# Blog: JavaScript and MySQL

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

Make sure you have already gone through the [Getting Started: JavaScript](https://softuni.bg/downloads/svn/soft-tech/May-2017/Software-Technologies-July-2017/08.%20Software-Technologies-JavaScript-Blog-Basic-Functionality/08.%20Software-Technologies-JS-Blog-Getting-Started-Lab.docx) guide. In this guide we will be using: [WebStorm](https://www.jetbrains.com/webstorm/) and [HeidiSQL](https://heidisql.com) GUI. The rest of the needed non-optional software is described in the guide above.

**Chapters from I to IV are for advanced users, but is recommended to be read. There’s a** [**skeleton**](http://svn.softuni.org/admin/svn/soft-tech/Jan-2018/Software-Technologies-Mar-2018/07.%20JavaScript-Blog-Basic-Functionality/07.%20Software-Technologies-JavaScript-Blog-Basic-Functionality-Skeleton.zip) **which you can use and start from chapter V.**

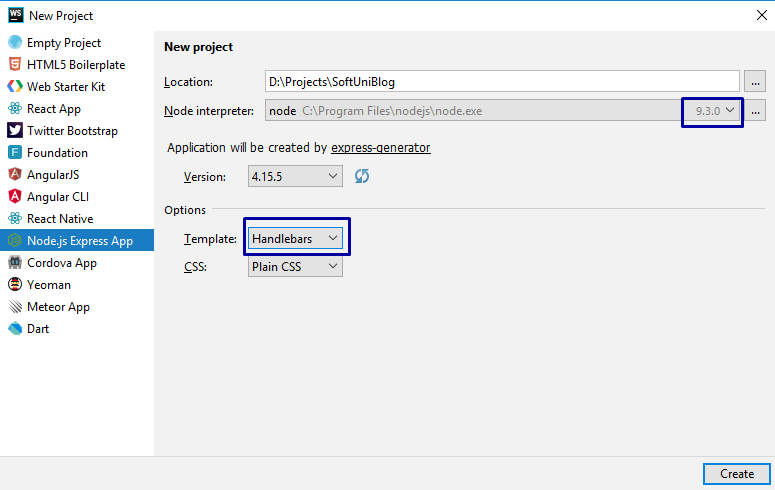
# Set Up Node.js Express Project

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

## Create the Project from your IDE

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





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

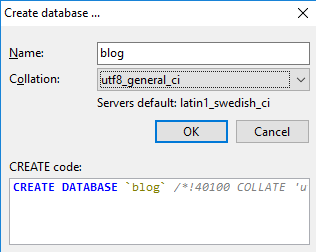
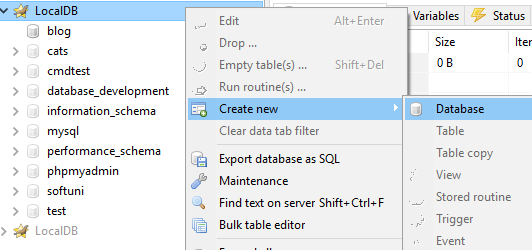
* + - * Also, choose the [Handlebars](http://handlebarsjs.com/) **Template**.
      * The recommended Express **versions** are any above **4.0.0**

## Create a Database

It’s time to create the database, which your app will use. We have two possible approaches here. Choose whichever one you prefer:

### Using the GUI

Open **HeidiSQL**, connect to the default instance (with port: **3306**) and create a database named “blog”. Don`t forget to start your MySQL Server via XAMPP.



### Using the Command Line

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

|  |
| --- |
| mysql -u root -p  //enter your password  CREATE DATABASE blog; |

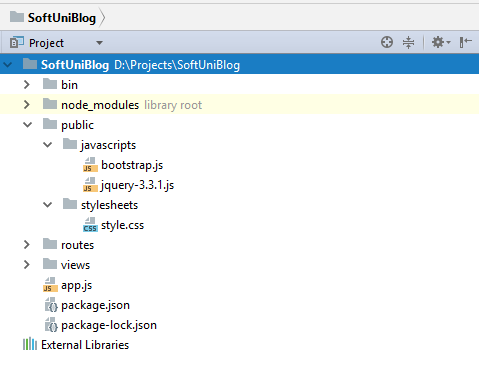
**Note** that in order to use command line you should have all **environment variables** set or if not, you should run the command line from the place where **mysql.exe** is (“in XAMPP directory”).

Once you set up your database, it’s time to get to work on the layout.

## 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](http://svn.softuni.org/admin/svn/soft-tech/Jan-2018/Software-Technologies-Mar-2018/07.%20JavaScript-Blog-Basic-Functionality/07.%20JS-Blog-Design.zip), where part of our **JavaScript** and **CSS** is included. In addition, we will need

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.

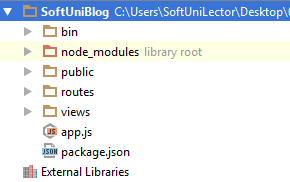
Setup a base layout as you wish or use the following one:

|  |
| --- |
| <!DOCTYPE **html**> <**html**> <**head**>  <**title**>SoftUni Blog</**title**>  <**link rel='stylesheet' href='/stylesheets/style.css'** />  <**script type= "text/javascript" src="/javascripts/jquery-3.3.1.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**.**fullName**}})</**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;** 2018 - Software University Foundation</**p**>  </**div**> </**footer**> </**html**> |

# Node.js Express App Base Project Overview

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

Initially the project comes with the following structure:



We can see several folders here. Let look at them one by one and see their content closely:

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

# User Authentication

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

## Create the User Entity

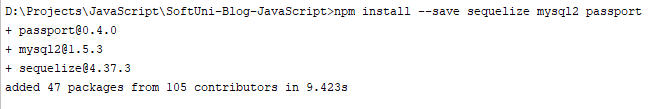
Our users should be stored in the database (**MySQL**). This means we need **U**sers table.

Let’s define rules for a user:

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

We won’t be using pure SQL. We will use Sequelize. Sequelize is a module that will make creating and manipulating collections easier. Open the console and install the following packages:

**npm install --save sequelize mysql2 passport**



**Now you can initialize Sequelize with the following command: sequelize init**

**There Is a chance you hit on an error. That sequelize is not a command. You are missing another module that you need to install globally – sequelize-cli. Just run the following command:**

****npm install -g sequelize-cli****

**Now you can run the command to initialize sequelize: sequelize init**

We got Sequelize generate some folders for us: models, migrations and seeders. You wont be needing seeders, nor the migrations folder, so you can safely delete them.   
There is also a config.json file, leave it for now, later we will create a config.js file to store all the configuration information.

JavaScript is a 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.

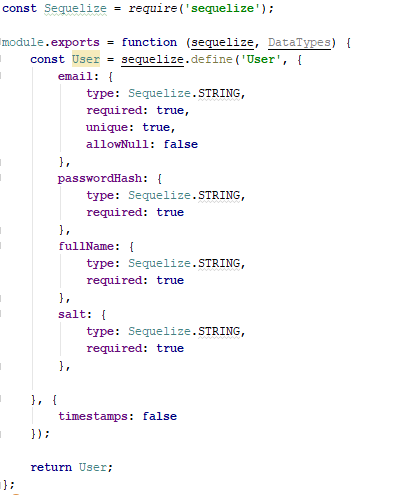
So if you check model/index.js file it has some mistakes, that we need to correct. In this file, we are requiring the modules we're going to be using. Then, we're reading the configuration specific to our current Node environment. If we don't have a Node environment defined, we're defaulting to development. Then, we are establishing a connection with our database, after which we read our models folder, discovering and importing any and all the models in it, adding them to the db object and applying relationships between the models, if such relationships exist. Since the generated models/index.js file is in ES5 syntax, we are going to refactor it to ES6 syntax. If you are not familiar with ES6 syntax, you can learn more about it, just google it.

The file should look like this:

|  |
| --- |
| const fs = *require*('fs'); const path = *require*('path'); const Sequelize = *require*('sequelize'); const basename = path.*basename*(module.filename); const env = *process*.env.NODE\_ENV || 'development'; const config = *require*(`${\_\_dirname}/../config/config.json`)[env]; const db = {};  let sequelize; if (config.use\_env\_variable) {  sequelize = new Sequelize(*process*.env[config.use\_env\_variable]); } else {  sequelize = new Sequelize(  config.database, config.username, config.password, config   ); }  fs  .*readdirSync*(\_\_dirname)  .filter(file =>  (file.indexOf('.') !== 0) &&  (file !== basename) &&  (file.slice(-3) === '.js'))  .forEach(file => {   const model = sequelize.import(path.*join*(\_\_dirname, file));  db[model.name] = model;  });  *Object*.keys(db).forEach(modelName => {  if (db[modelName].associate) {  db[modelName].associate(db);  } });  const models = *Object*.keys(db);  async function *create*(models) {  *console*.log('Initializing...');  await sequelize  .authenticate()  .then(function(err) {  *console*.log('\x1b[32m%s\x1b[0m','Connection has been established successfully.');  })  .catch(function (err) {  *console*.error('Unable to connect to the database!');  *process*.exit(1);  });  for (let i = 0; i < models.length; i++) {  const modelName = models[i];  try {  await db[modelName].sync();  models.splice(i, 1);  i--;  } catch (err) {  }  }  if (models.length > 0) *create*(models.slice()); }  *create*(models.slice());  db.sequelize = sequelize; db.Sequelize = Sequelize;  module.exports = db; |

Unfortunately, when we use "let" it gets highlighted in red. This is because we have to switch our JavaScript version to ECMAScript 6. Press [Alt + Enter] to popup the helper, then click [Enter] and everything should be fine.

Now create a file models/user.js. Here we are going to define our User:



Here is what our definition should look like. We create definition by using that sequelize module we already imported. The define function accepts a JavaScript [object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures). In plain words the above means: we will create a model where every entity will have:

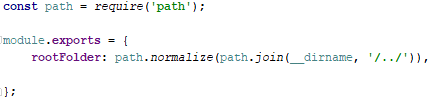
* email
* passwordHash
* fullName
* salt (will explain it later)

They all have the String type and they are all required. For more info on types in JavaScript read this [article](http://www.w3schools.com/js/js_datatypes.asp).

To finalize creating the **User** model, there are two things left to do: create and export a model. This means that every time someone requires our “user.js” file he will get the User model. Sequelize is going to provide all that. We can call our model with models.User.

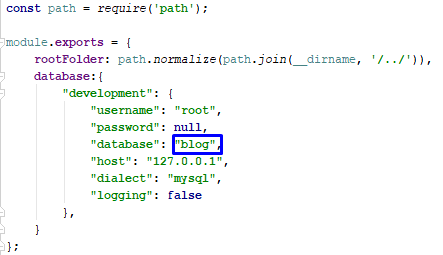
## Create Connection with MySQL

Before we start setting up our connection with database let’s create a config.js file in our config folder (configception).



Then just add another parameter 'database' and copy paste the content of config.json. You can take only the development part, because we are going to use it.

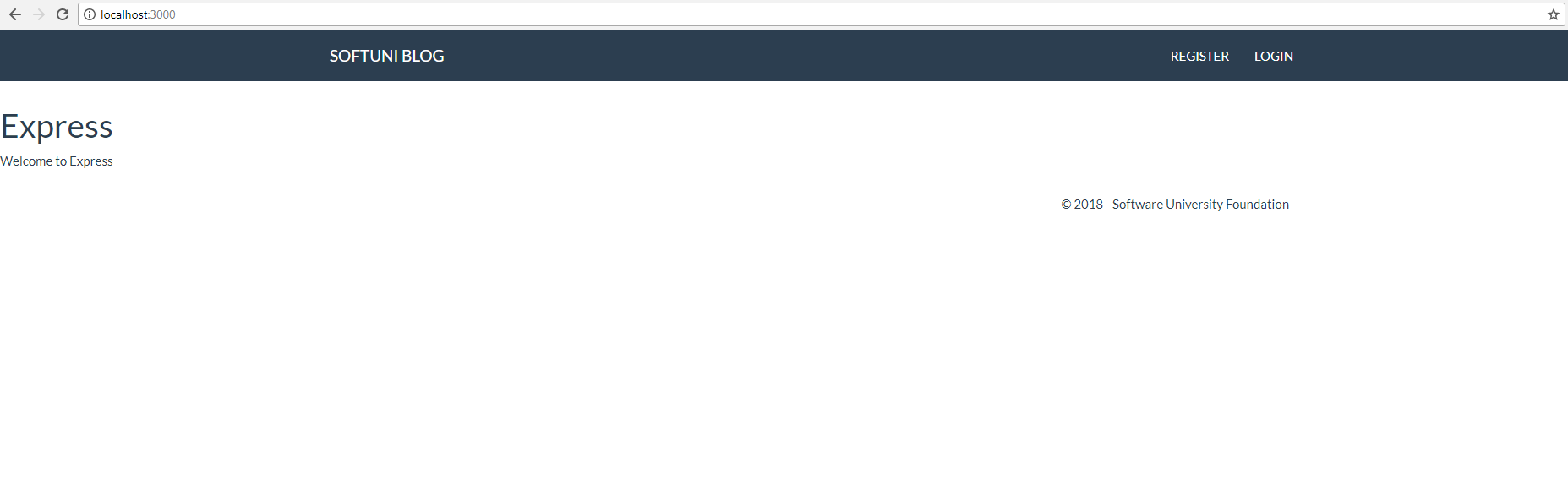
There, we will store information, which is needed to connect with our **database** (MySQL). Don’t forget to change the development database to **"blog",** the db we created before.



Now go back in our index.js in folder models and change the way it calls the database, because we deleted config.json. It should look like this:



We can check what we have done by now. Start the app and go to localhost:3000;



We can see that the style is loaded properly so we can continue developing our blog.

## Set Up the Security Configuration

We have our model ready. Now we need to create some **security configuration**.

First, create a folder named “**utilities**”. Inside of it, create a file named: “encryption.js“. There will be our logic for generating [salt](https://www.addedbytes.com/blog/why-you-should-always-salt-your-hashes/) and **hashing** our **password**. So, we’ll create two **functions** and make them **public** so they can be useful.

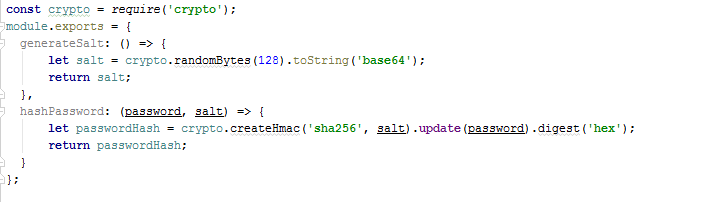
First, we will need some helper module (“crypto”). In order to make the functionality visible to the **outer world**, we will **export** an **object** which will have **two properties** which are [JavaScript](https://cdn.meme.am/cache/instances/folder522/500x/63119522.jpg) **functions**.



Next, let’s define the two functions (generateSalt and hashPassword) inside the module.exports statement:



Next, let’s fill in the relevant functionality for each function:



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

Create an “**express.js**” file in the “**config**” folder. We will put some **setup logic** inside it. Simply copy the “**app.js**” file and **remove** 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 makes the content in the "public" folder accessible for every user.* app.use(express.static(path.*join*(config.rootFolder, '/public')));   *// 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*('pesho'));   *// Session is storage for cookies, which will be de/encrypted with that 'secret' key.* app.use(*session*({secret: 'pesho', resave: false, saveUninitialized: false}));   *// For user validation we will use passport module.* app.use(passport.initialize());  app.use(passport.session());   app.use((req, res, next) => {  if (req.user) {  res.locals.user = req.user;  }  next();  });  }; |

Let’s talk about the modules we are using:

* express – wraps functionality that the **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 **encrypted data** about the **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 an object with different properties. See the [documentation](https://github.com/expressjs/body-parser).
* express-session – server-side **storage**. With that “**secret**” string, we can differentiate cookies from one another (by setting an ID to every cookie). It also keeps information about the **current user’s** **connection**. **Only for development uses**.
* passport – **security module** that uses the **session** to **save information about the user**. It requires a saving strategy (“**Facebook**”, “**Google**”, “**Local**”, etc.) and tells us which data from the user we’re going to put in the **cookie**. It binds **two functions**: logIn and logOut.

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

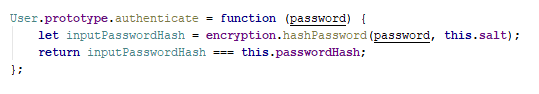


As you see, we have declared a function to authenticate a user by their **username** and **password**. This means that first, the **username** should exist in the database and second - the given **password** should be **equal** to the one in the **database** (**hashed** of course).

Additionally, our function receives a **third argument** called “**done**” – another function which will be invoked **inside** the current function.

The logic behind that is to pass any **errors** (if any have occurred) as the **first argument** and **the user** as the **second argument**. If we can’t authenticate the user, we return the errors in the first argument, and if we can, we just pass **null**. Otherwise, we return the **user**. This logic is needed to implement the 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 to log in).

Here we use authentication method from the **User’s** model**(user.js).** 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 **definition**:



The passport module will provide us with two functions (as said above) which means that it **automatically** takes care of **login/logout**. However, the input data may be called differently than “**email**” and “**password**” (aka in our **html** form the **input fields** can be **named differently**), therefore 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 to do that, we should tell it how to **differentiate users**.

The code for this looks like this:

|  |
| --- |
| 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);  })  }) }; |

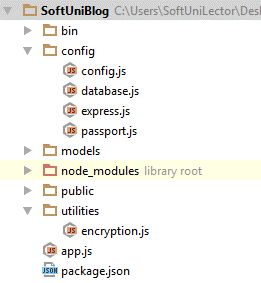
Let’s break down what it does:

* serializeUser – we give it the **user object** (user variable), and a donefunction. If we don’t have a user, we call done with **null**, and if we do, we return the user’s **id**.
* deserializeUser – we give it the **user id** (user variable), and a donefunction. If we don’t have a user, we call done with **null**, and if we do, we return the **user** as an **object**.

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 in the end:



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



## Register User

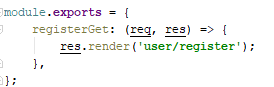
Now that we have our **authentication** strategy and entity **model**, let’s start creating some **views** to **register** our first user!

Create a “**user**” folder in the “**views**” 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**> |

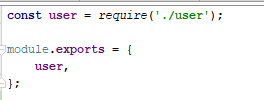
Now, after we have our user registration view, let’s create a **controller** to render it.

Let’s create a folder named “**controllers**”. Inside it, create a file called 'index.js' and another one 'user.js'. We will put everything all the logic, which will manipulate our User model there. **Add a function**, which will render the **html** passed above:

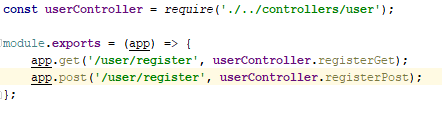


Our function in the controller will receive the request and response as parameters.

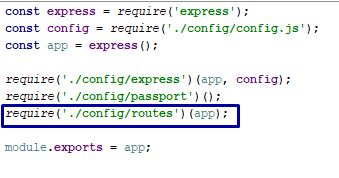
In out 'index.js'we should export all of our controllers.



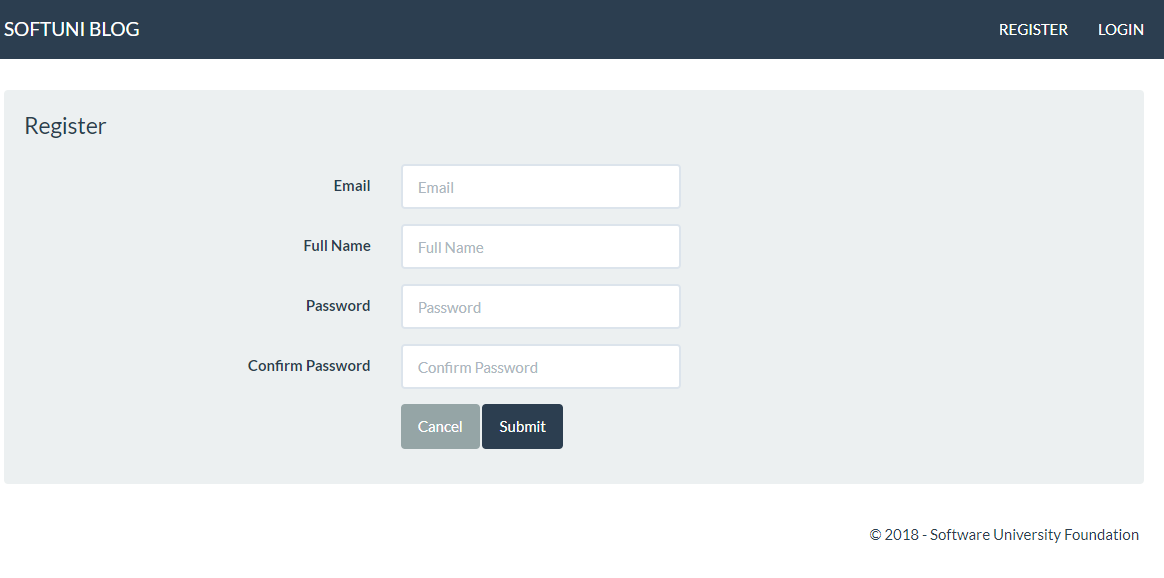
What we need now is to define routes (routes will say which controller when to be called). The logic of routes is simple and lay on [REST](http://www.andrewhavens.com/posts/20/beginners-guide-to-creating-a-rest-api/) API definition. Let’s **delete the routes folder** we have and create a “**routes.js**”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



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



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



In case there are some errors, don`t worry, probably you need to install some modules like express-session, just npm install them.

Our form got displayed (using a **GET** request)!

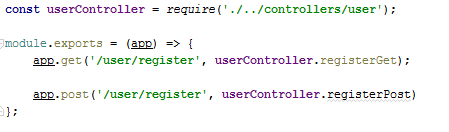
Let’s dive deeper in “**user/register.hbs**”. If we look at the <form> tag, we can see that the form has **two attributes: (key 🡺 value)** pairs:

* **method 🡺 post**
* **action 🡺 /user/register**

This simply means that whenever this form is submitted (aka the button of type “submit” is clicked). It will create a POST request towards the URL described above:



This means that we need to **create a new route** with **same URL**, but **different HTTP method (**POST**)**:

First, add the route to the in “**routes.js**” file:  


Second create a new action in the User’s controller. That action should do the following:



We need to 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). If we look at “**user/register.hbs**”, we can see that every input field has a name attribute (name="email" and so on):



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

Second, we need to write our **validation logic**, which needs to answer **two questions**:

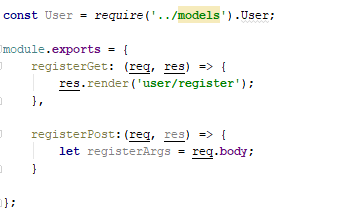
* Is the given email **already registered**?
* Do **both passwords match**?

If the answer to both of those questions is “**yes**”, only then can we register the user.

Before we can answer the questions, we need to figure out how we can search the models for data. Luckily, **Sequelize** gives us this functionality. All we need to do is require() the **User** model and then just use Sequelze’s built-in methods:



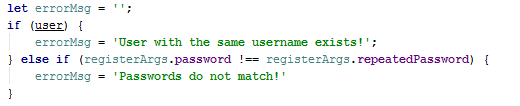
At this point, our user.js controller should look like this:



Now, back to validation. Let’s answer the validation questions. To do that, we need to **connect to our database** and **check** if there is **already a user with that email**. This can be done by using the Sequelize findOne() function. This command accepts an **object**, which we can use as a **filter**: "where:{email: registerArgs.email}"



Now, after we’ve found the user, let’s **validate** if the **user exists** and if the **passwords match**:



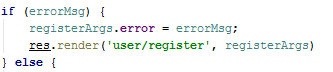
For every error case, we will create a **string** variable in which we’ll save the error message. Note that **JavaScript** is **weird** when speaking about **truthy** and **falsy** values. Read this [article](http://james.padolsey.com/javascript/truthy-falsey/) for further clarity.

Now, let’s analyze what the logic in the above screenshot does:

If the user variable returns anything, that’s **bad**, because it means there’s **already a user with that email** in the **database**. If there weren’t, it would have returned **null**.

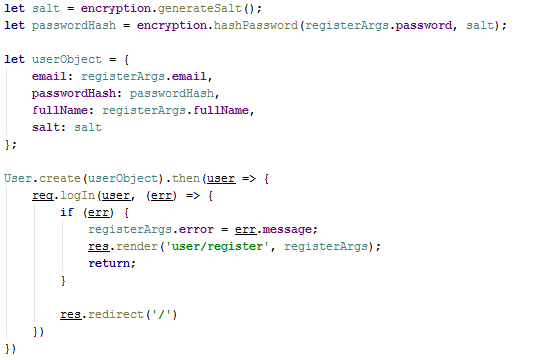
We can convert the **user** variable to true/false by just putting it in an ifstatement. An undefined value would be converted into false and an existing user, (which isn’t undefined) into true.

After we have our validations, we should check for any **errors**:



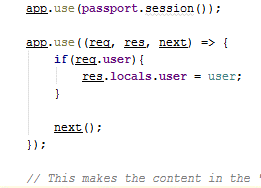
If any errors occurred, we will simply **reload** **the page**. The key thing here is that we will reload it **with the previous values** (so that the user doesn’t have their form cleared), and with the **error message**. The error message will be **displayed** in the layout (“**layout.hbs**”).

On the other hand, if we **didn’t have any errors**,we should insert a **new entity** in the database and **log** the user **in**:



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

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



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

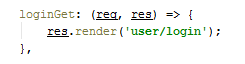
## Login Form

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

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

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

Then add the **action** in the **user controller**:



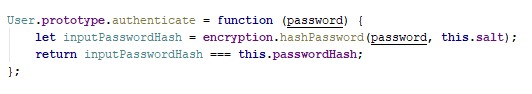
After that, extend the “**routes.js**” file with our **login** **get** action:



It’s time to go back to the **user** **controller** and create the **login logic**. Just like with registration, we need to answer a couple **validation questions** before we let the user log in:

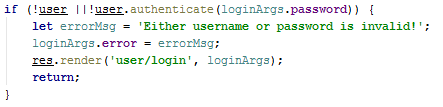
* Does the user exist?
* If so, does the **password** (when hashed) they gave us match the **hashed** **password** given in the input?

The easiest way to do that is to give every **User** a **validation function**. This is the easiest way because the users have all the needed information (**salt** and **passwordHash**). Go to the **User.js** in “**models**” folder and add this block of code:

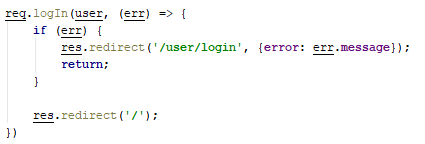


Make sure that this **method** appears **before** the one, which creates the User’s **model** assosiations.

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

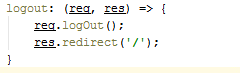


So, we have some **validation** on the input, what left is to actually **log** the **user in**. You may use the **same** logic **as** we used in the **registration** section.

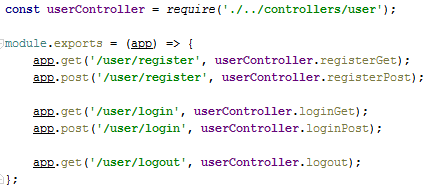


## Logout

Logging out is very simple. Let’s add it in the **user controller**:



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

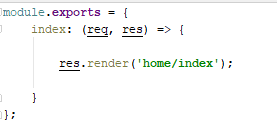


# Creating home controller

## Create home controller

As we changed the logic behind our routing, we cannot get our index page.   
So, we need to create another controller -> home.js

It will be responsible to control all of the routing for requests, not connected to our models.



Let`s move our index view in a separate folder, called home.

Now go to routes and add the get index request, you should require home controller so you can use it.

Now we see, that our index page is still using that express generated html.



Let`s change the content of the home page, just replace the current content with 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.User.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> |

## Refactor bin/www

Right now, our bin/www file us outdated, still, you can use the current one, generated when we started the project, if you want you can refactor it by yourself, or you can use this one:

|  |
| --- |
| #!/usr/bin/env node /\*\*  \* Module dependencies.  \*/ const app = require('../app'); const debug = require('debug')('softuniblog:server'); const http = require('http'); const usedPort = '8000';  /\*\*  \* Get port from environment and store in Express.  \*/ let port = normalizePort(process.env.PORT || usedPort); app.set('port', port);  /\*\*  \* Create HTTP server.  \*/ const server = http.createServer(app);  /\*\*  \* Listen on provided port, on all network interfaces.  \*/  server.listen(port); server.on('error', onError); server.on('listening', onListening);  /\*\*  \* Normalize a port into a number, string, or false.  \*/  function normalizePort(val) {  const port = parseInt(val, 10);   if (isNaN(port)) {  // named pipe  return val;  }   if (port >= 0) {  // port number  return port;  }   return false; }  /\*\*  \* Event listener for HTTP server "error" event.  \*/  function onError(error) {  if (error.syscall !== 'listen') {  throw error;  }   const bind = typeof port === 'string'  ? 'Pipe ' + port  : 'Port ' + port;   // handle specific listen errors with friendly messages  switch (error.code) {  case 'EACCES':  console.error(bind + ' requires elevated privileges');  process.exit(1);  break;  case 'EADDRINUSE':  console.error(bind + ' is already in use');  process.exit(1);  break;  default:  throw error;  } }  /\*\*  \* Event listener for HTTP server "listening" event.  \*/  function onListening() {  const addr = server.address();  const bind = typeof addr === 'string'  ? 'pipe ' + addr  : 'port ' + addr.port;  console.log(`Listening on http://localhost:${port}`); } |

# Creating Articles

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

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

We have a couple of **tables**, each of them has some **columns** which give us the opportunity to **store** **data in them**. This is what a typical SQL database looks like:

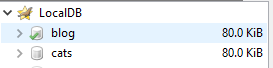


Now you can communicate with the database and execute [commands](https://docs.mongodb.com/v3.2/reference/command/).

You can create a database named “**blog**”.

Summary: We now know the simple definition of a database. We saw different ideas for implementing a database. Also, we learned how to start a MySQL server from which we can create and manipulate different databases.

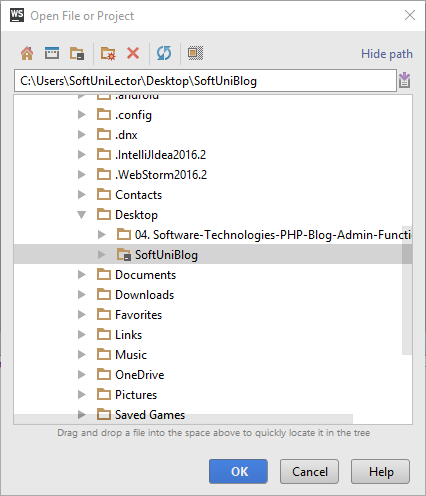
Here is how your connection **might** look like:



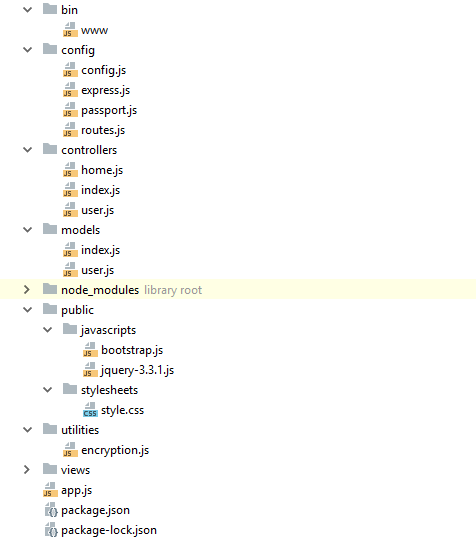
## Open/Create project

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





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



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

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

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

## Create the Article Model

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

* title
* content
* author
* date

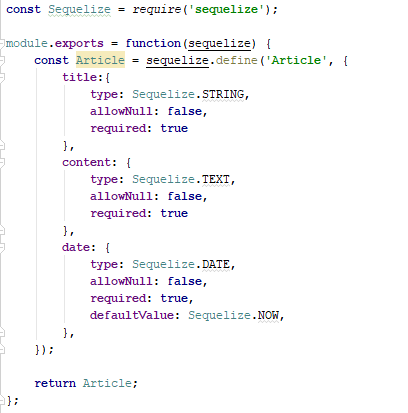
The interesting property here is the **author** property, because it is **already** a **model** in our database. Imagine that if every time we created an **article**, we **bound** that author’s **information** to the article. What happens when one **author** creates **50** **articles**? Would we store the same **author name** and **other** **properties** for every single one? Wouldn’t that be a huge waste of **memory**? Yes, so how can we resolve that problem?

The answer is what’s called a **reference** key (something **unique** for the author – like an **ID** or a **name**). When our author has an **ID**, instead of binding all the information to the article, we just **save** that author **ID** in the article.

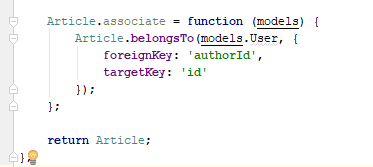
But how do we retrieve information about a specific **article’s author**, you ask. Well, we can always just **look them up** in the database, using the **ID**, which we saved inside the article. This is called (database) **relations**. One author has zero or many articles. We will cover on that more in the next chapter.

Let`s create another file in our models' folder => article.js.

Let`s write some code, we need to export the model we are going to create. We are using the sequelize way of defining models. Lets add required and allowNull to every attribute. And we can set a default value of our date attribute, so every time we create an article, the current date and time is stored.



But what about the author. Let`s add association for the Articles to the Users. User will be the author of many articles. So, we create ManyToOne association.

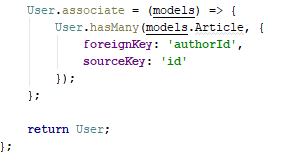


**Summary**: we now know how to create an entity, wrap it in a model and define a relation with another model.

## Create Author – Article Relationship

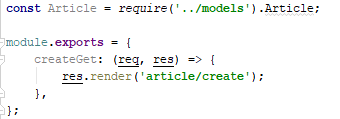
Our **program** is like our real world – it is **based** on **connections** and interactions between it’s elements. We have a **user** which has **zero** or **more** **articles**. This relation is called **one** to **many**. We might want the **articles** to have **tags**. Many articles with many tags. 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**.

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 definition. In database world this is called **Association**. Let’s do the migration in the **user’s definition**:



Summary: a database **relation** defines **connection** between two entities. The **relation** type depends on the point of **view**.

## Create the Article Controller

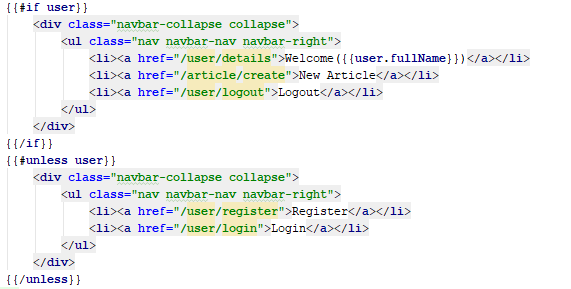
The next part will be creating the **article controller** where we will put all the logic connected directly with the **Article** model. Create an “**article.js**” file in the “**controllers**” folder. As a starter, we want to create a method which will render the form for creating an article. The controller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

Important thing about this way is that the code, initializing the Article model must be compiled before we try to access the model.

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

## Templating Article Form

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



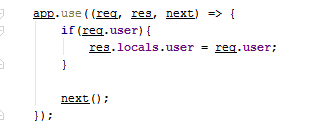
We can see that there is a lot of html but also there are multiple blocks of code which are not. These parts are for the **view engine**. Let’s explain what it does to the code.

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 the current example, we can see that we have an **if** statement (#if). If the **variable** passed to the “if” is **truthy**, all the html until the **{{/if}}** will be displayed.

Okay, but what if the variable is **falsy** and we want to display something different? We will use “**unless**” in the same way that we used “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 – it should be somewhere after **passport.session()**):



Another **thing** to mention: look at the first **<a> tag** – there is a block “{{**user.fullName**}}”. 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:

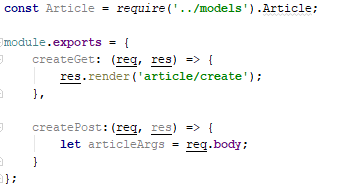
|  |
| --- |
| <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" method="POST" action="/article/create"**>  <**fieldset**>  <**legend**>New Article</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="articleTitle"**>Article 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](https://www.tutorialspoint.com/http/http_methods.htm) of the request will be, and **action** – the actual link where we want the data to go. So, wherever this form is submitted the request will go where the **action** attribute points.

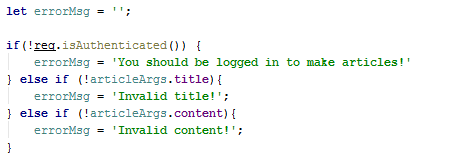
**Summary**: The **view engine** helps us **put** **logic** in our views and also helps us **display** even **more** information. The best thing here is that that logic can be put **directly** **into** our **html** code. ☺

## Finalize Article Creation

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

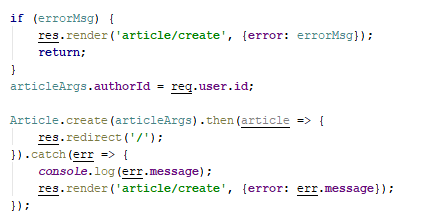


This is how the article controller should look for now. We have our article data parsed so start making some **validations**:



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

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



Here’s how the control flow goes:

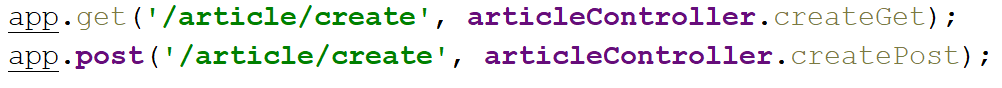
* If there are **any** **errors**, we will:
  + **Re-render** the same page, but this time, we’ll pass an object **with** an “**error**” property, whose textwill be displayed in the “**layout.hbs**”.
* If there **isn’t any error**, we will:
  + **Assign** the **author’s** **id** to the **article** object

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).

Now we need to **add** the routes to our “**routes.js**” file. First require the articleController:



And **attach** **functions** to the POST and GET methods on the **“/article/create**” URL:

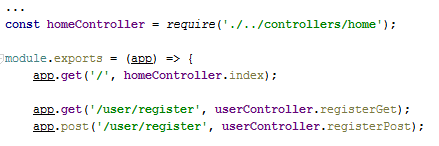


***If you skipped creating the home controller and the index view****:*

**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'**);   } }; |

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



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

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

# Read Articles

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

## List Articles

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

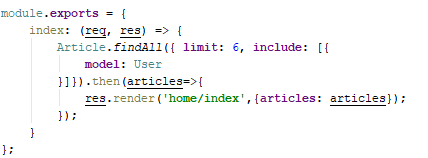
|  |
| --- |
| <div class="container body-content">  <div class="row">  {{#each articles}}  <div class="col-md-6">  <article>  <header>  <h2>{{this.title}}</h2>  </header>   <p>{{this.content}}  </p>   <small class="author">  {{this.User.fullName}}  </small>   <footer>  <div class="pull-right">  <a class="btn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

Here, we use Handlebars’ full strength. We are using an “**each**” construction (which works the same way as foreach in some languages).

We go through **every article** which was passed to us. For every single one we will display:

* its **title** (using **this** means that we are iterating over the **current** **article**)
* its **content**
* its **author**

The interesting part here is that we pass this statement: “**this.User.fullName**”. Remember when we created the Article **model**? We created a relation with the user? Here comes the cruicial point in getting the whole information (from our relation). Let’s see how we will get that information from the “**home**” controller:



What this will do is:

* Retrieve all the articles
* Get **only the first** **6** of them
* Include in the object the User, associated with each article.
* Send them to the “home/index” view.

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

Here is how the article should appear in our homepage:



Summary: We now know how we can **iterate** over an **object** in our **view** engine.

## 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.

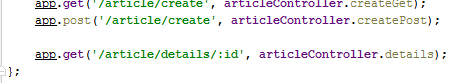
Again, our first step is to **generate the view**. This means that we have to create a file named “**details.hbs**” in our “**views/article**” folder:

|  |
| --- |
| <**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"**>  {{**this**.**User**.**fullName**}}  </**small**>   <**footer**>  <**div class="pull-right"**>  <**a class="btn btn-default btn-xs" href="/"**>Back **&raquo;**</**a**>  </**div**>  </**footer**>  </**article**>  </**div**>  </**div**> </**div**> |

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

Whenever we want to so see the **details** of an article, it’s only logical to **tell the server** which article we want to see. This information will be sent through the **URL**. 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** by its **id** and then **pass it** on to the **view engine**.

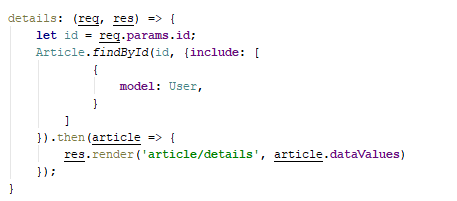
How do we get the information from the link? We will use req.params. But first let’s look how our routing will look in “**routes.js**”**:**



Just add the “**/article/details/:id**” part. This means that in the end of our link, we are expecting a **parameter** named “**id**”. Later on when using req.params, we can **access** that parameter by just **getting** its name as a property of the req.params object.

So, if we want to get a parameter with the name “**id**”, we will write req.params.id. This is how we get **parameters** from our **URL** (*note: req.params is different from req.body*).

Let’s implement the details page in **article** **controller** (**article.js**):



Now everything should be implemented and we should be able to get an article’s details when we press the **read more** button next to an article!

**Summary**: We saw how to **display** more **detailed** **information** about an **article**. We **passed** the needed **parameters** in our **URL**, which we can easily work with from the **server’s side**. This way we can get **information** about an **article** by just giving the server its **id**.

# **Continue the project**

Congratulations, you’ve just implemented a basic blog system, using Note.JS as a **server**, ExpressJS as a router, Sequelize as an **ORM**, MySQL as a **database** and Handlebars as a **view engine**! Feel free to implement any additional functionality using your newfound knowledge (try to implement profile view for every user || add comments ). Happy coding! ☺