedAdmin()**” function **in** our “**User.js**” **model**:

On top of the file add the “**Role**” model:

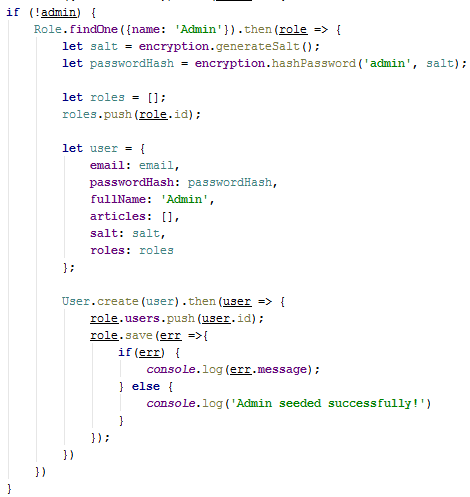


Then actually create “**seedAdmin**” (at bottom of the file):



Our logic is: we have a default email (here is: “admin@softuni.bg”). This email will be used to create an admin user **only**. **However**, if we find **any** user **with** this **email**, that means **we** **have** an **admin** user already and we have no need of creating another one.

Here is the logic of creating the user (it is similar to what we have done in our “**user**” controller):



Then go back and use it in **database.js**:



**Summary**: We have accomplished the **initial** **creation** **of** an **admin** user. Next time when we run the web server next to the “MongoDB ready!” message there should be the **message** about **admin** creating **too**!

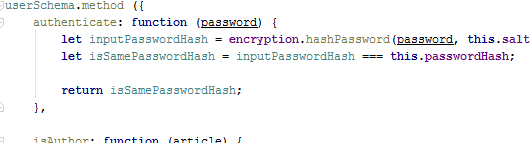


# Authentication and Authorization (Validations)

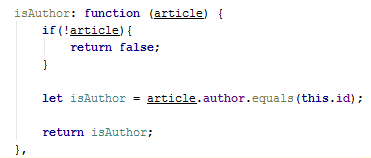
We have our roles ready, admin user seeded and user are now registered with default role. Let’s start making validations in our blog project.

## Create Helper Functions

In order to make **validations** look smoother and **easier** we should **define** some utility **functions** in our “**User**” **model**. These functions should be accessible for every user document, so let’s add it to the **user’s** **schema**:

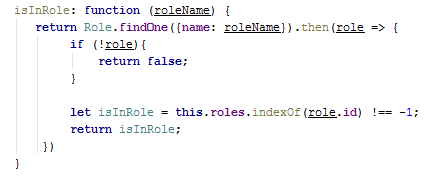


The first function (method) is define whether current user object is an author of an article or not:



Simple keeps our code of repetition.

The second method is about whether current user is in given role or not:

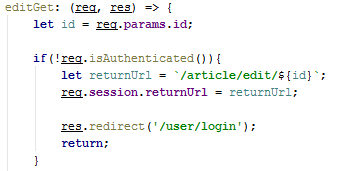


**However**, there is something tricky here. Since **we** **have** to **check** if the **role** is given to use exist **in** **database** and get its Id **and** since we **cannot** **get** that **directly** (we have to use “.**then()**” function – **promise**) we are forced to **return** a **promise** as well. It will **change** our validation **logic** **slightly**. Look below for any clarity.

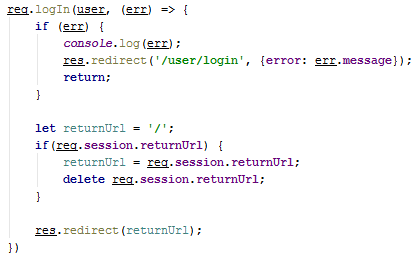
Summary: In the end our “userSchema.metnod()” should have 3 functions: “**authenticate**”, “**isAuthor**” and “**isInRole**”.

## Validate the Edit on Article

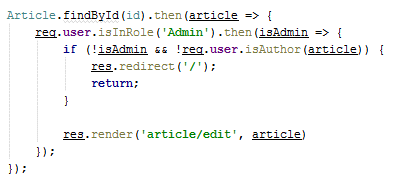
Time of validation has come. Now let’s forbid editing article for any user who is not the author or an admin. Go to the **article** **controller** and add the following logic:



What we use here is: “req.isAuthenticated()” – a function given by the “passport.js” module in order to **validate** if there is **currently** **logged** in **user**. And if user is not logged in we redirect to login page. It will be nice if we support the following functionality: after login, the user to be turned back from where he was (before the login). And the way to do it is simple – we just **attach** a “**returnUrl**” **property** **to** our “**req**.**session**” and then slightly changing the logic in the “**loginPost()**” method in our **User’s** **controller**:



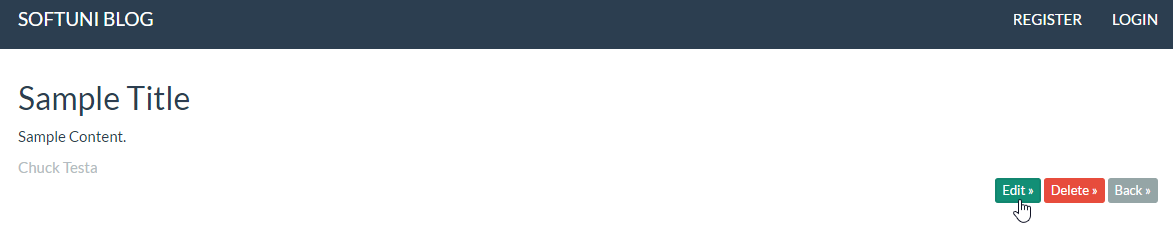
**Back** to the **article** **controller** ("editGet"), we need to add **next** another layer of security – make a concrete **check** if the current user **is** the **author** of the article **or** an **admin**:



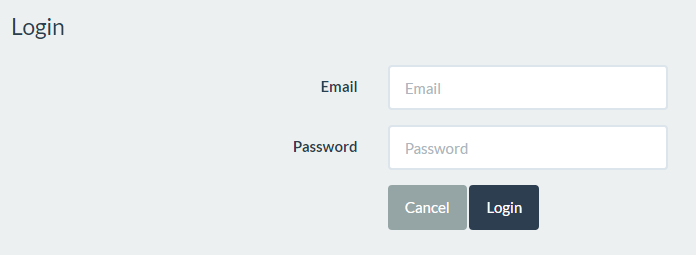
Since our check for the admin role returns a promise we are obligated to make our validation logic in the “.then()” function, **after** we have the result of the **role** **check**.

We have our validations done. Let’s test them:

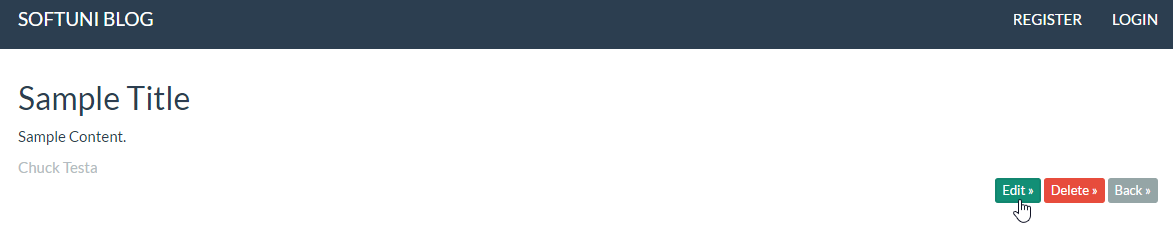
Try to **edit** an **article** when:

* 1. You are **not** **logged** in:  
     

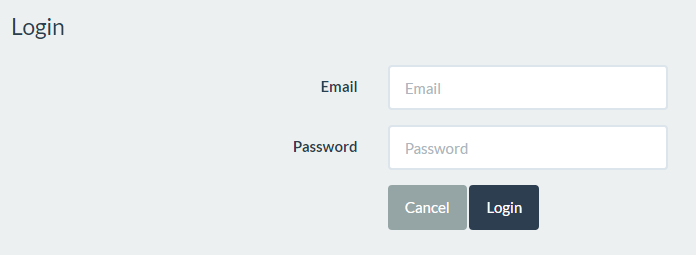
The following window should be login form:



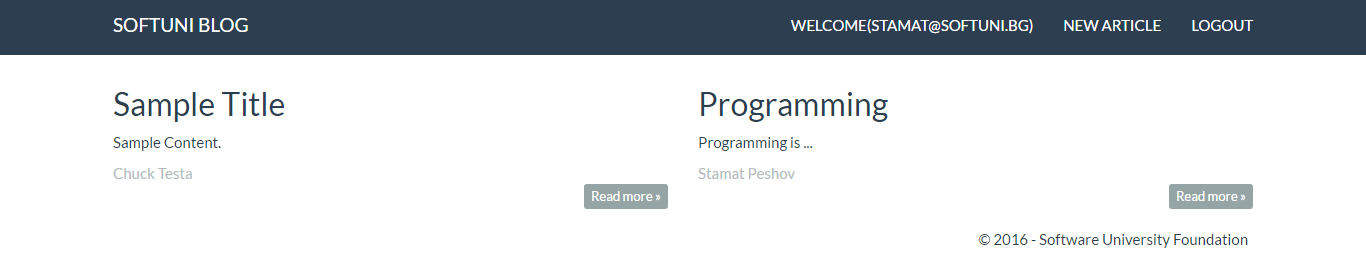
* 1. When you **are not** **logged** in, then you log in and you are **not** an **admin** **nor** article’s **author**:

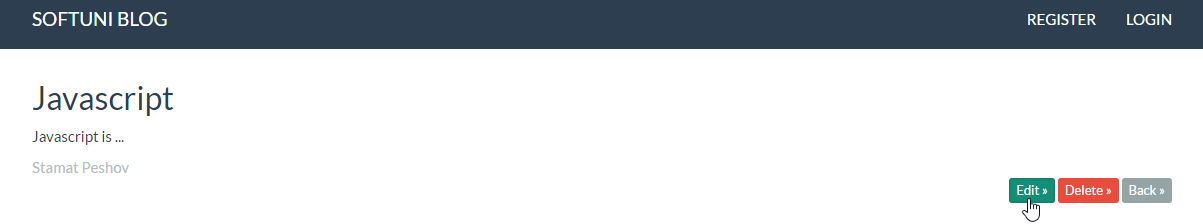


Log in:

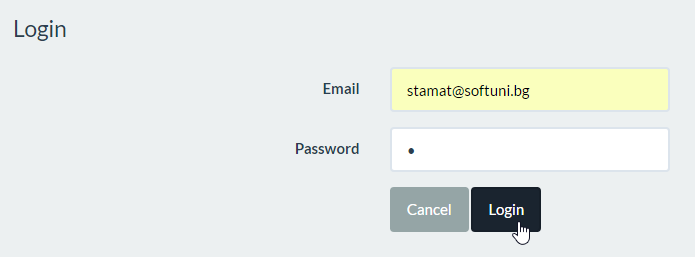


The **next** **page** should be the **home** page:

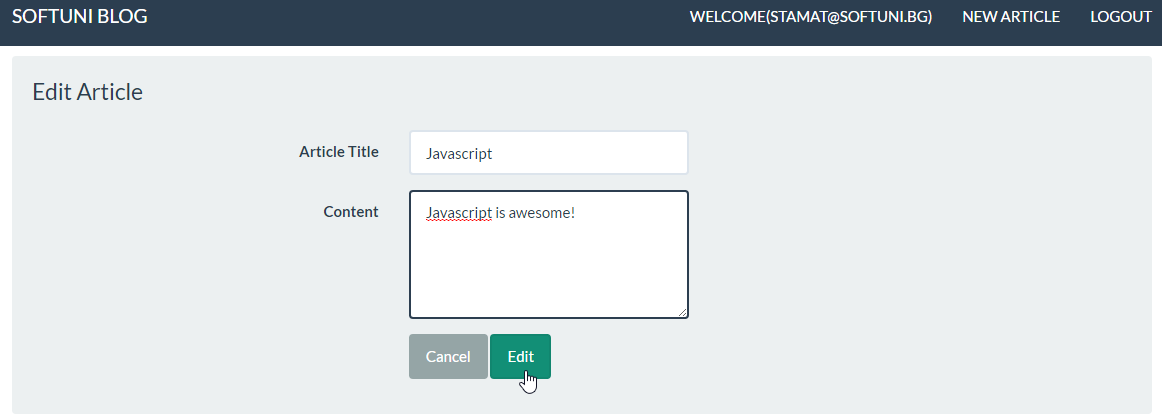


* 1. When you are **not** **logged** **in**, login and you are article’s author:  
     

Then the **login** form should appear:



Then the **edit** of **article** form:



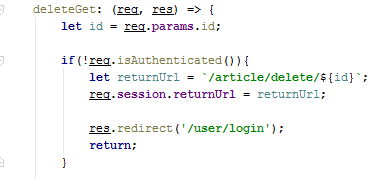
In the end, the article should be **changed**. :)



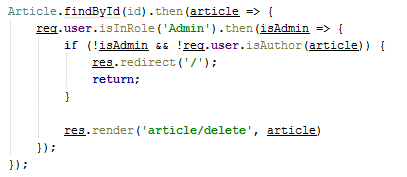
So far so good, we have tested our application and it’ seems okay.

## Validate Article Delete

The validation of the article delete operation is absolutely the same. First we add user **authentication** (in article controller):



Then we add the **authorization**:



**Summary**: With last two steps, we made **validations** on our **Edit**/**Delete**. Everything seems to be working just [fine](https://i.imgflip.com/1e1vha.jpg).

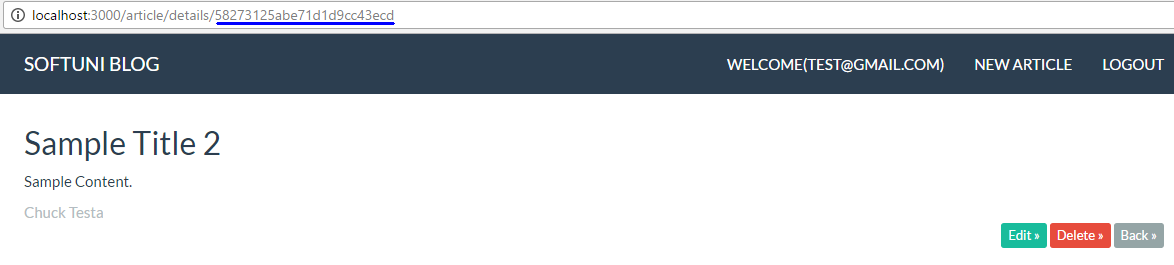
## Article Edit with Postman or RESTClient

Let’s start with an example. Open any REST client you use (**Postman**, **RESTClient** etc.)

We will be showing how we can **edit** an **article** in our server without even being in our website just **using** **Internet** **and** one of those **REST** **API’s** from above. First, if you **don’t** **have** any **articles**, simply **create** one.

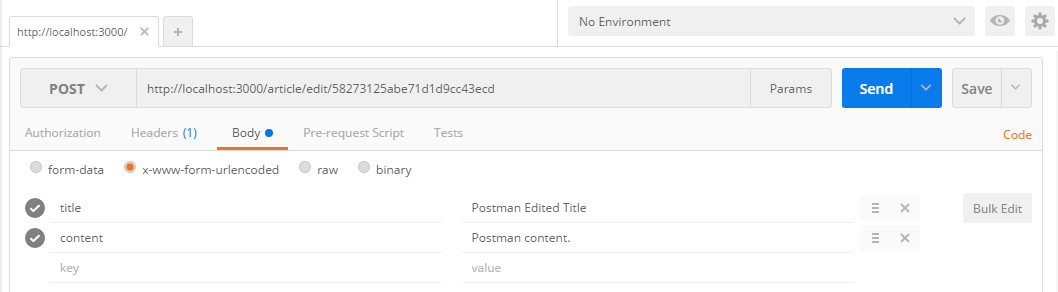
Because for the both example we will need the article’s Id, displayed in our browser.

For the current example the Id is: **58273125abe71d1d9cc43ecd**.



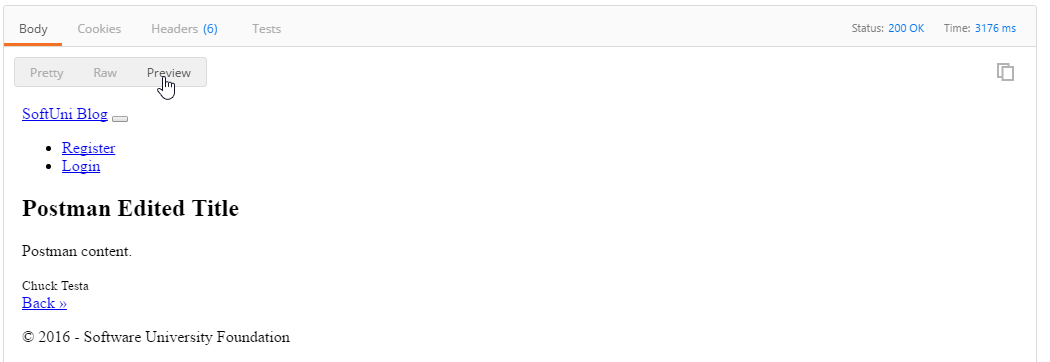
Let’s start with **Postman** (**if** you are **not** using Postman **skip** this **step**):

Few things to do here:



1. Make sure that **HTTP Method** is **POST**, not GET or any other.
2. The inserted url will be: <http://localhost:3000/article/edit/> + your article’s Id – in our case: **58273125abe71d1d9cc43ecd**.
3. Make sure to click on the ”Body” section and choose “**x-www-form-urlencoded**”. Why?
4. Add **two** **key-value pairs**, one for the “**title**” property and one for the “**content**”.
5. Click “Send”.

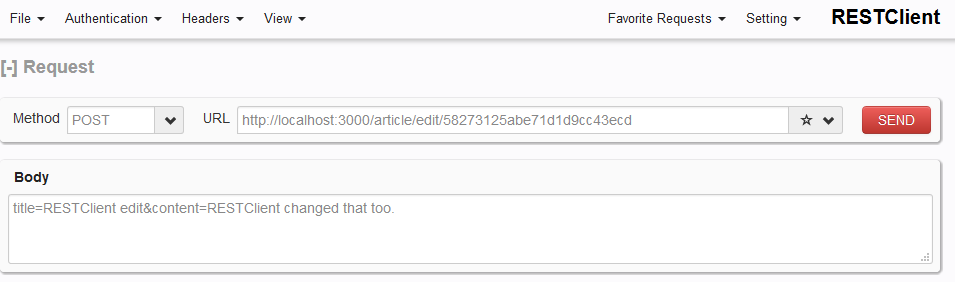
Now, you may receive some response looking something like this (click the “Preview” section):

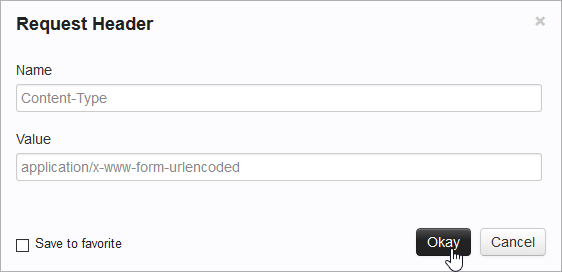


The response is “simple” since the stylesheet file (“style.css”) is not even asked. We have received the html only and is looking like we made some changes to this article. Now go back on your website and see for yourself:



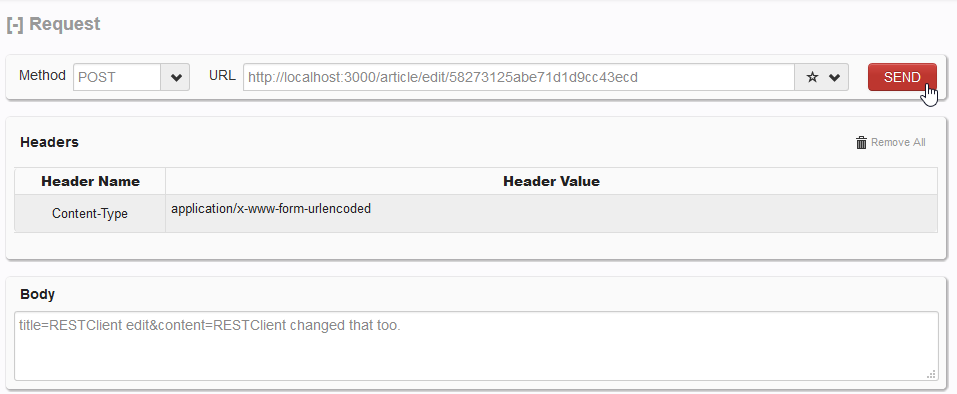
**Or** if you are using **RESTClient** do the following:

1. Make sure that **HTTP Method** is **POST**, not GET or any other.
2. The inserted url will be: <http://localhost:3000/article/edit/> + your article’s Id – in our case: **58273125abe71d1d9cc43ecd**.
3. Add **two** **key-value pairs**, one for the “**title**” property and one for the “**content**”. But this time in a **single** **string**, **separated** by ‘**&**’. The **format** must be: “**key=value&key=value**” and so on.  
   
4. Next click on “Headers” section and then on “Custom headers”:  
   



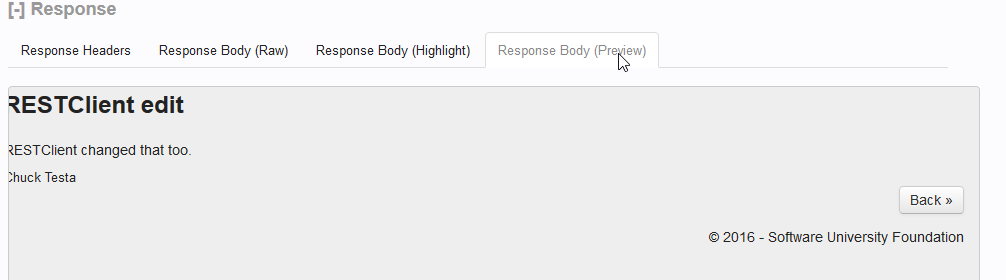
Insert name - “Content-Type”, and value – “application/x-www-form-urlencoded”. Click “Okay”.

The page should look like this:



1. Click “Send”.

The response in “Response Body (**Preview**)” section should look like this (a little bit “broken”, but it’s fine):



Go to article’s details and see if change is made:



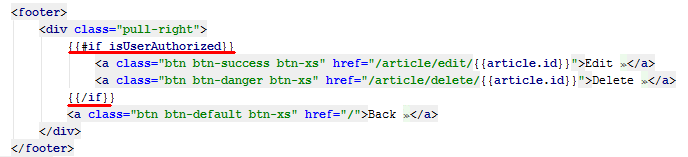
We edited our article content. We might have secured our views from displaying any CRUD options but still not to be secured from any “advanced“ user.

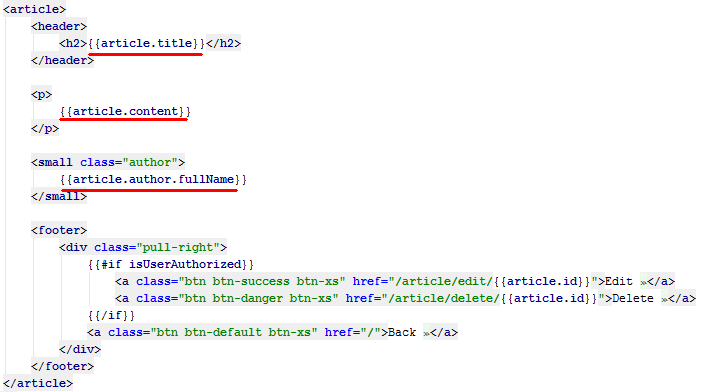
**Summary**: now we know that **authenticating** our **GET** requests **is** a bad option and we saw **how** **easy** is to make a lot of **troubles** if we are **not** making our **validation** **properly**.

**Suggestion**: make **validation** of **all** **POST** requests in **same** manner we did with for the **GET** ones.

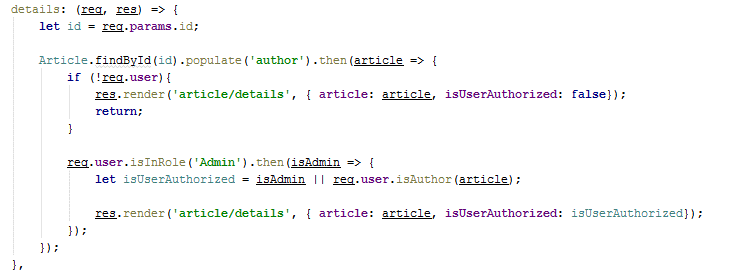
## Create View Validation

We have the tools for validating so far. So let’s do some validations in our views. Our target will be “article/details” view. Since we are using Handlebars we cannot make complex conditions in our if-clauses (using boolean expressions with logical operators as “&” or “||” is not supported). This is why we have to make the validation in our controller and just pass it to the view as a single boolean. In our **article** “**details.hbs”** view:

Separate variable (from article) means that we have to **change** our **object** **references** in “**details.hbs**”:



Now go back to **Article’s** **controller**:



Summary: Here is how the “details“ function should look like. First we check if the user is authenticated, then if it is authorized and we pass the result as a separate variable.