

Blog: Javascript and MongoDB

This document defines a complete walkthrough of creating a **Blog** application with the [Express.js](#) Framework, from setting up the framework through [authentication](#) module, ending up with creating a **CRUD** around [MongoDB](#) entities using [Mongoose](#) object-document model module.

Make sure you have already gone through the [Getting Started: Javascript](#) guide. In this guide we will be using: [WebStorm](#) and [RoboMongo](#) GUI. The rest of the needed non-optional software is described in the guide above.

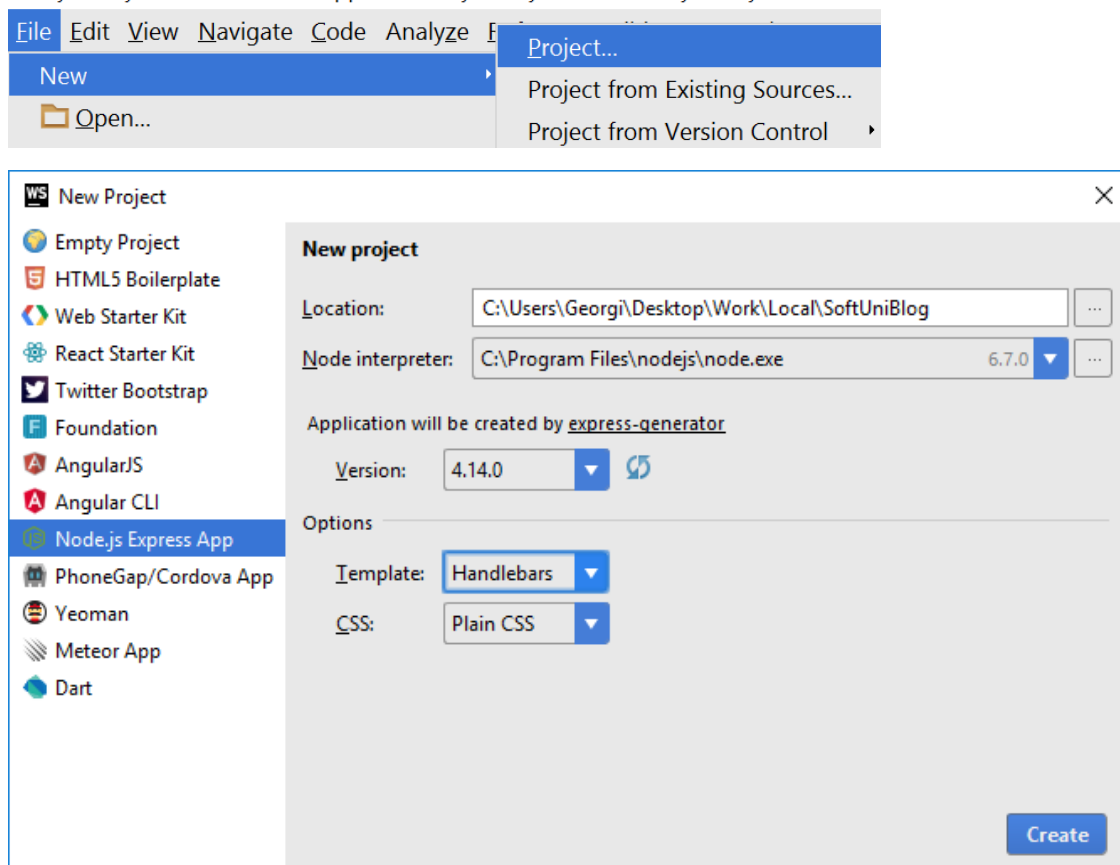
Chapters from I to III are for advanced users, but is recommended to be read. There's a [skeleton](#) which you can use and start from chapter IV.

I. 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

1. 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**)

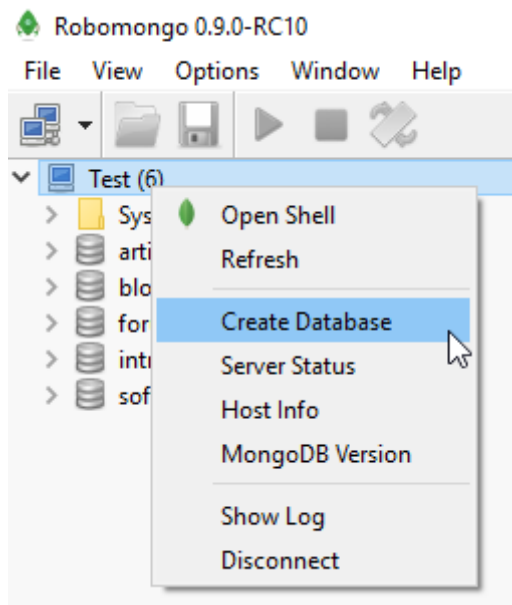


Make sure that you have [Node interpreter](#) installed and the chosen directory is the right one.

- Also choose **Template** to be [Handlebars](#).
- Express recommended **versions** are any above: 4.0.0

2. Create Database

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

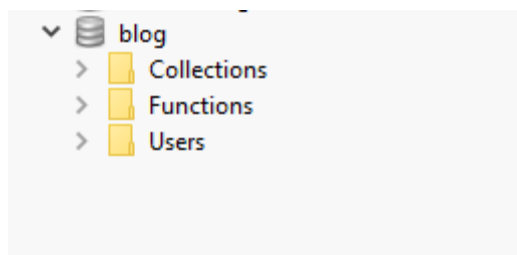
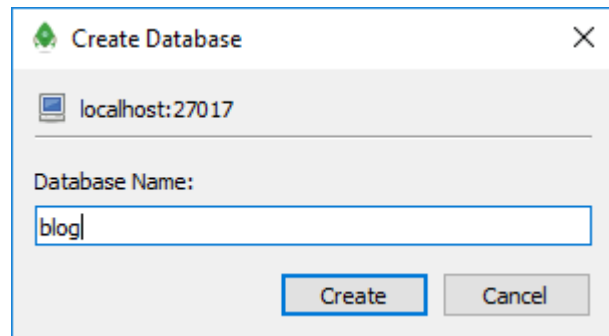


mongo

use blog

```
db.users.insert({"email": "test@gmail.com"})
```

```
db.users.find({})
```



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

```
C:\Users\SoftUniLector\Desktop>mongo
MongoDB shell version: 3.2.10
connecting to: test
> use blog
switched to db blog
> db.users.insert({"email":"test@gmail.com"})
WriteResult({ "nInserted" : 1 })
> db.users.find({})
{ "_id" : ObjectId("581f272958045ba54194deef"), "email" : "test@gmail.com" }
>
```

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 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).

```

C:\Users\Georgi\Desktop>mongod --dbpath D:\MongoDB\data
2016-11-04T19:59:01.079+0200 I CONTROL [initandlisten] MongoDB starting : pid=13328 port=27017 dbpath=D:\MongoDB\data 6
4-bit host=DESKTOP-QOL82B8
2016-11-04T19:59:01.081+0200 I CONTROL [initandlisten] targetMinOS: Windows 7/Windows Server 2008 R2
2016-11-04T19:59:01.082+0200 I CONTROL [initandlisten] db version v3.2.10
2016-11-04T19:59:01.082+0200 I CONTROL [initandlisten] git version: 79d9b3ab5ce20f51c272b4411202710a082d0317
2016-11-04T19:59:01.082+0200 I CONTROL [initandlisten] OpenSSL version: OpenSSL 1.0.1t-fips 3 May 2016
2016-11-04T19:59:01.082+0200 I CONTROL [initandlisten] allocator: tcmalloc
2016-11-04T19:59:01.082+0200 I CONTROL [initandlisten] modules: none
2016-11-04T19:59:01.082+0200 I CONTROL [initandlisten] build environment:
2016-11-04T19:59:01.082+0200 I CONTROL [initandlisten] distmod: 2008plus-ssl
2016-11-04T19:59:01.082+0200 I CONTROL [initandlisten] distarch: x86_64
2016-11-04T19:59:01.082+0200 I CONTROL [initandlisten] target_arch: x86_64
2016-11-04T19:59:01.082+0200 I CONTROL [initandlisten] options: { storage: { dbPath: "D:\MongoDB\data" } }
2016-11-04T19:59:01.083+0200 I - [initandlisten] Detected data files in D:\MongoDB\data created by the 'wiredTig
r' storage engine, so setting the active storage engine to 'wiredTiger'.
2016-11-04T19:59:01.084+0200 I STORAGE [initandlisten] wiredtiger_open config: create,cache_size=4G,session_max=20000,e
viction=(threads_max=4),config_base=false,statistics=(fast),log=(enabled=true,archive=true,path=journal,compressor=snapp
y),file_manager=(close_idle_time=100000),checkpoint=(wait=60,log_size=2GB),statistics_log=(wait=0),
2016-11-04T19:59:02.048+0200 I NETWORK [HostnameCanonicalizationWorker] Starting hostname canonicalization worker
2016-11-04T19:59:02.048+0200 I FTDC [initandlisten] Initializing full-time diagnostic data capture with directory 'D
:\MongoDB\data\diagnostic.data'
2016-11-04T19:59:02.050+0200 I NETWORK [initandlisten] waiting for connections on port 27017

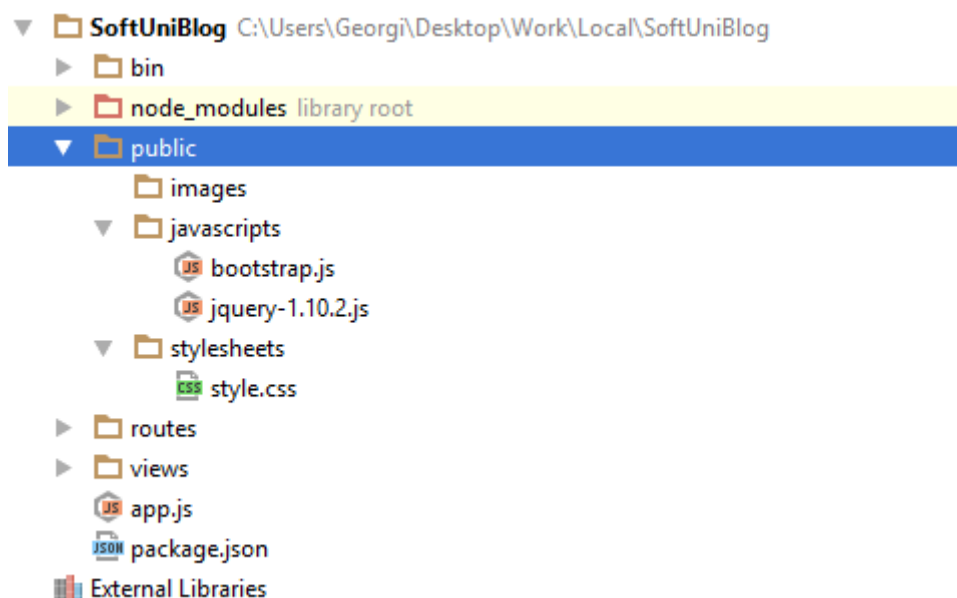
```

3. 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](#), where part of our **JavaScript** and **CSS** is included. In addition, we will need

1. [Bootstrap Date Time picker](#) for choosing dates in our forms
2. [Moment JS](#) 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:

```

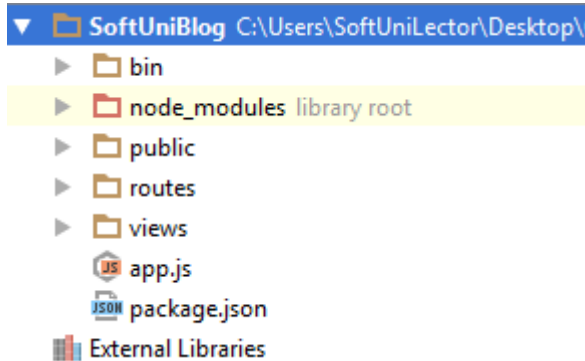
<!DOCTYPE html>
<html>
<head>
<title>SoftUni Blog</title>
<link rel='stylesheet' href='/stylesheets/style.css' />
<script src="/javascripts/jquery-1.10.2.js"></script>
<script src="/javascripts/bootstrap.js"></script>
</head>
<body>
<header>
<div class="navbar navbar-default navbar-fixed-top text-uppercase">
<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>

```

II. 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 are they 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" 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.

III. 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.

1. Creating User Entity

Our users should be stored in the database (**MongoDB**). This means we need **Users** collection. **Collections** are represented as an array [JSON](#) objects. In Mongo these objects are called **Documents**.

Let's define rules for a user:

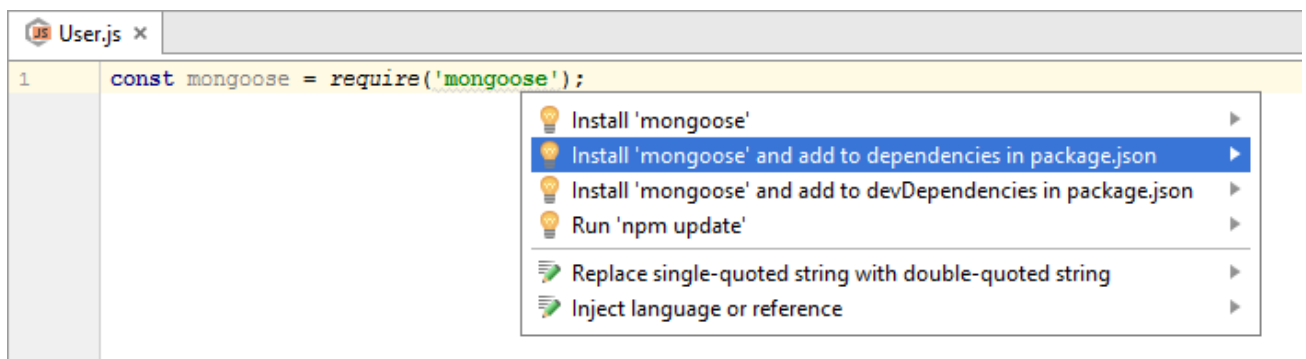
- Should have a unique login name, let's say **email**
- Should have a **passwordHash** (which we will won't save in it's pure view)
- Should have a full name, let's say **fullName**

We won't user pure MongoDB. We will use Mongoose. [Mongoose](#) 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 (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.

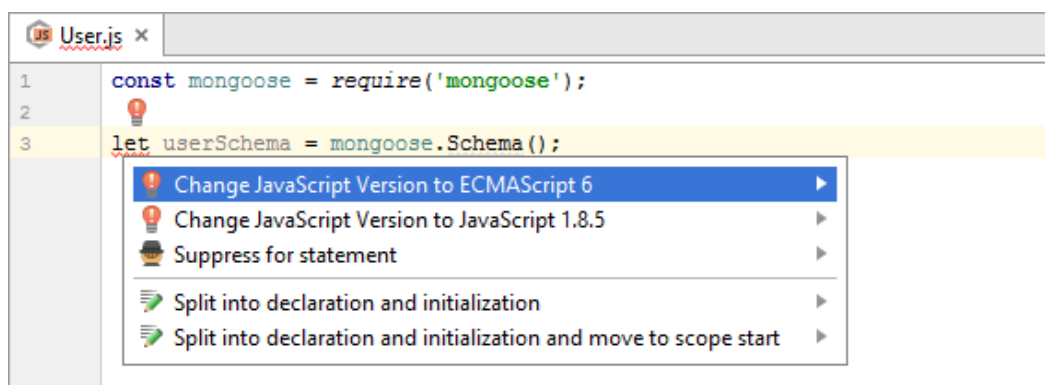


The screenshot shows a code editor with a file named 'User.js'. The first line of code is `const mongoose = require('mongoose');`. A context menu is open over the code, listing several actions: 'Install 'mongoose'', 'Install 'mongoose' and add to dependencies in package.json' (highlighted), 'Install 'mongoose' and add to devDependencies in package.json', 'Run 'npm update'', 'Replace single-quoted string with double-quoted string', and 'Inject language or reference'.

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**" - it's 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.



The screenshot shows the same code editor with the second line of code `let userSchema = mongoose.Schema();`. The 'let' keyword is highlighted in red. A context menu is open over the code, listing several actions: 'Change JavaScript Version to ECMAScript 6' (highlighted), 'Change JavaScript Version to JavaScript 1.8.5', 'Suppress for statement', 'Split into declaration and initialization', and 'Split into declaration and initialization and move to scope start'.

```

let userSchema = mongoose.Schema(
  {
    email: {type: String, required: true, unique: true},
    passwordHash: {type: String, required: true},
    fullName: {type: String, required: true},
    salt: {type: String, required: true}
  }
);

```

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](#). 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](#).

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:

```

const User = mongoose.model('User', userSchema);

module.exports = User;

```

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.

2. 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**).

```

const path = require('path');

module.exports = {
  development: {
    rootFolder: path.normalize(path.join(__dirname, '../')),
    connectionString: 'mongodb://localhost:27017/blog'
  },
  production: {}
};

```

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:


```

const mongoose = require('mongoose');

module.exports = (config) => {
  mongoose.connect(config.connectionString);

  let database = mongoose.connection;
  database.once('open', (error) => {
    if (error) {
      console.log(error);
      return;
    }

    console.log('MongoDB ready!')
  });

  require('./../models/User');
};

```

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

```

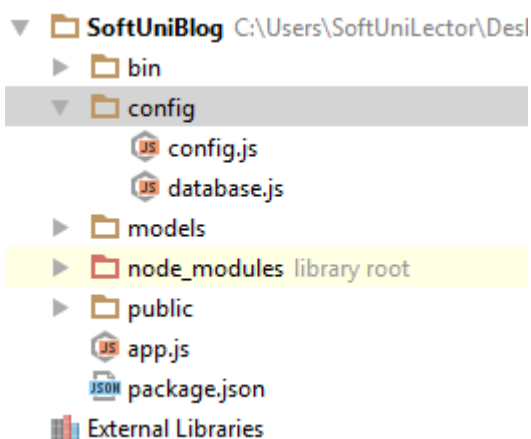
...
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

const config = require('./config/config');

let env = 'development';
require('./config/database')(config[env]);

```

The project structure should something like this:



3. 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** 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.


```
const crypto = require('crypto');
module.exports = {
  generateSalt: () => {

  },
  hashPassword: (password, salt) => {

  }
};
```

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).

```
const crypto = require('crypto');
module.exports = {
  generateSalt: () => {
    let salt = crypto.randomBytes(128).toString('base64');
    return salt;
  },
  hashPassword: (password, salt) => {
    let passwordHash = crypto.createHmac('sha256', salt).update(password).digest('hex');
    return passwordHash;
  }
};
```

Salt will be generated by firstly creating array of 128 random bytes, which are later going to be converted to their base64 presentation. For our hashing logic, it is used the SHA256 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:

```
const express = require('express');
const path = require('path');
const cookieParser = require('cookie-parser');
const bodyParser = require('body-parser');
const session = require('express-session');
const passport = require('passport');

module.exports = (app, config) => {
  // View engine setup.
  app.set('views', path.join(config.rootFolder, '/views'));
  app.set('view engine', 'hbs');

  // This set up which is the parser for the request's data.
  app.use(bodyParser.json());
  app.use(bodyParser.urlencoded({extended: true}));

  // We will use cookies.
  app.use(cookieParser());

  // Session is storage for cookies, which will be de/encrypted with that 'secret' key.
  app.use(session({secret: 's3cr3t5tring', resave: false, saveUninitialized: false}));

  // For user validation we will use passport module.
  app.use(passport.initialize());
  app.use(passport.session());

  // This makes the content in the "public" folder accessible for every user.
  app.use(express.static(path.join(config.rootFolder, 'public')));
};
```

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](#).
- **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 request: **login** and **logout**.

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

```
const passport = require('passport');
const LocalPassport = require('passport-local');
const User = require('../models/User');

const authenticateUser = (username, password, done) => {
  User.findOne({email: username}).then(user => {
    if(!user){
      return done(null, false);
    }

    if (!user.authenticate(password)) {
      return done(null, false);
    }

    return done(null, user);
  });
};
```

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 receives 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**:

```
userSchema.method ({
  authenticate: function (password) {
    let inputPasswordHash = encryption.hashPassword(password, this.salt);
    let isSamePasswordHash = inputPasswordHash === this.passwordHash;
    return isSamePasswordHash;
  }
});

const User = mongoose.model('User', userSchema);
```

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.

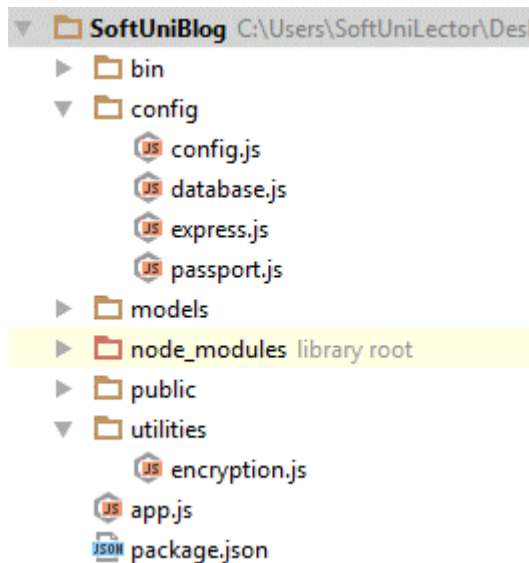
- **serializeUser** – given the whole **user** object **return** the **unique identifier**(in our case pass it to the “done” function).
- **deserializeUser** – given that **unique identifier** from the serialize function **return** the whole **user** object (again passed to “done” function).

```
module.exports = () => {  
  passport.use(new LocalPassport({  
    usernameField: 'email',  
    passwordField: 'password'  
  }, authenticateUser));  
  
  passport.serializeUser((user, done) => {  
    if (!user) {  
      return done(null, false);  
    }  
  
    return done(null, user.id);  
  });  
  
  passport.deserializeUser((id, done) => {  
    User.findById(id).then((user) => {  
      if (!user) {  
        return done(null, false)  
      }  
  
      return done(null, user);  
    })  
  })  
};
```

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:

```
const express = require('express');  
const config = require('./config/config');  
const app = express();  
  
let env = 'development';  
require('./config/database')(config[env]);  
require('./config/express')(app, config[env]);  
require('./config/passport')();  
  
module.exports = app;
```

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



4. 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>
      </fieldset>
    </form>
  </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:

```

module.exports = {
  registerGet: (req, res) => {
    res.render('user/register');
  }
};

```

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](#) 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:

```

const userController = require('../controllers/user');

module.exports = (app) => {
  app.get('/user/register', userController.registerGet);
};

```

Now, require it in our **app.js** file:

```

const express = require('express');
const config = require('./config/config');
const app = express();

let env = 'development';
require('./config/database')(config[env]);
require('./config/express')(app, config[env]);
require('./config/passport')();
require('./config/routes')(app);

module.exports = app;

```

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

SOFTUNI BLOG

REGISTERLOGIN

Register

Email

Email

Full Name

Full Name

Password

Password

Confirm Password

Confirm Password

Cancel

Submit

© 2016 - Software University Foundation

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:

```
<form class="form-horizontal" method="post" action="/user/register">
  <fieldset>
```

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):

```
const userController = require('../controllers/user');

module.exports = (app) => {
  app.get('/user/register', userController.registerGet);

  app.post('/user/register', userController.registerPost)
};
```

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):

```
<input type="text" class="form-control" id="inputEmail" placeholder="Email" name="email" /
iv>
```

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

```
registerPost:(req, res) => {
  let registerArgs = req.body;
```

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:

```
User.findOne({email: registerArgs.email});
```

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:

```
let user = User.findOne({email: registerArgs.email});
```

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:

```
User.findOne({email: registerArgs.email}).then(user => {  
  console.log(user);  
});
```

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

```
let errorMsg = '';  
if (user) {  
  errorMsg = 'User with the same username exists!';  
} else if (registerArgs.password !== registerArgs.repeatedPassword) {  
  errorMsg = 'Passwords do not match!'  
}
```

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](#) 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 (errorMsg) {  
  registerArgs.error = errorMsg;  
  res.render('user/register', registerArgs)  
} else {
```

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:

```
let salt = encryption.generateSalt();
let passwordHash = encryption.hashPassword(registerArgs.password, salt);

let userObject = {
  email: registerArgs.email,
  passwordHash: passwordHash,
  fullName: registerArgs.fullName,
  salt: salt
};

User.create(userObject).then(user => {
  req.login(user, (err) => {
    if (err) {
      registerArgs.error = err.message;
      res.render('user/register', registerArgs);
      return;
    }

    res.redirect('/')
  })
})
```

*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:

```
app.use(passport.session());

app.use((req, res, next) => {
  if(req.user){
    res.locals.user = user;
  }

  next();
});

// This makes the content in the '
```

We have just declared a [middleware](#), which will simply make our current user visible for both the views and the controllers.

5. Login Form

We will create our login functionality 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:

```
loginGet: (req, res) => {
  res.render('user/login');
},
```

After that extend the “routes.js”:

```
app.get('/user/login', userController.loginGet);
app.post('/user/login', userController.loginPost);
```

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:

```
userSchema.method ({
  authenticate: function (password) {
    let inputPasswordHash = encryption.hashPassword(password, this.salt);
    let isSamePasswordHash = inputPasswordHash === this.passwordHash;

    return isSamePasswordHash;
  }
});
```

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:

```
if (!user || !user.authenticate(loginArgs.password)) {  
  let errorMsg = 'Either username or password is invalid!';  
  loginArgs.error = errorMsg;  
  res.render('user/login', loginArgs);  
  return;  
}
```

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.

6. Logout

Logging out is very simple:

```
logout: (req, res) => {  
  req.logout();  
  res.redirect('/');  
}
```

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

```
const userController = require('../controllers/user');  
  
module.exports = (app) => {  
  app.get('/user/register', userController.registerGet);  
  app.post('/user/register', userController.registerPost);  
  
  app.get('/user/login', userController.loginGet);  
  app.post('/user/login', userController.loginPost);  
  
  app.get('/user/logout', userController.logout);  
};
```

IV. Creating Articles

1. 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:

email	passwordHash	fullName	salt
test@test.com	SecretPasswordHash	Chuck Testa	s3cretsalt

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:

```
{
  "_id" : ObjectId("5821a992a9b7a221a830fbf0"),
  "email" : "test@test.com",
  "passwordHash" : "SecretPasswordHash",
  "fullName" : "Chuck Testa",
  "salt" : "s3cretsalt"
}
```

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

Enough talk, let's do some action:

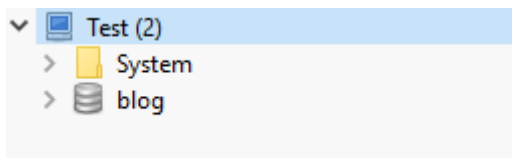
1. Open MongoDB connection – open Command Prompt and type **"mongod --dbpath \"D:\test\example\""**. What this will do is: create server (locally) on some of the computer's port (default is 27017) and will wait for any contact (command).
2. Connect to the newly-created server: depends on whether you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

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

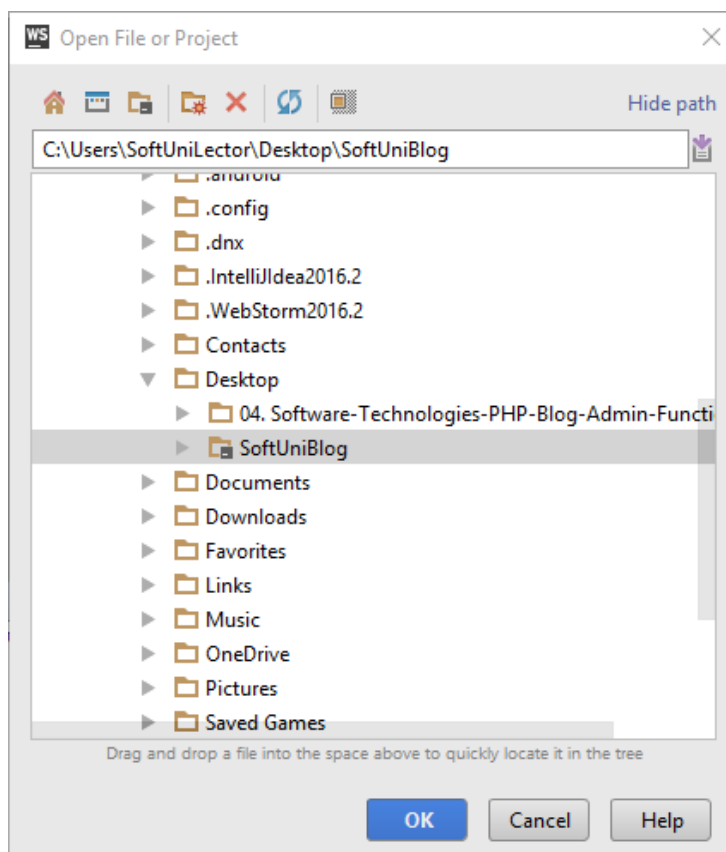
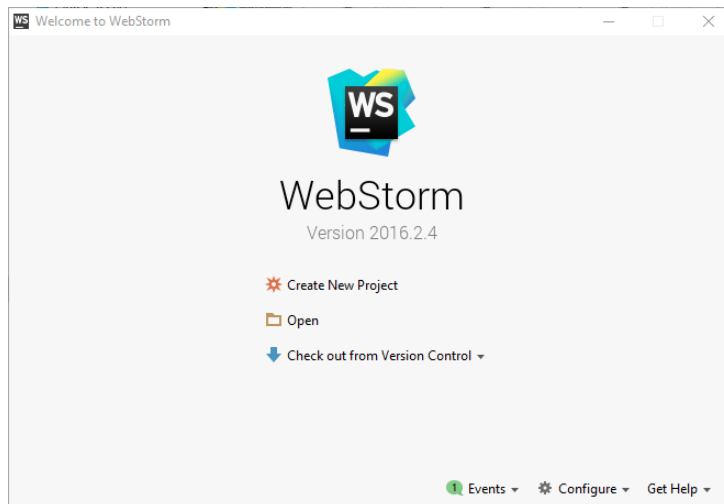
Summary: Now we know simple definition of a database. We saw different ideas behind implementing a database. Also how to start a MongoDB server from which we can create and manipulate different databases.

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

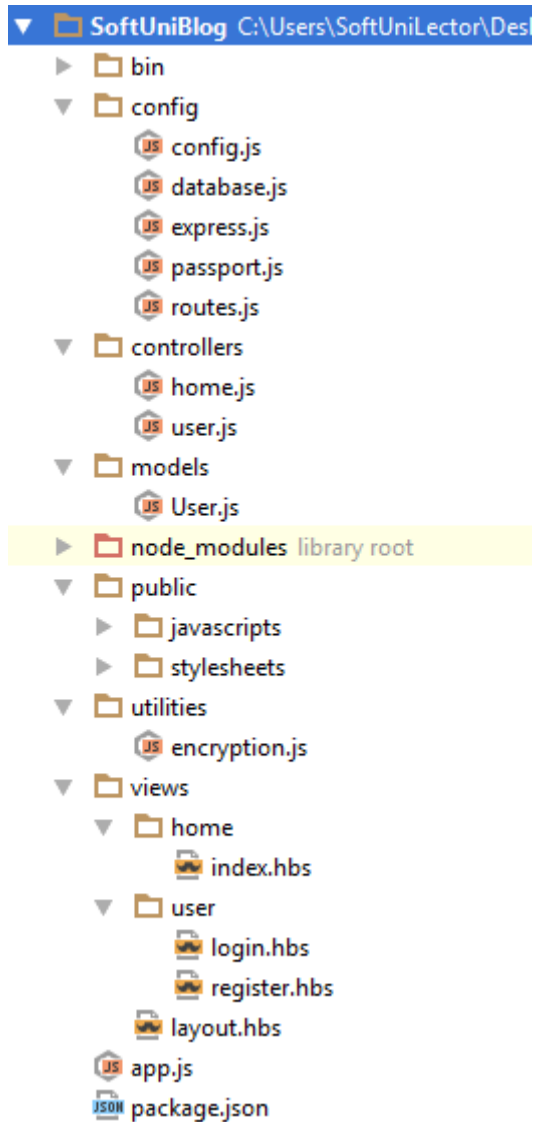


2. Open/Create project

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](#), [Ajax](#) 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](#).

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

3. 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 everytime 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:**

```
const mongoose = require('mongoose');

let articleSchema = mongoose.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...):

```
const mongoose = require('mongoose');

let articleSchema = mongoose.Schema({
  title: {type: String, required: true},
  content: {type: String, required: true},
  author: {type: mongoose.Schema.Types.ObjectId, required: true, ref: 'User'},
  date: {type: Date, default: Date.now()}
});

const Article = mongoose.model('Article', articleSchema);
```

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 functionality 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](#) for more explanation.

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

```
module.exports = Article;
```

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:

```
require('../models/User');
require('../models/Article');
```

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

4. 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**:

```
let userSchema = mongoose.Schema({
  email: {type: String, required: true, unique: true},
  passwordHash: {type: String, required: true},
  fullName: {type: String, required: true},
  articles: {type: [mongoose.Schema.Types.ObjectId], default: []},
  salt: {type: String, required: true}
});
```

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**.

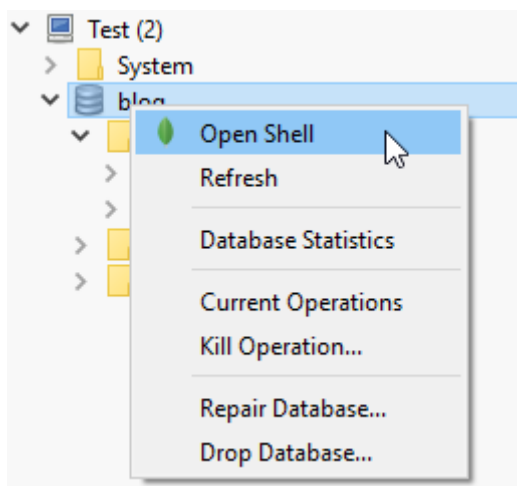
5. Migrations

In MongoDB world, where we don't have tables and columns, relations between models are more loose. The whole [responsibility](#) of handling a migration is given to the programmer. There are some [frameworks](#) 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):

```
Test localhost:27017 blog
db.getCollection('users').update({articles: {$exists: false}}, { $set: {articles: []}}, {multi: true})
0.003 sec.
Updated 0 record(s) in 3ms
```

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](#).

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.

6. 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:

```
const Article = require('mongoose').model('Article');

module.exports = {
  createGet: (req, res) => {
    res.render('article/create');
  },
}
```

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.

7. 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**:

```
{{#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}}
```

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 **falsey** 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()**):

```
} app.use((req, res, next) => {
}   if(req.user){
}     res.locals.user = req.user;
}   }
}   next();
}   };
```

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:

```

<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>

```

Our **form** html tag contains two important attributes: **method** – which defines what the HTTP [method](#) 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.

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. 😊

8. 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:

```

module.exports = {
  createGet: (req, res) => {
    |   res.render('article/create');
  },

  createPost: (req, res) => {
    |   let articleArgs = req.body;
  }
}

```

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

```

let errorMsg = '';

if(!req.isAuthenticated()) {
  errorMsg = 'You should be logged in to make articles!'
} else if (!articleArgs.title){
  errorMsg = 'Invalid title!';
} else if (!articleArgs.content){
  errorMsg = 'Invalid content!';
}

```

`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 (errorMsg) {
  res.render('article/create', {error: errorMsg});
  return;
}

articleArgs.author = req.user.id;
Article.create(articleArgs).then(article => {
  req.user.articles.push(article.id);
  req.user.save(err => {
    if (err){
      res.redirect('/', {error: err.message});
    }else {
      res.redirect('/');
    }
  });
});

```

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** :

```

require('../models/User');
require('../models/Article');

```

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

```
...
const homeController = require('../controllers/home');

module.exports = (app) => {
  app.get('/', homeController.index);

  app.get('/user/register', userController.registerGet);
  app.post('/user/register', userController.registerPost);
}
```

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**.

V. Read, Update and Delete Articles

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

1. 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:

```
<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>
```

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 crucial point

in getting the whole information (from our relation). Let's see how we will get that information from the "home" controller:

```
const mongoose = require('mongoose');
const Article = mongoose.model('Article');

module.exports = {
  index: (req, res) => {
    Article.find({}).limit(6).populate('author').then(articles => {
      res.render('home/index', {articles: articles});
    })
  }
};
```

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 additional 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:

Some title

Some context

Chuck Testa

Read more »

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.

2. 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:

```

details: (req, res) => {
  let id = req.params.id;

  Article.findById(id).populate('author').then(article => {
    res.render('article/details', article)
  });
}

```

Whenever we want to 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":

```

app.get('/article/create', articleController.createGet);
app.post('/article/create', articleController.createPost);

app.get('/article/details/:id', articleController.details);
};

```

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 its 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.

3. 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:

```
editGet: (req, res) => {
  let id = req.params.id;

  Article.findById(id).then(article => {
    res.render('article/edit', article)
  });
},
```

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:

```
app.get('/article/edit/:id', articleController.editGet);
```

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):

Sample Title

Sample Content.

Chuck Testa

Read more »

We will be rerouted to it’s details page:

Sample Title

Sample Content.

Chuck Testa

Edit » Delete » Back »

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

Edit Article

Article Title

Sample Title

Content

Sample Content.

Cancel

Edit

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:

```
app.post('/article/edit/:id', articleController.editPost);
```

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

```
editPost: (req, res) => {
  let id = req.params.id;

  let articleArgs = req.body;

  let errorMsg = '';
  if (!articleArgs.title) {
    errorMsg = 'Article title cannot be empty!';
  } else if (!articleArgs.content) {
    errorMsg = 'Article content cannot be empty!';
  }

  if (errorMsg) {
    res.render('article/edit', {error: errorMsg})
  } else {
```

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

```
} else {  
  Article.update({_id: id}, {$set: {title: articleArgs.title, content: articleArgs.content}})  
    .then(updateStatus => {  
      res.redirect(`/article/details/${id}`);  
    })  
}
```

What this code will do is: find an article with specified **id** (the article we want to edit) and we will change it's 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 it's details page. And **volia**, we are done with editing our article, so let's test it:

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

Sample Title

Sample Content.

Chuck Testa

Read more »

Then click on the edit button:

Sample Title

Sample Content.

Chuck Testa

Edit » Delete » Back »

Somehow change your article's content or title:

Edit Article

Article Title

Sample Title Edited.

Content

Sample Content.

Cancel

Edit

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

Sample Title Edited.

Sample Content.

Chuck Testa

Edit Delete Back

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

4. Delete Articles

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

```
deleteGet: (req, res) => {
  let id = req.params.id;

  Article.findById(id).then(article => {
    res.render('article/delete', article)
  });
},
```

view – route – controller.

As you may 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:

```
app.get('/article/delete/:id', articleController.deleteGet);
```

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

Sample Title Edited.

Sample Content.

Chuck Testa

Read more »

Sample Title Edited.

Sample Content.

Chuck Testa

Edit » Delete » Back »

Delete Article

Article Title

Sample Title Edited.

Content

Sample Content.

Cancel

Delete

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:

```
app.post('/article/delete/:id', articleController.deletePost);
```

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

```
deletePost: (req, res) => {  
  let id = req.params.id;  
  Article.findOneAndRemove({_id: id}).populate('author').then(article => {  
    let author = article.author;
```

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**:

```
let author = article.author;

// Index of the article's ID in the author's articles.
let index = author.articles.indexOf(article.id);

if(index < 0) {
  let errorMsg = 'Article was not found for that author!';
  res.render('article/delete', {error: errorMsg})
} else {
  // Remove count elements after given index (inclusive).
  let count = 1;
  author.articles.splice(index, count);
  author.save().then((user) => {
    res.redirect('/');
  });
}
```

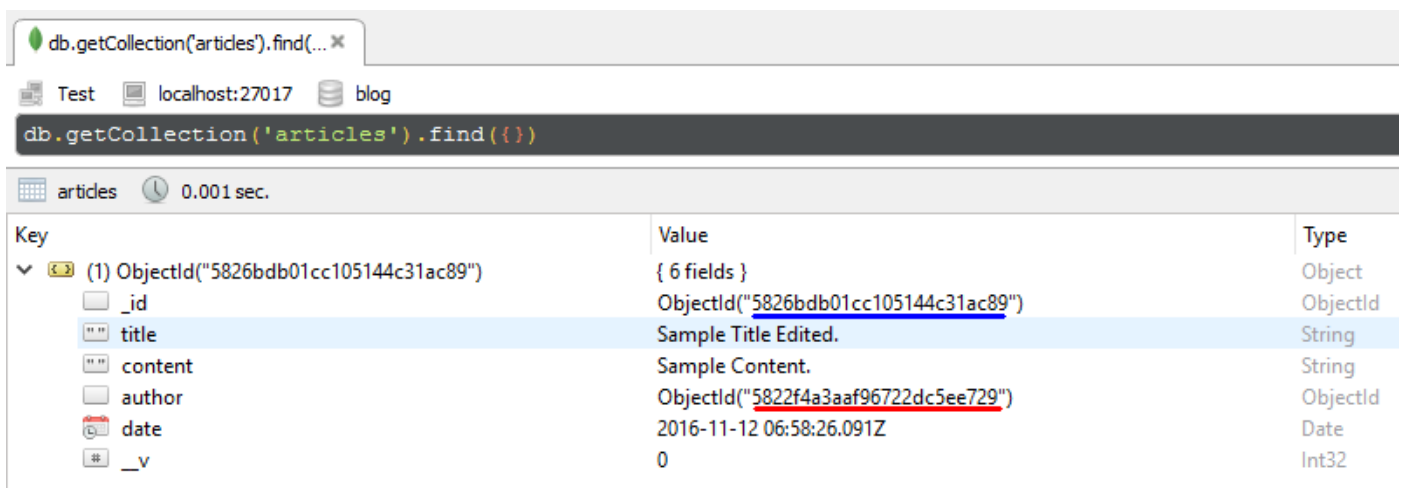
Here are the specifics of how to delete a specific element in array. For more information visit this [thread](#). But the logic is simple: **first** find the **index of** the specified **element** and **then remove** the value **from** the resulted **index** (if index is valid). The **remove** is done 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:



db.getCollection('articles').find(...)		
Test	localhost:27017	blog
db.getCollection('articles').find({})		
articles 0.001 sec.		
Key	Value	Type
(1) ObjectId("5826bdb01cc105144c31ac89")	{ 6 fields }	Object
_id	ObjectId("5826bdb01cc105144c31ac89")	ObjectId
title	Sample Title Edited.	String
content	Sample Content.	String
author	ObjectId("5822f4a3aaf96722dc5ee729")	ObjectId
date	2016-11-12 06:58:26.091Z	Date
_v	0	Int32

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**):

▼ (6) ObjectId("5822f4a3aaf96722dc5ee729")	{ 8 fields }	Object
_id	ObjectId("5822f4a3aaf96722dc5ee729")	ObjectId
email	test@gmail.com	String
passwordHash	38941883bb0f1181850efdbc60dba33f020f4c26293c41b9f5b3c7b72d...	String
fullName	Chuck Testa	String
salt	Kayohfd65jFWvsldUpghuOuWQRiG0kF4ea6lmM3ZW+qW8AMXkS...	String
▼ articles	[1 element]	Array
[0]	ObjectId("5826bdb01cc105144c31ac89")	ObjectId
_v	4	Int32
> roles	[1 element]	Array

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:

Sample Title Edited.

Sample Content.

Chuck Testa

Read more »

Then click the “**Delete**” button:

Sample Title Edited.

Sample Content.

Chuck Testa

Edit » Delete » Back »

“Delete” button one more time:

Delete Article

Article Title

Sample Title Edited.

Content

Sample Content.

Cancel

Delete

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:

```

db.getCollection('articles').find({})
db.getCollection('users').find({})

Test localhost:27017 blog

db.getCollection('articles').find({})

0.001 sec.

Fetched 0 record(s) in 1ms

```

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

We are one way to go, let’s see if the author’s is probably updated as well:

▼ (6) ObjectId("5822f4a3aaf96722dc5ee729")	{ 8 fields }	Object
_id	ObjectId("5822f4a3aaf96722dc5ee729")	ObjectId
email	test@gmail.com	String
passwordHash	38941883bb0f1181850efdbc60dba33f020f4c26293c41b9f5b3c7b72d...	String
fullName	Chuck Testa	String
salt	Kayohfd65jFWvsldUpghuOuWQRiG0kF4ea6lmM3ZW+qW8AMXkS...	String
▼ articles	[0 elements]	Array
# _v	5	Int32
> roles	[1 element]	Array

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 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.

VI. 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 on 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**.

1.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”:

```
const mongoose = require('mongoose');

let roleSchema = mongoose.Schema({
  name: {type: String, required: true, unique: true},
  users: [{type: mongoose.Schema.Types.ObjectId, ref: 'User'}]
});

const Role = mongoose.model('Role', roleSchema);

module.exports = Role;
```

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

2.Create User – Role relation

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

```
let userSchema = mongoose.Schema({
  {
    email: {type: String, required: true, unique: true},
    passwordHash: {type: String, required: true},
    fullName: {type: String, required: true},
    articles: {type: [mongoose.Schema.Types.ObjectId], default: []},
    salt: {type: String, required: true}
  }
});
```

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:

```
let userSchema = mongoose.Schema({
  {
    email: {type: String, required: true, unique: true},
    passwordHash: {type: String, required: true},
    fullName: {type: String, required: true},
    articles: [{type: mongoose.Schema.Types.ObjectId, ref: 'Article'}],
    salt: {type: String, required: true},
  }
});
```

Then add the “roles” property:

```
let userSchema = mongoose.Schema({
  {
    email: {type: String, required: true, unique: true},
    passwordHash: {type: String, required: true},
    fullName: {type: String, required: true},
    articles: [{type: mongoose.Schema.Types.ObjectId, ref: 'Article'}],
    roles: [{type: mongoose.Schema.Types.ObjectId, ref: 'Role'}],
    salt: {type: String, required: true},
  }
});
```

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.

3. 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**.

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

```
module.exports.initialize = () => {  
  Role.findOne({name: 'User'}).then(role => {  
    if(!role){  
      Role.create({name: 'User'});  
    }  
  });  
  
  Role.findOne({name: 'Admin'}).then(role => {  
    if(!role){  
      Role.create({name: 'Admin'});  
    }  
  });  
};
```

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:

```
...  
require('../models/Role').initialize();  
require('../models/User');
```

4. 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:

```
const user = require('mongoose').model('user');  
const Role = require('mongoose').model('Role');  
const encryption = require('../utilities/encryption');
```

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

```

let roles = [];
Role.findOne({name: 'User'}).then(role => {
  roles.push(role.id);

  userObject.roles = roles;
  User.create(userObject).then(user => {
    role.users.push(user);
    role.save(err => {
      if(err) {
        registerArgs.error = err.message;
        res.render('user/register', registerArgs);
      }
      else {
        req.login(user, (err) => {
          if (err) {
            registerArgs.error = err.message;
            res.render('user/register', registerArgs);
            return;
          }

          res.redirect('/');
        })
      }
    })
  });
});
});

```

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**.

5. Seed Admin

With now having an "Admin" role, would be nice to seed an admin user? Just for test uses, yeah, you [know](#).. Let's **create** a **"seedAdmin()"** function in our **"User.js"** model:

On top of the file add the **"Role"** model:

```

const user = require('mongoose').model('User');
const Role = require('mongoose').model('Role');
const encryption = require('../utilities/encryption')

```

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

```
module.exports.seedAdmin = () => {  
  let email = 'admin@softuni.bg';  
  User.findOne({email: email}).then(admin => {  
    if (!admin) {
```

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):

```
if (!admin) {  
  Role.findOne({name: 'Admin'}).then(role => {  
    let salt = encryption.generateSalt();  
    let passwordHash = encryption.hashPassword('admin', salt);  
  
    let roles = [];  
    roles.push(role.id);  
  
    let user = {  
      email: email,  
      passwordHash: passwordHash,  
      fullName: 'Admin',  
      articles: [],  
      salt: salt,  
      roles: roles  
    };  
  
    User.create(user).then(user => {  
      role.users.push(user.id);  
      role.save(err => {  
        if (err) {  
          console.log(err.message);  
        } else {  
          console.log('Admin seeded successfully!')  
        }  
      });  
    });  
  });  
}
```

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

```
require(' ../models/Role ').initialize(),  
require(' ../models/User ').seedAdmin();  
require(' ../models/Article ');
```

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**!

```
MongoDB ready!  
Admin seeded successfully!
```

VII. 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.

I. 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**:

```
userSchema.method ({
  authenticate: function (password) {
    let inputPasswordHash = encryption.hashPassword(password, this.salt
    let isSamePasswordHash = inputPasswordHash === this.passwordHash;

    return isSamePasswordHash;
  },

  isAuthor: function (article) {
```

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

```
isAuthor: function (article) {
  if(!article){
    return false;
  }

  let isAuthor = article.author.equals(this.id);

  return isAuthor;
},
```

Simple keeps our code of repetition.

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

```
isInRole: function (roleName) {
  return Role.findOne({name: roleName}).then(role => {
    if (!role){
      return false;
    }

    let isInRole = this.roles.indexOf(role.id) !== -1;
    return isInRole;
  })
}
```

However, there is something tricky here. Since **we have** to **check** if the **role** is given to use exist **in database** and get it’s 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**”.

II. 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:

```
editGet: (req, res) => {
  let id = req.params.id;

  if(!req.isAuthenticated()){
    let returnUrl = `/article/edit/${id}`;
    req.session.returnUrl = returnUrl;

    res.redirect('/user/login');
    return;
  }
}
```

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**:

```
req.login(user, (err) => {
  if (err) {
    console.log(err);
    res.redirect('/user/login', {error: err.message});
    return;
  }

  let returnUrl = '/';
  if(req.session.returnUrl) {
    returnUrl = req.session.returnUrl;
    delete req.session.returnUrl;
  }

  res.redirect(returnUrl);
})
```

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**:

```
Article.findById(id).then(article => {
  req.user.isInRole('Admin').then(isAdmin => {
    if (!isAdmin && !req.user.isAuthor(article)) {
      res.redirect('/');
      return;
    }

    res.render('article/edit', article)
  });
});
```

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:

SOFTUNI BLOG

REGISTERLOGIN

Sample Title

Sample Content.

Chuck Testa

Edit »Delete »Back »

The following window should be login form:

Login

Email

Email

Password

Password

CancelLogin

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

SOFTUNI BLOG

REGISTERLOGIN

Sample Title

Sample Content.

Chuck Testa

Edit »Delete »Back »

Log in:

Login

Email

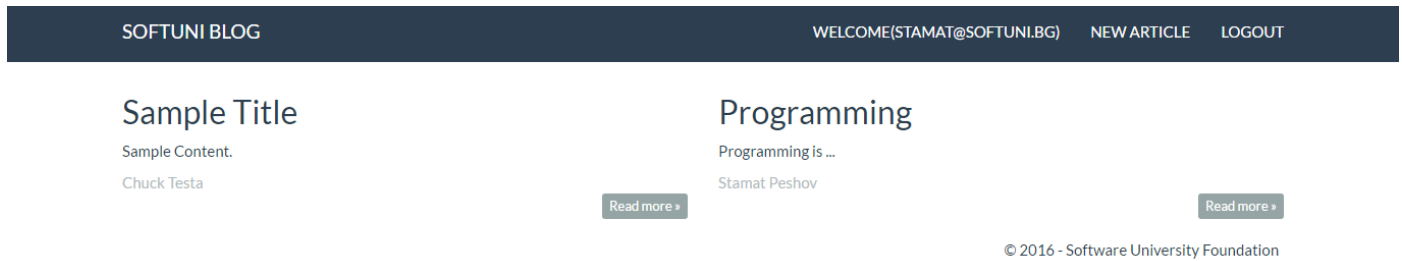
Email

Password

Password

CancelLogin

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



3. 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**. :)

Javascript

Javascript is awesome!

Stamat Peshov

[Edit »](#) [Delete »](#) [Back »](#)

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

III. Validate Article Delete

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

```

} deleteGet: (req, res) => {
    let id = req.params.id;

    if(!req.isAuthenticated()){
        let returnUrl = `/article/delete/${id}`;
        req.session.returnUrl = returnUrl;

        res.redirect('/user/login');
        return;
    }
}

```

Then we add the **authorization**:

```

Article.findById(id).then(article => {
    req.user.isInRole('Admin').then(isAdmin => {
        if (!isAdmin && !req.user.isAuthor(article)) {
            res.redirect('/');
            return;
        }

        res.render('article/delete', article)
    });
});

```

Summary: With last two steps we made **validations** on our **Edit/Delete**. Everything seems to be working just [fine](#).

IV. 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**.

localhost:3000/article/details/58273125abe71d1d9cc43ecd

SOFTUNI BLOG

WELCOME(TEST@GMAIL.COM)

NEW ARTICLE

LOGOUT

Sample Title 2

Sample Content.

Chuck Testa

Edit » Delete » Back »

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

Few things to do here:

The screenshot shows the Postman interface with a POST request configured. The URL is `http://localhost:3000/article/edit/58273125abe71d1d9cc43ecd`. The 'Body' tab is selected, and the 'x-www-form-urlencoded' format is chosen. Two key-value pairs are added: 'title' with value 'Postman Edited Title' and 'content' with value 'Postman content.'.

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 screenshot shows the Postman response preview. The status is 200 OK and the time is 3176 ms. The response is rendered in HTML, showing the 'SoftUni Blog' header, navigation links for 'Register' and 'Login', the title 'Postman Edited Title', the content 'Postman content.', the author 'Chuck Testa', a 'Back »' link, and the footer '© 2016 - Software University Foundation'.

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:

Postman Edited Title


Postman content.

Chuck Testa

Edit » Delete » Back »

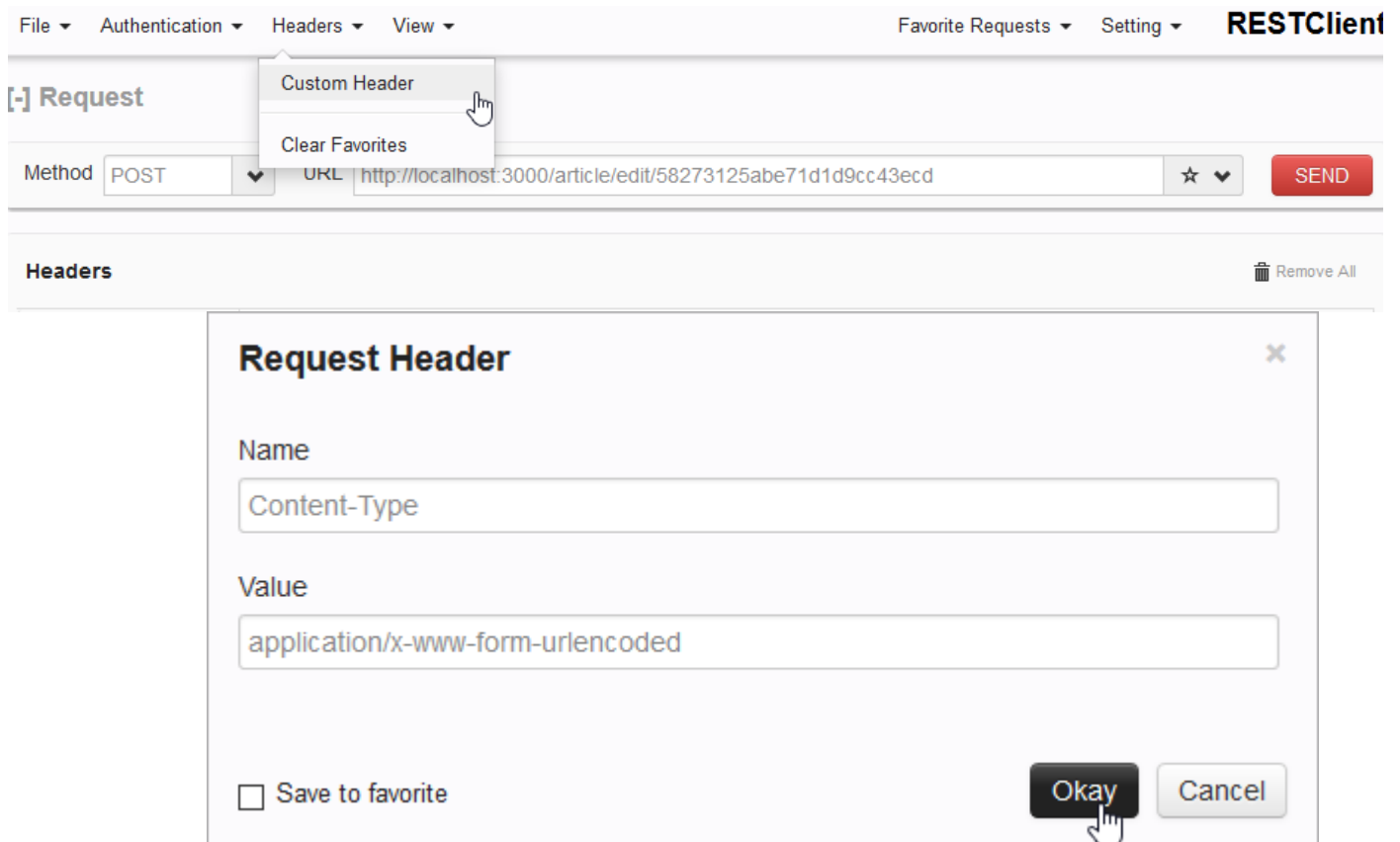
Or if you are using **RESTClient** do the following:

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.



The screenshot shows the RESTClient application window. At the top, there are menu items: File, Authentication, Headers, View, Favorite Requests, Setting, and the RESTClient logo. Below the menu is a section titled "[-] Request". It contains a "Method" dropdown set to "POST" and a "URL" text field containing "http://localhost:3000/article/edit/58273125abe71d1d9cc43ecd". To the right of the URL is a star icon and a "SEND" button. Below the request section is a "Body" text area containing the text "title=RESTClient edit&content=RESTClient changed that too."

4. Next click on “Headers” section and then on “Custom headers”:



The screenshot shows the RESTClient application window with the "Headers" section selected. A dropdown menu is open, showing "Custom Header" and "Clear Favorites". Below the menu, the "Method" is "POST" and the "URL" is "http://localhost:3000/article/edit/58273125abe71d1d9cc43ecd". A "SEND" button is visible. Below the request section, the "Headers" section is active, showing a "Request Header" dialog box. The dialog has a "Name" field with "Content-Type" and a "Value" field with "application/x-www-form-urlencoded". There is a "Save to favorite" checkbox and "Okay" and "Cancel" buttons. A hand cursor is pointing at the "Okay" button.

Insert name - “Content-Type”, and value – “application/x-www-form-urlencoded”. Click “Okey”.
The page should look like this:

[-] Request

Method POST
URL http://localhost:3000/article/edit/58273125abe71d1d9cc43ecd
★
SEND

Headers

Remove All

Header Name	Header Value
Content-Type	application/x-www-form-urlencoded

Body

title=RESTClient edit&content=RESTClient changed that too.

5. Click “Send”.

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

[-] Response

Response Headers
Response Body (Raw)
Response Body (Highlight)
Response Body (Preview)

RESTClient edit

RESTClient changed that too.
Chuck Testa

Back »

© 2016 - Software University Foundation

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

RESTClient edit

RESTClient changed that too.

Chuck Testa

Edit »
Delete »
Back »

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.

V. 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:


```

<footer>
  <div class="pull-right">
    {{#if isUserAuthorized}}
      <a class="btn btn-success btn-xs" href="/article/edit/{{article.id}}">Edit </a>
      <a class="btn btn-danger btn-xs" href="/article/delete/{{article.id}}">Delete </a>
    {{/if}}
    <a class="btn btn-default btn-xs" href="/">Back </a>
  </div>
</footer>

```

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

```

<article>
  <header>
    <h2>{{article.title}}</h2>
  </header>

  <p>
    {{article.content}}
  </p>

  <small class="author">
    {{article.author.fullName}}
  </small>

  <footer>
    <div class="pull-right">
      {{#if isUserAuthorized}}
        <a class="btn btn-success btn-xs" href="/article/edit/{{article.id}}">Edit </a>
        <a class="btn btn-danger btn-xs" href="/article/delete/{{article.id}}">Delete </a>
      {{/if}}
      <a class="btn btn-default btn-xs" href="/">Back </a>
    </div>
  </footer>
</article>

```

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

```

details: (req, res) => {
  let id = req.params.id;

  Article.findById(id).populate('author').then(article => {
    if (!req.user) {
      res.render('article/details', { article: article, isUserAuthorized: false });
      return;
    }

    req.user.isInRole('Admin').then(isAdmin => {
      let isUserAuthorized = isAdmin || req.user.isAuthor(article);

      res.render('article/details', { article: article, isUserAuthorized: isUserAuthorized });
    });
  });
},

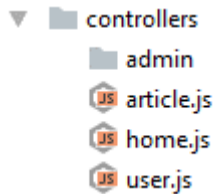
```

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.

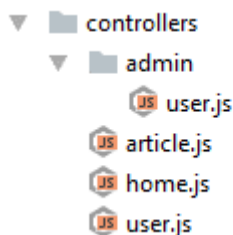
VIII. Creating the Admin Panel

1. Listing All Users

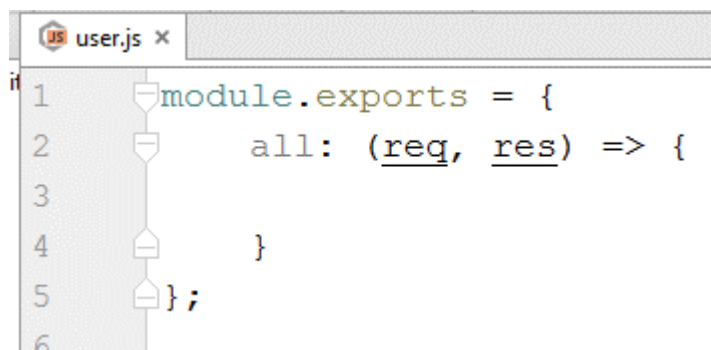
The first thing we are going to create is listing our users. Let's start by **creating folder named "admin"** in the **Controllers** folder:



Create another **"user"** controller in the **Admin** folder. You should receive something like this:



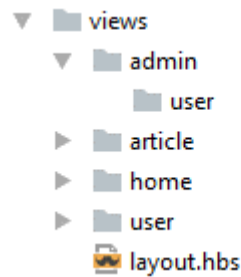
We will use this controller **to manipulate our users**. Let's create our user listing function. It will take all users and list them in a table:



What we have to do is to take **all users** from database. Mark them as admins in order to pass that information to the view (so the view can highlight them on display):

```
const User = require('mongoose').model('User');  
  
module.exports = {  
  all: (req, res) => {  
    User.find({}).then(users => {  
  
      for (let user of users) {  
        user.isInRole('Admin').then(isAdmin => {  
          user.isAdmin = isAdmin;  
        });  
      }  
  
      res.render('admin/user/all', {users: users})  
    });  
  }  
};
```

Let's create our view. But first create "admin" folder and another "user" folder in our views:



There create file "all.hbs" for the view. Our view will be a **table**. It will show the **admins in blue** color, and the **normal users in gray**:

```

<div class="container body-content">
  <div class="well">
    <h2>All Users</h2>
    <div class="row">
      <table class="table table-striped table-hover ">
        <thead>
          <tr>
            <th>#</th>
            <th>Full Name</th>
            <th>Email</th>
            <th>Actions</th>
          </tr>
        </thead>
        <tbody>
          {{#each users}}
            <tr {{#if this.isAdmin }} class="info" {{/if}}>
              <td>{{ this.id }}</td>
              <td>{{ this.fullName }}</td>
              <td>{{ this.email }}</td>
              <td>
                <a
href="/admin/user/edit/{{this.id}}">Edit</a>
                <a
href="/admin/user/delete/{{this.id}}">Delete</a>
              </td>
            </tr>
          {{/each}}
        </tbody>
      </table>
    </div>
  </div>
</div>
  
```

Let's create an **admin button** in the **base layout**. Find the part:

```

<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>
  
```

Add the following if statement above it:

```
{{#if isAdmin}}
  <li class="dropdown">
    <a href="#" class="dropdown-toggle" data-toggle="dropdown"
role="button" aria-expanded="false">Admin<span
class="caret"></span></a>
    <ul class="dropdown-menu" role="menu">
      <li><a href="/admin/user/all">Users</a></li>
      <li><a href="/admin/category/all">Categories</a></li>
    </ul>
  </li>
{{/if}}
```

Here, how should it look like:

```
{{#if user}}
  <div class="navbar-collapse collapse">
    <ul class="nav navbar-nav navbar-right">
      {{#if isAdmin}}
        <li class="dropdown">
          <a href="#" class="dropdown-toggle" data-toggle="dropdown"
role="button" aria-expanded="false">Admin<span class="caret"></span></a>
          <ul class="dropdown-menu" role="menu">
            <li><a href="/admin/user/all">Users</a></li>
            <li><a href="/admin/category/all">Categories</a></li>
          </ul>
        </li>
      {{/if}}
      <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}}
```

There are few things to setup now, **routing** and **authorization**. Let's start with **authorization** – go to “express.js” in “res.locals” middleware. There we will set “isAdmin” variable public to the view so we can be able to see the **admin dropdown**:

```
app.use((req, res, next) => {
  if(req.user) {
    res.locals.user = req.user;
    req.user.isInRole('Admin').then(isAdmin => {
      res.locals.isAdmin = isAdmin;
      next();
    })
  } else {
    next();
  }
});
```

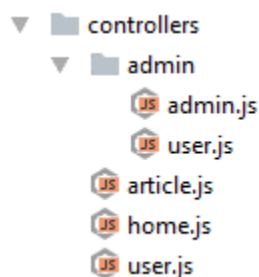
Go to routes.js and add an **authorization middleware** on the **bottom**:

```
app.get('/article/delete/:id', articleController.deleteGet);
app.post('/article/delete/:id', articleController.deletePost);

app.use((req, res, next) => {
  if (req.isAuthenticated()) {
    req.user.isInRole('Admin').then(isAdmin=>{
      if(isAdmin){
        next();
      } else{
        res.redirect('/');
      }
    })
  } else {
    res.redirect('/user/login');
  }
});
```

What this will do is make every **next routing below** it to be accessible only to **Admin**.

We are done with authorization for our admin controller so far, wait did we wrote **admin controller**? We don't have one – so go and create it in our **admin folder**:



Then insert the following code:

```
const userController = require('./user');

module.exports = {
  user: userController,
};
```

Go to the “**route.js**” and require it (on top of the file):

```
const homeController = require('../controllers/home'),
const adminController = require('../controllers/admin/admin');
```

Then **after** the **authorization** middleware the use the **"admin"** controller.

```
app.use((req, res, next) => {
  if (req.isAuthenticated()) {
    req.user.isInRole('Admin').then(isAdmin=>{
      if(isAdmin) {
        next();
      } else {
        res.redirect('/');
      }
    })
  } else {
    res.redirect('/user/login');
  }
});
```

```
app.get('/admin/user/all', adminController.user.all);
```

So far so good, let see what we have done.

Now let's go to our home page and log in as admin - you can see the **admin drop-down menu**. Go to the user listing page:

SOFTUNI BLOG

ADMIN ▾

WELCOME(ADMIN@SOFTUNI.BG)

NEW ARTICLE

LOGOUT

USERS

All Users

#	Full Name	Email	Actions
582b1c544d1f98190811d0a8	Admin	admin@softuni.bg	Edit Delete
582b3855ae05e3117066b639	Georgi Stoimenov	g.stoimenov@softuni.bg	Edit Delete

We can see that I have two users: 1 admin and 1 normal. Let's create the **edit** and **delete functionality**.

2. Editing User

First, we want to answer the question – "What do we want to edit?". The answer is simple – **everything**, including the roles. We have create another edit operation, let's do it like we used to. Go to **user controller (admin)** and add "**editGet**" function:

```
editGet: (req, res) => {
  let id = req.params.id;

  User.findById(id).then(user => {
    Role.find({}).then(roles => {
      for (let role of roles) {
        if (user.roles.indexOf(role.id) !== -1) {
          role.isChecked = true;
        }
      }

      res.render('admin/user/edit', {user: user, roles: roles})
    })
  });
},
```

Since we want to **edit** user's **roles**, we should display current **user's** info and **all** possible **roles** to be taken in our view. Speaking of which we have to create - go "**views/admin/user**" and create "**edit.hbs**":

```
<div class="container body-content span=8 offset=2">
  <div class="well">
    <form class="form-horizontal"
action="/admin/user/edit/{{ user.id }}" method="post">
      <fieldset>
        <legend>Edit User - {{ user.fullName }}</legend>
        <div class="form-group">
          <label class="col-sm-4 control-label"
for="email">Email</label>
          <div class="col-sm-4 ">
            <input class="form-control" id="email"
value="{{ user.email }}" name="email" required type="text">
          </div>
        </div>
        <div class="form-group">
          <label class="col-sm-4 control-label"
for="fullName">Full Name</label>
          <div class="col-sm-4 ">
            <input type="text" class="form-control"
id="fullName" value="{{ user.fullName }}" name="fullName" required>
          </div>
        </div>
        <div class="form-group">
          <label class="col-sm-4 control-label"
for="password">Password</label>
          <div class="col-sm-4">
            <input type="password" class="form-control"
id="password" placeholder="Password" name="password">
          </div>
        </div>
        <div class="form-group">
          <label class="col-sm-4 control-label"
```



```

for="confirmedPassword">Confirm Password</label>
    <div class="col-sm-4">
        <input type="password" class="form-control"
id="confirmedPassword" placeholder="Password"
name="confirmedPassword">
    </div>
</div>
<div class="form-group">
    {{#each roles}}
        <div class="col-sm-4 col-sm-offset-4">
            <input type="checkbox" name="roles"
value="{{ this.name }}" {{#if this.isChecked}} checked {{/if}}/>
            {{this.name}}
        </div>
    {{/each}}
</div>
<div class="form-group">
    <div class="col-sm-4 col-sm-offset-4">
        <a class="btn btn-default"
href="/admin/user">Cancel</a>
        <button type="submit" class="btn btn-
success">Edit</button>
    </div>
</div>
</fieldset>
</form>
</div>
</div>

```

Go back to the user controller (**admin**) and add the post function:

```

editPost: (req, res) => {
    let id = req.params.id;
    let userArgs = req.body;

```

First we want to make some **validations**. Not only we will check for **null** all values but we will check if there is any user with that email and with different id given (because the **email** may **not** be **changed**, but if we **want** to **change** it we have to have sure **no one** uses it):

```

User.findOne({email: userArgs.email, _id: {$ne: id}}).then(user => {
    let errorMsg = '';
    if (user) {
        errorMsg = 'User with the same username exists!';
    } else if (!userArgs.email) {
        errorMsg = 'Email cannot be null!'
    } else if (!userArgs.fullName) {
        errorMsg = 'Name cannot be null!'
    } else if (userArgs.password !== userArgs.confirmedPassword) {
        errorMsg = 'Passwords do not match!'
    }
}

```

Then we will get the **roles** selected by the admin in our “**roles**” **array**: bad thing about it – it will give us the name. This means that we have to **search** in our **database** and **find** all **roles** with names given:

```
Role.find({}).then(roles => {
  if(!userArgs.roles) {
    userArgs.roles = [];
  }

  let newRoles = roles.filter(role => {
    return userArgs.roles.indexOf(role.name) !== -1;
  }).map(role => {
    return role.id;
  });
});
```

Then we make our **update** statement – we update user info and we check if the **admin** wants to change the password. If any **password** is inserted we will call our help “**encryption**” module to generate **new password**:

```
User.findOne({_id : id}).then(user => {
  user.email = userArgs.email;
  user.fullName = userArgs.fullName;

  let passwordHash = user.passwordHash;
  if (userArgs.password) {
    passwordHash = encryption.hashPassword(userArgs.password, user.salt);
  }

  user.passwordHash = passwordHash;
  user.roles = newRoles;

  user.save((err) => {
    if (err) {
      res.redirect('/');
    } else {
      res.redirect('/admin/user/all');
    }
  })
})
})
```

```
const role = require('mongoose').model('role');
const encryption = require('../../utilities/encryption');
```

Here is how the method should look(partially):

```
editPost: (req, res) => {
  let id = req.params.id;
  let userArgs = req.body;

  User.findOne({email: userArgs.email, _id: {$ne: id}}).then(user => {
    let errorMsg = '';
    if (user) {
      errorMsg = 'User with the same username exists!';
    } else if (!userArgs.email) {...} else if (!userArgs.fullName) {...} else if (userArgs.password !== userArgs.confirmedPassword) {
      errorMsg = 'Passwords do not match!'
    }

    if (errorMsg) {
      userArgs.error = errorMsg;
      res.render('admin/user/edit', userArgs);
    } else {
      Role.find({}).then(roles => {
        let newRoles = roles.filter(role => {
          return userArgs.roles.indexOf(role.name) !== -1;
        }).map(role => {
          return role.id;
        });

        User.findOne({_id: id}).then(user => {
          user.email = userArgs.email;
          user.fullName = userArgs.fullName;

          let passwordHash = user.passwordHash;
          if (userArgs.password) {
            passwordHash = encryption.hashPassword(userArgs.password, user.salt);
          }

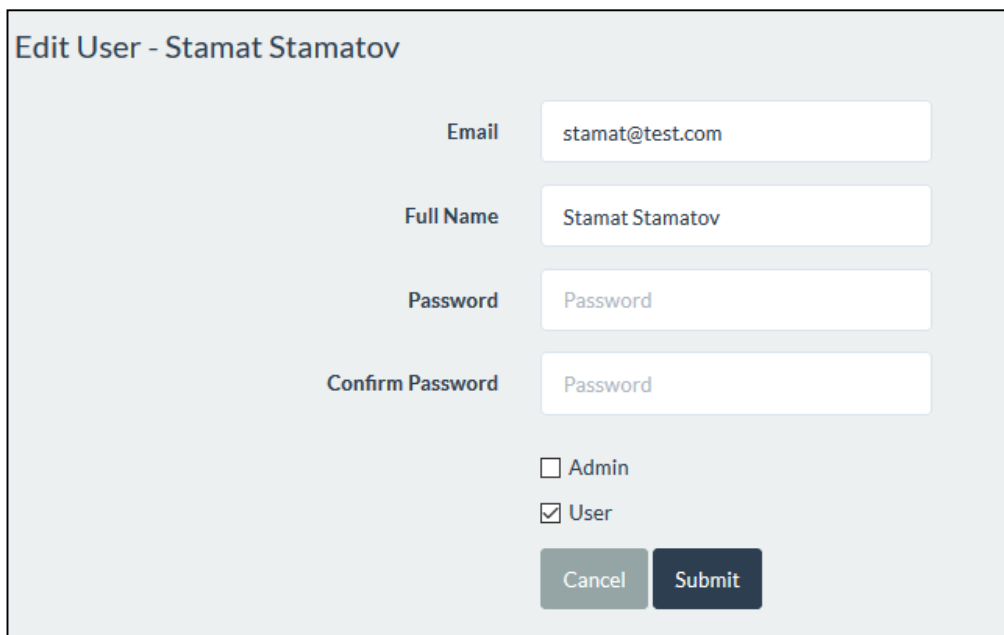
          user.passwordHash = passwordHash;
          user.roles = newRoles;
        });
      });
    }
  });
}
```

Go back to the „route.js“ and add the script we wrote above:

```
app.get('/admin/user/all', adminController.user.all);

app.get('/admin/user/edit/:id', adminController.user.editGet);
app.post('/admin/user/edit/:id', adminController.user.editPost);
```

Let's see how it looks:



That should be everything. If you test the form now, you can see that it works.

***Optional:** update also the Roles collection in database.

3. Deleting User

Editing users is nice, but we may want to delete users as well. To do that we should start with creating new function in our **controller**:

```
deleteGet: (req, res) => {  
  let id = req.params.id;  
  User.findById(id).then(user => {  
    res.render('admin/user/delete', { userToDelete: user })  
  });  
}
```

Then we will continue with viewing the form ("views/admin/user/delete.hbs"):

```
<div class="container body-content span=8 offset=2">  
  <div class="well">  
    <form class="form-horizontal"  
action="/admin/user/delete/{{userToDelete.id}}" method="post">  
      <fieldset>  
        <legend>Delete User - {{ userToDelete.fullName  
}}</legend>  
        <div class="form-group">  
          <label class="col-sm-4 control-label"  
for="email">Email</label>  
          <div class="col-sm-4 ">  
            <input class="form-control" id="email"  
value="{{ userToDelete.email }}" name="email" required type="text"  
disabled>  
          </div>  
        </div>  
        <div class="form-group">  
          <label class="col-sm-4 control-label"  
for="fullName">Full Name</label>  
          <div class="col-sm-4 ">  
            <input type="text" class="form-control"  
id="fullName" value="{{ userToDelete.fullName }}" name="fullName"  
disabled>  
          </div>  
        </div>  
        <div class="form-group">  
          <div class="col-sm-4 col-sm-offset-4">  
            <a class="btn btn-default"  
href="/admin/user">Cancel</a>  
            <button type="submit" class="btn btn-  
danger">Delete</button>  
          </div>  
        </div>  
      </fieldset>  
    </form>  
  </div>  
</div>
```

Let's see how it looks (this step will be available if you have the **routing** ☺):

Delete User - Mihail Todorov

Email	<input type="text" value="misho@softuni.bg"/>
Full Name	<input type="text" value="Mihail Todorov"/>
<div><input type="button" value="Cancel"/> <input type="button" value="Delete"/></div>	

Let's finish our logic in the **user controller** (admin):

```
deletePost: (req, res) => {
  let id = req.params.id;

  User.findOneAndRemove({_id: id}).then(user => {
    user.prepareDelete();
    res.redirect('/admin/user/all');
  })
}
```

However, we are talking about deleting a user. This means **two** things: we have to **delete all articles**, and remove the deleted **user reference** in **Role collection**. We are going to write a helper function in our user schema to help us with keeping our data stable in database. Go to "**User.js**" and add:

Inside the:

```
userSchema.method ({
```

Add this method:

```
prepareDelete: function () {
  for (let role of this.roles) {
    Role.findById(role).then(role => {
      role.users.remove(this.id);
      role.save();
    })
  }

  let Article = mongoose.model('Article');
  for (let article of this.articles) {
    Article.findById(article).then(article => {
      article.prepareDelete();
      article.remove();
    })
  }
},
```

What will this code do is simply help us when we **delete** a **user** to delete its presence anywhere in **database**. What you can discover here is that we have write one more **prepare deletion** method and it's for our **article**.

4. Article Consistency Update

Let's go into the "Article.js" and add those two methods (make sure they are **above** making the model):

```
articleSchema.method({
  prepareInsert: function () {
    let User = mongoose.model('User');
    User.findById(this.author).then(user => {
      user.articles.push(this.id);
      user.save();
    });
  },

  prepareDelete: function () {
    let User = mongoose.model('User');
    User.findById(this.author).then(user => {
      // If user is not deleted already - when we delete from User.
      if(user) {
        user.articles.remove(this.id);
        user.save();
      }
    });
  },
});
```

```
const Article = mongoose.model('Article', articleSchema);
module.exports = Article;
```

What are doing now is making our database up-to-date. In order to that let's invoke the above two actions when **creating** and **deleting** article respectively. Go to the **article** controller – in our "createPost" function and remove the highlighted code:

```
articleArgs.author = req.user.id;
Article.create(articleArgs).then(article => {
  req.user.articles.push(article.id);
  req.user.save(err => {
    if (err) {
      res.redirect('/', {error : err.message});
    } else {
      res.redirect('/');
    }
  });
});
```

Add the method we just created for our model. This is how it should look (only this part is **changed**):

```
...
articleArgs.author = req.user.id;
Article.create(articleArgs).then(article => {
  article.prepareInsert();
  res.redirect('/');
});
```

And in our “**deletePost**” function.

Delete everything in your “**findOneAndRemove**” call:

```
Article.findOneAndRemove({_id: id}).populate('author').then(article => {
  let author = article.author;

  // Index of the article's ID in the author's articles.
  let index = author.articles.indexOf(article.id);

  if(index < 0) {
    let errorMsg = 'Article was not found for that author!';
    res.render('article/delete', {error: errorMsg})
  } else {
    // Remove count elements after given index (inclusive).
    let count = 1;
    author.articles.splice(index, count);
    author.save().then(user => {
      res.redirect('/');
    });
  }
});
```

Instead call “**prepareDelete**”. Our **whole function** might look something like this:

```
Article.findOneAndRemove({_id: id}).then(article => {
  article.prepareDelete();
  res.redirect('/');
});
```

5. User Consistency Update

Now we are done with keeping our **data consistent** for our **article** create/delete process. Now go back (“**User.js**”) and in the **user’s schema** add another function “**prepareInsert**”.

```
prepareInsert: function () {
  for (let role of this.roles) {
    Role.findById(role).then(role => {
      role.users.push(this.id);
      role.save();
    });
  }
}
```


This means that we have to rewrite our user register logic and remove some useless code:

```
    userObject.roles = roles;
    User.create(userObject).then(user => {
        role.users.push(user);
        role.save(err => {
            if(err) {
                registerArgs.error = err.message;
                res.render('user/register', registerArgs);
            }
            else {
                req.login(user, (err) => {
```

Remove the highlighted code and add:

```
    user.prepareInsert();
    req.login(user, (err) => {
```

Here is how the whole process **after user creation** might look:

```
let roles = [];
Role.findOne({name: 'User'}).then(role => {
    roles.push(role.id);

    userObject.roles = roles;
    User.create(userObject).then(user => {
        user.prepareInsert();
        req.login(user, (err) => {
            if (err) {
                registerArgs.error = err.message;
                res.render('user/register', registerArgs);
                return;
            }

            res.redirect('/');
        })
    })
});
```

We had to refactor our code. But remember when done right refactoring always pays off. 😊

Before we go simply add routes in our “route.js”:

```
app.get('/admin/user/delete/:id', adminController.user.deleteGet);
app.post('/admin/user/delete/:id', adminController.user.deletePost);
```

Summary: We have completed our **CRUD** operations over **users**. We can make admins now, rename users and even **change** their **password**! We also done some **refactoring** and made our **code** a little more **structured**!

IX. Creating Categories

Let's improve our blog a bit. We will create categories for our articles.

1. Creating the Category Model

What we will have here: each **article** may have **only one category**. This means that **one category** may have many **articles**. Category also has name. Go to “models” folder and create the “Category.js” model:

Here is how our model should look. We have added prepare delete method so when we delete on category to delete all articles with it.

```
const mongoose = require('mongoose');

let categorySchema = mongoose.Schema({
  name: {type: String, required: true, unique: true},
  articles: [{type: mongoose.Schema.Types.ObjectId, ref: 'Article'}]
});

categorySchema.method({
  prepareDelete: function () {
    let Article = mongoose.model('Article');
    for (let article of this.articles) {
      Article.findById(article).then(article => {
        article.prepareDelete();
        article.remove();
      })
    }
  }
});

categorySchema.set('versionKey', false);

const Category = mongoose.model('Category', categorySchema);
```

We set “versionKey” to false in order to keep consistency easier (and with no errors).

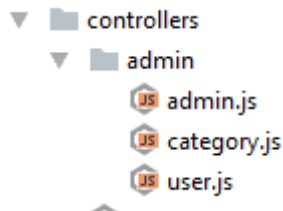
2. Creating the Relation

As we said in the previous step one category – many articles, one article – one category. What is left for us is update the article model:

```
author: {type: mongoose.Schema.Types.ObjectId, required: true, ref: 'User'},
category: {type: mongoose.Schema.Types.ObjectId, required: true, ref: 'Category'},
date: {type: Date, default: Date.now()};
```

3. Listing Categories in the Admin Panel

Right now, **we can't create articles**. That is a huge problem, having in mind that we're creating a blog. In order to fix that we **need to create at least one category** and give the user option to choose in which category he wants to post his article. Let's create new **category** controller in our “admin” folder. When we create the controller.



You may ask why do we want the controller to be in our admin section. Because we want **only admins** to be able to **create, update** or **delete** categories.

Let's start by the method who is going to list all available categories.

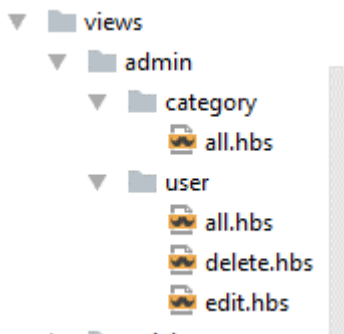
```
const Category = require('mongoose').model('Category');

module.exports = {
  all: (req, res) => {
    Category.find({}).then(categories => {
      res.render('admin/category/all', { categories: categories});
    })
  }
};
```

We have just required our **Category** model, but should it be called in our “**database.js**” beforehand? Go to “database.js” and require it.

```
require('../models/Article');
require('../models/Category');
```

Now that we are ready to pass some categories to the view – let's create the view itself. In “**views/admin**” folder create “**category**” folder:



Create “**all.hbs**” file and insert the following code:

```
<div class="container body-content">
  <div class="well">
    <h2>All Categories -
    <a href="/admin/category/create" class="btn btn-warning">Create New</a>
    </h2>
    <div class="row">
      <table class="table table-striped table-hover ">
        <thead>
          <tr>
            <th>#</th>
            <th>Name</th>
            <th>Actions</th>
          </tr>
```

```

        </thead>
        <tbody>
          {{#each categories}}
            <tr>
              <td>{{ this.id }}</td>
              <td>{{ this.name }}</td>
              <td>
                <a
href="/admin/category/edit/{{this.id}}">Edit</a>
                <a
href="/admin/category/delete/{{this.id}}">Delete</a>
              </td>
            </tr>
          {{/each}}
        </tbody>
      </table>
    </div>
  </div>
</div>

```

What we have to do is add the routing:

```
app.get('/admin/category/all', adminController.)
```

However, we don't have our **category controller**, go and require it in "admin.js" file:

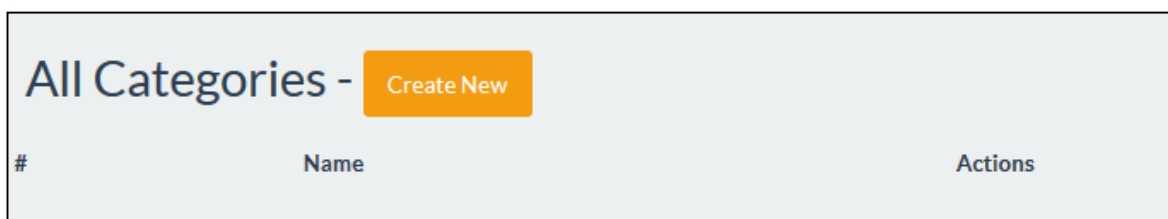
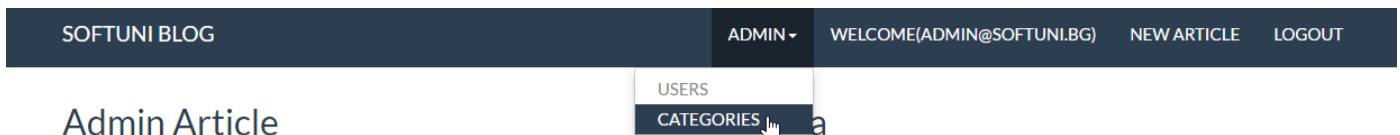
```
const userController = require('./user');
const categoryController = require('./category');
```

```
module.exports = {
  user: userController,
  category: categoryController
};
```

Now, back to the "route.js":

```
app.get('/admin/category/all', adminController.category.all);
```

Let's start it and see how it looks:



Pretty standard, but it will do the job. Next step – Creating categories.

4. Creating Categories

You probably can imagine that we need to **create new function** in the **category** controller, that will **create the categories**:

```
createGet: (req, res) => {  
  |   res.render('admin/category/create');  
  },  
}
```

This means that we have to **create** the **view** again. Go into “view/admin/category” and add “**create.hbs**”:

```
<div class="container body-content span=8 offset=2">  
  <div class="well">  
    <form class="form-horizontal" action="/admin/category/create"  
method="POST">  
      <fieldset>  
        <legend>New Category</legend>  
  
        <div class="form-group">  
          <label class="col-sm-4 control-label"  
for="name">Category Name</label>  
          <div class="col-sm-4 ">  
            <input type="text" class="form-control"  
id="name" placeholder="Category Name" name="name">  
          </div>  
        </div>  
  
        <div class="form-group">  
          <div class="col-sm-4 col-sm-offset-4">  
            <a class="btn btn-default"  
href="/admin/category/all">Cancel</a>  
            <button type="submit" class="btn btn-  
primary">Submit</button>  
          </div>  
        </div>  
      </fieldset>  
    </form>  
  </div>  
</div>
```

Then back into our controller, figuring out how to create categories:

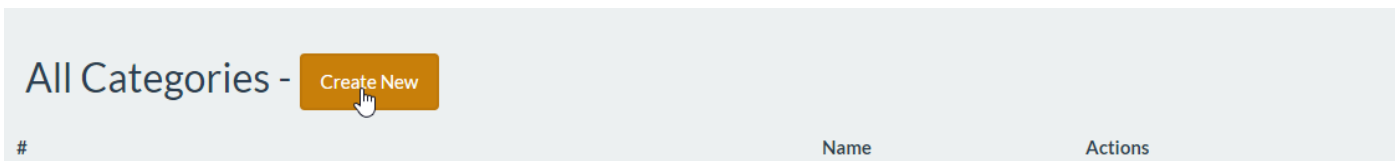
```
createPost: (req, res) => {
  let categoryArgs = req.body;

  if(!categoryArgs.name) {
    let errorMsg = 'Category name cannot be null!';
    categoryArgs.error = errorMsg;
    res.render('admin/category/create', categoryArgs);
  }
  else {
    Category.create(categoryArgs).then(category => {
      res.redirect('/admin/category/all');
    })
  }
},
```

One more thing to connect the dots:

```
app.get('/admin/category/create', adminController.category.createGet);
app.post('/admin/category/create', adminController.category.createPost);
```

We should fix the link on **the create button in the listing view** now:



And when we click it, we should see this form:

New Category

Category Name

Category Name

Cancel Submit

Now you should be able to create categories and list them like that:

All Categories - Create New

#	Name	Actions
1	News	Edit Delete
3	Programming Languages	Edit Delete
2	Technology	Edit Delete

Now we can finally fix the article creation function.

5. Article Consistency Update

Now we have to **update** our **article creation**. Few things here: **add** one more **field** to the form, this field must be with **values** from our **database** (if any). Therefore, we should think about **consistency** in our database. To should go back to the “**Article.js**” and improve our methods in the **article’s schema**:

Our “**prepareInsert**” function:

```
articleSchema.method({
  prepareInsert: function () {
    let User = mongoose.model('User');
    User.findById(this.author).then(user => {
      user.articles.push(this.id);
      user.save();
    });

    let Category = mongoose.model('Category');
    Category.findById(this.category).then(category => {
      // If the article is created without category - if there are no categories.
      if (category) {
        category.articles.push(this.id);
        category.save();
      }
    });
  },
});
```

And “**prepareDelete**” function after that:

```
prepareDelete: function () {
  let User = mongoose.model('User');
  User.findById(this.author).then(user => {
    // If user is not deleted already - when we delete from User.
    if (user) {
      user.articles.remove(this.id);
      user.save();
    }
  });

  let Category = mongoose.model('Category');
  Category.findById(this.category).then(category => {
    // If the category is not already deleted.
    if (category) {
      category.articles.remove(this.id);
      category.save();
    }
  });
},
```

Now, when we try to **delete** an **article** and we use “**prepareDelete**” we will delete it’s reference in the **category** where it **belonged** to (if any). **Same** goes for **adding** article: if we add **article** we will **add** the article’s id to **category**.

6. Article Creation

In order to include category into the article we can start from the view. Go to “views/article/create.hbs”. Below the “content” section put this:

```
<div class="form-group">
  <label class="col-sm-4 control-label"
for="category">Category</label>
  <div class="col-sm-6">
    <select class="form-control" id="select" name="category">
      {{#each categories}}
        <option
value="{{this.id}}">{{this.name}}</option>
      {{/each}}
    </select>
  </div>
</div>
```

It may look like this:

```
<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="conter
  </div>
</div>

<div class="form-group">
  <label class="col-sm-4 control-label" for="category">Category</label>
  <div class="col-sm-6">
    <select class="form-control" id="select" name="category">
      {{#each categories}}
        <option value="{{this.id}}">{{this.name}}</option>
      {{/each}}
    </select>
  </div>
</div>
```

Okay, we have the view to display the categories, now we have to pass them from the **controller (article)**:

```
createGet: (req, res) => {
  if (!req.isAuthenticated()) {
    let returnUrl = '/article/create';
    req.session.returnUrl = returnUrl;

    res.redirect('/user/login');
    return;
  }

  Category.find({}).then(categories => {
    res.render('article/create', {categories: categories});
  });
},
```

Require the category on top of the file and then update “createGet” function.

```
const Article = require('mongoose').model('Article');
const Category = require('mongoose').model('Category');
```


Now, let's see if everything is fine. **Log in**, click to **"Create Article"** and see if **any categories** are **popping up**. Note that you should have any in database in order to show up.

7. Listing All Categories on the Home Page

Now that we have categories, we want to change our home page quite a bit. We want **all categories** to be **shown on the front page** as **hyperlinks** and when you **click on a category** to get redirected to a view which will hold all articles in the given category.

First of all, we want to edit our **home** controller. We want to create a **new function**:

```
const Article = mongoose.model('Article');
const User = mongoose.model('User');
const Category = mongoose.model('Category');

listCategoryArticles: (req, res) => {
  let id = req.params.id;

  Category.findById(id).populate('articles').then(category => {
    User.populate(category.articles, {path: 'author'}, (err) => {
      if (err) {
        console.log(err.message);
      }

      res.render('home/article', {articles: category.articles});
    });
  });
}
```

The idea behind it, is to create a new view, which will display **all of the articles** that are **in the selected category**. The view should look like this and should be in **"views/home/article.hbs"**:

```
<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>

```

Its **identical** with what we have currently in the "home/index" view. We will change that now. Go back to your **home** controller and find the **index()** function:

```

index: (req, res) => {
    Article.find({}).limit(6).populate('author').then(articles => {
        res.render('home/index', {articles: articles});
    })
},

```

We will change that to:

```

index: (req, res) => {
    Category.find({}).then(categories => {
        res.render('home/index', {categories: categories});
    })
},

```

As you can see, **instead** of **sending all articles** to the view, we are **sending all categories**. We will edit the "home/index" view to look like this:

```

<div class="container body-content">
  <div class="row">
    {{#each categories}}
      <div class="col-md-6">
        <header>
          <h2>
            <a href="/category/{{ this.id }}">
              {{ this.name }} ({{
this.articles.length }})
            </a>
          </h2>
        </header>
      </div>
    {{/each}}
  </div>
</div>

```

Next, we want to add the routing in order to display the homepage:

```

module.exports = (app) => {
  app.get('/', homeController.index);
  app.get('/category/:id', homeController.listCategoryArticles);
}

```

If we run the blog right now, we could see this:

SOFTUNI BLOG

REGISTER LOGIN

Programming (3)

Blogging (0)

The **number next** to each **category name** is the **count of articles** we have in the category. If we **click on a category**, we should **see the articles**, like we did on the **old home page**:

Admin Article

In computing, JavaScript () is a high-level, dynamic, untyped, and interpreted programming language. It has been standardized in the ECMAScript language specification. Alongside HTML and CSS, JavaScript is one of the three core technologies of World Wide Web content production; the majority of websites employ it, and all modern Web browsers support it without the need for plug-ins.[6] JavaScript is prototype-based with first-class functions, making it a multi-paradigm language, supporting object-oriented,[8] imperative, and functional programming styles.[6] It has an API for working with text, arrays, dates and regular expressions, but does not include any I/O, such as networking, storage, or graphics facilities, relying for these upon the host environment in which it is embedded.

Admin

Read more »

Sample Title

Sample Content

Admin

Read more »

Example

Example

Admin

Read more »

But what if we want to **edit or delete an article**?

8. Article Edit and Delete

If we want to **edit or delete articles**, we want to only **add the drop-down for our categories** to the **views**.

In the view "**article/edit**" find the row, below "Content" add this:

```

<div class="form-group">
  <label class="col-sm-4 control-label"
for="category">Category</label>
  <div class="col-sm-6">
    <select class="form-control" id="select" name="category">
      {{#each categories}}
        <option value="{{this.id}}">{{this.name}}</option>
      {{/each}}
    </select>
  </div>
</div>

```

Then in **article controller** – we have categories to display we have to send them to the view (Go to “editGet”):

```

editGet: (req, res) => {
  let id = req.params.id;

  if(!req.isAuthenticated()){
    let returnUrl = `/article/edit/${id}`;
    req.session.returnUrl = returnUrl;

    res.redirect('/user/login');
    return;
  }

  Article.findById(id).then(article => {
    req.user.isInRole('Admin').then(isAdmin => {
      if (!isAdmin && !req.user.isAuthor(article)) {...}
      Category.find({}).then(categories =>{
        article.categories = categories;

        res.render('article/edit', article)
      });
    });
  });
},

```

Then in article controller – “editPost” function change from:

```

Article.update({_id: id},
  {$set:
    {
      title: articleArgs.title,
      content: articleArgs.content,
    }}).then(updateStatus => {
    res.redirect(`/article/details/${id}`);
  })

```

To:

```
Article.findById(id).populate('category').then(article => {
  if (article.category.id !== articleArgs.category) {
    article.category.articles.remove(article.id);
    article.category.save();
  }

  article.category = articleArgs.category;
  article.title = articleArgs.title;
  article.content = articleArgs.content;

  article.save((err) => {
    if(err) {
      console.log(err.message);
    }

    Category.findById(article.category).then(category => {
      if(category.articles.indexOf(article.id) === -1){
        category.articles.push(article.id);
        category.save();
      }

      res.redirect(`/article/details/${id}`);
    })
  });
});
```

This will allow us to **edit the category** of a **given article**. To **delete a given article**, we will open the **"article/delete"** view and above the buttons (Submit and Reset):

```
<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>
```

Above it, add the following code:

```
<div class="form-group">
  <label class="col-sm-4 control-label"
for="articleContent">Category</label>
  <div class="col-sm-6">
    <select name="category" class="form-control" id="select"
disabled>
      <option
value="{{ category.id }}">{{ category.name }}</option>
    </select>
  </div>
</div>
```

And it should look like this:

```

<div class="form-group">
  <label class="col-sm-4 control-label" for="articleContent">Category</label>
  <div class="col-sm-6">
    <select name="category" class="form-control" id="select" disabled>
      <option value="{{category.id}}">{{category.name}}</option>
    </select>
  </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>

```

There is no need to change anything in our controller now, because we have updated the “**prepareDelete**” function which is now removing article’s reference from the **Category** collection.

What if we want to **edit a category**? The answer will wait for you in the next part of this chapter. 😊

9. Editing Categories

It's time to create the **edit** function (category controller):

```

editGet: (req, res) => {
  let id = req.params.id;

  Category.findById(id).then(category => {
    res.render('admin/category/edit', {category: category});
  });
},

```

The **view** will be the same as the **create** view but with filled **category** name:

```

<div class="container body-content span=8 offset=2">
  <div class="well">
    <form class="form-horizontal"
action="/admin/category/edit/{{category.id}}" method="POST">
      <fieldset>
        <legend>Edit Category</legend>

        <div class="form-group">
          <label class="col-sm-4 control-label"
for="name">Category Name</label>
          <div class="col-sm-4">
            <input type="text" class="form-control"
id="name" placeholder="Category Name" name="name" value="{{
category.name }}">
          </div>

```

```

        </div>
        <div class="form-group">
          <div class="col-sm-4 col-sm-offset-4">
            <a class="btn btn-default"
href="/admin/category/all">Cancel</a>
            <button type="submit" class="btn btn-
success">Submit</button>
          </div>
        </div>
      </fieldset>
    </form>
  </div>
</div>

```

Now let's create action when the above form is submitted:

```

editPost: (req, res) => {
  let id = req.params.id;

  let editArgs = req.body;

  if(!editArgs.name) {
    let errorMessage = 'Category name cannot be null';

    Category.findById(id).then(category => {
      res.render('admin/category/edit', {category: category, error: errorMessage});
    });
  } else {
    Category.findOneAndUpdate({_id:id}, {name: editArgs.name}).then(category => {
      res.redirect('/admin/category/all');
    });
  }
}

```

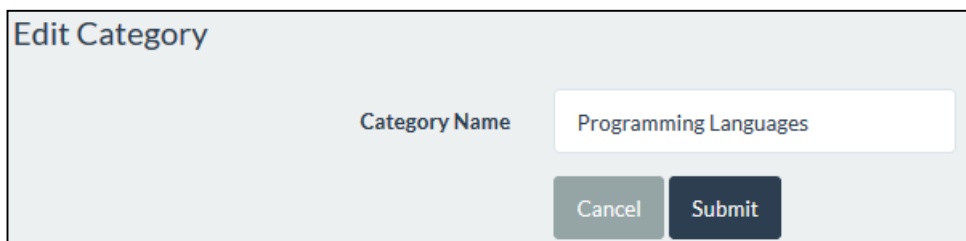
And don't forget the **routings**:

```

app.get('/admin/category/edit/:id', adminController.category.editGet);
app.post('/admin/category/edit/:id', adminController.category.editPost);

```

If we try to edit a category, we should see this:



That's all, the **last part** that we have in this chapter is the **category deletion**.

10. Deleting Categories

Like in the **edit part**, we will start with the function in our **category**:

```
deleteGet: (req, res) => {
  let id = req.params.id;

  Category.findById(id).then(category => {
    res.render('admin/category/delete', {category: category});
  });
},
```

Now we only need to create the **delete view**:

```
<div class="container body-content span=8 offset=2">
  <div class="well">
    <form class="form-horizontal"
action="/admin/category/delete/{{category.id}}" method="POST">
      <fieldset>
        <legend>Delete Category</legend>

        <div class="form-group">
          <label class="col-sm-4 control-label"
for="name">Category Name</label>
          <div class="col-sm-4 ">
            <input type="text" class="form-control"
id="name" placeholder="Category Name" name="name"
value="{{ category.name }}" disabled>
          </div>
        </div>
        <div class="form-group">
          <div class="col-sm-4 col-sm-offset-4">
            <a class="btn btn-default"
href="/admin/category/all">Cancel</a>
            <button type="submit" class="btn btn-
danger">Delete</button>
          </div>
        </div>
      </fieldset>
    </form>
  </div>
</div>
```

The special part here that we **delete all articles** that are **inside of our category**. We **should not delete the category** if it's **not empty**. We've seen **code similar to this** in **all functions** in **our other controllers**. Now we need the logic that will **delete the category**:


```

deletePost: (req, res) => {
  let id = req.params.id;

  Category.findOneAndRemove({_id: id}).then(category => {
    category.prepareDelete();
    res.redirect('/admin/category/all');
  });
}

```

And don't forget the **routing**!

```

app.get('/admin/category/delete/:id', adminController.category.deleteGet);
app.post('/admin/category/delete/:id', adminController.category.deletePost);

```

One more thing. Go to the “User.js” and add **before** making the model:

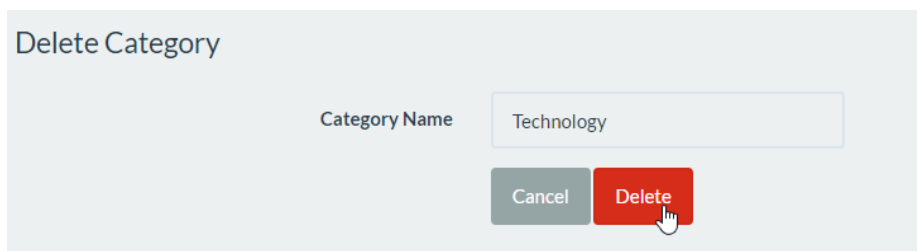
```

userSchema.set('versionKey', false);

const User = mongoose.model('User', userSchema);

```

Let's how it looks:



We're ready with this chapter as well.

X. Creating Tags

1. Creating the Tag Model

Creating the **Tag** model is really similar to the **Category** one. It will contain **only one field** called “name”, which **must be unique** and **required**.

```

const mongoose = require('mongoose');

let tagsSchema = mongoose.Schema({
  name: {type: String, required: true, unique: true},
  articles: [{type: mongoose.Schema.Types.ObjectId, ref: 'Article'}]
});

```

One tag will have **many** articles. One article may have many articles. Speaking of **relation**, we should define our one helper functions “**prepareInsert**”:

```

}tagsSchema.method({
  prepareInsert: function () {
    let Article = mongoose.model('Article');
    for (let article of this.articles) {
      Article.findById(article).then(article => {
        if (article.tags.indexOf(this.id) === -1) {
          article.tags.push(this.id);
          article.save();
        }
      });
    }
  },

  deleteArticle: function (articleId) {
    this.articles.remove(articleId);
    this.save();
  }
});

```

We are going to **add** and **remove** tags **dynamically** (without any additional **view**) so let's do one more helper **method** to help us maintaining data properly.

It is simple **function** which **removes from** the current **tag's articles** the reference of a **passed article**.

And before we create our model (that “**mongoose.model**” thing) make sure to set the version key property to false. At **bottom** of the file **add**:

```

tagsSchema.set('versionKey', false);
const Tag = mongoose.model('Tag', tagsSchema);

module.exports = Tag;

```

One more thing, go to “**database.js**” and require the module we have just created:

```

require('../models/Category');
require('../models/Tag');

```

2. Mapping Tags to Articles

The relation between tags and articles should be **Many-To-Many**. Let's go to **Article.js** and update the schema:

```

let articleSchema = mongoose.Schema({
  title: {type: String, required: true},
  content: {type: String, required: true},
  author: {type: mongoose.Schema.Types.ObjectId, required: true, ref: 'User'},
  category: {type: mongoose.Schema.Types.ObjectId, required: true, ref: 'Category'},
  tags: [{type: mongoose.Schema.Types.ObjectId, required: true, ref: 'Tag'}],
  date: {type: Date, default: Date.now()}
});

```

However, we have tags now we should **update** the **methods** that prepare the **removal** and the **addition** of a single **article**. Go to “**prepareInsert**” function and add:

```

let Tag = mongoose.model('Tag');
for (let tagId of this.tags){
  Tag.findById(tagId).then(tag => {
    if (tag) {
      tag.articles.push(this.id);
      tag.save();
    }
  });
}

```

The whole method should look like this:

```

prepareInsert: function () {
  let User = mongoose.model('User');
  User.findById(this.author).then(user => {
    user.articles.push(this.id);
    user.save();
  });

  let Category = mongoose.model('Category');
  Category.findById(this.category).then(category => {
    // If the article is created without category - if there are no categories.
    if (category) {
      category.articles.push(this.id);
      category.save();
    }
  });

  let Tag = mongoose.model('Tag');
  for (let tagId of this.tags){
    Tag.findById(tagId).then(tag => {
      if (tag) {
        tag.articles.push(this.id);
        tag.save();
      }
    });
  }
}

```

Now add the following code to the “**prepareDelete**” function:

```

let Tag = mongoose.model('Tag');
for (let tagId of this.tags){
  Tag.findById(tagId).then(tag => {
    if (tag) {
      tag.articles.remove(this.id);
      tag.save();
    }
  });
}

```

The whole function must be something like:

```

prepareDelete: function () {
  let User = mongoose.model('User');
  User.findById(this.author).then(user => {
    // If user is not deleted already - when we delete from User.
    if(user) {
      user.articles.remove(this.id);
      user.save();
    }
  });

  let Category = mongoose.model('Category');
  Category.findById(this.category).then(category => {
    // If the category is not already deleted.
    if (category) {
      category.articles.remove(this.id);
      category.save();
    }
  });

  let Tag = mongoose.model('Tag');
  for (let tagId of this.tags) {
    Tag.findById(tagId).then(tag => {
      if (tag) {
        tag.articles.remove(this.id);
        tag.save();
      }
    });
  }
}

```

It's pretty much like the upper function but instead of "push" we use "remove". ☺

One more thing before we are ready to create our tags. Let's add **one more** helper function:

```

deleteTag: function (tagId) {
  this.tags.remove(tagId);
  this.save();
}

```

3. Creating Tags

The **Tag** creation process should happen when you create article. That means that the **Article** create view will have one more input field of type **text** that will keep all of the tags with comma between each tag. When you create the tags you should read the input field then you should **split the string** that you receive by ',' and (**space**) and then insert each tag in the database (if not existing already). That's ok, but things are a more complicated since we are the ones who keep the consistency in our database. Let's write the algorithm for that in our "**Tags.js**" and export it as function:

```
module.exports = Tag;
```

```
module.exports.initializeTags = function (newTags, articleId) {
```

Make sure that function to be **below** the main “module.exports”. The function will require **tag names** and an article id. What our function will do is: **create** any currently **non-existing tag** and add the given article to it’s articles list:

```
for (let newTag of newTags) {
  if(newTag) {
    Tag.findOne({name: newTag}).then(tag => {
      // If is existing - insert the article in it.
      if (tag) {
        if(tag.articles.indexOf(articleId) === -1) {
          tag.articles.push(articleId);
          tag.prepareInsert();
          tag.save();
        }
      }
      // If not - create and insert the article.
      else {
        Tag.create({name: newTag}).then(tag => {
          tag.articles.push(articleId);
          tag.prepareInsert();
          tag.save();
        })
      }
    });
  }
}
```

We have completed the function for initializing tags. Now we have to call it. But first update the view (“article/create.hbs”). **Above** the form **buttons** add another text field:

```
<div class="form-group">
  <label class="col-sm-4 control-label" for="article_tags">Tags</label>
  <div class="col-sm-4 ">
    <input type="text" class="form-control" id="article_tags"
placeholder="Tags"
name="tagNames">
  </div>
</div>
```

Let's go to our **controller (article)**. Go to “createPost” function and **update** the **logic** about creating the article like this (only **add** the currently **missing code** in your function):

```

articleArgs.author = req.user.id;
articleArgs.tags = [];
Article.create(articleArgs).then(article => {
  // Get the tags from the input, split it by space or semicolon,
  // then remove empty entries.
  let tagNames = articleArgs.tagNames.split(/\s+|,/).filter(tag => {return tag});
  initializeTags(tagNames, article.id);

  article.prepareInsert();
  res.redirect('/');
});

```

We want to use that “initializeTags” method but we have to **require** it first:

```

const category = require('mongoose').model('Category');
const initializeTags = require('./../models/Tag').initializeTags;

```

This should be our logic when we create tags. On next part we will discuss how to edit and delete tags.

4. Editing Tags (Article)

When you **edit or delete tags**, you should do it when you **edit or delete articles**. Go to the “article/edit” view and above the form **buttons** add the following:

```

<div class="form-group">
  <label class="col-sm-4 control-label" for="tags">Tags</label>
  <div class="col-sm-4 ">
    <input type="text" class="form-control" id="tags" name="tags"
value="{{ tagNames }}">
  </div>
</div>

```

Now with editing an article we pass it’s tags so we have to back to **article controller** and **update** our “editGet” function:

```

Article.findById(id).populate('tags').then(article => {
  req.user.isInRole('Admin').then(isAdmin => {
    if (!isAdmin && !req.user.isAuthor(article)) {
      res.redirect('/');
      return;
    }
    Category.find({}).then(categories =>{
      article.categories = categories;

      article.tagNames = article.tags.map(tag => {return tag.name});
      res.render('article/edit', article)
    });
  });
});

```

So we pass the article’s tags for editing. We should be able to see them now (separated with “,” – Javascript automatically will do a **string join** over our array).

Article Title

Content

Here we will discuss about the newest technology in Javascript ...

Category

Tags

Here is how it should look (example).

When the form is **submitted** we should update our both tags and articles (on "editPost"):

```
Article.findById(id).populate('category tags').then(article => {
  if (article.category.id !== articleArgs.category) {
    article.category.articles.remove(article.id);
    article.category.save();
  }

  article.category = articleArgs.category;
  article.title = articleArgs.title;
  article.content = articleArgs.content;

  let newTagNames = articleArgs.tags.split(/\s+/,).filter(tag => {return tag});

  // Get me the old article's tags which are not
  // re-entered.
  let oldTags = article.tags
    .filter(tag => {
      return newTagNames.indexOf(tag.name) === -1;
    });

  for(let tag of oldTags) {
    tag.deleteArticle(article.id);
    article.deleteTag(tag.id);
  }

  initializeTags(newTagNames, article.id);

  Category.findById(article.category).then(category => {
    if(category.articles.indexOf(article.id) === -1) {
      category.articles.push(article.id);
    }
  });
});
```

Here is how the method should change. In the next part we will see how to delete tags.

5. Delete Tags (Article)

There is actually nothing special here. Just go to the view and another input field below the **categories** input field.

```
<div class="form-group">
  <label class="col-sm-4 control-label" for="article_tags">Tags</label>
  <div class="col-sm-4 ">
    <input type="text" class="form-control" id="article_tags"
    name="tags" value="{{ tagNames }}" disabled>
  </div>
</div>
```

Now we added new field for the view let's just sent it (in article **controller - deleteGet**). **Update** only the **highlighted** parts:


```
Article.findById(id).populate('category tags').then(article => {
  req.user.isInRole('Admin').then(isAdmin => {
    if (!isAdmin && !req.user.isAuthor(article)) {
      res.redirect('/');
      return;
    }

    article.tagNames = article.tags.map(tag => {return tag.name});
    res.render('article/delete', article)
  });
});
```

And because we have changed “prepareDelete” method for the Article model we do not have anything more in order to delete articles and their tags.

6. Searching by Tags

Here, you should print **each article with its tags and create hyperlink for each tag**. Those hyperlinks will lead to a **new view** that will **render only the articles that contain that tag**.

First of all, we should display **all tags for each article**. We will do that in the **single article view** first (“views/article/details.hbs”).

You should be familiar with this part of the code. In the beginning of the **footer** we will add new **<div>** that will contain our tags:

```
<div class="pull-left">
  {{# each article.tags}}
    <a class="btn btn-default btn-xs"
href="/tag/{{this.name}}">{{ this.name }}</a>
  {{/each}}
</div>
```

The end result should look like this:

```
<footer>
  <div class="pull-left">
    {{# each article.tags}}
      <a class="btn btn-default btn-xs" href="/tag/{{this.name}}">{{ this.name }}</a>
    {{/each}}
  </div>
  <div class="pull-right">
```

We have to go back to the **article** controller. And pass the much needed tags:

```
details: (req, res) => {
  let id = req.params.id;

  Article.findById(id).populate('author tags').then(article => {
    if (!req.user) {
      res.render('article/details', { article: article, isUserAuthorized: false });
      return;
    }

    req.user.isInRole('Admin').then(isAdmin => {
      let isUserAuthorized = isAdmin || req.user.isAuthor(article);

      res.render('article/details', { article: article, isUserAuthorized: isUserAuthorized });
    });
  });
},
```

If we open an article now, we should see something like this:

Technology

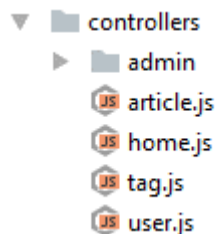
Here we will talk about the newest technologies in Javascript...

Admin

Technology Javascript

Edit Delete Back

In the **lower left part** of the article you can see the tags. The links are **not working** for now, but we will fix that. First, let's create new **Tag** controller:



Let's create a function, with which we will display all articles by given tag:

```
const Article = require('mongoose').model('Article');
const Tag = require('mongoose').model('Tag');

module.exports = {
  listArticlesByTag: (req, res) => {
    let name = req.params.name;

    Tag.findOne({name: name}).then(tag => {
      Article.find({tags: tag.id}).populate('author tags')
        .then(articles => {
          res.render('tag/details', {articles: articles, tag: tag});
        })
    })
  }
};
```

Basically, it means get the **tag** and give it to the view. Speaking of which we should create. Go to **“views”** folder and **add** another **folder** name **“tag”** and lastly create **“details.hbs”** with the following code:

```
<div class="container body-content">
  <h1>Searching by tag - {{ tag.name }}</h1>
  <div class="row">
    {{#each articles}}
    <div class="col-md-6">
      <article>
        <header>
          <h2>{{ this.title }}</h2>
        </header>

        <p>
          {{ this.content }}
        </p>
        <p>
          {{#each this.tags}}
          <a class="btn btn-default btn-xs"
            href="/tag/{{this.name}}">{{ this.name }}</a>
          {{/each}}
        </p>
      </article>
    </div>
  </div>
</div>
```

```

        </p>
        <small class="author">
            {{ article.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>

```

One thing to make it work – the routing. Make sure you add the code below before the authentication middleware:

```
app.get('/tag/:name', tagController.listArticlesByTag);
```

Make sure to **require** the tag controller:

```
const tagController = require(' ../controllers/tag');
```

Let's see what we get when we **click on a tag**:

Searching by tag - Technology

Technology

Here we will talk about the newest technologies in Javascript...

Technology Javascript

Read more »

What is Technology?

Technology ("science of craft", from Greek τέχνη, techne, "art, skill, cunning of hand"; and -λογία, -logia[2]) is the collection of techniques, skills, methods and processes used in the production of goods or services or in the accomplishment of objectives, such as scientific investigation. Technology can be the knowledge of techniques, processes, and the like, or it can be embedded in machines, computers, devices, and factories, which can be operated by individuals without detailed knowledge of the workings of such things. The human species' use of technology began with the conversion of natural resources into simple tools. The prehistoric discovery of how to control fire and the later Neolithic Revolution increased the available sources of food and the invention of the wheel helped humans to travel in and control their environment. Developments in historic times, including the printing press, the telephone, and the Internet, have lessened physical barriers to communication and allowed humans to interact freely on a global scale. The steady progress of military technology has brought weapons of ever-increasing destructive power, from clubs to nuclear weapons.

Technology

Read more »

We see **all of the articles with the given tag**, with the **rest of their tags**. We can **implement that in the category listing** as well. To do that simply open the **"home/articles"** view and find:

```

<p>
    {{ this.content }}
</p>
<small class="author">
    {{ this.author.fullName }}
</small>

```

After the **content** write the following code:

```
<p>
  {{#each this.tags }}
    <a class="btn btn-default btn-xs"
      href="/tag/{{this.name}}">{{ this.name }}</a>
  {{/each}}
</p>
```

The result should be this:

```
<p>
  {{ this.content }}
</p>

<p>
  {{#each this.tags }}
    <a class="btn btn-default btn-xs"
      href="/tag/{{this.name}}">{{ this.name }}</a>
  {{/each}}
</p>
<small class="author">
  {{ this.author.fullName }}
</small>
```

But in order to display tags we should populate them from the controller. Go to “controllers/**home**” and **update** the function which is displaying the articles:

```
listCategoryArticles: (req, res) => {
  let id = req.params.id;

  Category.findById(id).populate('articles').then(category => {
    User.populate(category.articles, {path: 'author'}, (err) =>{
      if (err) {
        console.log(err.message);
      }

      Tag.populate(category.articles, {path: 'tags'}, (err) =>{
        if (err) {
          console.log(err.message);
        }

        res.render('home/article', {articles: category.articles})
      });
    });
  });
};
```

Remember to require the **Tag** model in the beginning of our file:

```
const Category = mongoose.model('Cat');
const Tag = mongoose.model('Tag');
```

Let's see the listings in a random category:

Technology

Here we will talk about the newest technologies in Javascript...

Technology Javascript

Admin

Read more »

What is Technology?

Technology ("science of craft", from Greek τέχνη, techne, "art, skill, cunning of hand"; and -λογία, -logia) is the collection of techniques, skills, methods and processes used in the production of goods or services or in the accomplishment of objectives, such as scientific investigation. Technology can be the knowledge of techniques, processes, and the like, or it can be embedded in machines, computers, devices, and factories, which can be operated by individuals without detailed knowledge of the workings of such things. The human species' use of technology began with the conversion of natural resources into simple tools. The prehistoric discovery of how to control fire and the later Neolithic Revolution increased the available sources of food and the invention of the wheel helped humans to travel in and control their environment. Developments in historic times, including the printing press, the telephone, and the Internet, have lessened physical barriers to communication and allowed humans to interact freely on a global scale. The steady progress of military technology has brought weapons of ever-increasing destructive power, from clubs to nuclear weapons.

Technology

Admin

Read more »

© 2016 - Software University Foundation

This is everything. We want to **congratulate you** if you got to this point and **thank you** for your patience.

We hope that **you've enjoyed working** with **Node.js**, **Express** and **MongoDB**. Farewell, my friends! 😊