

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.

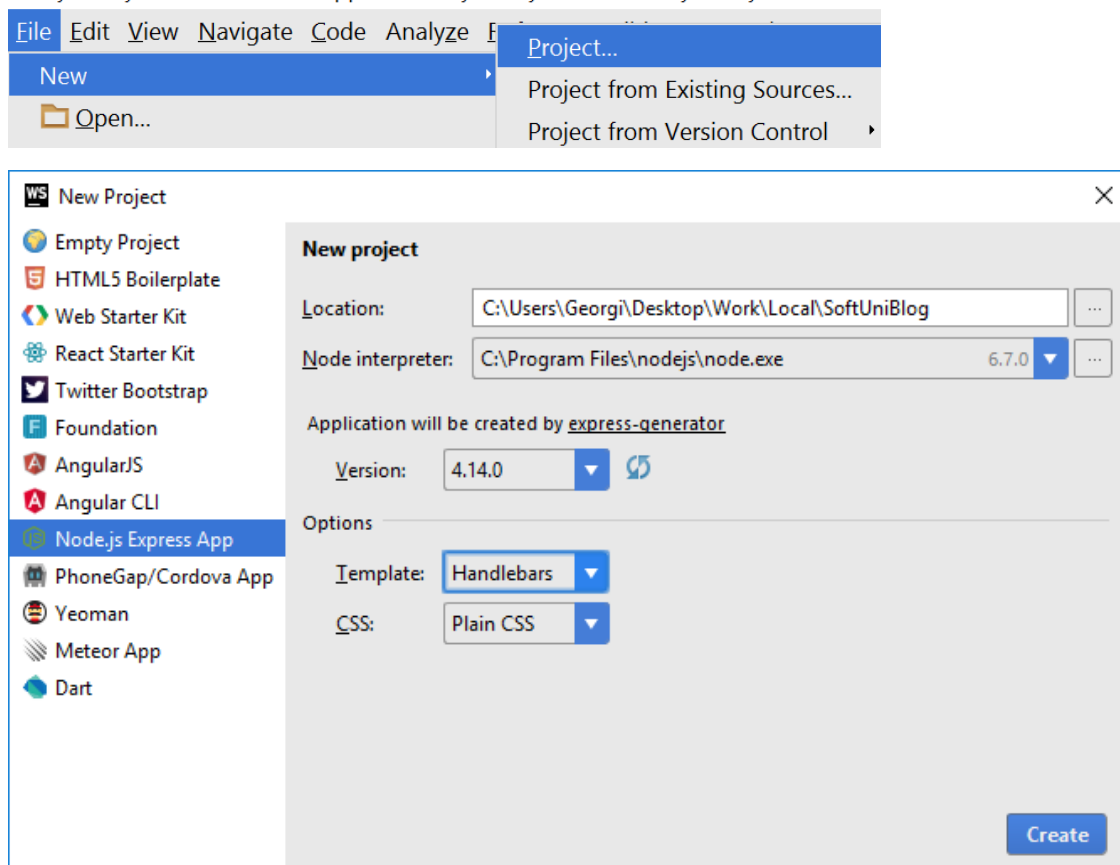
Chapters from I to III are for advanced users, but is recommended to be read. There's a [skeleton](#) which you can use and start from chapter IV.

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**)

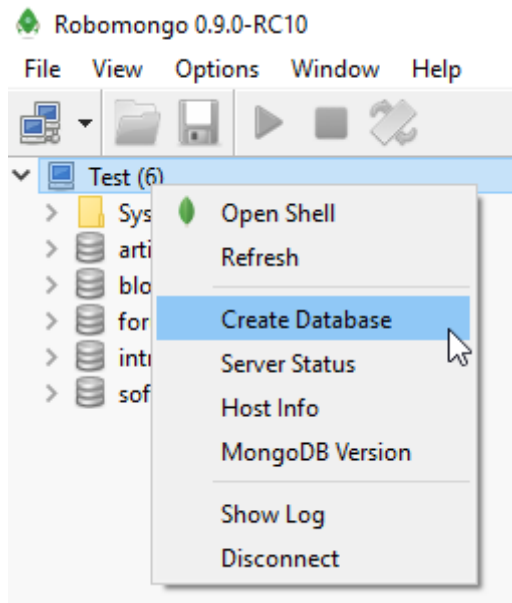


Make sure that you have [Node interpreter](#) installed and the chosen directory is the right one.

- 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**”.

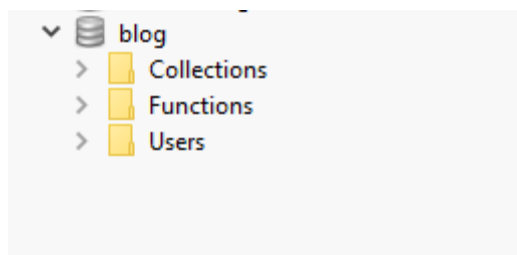
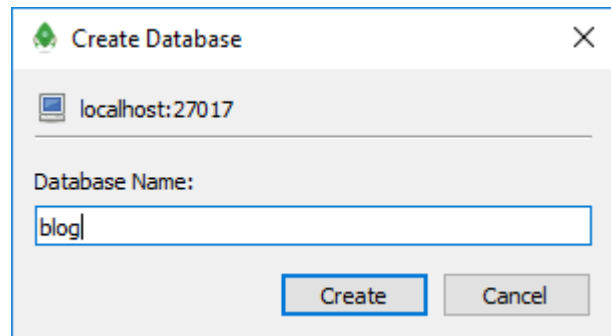


mongo

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

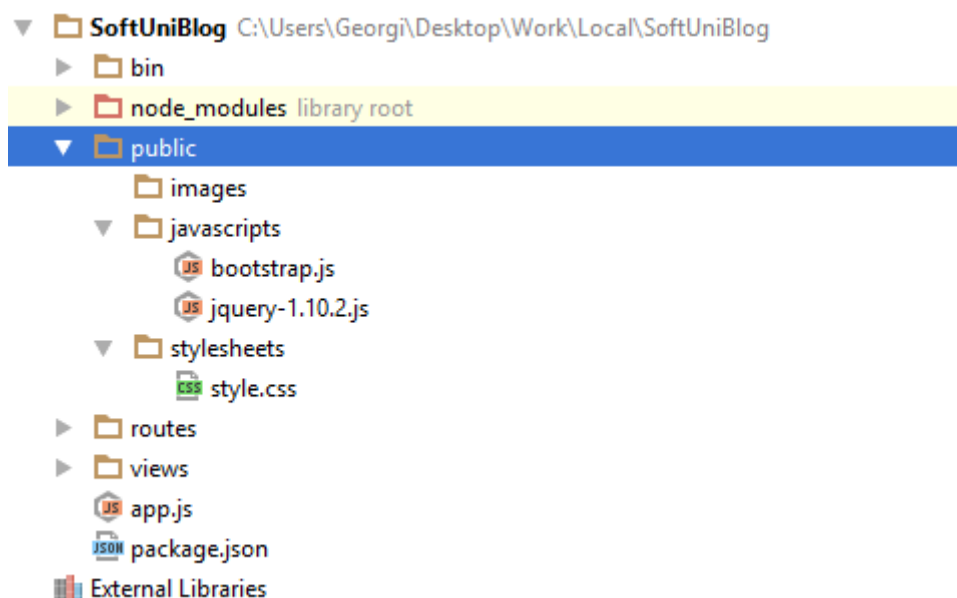
```

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:

```

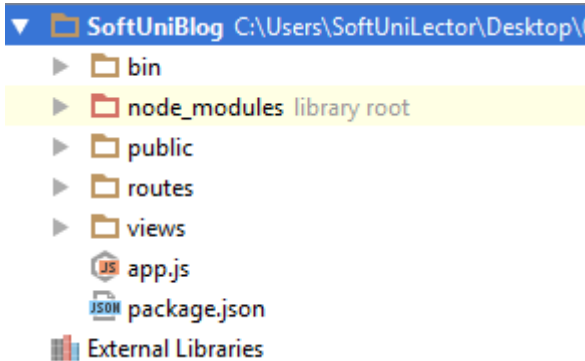
<!DOCTYPE html>
<html>
  <head>
    <title>SoftUni Blog</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
    <script src="/javascripts/jquery-1.10.2.js"></script>
    <script src="/javascripts/bootstrap.js"></script>
  </head>
  <body>
    <header>
      <div class="navbar navbar-default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

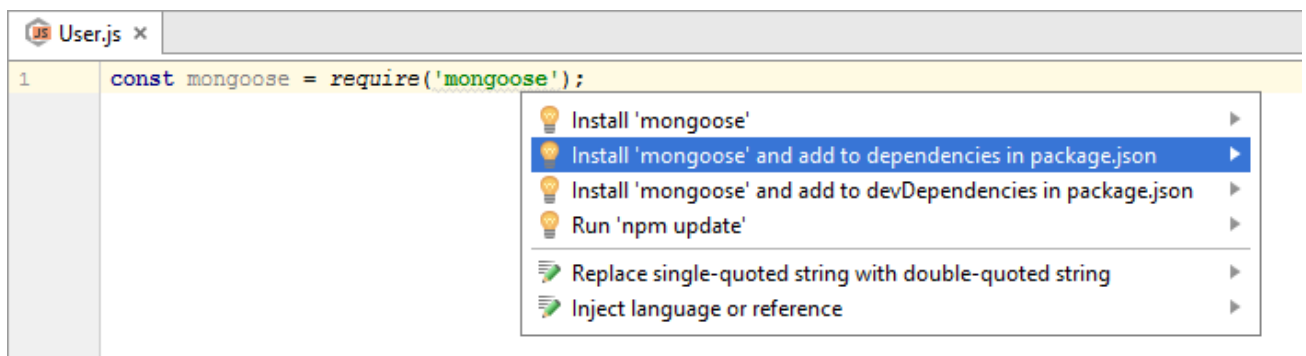
- Should have a unique login name, let's say **email**
- Should have a **passwordHash** (which we will won't save in it's pure view)
- Should have a full name, let's say **fullName**

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.

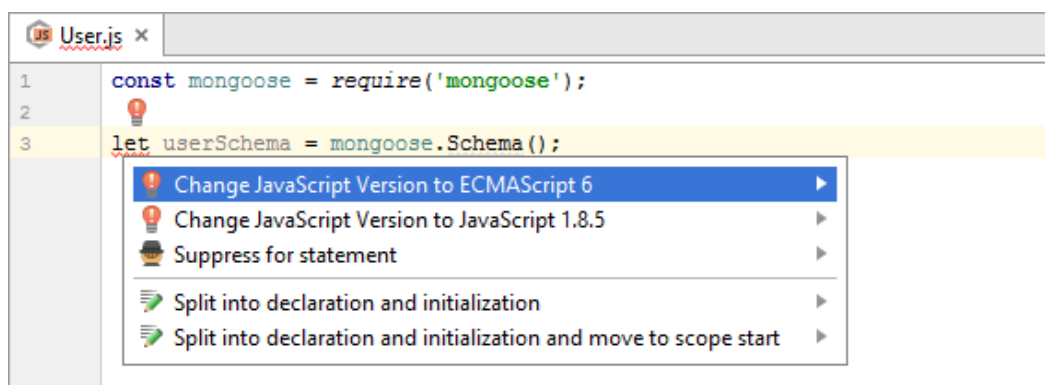


The screenshot shows a code editor with a file named 'User.js'. The first line of code is `const mongoose = require('mongoose');`. A context menu is open over the code, showing several options: 'Install 'mongoose'', 'Install 'mongoose' and add to dependencies in package.json' (highlighted), 'Install '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 file 'User.js'. The second line of code is `let userSchema = mongoose.Schema();`. The 'let' keyword is highlighted in red. A context menu is open over the code, showing several options: '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:

```

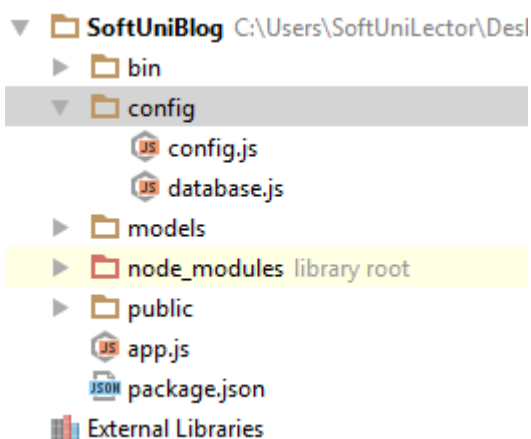
...
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

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.

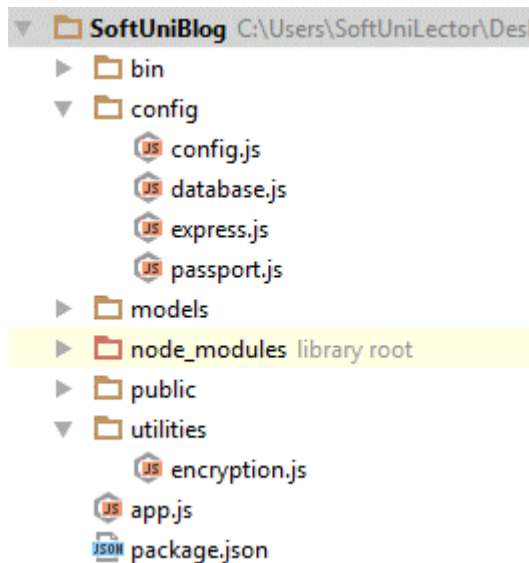
- **serializeUser** – given the whole **user** object **return** the **unique identifier**(in our case pass it to the “done” function).
- **deserializeUser** – given that **unique identifier** from the serialize function **return** the whole **user** object (again passed to “done” function).

```
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>
        <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;
```

SOFTUNI BLOG

REGISTERLOGIN

Register

Email

Email

Full Name

Full Name

Password

Password

Confirm Password

Confirm Password

Cancel

Submit

© 2016 - Software University Foundation

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

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

```
<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" />
```

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:

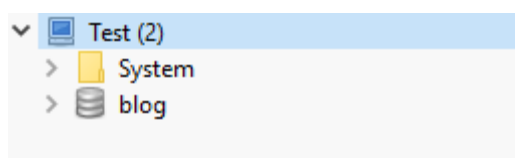
1. Open MongoDB connection – open Command Prompt and type “**mongod --dbpath “D:\test\example”**”. What this will do is: create server (locally) on some of the computer's port (default is 27017) and will wait for any contact (command).
2. Connect to the newly-created server: depends on whether you are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

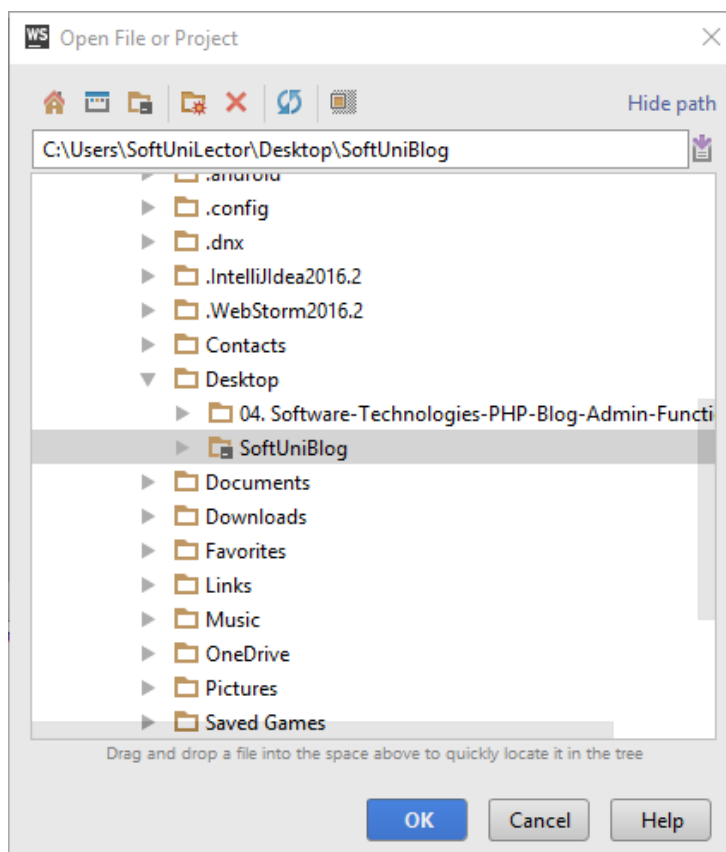
Summary: Now we know simple definition of a database. We saw different ideas behind implementing a database. Also how to start a MongoDB server from which we can create and manipulate different databases.

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

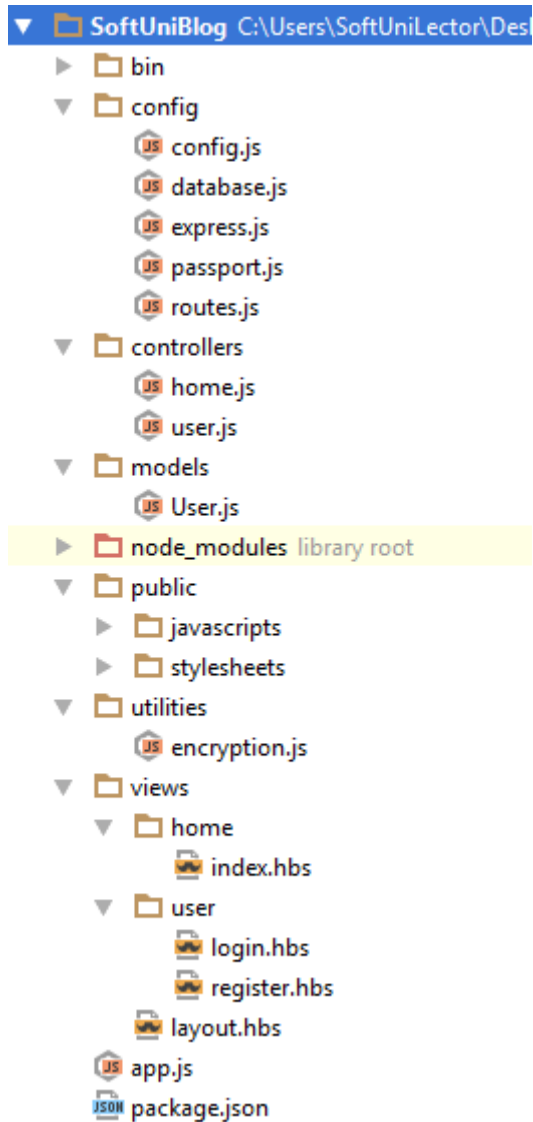


2. Open/Create project

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:

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.

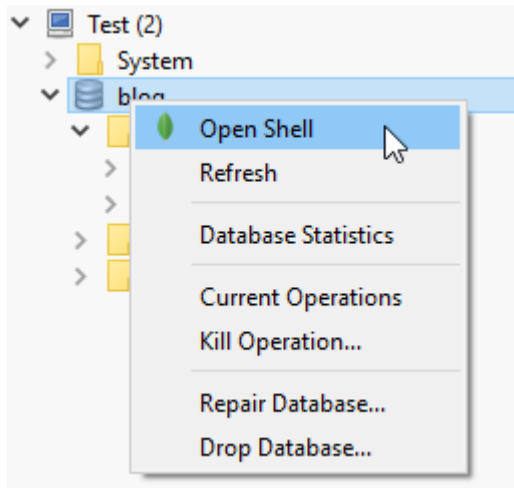
The whole [responsibility](#) of handling a migration is given to the programmer. There are some [frameworks](#) that might help us with that, but for the scope of our project it would be simply overkill to use one.

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.

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 (req.user) {
  res.locals.user = req.user;
}
```

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/{{article.id}}">Edit &raquo;</a>
            <a class="btn btn-danger btn-xs"
href="/article/delete/{{article.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 get from the server side. Providing the flexibility to display information for every article in the database.

VI. Coming Soon: Update, Delete Article and Authorization