User Authentication with Passport and Koa

Last updated: Jan 2, 2018 • node, koa, auth, mocha, testing

Passport is a library that provides a simple authentication middleware for Node.js.

This tutorial looks at how to set up a local authentication strategy with Node, Koa, and koa-passport, where users can sign up and log in using a username and password. We'll also use Postgres for storing user information and Redis for session management.

Last updated on July, 25 2018 to update a failing test.

Parts

This article is part of a 4-part Koa and Sinon series...

1. Building a RESTful API with Koa and Postgres
2. Stubbing HTTP Requests with Sinon
3. User Authentication with Passport and Koa (this article)
4. Stubbing Node Authentication Middleware with Sinon

Main NPM Dependencies

1. Koa v2.3.0
2. Mocha v3.5.0

7. koa-router v7.2.1
8. koa-bodyparser v4.2.0
9. koa-passport v4.0.1
10. koa-session v5.5.1
11. passport-local v1.0.0
12. bcrypt.js v2.4.3
13. koa-redis v3.1.1

Contents

Objectives

By the end of this tutorial, you will be able to...

1. Discuss the overall client/server authentication workflow
2. Add Passport and passport-local to a Koa app
3. Configure bcrypt.js for salting and hashing passwords
4. Practice test driven development
5. Register and authenticate a user
6. Utilize sessions to store user information via koa-session
7. Explain why you may want to use an external session store to store session data
8. Set up an external session store with Redis
9. Render HTML pages via server-side templating

Project Setup

Start by cloning down the base Koa project:

Then, check out the v2 tag to the master branch and install the dependencies:

```
$ git checkout tags/v2 -b master
$ npm install
```

Take a quick look at the code along with the project structure:

```
├── knexfile.js
├── package.json
├── src
│   └── server
│       ├── db
│       │   ├── connection.js
│       │   ├── migrations
│       │   │   └── 20170817152841_movies.js
│       │   ├── queries
│       │   │   └── movies.js
│       │   └── seeds
│       │       └── movies_seed.js
│       ├── index.js
│       └── routes
│           ├── index.js
│           └── movies.js
└── test
    ├── routes.index.test.js
    ├── routes.movies.test.js
    └── sample.test.js
```

This is just a basic RESTful API, with the following routes:

URL	HTTP Verb	Action
/api/v1/movies	GET	Return ALL movies
/api/v1/movies/:id	GET	Return a SINGLE movie
/api/v1/movies	POST	Add a movie
/api/v1/movies/:id	PUT	Update a movie
/api/v1/movies/:id	DELETE	Delete a movie

> *Want to learn how to build this project? Review the Building a RESTful API With Koa and Postgres blog post.*

```
$ psql

# CREATE DATABASE koa_api;
CREATE DATABASE
# CREATE DATABASE koa_api_test;
CREATE DATABASE
# \q
```

Ensure the tests pass:

```
$ npm test

Server listening on port: 1337
  routes : index
    GET /
      ✓ should return json

  routes : movies
    GET /api/v1/movies
      ✓ should return all movies
    GET /api/v1/movies/:id
      ✓ should respond with a single movie
      ✓ should throw an error if the movie does not exist
    POST /api/v1/movies
      ✓ should return the movie that was added
      ✓ should throw an error if the payload is malformed
    PUT /api/v1/movies
      ✓ should return the movie that was updated
      ✓ should throw an error if the movie does not exist
    DELETE /api/v1/movies/:id
      ✓ should return the movie that was deleted
      ✓ should throw an error if the movie does not exist

  Sample Test
    ✓ should pass


  11 passing (624ms)
```

Apply the migrations, and seed the database:

```
$ knex migrate:latest --env development
$ knex seed:run --env development
```

Run the Koa server, via `npm start`, and navigate to http://localhost:1337/api/v1/movies. You should see something similar to:

```
      "name": "The Land Before Time",
      "genre": "Fantasy",
      "rating": 7,
      "explicit": false
    },
    {
      "id": 2,
      "name": "Jurassic Park",
      "genre": "Science Fiction",
      "rating": 9,
      "explicit": true
    },
    {
      "id": 3,
      "name": "Ice Age: Dawn of the Dinosaurs",
      "genre": "Action/Romance",
      "rating": 5,
      "explicit": false
    }
  ]
}
```

User Model

Generate a new migration template for the user model:

```
$ knex migrate:make users
```

Then update the newly created file:

```
exports.up = (knex, Promise) => {
  return knex.schema.createTable('users', (table) => {
    table.increments();
    table.string('username').unique().notNullable();
    table.string('password').notNullable();
  });
};

exports.down = (knex, Promise) => {
  return knex.schema.dropTable('users');
};
```

Apply the migration against the development database:

```
$ knex migrate:latest --env development
$ npm install koa-passport@4.0.1 --save
```

Then, update *src/server/index.js* to add Passport to the app middleware along with koa-session, which is used for managing sessions:

```
const Koa = require('koa');
const bodyParser = require('koa-bodyparser');
const session = require('koa-session');
const passport = require('koa-passport');

const indexRoutes = require('./routes/index');
const movieRoutes = require('./routes/movies');

const app = new Koa();
const PORT = process.env.PORT || 1337;

// sessions
app.keys = ['super-secret-key'];
app.use(session(app));

// body parser
app.use(bodyParser());

// authentication
require('./auth');
app.use(passport.initialize());
app.use(passport.session());

// routes
app.use(indexRoutes.routes());
app.use(movieRoutes.routes());

// server
const server = app.listen(PORT, () => {
  console.log(`Server listening on port: ${PORT}`);
});

module.exports = server;
```

> *In production, make sure to update the secret key, `app.keys`. For example, you can use Python to generate a secure key:*
>
> ```
> $ python3
> >> import os
> >> os.urandom(24)
> b'3\xa5\xfa\xc6\xfb\x0e\x1dA\x19-U\x15Y\x9e2]\x92/\x97\x8d\xecsJ\xb7'
> ```

Sessions are stored in a cookie by default on the client-side, unencrypted. We'll stick with this for now, just to get things up and running, but we'll refactor and add Redis before all is said and done.

Before moving on, let's handle serializing and de-serializing the user information to the session. Create a new file called *auth.js* in "src/server":

```
const passport = require('koa-passport');
const knex = require('./db/connection');

passport.serializeUser((user, done) => { done(null, user.id); });

passport.deserializeUser((id, done) => {
  return knex('users').where({id}).first()
  .then((user) => { done(null, user); })
  .catch((err) => { done(err,null); });
});
```

Passport Local Strategy

Next, install the passport-local strategy, which is used for authenticating with a username and password:

```
$ npm install passport-local@1.0.0 --save
```

Update *auth.js* like so:

```
const passport = require('koa-passport');
const LocalStrategy = require('passport-local').Strategy;

const knex = require('./db/connection');

const options = {};

passport.serializeUser((user, done) => { done(null, user.id); });

passport.deserializeUser((id, done) => {
  return knex('users').where({id}).first()
  .then((user) => { done(null, user); })
  .catch((err) => { done(err,null); });
});

passport.use(new LocalStrategy(options, (username, password, done) => {
  knex('users').where({ username }).first()
  .then((user) => {
    }
  })
  .catch((err) => { return done(err); });
}));
```

Here, we check if the user exists and the password matches what's in the database and then pass the results back to Passport via the callback:

- *Does the username exist?*
  - No? then `false` is returned
  - Yes? *Does the password match?*
    - No? `false` is returned
    - Yes? The user object is returned and then the `id` is serialized to the session

> *You probably noticed that we are checking that the provided password is literally the same as the password pulled from the database, so we are storing the password in plain text. We'll update this after we add the main routes.*

Routes and Tests

Like the majority of my tutorials, we'll write tests first. That said, we will *only* be testing the happy paths. It's up to you to add tests for handling errors.

Routes:

URL	HTTP Verb	Authenticated?	Result
/auth/register	GET	No	Render the register view
/auth/register	POST	No	Register a new user
/auth/login	GET	No	Render the login view
/auth/login	POST	No	Log a user in
/auth/status	GET	Yes	Render the status page
/auth/logout	GET	Yes	Log a user out

Full Authentication flow:

1. The end user provides a username and a password and the credentials are sent to the server-side

Create a new file called *routes.auth.test.js* in "test":

```
process.env.NODE_ENV = 'test';

const chai = require('chai');
const should = chai.should();
const chaiHttp = require('chai-http');
chai.use(chaiHttp);

const server = require('../src/server/index');
const knex = require('../src/server/db/connection');

describe('routes : auth', () => {

  beforeEach(() => {
    return knex.migrate.rollback()
    .then(() => { return knex.migrate.latest(); })
    .then(() => { return knex.seed.run(); });
  });

  afterEach(() => {
    return knex.migrate.rollback();
  });

});
```

This is just a boilerplate for the tests.

Register - GET

This route serves up a view with an HTML form for users to register with.

Test

Start with a test:

```
describe('GET /auth/register', () => {
  it('should render the register view', (done) => {
    chai.request(server)
    .get('/auth/register')
    .end((err, res) => {
      should.not.exist(err);
      res.redirects.length.should.eql(0);
      res.status.should.eql(200);
      res.type.should.eql('text/html');
      res.text.should.contain('<h1>Register</h1>');
      res.text.should.contain(
```

Run the tests. You should see the following error:

```
Uncaught AssertionError: expected [Error: Not Found] to not exist
```

Now let's write the code to get it to pass…

Code

First, add a new file to the "src/server/routes" folder called *auth.js*:

```
const Router = require('koa-router');
const passport = require('koa-passport');
const fs = require('fs');
const queries = require('../db/queries/users');

const router = new Router();

router.get('/auth/register', async (ctx) => {
  ctx.type = 'html';
  ctx.body = fs.createReadStream('./src/server/views/register.html');
});

module.exports = router;
```

Add another new file called *users.js* to "src/server/db/queries". Leave the file empty for now. Create the "views" folder, and then add the *register.html* template:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Register</title>
</head>
<body>
  <h1>Register</h1>
  <form action="/auth/register" method="post">
    <p><label>Username: <input type="text" name="username"/></label></p>
    <p><label>Password: <input type="password" name="password"/></label></p>
    <p><button type="submit">Register</button></p>
  </form>
</body>
</html>
```

Register the auth routes in *src/server/index.js*:

```
const indexRoutes = require('./routes/index');
const movieRoutes = require('./routes/movies');
const authRoutes = require('./routes/auth');

const app = new Koa();
const PORT = process.env.PORT || 1337;

// sessions
app.keys = ['super-secret-key'];
app.use(session(app));

// body parser
app.use(bodyParser());

// authentication
require('./auth');
app.use(passport.initialize());
app.use(passport.session());

// routes
app.use(indexRoutes.routes());
app.use(movieRoutes.routes());
app.use(authRoutes.routes());

// server
const server = app.listen(PORT, () => {
  console.log(`Server listening on port: ${PORT}`);
});

module.exports = server;
```

Ensure the tests now pass:

```
$ npm test

Server listening on port: 1337
  routes : auth
    GET /auth/register
      ✓ should render the register view

  routes : index
    GET /
      ✓ should return json

  routes : movies
    GET /api/v1/movies
      ✓ should return all movies
    GET /api/v1/movies/:id
    PUT /api/v1/movies
      ✓ should return the movie that was updated
      ✓ should throw an error if the movie does not exist
    DELETE /api/v1/movies/:id
      ✓ should return the movie that was deleted
      ✓ should throw an error if the movie does not exist

  Sample Test
    ✓ should pass


  12 passing (868ms)
```

Register - POST

Test

Again, start with a test:

```
describe('POST /auth/register', () => {
  it('should register a new user', (done) => {
    chai.request(server)
    .post('/auth/register')
    .send({
      username: 'michael',
      password: 'herman'
    })
    .end((err, res) => {
      should.not.exist(err);
      res.redirects[0].should.contain('/auth/status');
      done();
    });
  });
});
```

The test should fail with the following error, since the route does not exist:

```
Uncaught AssertionError: expected [Error: Not Found] to not exist
```

Code

Add the route handler:

```
router.post('/auth/register', async (ctx) => {
  const user = await queries.addUser(ctx.request.body);
  return passport.authenticate('local', (err, user, info, status) => {
        ctx.body = { status: 'error' };
      }
    })(ctx);
  });
```

Here, if the user is successfully added to the database, we call the `login` method, from koa-passport, to trigger the creation of the session and then redirect the user to `/auth/status`

For the `addUser()` helper, add the following code to *src/server/db/queries/users.js*:

```
const knex = require('../connection');

function addUser(user) {
  return knex('users')
  .insert({
    username: user.username,
    password: user.password
  })
  .returning('*');
}

module.exports = {
  addUser,
};
```

Ensure the tests pass.

Status

Add the route handler:

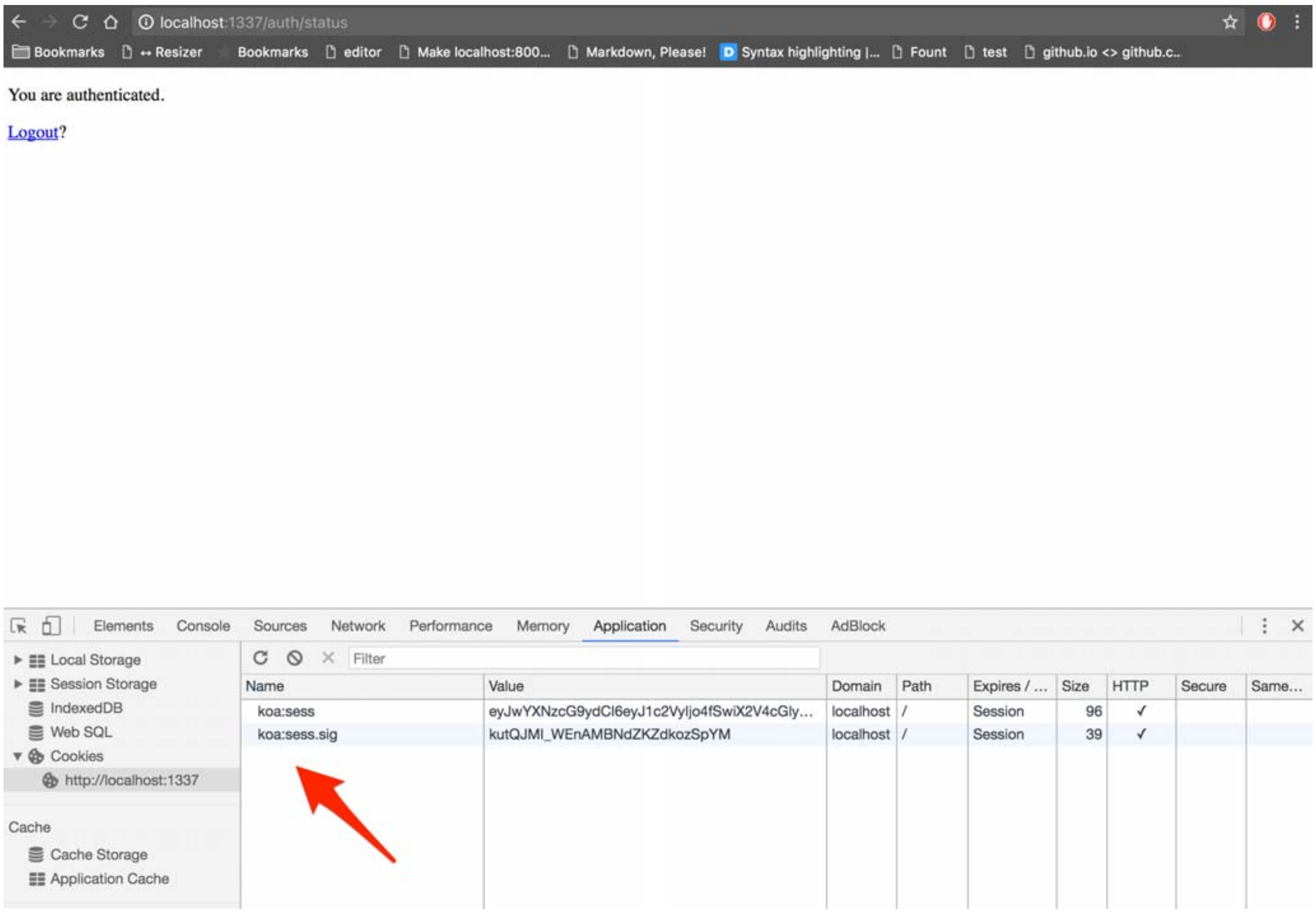
```
router.get('/auth/status', async (ctx) => {
  if (ctx.isAuthenticated()) {
    ctx.type = 'html';
    ctx.body = fs.createReadStream('./src/server/views/status.html');
  } else {
    ctx.redirect('/auth/login');
  }
});
```

For this route, we'll skip the tests since we'll have to stub the `isAuthenticated()` method and manually set a cookie.

Add the template:

```
</head>
<body>
  <p>You are authenticated.</p>
  <p><a href="/auth/logout">Logout</a>?</p>
</body>
</html>
```

To test, fire up the server via `npm start` and navigate to http://localhost:1337/auth/status. Register a new user. You should be redirected to `auth/status` and a cookie should be set:



You are authenticated.

Logout?

Name	Value	Domain	Path	Expires / ...	Size	HTTP	Secure	Same...
koa:sess	eyJwYXNzcG9ydCl6eyJ1c2Vyljo4fSwiX2V4cGly...	localhost	/	Session	96	✓		
koa:sess.sig	kutQJMI_WEnAMBNdZKZdkozSpYM	localhost	/	Session	39	✓		

Login - GET

For this route, we'll serve up a view with an HTML form for users to log in with.

Test

```
describe('GET /auth/login', () => {
  it('should render the login view', (done) => {
    chai.request(server)
    .get('/auth/login')
    .end((err, res) => {
      should.not.exist(err);
```

```
        '<p><button type="submit">Log In</button></p>');
      done();
    });
  });
});
```

Code

Add the route handler:

```
router.get('/auth/login', async (ctx) => {
  if (!ctx.isAuthenticated()) {
    ctx.type = 'html';
    ctx.body = fs.createReadStream('./src/server/views/login.html');
  } else {
    ctx.redirect('/auth/status');
  }
});
```

Then, add the *login.html* template:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Login</title>
</head>
<body>
  <h1>Login</h1>
  <form action="/auth/login" method="post">
    <p><label>Username: <input type="text" name="username"/></label></p>
    <p><label>Password: <input type="password" name="password"/></label></p>
    <p><button type="submit">Log In</button></p>
  </form>
</body>
</html>
```

The tests should now pass.

Login - POST

Test

```
describe('POST /auth/login', () => {
  it('should login a user', (done) => {
    chai.request(server)
    .end((err, res) => {
      res.redirects[0].should.contain('/auth/status');
      done();
    });
  });
});
```

Code

```
router.post('/auth/login', async (ctx) => {
  return passport.authenticate('local', (err, user, info, status) => {
    if (user) {
      ctx.login(user);
      ctx.redirect('/auth/status');
    } else {
      ctx.status = 400;
      ctx.body = { status: 'error' };
    }
  })(ctx);
});
```

Let's also create a new Knex seed file to add a test user to the database:

```
$ knex seed:make users
```

Add the following code to the newly created seed in "src/server/db/seeds":

```
exports.seed = (knex, Promise) => {
  return knex('users').del()
  .then(() => {
    return Promise.join(
      knex('users').insert({
        username: 'jeremy',
        password: 'johnson'
      })
    );
  });
};
```

The tests should now pass. Before moving on, try adding a few more tests to handle errors as well.

Logout

```
    ctx.logout();
    ctx.redirect('/auth/login');
  } else {
    ctx.body = { success: false };
    ctx.throw(401);
  }
});
```

Manually test by registering a new user. If all is well, you should be redirected to `auth/status` and a cookie should be set. Then ensure that the cookie is removed after you log out.

Make sure all tests pass before moving on.

Password Hashing

Install bcrypt.js to handle the salting and hashing of passwords:

```
$ npm install bcryptjs@2.4.3 --save
```

Start by adding a helper method called `comparePassword` to *src/server/auth.js*:

```
function comparePass(userPassword, databasePassword) {
  return bcrypt.compareSync(userPassword, databasePassword);
}
```

Add the import as well:

```
const bcrypt = require('bcryptjs');
```

This helper can now be used when we pull a user from the database and check that the passwords are equal:

```
passport.use(new LocalStrategy(options, (username, password, done) => {
  knex('users').where({ username }).first()
  .then((user) => {
    if (!user) return done(null, false);
    if (!comparePass(password, user.password)) {
      return done(null, false);
    } else {
      return done(null, user);
    }
  })
```

```
const bcrypt = require('bcryptjs');
const knex = require('../connection');

function addUser(user) {
  const salt = bcrypt.genSaltSync();
  const hash = bcrypt.hashSync(user.password, salt);
  return knex('users')
  .insert({
    username: user.username,
    password: hash,
  })
  .returning('*');
}

module.exports = {
  addUser,
};
```

Do the same for the user seed in *src/server/db/seeds/users.js*:

```
const bcrypt = require('bcryptjs');

exports.seed = (knex, Promise) => {
  const salt = bcrypt.genSaltSync();
  const hash = bcrypt.hashSync('johnson', salt);
  return knex('users').del()
  .then(() => {
    return Promise.join(
      knex('users').insert({
        username: 'jeremy',
        password: hash,
      })
    );
  });
};
```

Now, instead of adding a plain text password to the database, we salt and hash it first.

Drop and recreate the `koa_api` database, apply the migrations, and then run the server and manually test everything out.

Finally, make sure the tests still pass:

```
$ npm test

Server listening on port: 1337
  routes : auth
```

```
      ✓ should render the login view
    POST /auth/login
      ✓ should login a user (99ms)

  routes : index
    GET /
      ✓ should return json

  routes : movies
    GET /api/v1/movies
      ✓ should return all movies
    GET /api/v1/movies/:id
      ✓ should respond with a single movie
      ✓ should throw an error if the movie does not exist
    POST /api/v1/movies
      ✓ should return the movie that was added
      ✓ should throw an error if the payload is malformed
    PUT /api/v1/movies
      ✓ should return the movie that was updated
      ✓ should throw an error if the movie does not exist
    DELETE /api/v1/movies/:id
      ✓ should return the movie that was deleted
      ✓ should throw an error if the movie does not exist

  Sample Test
    ✓ should pass


  15 passing (3s)
```

Redis Session Store

It's a good idea to move session data out of memory and into an external session store as you begin scaling your application.

For example, if you scale horizontally and start spinning up new instances of the same Node application to share the load, then users would need to log in to each instance separately if sessions are stored in memory. On the other hand, if sessions are stored in an external session store (like Redis), session data can be shared across all instances of the app. In the latter case, users would need to log in just once.

To utilize Redis as the session store, first install koa-redis:

```
$ npm install koa-redis@3.1.1 --save
```

Then, update the `koa-session` middleware config in *src/server/index.js*:

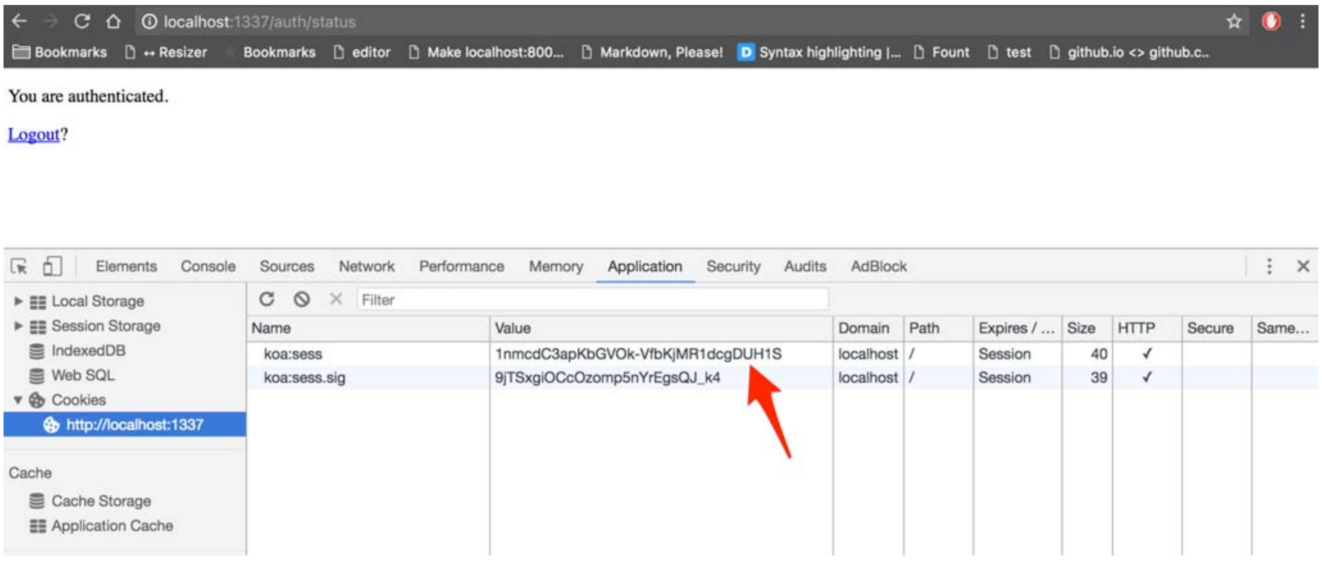
Add the dependency:

```
const RedisStore = require('koa-redis');
```

Take note of the default options for koa-redis, making any necessary changes. Then, download and install Redis (if necessary) and spin up the server in a new terminal tab:

```
$ redis-server
```

Fire up the app and register a new user, taking note of the cookie:



You are authenticated.

Logout?

Name	Value	Domain	Path	Expires / ...	Size	HTTP	Secure	Same...
koa:sess	1nmcdC3apKbGVOk-VfbKjMR1dcgDUH1S	localhost	/	Session	40	✓		
koa:sess.sig	9jTSxgiOCcOzomp5nYrEgsQJ_k4	localhost	/	Session	39	✓		

Within another new terminal tab, open the Redis client and make sure that key can be found:

```
$ redis-cli

127.0.0.1:6379> keys 1nmcdC3apKbGVOk-VfbKjMR1dcgDUH1S
1) "1nmcdC3apKbGVOk-VfbKjMR1dcgDUH1S"
127.0.0.1:6379> exit
```

Run the tests one final time.

Conclusion

In this tutorial, we went through the process of adding authentication to a Koa app with Passport. Turn back to the objectives. Review each one. What did you learn?

The full code can be found in the v3 tag of the node-koa-api repository.

Please share your comments, questions, and/or tips in the comments below.

14

24 Comments Michael Herman 1 Login

Recommend 1 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Rodrigo Leite • 2 years ago

I suggest updating the article to not using bcrypt's synchronous methods

5 ^ | ⌄ • Reply • Share ›

Mederic Burlet • a month ago

for people using typescript the password.authenticate will not work you need:

```
router.post('/login', async (ctx, next) => {
await Passport.authenticate('local', (err, user, info, status) => {
// do stuff

})(ctx, next);
```

^ | ⌄ • Reply • Share ›

coco tao • 2 years ago

Thanks for your great tutorial! I tried passport-jwt or passport-local in koa2, I found it couldn't access to strategy authenticate() method....

^ | ⌄ • Reply • Share ›

@dgoore • 2 years ago

slidenerd → @dgoore • a year ago • edited

Google the post "Stop Using JWT for sessions"

1 ^ | ⌄ • Reply • Share ›

Ben Armstrong • 2 years ago

Hi I'm aiming for 100% code coverage with my tests but I'm struggling to test the passport.serialize/deserialize and local strategy.

I have stubbed authentication for my routes tests but the only way I see to test this is to unstub the passport.authenticate for some tests.

I was wondering if you had any ideas?

^ | ⌄ • Reply • Share ›

michaelherman Mod → Ben Armstrong • 2 years ago

Can you not create a new test file and just not use the stub for those particular set of tests?

1 ^ | ⌄ • Reply • Share ›

Ben Armstrong → michaelherman • 2 years ago

Hi, that's what I have currently done but I am struggling to test the deserialize part of the configuration, do you know how I could accomplish this?

2 ^ | ⌄ • Reply • Share ›

michaelherman Mod → Ben Armstrong • 2 years ago

I wouldn't worry about testing the actual functionality of deserialize since it's being tested already in the external library.

1 ^ | ⌄ • Reply • Share ›

ryqaz → Ben Armstrong • 2 years ago

disqus_idl23JRXMY briefly

^ | ⌄ • Reply • Share ›

Sir Robert Burbridge • 2 years ago • edited

I wanted to work through the completed example to get a holistic understanding. When I check out (master or v4) and run the knex seed and migrations, the server runs (e.g. the hello world, and GET /auth/...).

[illegible]

Sir Robert Burbridge ➦ Sir Robert Burbridge • 2 years ago

I figured out the problem. I had set up redis but hadn't started the server. That meant the sessioning was waiting indefniitely for the redis server to respond.

1 ^ | ˅ • Reply • Share ›

André Cruz • 2 years ago

Thank you for the article, it helped a lot.

I have a question though. In the '/auth/register' route, why do we need to run passport.authenticate() after having just created the new user? Can't we just call 'ctx.login(user)' immediately? It seems weird to verify the credentials of the user we have just created.

^ | ˅ • Reply • Share ›

michaelherman Mod ➦ André Cruz • 2 years ago

I think you're right - you can probably just call 'ctx.login(user)'.

1 ^ | ˅ • Reply • Share ›

whateverrrr • 2 years ago • edited

Great tutorial. Just gonna point out some minor issues which might confuse some people:

When adding the Register GET, we set a requirement in the auth.js route:
const queries = require('../db/queries/users');

Which is not actually added until the next point in the tutorial. Just create an empty file until then.

For Register POST, the error message:
Uncaught AssertionError: expected [Error: Not Found] to not exist
Is inaccurate, as this is returned by
should.not.exist(err)
which is not present in that test. Add it or ignore it, it doesn't really matter.

^ | ˅ • Reply • Share ›

michaelherman Mod ➦ whateverrrr • 2 years ago

Thanks for the feedback. I updated the tutorial. Best!

^ | ˅ • Reply • Share ›

up, and configured the db according to it. When I send POST to /auth/register it inserts to db, but then no redirection happens. In the router.post function " const user = await queries.addUser(ctx.request.body);" this returns the inserted entry successfully. However the "return passport.authenticate('local' ..... " doesn't enter the if(user) statement. I'm quite new to koajs and passportjs and actually new JS syntax. So I couldn't figure out what might be the problem. I've downloaded the v3 compared with my code and didn't catch any differences. What might be the problem? Thanks again.

• Reply • Share ›

michaelherman Mod ➜ Barış Güvercin
• 3 years ago • edited

You can debug by adding a few console.log statements above the if statement:

```
console.log(err)
console.log(user)
```

• Reply • Share ›

Barış Güvercin ➜ michaelherman • 3 years ago

I've added these, just above if statement
console.log(err);
console.log(user);
console.log(ctx.request.body);
then got,
null , false , { email: 'test2', password: 'test2' }
maybe passport couldn't parse the request internally

• Reply • Share ›

Barış Güvercin ➜ Barış Güvercin
• 3 years ago • edited

Finally it worked out, I just added some options to LocalStrategy
const options = {
usernameField: 'email',
passwordField: 'password' };
According to passport-local
"By default, LocalStrategy expects to find

Questions? michael at mherman dot org

Back to top