Link: https://hackhands.com/mongodb-crud-mvc-way-with-passport-authentication/

Posted on February 15th 2015

**MongoDB CRUD the MVC way with Passport Authentication**

**TL;DR**

This is a third post (first one is about [getting started on the MEAN stack](https://hackhands.com/how-to-get-started-on-the-mean-stack/), and the second one is all about[Node.js and Express](https://hackhands.com/delving-node-js-express-web-framework/)) in a series of posts which will teach you how to take the advantage of the MEAN stack in becoming a full-stack JavaScript developer.

In this post you’ll learn all about [MongoDB](http://www.mongodb.org/) and how Node.js and Express work together with it - and all that in a MVC way. First you'll go over the basic CRUD commands, then you'll look at [Mongoose](http://mongoosejs.com/) and finally you'll learn how to add a register/login functionality by providing the options like email/password, Facebook and Twitter by using the [Passport](http://passportjs.org/) module.

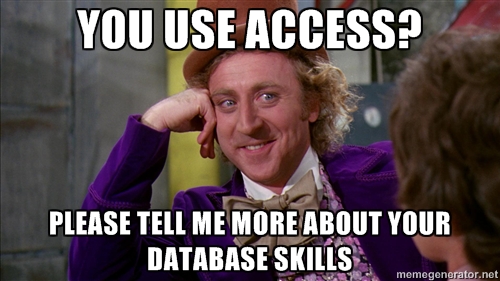
*Disclaimer: code is based on the*[*MEAN.JS framework*](http://meanjs.org/)*and the tutorial partly follows the structure from*[*MEAN Web Development by Amos Haviv*](http://amzn.to/1cvHyth)*http://ir-na.amazon-adsystem.com/e/ir?t=nikolab-20&l=as2&o=1&a=B00NXWI1BM, the author of the MEAN.JS framework.*

**Le finished project download**

If you want to follow along but you dread typing, you can [clone the finished project from Github](https://github.com/Hitman666/MEAN_MVC_3rdTutorial.git). If you would like to test the project, you have to:

* run **npm install** after cloning
* make sure that you have **mongod** running on default 27017 port
* run **node server**from the root of the application (where the **server.js** file is placed)
* *if you'll want to test the Facebook and Twitter logins (and they don't work out of the box for you) you'll have to create your own Facebook and Twitter applications by following the steps in the*[*User authentication with Passport*](https://hackhands.com/mongodb-crud-mvc-way-with-passport-authentication/#User_authentication_with_Passport)

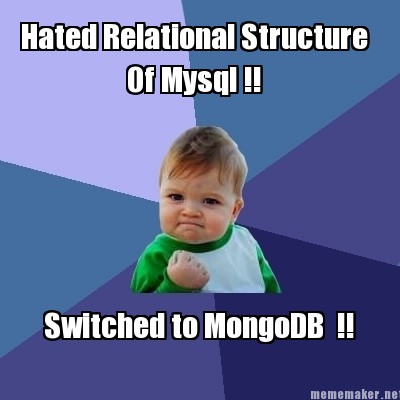
**Let's demystify this MongoDB thing**



If you're developing for the web, it's almost certain that you're using a relational database ([PostgreSQL](http://www.postgresql.org/" \t "_blank),[MySQL](http://www.mysql.com/" \t "_blank), [MariaDB](https://mariadb.org/" \t "_blank), ...) to store your data. But, you may have found your self struck with scaling problems as your application started to grow. Many different storage design patterns emerged in an attempt to solve this problem (key-value storage, column storage, object storage), but the most popular one is **document storage**.



Generally, MongoDB is one of the [NoSQL](http://en.wikipedia.org/wiki/NoSQL) databases which is a term coined by [Carlo Strozzi](http://en.wikipedia.org/wiki/NoSQL#History). NoSQL systems are also called *Not only SQL* to emphasize that they may also support SQL-like query languages.



Document oriented databases handle data differently from a relational database. Instead of using **tables**, they store hierarchical **documents** in a standard **JSON** format. For example, in a document based database (the cliche *blog* example) the *post* will be stored **completely** as a **single document** that can later be queried, opposed to SQL where you would at least have 2 tables (*posts*, *comments*). In turn, this makes read operations faster since you don't have to query multiple tables.

**Key features of MongoDB**

**BSON**

BSON (Binary JSON) is a binary-encoded serialization of JSON-like documents, and it is designed to be more efficient in size and speed. BSON documents are a simple data structure representation of objects and arrays in a **key-value format**. A document consists of a list of elements, each with a string typed **field name** and a typed **field value**. Another big advantage of the BSON format is the use of the **\_id** field as**primary key**.

The main problem of key-value stores is their limited query capabilities, which means your data can only be queried using the key field. MongoDB supports dynamically structured queries out of the box (so called[ad hoc queries](http://stackoverflow.com/questions/2460954/what-is-ad-hoc-query)), by indexing BSON documents and using a unique query language.

MongoDB query language is similar to SQL:

SELECT \* FROM todos WHERE title LIKE '%milk%';</pre>

which selects all the rows from the **todos**table which contain a *milk* word in a **title** field. The same query in MongoDB looks like this:

db.todos.find({ title:/milk/ });

MongoDB solves the issue of querying speed (as the database grows) by using a mechanism called**indexing**. [Indexes](http://en.wikipedia.org/wiki/Database_index) are a unique data structure that enable the database engine to efficiently resolve queries. You can learn more about indexes [here](http://docs.mongodb.org/manual/indexes/).

**Replica set**

MongoDB uses a so called **replica set** to provide data **redundancy** andimproved **availability**. Replication can help to recover data from (unfortunate) hardware failures. A replica set is a set of MongoDB services that host the **same dataset**. One service is used as the **primary**and others as **secondaries**. All replicas support read operations, but only the primary instance can use the **write** operations. When a write operation occurs, the primary will inform the secondaries about the changes and make sure they've applied it to their datasets' replication.

Another robust feature of the MongoDB replica set is its **automatic failover** which kicks into place when one of the set members can't reach the primary instance for more than 10 seconds, the replica set will automatically elect and promote a secondary instance as the new primary. When the old primary comes back online, it will rejoin the replica set as a *secondary*.

**Sharding**



Scaling is not a problem. **Until it is**. If you create a small application for a dozen of users you don't have to worry (you would probably get away even with a few thousand users), but if you're application becomes Facebook popular, the scaling problem will hit you in the face.



So, if you get in trouble you have two possibilities to solve the scaling issue:

* **Vertical scaling** - easier, because you only have to buy additional hardware (RAM, CPU) of a **single machine**. But, this can only be done up to a certain point when buying additional hardware becomes way **more expensive** compared to splitting the load between several smaller (cheaper) machines.
* **Horizontal scaling** - more complicated and it's done using several machines. Each machine handles a part of the load, providing better overall performance. The challenge, however, is how to properly divide the data between different machines and manage the read/write operations between them.

MongoDB supports **horizontal scaling**, called sharding. **Sharding** is the process of splitting the data between different machines (shards), where each shard holds a portion of the data and functions as a separate database. How to deploy a shard cluster goes beyond the scope of this tutorial, but you can read more about it on the [official site](http://docs.mongodb.org/manual/tutorial/deploy-shard-cluster/).

**Let's query the MongoDB via terminal**



MongoDB shell (mongo.exe on Windows machine, mongo binary on Linux) will automatically connect to the default **test** database if you run it without any parameters. In order to switch to another database (in this example called *todos*) execute the following command:

**use** todos

Databases and collections are lazily created when you insert your first document, which means that if you use the **use todos** command from above, but you don't insert any documents, the database will not be created.

In order to show all the available databases execute the following command:

**show** dbs

A MongoDB **collection** is a **list** of MongoDB **documents** and is the equivalent of a  
relational database **table**. Unlike a table, a collection **doesn't** enforce any type of **schema**. To show all available collections in the database execute:

**show** collections

**Create**

You can use three commands to create documents in MongoDB:

* insert
  + db.todos.insert({"title":"Write a post", "user": "nikola"})
* update
  + usually used to update an existing document, but if you set the **upsert** flag, it will create a new document if it does not exist:
* db.todos.**update**({
* "user": "nikola"
* },
* {
* "title": "Buy Bitcoins",
* "user": "nikola"
* },
* {
* **upsert**: true
* }
* )
* save
  + creates a new document even if the exact one (content wise) exists:
* **db**.todos.**save**({"title":"Write a post", "user": "nikola"})

**Read**

To find all the documents of the *todos* collection, execute:

db.todos.find()

To find all the *todos* of the user *nikola* execute:

db.todos.**find**({ "user": "nikola" })

To find all *todos* that were created by either *nikola* or *josipa*, you can use the **$in** operator:

**db**.todos.find({ "user": { $in: ["josipa", "nikola"] } })

Another way to find all *todos* that were created by either *nikola* or *josipa*:

**db**.todos.find( { $or: [{ "user": "nikola" }, { "user": "josipa" }] })

To find all *todos* that were created by *nikola* and have a priority greater than 3:

**db**.todos.find({ "user": "nikola", "priority": { $gt: 3 } })

**Update**

The *update()* method takes three arguments to update existing documents:

1. selection that indicates which documents to update
2. update statement
3. options object

In the following example, the first argument is telling MongoDB to look for all the documents created by*nikola* (in the *todos* collection), the second argument tells it to update the *title* field, and the third is forcing it to execute the update operation on all the documents it finds, since the **default** behavior is to update a**single** document:

**db**.todos.**update**({

"user": "nikola"

},

{

$set: {

"title": "Postpone blog post"

}

},

{

multi: true

}

)

**Delete**

To remove all the documents from the *todos* collection execute:

db.todos.remove()

To remove the **first** document from the todos collection made by user *nikola* execute:

db.todos.**remove**({ "user": "nikola" }, **true**)

In order to remove **all** of the documents made by *nikola* in the *todos* collection just omit the **true** flag.

**Mongoose**

[Mongoose](http://mongoosejs.com/) is a Node.js ODM (object document mapper) module that translates your objects in code to the document representation of the data in MongoDB. *You may be familiar with ORM, and you can see the difference explained*[*here*](http://stackoverflow.com/questions/12261866/what-is-the-difference-between-an-orm-and-an-odm)*.* While MongoDB doesn't enforce a schema, Mongoose offers a stricter schema approach.

**Continue with the application**

The following examples continue where we left off in the [second tutorial](https://hackhands.com/delving-node-js-express-web-framework/). So, feel free to [clone the second project from GitHub](https://github.com/Hitman666/MEAN_MVC_2ndTutorial) and follow along.

**Connect to MongoDB**

First, you need to install Mongoose by issuing the following command (once you're in the project root folder - *and, just so I won't have to stress this every time; from now on every****npm install****command should be done from the project's root folder (where's your server.js file)*):

npm install mongoose --save

To connect to MongoDB, you need to use the MongoDB connection URI, which looks like this:

mongodb://username:password@hostname:port/database

But, since you're connecting on a localhost (your own computer), you can skip the username and password part and use:

mongodb://localhost/todos

The simplest thing you could do is define this connection URI variable directly in your *config/\_\_express.js*configuration file and use the Mongoose module to connect to the database:

var uri = 'mongodb://localhost/todos';

var db = **require**('mongoose').connect(uri);

but this is bad practice. The proper way to store application variables is to use your environment configuration file. In the **config** folder create an **env** folder and inside it create a **development.js** file with the following code:

**var** port = 1337;

**module**.exports = {

port: port,

db: 'mongodb://localhost/todos'

};

Now, in your **config** folder, create a new file named **mongoose.js** with the following code:

**var** config = **require**('./config'),

mongoose = **require**('mongoose');

**module**.exports = **function**() {

**var** db = mongoose.connect(config.db);

**return** db;

};

**Environment configuration file**

Hopeful reader as you are, you may have noticed that you're requiring the **./config** file which you do not yet have. To fix this, create a **config.js** file in the **config** folder and place the following code inside:

**module**.exports = **require**('./env/' + process.env.NODE\_ENV + '.js');

This will let you load the correct db (and other variables that we'll define later) variable based on your environment (be it a *development* or *production*) configuration file.

The **process.env** is a global variable that allows you to access predefined environment variables, and the most common one is **NODE\_ENV**. To set it in a Windows environment execute the following command in your command prompt:

**set** NODE\_ENV=development

and in an Unix-based environment execute:

**export** NODE\_ENV=development

Now, update the **server.js** file to the following content (most of the code you already have, you're just adding the Mongoose specific code, along with environment variable setting):

process.env.NODE\_ENV = process.env.NODE\_ENV || 'development';

var config = **require**('./config/config'),

mongoose = **require**('./config/mongoose'),

express = **require**('./config/express'),

var db = mongoose(),

app = express();

app.listen(config.port);

**module**.exports = app;

**console**.log(process.env.NODE\_ENV + ' server running at http://localhost:' + config.port);

Mongoose configuration file has to be loaded **before** any other configuration in the *server.js* file (except the*config* module) because any module that is loaded after this module will be able to use its models without loading it by itself.

*If you execute****node server****command in your terminal and get an error like 'Error: failed to connect to [localhost:27017]', make sure your MongoDB instance (mongod) is running. Also, if you get another error like 'ERROR: Insufficient free space for \_\_journal files', make sure that you have at least 3379MB available on the drive where you installed it.*

**Mongoose schemas**

Since MongoDB doesn't enforce the documents to have the same structure, Mongoose offers just that by using a **Schema** object to define a list of properties, each with its own type and constraints.

To create a schema, create a new file named **user.server.model.js**in the **app/models** folder and paste the following code:

**var** mongoose = **require**('mongoose'),

Schema = mongoose.Schema;

**var** UserSchema = **new** Schema({

name: **String**,

email: **String**,

username: **String**,

password: **String**,

});

mongoose.model('User', UserSchema);

In order to use the *User* model, you need to include this file by adding the following *require* in the**config/mongoose.js** file (just before the **return db;** statement):

**require**('../app/models/user.server.model');

**Create a user**

In order to keep things nice and tidy, you'll create a *Users* controller which will handle all the requests for user related operations. In the **app/controllers** folder create a **users.server.controller.js** file and place the following code in it:

**var** User = **require**('mongoose').model('User');

exports.create = **function**(req, res, next) {

**var** user = **new** User(req.body);

user.save(**function**(err) {

**if** (err) {

**return** next(err);

}

**else** {

res.json(user);

}

});

};

In the **app/routes** folder create a file named **users.server.routes.js** and paste the following code:

**var** users = **require**('../../app/controllers/users.server.controller');

**module**.exports = **function**(app) {

app.route('/users').post(users.create);

};

In the **config/express.js** file add the following route definition:

require('../app/routes/users.server.routes.js')(**app**);

This is all we need for the RESTful user saving API, and in order to test it run

$ node server

and issue the POST request with the following request body:

{

"name": "Kevin",

"email": "kevin@mitnick.com",

"username": "Condor",

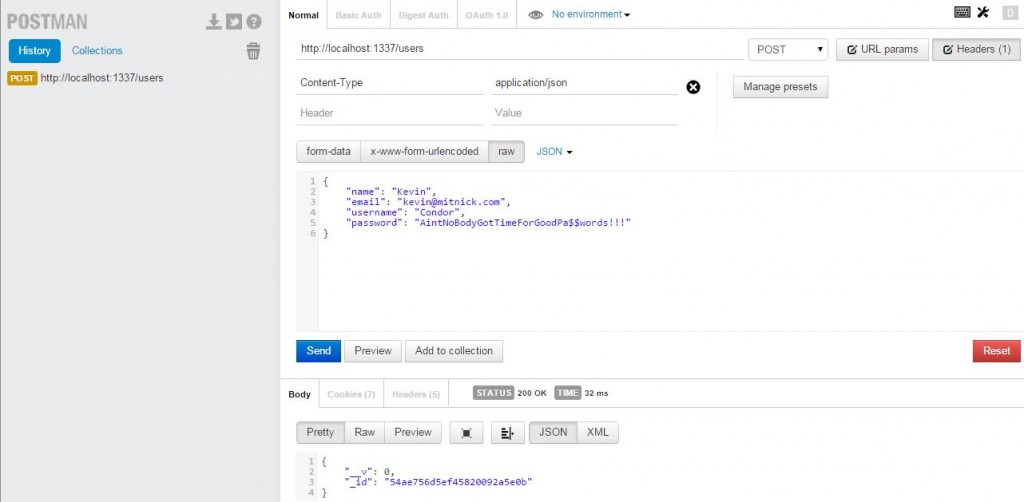
"password": "AintNoBodyGotTimeForGoodPa$words!!!"

}

You can use the **curl** command like this:

$ curl -X POST -H "Content-Type: application/json" -d '{"name": "Kevin", "email": "kevin@mitnick.com", "username": "Condor", "password": "AintNoBodyGotTimeForGoodPa$words!!!"}' localhost:1337/users

but I suggest you use a GUI tool since it will make things easier. I personally use [Postman](https://www.google.hr/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0CB0QFjAA&url=https%3A%2F%2Fchrome.google.com%2Fwebstore%2Fdetail%2Fpostman-rest-client%2Ffdmmgilgnpjigdojojpjoooidkmcomcm&ei=GniuVMHZFYH9UMiSg9gC&usg=AFQjCNFL71vN61QG0LKlw7VDJvIZDprjHA&sig2=DUyqMgSFn1BFnWzvNlf_Sg) in Chrome, but I'm sure there's also a dozen of plugins for your favorite browser. However, if you followed all the steps you will notice that this curl command or Postman's POST aren't working (sneaky, huh? ;)):



You need install the [body-parser](https://github.com/expressjs/body-parser) module which provides several middlewares to handle request data, and you can do that by executing the following command:

$ npm install body-parser --save

After this, update your **express.js** file to look like this:

**var** config = **require**('./config'),

express = **require**('express'),

bodyParser = **require**('body-parser');

module.exports = **function**() {

**var** app = express();

app.**use**(bodyParser.urlencoded({

extended: **true**

}));

app.**use**(bodyParser.json());

app.set('views', './app/views');

app.set('view engine', 'ejs');

**require**('../app/routes/index.server.routes.js')(app);

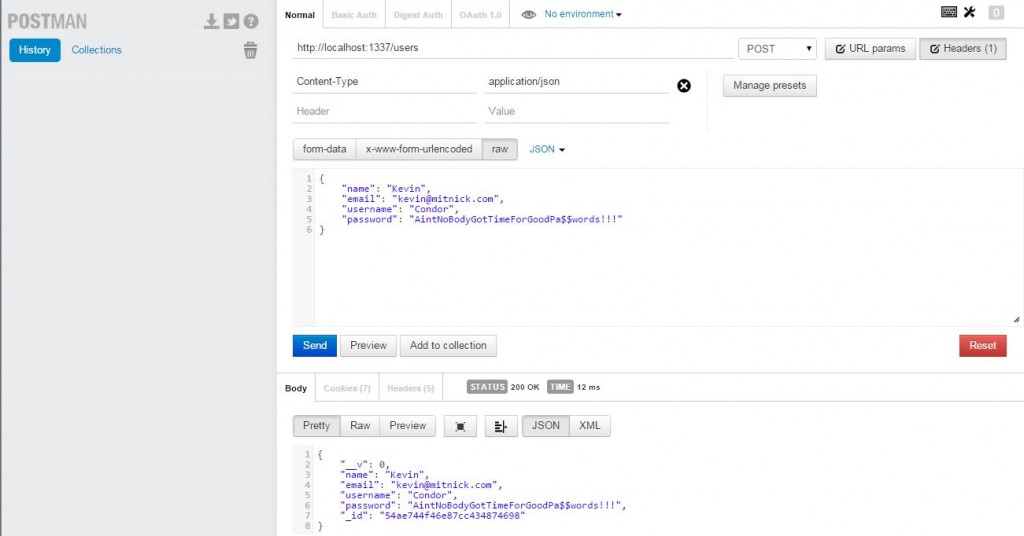
**require**('../app/routes/users.server.routes.js')(app);

app.**use**(express.**static**('./public'));

**return** app;

};

If you run **node server** now and try to POST the data again with Postman, you will see something like:



*Important to note is that in my case (with Postman) I had to add the Content-Type header (application/json) manually for the request.*

**Find all users**

Great, so you created a new user in the *users* collection. In order to list existing users add the following code to your **app/controllers/user.server.controller.js** file:

exports.list = **function**(req, res, next) {

User.find({}, **function**(err, users) {

**if** (err) {

**return** **next**(err);

}

**else** {

res.json(users);

}

});

};

Now you need to set up a route for this new method and in order to do so, go to your**app/routes/users.server.routes.js**file and change the **app.route** line to:

**app**.route('/users').**post**(users.create).get(users.**list**);

In order to test this run **node server** and enter <http://localhost:1337/users> in your browser.

**Find one user**

So, previous functionality gives you the list of all the users, but in order to show only one specific user in the **app/controllers/users.server.controller.js** file add the following code:

exports.read = **function**(req, res) {

res.json(req.user);

};

exports.userByID = **function**(req, res, next, id) {

User.findOne({

\_id: id

},

**function**(err, user) {

**if** (err) {

**return** **next**(err);

}

**else** {

req.user = user;

**next**();

}

}

);

};

The **read()** method just responds with a JSON representation of the **req.user** object. The **userById()**method is populating the req.user object, which you will use as a middleware to deal with the manipulation of single documents when performing read, delete, and update operations.

Now, modify your **app/routes/users.server.routes.js** file to look like this:

**var** users = require('../../**app**/controllers/users.server.controller');

module.exports = function(**app**) {

**app**.route('/users').**post**(users.create).get(users.**list**);

**app**.route('/users/:userId').get(users.**read**);

**app**.param('userId', users.userByID);

};

A colon before a substring in a route definition (in Express) means that this substring will be handled as a**request parameter**. To handle the population of the **req.user** object, you use the **app.param()** method that defines a middleware to be executed before any other middleware that uses that parameter. In your case the **users.userById()** method will be executed **before** any other middleware registered with the userId parameter, which in this case is the **users.read()** middleware.

*To test this, run****node server****, then navigate to*[*http://localhost:1337/users*](http://localhost:1337/users)*in your browser and grab one of your users'****\_id****values (if you created more users by playing with Postman, if not - what are you waiting for? ;)), and navigate to*[*http://localhost:1337/users/[\_id]*](http://localhost:1337/users/%5b_id%5d)*,*  
*replacing the [\_id] part with the user's \_id value.*

Mongoose offers a lot more options for searching and you can learn more about them in the [official documentation](http://mongoosejs.com/docs/api.html#query_Query-find).

**Update the user**

Mongoose model has several available methods to update an existing  
document: [update()](http://mongoosejs.com/docs/api.html#model_Model.update), [findOneAndUpdate()](http://mongoosejs.com/docs/api.html" \l "model_Model.findOneAndUpdate" \t "_blank), and [findByIdAndUpdate()](http://mongoosejs.com/docs/api.html" \l "model_Model.findByIdAndUpdate" \t "_blank). Since you're already using the userById() method, you're going to use the findByIdAndUpdate() method.

In your **app/controllers/users.server.controller.js** file add a new update() method:

exports.update = **function**(req, res, next) {

User.findByIdAndUpdate(req.user.id, req.body, **function**(err, user) {

**if** (err) {

**return** **next**(err);

}

**else** {

res.json(user);

}

});

};

Now update your **app/routes/users.server.routes.js** file to this (*.put(users.update)* is the novelty in this file):

**var** users = require('../../**app**/controllers/users.server.controller');

module.exports = function(**app**) {

**app**.route('/users').**post**(users.create).get(users.**list**);

**app**.route('/users/:userId').get(users.**read**).put(users.**update**);

**app**.param('userId', users.userByID);

};

You can test this with a PUT command:

$ curl -X PUT -H "Content-Type: application/json" -d '{"name": "UpdatedName"}' localhost:1337/users/[\_id]

where **[\_id]** is, of course, the valid *id* of some user.

**Delete the user**

I hope that by now you're seeing the pattern here - Mongoose model has several available methods to remove an existing document:

* [remove()](http://mongoosejs.com/docs/api.html#model_Model.remove)
* [findOneAndRemove()](http://mongoosejs.com/docs/api.html#model_Model.findOneAndRemove)
* [findByIdAndRemove()](http://mongoosejs.com/docs/api.html#model_Model.findByIdAndRemove)

Since you're already using the userById() method, you're going to use the findByIdAndRemove() method.

In your **app/controllers/users.server.controller.js** file add a new delete() method:

exports.**delete** = **function**(req, res, next) {

req.user.remove(**function**(err) {

**if** (err) {

**return** next(err);

}

**else** {

res.json(req.user);

}

})

};

Now, in **app/routes/users.server.routes.js** file change the route with the *userId* parameter to be like this:

app.route('/users/:userId').**get**(users.**read**).**put**(users.**update**).**delete**(users.**delete**);

To test the delete method, run **node server** and issue a DELETE request, example with curl (OFC replace [*id*] as before):

$ curl -X DELETE localhost:1337/users/[id]

**Few additional goodies**

**Indexes**

A **unique** index validates the uniqueness of a document field across a collection. Since you most likely want the *username* to be unique, you have to add this to your *User* model:

**var** UserSchema = new Schema({

...

username: {

**type**: **String**,

unique: true

},

...

});

Mongoose also supports **secondary indexes** using the **index** property. So, if you know that your application will use a lot of queries with the *email* field, you could optimize these queries by creating an *email \_secondary* index:

**var** UserSchema = new Schema({

...

email: {

**type**: **String**,

index: true

},

...

});

**Mongoose middleware**

Mongoose has **pre** and **post** middlewares which execute, as you may imagine, pre and post some operation. Template looks like this:

UserSchema.pre('save', **function**(next) {

**if** (something) {

next()

} **else** {

next(**new** **Error**('No can do, sir!'));

}

});

So, basically, before you save something to MongoDB some action is performed before that - you will use this later for hashing the password before saving it to the MongoDB.

*Mongoose lets you configure a lot more options like*[*default values*](http://mongoosejs.com/docs/2.7.x/docs/defaults.html)*,*[*custom getters/setters*](http://mongoosejs.com/docs/2.7.x/docs/getters-setters.html)*,*[*virtual attributes*](http://mongoosejs.com/docs/2.7.x/docs/virtuals.html)*,*[*custom methods and statics*](http://mongoosejs.com/docs/2.7.x/docs/methods-statics.html)*,*[*custom validation*](http://mongoosejs.com/docs/2.7.x/docs/validation.html)*,*[*custom references to other document inside a document*](http://mongoosejs.com/docs/2.7.x/docs/populate.html)*about which you can learn more on the respective links.*

**User authentication with Passport**



[Passport](http://passportjs.org/) is a Node.js authentication middleware that authenticates requests sent to your Express application. It offers both local authentication (username/password) and OAuth authentication (Facebook, Twitter, Github, Google, ... - full list [here](http://passportjs.org/guide/providers/).) methods called **strategies**.

In order to install Passport execute:

$ npm install passport --save

Now, add the following two lines in the **server.js** file:

passport = **require**('./config/passport');

...

**var** passport = passport();

Also, require it in the **config/express.js** file like this:

passport = require('passport');

//use this code before any route definitions

app.use(passport.initialize());

app.use(passport.session());

Now you need to install at least one **authentication strategy**, and here I'll show you three: local strategy (username/password) authentication, Facebook OAuth authentication and Twitter OAuth authentication with exact steps.

**Local authentication strategy**

First, install the local strategy authentication module:

$ npm install passport-local --save

Now, in the config **folder**, create a new folder called **strategies** and a new file **local.js** with the following code:

**var** passport = **require**('passport'),

LocalStrategy = **require**('passport-local').Strategy,

User = **require**('mongoose').model('User');

**module**.exports = **function**() {

passport.use(**new** LocalStrategy(**function**(username, password, done) {

User.findOne(

{username: username},

**function**(err, user) {

**if** (err) {

**return** done(err);

}

**if** (!user) {

**return** done(null, false, {message: 'Unknown user'});

}

**if** (!user.authenticate(password)) {

**return** done(null, false, {message: 'Invalid password'});

}

**return** done(null, user);

}

);

}));

};

Here you first require the Passport module, then the local strategy module's Strategy object, and finally your *User* Mongoose model. Then, you register the strategy using the **passport.use()** method that uses an instance of the LocalStrategy object. Inside the callback function you use the *User* Mongoose model to find a user with the passed username and try to authenticate it.

Now, create a **passport.js** file in the **config** folder and add this code:

**var** passport = **require**('passport'),

mongoose = **require**('mongoose');

**module**.exports = **function**() {

**var** User = mongoose.model('User');

passport.serializeUser(**function**(user, done) {

done(null, user.id);

});

passport.deserializeUser(**function**(id, done) {

User.findOne(

{\_id: id},

'-password',

**function**(err, user) {

done(err, user);

}

);

});

**require**('./strategies/local.js')();

};

With **passport.serializeUser()** and **passport.deserializeUser()** methods you defined how Passport will handle user serialization. When a user is authenticated, Passport will save its \_id property to the session. When the user object will be needed Passport will use the \_id property to fetch the user object from the database. Also, you included the local strategy  
configuration file so once you load the Passport configuration file in server.js, it then loads its strategies configuration file. The **-password** option is set so that Mongoose doesn't fetch the password field.

Next, you need to modify your *User* model to support Passport authentication, and to do that update**models/user.server.model.js** file to this code:

**var** mongoose = **require**('mongoose'),

crypto = **require**('crypto'),

Schema = mongoose.Schema;

**var** UserSchema = **new** Schema({

name: **String**,

email: **String**,

username: {

type: **String**,

trim: true,

unique: true

},

password: **String**,

provider: **String**,

providerId: **String**,

providerData: {},

todos: {}//we will use this in the next tutorial to store TODOs

});

UserSchema.pre('save',

**function**(next) {

**if** (**this**.password) {

**var** md5 = crypto.createHash('md5');

**this**.password = md5.update(**this**.password).digest('hex');

}

next();

}

);

UserSchema.methods.authenticate = **function**(password) {

**var** md5 = crypto.createHash('md5');

md5 = md5.update(password).digest('hex');

**return** **this**.password === md5;

};

UserSchema.statics.findUniqueUsername = **function**(username, suffix, callback) {

**var** \_this = **this**;

**var** possibleUsername = username + (suffix || '');

\_this.findOne(

{username: possibleUsername},

**function**(err, user) {

**if** (!err) {

**if** (!user) {

callback(possibleUsername);

}

**else** {

**return** \_this.findUniqueUsername(username, (suffix || 0) + 1, callback);

}

}

**else** {

callback(null);

}

}

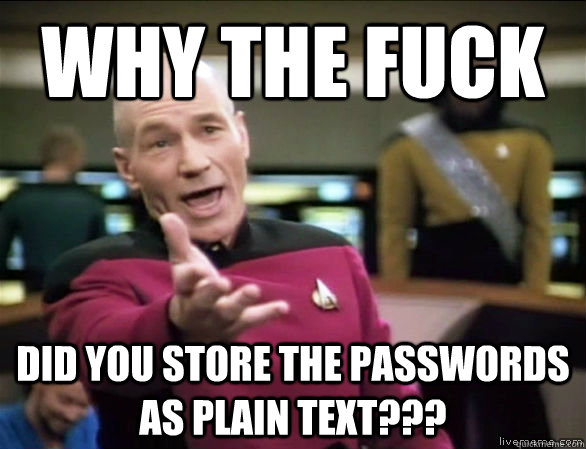
);

};

mongoose.model('User', UserSchema);

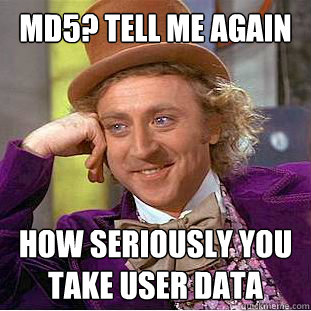
In this mumbo jumbo you've added three fields to your UserSchema object:

* **provider** property - the strategy used to register the user
* **providerId** property - the user identifier for the authentication strategy
* **providerData** property - you'll use it later to store the user object retrieved from OAuth providers



Next, you created a **pre-save middleware** (mentioned above in the Mongoose section) to handle the users' password hashing, since storing a clear text passwords in the database (any database for that matter) is a big fat NoNo. The pre-save middleware creates a MD5 hash of the password (using the [crypto](http://nodejs.org/api/crypto.html) module) and then it replaces the current user password with a hashed one.

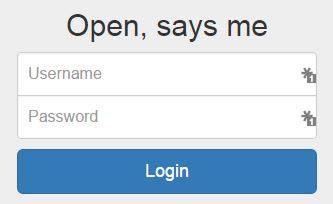
Before you jump the boat and go like:



you can use a more advanced example which uses salts [here](http://stackoverflow.com/questions/17201450/salt-and-hash-password-in-nodejs-w-crypto).

Also, you added an **authenticate()** method, which accepts a string password argument, which it then hashes and compares to the current users hashed password. Finally, you added the**findUniqueUsername()** static method, which is used to find an available unique username for new users (used later with OAuth authentication examples).

**Login page**



You need a login page where you'll give your users options to login in. You'll use a small [Bootstrap login](http://getbootstrap.com/examples/signin/)example to make it look nicer. Create a new **login.ejs** file in the **views** folder and paste the following code:

<!DOCTYPE html>

<**html**>

<**head**>

<**title**>

<%=title %>

</**title**>

<**link** href="http://getbootstrap.com/dist/css/bootstrap.min.css" rel="stylesheet">

<**link** href="http://getbootstrap.com/examples/signin/signin.css" rel="stylesheet">

</**head**>

<**body**>

<% **for**(var i **in** messages) { %>

<**div** class="flash">

<%= messages[i] %>

</**div**>

<% } %>

<**div** class="container">

<**form** class="form-signin" action="/login" method="post">

<**h2** class="form-signin-heading text-center">Open, says me</**h2**>

<**label** for="username" class="sr-only">Username</**label**>

<**input** type="text" name="username" id="username" class="form-control" placeholder="Username" required autofocus>

<**label** for="password" class="sr-only">Password</**label**>

<**input** type="password" name="password" id="password" class="form-control" placeholder="Password">

<**button** class="btn btn-lg btn-primary btn-block" type="submit">Login</**button**>

</**form**>

<**div** class="row">

<**div** class="center-block text-center">

<**a** href="/oauth/facebook">Login with Facebook</**a**> |

<**a** href="/oauth/twitter">Login with Twitter</**a**>

</**div**>

</**div**>

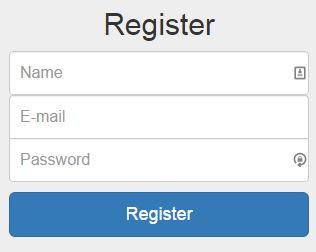
</**div**>

</**div**>

</**body**>

</**html**>

**Register page**



Create a new **register.ejs** file in the **views** folder and paste the following code:

<!DOCTYPE html>

<**html**>

<**head**>

<**title**>

<%=title %>

</**title**>

<**link** href="http://getbootstrap.com/dist/css/bootstrap.min.css" rel="stylesheet">

<**link** href="http://getbootstrap.com/examples/signin/signin.css" rel="stylesheet">

</**head**>

<**body**>

<% **for**(var i **in** messages) { %>

<**div** class="flash"><%= messages[i] %></**div**>

<% } %>

<**div** class="container">

<**form** class="form-signin" action="/register" method="post">

<**h2** class="form-signin-heading text-center">Register</**h2**>

<**label** for="username" class="sr-only">Username</**label**>

<**input** type="text" id="username" name="username" class="form-control" placeholder="Username" required autofocus>

<**label** for="name" class="sr-only">Name</**label**>

<**input** type="text" id="name" name="name" class="form-control" placeholder="Name" required autofocus>

<**label** for="email" class="sr-only">E-mail</**label**>

<**input** type="email" id="email" name="email" class="form-control" placeholder="E-mail" required>

<**label** for="password" class="sr-only">Password</**label**>

<**input** type="password" name="password" id="password" class="form-control" placeholder="Password" required>

<**button** class="btn btn-lg btn-primary btn-block" type="submit">Register</**button**>

</**form**>

<**div** class="row">

<**div** class="center-block text-center">

<**a** href="/oauth/facebook">Login with Facebook</**a**> |

<**a** href="/oauth/twitter">Login with Twitter</**a**>

</**div**>

</**div**>

</**div**>

</**body**>

</**html**>

One thing you should note in the login.ejs and register.ejs files are the **name** parameters of the input fields - they correspond to the Mongoose *User* model fields.

**Users controller modifications**

Change the **app/controllers/users.server.controller.js** file to look like this:

**var** User = **require**('mongoose').model('User'),

passport = **require**('passport');

**var** getErrorMessage = **function**(err) {

**var** message = '';

**if** (err.code) {

**switch** (err.code) {

**case** 11000:

**case** 11001:

message = 'Username already exists';

**break**;

**default**:

message = 'Something went wrong';

}

}

**else** {

**for** (**var** errName **in** err.errors) {

**if** (err.errors[errName].message)

message = err.errors[errName].message;

}

}

**return** message;

};

exports.renderLogin = **function**(req, res, next) {

**if** (!req.user) {

res.render('login', {

title: 'Log-in Form',

messages: req.flash('error') || req.flash('info')

});

}

**else** {

**return** res.redirect('/');

}

};

exports.renderRegister = **function**(req, res, next) {

**if** (!req.user) {

res.render('register', {

title: 'Register Form',

messages: req.flash('error')

});

}

**else** {

**return** res.redirect('/');

}

};

exports.register = **function**(req, res, next) {

**if** (!req.user) {

**var** user = **new** User(req.body);

**var** message = null;

user.provider = 'local';

user.save(**function**(err) {

**if** (err) {

**var** message = getErrorMessage(err);

req.flash('error', message);

**return** res.redirect('/register');

}

req.login(user, **function**(err) {

**if** (err)

**return** next(err);

**return** res.redirect('/');

});

});

}

**else** {

**return** res.redirect('/');

}

};

exports.logout = **function**(req, res) {

req.logout();

res.redirect('/');

};

The **register()** method uses your *User* model to create new users, so that it first creates a user object from the HTTP request body. Then, it tries to save it to MongoDB and if an error occurs, the register() method will use the **getErrorMessage()** method to fetch the errors. If the user was created successfully, the user session will be created using the **req.login()** method provided by Passport module. After the login operation is completed, a user object will be inside the req.user object.

**Le errors**

The [Connect-Flash](https://www.npmjs.com/package/connect-flash) module stores temporary messages in a session object called **flash**. Messages stored in the flash object will be cleared once they are presented to the user.

That's great stuff, install it:

npm install connect-flash --save

Then, add the following lines to your **express.js** file:

**var** flash = **require**('connect-flash');

app.**use**(flash());

The Connect-Flash module exposes the **req.flash()** method, which allows you to create and retrieve flash messages.

**Show me the routes**

You need to wire up the routes for login and register, so in **users.server.routes.js** add:

**app**.route('/users').**post**(users.create).get(users.**list**);

**app**.route('/users/:userId').get(users.**read**).put(users.**update**).delete(users.delete);

**app**.param('userId', users.userByID);

**app**.route('/register')

.get(users.renderRegister)

.**post**(users.register);

**app**.route('/login')

.get(users.renderLogin)

.**post**(passport.authenticate('**local**', {

successRedirect: '/',

failureRedirect: '/login',

failureFlash: true

}));

**app**.get('/logout', users.logout);

In the **index.server.controller.js** file add:

exports.render = **function**(req, res) {

res.render('index', {

title: 'MEAN MVC',

user: req.user ? req.user.username : ''

});

};

In **index.ejs** file add:

<% **if** ( user ) { %>

<**h2**>Hello <%=user%> </**h2**>

<**a** href="/logout">Logout</**a**>

<% } **else** { %>

<**a** href="/register">Register</**a**>

<**a** href="/login">Login</**a**>

<% } %>

If you run this with **node server** now, you will get the error '*Error: req.flash() requires sessions*'. Well, you can fix this quickly by installing the [sessions](https://github.com/expressjs/session) module:

$ npm install express-session --save

and require it in the **express.js** file and add the *use* statement:

**var** session = **require**('express-session');

app.**use**(session({

saveUninitialized: **true**,

resave: **true**,

secret: 'OurSuperSecretCookieSecret'

}));

**Facebook OAuth strategy**

[OAuth](http://oauth.net/) is an authentication protocol that allows users to register with your web application using an external *provider*, without the need to input their username and password.



OAuth is mainly used by social platforms (Facebook, Twitter, ...) to allow users to register with other websites using their social account.

In **app/controllers/users.server.controller.js** file add the following module method:

exports.saveOAuthUserProfile = **function**(req, profile, done) {

User.findOne({

provider: profile.provider,

providerId: profile.providerId

},

**function**(err, user) {

**if** (err) {

**return** done(err);

}

**else** {

**if** (!user) {

**var** possibleUsername = profile.username || ((profile.email) ? profile.email.split('@')[0] : '');

User.findUniqueUsername(possibleUsername, null, **function**(availableUsername) {

profile.username = availableUsername;

user = **new** User(profile);

user.save(**function**(err) {

**if** (err) {

**var** message = \_this.getErrorMessage(err);

req.flash('error', message);

**return** res.redirect('/signup');

}

**return** done(err, user);

});

});

}

**else** {

**return** done(err, user);

}

}

}

);

};

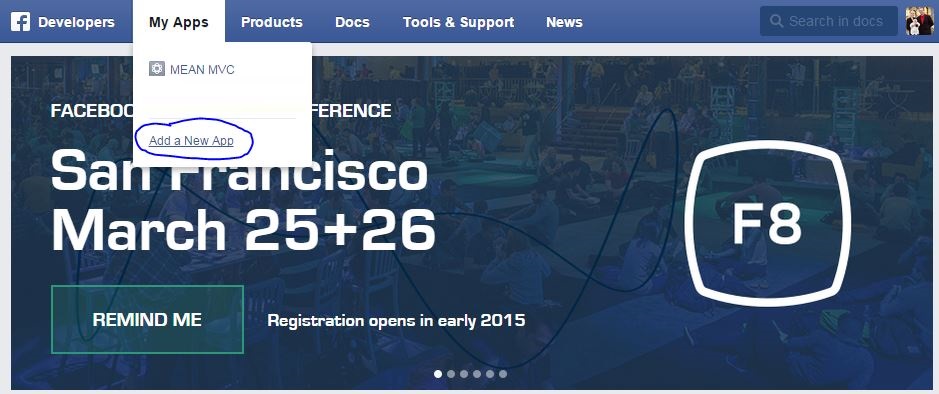
This method accepts a user profile, and then looks for an existing user with *providerId* and *provider*properties. If the user is found, it calls the done() callback method with the user's MongoDB document. However, if an existing user isn't found, it will find a unique username using the \_User \_model's findUniqueUsername() static method and save a new user instance. If an error occurs, the saveOAuthUserProfile() method will use the req.flash() and getErrorMessage() methods to report the error; otherwise, it will pass the user object to the done() callback method.

Now, install the Facebook strategy module:

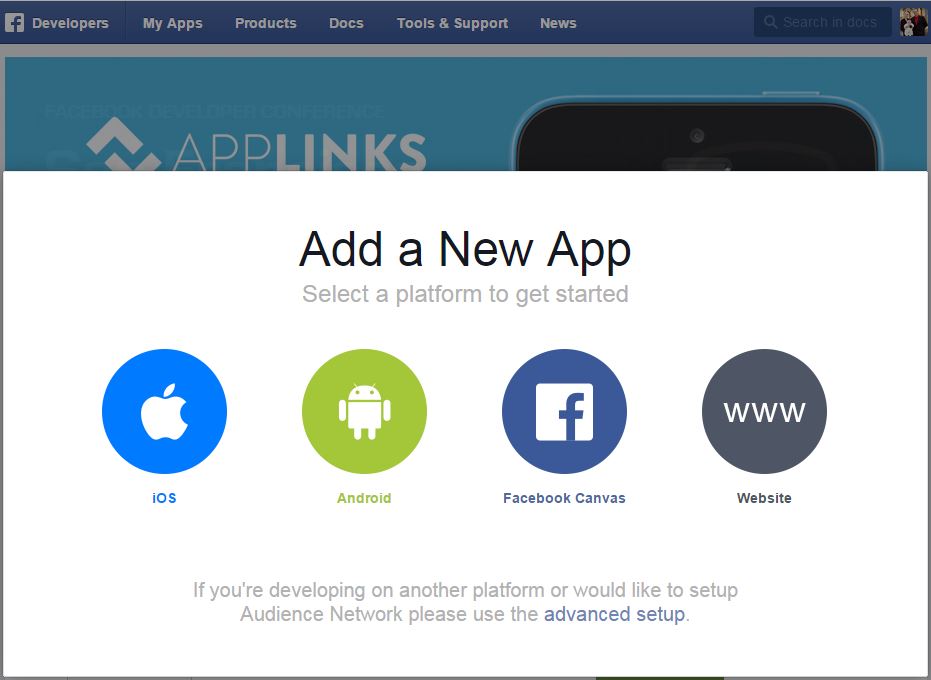
npm install passport-facebook --save

**Create a Facebook application**

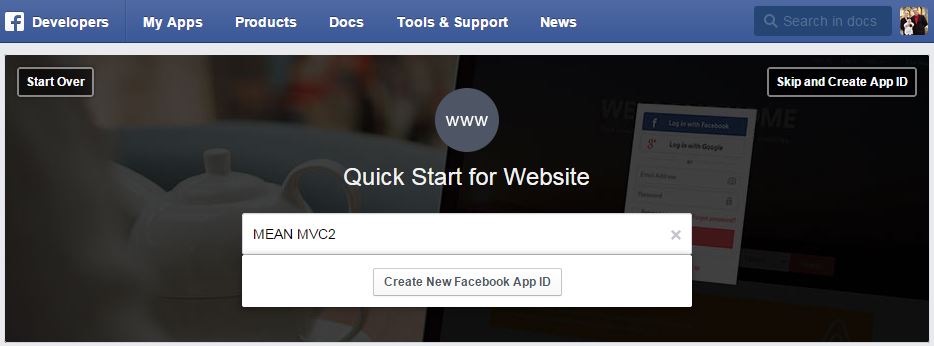
Before you can configure your Facebook strategy, you have to go to Facebook's developer home page at<https://developers.facebook.com/> and create a new Facebook application:



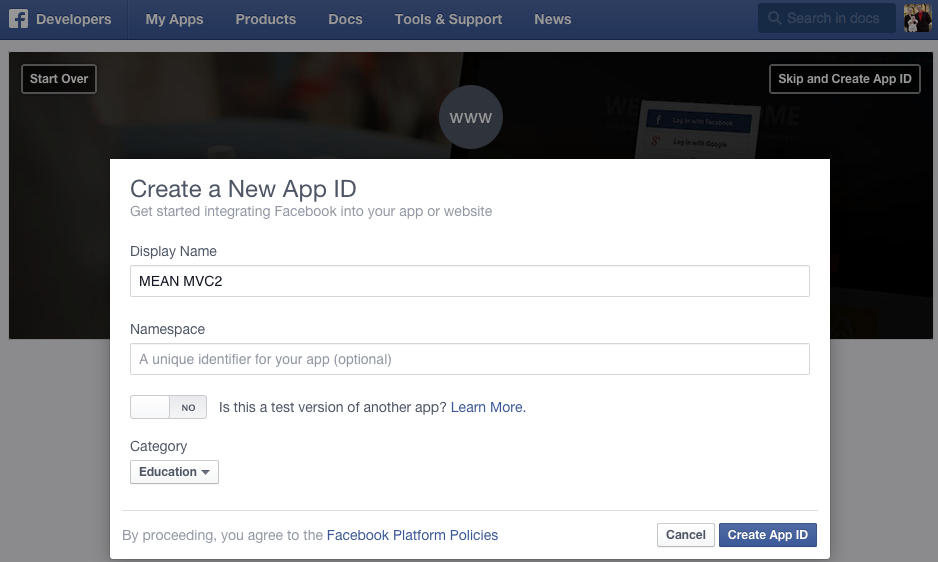
On the next dialog select Website:



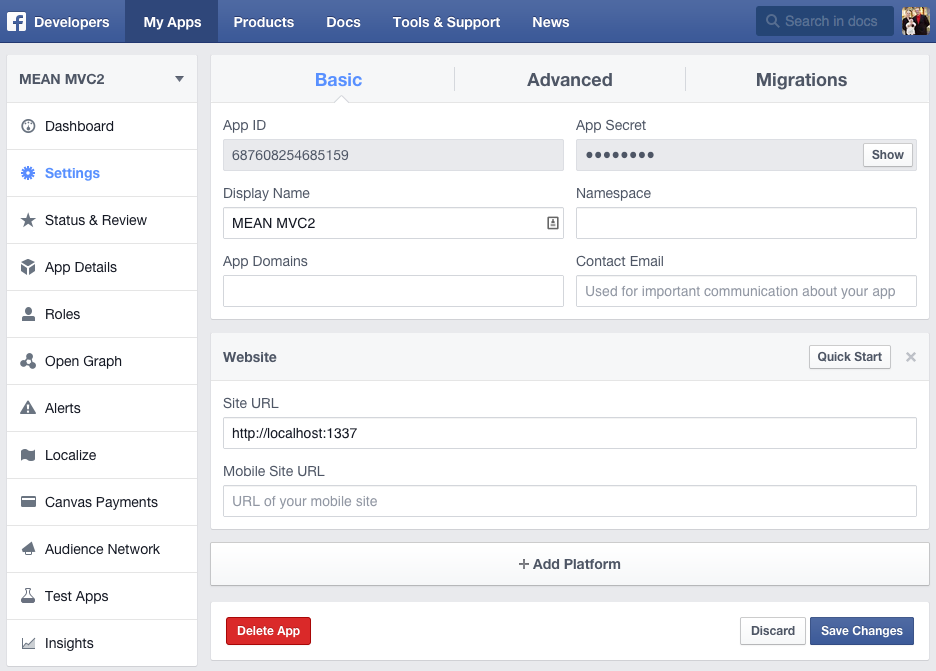
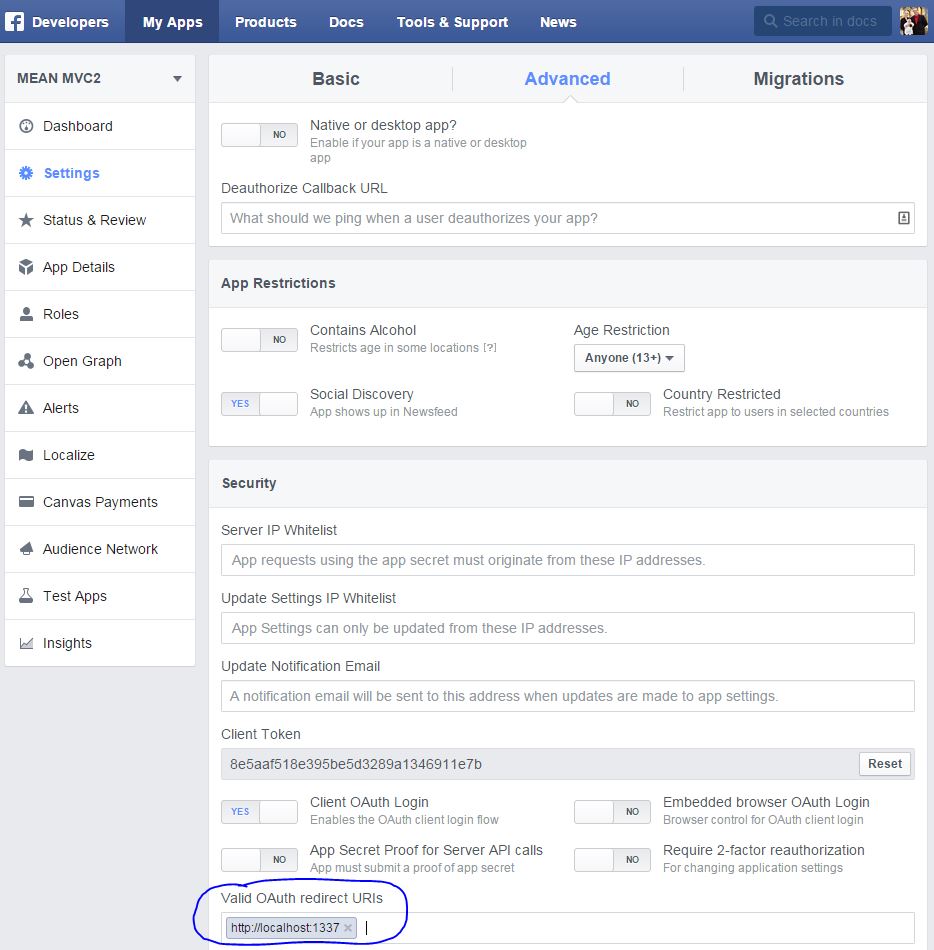
Click on the *Skip and Create App ID* button:



On the next dialog enter the *Display Name* of the application, select the *Category* and click on the \_Create App ID \_button:



You need to do few more things:

1. Click on the **Settings**, then **+ Add Platform**and select **Website**. Enter [http://localhost:1337](http://localhost:1337/) as the**Site URL** and **Save Changes.**  
   
2. Take special note of the **App ID** and **App Secret** values, you'll need them later in code.
3. Click on the **Advanced** tab and insert [http://localhost:1337](http://localhost:1337/) for the **Valid OAuth redirect URIs**and**Save Changes**:  
   

**Set the strategy**

Go to the **config/env/development.js** file and change it as follows:

**var** port = 1337;

**module**.exports = {

port: port,

db: 'mongodb://localhost/todos',

facebook: {

clientID: 'App ID',

clientSecret: 'App Secret',

callbackURL: 'http://localhost:'+ port +'/oauth/facebook/callback'

}

};

You need to enter your own App ID as cliendID, and App secret as clientSecret (values which you got when you made your Facebook applicaiton).

Now, go to your **config/strategies** folder, and create a new file named **facebook.js** and place the following code:

**var** passport = **require**('passport'),

url = **require**('url'),

FacebookStrategy = **require**('passport-facebook').Strategy,

config = **require**('../config'),

users = **require**('../../app/controllers/users.server.controller');

**module**.exports = **function**() {

passport.use(**new** FacebookStrategy({

clientID: config.facebook.clientID,

clientSecret: config.facebook.clientSecret,

callbackURL: config.facebook.callbackURL,

passReqToCallback: true

},

**function**(req, accessToken, refreshToken, profile, done) {

**var** providerData = profile.\_json;

providerData.accessToken = accessToken;

providerData.refreshToken = refreshToken;

**var** providerUserProfile = {

name: profile.name.givenName,

email: profile.emails[0].value,

username: profile.username,

provider: 'facebook',

providerId: profile.id,

providerData: providerData

};

users.saveOAuthUserProfile(req, providerUserProfile, done);

}));

};

In this code you first require the needed modules, and then you register the strategy using the passport.use() method where you create an instance of a FacebookStrategy object which you populate with the needed parameters. Inside the callback function, you create a new user object using the Facebook profile information and the controller's saveOAuthUserProfile() method.

Finally, in the **config/passport.js** file require this strategy:

**require**('./strategies/facebook.js')();

**Set the routes**

Now, all that you have to do is set the routes needed to authenticate users via Facebook and include links to those routes in your login and register pages.

In **app/routes/users.server.routes.js** file add the following code after the local strategy routes definition:

app.get('/oauth/facebook', passport.authenticate('facebook', {

failureRedirect: '/login',

scope:['email']

}));

app.get('/oauth/facebook/callback', passport.authenticate('facebook', {

failureRedirect: '/login',

successRedirect: '/',

scope:['email']

}));

*The scope:['email'] is important as per*[*these instructions*](http://stackoverflow.com/questions/22880876/passport-facebook-authentication-is-not-providing-email-for-all-fbaccounts)*.*

The first route uses the passport.authenticate() method to start the user authentication process, and the second route uses the same method, but this time to finish the authentication process once the user has linked his Facebook profile.

**Set the links**

Finally, in **app/views/register.ejs** and **app/views/login.ejs**files add the following line of code right before the closing BODY tag:

<a href="/oauth/facebook">Login with Facebook</a>

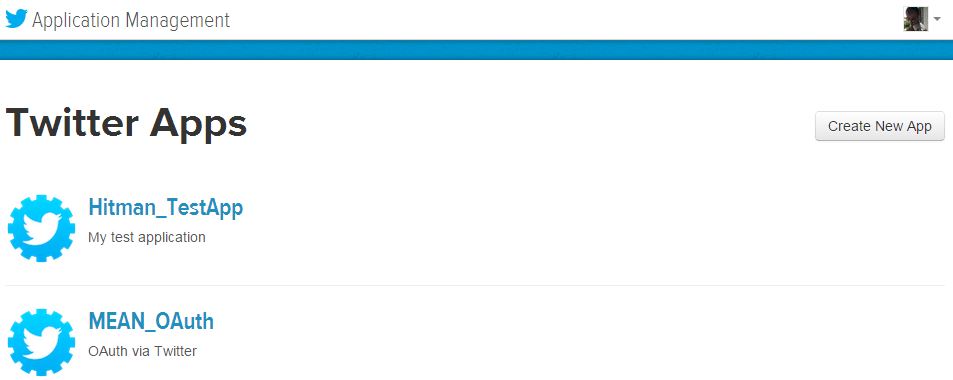
**Twitter OAuth strategy**

First, install the Twitter strategy module:

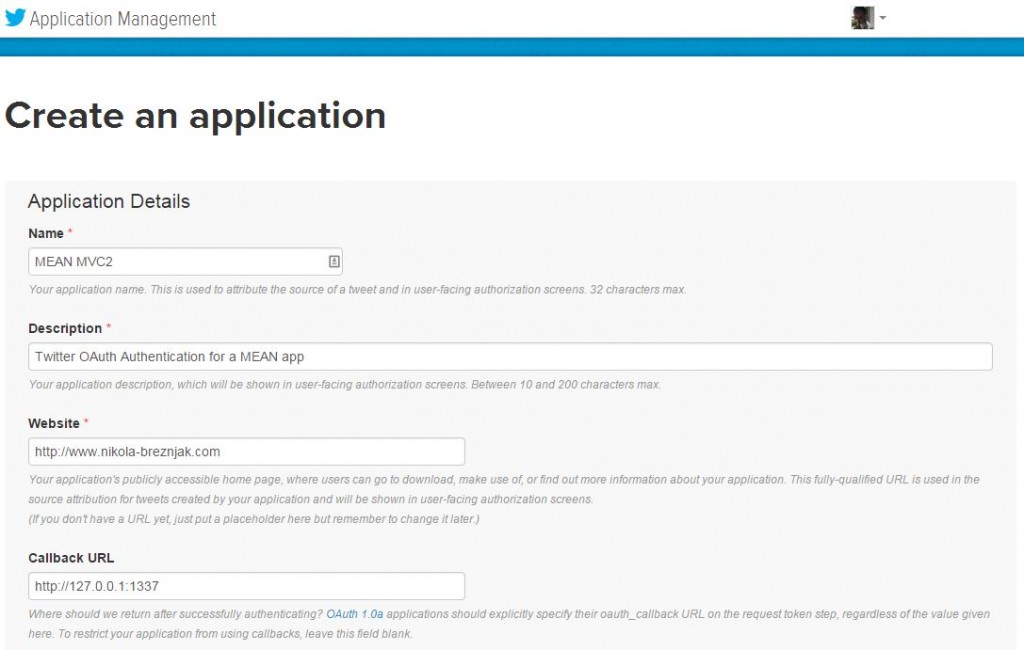
npm install passport-twitter --save

**Create a Twitter application**

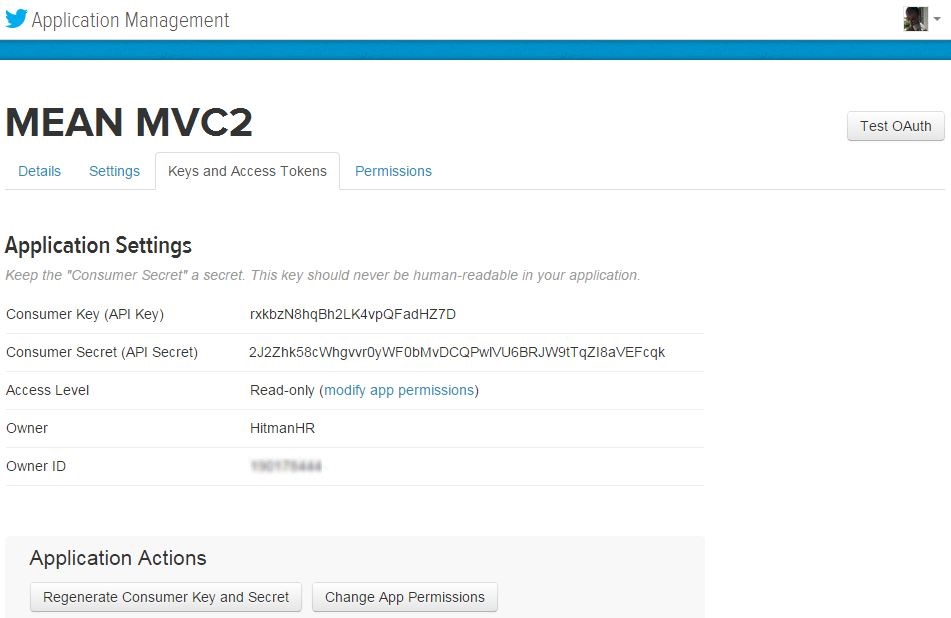
Go to [Twitter developer page](https://apps.twitter.com/) and click on the \_Create New App \_button:



Set *Name*, *Description*, *Website* and most importantly *Callback URL*:



Click on the *Keys and Access Tokens* tab:



**Set the strategy**

Take note of the *Consumer Key (API key)* and *Consumer Secret (API secret)* values; enter them in the**config/env/development.js** file:

twitter: {

clientID: 'API key',

clientSecret: 'API secret',

callbackURL: 'http://localhost:1337/oauth/twitter/callback'

}

Now, create a new **twitter.js** file in the **config/strategies** folder and put the following code inside:

**var** passport = **require**('passport'),

url = **require**('url'),

TwitterStrategy = **require**('passport-twitter').Strategy,

config = **require**('../config'),

users = **require**('../../app/controllers/users.server.controller');

**module**.exports = **function**() {

passport.use(**new** TwitterStrategy({

consumerKey: config.twitter.clientID,

consumerSecret: config.twitter.clientSecret,

callbackURL: config.twitter.callbackURL,

passReqToCallback: true

},

**function**(req, token, tokenSecret, profile, done) {

**var** providerData = profile.\_json;

providerData.token = token;

providerData.tokenSecret = tokenSecret;

**var** providerUserProfile = {

fullName: profile.displayName,

username: profile.username,

provider: 'twitter',

providerId: profile.id,

providerData: providerData

};

users.saveOAuthUserProfile(req, providerUserProfile, done);

}));

};

In this code you first require the needed modules, and then you register the strategy using the passport.use() method where you create an instance of a TwitterStrategy object which you populate with the needed parameters. Inside the callback function, you created a new user object using the Twitter profile information and the controller's saveOAuthUserProfile() method.

In the **config/passport.js** file add the following line (after the line where you require Facebook strategy):

**require**('./strategies/twitter.js')();

**Set the routes**

Now add the routes in the **app/routes/users.server.routes.js** file:

app.get('/oauth/twitter', passport.authenticate('twitter', {

failureRedirect: '/login'

}));

app.get('/oauth/twitter/callback', passport.authenticate('twitter', {

failureRedirect: '/login',

successRedirect: '/'

}));

Same as in the Facebook case, the first route uses the passport.authenticate() method to start the user authentication process, and the second route uses the same method, but this time to finish the authentication process once the user has linked his Twitter profile.

**Set the links**

Finally, in **app/views/register.ejs** and **app/views/login.ejs**files add the following line of code right before the closing BODY tag:

<a href="/oauth/twitter">Login with Twitter</a>

**Other OAuth providers**

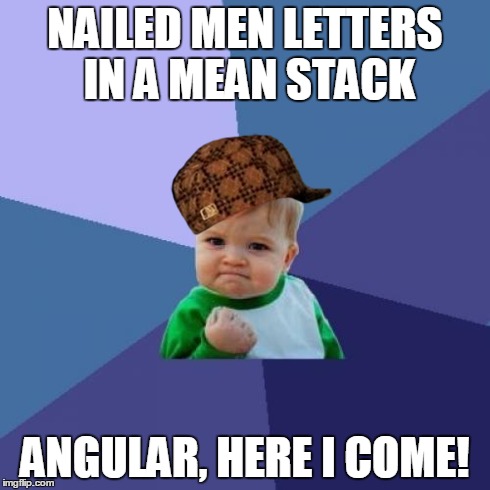
I think you go the gist on how to add OAuth to your MEAN application. Passport has similar support for many additional OAuth providers, and to learn more visit http://passportjs.org/guide/providers/.

**Le wrap up**

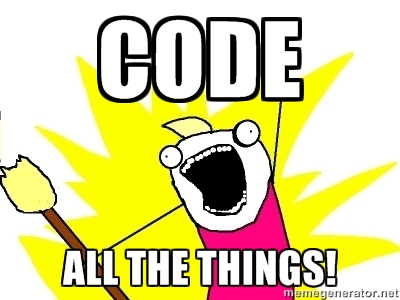
Wow, it's been a ride! Since you made it to the end, congratulations:



In the next and last part of the MEAN puzzle, you'll tackle Angular, where you'll make an interface to save your tasks by using the TODO MVC project from the [first post](https://hackhands.com/how-to-get-started-on-the-mean-stack/).



Ok, enough memes, take a break until next tutorial. I'm kidding - you go and



until you get this in your muscle memory!