# **Blog: JavaScript and MySQL**

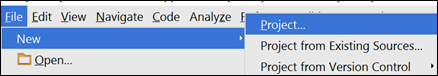
Този документ дефинира пълен пътеводител за създаване на приложение за блог с Express.js Framework от настройването на рамката чрез модула за удостоверяване, завършвайки с създаването на CRUD около MySQL обекти, използвайки Sequelize модел на обект-документ модел.  
Уверете се, че вече сте преминали през ръководството за начинаещи: JavaScript. В това ръководство ще използваме: WebStorm и HeidiSQL GUI. Останалата част от необходимия софтуер, който не е задължителен, е описан в ръководството по-горе.  
Главите от I до IV са за напреднали потребители, но се препоръчва да бъдат прочетени. Има скелет, който можете да използвате и да започнете от глава V.

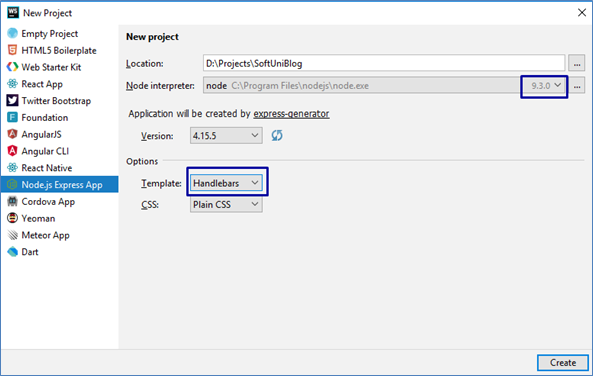
# I. Set Up Node.js Express Project

WebStorm идва директно с структурата на проекта плюс не е нужно да изтегляме никакви приставки, за да разработим нашето приложение Node.js / Express.js

## 1. Create the Project from your IDE

1. Създайте проекта от вашия IDE  
След като инсталирате плъгините и стартирате IDE, ще имате в контекстното меню "Създаване на проект" или "Node.js и NPM" -> "Express приложение Node.js" (IntelliJ с плъгин Node.js), или директно " Експресно приложение "Node.js" (WebStorm)



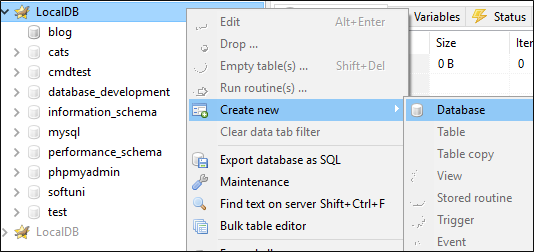


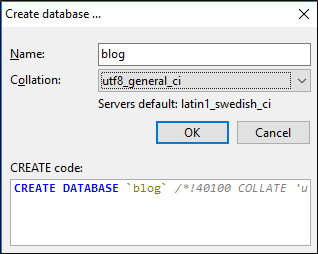
Уверете се, че имате инсталиран интерпретаторен възел и избраната директория е правилната.  
• Също така изберете Шаблона за дръжките.  
• Препоръчителните версии на Express са над 4.0.0

## 1. Create a Database

2. Създайте база данни  
Време е да създадете базата данни, която приложението ви ще използва. Тук имаме два възможни подхода. Изберете това, което предпочитате:

### Using the GUI Използване на GUI Отворете HeidiSQL, свържете се с по подразбиране (с порт: 3306) и създайте база данни с име "блог". Не забравяйте да стартирате MySQL сървъра си чрез XAMPP.





### Using the Command Line

Използване на командния ред  
Или ако искате да го направите, използвайте командния ред, използвайте следните команди:

mysql -u root -p

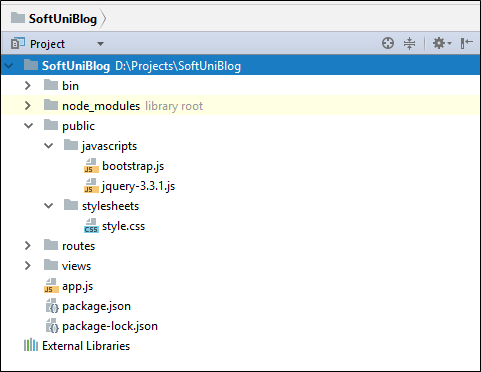
//enter your password

CREATE DATABASE blog;

Забележете, че за да използвате командния ред, трябва да имате всички променливи на средата, или ако не, трябва да стартирате командния ред от мястото, където mysql.exe е ("в директорията XAMPP").  
След като настроите базата данни, е време да започнете работа по оформлението.

## 3. Setup Layout 3. Разпределение на настройките

## Ще се нуждаем от основно оформление за всички наши шаблони. Тъй като използваме Bootstrap, ще се нуждаем от CSS, включен във всички страници, и свързаните с тях скриптове. Можем да изтеглим примерния скелет за дизайн на блог от тук, където част от нашия JavaScript и CSS е включена. Освен това ще ни трябва Всички наши стилове и скриптове трябва да включим в нашия проект. Трябва да добавим стилове в публичните стилове и публичните си скриптове в публични / javascript. Ще добавим тези две библиотеки, когато имаме нужда от тях:

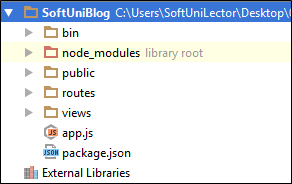


След това трябва да използваме тези стилове и скриптове, за да създадем базово оформление в изгледите / layout.hbs.  
Настройте основното оформление според желанието си или използвайте следното:

|  |
| --- |
| <!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**> |

# II. Node.js Express App Base Project Overview

Node.js е платформа, написана в JavaScript и осигурява функционалност отзад. Експрес е модул (за момента можем да свържем модула като клас, който осигурява известна функционалност), който обвива Node.js по начин, който прави кодирането по-бързо и по-лесно и е подходящ за MVC архитектура.  
Първоначално проектът е със следната структура:



Тук можем да видим няколко папки. Позволете да ги разгледате един по един и да видите тяхното съдържание отблизо:  
• **bin** - съдържа един файл, наречен www, който е отправната точка на нашата програма. Самият файл съдържа необходимата логика за конфигуриране, за да може локално да се стартира сървърът.  
• **node\_modules** (корен на библиотеката) - доколкото се казва в тази папка, поставяме всяка библиотека (модул), от която зависи проектът ни.  
• обществено - тук идва интересната част. Всичко, което е в нашата публична папка (файлове, изображения и т.н.) ще бъде достъпно за всеки потребител. Покриваме по-късно.  
• **routes (**маршрути) - папка, в която ще поставим конфигурациите на маршрутите си. За да стане ясно: маршрутите са отговорни за разпространението на работата в нашия проект (например, когато потребителят се опита да влезе в "www.oursite.com/user/login", за да се обади на конкретен контролер или модул, който отговаря за показването на данните за вход). По-късно ще заменим папката с routes.js в нашата конфигурационна папка.  
• **views (**изгледи) - както в предишния блог (PHP), отново имаме папка с имена на изгледи. Там ще запазим изгледите за нашия модел. Отново ще използваме шаблони с помощта на двигателя за преглед на дръжките.  
• **app.js** - скрипта, който съдържа нашата сървърна логика.  
• **package.json** - файл, съдържащ информация за проекта (като име на проекта, лиценз и т.н.). Най-важното е, че има част от "зависимостите", в която са всички имена и версии на всеки модул, който използват нашите проекти.

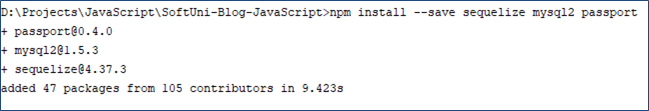
# III. User Authentication

Трябва да въведем самоличността на потребителя от себе си. Надяваме се да използваме някои модули за сигурност, които да ни помогнат с това. Но първо нека започнем с нашата потребителска единица.

## 1. Create the User Entity

1. Създайте потребителското лице  
Нашите потребители трябва да се съхраняват в базата данни (MySQL). Това означава, че имаме нужда от таблицата "Потребители".  
Нека дефинираме правилата за даден потребител:  
• Трябва да има уникално име за вход, да речем имейл  
• Трябва да има парола (която ние няма да запазим в нейния чист изглед)  
• Трябва да има пълно име, да речем fullName  
Няма да използваме чист SQL. Ще използваме Sequelize. Sequelize е модул, който ще улесни създаването и манипулирането на колекциите. Отворете конзолата и инсталирайте следните пакети:

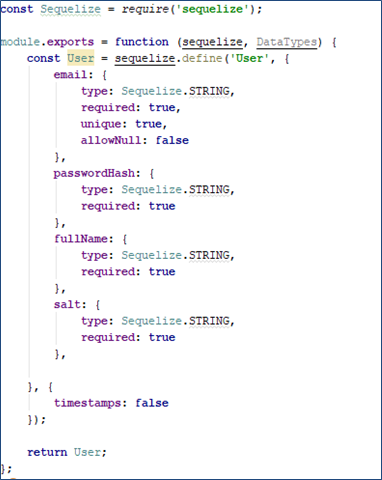
npm install --save sequelize mysql2 passport



Сега можете да инициализирате Sequelize със следната команда: sequelize init  
Има шанс да ударите грешка. Това продължаване не е команда. Липсва друг модул, който трябва да инсталирате глобално - продължение на кли. Просто изпълнете следната команда:  
npm install -g продължаване-кли  
Сега можете да стартирате командата, за да инициализирате продължаването: продължете init  
Имаме Sequelize генериране на някои папки за нас: модели, миграции и сеялки. Няма да се нуждаете от сеялки, нито от папката за мигриране, за да можете безопасно да ги изтриете.  
Има и файл config.json, оставете го за сега, по-късно ще създадем файл config.js, за да съхраните цялата конфигурационна информация.  
JavaScript е динамично написан език. Видът на нашите променливи се определя, когато проектът се изпълнява. Това се нарича JIT (или компилация Just In Time). Ето защо този език е бавен в сравнение с C ++ и дори C # / Java. Имаме няколко ключови думи, за да декларираме и инициализираме променлива (var, let и const - и не използвайте var - просто не). Използвайте const, когато създавате константна стойност и я оставяте за всяка друга употреба.  
Така че, ако проверите файла модел / index.js, има някои грешки, които трябва да коригираме. В този файл изискваме модулите, които ще използваме. След това четем конфигурацията, специфична за текущата ни среда на възли. Ако нямаме дефинирана среда на възел, ние сме в неравностойно положение. След това създаваме връзка с нашата база данни, след което четем папката на нашите модели, откриваме и импортираме всички и всички модели в нея, добавяйки ги към db обекта и прилагайки взаимоотношенията между моделите, ако такива взаимоотношения съществуват. Тъй като генерираните модели / index.js файла са в синтаксиса ES5, ще го презаредим на ES6 синтаксис. Ако не сте запознати със синтаксиса ES6, можете да научите повече за него, просто го.  
Файлът трябва да изглежда така:

|  |
| --- |
| 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; |

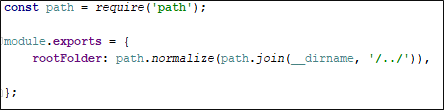
За съжаление, когато използваме "позволи", той се маркира в червено. Това е така, защото трябва да преминете нашата версия на JavaScript към ECMAScript 6. Натиснете [Alt + Enter], за да изведете помощника, след което щракнете върху [Enter] и всичко трябва да е наред.  
Сега създайте файл models / user.js. Тук ще определим нашия потребител:



Ето какво трябва да изглежда нашето определение. Ние създаваме определение, като използваме този модул за продължаване, който вече сме импортирали. Дефиниращата функция приема обект на JavaScript. С обикновените думи горното означава: ще създадем модел, в който всяко лице ще има:  
• електронна поща  
• passwordHash  
• пълно име  
• сол (ще го обясня по-късно)  
Всички те имат тип Струнни и всички те са задължителни. За повече информация за типовете в JavaScript прочетете тази статия.  
За да финализирате създаването на потребителския модел, има две неща, които трябва да направите: да създадете и експортирате модел. Това означава, че всеки път, когато някой изисква нашия "user.js" файл, той ще получи потребителския модел. Sequelize ще предостави всичко това. Можем да наричаме нашия модел с models.User.

## 2. Create Connection with MySQL

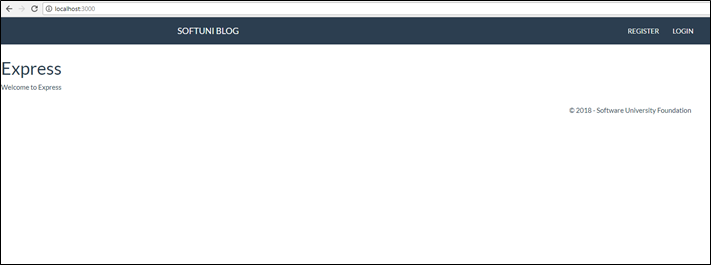
2. Създайте връзка с MySQL  
Преди да започнем да настройваме връзката си с база данни, нека създадем файл config.js в нашата конфигурираща папка (configception).



След това просто добавете друг параметър "база данни" и копирайте вмъкнете съдържанието на config.json. Можете да вземете само частта за развитие, защото ще го използваме.  
Там ще съхраняваме информация, която е необходима за връзка с нашата база данни (MySQL). Не забравяйте да промените базата данни за разработки на "блог", който създадохме преди.



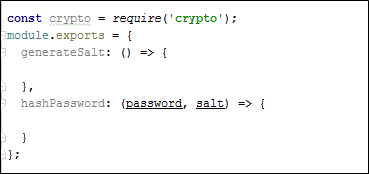
Сега се върнете в нашия index.js в моделите на папките и променете начина, по който той нарича базата данни, защото сме изтрили config.json. Трябва да изглежда така:  
 Можем да проверим какво сме направили досега. Стартирайте приложението и отидете на localhost: 3000;



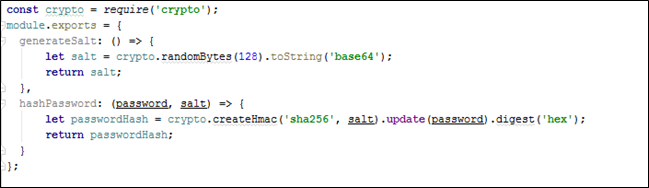
Можем да видим, че стилът е зареден правилно, за да продължим да развиваме нашия блог.

## 3. Set Up the Security Configuration

3. Настройте конфигурацията за защита  
Нашият модел е готов. Сега трябва да създадем конфигурация за сигурност.  
Първо, създайте папка, наречена "комунални услуги". Вътре в него създайте файл с име: "encryption.js". Ще има нашата логика за генериране на сол и хеширане на нашата парола. Така че, ще създадем две функции и ще ги направим обществени, за да могат да бъдат полезни.  
Първо, ще се нуждаем от помощен модул ("crypto"). За да направим функционалността видима за външния свят, ще експортираме обект, който ще има две свойства, които са функции на JavaScript.  
 След това нека дефинираме двете функции (generateSalt и hashPassword) в изявлението module.exports:



След това нека изпълним съответната функционалност за всяка функция:



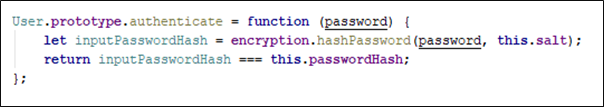
Солта ще бъде генерирана чрез създаване на масив от 128 произволни байта, които по-късно ще бъдат конвертирани в представянето им base64. За нашата хешираща логика използваме алгоритъма за хеширане SHA256.  
Създайте файл "express.js" в папката "config". В него ще поставим някаква логика за настройка. Трябва само да копирате файла "app.js" и да премахнете част от кода там и да добавите модулите за удостоверяване - трябва да изглежда така:

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

Нека да поговорим за модулите, които използваме:  
• **express** (функцията за експрес) - обвиване, която платформата Node.js осигурява, като същевременно прави кодирането по - лесно и по - бързо. Вижте примера с "express.static". Това, което прави, е да вземете предоставения осигурен файл (който е резултат от модула по-долу), статичен. Това означава, че абсолютно всички файлове в този път се виждат от никого на нашия сървър (без ограничения).  
• **path** - полезни функции за доставка на пътя за свързване на файловите пътища (относителни или абсолютни - няма значение) или каквито и да е инструменти, необходими при използване на файлови пътища.  
• **cookie-parser** - "бисквитките" съдържат криптирани данни за текущия потребител и те се изпращат при всяко искане. С този модул активираме работа с "бисквитки".  
• **Body-parser** - анализира данните от тялото на молбата и я прави достъпна, като просто ги картографира като обект с различни свойства. Вижте документацията.  
• **express-session** (експресна сесия) - съхранение на сървъра. С този "таен" низ ние можем да разграничим бисквитките един от друг (чрез задаване на идентификатор на всяка бисквитка). Също така съхранява информация за връзката на текущия потребител. Само за разработки.  
• **passport** - модул за защита, който използва сесията, за да запазва информация за потребителя. Тя изисква стратегия за спестяване ("Facebook", "Google", "Местна" и т.н.) и ни казва кои данни от потребителя ще поставим в "бисквитката". Той свързва две функции: logIn и logOut.  
Сега да създадем файл "passport.js" в папката "config" и да изберем стратегия за удостоверяване за вход:



Както виждате, ние обявихме функция за удостоверяване на потребител чрез потребителското му име и парола. Това означава, че първо, потребителското име трябва да съществува в базата данни, а втората - дадена парола трябва да бъде равна на тази в базата данни (разбира се, има хеш).  
Освен това, нашата функция получава трети аргумент, наречен "done" - друга функция, която ще бъде извикана в текущата функция.  
Логиката зад това е да се предадат всички грешки (ако има такива) като първи аргумент и потребителят като втори аргумент. Ако не можем да удостоверим потребителя, връщаме грешките в първия аргумент и ако можем, ние просто преминем нула. В противен случай връщаме потребителя. Тази логика е необходима, за да се приложи стратегията за влизане в паспорта. В този проект ще използваме стратегията "Местен паспорт". Това означава, че текущият потребител ще бъде оторизиран само в границите на нашето приложение (можете да имате стратегия за паспортите във Facebook, където ще използвате идентификационните данни на Facebook за влизане).  
Тук използваме метода на удостоверяване от модела на потребителя (user.js). Това е задачата да се види дали текущата парола съвпада с оригиналната. Ето логиката в дефиницията на потребителя:



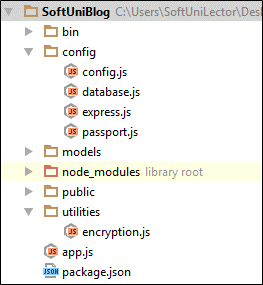
Паспортният модул ще ни предостави две функции (както е казано по-горе), което означава, че той автоматично се грижи за вход / изход. Въпреки това, входните данни могат да бъдат извикани по различен начин от "имейл" и "парола" (т.е. в html формуляра входните полета могат да бъдат наименувани по различен начин), поради което можем да преминем някои конфигурационни обекти, в които можем да настроим тези имена (usernameField: потребителско име). За да завършим тази стратегия, трябва да я пренесем в паспорния модул, като използваме ключовата дума: "използвай".  
След това ще трябва да въведем две функции за нашия паспортен модул. Те се наричат: serializeUser и deserializeUser. Паспортът е отговорен за разграничаване на потребителите (като паспорт в реалния живот), за да направим това, трябва да го кажем как да разграничим потребителите.  
Кодът за това изглежда така:

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);  
 })  
 })  
 };

Нека разбием какво прави:  
• **serializeUser** - даваме обекта на потребителя (потребителска променлива) и изпълнена функция. Ако нямаме потребител, обаждаме се с нула и ако го направим, връщаме идентификатора на потребителя.  
• **deserializeUser** - даваме му ID на потребителя (потребителска променлива) и изпълнена функция. Ако нямаме потребител, ние наричаме направено с null и ако го направим, връщаме потребителя като обект.  
Тъй като премествахме много нашата логика в модула "express.js", можем безопасно да го премахнем от "app.js". Ето как трябва да изглежда "app.js" в края:



Ето как изглежда структурата на проекта след добавянето на тези три модула:

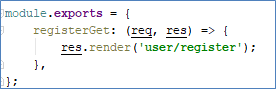


## 4. Register User

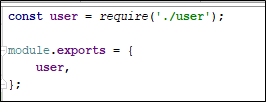
4. Регистрирайте потребителя  
Сега, когато имаме нашата стратегия за автентификация и образец на образ, нека започнем да създаваме няколко изгледа, за да регистрираме първия си потребител!  
Създайте папка "потребител" в папката "изгледи". Поставете файла "register.hbs" в него и копирайте следния HTML:

<**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**>

Сега, след като имаме потребителска регистрация, нека създадем контролер, който да го направи.  
Нека да създадем папка, наречена "контролери". Вътре в него създайте файл, наречен "index.js", и друг "user.js". Ще поставим всичко всичката логика, която ще манипулира нашия потребителски модел там. Добавете функция, която ще направи HTML преминали по-горе:



Нашата функция в контролера ще получи искането и отговора като параметри.  
В "index.js" трябва да експортирате всички контролери.



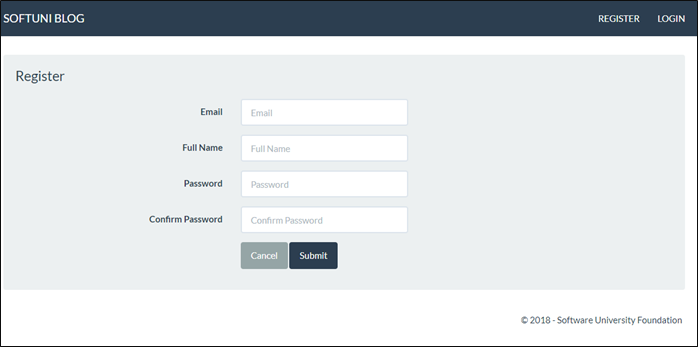
Това, от което се нуждаем сега, е да дефинираме маршрути (маршрутите ще кажат кой контролер ще бъде наречен). Логиката на маршрутите е проста и се определя от REST API. Нека да изтрием папката с маршрути, които имаме, и да създадем файл "routes.js" в папката "config", където можем да се справим с всички искания:



Сега го изискайте в нашия app.js файл:



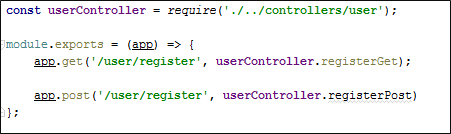
Ако всичко е наред и ние стартираме сървъра, когато отидем на localhost: 3000 / user / register, трябва да се появи следното:



В случай, че има някои грешки, не се притеснявайте, може би трябва да инсталирате някои модули като експресна сесия, просто да ги инсталирате.  
Нашият формуляр се покаже (използвайки заявка GET)!  
Да се потопим по-дълбоко в "user / register.hbs". Ако погледнем маркера <form>, можем да видим, че формулярът има два атрибута: (ключ  стойност) двойки:  
• **method (**метод) **-> post**  
• **action -> /user/register**  
Това просто означава, че всеки път, когато се подаде този формуляр (т.е. бутонът тип "submit" е натиснат). Тя ще създаде заявка за POST към URL адреса, описан по-горе:



Това означава, че трябва да създадем нов маршрут със същия URL адрес, но с различен HTTP метод (POST):  
Първо, добавете маршрута към файла "routes.js":



На второ място създайте ново действие в контролера на потребителя. Това действие трябва да направи следното:



Трябва да анализираме входните данни. Можем да го намерим в тялото на молбата. Можете да получите конкретни аргументи от него, като предадете името на полето за въвеждане (взети от html). Ако погледнем "user / register.hbs", можем да видим, че всяко поле за въвеждане има атрибут за име (name = "email" и т.н.):



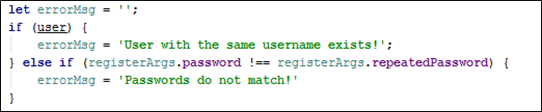
Така че, ако искаме да вземем стойността "имейл", можем да го направим с: "registerArgs.email". За по-голяма яснота разгледайте снимките по-долу.  
Второ, трябва да напишем нашата логика за валидиране, която трябва да отговори на два въпроса:  
• Вече е регистриран даден имейл?  
• И двете пароли съвпадат ли?  
Ако отговорът на двата въпроса е "да", само тогава можем да регистрираме потребителя.  
Преди да можем да отговорим на въпросите, трябва да разберем как можем да търсим моделите за данни. За щастие, Sequelize ни дава тази функционалност. Всичко, което трябва да направите, е да изискате () потребителския модел и след това просто да използвате вградените методи на Sequelze:  
 В този момент нашият контролер user.js трябва да изглежда така:



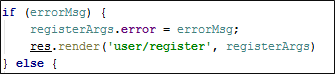
Сега, обратно към валидирането. Нека да отговорим на въпросите за валидацията. За да направите това, трябва да се свържем с нашата база данни и да проверим дали вече има потребител с този имейл. Това може да стане, като използвате функцията Sequelize findOne (). Тази команда приема обект, който можем да използваме като филтър: "къде: {email: registerArgs.email}"



Сега, след като намерим потребителя, нека да потвърдим дали потребителят съществува и дали паролите съвпадат:



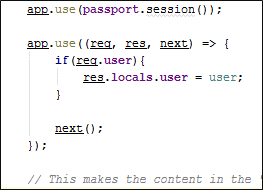
За всеки случай на грешка ще създадем низ променлива, в която ще запазим съобщението за грешка. Имайте предвид, че JavaScript е странен, когато говорим за трагични и фалшиви стойности. Прочетете тази статия за по-голяма яснота.  
Сега нека анализираме какво прави логиката в горната екранна снимка:  
Ако потребителската променлива върне нещо, това е лошо, защото това означава, че вече има потребител с този имейл в базата данни. Ако не бяха, щеше да се върне на нула.  
Можем да конвертираме потребителската променлива в true / false чрез просто въвеждане в if statement. Неопределена стойност би била преобразувана в фалшива и съществуващ потребител (който не е дефиниран) в истината.  
След като получим валидността си, трябва да проверим за грешки:



Ако възникнат грешки, просто ще презаредим страницата. Основното нещо е, че ще го презаредим с предишните стойности (така че потребителят да не е изтрит от формуляра си) и със съобщението за грешка. Съобщението за грешка ще се покаже в оформлението ("layout.hbs").  
От друга страна, ако не сме имали грешки, трябва да вмъкнем нов обект в базата данни и да регистрираме потребителя в:



Не забравяйте да изискате модела "Потребител" и модула за "кодиране"!  
Едно последно нещо, преди да преминем към формуляра за вход - отидете в "express.js" файла и добавете следното:



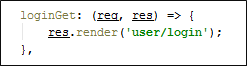
Току-що обявихме мидълуер, който просто ще направи видим текущия ни потребител и за контролите.

## 5. Login Form

5. Формуляр за вход  
Ще създадем функцията за вход, по същия начин, по който създадохме регистъра. В предишната стъпка направихме следното: регистрационен формат -> контролер -> маршрут -> контролер.  
Създайте "login.hbs" в папките "view / user":

|  |
| --- |
| <**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**> |

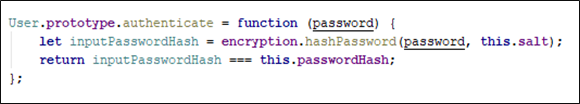
Then add the **action** in the **user controller(**След това добавете действието в потребителския контролер):



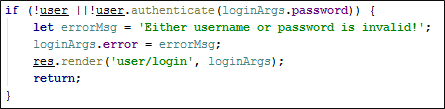
След това разширете файла "routes.js" с действието ни за влизане в профила:



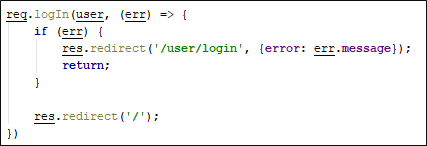
Време е да се върнете към контролера на потребителя и да създадете логика за вход. Както при регистрацията, трябва да отговорим на няколко въпроса за валидиране, преди да позволим на потребителя да влезе в профила:  
• Потребителят съществува ли?  
• Ако е така, паролата (при хеширане), която ни даде, съответства на хешираната парола, дадена във входа?  
Най-лесният начин да направите това е да дадете на всеки потребител функция за валидиране. Това е най-лесният начин, защото потребителите имат цялата необходима информация (сол и passwordHash). Отидете в папката User.js в "models" и добавете този блок код:



Уверете се, че този метод се показва преди този, който създава асоциациите на моделите на потребителя.  
Отново на контролера. Въведете заявка за търсене (известна още като User.findOne) и потвърдете входа на потребителя:

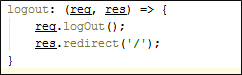


Така че, имаме известно потвърждение на входа, какво остава всъщност да влезете в системата. Можете да използвате същата логика, която използвахме в секцията за регистрация.

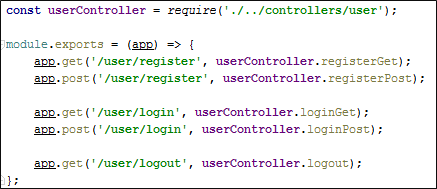


## 6. Logout

6. Изход  
Излизането е много проста. Да го добавим в контролера на потребителя:



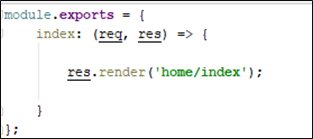
Добавете маршрута за изход. Ето как "routes.js" трябва да разгледа в този момент:



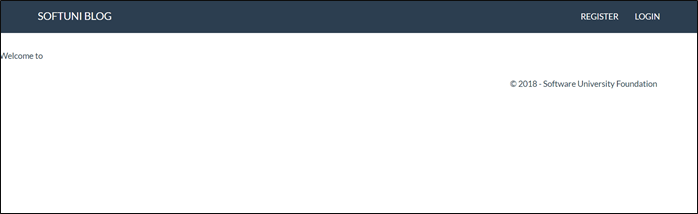
# IV. Creating home controller

## 1. Create home controller

1. Създайте домашен контролер  
Тъй като сме променили логиката зад маршрута си, не можем да получим индексната ни страница.  
Така че, трябва да създадем друг контролер -> home.js  
Той ще бъде отговорен да контролира всички маршрути за заявки, които не са свързани с нашите модели.



Да преместим индекса си в отделна папка, наречена "дом".  
Сега отидете на маршрути и добавете заявката за индекс за търсене, трябва да имате нужда от домашен контролер, за да можете да го използвате.  
Сега виждаме, че нашата индексна страница все още използва този експресиран генериран html.

Да променим съдържанието на началната страница, просто заменете текущото съдържание със следното:

|  |
| --- |
| <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> |

## 2. Refactor bin/www

2. Рефактор / WWW  
Точно сега нашият бин / www файл ни остаря, но все пак можете да използвате текущия, генериран при стартирането на проекта, ако искате да можете да го репарирате сами, или можете да го използвате:

#!/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}`);  
 }

# V. Creating Articles

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

1. Стартирайте MySQL (само ако сте тук от самото начало)  
Преди да пуснете шунка на MySQL, нека изясним някои неща. MySQL е база данни RDMS. Но каква е базата данни? Базата данни е само място за съхранение на информация. Засега можем да приемем, че база данни е само куп таблици, в които съхраняваме информация (SQL). Ето как изглежда нашата таблица за потребители от предишните стъпки:  
Имаме няколко маси, всяка от които има няколко колони, които ни дават възможност да съхраняваме данни в тях. Това изглежда като типична SQL база данни:



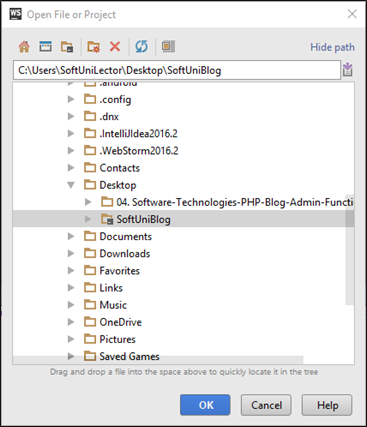
Сега можете да общувате с базата данни и да изпълнявате команди.  
Можете да създадете база данни с име "блог".  
Резюме: Сега знаем простото определение на база данни. Видяхме различни идеи за внедряване на база данни. Също така научихме как да стартираме MySQL сървър, от който можем да създаваме и манипулираме различни бази данни.  
Ето как може да изглежда връзката ви:



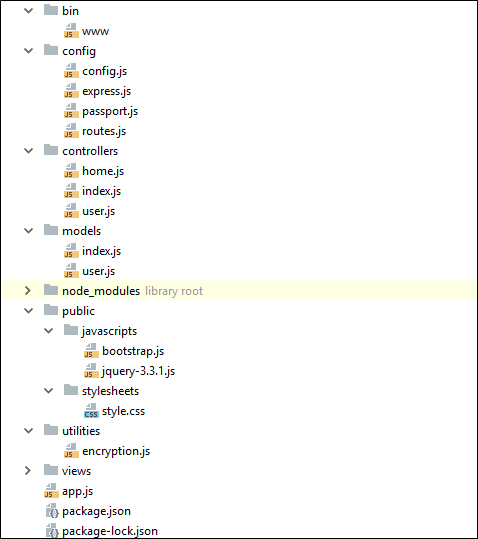
## 2. Open/Create project

2. Отворете / създайте проекта  
Нашата база данни е готова. Нека да продължим напред и да заредим скелета. Кликнете върху отворения и намерете изтегления и разкопчания проект на скелет:





Имайте предвид, че проектът за скелет има още един контролер, наречен "дом" и още една папка в "изгледи", също така наречена "дом". Не се притеснявайте, ако в момента нямате такива, ние ще говорим за тях по-късно. Ето как трябва да изглежда структурата на проекта:



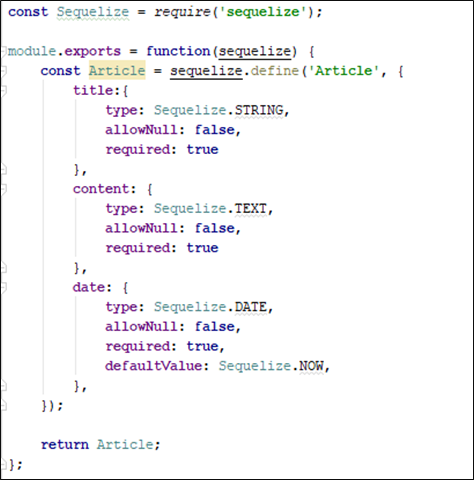
Това е нашият проект Node.js. В предишните стъпки видяхме как сме стигнали тук. Сега нека поговорим малко за възел:  
Node.JS е платформа, написана в JavaScript, която осигурява функционалност отзад. Това ни дава много гъвкавост, защото нашият front-end (HTML, използващ jQuery, Ajax и т.н.) също използва JavaScript. Това прави комуникацията по-лесна. Той е бърз, защото използва C ++ зад кулисите, а също така, защото е способен да извършва асинхронни повиквания. Тя използва системата за събития.  
Резюме: Изтеглехме проекта и сме готови за по-нататъшни действия!

## 3. Create the Article Model

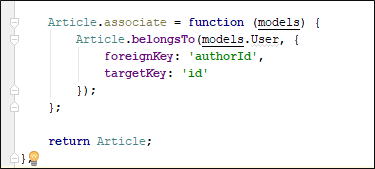
3. Създайте модел на статията  
Време е да създадем нашия основен обект - членът. Той ще има следните свойства:

* title
* content
* author
* date

Интересното тук е собственикът на автора, защото вече е модел в нашата база данни. Представете си, че ако всеки път създадохме статия, ние обвързахме тази информация на автора със статията. Какво се случва, когато един автор създава 50 статии? Бихме ли запазили едно и също име на автор и други свойства за всеки един? Не би ли било огромно разхищение на паметта? Да, как можем да разрешим този проблем?  
Отговорът е това, което се нарича референтен ключ (нещо уникално за автора - като идентификационен номер или име). Когато нашият автор има идентификатор, вместо да обвърже цялата информация със статията, ние само запазваме идентификационния номер на автора в статията.  
Но как да извлечем информация за автора на конкретна статия, питате. Е, винаги можем да ги търсим в базата данни, използвайки идентификационния номер, който запазихме в статията. Това се нарича (база данни) отношения. Един автор има нула или много статии. В следващата глава ще разгледаме още това.  
Да създадем друг файл в папката на нашите модели => article.js.  
Нека да напишем някакъв код, трябва да експортираме модела, който ще създадем. Използваме по-нататъшния начин за дефиниране на моделите. Позволява да добавите задължителни и allowNull за всеки атрибут. И можем да зададем стойност по подразбиране на нашия атрибут за дата, така че всеки път, когато създаваме статия, се съхранява текущата дата и час.



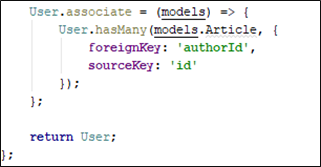
Но какво да кажем за автора. Да добавяме асоциация за статиите на потребителите. Потребителят ще бъде автор на много статии. Така че, ние създаваме много асоциация.



Обобщение: сега знаем как да създадем обект, да го обвием в модел и да определим връзка с друг модел.

## 4. Create Author – Article Relationship

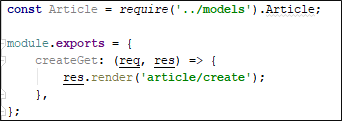
4. Създайте връзка между автор и член  
Нашата програма е като нашия реален свят - тя се основава на връзки и взаимодействия между нейните елементи. Имаме потребител, който има нула или повече статии. Тази връзка се нарича една към много. Може би искаме статиите да имат етикети. Много статии с много етикети. Отново отношение - това се нарича мнозина за мнозина. Нашите статии могат да имат категории. Една статия - една категория, от тази страна изглежда една връзка. Е, това е вярно, но имайте предвид, че една категория може да има много статии. Ето заключението: връзките могат да бъдат: един към един, един към много, много към много.  
Нека да се върнем към връзката автор - член. Една статия ще има един автор. Ние я определихме с имот в модела на статията. За да завършим връзката, трябва да сменим определението на настоящия потребител. В света на базата данни това се нарича Асоциация. Да направим миграцията в дефиницията на потребителя:



Обобщена информация: връзката на базата данни определя връзката между две единици. Типът на връзката зависи от гледната точка.

## 5. Create the Article Controller

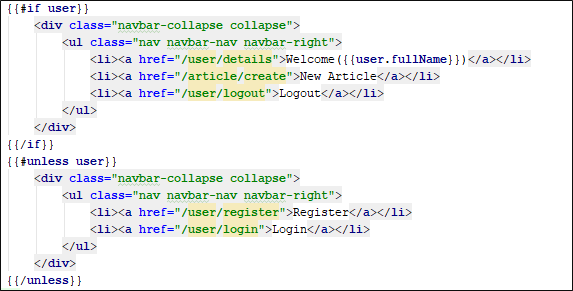
5. Създайте контролера на статията  
Следващата част ще създаде контролера за статии, където ще поставим цялата логика, свързана директно с модела на статията. Създайте файл "article.js" в папката "контролери". Като стартер искаме да създадем метод, който да направи формуляра за създаване на статия. Контролерът може да изглежда така:



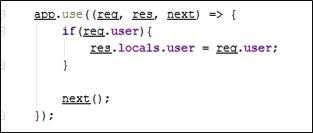
Важно нещо по този начин е, че кодът, инициализиращ модела на артикула, трябва да бъде съставен преди да се опитаме да получим достъп до модела.  
С горния код се налага да създадете изглед, който ще направи формуляра за създаване на статия.

## 6. Templating Article Form

6. Подготвяне на формуляра за членство  
В началото на създаването на проекта казахме, че ще използваме двигателя за преглед на дръжките. Така че, този път, вместо да копираме html и да се движим напред, нека видим как се прави шаблонирането. Като пример ще вземем layout.hbs:



Можем да видим, че има много HTML, но също така има няколко блока от кода, които не са. Тези части са за двигателя за преглед. Нека обясним какво прави с кода.  
С двойно къдрави скоби "{{", казваме, че следващата част няма да бъде html, а команда за нашия изглед. Този обхват за командата завършва с следващите затварящи скосени скоби - "}}". В настоящия пример можем да видим, че имаме израз if (#if). Ако променливата, предавана на "if", е "truthy", цялата html, докато {{/ if}} се покаже.  
Добре, но какво ще стане, ако променливата е фалшива и искаме да покажем нещо различно? Ще използваме "освен", по същия начин, по който използвахме "ако".  
В крайна сметка резултатът ще бъде: ако има потребител, влезли в него, покажете първия блог html (с "Добре дошли", "Изход" и т.н.), иначе показвате другия блог html ("Регистрация", "Вход" и т.н. .)  
Но как виждат тезата за текущия потребител? Погледнете "express.js" - има мидълуер, който обвързва потребителя по начин, който позволява да бъде видим за изгледа (ако нямате този мидълуер - той трябва да бъде някъде след паспорт.сесия ()):



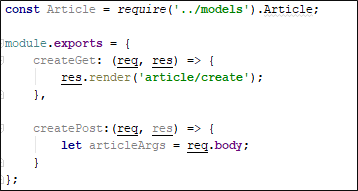
Друго нещо, което трябва да спомена: погледнете първия <a> маркер - има блок "{{user.fullName}}". Това просто означава, че не само можем да използваме "потребител" като булев, но всъщност да вземем данни от него! Има повече команди, които да използвате (като "всеки"), но ще ги покрием по-късно. Засега нека да се върнем към статията. Нуждаем се от изглед, който да показва HTML формат. В тази форма ще бъдат добавени данни (заглавие, съдържание и т.н.) и ще трябва да ги вземем в нашата логика (например контролер). Създайте файл "views / article / create.hbs":

|  |
| --- |
| <**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**> |

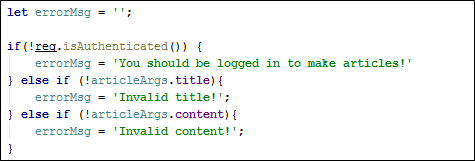
Нашият формуляр html tag съдържа два важни атрибута: метод - който определя как HTTP методът на искането ще бъде и действие - действителната връзка, където искаме данните да отидат. Така че, навсякъде, където е подадена тази форма, молбата ще отиде там, където посочва атрибута за действие.  
Резюме: Двигателят за изгледи ни помага да поставяме логика в нашите възгледи и също така ни помага да показваме още повече информация. Най-хубавото е, че логиката може да бъде поставена директно в нашия HTML код. 

## 7. Finalize Article Creation

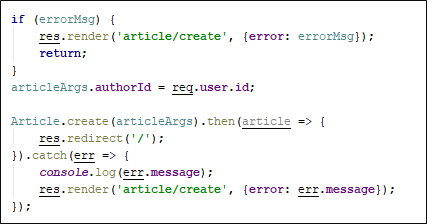
7. Да финализирате създаването на статии  
След като покажем формата си, е време да анализираме данните и да завършим създаването на статията. Върнете се в "controllers / article.js" и създайте друга функция, за да се справите с тази логика:



Ето как сега трябва да търси контролерът на статиите. Разполагаме с анализа на нашите статии, така че започнете да правите някои потвърждавания:



req.isAuthenticated () идва от паспорния модул и проверява дали в момента има регистриран потребител. Това валидиране е незадължително за момента. Други проверки валидират, ако заглавието / съдържанието е празно / недефинирано / null. Ако са такива, се създава съобщение за грешка.  
След всички потвърждения има две неща, които можем да направим: или да информираме потребителя, ако възникне грешка, или да създадем статия в статия и да я съхраним в базата данни:



Ето как върви управлението:  
• Ако има грешки, ние ще:  
o Преиздайте същата страница, но този път ще преминем обект със собственост "грешка", чийто текст ще се покаже в "layout.hbs".  
• Ако няма грешка, ние ще:  
o Задайте идентификатора на автора на обекта на статията  
Ето нашето пренасочване. Просто казваме къде да пренасочим (в нашия случай ще бъде само началната страница - "/") и да предаваме всяка допълнителна информация (обект) към двигателя за преглед (ако е необходимо).  
Сега трябва да добавим маршрутите към файла "routes.js". Първо изискайте articleController:



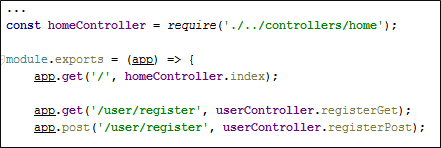
And **attach** **functions** to the POST and GET methods on the **“/article/create**” URL:



Ако сте пропуснали да създадете началния контролер и индексния изглед:  
Създайте папка, наречена "дом" в папката "изгледи". След това създайте празен файл "index.hbs". Отидете в папката "контролери" и добавете нов контролер, наречен "home.js". Вътре в него просто напишете:

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

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



Ако имате проблеми с тази настройка (или някоя друга), не се колебайте да погледнете от скелета.   
Резюме: Завършихме нашата логика за създаване на статии. Извършихме валидации и въз основа на тях можем да информираме нашия потребител за грешки. След като запазим статията в базата данни, актуализираме статиите на потребителя.

# VI. Read Articles

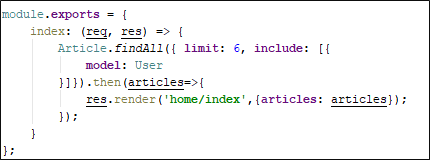
VI. Прочетете статиите  
В тази част ще се съсредоточим върху манипулирането на елемента на статията.

## 1. List Articles

1. Списък на членовете  
Това, което ще се опитаме да направим сега, е да покажем 6 статии с информация за всеки един от тях. Искаме да го направим в нашия дом, затова нека да разгледаме "home / index.hbs" и да спазваме следното:

|  |
| --- |
| <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> |

Тук използваме пълната сила на Handlebars. Използваме "всяка" конструкция (която работи по същия начин, както и на някои езици).  
Преглеждаме всяка статия, която ни беше предадена. За всеки един ще покажем:  
• its **title** - заглавието му (използвайки това означава, че итератираме по текущата статия)  
• неговото съдържание- its **content**   
• неговият автор- its **author**  
Интересното тук е, че ние предаваме това изявление: "this.User.fullName". Не забравяйте, когато създадохме модела на статията? Създадохме ли връзка с потребителя? Тук идва крайната точка за получаване на цялата информация (от нашата връзка). Нека видим как ще получим тази информация от "домашния" контролер:



Това, което ще направите, е:  
• Изтеглете всички статии  
• Вземи само първите 6 от тях  
• Включете в обекта потребител, свързан с всяка статия.  
• Изпратете ги в изгледа "home / index".  
Също така, забележете връзката "Прочетете повече": тя е "article / details / this.id". Това означава, че всяка статия, която искаме да покажем, ще има уникален маршрут (URL), базиран на идентификационния номер на статията. Ето как администраторът ни може да получи информация за статията, която искаме да видим. В следващата глава ще разгледаме по-дълбоко.  
Ето как трябва да се появи статията в нашата страница:



Резюме: Вече знаем как можем да повтаряме обект в нашия изглед.

## 2. Details Articles

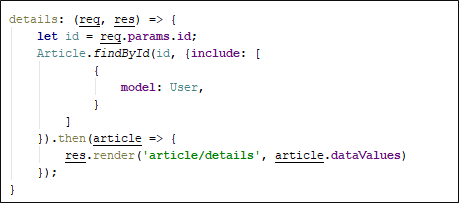
2. Подробни статии  
Забелязали ли сте бутона "Прочетете повече"? Нека я приложим. Искаме да покажем по-подробна информация за конкретната статия, когато кликнем върху нея.  
Отново първата ни стъпка е да генерираме гледката. Това означава, че трябва да създадем файл с име "details.hbs" в папката "виждания / статии":

|  |
| --- |
| <**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**> |

Имаме изгледа, сега нека го използваме в нашия контролер:  
Винаги, когато искаме да видим подробностите за статия, е логично да кажем на сървъра коя статия искаме да видим. Тази информация ще бъде изпратена чрез URL адреса. В URL адреса ще преминем "id" на статията (вече направихме в "index.hbs"). След като получим това "id" от страна на сървъра, можем да намерим конкретната статия по идентификационния й номер и след това да го предадем на двигателя за преглед.  
Как можем да получим информацията от връзката? Ще използваме req.params. Но първо нека да видим как нашето маршрутизиране ще изглежда в "routes.js":



Просто добавете част "/ article / details /: id". Това означава, че в края на връзката ни очакваме параметър с име "id". По-късно, когато използваме req.params, можем да осъществим достъп до този параметър, като просто получим името му като свойство на обекта req.params.  
Така че, ако искаме да получим параметър с името "id", ще напишем req.params.id. Ето как получаваме параметрите от нашия URL адрес (забележка: req.params е различен от req.body).  
Нека да приложим страницата с подробности в редактора на статии (article.js):



Сега всичко трябва да бъде изпълнено и трябва да сме в състояние да получим подробности за статия, когато натиснете бутона "Прочетете повече" до статия!  
Резюме: Видяхме как да показваме по-подробна информация за статия. Предавахме необходимите параметри в нашия URL адрес, от който можем лесно да работим от страна на сървъра. По този начин можем да получим информация за дадена статия, като просто даваме на сървъра неговата идентификация.

# VII. Continue the project

VII. Продължете проекта  
Поздравления, току-що сте внедрили основна система от блогове, като използвате Note.JS като сървър, ExpressJS като рутер, Sequelize като ORM, MySQL като база данни и Handlebars като двигател за преглед! Чувствайте се свободни да внедрите допълнителна функционалност, използвайки новите си познания (опитайте да внедрите изглед на профил за всеки потребител || добавете коментари). Честит кодиране!