
Passport-Local Mongoose

Passport-Local Mongoose is a Mongoose plugin that simplifies building username and password login with Passport.



Tutorials

Michael Herman gives a comprehensible walk through for setting up mongoose, passport, passport-local and passport-local-mongoose for user authentication in his blog post [User Authentication With Passport.js](#)

Installation

```
1 > npm install passport-local-mongoose
```

Passport-Local Mongoose does not require `passport` or `mongoose` dependencies directly but expects you to have these dependencies installed.

In case you need to install the whole set of dependencies

```
1 > npm install passport mongoose passport-local-mongoose
```

Usage

Plugin Passport-Local Mongoose

First you need to plugin Passport-Local Mongoose into your User schema

```
1 const mongoose = require('mongoose');  
2 const Schema = mongoose.Schema;  
3 const passportLocalMongoose = require('passport-local-mongoose');  
4  
5 const User = new Schema({});  
6  
7 User.plugin(passportLocalMongoose);  
8  
9 module.exports = mongoose.model('User', User);
```

You're free to define your User how you like. Passport-Local Mongoose will add a username, hash and salt field to store the username, the hashed password and the salt value.

Additionally, Passport-Local Mongoose adds some methods to your Schema. See the API Documentation section for more details.

Configure Passport/Passport-Local

You should configure Passport/Passport-Local as described in the Passport Guide.

Passport-Local Mongoose supports this setup by implementing a `LocalStrategy` and `serializeUser`/`deserializeUser` functions.

To setup Passport-Local Mongoose use this code

```
1 // requires the model with Passport-Local Mongoose plugged in
2 const User = require('./models/user');
3
4 // use static authenticate method of model in LocalStrategy
5 passport.use(new LocalStrategy(User.authenticate()));
6
7 // use static serialize and deserialize of model for passport session support
8 passport.serializeUser(User.serializeUser());
9 passport.deserializeUser(User.deserializeUser());
```

Make sure that you have mongoose connected to mongodb and you're done.

Simplified Passport/Passport-Local Configuration Starting from version 0.2.1, passport-local-mongoose adds a helper method `createStrategy` as static method to your schema. The `createStrategy` is responsible to setup passport-local `LocalStrategy` with the correct options.

```
1 const User = require('./models/user');
2
3 // CHANGE: USE "createStrategy" INSTEAD OF "authenticate"
4 passport.use(User.createStrategy());
5
6 passport.serializeUser(User.serializeUser());
7 passport.deserializeUser(User.deserializeUser());
```

The reason for this functionality is that when using the `usernameField` option to specify an alternative usernameField name, for example “email” passport-local would still expect your frontend login form to contain an input field with name “username” instead of email. This can be configured for passport-local but this is double the work. So we got this shortcut implemented.

Async/Await

Starting from version 5.0.0, passport-local-mongoose is async/await enabled by returning Promises for all instance and static methods except `serializeUser` and `deserializeUser`.

```
1 const user = new DefaultUser({username: 'user'});
2 await user.setPassword('password');
3 await user.save();
4 const { user } = await DefaultUser.authenticate('user', 'password');
```

Options

When plugging in Passport-Local Mongoose plugin, additional options can be provided to configure the hashing algorithm.

```
1 User.plugin(passportLocalMongoose, options);
```

Main Options

- `saltlen`: specifies the salt length in bytes. Default: 32
- `iterations`: specifies the number of iterations used in pbkdf2 hashing algorithm. Default: 25000
- `keylen`: specifies the length in byte of the generated key. Default: 512
- `digestAlgorithm`: specifies the pbkdf2 digest algorithm. Default: sha256. (get a list of supported algorithms with `crypto.getHashes()`)
- `interval`: specifies the interval in milliseconds between login attempts, which increases exponentially based on the number of failed attempts, up to `maxInterval`. Default: 100
- `maxInterval`: specifies the maximum amount of time an account can be locked. Default 300000 (5 minutes)
- `usernameField`: specifies the field name that holds the username. Defaults to 'username'. This option can be used if you want to use a different field to hold the username for example "email".
- `usernameUnique`: specifies if the username field should be enforced to be unique by a mongodb index or not. Defaults to true.
- `saltField`: specifies the field name that holds the salt value. Defaults to 'salt'.
- `hashField`: specifies the field name that holds the password hash value. Defaults to 'hash'.
- `attemptsField`: specifies the field name that holds the number of login failures since the last successful login. Defaults to 'attempts'.
- `lastLoginField`: specifies the field name that holds the timestamp of the last login attempt. Defaults to 'last'.

-
- `selectFields`: specifies the fields of the model to be selected from mongodb (and stored in the session). Defaults to 'undefined' so that all fields of the model are selected.
 - `usernameCaseInsensitive`: specifies the usernames to be case insensitive. Defaults to 'false'.
 - `usernameLowerCase`: convert username field value to lower case when saving an querying. Defaults to 'false'.
 - `populateFields`: specifies fields to populate in `findByUsername` function. Defaults to 'undefined'.
 - `encoding`: specifies the encoding the generated salt and hash will be stored in. Defaults to 'hex'.
 - `limitAttempts`: specifies whether login attempts should be limited and login failures should be penalized. Default: false.
 - `maxAttempts`: specifies the maximum number of failed attempts allowed before preventing login. Default: Infinity.
 - `unlockInterval`: specifies the interval in milliseconds, which is for unlock user automatically after the interval is reached. Defaults to 'undefined' which means deactivated.
 - `passwordValidator`: specifies your custom validation function for the password in the form:

```
js passwordValidator = function(password,cb){ if (someValidationErrorExists(password)){ return cb('this is my custom validation error message')} //return an empty cb()on success return cb() }
```

 Default: validates non-empty passwords.
 - `passwordValidatorAsync`: specifies your custom validation function for the password with promises in the form:

```
js passwordValidatorAsync = function(password){ return someAsyncValidation(password).catch(function(err){ return Promise.reject(err)}) }
```
 - `usernameQueryFields`: specifies alternative fields of the model for identifying a user (e.g. email).
 - `findByUsername`: Specifies a query function that is executed with query parameters to restrict the query with extra query parameters. For example query only users with field "active" set to `true`. Default:

```
function(model, queryParameters){ return model.findOne(queryParameters); }
```

. See the examples section for a use case.

Attention! Changing any of the hashing options (saltlen, iterations or keylen) in a production environment will prevent existing users from authenticating!

Error Messages Override default error messages by setting `options.errorMessages`.

- `MissingPasswordError`: 'No password was given'
- `AttemptTooSoonError`: 'Account is currently locked. Try again later'

-
- `TooManyAttemptsError`: 'Account locked due to too many failed login attempts'
 - `NoSaltValueStoredError`: 'Authentication not possible. No salt value stored'
 - `IncorrectPasswordError`: 'Password or username are incorrect'
 - `IncorrectUsernameError`: 'Password or username are incorrect'
 - `MissingUsernameError`: 'No username was given'
 - `UserExistsError`: 'A user with the given username is already registered'

Hash Algorithm

Passport-Local Mongoose use the pbkdf2 algorithm of the node crypto library. Pbkdf2 was chosen because platform independent (in contrary to bcrypt). For every user a generated salt value is saved to make rainbow table attacks even harder.

Examples

For a complete example implementing a registration, login and logout see the login example.

API Documentation

Instance methods

setPassword(password, [cb]) Sets a user password. Does not save the user object. If no callback `cb` is provided a `Promise` is returned.

changePassword(oldPassword, newPassword, [cb]) Changes a user's password hash and salt, resets the user's number of failed password attempts and saves the user object (everything only if oldPassword is correct). If no callback `cb` is provided a `Promise` is returned. If oldPassword does not match the user's old password, an `IncorrectPasswordError` is passed to `cb` or the `Promise` is rejected.

authenticate(password, [cb]) Authenticates a user object. If no callback `cb` is provided a `Promise` is returned.

resetAttempts([cb]) Resets a user's number of failed password attempts and saves the user object. If no callback `cb` is provided a `Promise` is returned. This method is only defined if `options.limitAttempts` is `true`.

Callback Arguments

- `err`
 - null unless the hashing algorithm throws an error
- `thisModel`
 - the model getting authenticated **if** authentication was successful otherwise false
- `passwordErr`
 - an instance of `AuthenticationError` describing the reason the password failed, else undefined.

Using `setPassword()` will only update the document's password fields, but will not save the document. To commit the changed document, remember to use Mongoose's `document.save()` after using `setPassword()`.

Error Handling

- `IncorrectPasswordError`: specifies the error message returned when the password is incorrect. Defaults to 'Incorrect password'.
- `IncorrectUsernameError`: specifies the error message returned when the username is incorrect. Defaults to 'Incorrect username'.
- `MissingUsernameError`: specifies the error message returned when the username has not been set during registration. Defaults to 'Field %s is not set'.
- `MissingPasswordError`: specifies the error message returned when the password has not been set during registration. Defaults to 'Password argument not set!'.
- `UserExistsError`: specifies the error message returned when the user already exists during registration. Defaults to 'User already exists with name %s'.
- `NoSaltValueStored`: Occurs in case no salt value is stored in the MongoDB collection.
- `AttemptTooSoonError`: Occurs if the option `limitAttempts` is set to true and a login attempt occurs while the user is still penalized.
- `TooManyAttemptsError`: Returned when the user's account is locked due to too many failed login attempts.

All those errors inherit from `AuthenticationError`, if you need a more general error class for checking.

Static methods

Static methods are exposed on the model constructor. For example to use `createStrategy` function use

```
1 const User = require('./models/user');  
2 User.createStrategy();
```

- `authenticate()` Generates a function that is used in Passport's `LocalStrategy`
- `serializeUser()` Generates a function that is used by Passport to serialize users into the session
- `deserializeUser()` Generates a function that is used by Passport to deserialize users into the session
- `register(user, password, cb)` Convenience method to register a new user instance with a given password. Checks if username is unique. See login example.
- `findByUsername()` Convenience method to find a user instance by its unique username.
- `createStrategy()` Creates a configured passport-local `LocalStrategy` instance that can be used in passport.

Examples

Allow only “active” users to authenticate

First, we define a schema with an additional field `active` of type Boolean.

```
1 const UserSchema = new Schema({  
2   active: Boolean  
3 });
```

When plugging in Passport-Local Mongoose, we set `usernameUnique` to avoid creating a unique mongodb index on field `username`. To avoid non active users being queried by mongodb, we can specify the option `findByUsername` that allows us to restrict a query. In our case we want to restrict the query to only query users with field `active` set to `true`. The `findByUsername` MUST return a Mongoose query.

```
1 UserSchema.plugin(passportLocalMongoose, {  
2   // Set usernameUnique to false to avoid a mongodb index on the  
3   // username column!  
4   usernameUnique: false,  
5   findByUsername: function(model, queryParameters) {  
6     // Add additional query parameter - AND condition - active: true  
7     queryParameters.active = true;
```

```
8     return model.findOne(queryParameters);
9   }
10  });
```

To test the implementation, we can simply create (register) a user with field `active` set to `false` and try to authenticate this user in a second step:

```
1  const User = mongoose.model('Users', UserSchema);
2
3  User.register({username: 'username', active: false}, 'password',
4    function(err, user) {
5      if (err) { ... }
6
7      const authenticate = User.authenticate();
8      authenticate('username', 'password', function(err, result) {
9        if (err) { ... }
10
11        // Value 'result' is set to false. The user could not be
12        // authenticated since the user is not active
13      });
14    });
```

Updating from 1.x to 2.x

The default digest algorithm was changed due to security implications from **sha1** to **sha256**. If you decide to upgrade a production system from 1.x to 2.x, your users **will not be able to login** since the digest algorithm was changed! In these cases plan some migration strategy and/or use the **sha1** option for the digest algorithm.

License

Passport-Local Mongoose is licensed under the MIT license.