<https://github.com/jgrisafe/express-sequelize-authentication-boilerplate.git>

Ok here we go…this is going to be quite a journey, so take a deep breath. If you’re here because you’re tired of reading tutorials that teach you how to set up passport.js or firebase authentication you’ve come to the right place. My goal for this article is to give you rich understanding of how an authentication service might be built from scratch. Keep in mind that user authentication is no joking matter, and in real world applications heavy security protocols would be implemented that are far beyond the scope of this post. Use this a starting point and do your due diligence afterwards to beef up your authentication if you plan on launching your app to the world. I will do my best to point out places where we are cutting corners on security and best practices.

Before starting this post, a basic understanding of express, ajax, and templating frameworks might be helpful. Although I’ll try to make this as comprehensive as possible.

First, let me tell you **what we will not do in this article:**

1. we will not use any authentication libraries.
2. we will not use any front-end framework like react or angular (to keep it simple and non-opinionated).
3. we will not use any css preprocessors such as sass.
4. we will not use any front-end javascript library besides jQuery ajax for form handling.

**What we will do in this article:**

1. we will make user authentication from scratch with no framework, library, or oauth.
2. we will use mySQL for our database
3. we will use minimal packages, writing custom code wherever practical
4. we will use handlebars only as a means to show what can be done with authentication (otherwise there would be little value!)
5. we will break down (almost) every single line of code

Ok now for the standard technology stack breakdown. You will need these items installed before moving forward.

1. mySQL: [mac](https://gist.github.com/nrollr/3f57fc15ded7dddddcc4e82fe137b58e) or [windows](https://dev.mysql.com/downloads/installer/).
2. [node](https://nodejs.org/en/). This application uses es6 syntax on the backend so upgrade your node to 8.0 or greater or you will need to replace all arrow functions and object destructuring with es5 syntax.
3. npm (should come with node if you installed it via the link above).
4. nodemon will help. npm install -g nodemon.
5. [postman](https://www.getpostman.com/)
6. [mySQL Workbench](https://www.mysql.com/products/workbench/) or [Sequel Pro (mac only)](https://www.sequelpro.com/)

Before diving in, lets set up our starting folder structure. Your app should contain two folders, client and server. And the client should have a public folder inside. That’s it for now.

app root|-client|---public|-server

Ok lets start with our express server. But first, create a package.json file in the root of your project, paste in the following code, and run npm install.

{  
 "name": "custom-auth-tutorial",  
 "version": "1.0.0",  
 "description": "Custom authentication in express.js",  
 "main": "server/index.js",  
 "scripts": {  
 "start": "node server",  
 "dev": "nodemon server --watch server"  
 },  
 "keywords": ["custom", "authentication", "express"],  
 "author": "your name <your\_email@domain.com",  
 "license": "MIT",  
 "dependencies": {  
 "bcrypt": "^2.0.0",  
 "body-parser": "^1.15.2",  
 "cookie-parser": "^1.4.3",  
 "dotenv": "^5.0.1",  
 "express": "^4.14.0",  
 "express-handlebars": "^3.0.0",  
 "mysql2": "^1.3.6",  
 "sequelize": "^4.3.1",  
 "sequelize-cli": "^4.0.0"  
 },  
}

Next, go ahead and create an index.jsfile inside of your server directory. This will be the home of our express server. Lets add some basic express boilerplate code to get us started:

require('dotenv').config()  
const express = require('express');  
const bodyParser = require('body-parser');  
const path = require('path');  
const cookieParser = require('cookie-parser');  
const exphbs = require('express-handlebars');// directory referencesconst clientDir = path.join(\_\_dirname, '../client');// set up the Express Appconst app = express();  
const PORT = process.env.PORT || 8080;// Express middleware that allows POSTing dataapp.use(bodyParser.urlencoded({ extended: true }));  
app.use(bodyParser.json());// serve up the public folder so we can request static// assets from the clientapp.use(express.static(`${clientDir}/public`));// start the express server  
app.listen(PORT, () => {  
 console.log(`app listening on port ${PORT}`);  
});

Navigate to the root of your project in your terminal, and run npm run dev. If you did not install nodemon, just run npm start. You should see the line app listening on port 8080.

The first thing we will do is set up our user model with sequelize. Lets install the sequelize-cli with the npm install -g sequelize-cli. If you can run the command sequelize, you can move forward.

Now lets cd into your server directory if you’re not already there, and run sequelize init. This should create all the boilerplate stuff needed to connect to our SQL database. The folder structure should now look like this:

app root|-client|---public|-server|---config|---migrations // ignore these|---models|------index.js // sequelize code, don't touch|---seeders|---index.js // the main express file we just made

You will not need to touch the index.js file inside the models folder nor the migrations directory. These are some sequelize file system code that will help link up our models in the future. If you want to look through it and try to understand it, I’ve pasted it and commented on it below. Otherwise please [click here to skip ahead](https://medium.com/@jgrisafe/custom-user-authentication-with-express-sequelize-and-bcrypt-667c4c0edef5#37aa).

// require the node packagesvar fs = require('fs');  
var path = require('path');  
var Sequelize = require('sequelize');// set a reference to this file's name so we can exclude it latervar basename = path.basename(\_\_filename);// create a variable called env which will pull from the process.env// or default to 'development' if nothing is specifiedvar env = process.env.NODE\_ENV || 'development';// require the config file that was generated by sequelize and use the// env variable we just created to pull in the correct db credsvar config = require(\_\_dirname + '/../config/config.json')[env];// initalize a db objectvar db = {};// we can set an optional property on our config objects call// 'use\_env\_variable' if wanted to set our db credentials on our// process.env. This is primarily used when deploying to a remote// server (in production)if (config.use\_env\_variable) {  
 var sequelize = new Sequelize(  
 process.env[config.use\_env\_variable], config  
 );  
} else {  
   
 // otherwise we use the config object to initialize our sequelize // instance var sequelize = new Sequelize(  
 config.database, config.username, config.password, config  
 );  
}// This gathers up all the model files we have yet to create, and// puts them onto our db object, so we can use them in the rest// of our applicationfs  
 .readdirSync(\_\_dirname)  
 .filter(file => {  
 return (file.indexOf('.') !== 0)  
 && (file !== basename)  
 && (file.slice(-3) === '.js');  
 })  
 .forEach(file => {  
 var model = sequelize['import'](path.join(\_\_dirname, file));  
 db[model.name] = model;  
 });Object.keys(db).forEach(modelName => {  
 if (db[modelName].associate) {  
 db[modelName].associate(db);  
 }  
});// export the main sequelize package with an uppercase 'S' and // our own sequelize instance with a lowercase 's'db.sequelize = sequelize;  
db.Sequelize = Sequelize;module.exports = db;

# The user model.

Now that we understand how the sequelize model system works (kind of), lets create our first model, the user model. Still inside of you server directory, go ahead and run this full command: sequelize model:generate —- name User —- attributes email:string,username:string,password:string

You can learn about the sequelize cli [here](http://docs.sequelizejs.com/manual/tutorial/migrations.html#creating-first-model-and-migration-). We should now have a User.jsfile inside our models folder with the following code. Go ahead and ignore the migrations folder for now, we will not be using that in this post. Also feel free to delete any 'use strict' statements and convert any var declarations into const declarations.

module.exports = (sequelize, DataTypes) => {  
 const User = sequelize.define('User', {  
 email: DataTypes.STRING,  
 username: DataTypes.STRING,  
 password: DataTypes.STRING  
 }, {}); User.associate = function(models) {  
 // associations can be defined here };  
   
 return User;  
};

Lets go ahead and set up our config file to use our local credentials. In your server/config/config.json paste in your local db credentials:

{  
 "development": {  
 "username": "root", // your sql username  
 "password": "root", // your sql password (may be null)  
 "database": "custom\_auth\_tutorial", // db name  
 "host": "127.0.0.1", // local host  
 "dialect": "mysql"  
 }  
}

Next, open up your sql gui, such as mySequel Workbench or SequelPro (for mac), connect to local host, and create a db named custom\_auth\_tutorial. The SQL statement for this is create database custom\_auth\_tutorial;

Finally, lets sync up our db to our express app. Back in your index.js file lets require our db at the top of the page like so:

// Requiring our models for syncingconst db = require('./models/index');

And change this code at the bottom:

// start the express server  
app.listen(PORT, () => {  
 console.log(`app listening on port ${PORT}`);  
});

to this code, with our asynchronous db sync:

// sync our sequelize models and then start server// force: true will wipe our database on each server restart// this is ideal while we change the models arounddb.sequelize.sync({ force: true }).then(() => {  
   
 // inside our db sync callback, we start the server // this is our way of making sure the server is not listening  // to requests if we have not made a db connection app.listen(PORT, () => {  
 console.log(`App listening on PORT ${PORT}`);  
 });  
});

The force: true option our sync method will force our tables to re-create themselves as we add on to them. We will remove this at the end.

Ok now is a good time to start up your server to check for errors. If you’re running nodemon, the server should have already restarted. Sequelize will create our users table for us, if it does not already exist. It’s a pretty sweet framework. You should see some output in the console that looks like this:

Executing (default): CREATE TABLE IF NOT EXISTS `Users` (`id` INTEGER NOT NULL auto\_increment , `email` VARCHAR(255), `username` VARCHAR(255), `password` VARCHAR(255), `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL, PRIMARY KEY (`id`)) ENGINE=InnoDB;  
Executing (default): SHOW INDEX FROM `Users` FROM `custom\_auth\_tutorial`

Alright, we’re making good progress. Now lets create another Model, to store our auth tokens. The reason we don’t want to store our auth tokens directly on the user model is so that we can be logged in from multiple devices. In order to keep this tutorial as simple as possible, we are going to be generating random strings for our auth tokens. In a real-world application you would most likely be using [JWTs](https://jwt.io/) with expiration dates and such. Still inside your server directory, run the following full command:  
sequelize model:generate —- name AuthToken —- attributes token:string

You should now have an AuthToken.js file inside your models folder that looks like this:

module.exports = (sequelize, DataTypes) => {  
   
 const AuthToken = sequelize.define('AuthToken', {  
 token: DataTypes.STRING  
 }, {});  
   
 AuthToken.associate = function(models) {  
 // associations can be defined here }; return AuthToken;  
};

We are gong to create an association so that we can create a link between our auth tokens and our users. If you’re unfamiliar with SQL relations, this is done by what is called a [foreign key](https://www.w3schools.com/sql/sql_foreignkey.asp). Fortunately, sequelize takes care of this for us as well. Inside the associate function above we are going to paste the statement AuthToken.belongsTo(models.User). When we save and run our server again, the auth\_tokens table will be created with a column named userId, which will correspond to the user who owns that token. The auth\_token model should now look like this:

module.exports = (sequelize, DataTypes) => {  
   
 const AuthToken = sequelize.define('AuthToken', {  
 token: DataTypes.STRING  
 }, {});  
   
 // set up the associations so we can make queries that include // the related objects AuthToken.associate = function({ User }) {  
 AuthToken.belongsTo(User);  
 }; return AuthToken;  
};

Back in your users model, go ahead and past the following line inside it’s associations function: User.hasMany(models.AuthToken);. This doesn’t alter the user table at all, but it established a two-way connection and allows us to write queries that include our auth tokens with our users. We are also going to add some validations to our user model, to ensure that the crucial columns are never allowed to be null and must be unique. The user model should now look like this:

module.exports = (sequelize, DataTypes) => { const user = sequelize.define('User', {  
 email: {  
 type: DataTypes.STRING,  
 unique: true,  
 allowNull: false  
 },  
 username: {  
 type: DataTypes.STRING,  
 unique: true,  
 allowNull: false  
 },  
 password: {  
 type: DataTypes.STRING,  
 allowNull: false  
 }  
 }, {}); User.associate = function(models) {  
 User.hasMany(models.AuthToken);  
 }; return User;  
};

Ok we’ve almost laid the ground work for our authentication process. Now we are going to create the method that actually generates the random auth token. In addition, we are going to add a validation to the model to ensure the token column is not null. Our AuthToken model should now look like this:

module.exports = (sequelize, DataTypes) => {  
  
 const AuthToken = sequelize.define('AuthToken', {  
 token: {  
 type: DataTypes.STRING,  
 allowNull: false,  
 }  
 }, {});  
  
 // set up the associations so we can make queries that include // the related objects AuthToken.associate = function({ User }) {  
 AuthToken.belongsTo(User);  
 };  
  
 // generates a random 15 character token and // associates it with a user AuthToken.generate = async function(UserId) {  
 if (!UserId) {  
 throw new Error('AuthToken requires a user ID')  
 }  
  
 let token = '';  
  
 const possibleCharacters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' +  
 'abcdefghijklmnopqrstuvwxyz0123456789';  
  
 for (var i = 0; i < 15; i++) {  
 token += possibleCharacters.charAt(  
 Math.floor(Math.random() \* possibleCharacters.length)  
 );  
 }  
  
 return AuthToken.create({ token, UserId })  
 }  
  
 return AuthToken;  
};

PHEW! This is getting long. Lets recap what we’ve done so far before we create our first controller / routes.

So far we’ve…

1. created a basic express server.
2. integrated sequelize and initialized our user and auth tokens models
3. establised the model associations
4. created a class method on the AuthToken model for generating new instances that can be assigned to users

**The user controller…**

If you’re not familiar the with [MVC pattern](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller), the controller is where the app interacts with the outside world. Specifically, where the requests are handled. You may want to read the link above to gain some familiarity before moving forward.

In you server directory, go ahead a create a controllers folder, and inside that, a file named user-controller.js. Your app structure should now look like this:

app root|-client|---public|-server|---config|---controllers|------user-controller.js|---migrations|---models|------index.js|------user.js|------auth\_token.js|---seeders|---index.js

To set up our routes, are are going to be using the express middleware called [Express Router.](https://expressjs.com/en/4x/api.html#router) This allows us define our routes without having to pass the express app object from one module to another. In your user-controller.jsfile go ahead and paste the following code:

const express = require('express');  
const router = express.Router();// grab the User model from the models folder, the sequelize  
// index.js file takes care of the exporting for us and the   
// syntaxt below is called [destructuring](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Object_destructuring), its an es6 feature  
const { User } = require("../models");// routes will go here// export the router so we can pass the routes to our servermodule.exports = router;

This is a basic boiler plate for a simple controller using express router. In a larger application we may even abstract our routes into their own files and have the controller logic there, but we are going to keep it simple for now. The controller is basically the point of communication between the client and the server. It handles the requests that come in, and chooses how the app is going to handle them. As a rule of thumb, think skinny controllers, fat models.We should strive to put as much logic as we can in the models and leave the routing to the controllers.

Lets add the three basic routes we need for authentication, **register, login, logout, and me** routes**.**We are going to be using some es6 async/await code so take a minute and [read up on that](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) if you are unfamiliar.

const express = require('express');  
const bcrypt = require('bcrypt');  
  
const router = express.Router();  
  
// grab the User model from the models folder, the sequelize// index.js file takes care of the exporting for us and the// syntax below is called destructuring, its an es6 featureconst { User } = require('../models');  
  
/\* Register Route========================================================= \*/router.post('/register', async (req, res) => {  
  
 // hash the password provided by the user with bcrypt so that // we are never storing plain text passwords. This is crucial // for keeping your db clean of sensitive data const hash = bcrypt.hashSync(req.body.password, 10);  
  
 try {  
 // create a new user with the password hash from bcrypt let user = await User.create(  
 Object.assign(req.body, { password: hash })  
 );  
  
 // data will be an object with the user and it's authToken let data = await user.authorize();  
  
 // send back the new user and auth token to the // client { user, authToken } return res.json(data);  
  
 } catch(err) {  
 return res.status(400).send(err);  
 }  
  
});  
  
/\* Login Route========================================================= \*/router.post('/login', async (req, res) => {  
 const { username, password } = req.body;  
  
 // if the username / password is missing, we use status code 400 // indicating a bad request was made and send back a message if (!username || !password) {  
 return res.status(400).send(  
 'Request missing username or password param'  
 );  
 }  
  
 try {  
 let user = await User.authenticate(username, password)  
  
 user = await user.authorize();  
  
 return res.json(user);  
  
 } catch (err) {  
 return res.status(400).send('invalid username or password');  
 }  
  
});  
  
/\* Logout Route========================================================= \*/router.delete('/logout', async (req, res) => {  
  
 // because the logout request needs to be send with // authorization we should have access to the user // on the req object, so we will try to find it and // call the model method logout const { user, cookies: { auth\_token: authToken } } = req  
  
 // we only want to attempt a logout if the user is // present in the req object, meaning it already // passed the authentication middleware. There is no reason // the authToken should be missing at this point, check anyway if (user && authToken) {  
 await req.user.logout(authToken);  
 return res.status(204).send()  
 }  
  
 // if the user missing, the user is not logged in, hence we // use status code 400 indicating a bad request was made // and send back a message return res.status(400).send(  
 { errors: [{ message: 'not authenticated' }] }  
 );  
});  
  
/\* Me Route - get the currently logged in user========================================================= \*/router.get('/me', (req, res) => {  
 if (req.user) {  
 return res.send(req.user);  
 }  
 res.status(404).send(  
 { errors: [{ message: 'missing auth token' }] }  
 );  
});  
  
// export the router so we can pass the routes to our servermodule.exports = router;

# ****The User Model Methods****

To wrap up the server side logic we are going to add some methods to the user that we referenced above. Explanations are in the comments of the code. The User model should now look like the following code:

const bcrypt = require('bcrypt');  
  
module.exports = (sequelize, DataTypes) => {  
 const User = sequelize.define('User', {  
 email: {  
 type: DataTypes.STRING,  
 unique: true,  
 allowNull: false,  
 },  
 username: {  
 type: DataTypes.STRING,  
 unique: true,  
 allowNull: false,  
 },  
 password: {  
 type: DataTypes.STRING,  
 allowNull: false,  
 },  
 });  
  
 // set up the associations so we can make queries that include // the related objects User.associate = function ({ AuthToken }) {  
 User.hasMany(AuthToken);  
 };  
  
 // This is a class method, it is not called on an individual // user object, but rather the class as a whole. // e.g. User.authenticate('user1', 'password1234') User.authenticate = async function(username, password) {  
  
 const user = await User.findOne({ where: { username } });  
  
 // bcrypt is a one-way hashing algorithm that allows us to  // store strings on the database rather than the raw // passwords. Check out the docs for more detail if (bcrypt.compareSync(password, user.password)) {  
 return user.authorize();  
 }  
  
 throw new Error('invalid password');  
 }  
  
 // in order to define an instance method, we have to access // the User model prototype. This can be found in the // sequelize documentation User.prototype.authorize = async function () {  
 const { AuthToken } = sequelize.models;  
 const user = this  
  
 // create a new auth token associated to 'this' user // by calling the AuthToken class method we created earlier // and passing it the user id const authToken = await AuthToken.generate(this.id);  
  
 // addAuthToken is a generated method provided by // sequelize which is made for any 'hasMany' relationships await user.addAuthToken(authToken);  
  
 return { user, authToken }  
 };  
  
  
 User.prototype.logout = async function (token) {  
  
 // destroy the auth token record that matches the passed token sequelize.models.AuthToken.destroy({ where: { token } });  
 };  
  
 return User;  
};

# Authentication Middleware

We need a way to get the auth token from the cookie, so we can verify a user is logged in when they hit an authenticated route. Go ahead and create a middleware directory in your server folder, then add the file custom-auth-middleware.js . Paste in the following code:

const { User, AuthToken } = require('../models');  
  
module.exports = async function(req, res, next) {  
  
 // look for an authorization header or auth\_token in the cookies const token =  
 req.cookies.auth\_token || req.headers.authorization;  
  
 // if a token is found we will try to find it's associated user // If there is one, we attach it to the req object so any // following middleware or routing logic will have access to // the authenticated user. if (token) {  
   
 // look for an auth token that matches the cookie or header const authToken = await AuthToken.find(  
 { where: { token }, include: User }  
 );  
  
 // if there is an auth token found, we attach it's associated // user to the req object so we can use it in our routes if (authToken) {  
 req.user = authToken.User;  
 }  
 }  
 next();  
}

Back in your main server file, go ahead a require the customAuthMiddleware at the top of the page, and then add these lines to your file, after the bodyparser code:

// top of page  
const customAuthMiddleware = require('./middleware/custom-auth-middleware');// ... after app.use(bodyParser.json());// use the cookie-parser to help with auth token, // it must come before the customAuthMiddlewareapp.use(cookieParser());  
app.use(customAuthMiddleware);