Authentication for Web APIs

using JSON Web Tokens and Passport

Frank Walsh, 2021

Agenda This Week

- Schema Methods and Sessions
- JSON Web Tokens (JWT)
- Authentication
  - Salting with BCrypt
- Passport
- Mongoose Middleware(hooks)
- Use Case – Login/Register for React App using JWT/Passport

Schema Methods

Example: Using Schema Methods for Simple Authentication

- Restrict access to API (require authentication):
  - Create users schema with methods for
    - Finding users
    - Checking password
  - Use **express-session** middleware to create and manage user session (using cookies)
  - Create an authentication route to set up “session”
  - Create your own authentication middleware and place it on /api/movies route

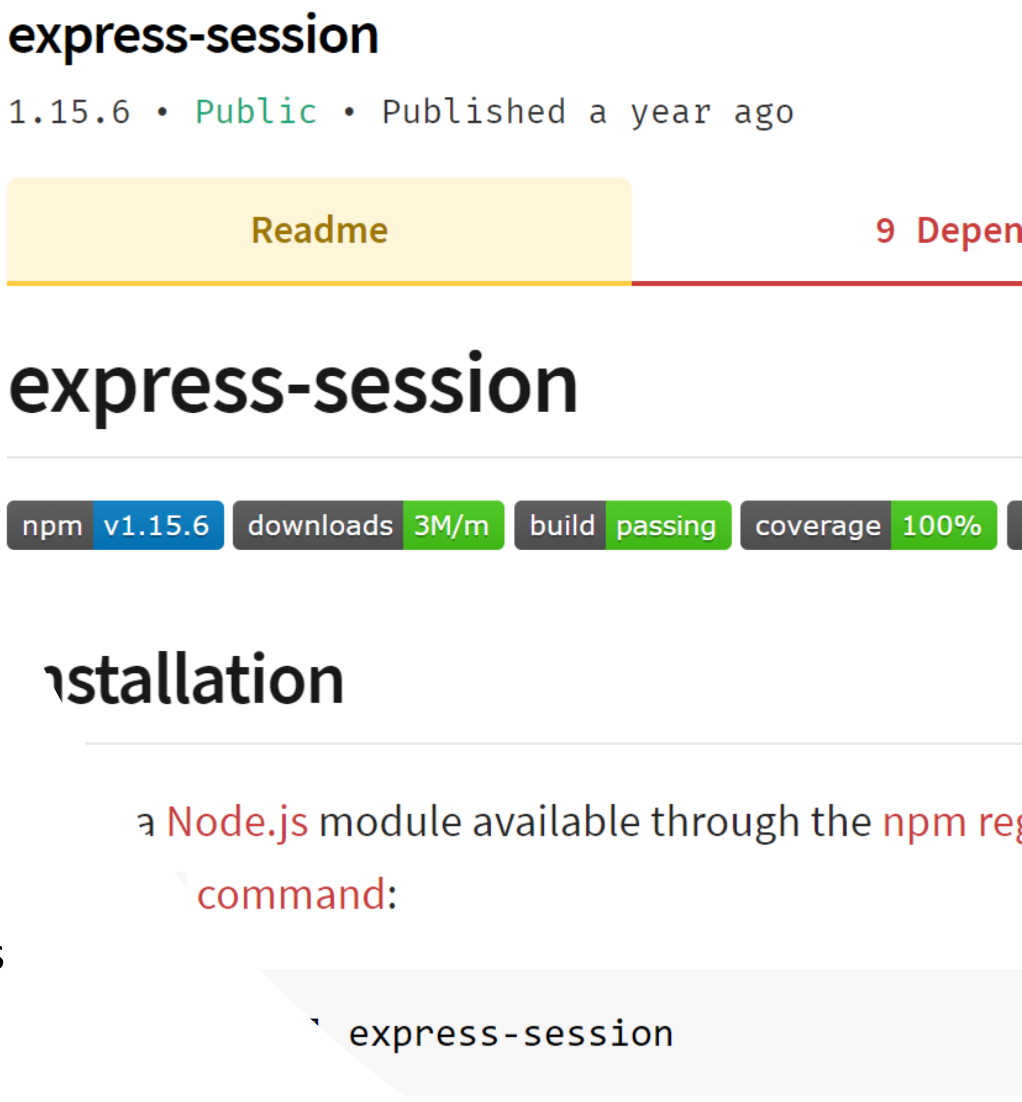
Aside: Sessions

- Requests to Express apps are stand-alone by default
  - no request can be linked to another.
  - By default, no way to know if this request comes from a client that already performed a request previously.

- Sessions are a mechanism that makes it possible to “know” who sent the request and to associate requests.

- Using Sessions, every user of you API is assigned a unique session:
  - Allows you to store state.

- The express-session module is middleware that provides sessions for Express apps.


express-session

1.15.6 • Public • Published a year ago

Readme 9 Depen

express-session

npm v1.15.6 downloads 3M/m build passing coverage 100%

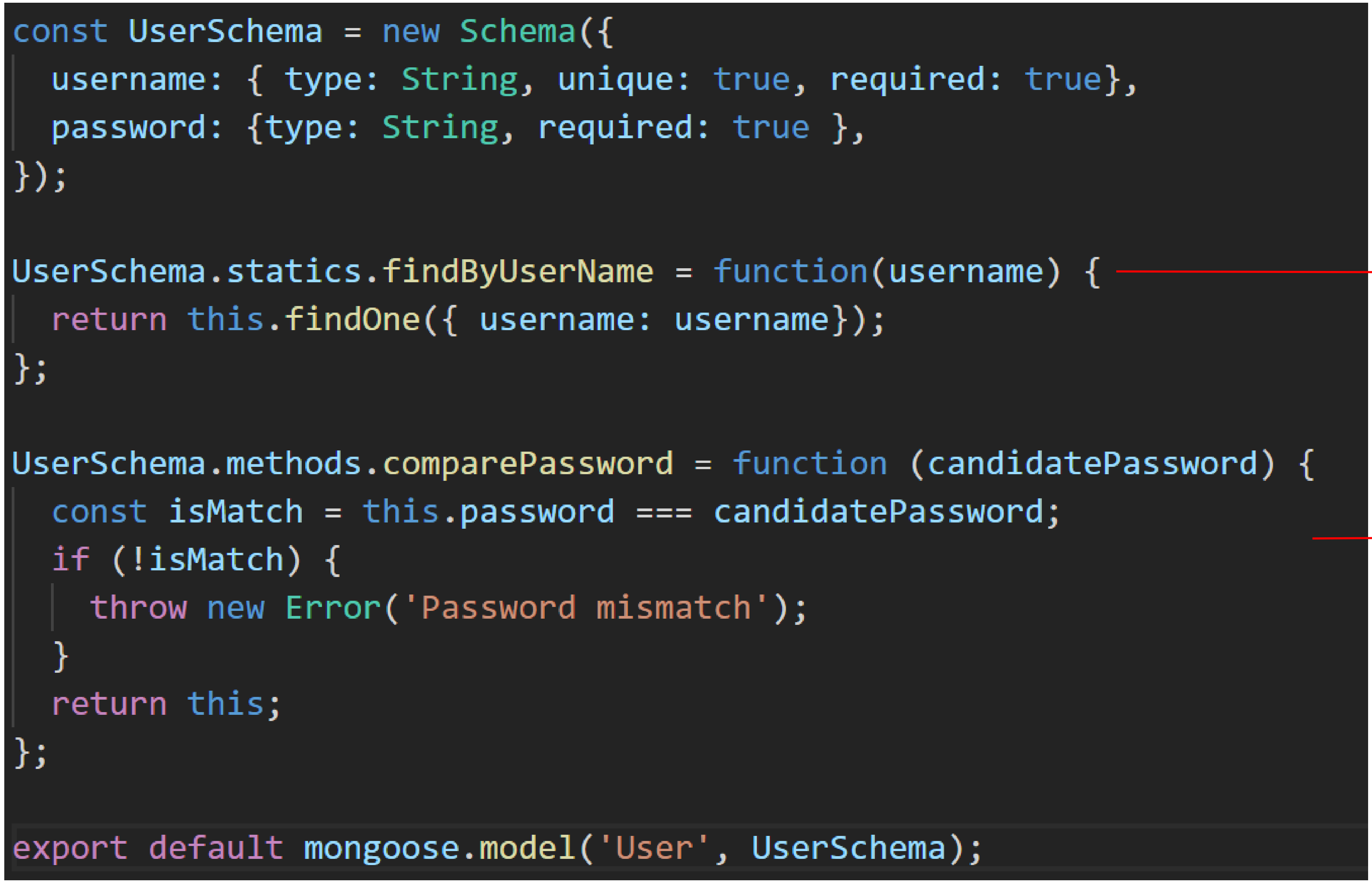
ıstallation

a Node.js module available through the npm reg

command:

express-session

1. User Schema with Static & Instance Methods



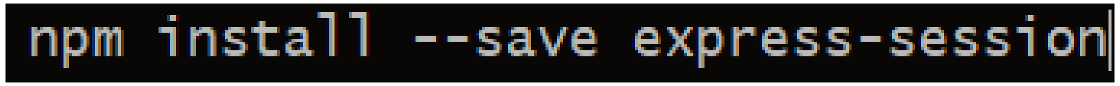
```
const UserSchema = new Schema({
  username: { type: String, unique: true, required: true},
  password: {type: String, required: true },
});

UserSchema.statics.findByUserName = function(username) {
  return this.findOne({ username: username});
};

UserSchema.methods.comparePassword = function (candidatePassword) {
  const isMatch = this.password === candidatePassword;
  if (!isMatch) {
    throw new Error('Password mismatch');
  }
  return this;
};

export default mongoose.model('User', UserSchema);
```

Static Method: belongs to schema. Independent of any document instance

Instance Method: belongs to a specific document instance.

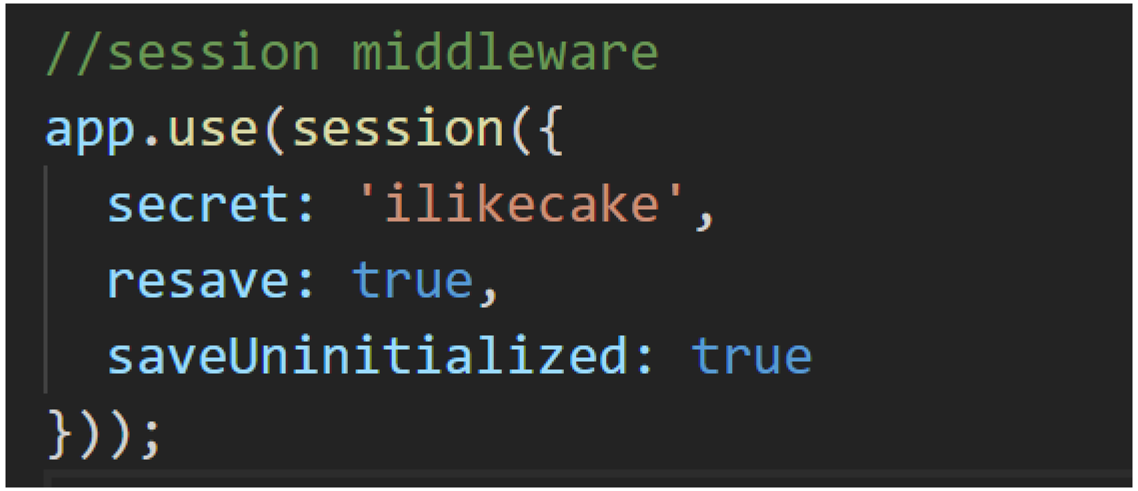
2. express-session middleware

- Session middleware that stores session data on server-side
  - Puts a unique ID on client



```
npm install --save express-session
```

- Add to Express App middleware stack:

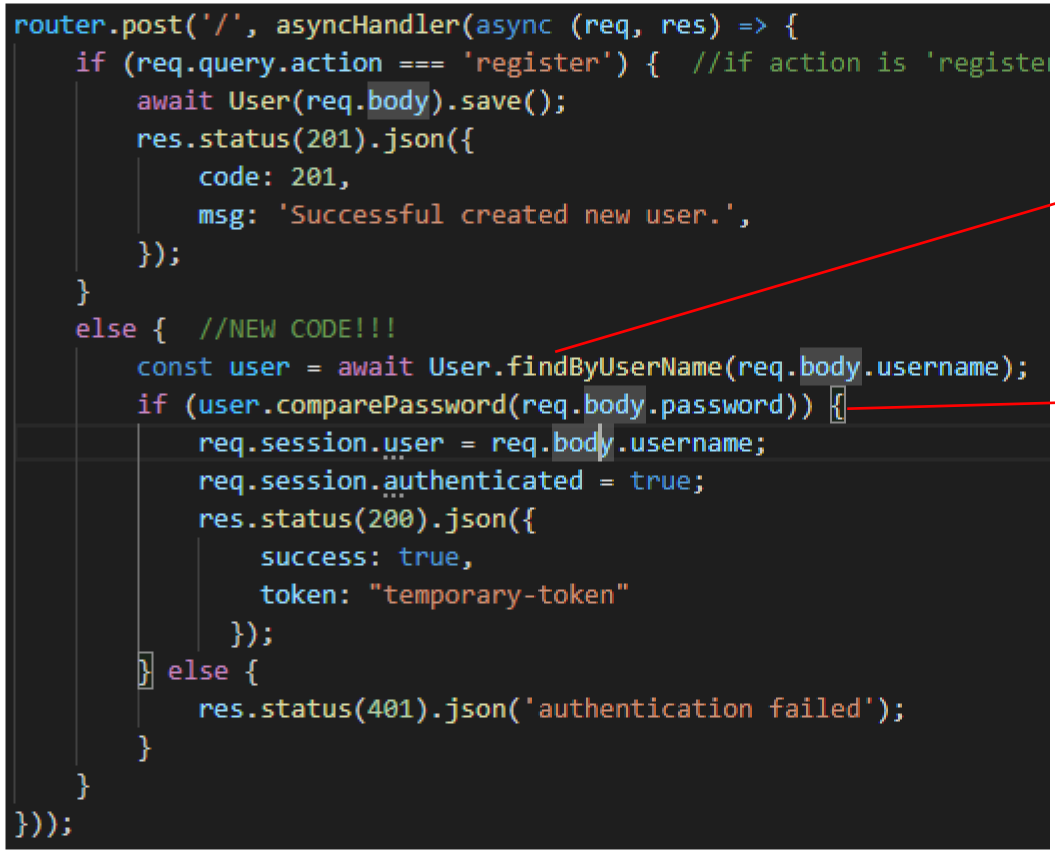


```
//session middleware
app.use(session({
  secret: 'ilikecake',
  resave: true,
  saveUninitialized: true
}));
```

3. Use User Route to authenticate

- Use **/api/user** to authenticate, passing username and password in HTTP body

/api/users/index.js

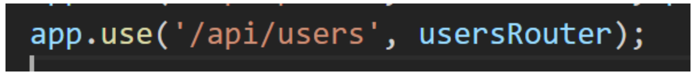


```
router.post('/', asyncHandler(async (req, res) => {
    if (req.query.action === 'register') {  //if action is 'registe
        await User(req.body).save();
        res.status(201).json({
            code: 201,
            msg: 'Successful created new user.',
        });
    }
    else {  //NEW CODE!!!
        const user = await User.findByUserName(req.body.username);
        if (user.comparePassword(req.body.password)) {
            req.session.user = req.body.username;
            req.session.authenticated = true;
            res.status(200).json({
                success: true,
                token: "temporary-token"
              });
        } else {
            res.status(401).json('authentication failed');
        }
    }
}));
```

Using static method to find User document

Using instance method to check password

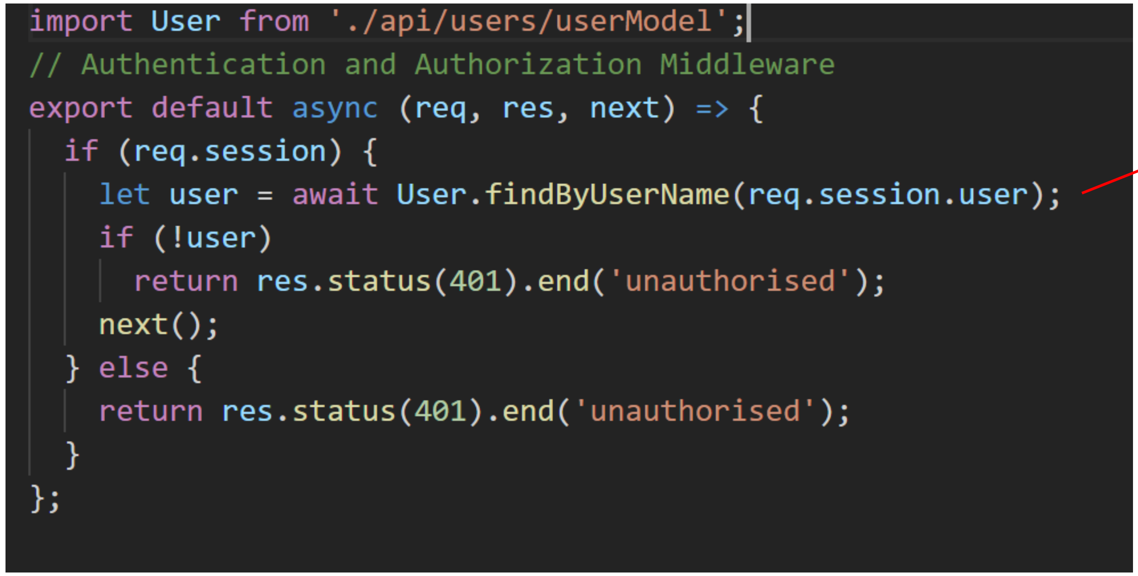
/index.js



```
app.use('/api/users', usersRouter);
```

4. Add Authentication Middleware

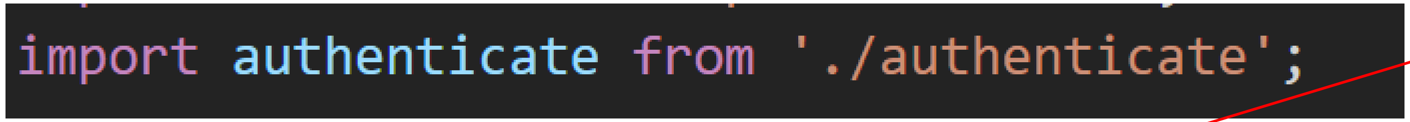
authenticate.js



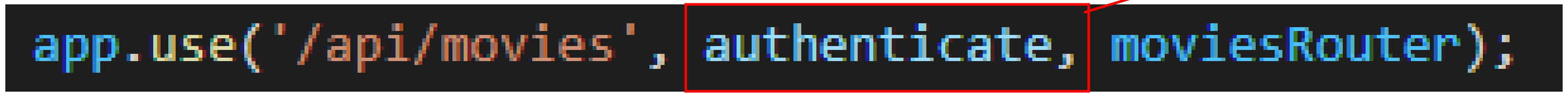
```
import User from './api/users/userModel';
// Authentication and Authorization Middleware
export default async (req, res, next) => {
  if (req.session) {
    let user = await User.findByUserName(req.session.user);
    if (!user)
      return res.status(401).end('unauthorised');
    next();
  } else {
    return res.status(401).end('unauthorised');
  }
};
```

Checks for user ID in session object.
If exists, called next middleware function, otherwise end req/res cycle with 401

index.js



```
import authenticate from './authenticate';
```



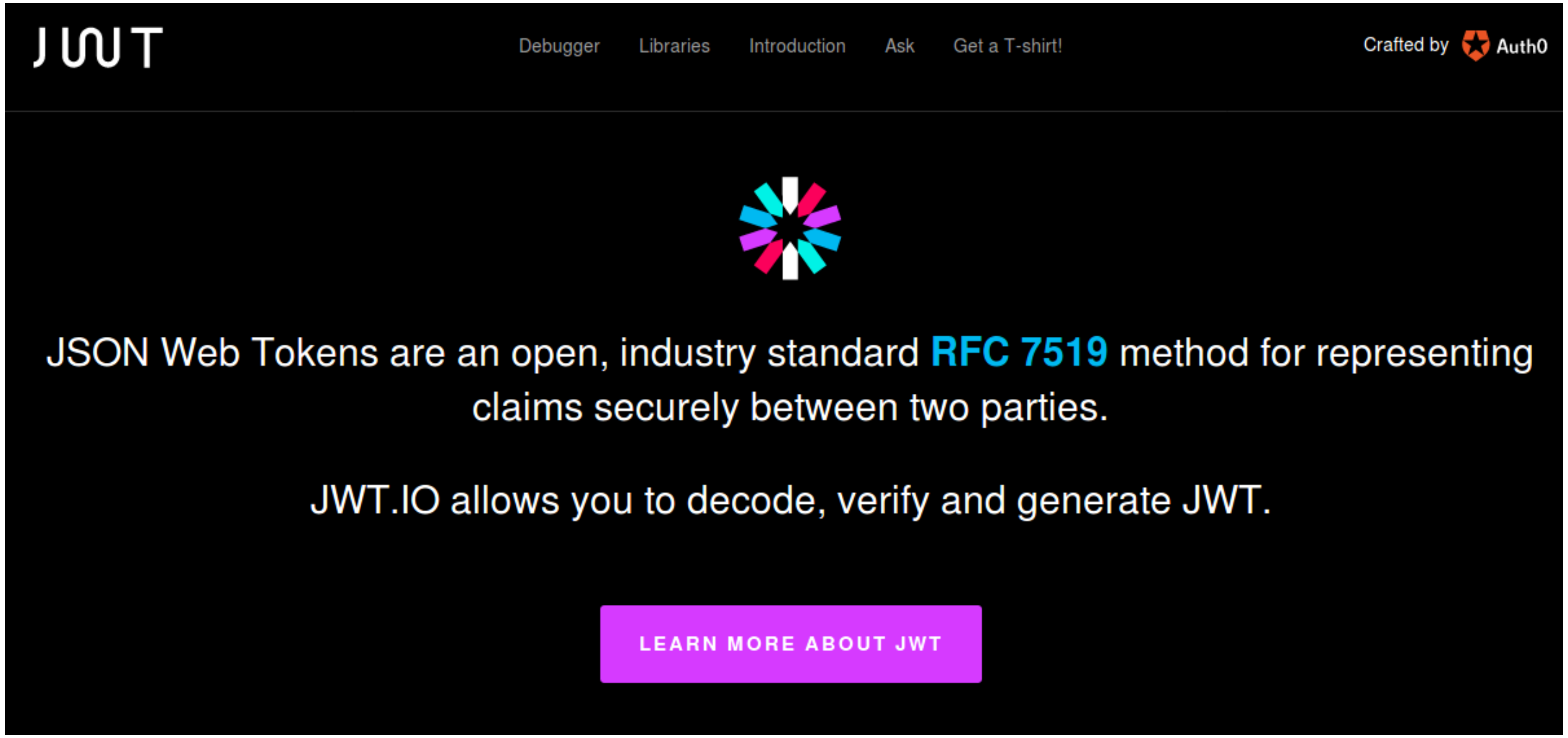
```
app.use('/api/movies', authenticate, moviesRouter);
```

Authentication middleware applied on /api/movies route.

Javascript Web Tokens

Authentication options

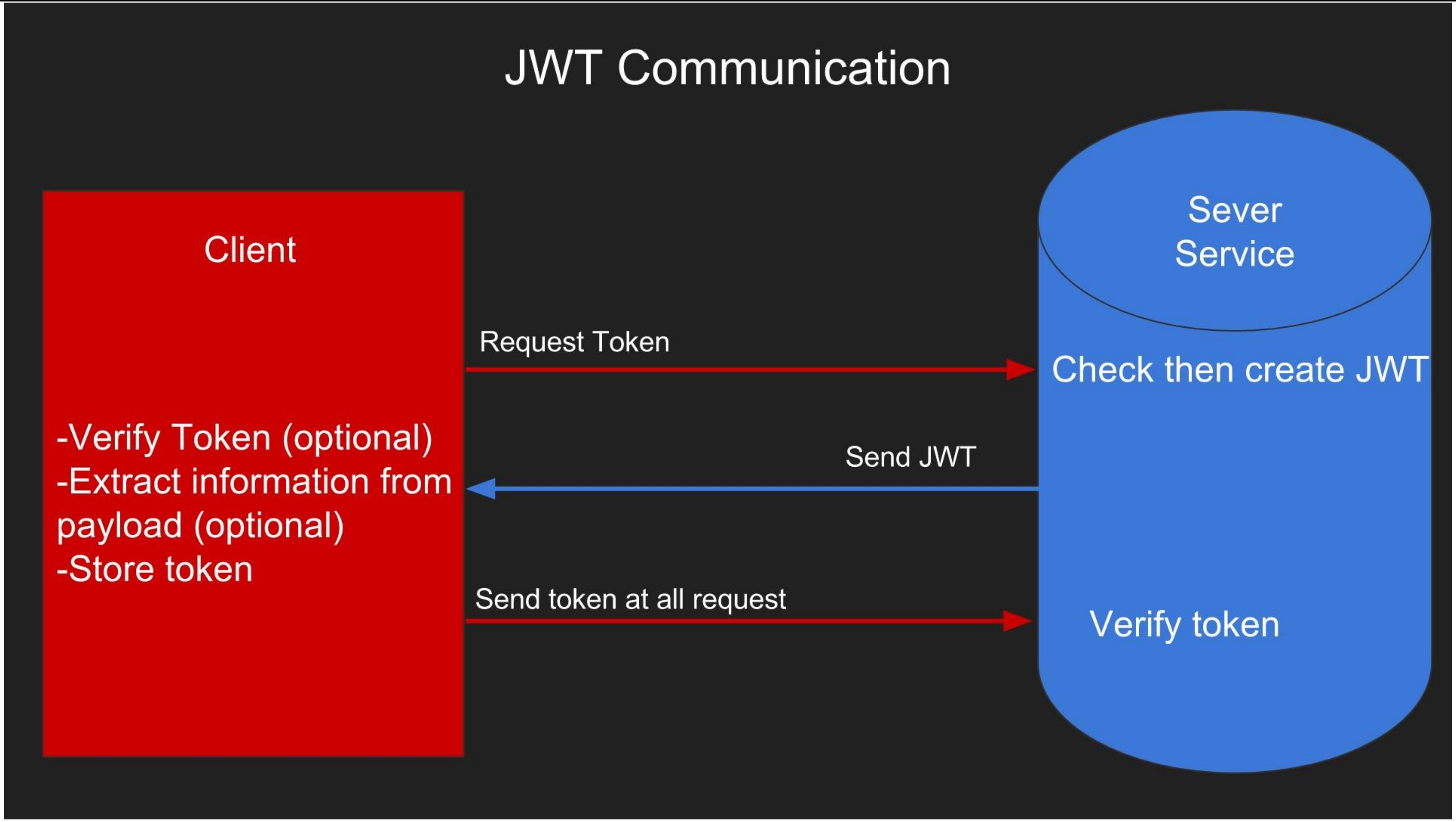
- Many solutions for Authentication
  - Cookies, basic-auth, JWT, OAuth.
  - Web-based Identity Federation/3rd Party (Firebase)
- JSON Web Tokens (JWT)
  - Tokens means no need to keep sessions or cookies
  - In keeping with REST stateless principle – token sent on each request
  - Token stored on client, usually in local storage of client.



JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties.

JWT.IO allows you to decode, verify and generate JWT.

JSON Web Tokens

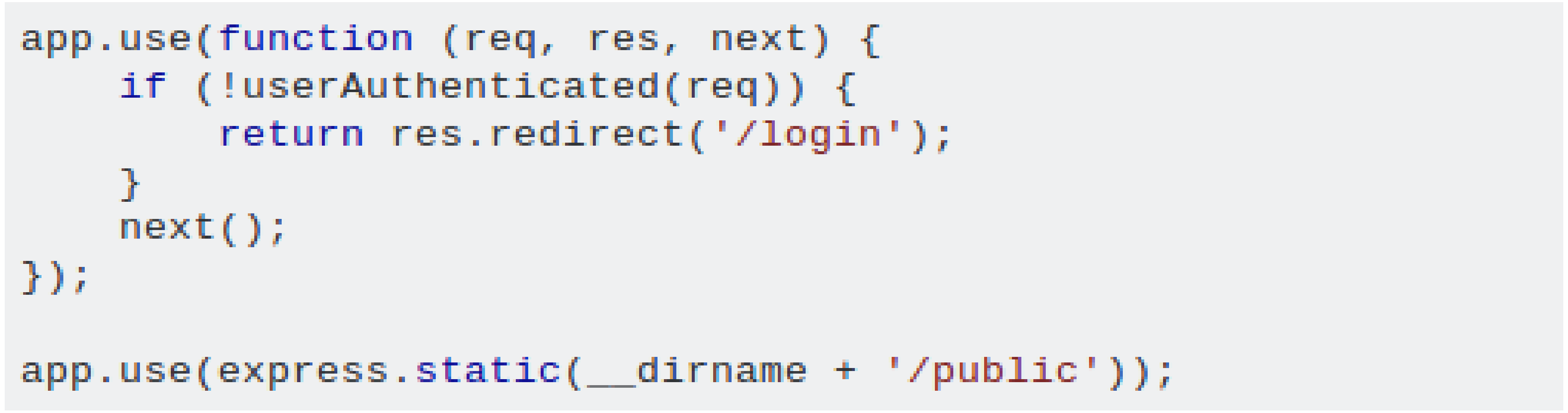


Username and Password Scenario

- Scenario
  - User signs up to access an API (username & password)
  - Create a new user in database
  - Use new username to create a JWT
  - Send JWT back to user
  - User stores JWT
  - JWT used on every subsequent request to protected resource
- Authentication and Identification
  - ...because username was used to generate JWT.

Authentication Middleware

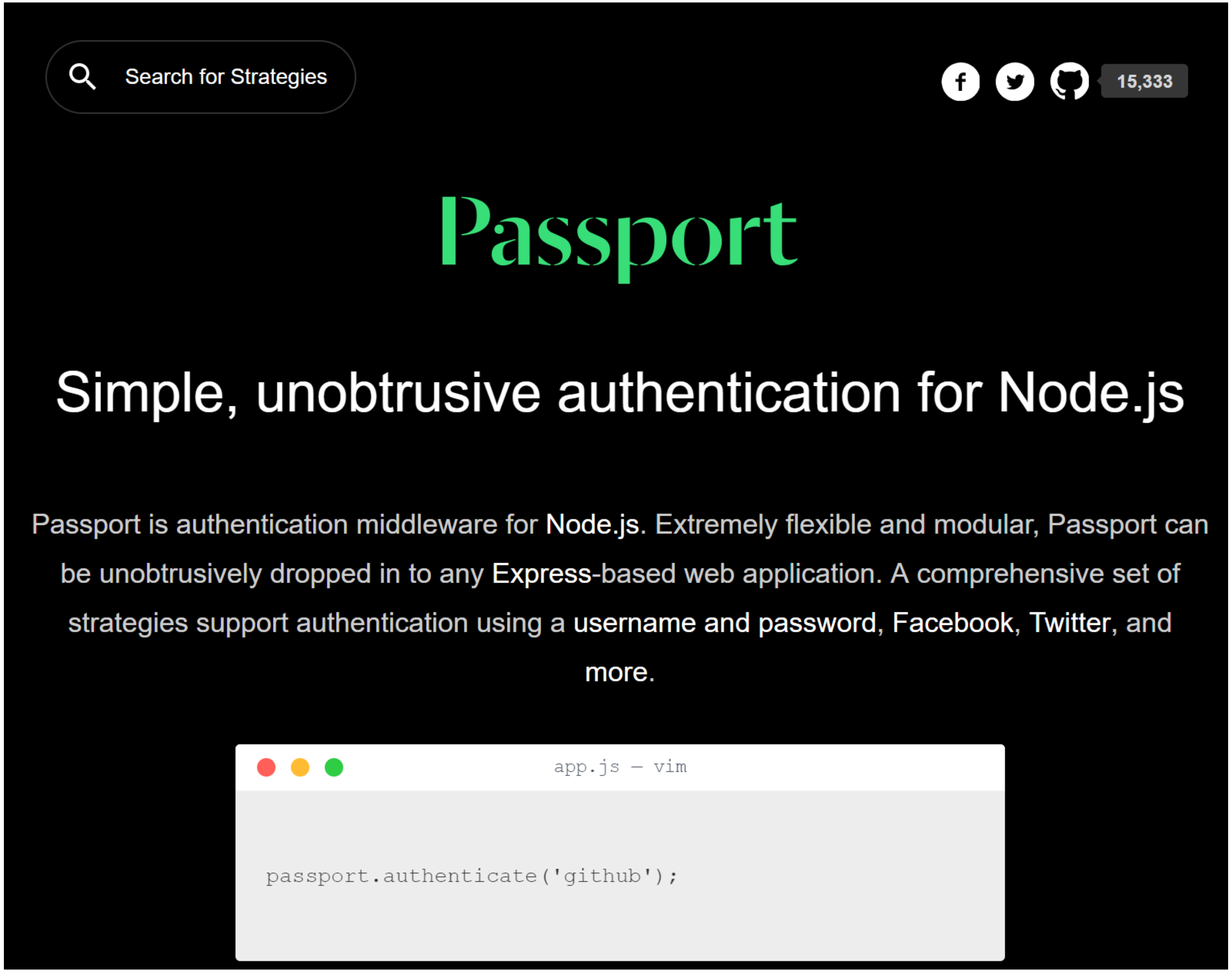
- Need express middleware to manage user login
- Need Express middleware to restrict access to sensitive routes.
- Options
  - Roll our own(previous express-sessions example…)
  - Use existing framework/package



```
app.use(function (req, res, next) {
    if (!userAuthenticated(req)) {
        return res.redirect('/login');
    }
    next();
});

app.use(express.static(__dirname + '/public'));
```

Passport

- Passport is authentication middleware
- Flexible and modular.
- Easy to retrospectively drop into an Express app.
- Lots of "strategies" for authentication
    - Username/Password
    - Facebook
    - Twitter


Search for Strategies

15,333

Passport

Simple, unobtrusive authentication for Node.js

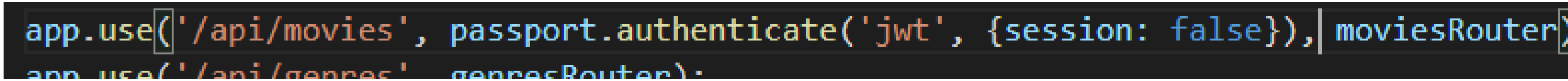
Passport is authentication middleware for Node.js. Extremely flexible and modular, Passport can be unobtrusively dropped in to any Express-based web application. A comprehensive set of strategies support authentication using a username and password, Facebook, Twitter, and more.

app.js — vim

```
passport.authenticate('github');
```

Passport Overview

- Passport offers different authentication mechanisms as **Strategies**
  - You install just the modules you require for a particular strategy
- Authenticate by calling passport.authenticate()
  - specify which strategy to use.
- The **authenticate()** function signature is a standard Express middleware function…
  - Just drop it in..



```
app.use('/api/movies', passport.authenticate('jwt', {session: false}), moviesRouter
app.use('/api/genres', genresRouter);
```

Requirements for Authentication: movie-api

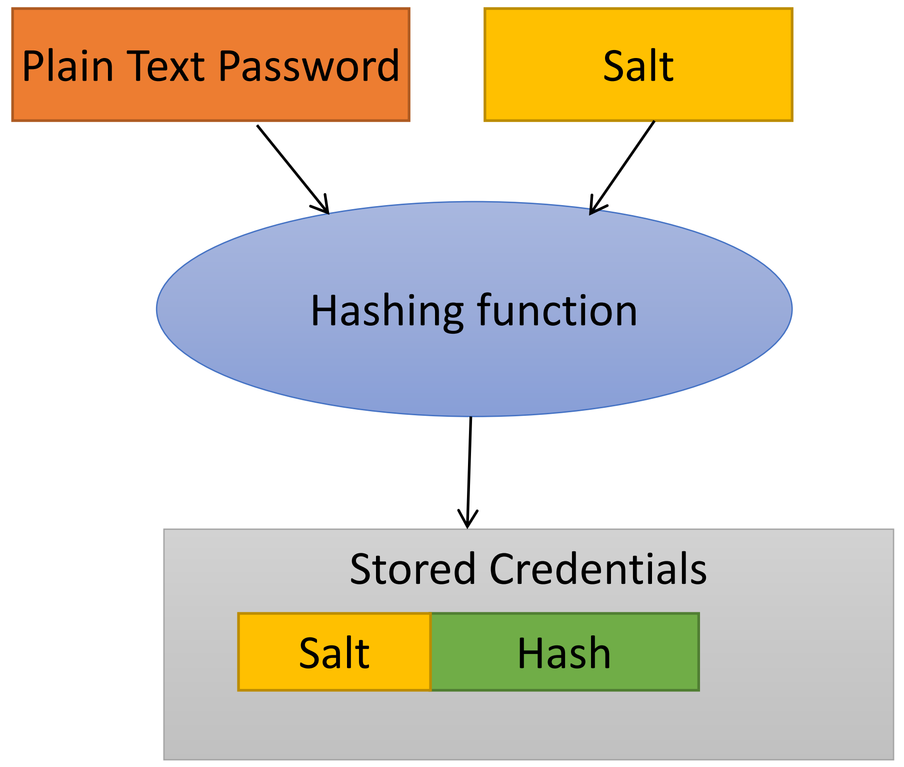
- Restrict access to authenticated users.
- Provide **User API** to login/register.
- Users should only have to log in once: Ideally identified and authenticated in subsequent requests.
- Username and Password authentication.
- No clear case passwords like last week!!! Hash/Salt all passwords in MongDB

Web authentication – credentials

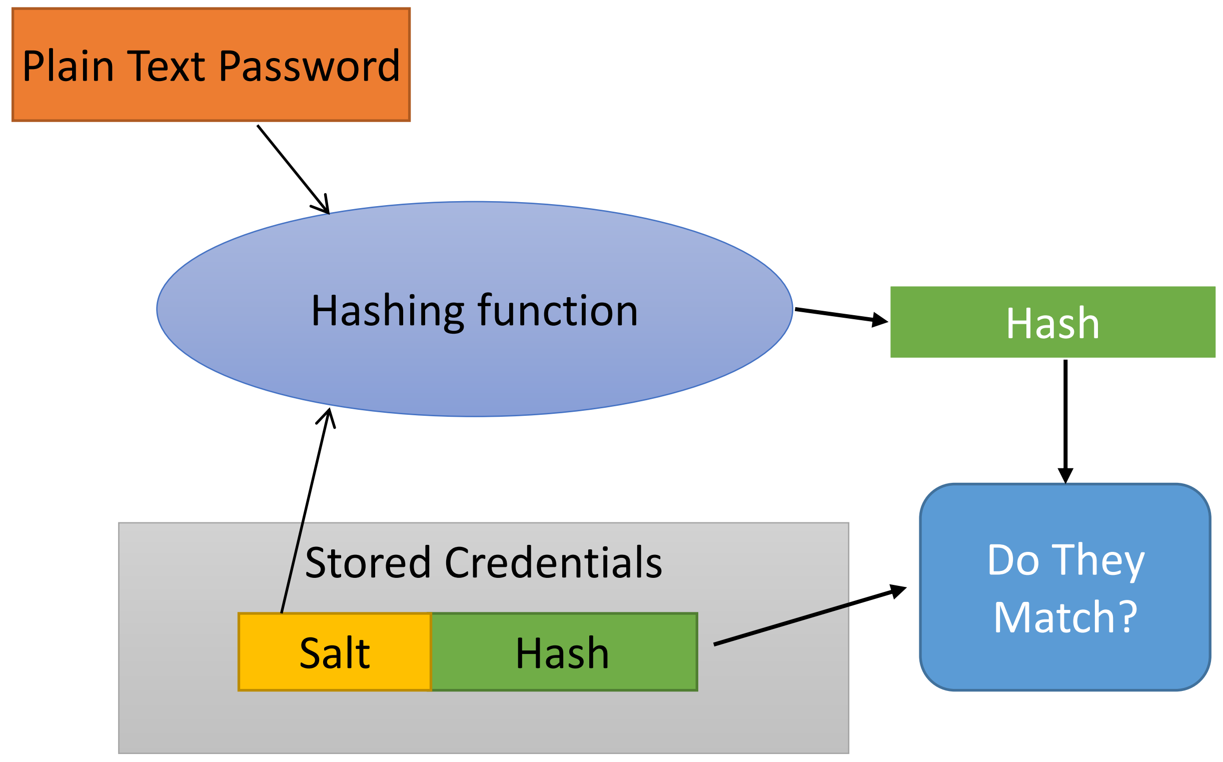
- Credentials should be stored securely in a centralised location
  - Should only be readable by suitably privileged users
  - Credentials should not find their way into hidden fields, headers, cookies
  - Should not be “hard coded”
- Passwords should be **“salted”** and **“hashed”**
  - Salting involves appending random bits to each password
  - Salted password is then hashed (i.e. one-way encrypted) for storage
- Objective is to store something derived from the password that allows an entered candidate password to be checked …
  - … but such that the password cannot be retrieved (by *anybody*, even an administrator)

Passwords & Salting

Password Creation



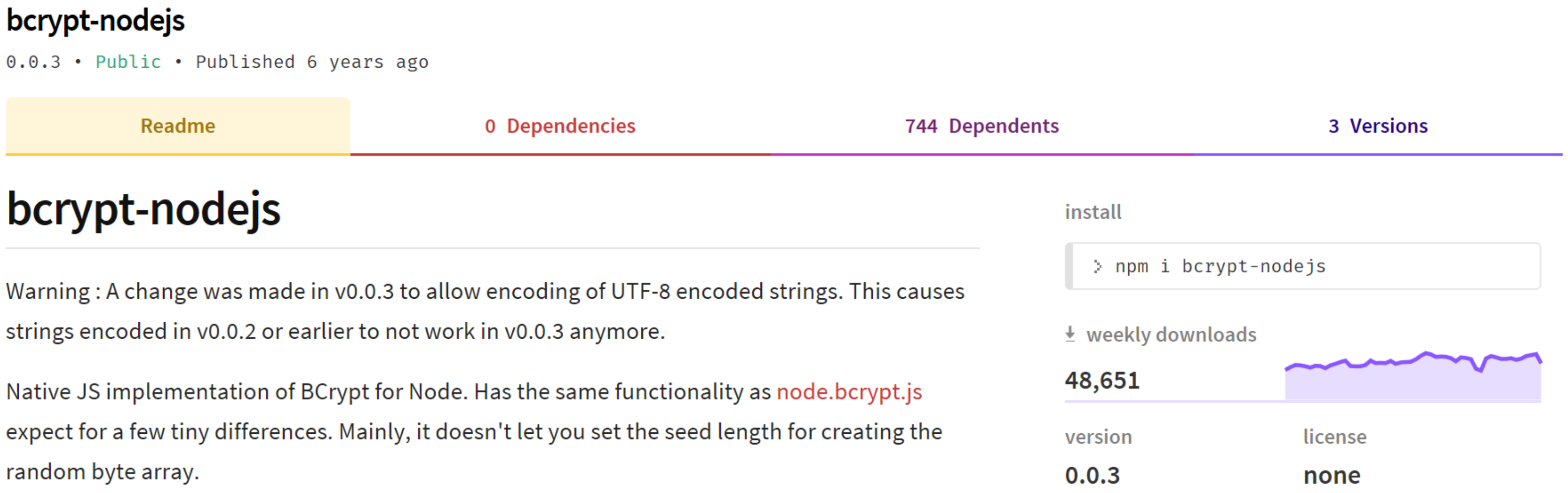
Password Verification



Why Salt?

- Frustrates dictionary attacks.
- Prevents duplicate passwords appearing as duplicates in password db (using different Salts)
- Protects users where same password is reused on different systems/sites.

Salting and Encrypting in Node.js/Express



bcrypt-nodejs

0.0.3 • Public • Published 6 years ago

Readme | 0 Dependencies | 744 Dependents | 3 Versions

bcrypt-nodejs

Warning : A change was made in v0.0.3 to allow encoding of UTF-8 encoded strings. This causes strings encoded in v0.0.2 or earlier to not work in v0.0.3 anymore.

Native JS implementation of BCrypt for Node. Has the same functionality as node.bcrypt.js expect for a few tiny differences. Mainly, it doesn't let you set the seed length for creating the random byte array.

install

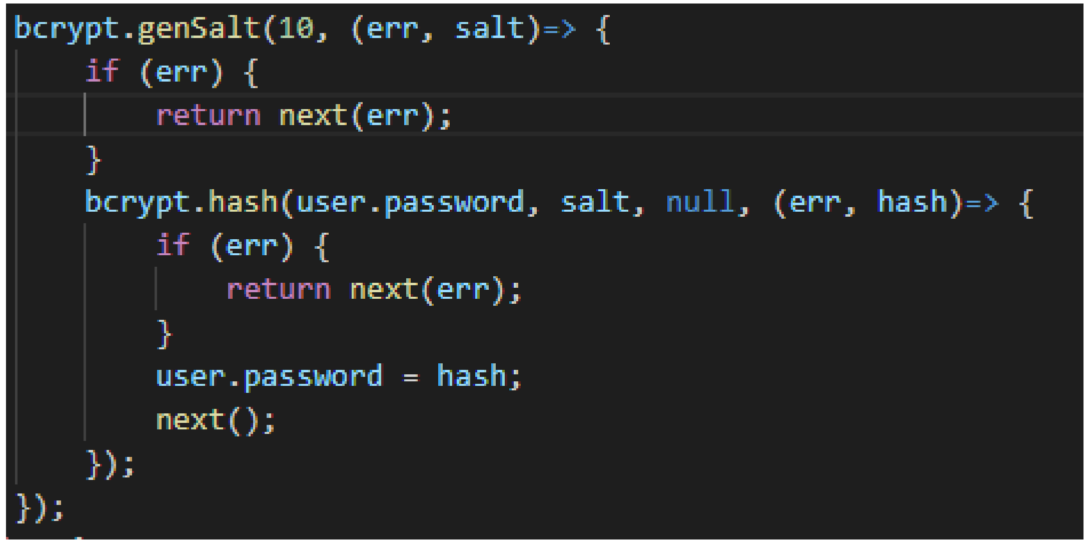
```
> npm i bcrypt-nodejs
```

weekly downloads

48,651

version	license
0.0.3	none

- Several NPM packages available.
- Also in other languages (Java)

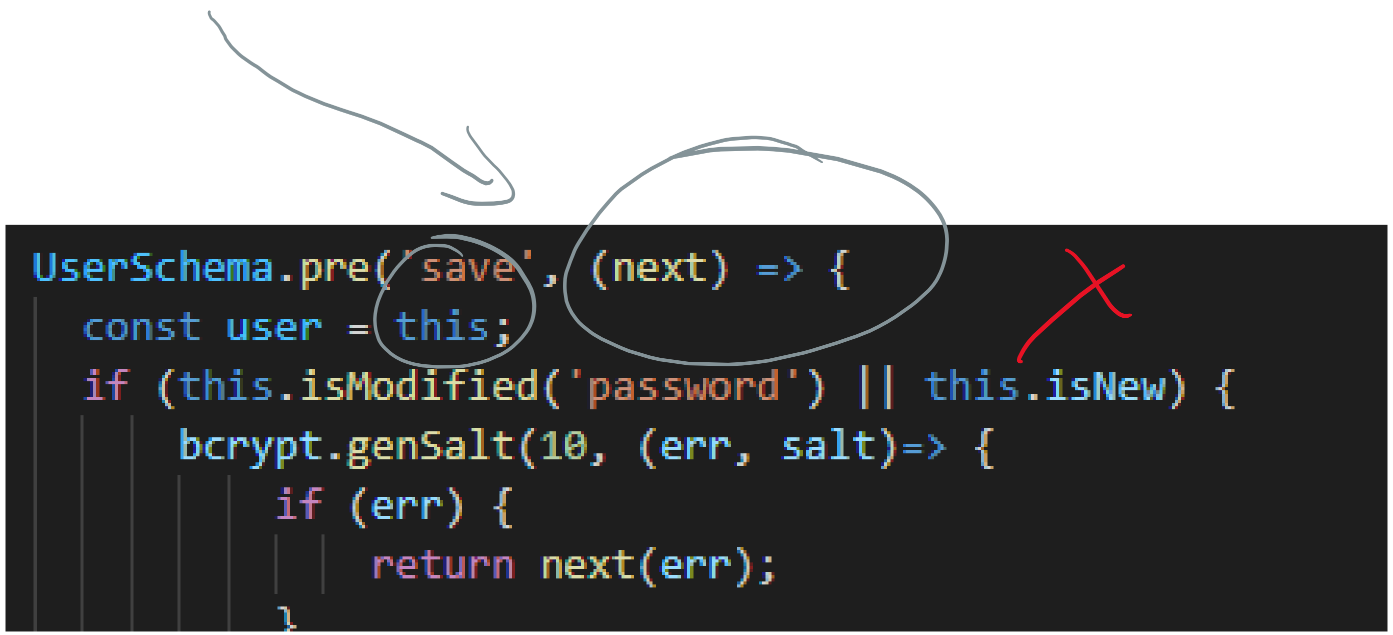


```
bcrypt.genSalt(10, (err, salt)=> {
    if (err) {
        return next(err);
    }
    bcrypt.hash(user.password, salt, null, (err, hash)=> {
        if (err) {
            return next(err);
        }
        user.password = hash;
        next();
    });
});
```

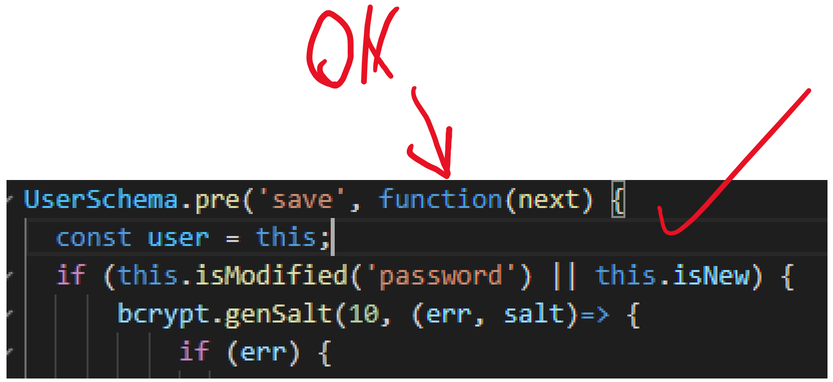
Encrypting - Mongoose User Model

What About this?

- In regular functions the **this** keyword represented the object that called the function
- With arrow (=>) functions, there are no binding of **this**.
  - "this" won't work!



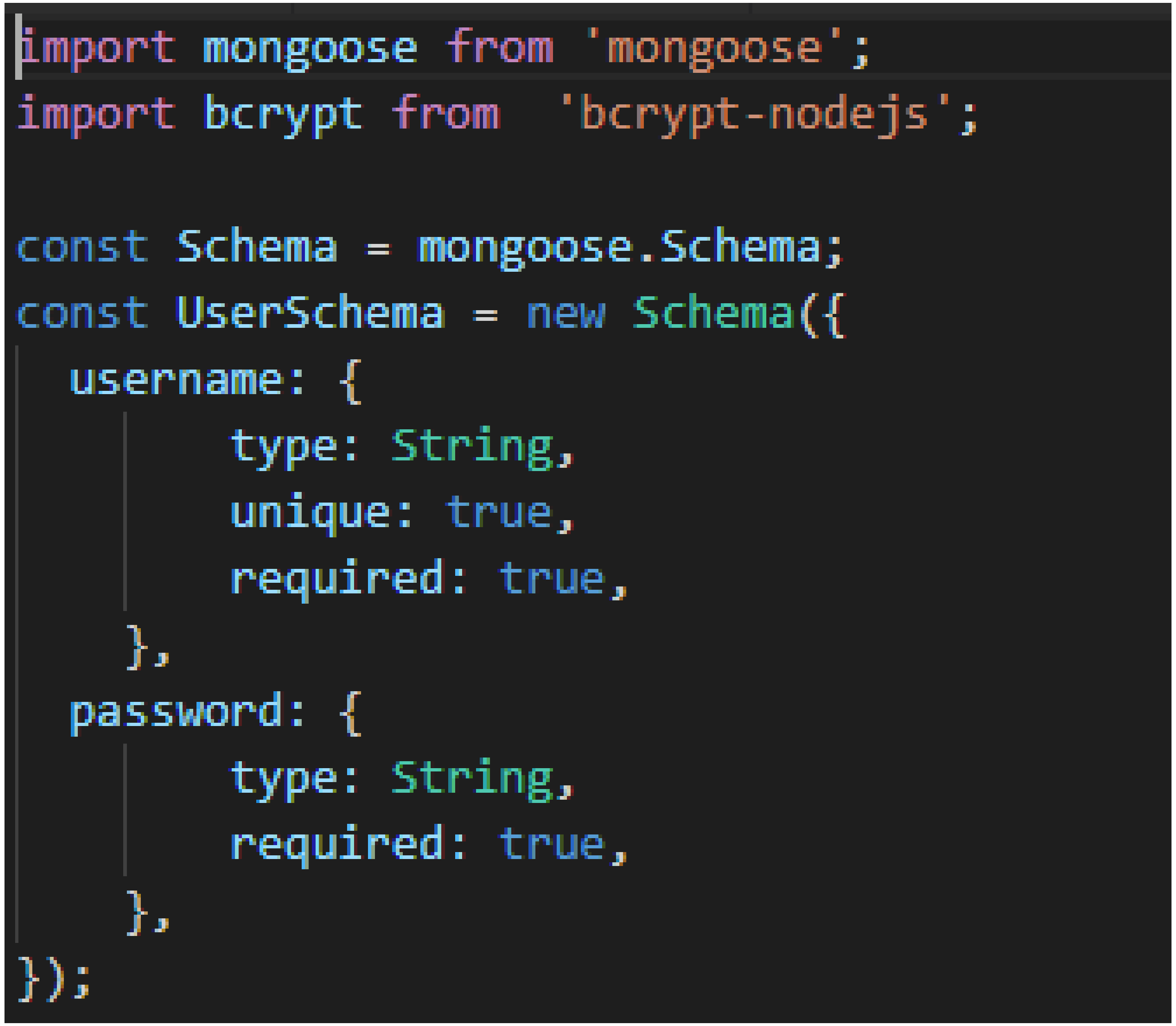
```
UserSchema.pre('save', (next) => {
  const user = this;
  if (this.isModified('password') || this.isNew) {
    bcrypt.genSalt(10, (err, salt)=> {
      if (err) {
        return next(err);
      }
```



```
UserSchema.pre('save', function(next) {
  const user = this;
  if (this.isModified('password') || this.isNew) {
    bcrypt.genSalt(10, (err, salt)=> {
      if (err) {
```

Create Mongoose User Model

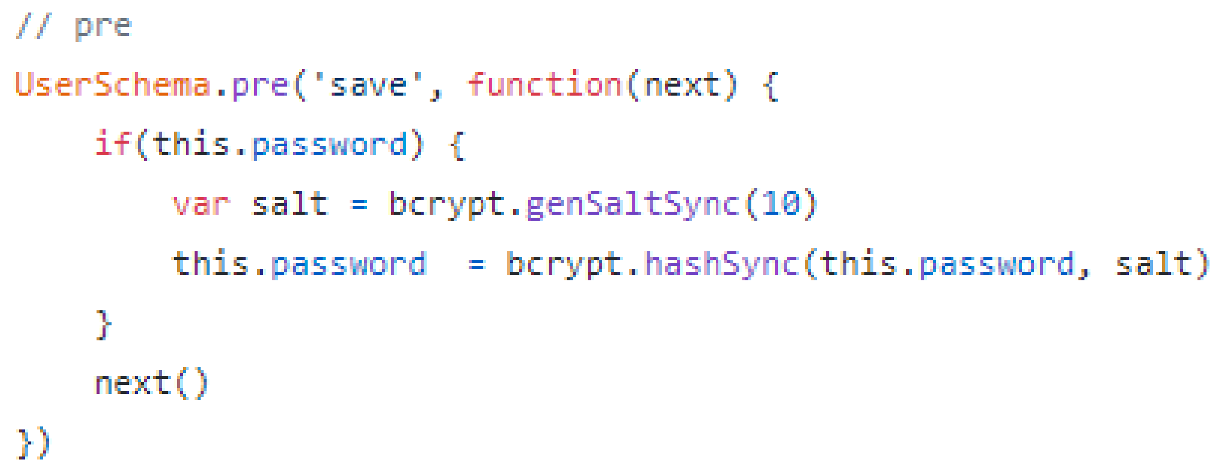
Use Mongoose to specify user model:



```
import mongoose from 'mongoose';
import bcrypt from  'bcrypt-nodejs';

const Schema = mongoose.Schema;
const UserSchema = new Schema({
  username: {
        type: String,
        unique: true,
        required: true,
    },
  password: {
        type: String,
        required: true,
    },
});
```

Mongoose Middleware: Hash/Salt Passwords

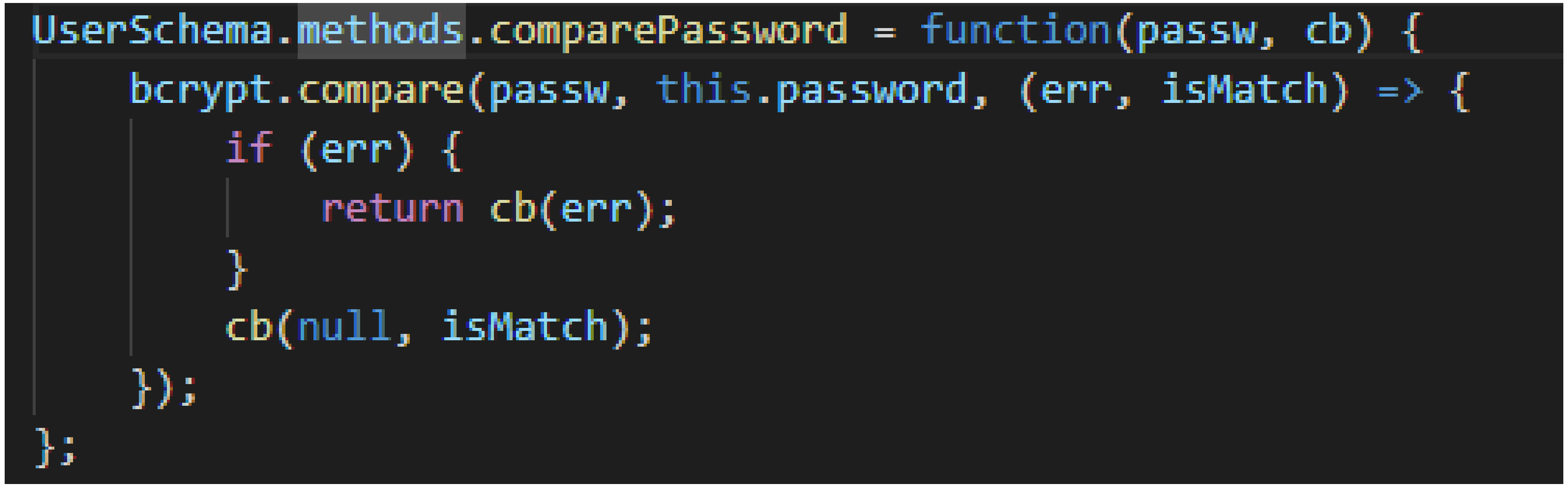
- Mongoose supports Middleware (also called pre and post *hooks*).
- Can use, like Express middleware, to process documents
- Use **bcrypt** package to hash and salt passwords



```
// pre
UserSchema.pre('save', function(next) {
    if(this.password) {
        var salt = bcrypt.genSaltSync(10)
        this.password  = bcrypt.hashSync(this.password, salt)
    }
    next()
})
```

Mongoose Methods: compare passwords

- You can define instance and static methods in Mongoose Schemas.
- For authentication, define a comparePassword(..) instance method
  - Use this to authenticate users
  - **Bcrypt** used to compare with hashed/salted password.



```
UserSchema.methods.comparePassword = function(passw, cb) {
    bcrypt.compare(passw, this.password, (err, isMatch) => {
        if (err) {
            return cb(err);
        }
        cb(null, isMatch);
    });
};
```

User API: User Routes

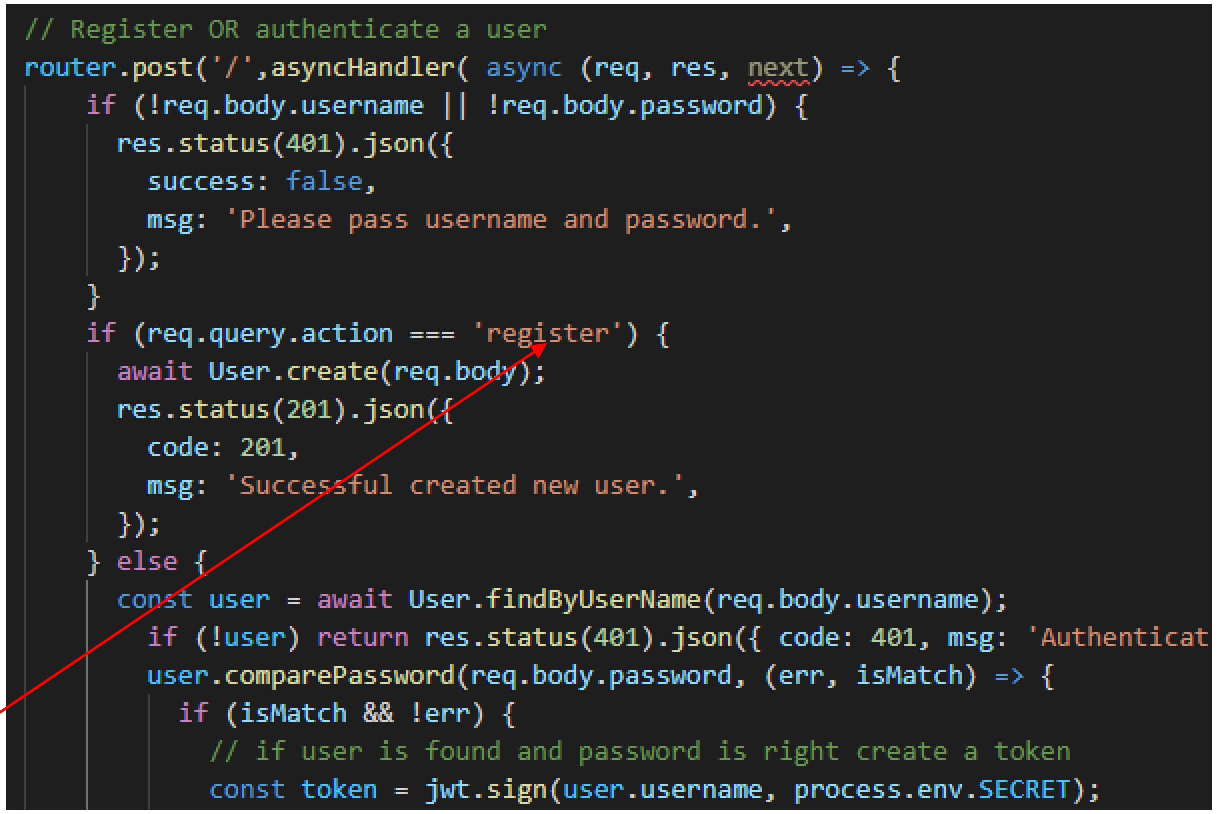
- Create new router to support following API

Route	GET	POST	PUT	DELETE
/api/users	List all users	Register/ Authenticate User	N/A	N/A

User API: Register new user

- Will use query string of URL to indicate action to take on resource
  - **Action===register** will register new user

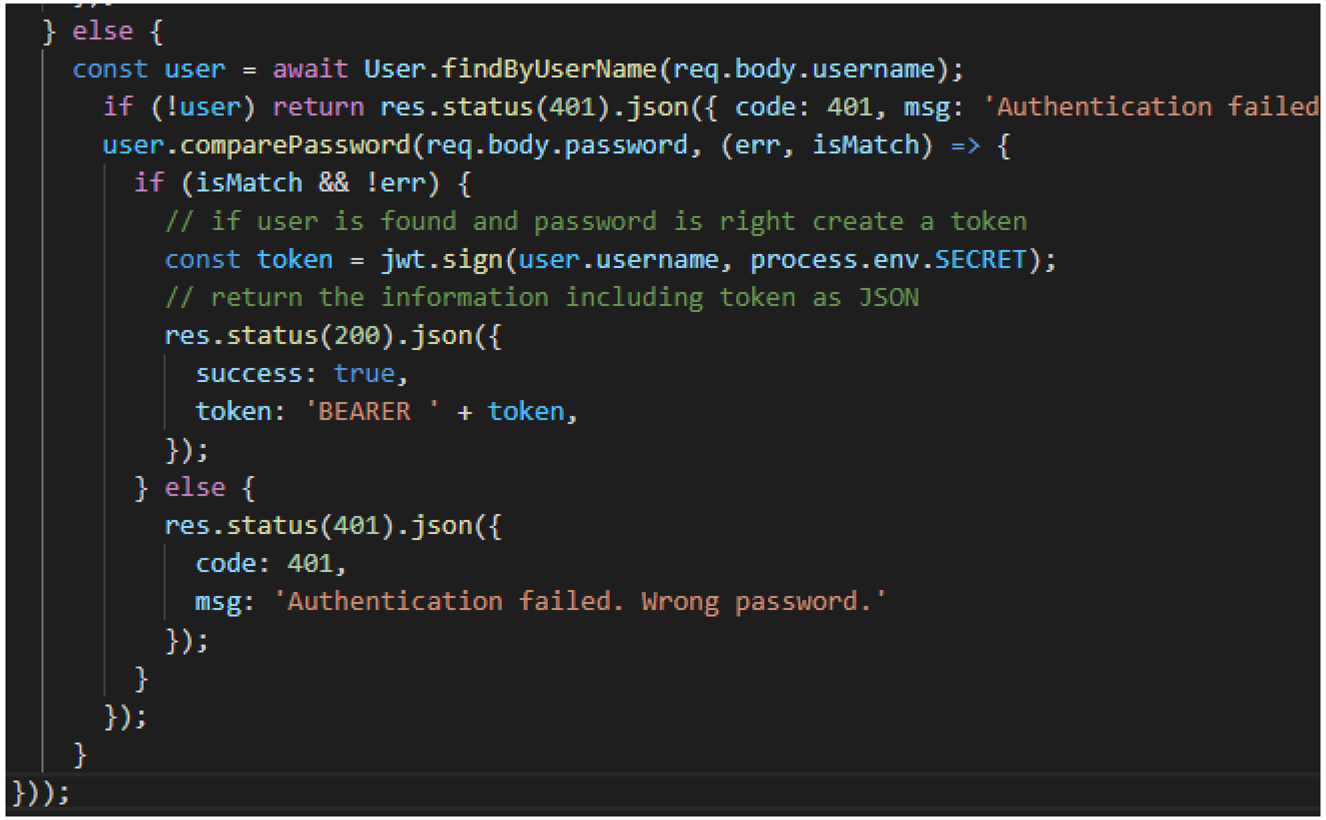
http://localhost:8080/api/users?action=register



```
// Register OR authenticate a user
router.post('/',asyncHandler( async (req, res, next) => {
    if (!req.body.username || !req.body.password) {
      res.status(401).json({
        success: false,
        msg: 'Please pass username and password.',
      });
    }
    if (req.query.action === 'register') {
      await User.create(req.body);
      res.status(201).json({
        code: 201,
        msg: 'Successful created new user.',
      });
    } else {
      const user = await User.findByUserName(req.body.username);
        if (!user) return res.status(401).json({ code: 401, msg: 'Authenticat
        user.comparePassword(req.body.password, (err, isMatch) => {
          if (isMatch && !err) {
            // if user is found and password is right create a token
            const token = jwt.sign(user.username, process.env.SECRET);
```

User API: Authenticate User

- Find user and compare password using user model
- Generate and return JWT token using username field
- **Client needs to keep token for subsequent messaging**
  - **store JWT in local storage.**



```
  } else {
    const user = await User.findByUserName(req.body.username);
      if (!user) return res.status(401).json({ code: 401, msg: 'Authentication failed
      user.comparePassword(req.body.password, (err, isMatch) => {
        if (isMatch && !err) {
          // if user is found and password is right create a token
          const token = jwt.sign(user.username, process.env.SECRET);
          // return the information including token as JSON
          res.status(200).json({
            success: true,
            token: 'BEARER ' + token,
          });
        } else {
          res.status(401).json({
            code: 401,
            msg: 'Authentication failed. Wrong password.'
          });
        }
      });
    }
}));
```

Users API: User Collection

Users Collection	
_id	"5ad46fccada1ab2d67b349ec"
username	"user1"
password	"$2a$10$9r3v12AvPPSkcpJXiohGgehGY50gvgWFV9AAABi37[illegible]aggsPmxBdwW"
__v	0
1	
_id	"5ad46fccada1ab2d67b349ed"
username	"user2"
password	"$2a$10$YZlmbnUSZhBq9FAsAqKTyOJk8uXEweC7XtTNY/ozu8aMGXDW07Xxa"
__v	0

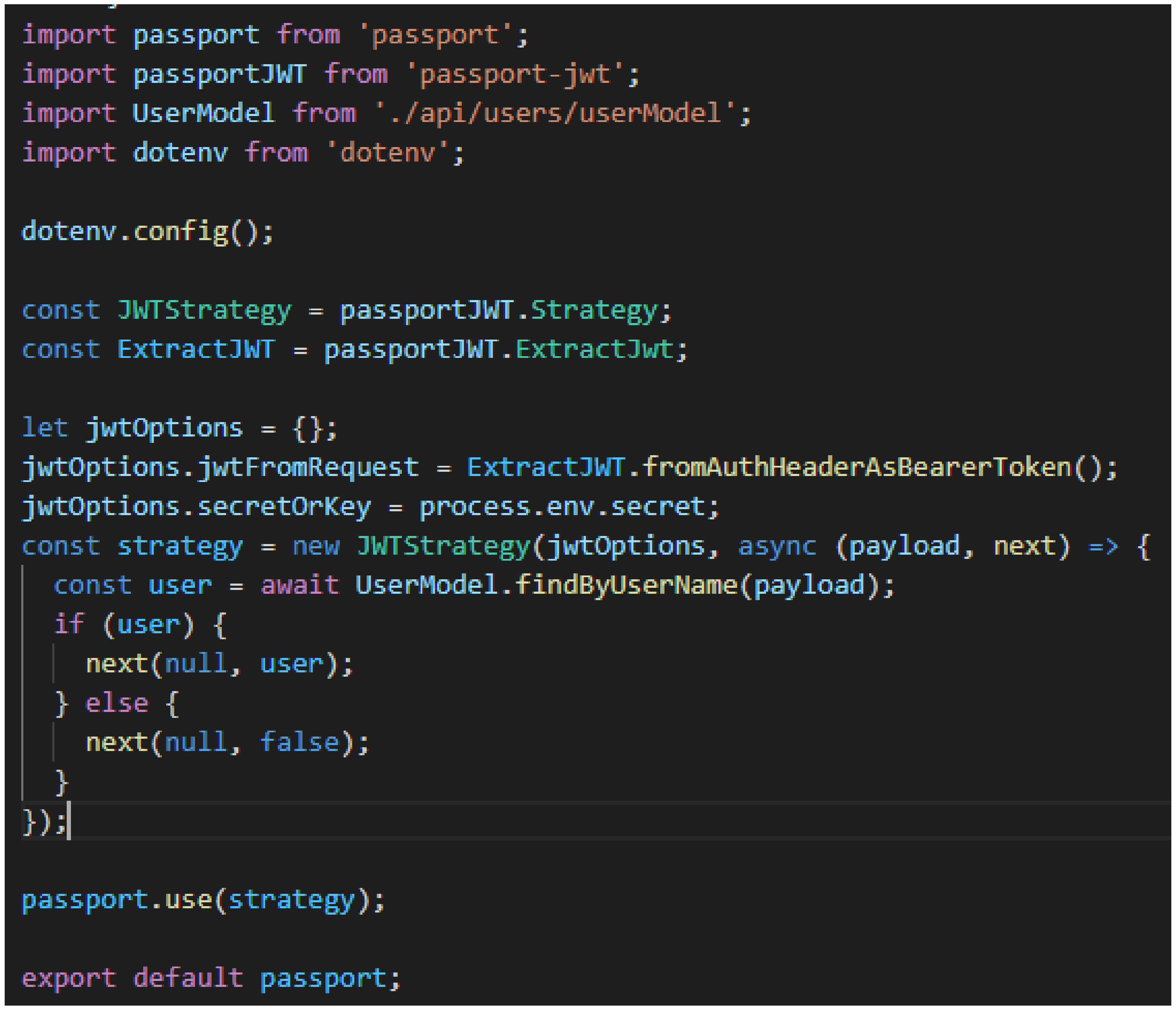
Hashed/Salted value for password "test1"

Protecting Routes with Passport

Protecting API Routes: Passport JWT Policy

- Passport strategies are a middleware functions that a requests runs through before getting to the actual route.
- If the authentication strategy fails,
  - callback will be called with an error
  - the route will not be called and a 401 Unauthorized response will be sent.

/auth/index.js

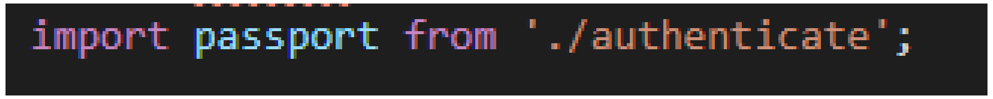


```
import passport from 'passport';
import passportJWT from 'passport-jwt';
import UserModel from './api/users/userModel';
import dotenv from 'dotenv';

dotenv.config();

const JWTStrategy = passportJWT.Strategy;
const ExtractJWT = passportJWT.ExtractJwt;

let jwtOptions = {};
jwtOptions.jwtFromRequest = ExtractJWT.fromAuthHeaderAsBearerToken();
jwtOptions.secretOrKey = process.env.secret;
const strategy = new JWTStrategy(jwtOptions, async (payload, next) => {
  const user = await UserModel.findByUserName(payload);
  if (user) {
    next(null, user);
  } else {
    next(null, false);
  }
});

passport.use(strategy);

export default passport;
```

Protecting API Routes: initialise and add Middleware

In ***/index.js*** of express app

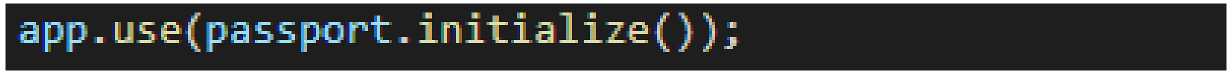
```
// import passport configured with JWT strategy
```



```
import passport from './authenticate';
```

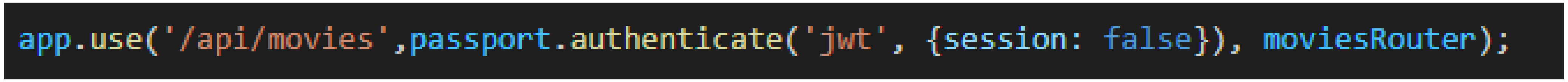
…

```
// initialise passport
```



```
app.use(passport.initialize());
```

```
// Add passport.authenticate(..)  to middleware stack for protected routes
```

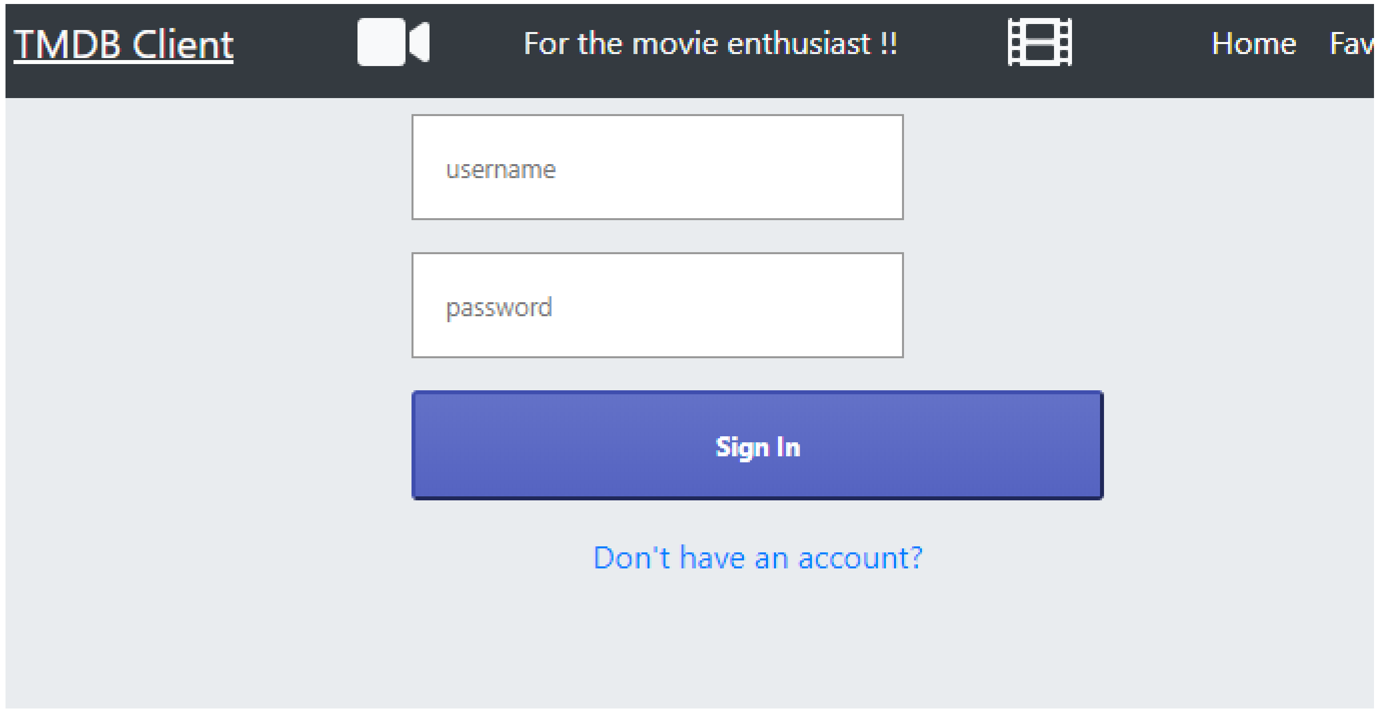


```
app.use('/api/movies',passport.authenticate('jwt', {session: false}), moviesRouter);
```

React Apps and JWT

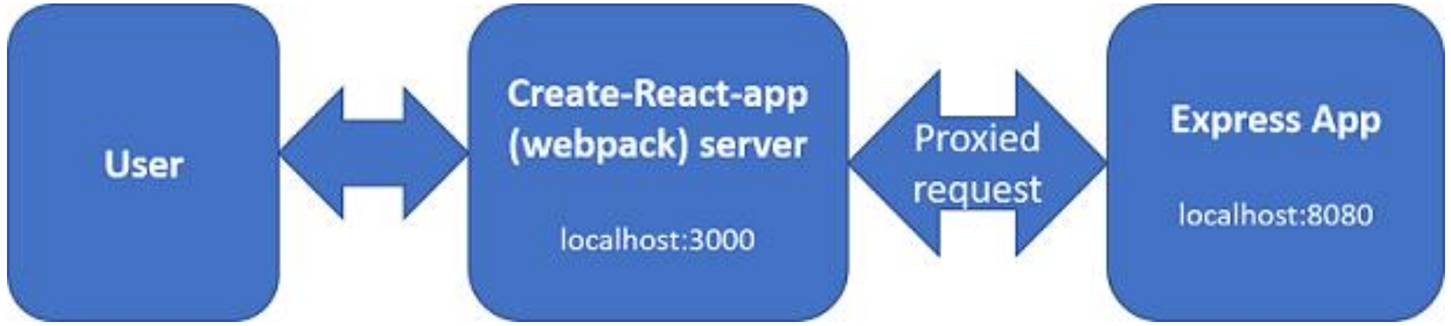
MovieDB App

- We want to:
  - Replace with calls to MovieDB API
  - Provide login/signin capabilities.
  - Only allow signed in users to see Movies and add stuff



Proposed Architecture

- Create-React-app uses Webpack development server.
- MovieDB API is an Express.js app.
- Configure Webpack server to "proxy" any unknown requests to Express app
  - Just need **"proxy":"http://localhost:8080"** entry in package.json.
- Removes Cross-Origin-Resource-Sharing (CORS) issues with the browser

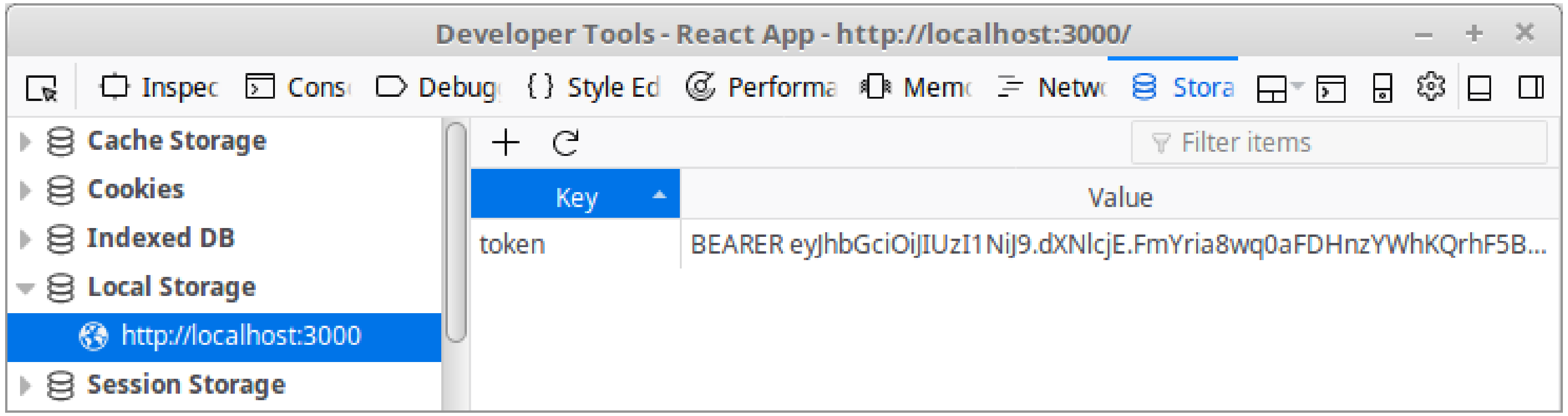


JavaWebToken Storage

- Most browsers/devices have **local storage** .Can access using **localStorage** object.

```
localStorage.setItem('token', token);
```

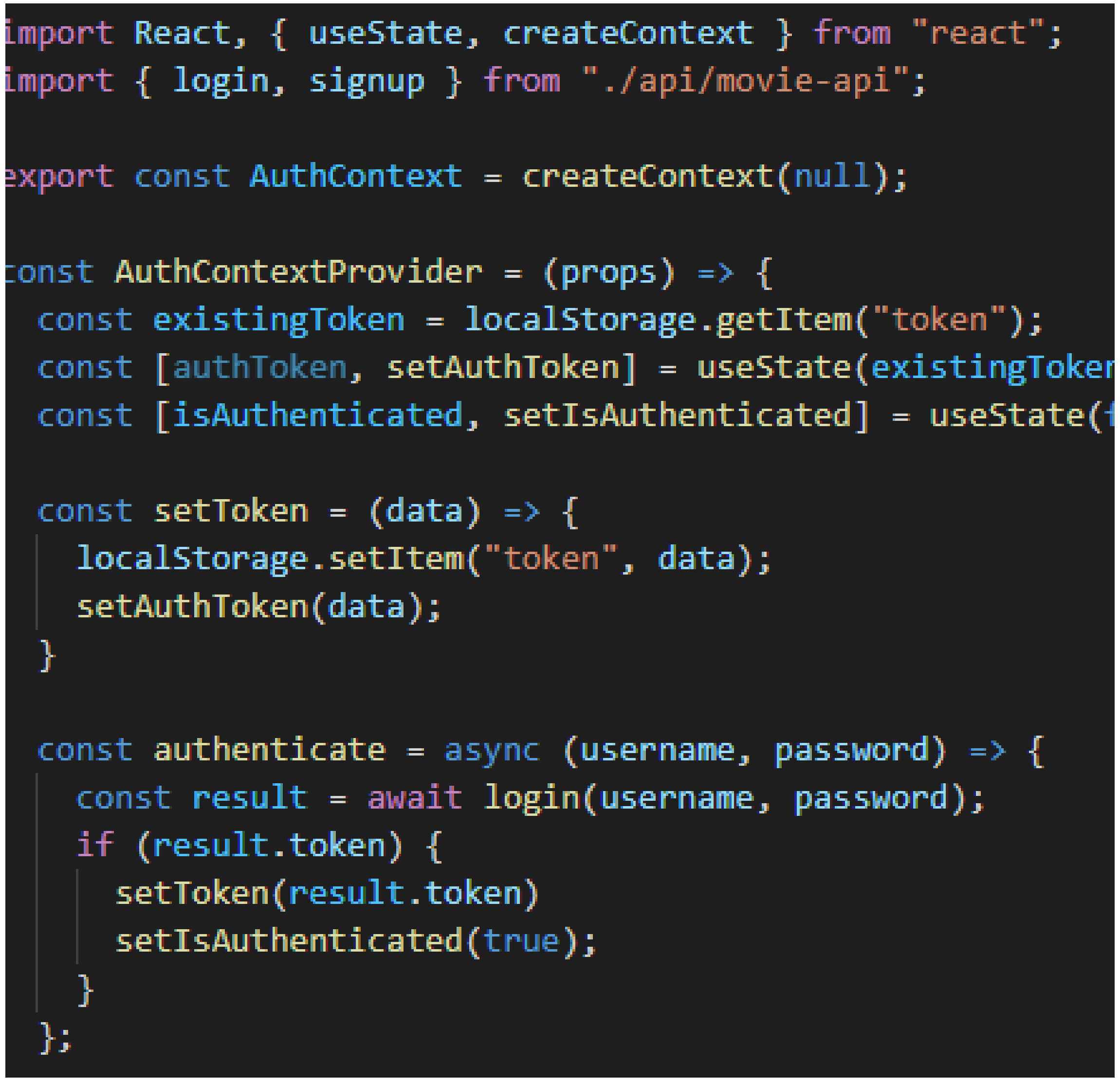
```
 const token =     localStorage.getItem('token');
```



Key	Value
token	BEARER eyJhbGciOiJIUzI1NiJ9.dXNlcjE.FmYria8wq0aFDHnzYWhKQrhF5B...

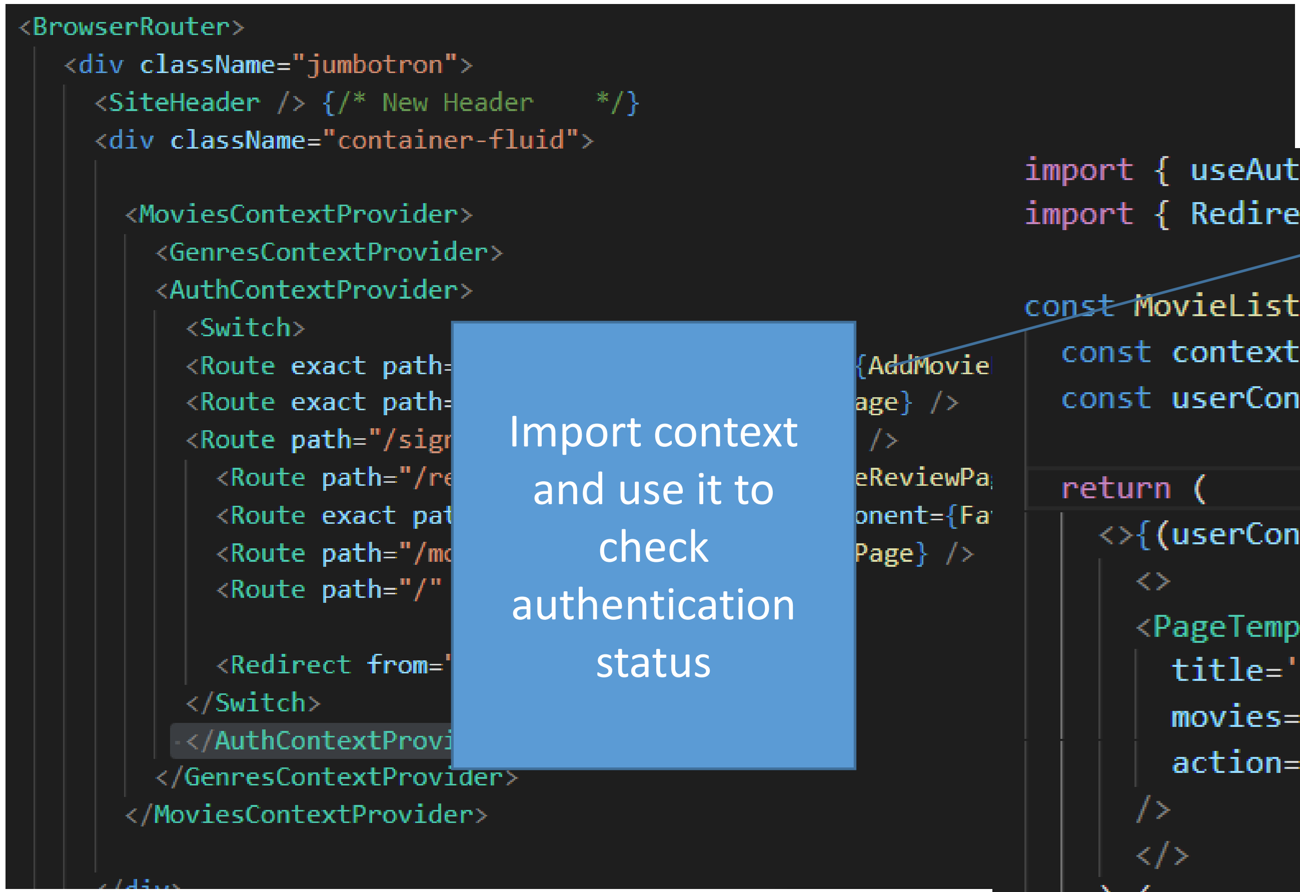
Contexts

- Create a Authentication Context in MovieDB React App.
- As with Movie and Genre contexts, use it to pass data through the component tree
- Share authentication details between components



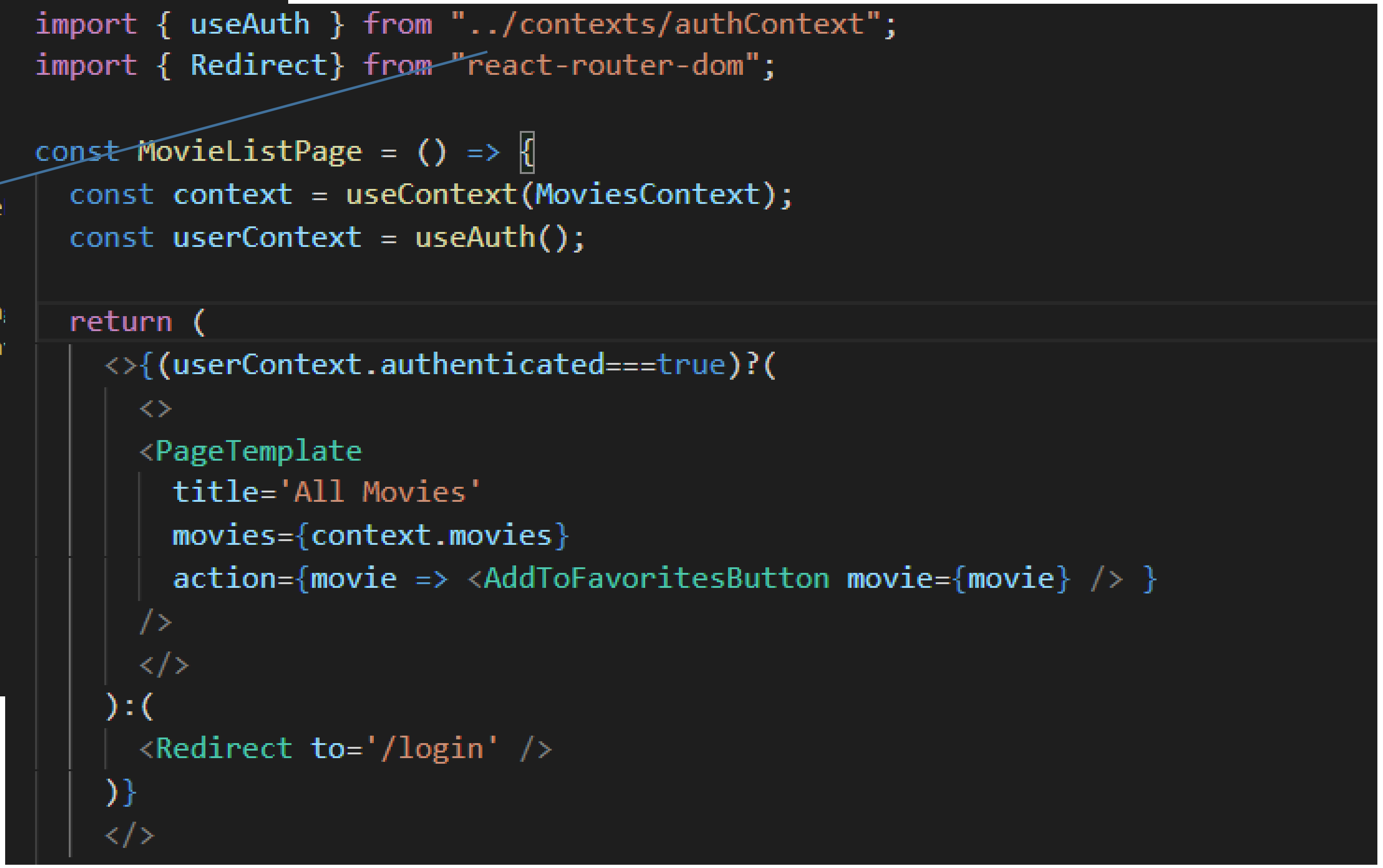
```
import React, { useState, createContext } from "react";
import { login, signup } from "./api/movie-api";

xport const AuthContext = createContext(null);

onst AuthContextProvider = (props) => {
  const existingToken = localStorage.getItem("token");
  const [authToken, setAuthToken] = useState(existingToke
  const [isAuthenticated, setIsAuthenticated] = useState(

  const setToken = (data) => {
    localStorage.setItem("token", data);
    setAuthToken(data);
  }

  const authenticate = async (username, password) => {
    const result = await login(username, password);
    if (result.token) {
      setToken(result.token)
      setIsAuthenticated(true);
    }
  };
```

Use Context Provider in React App



```
<BrowserRouter>
  <div className="jumbotron">
    <SiteHeader /> {/* New Header    */}
    <div className="container-fluid">

      <MoviesContextProvider>
        <GenresContextProvider>
        <AuthContextProvider>
          <Switch>
          <Route exact path=                         {AddMovie
          <Route exact path=                        age} />
          <Route path="/sigr                        />
            <Route path="/re                        eReviewPa
            <Route exact pat                        onent={Fa
            <Route path="/mc                        Page} />
            <Route path="/"

            <Redirect from='
          </Switch>
          </AuthContextProvi
        </GenresContextProvider>
      </MoviesContextProvider>

    </div>
```

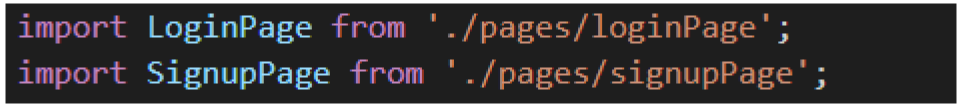
Import context and use it to check authentication status



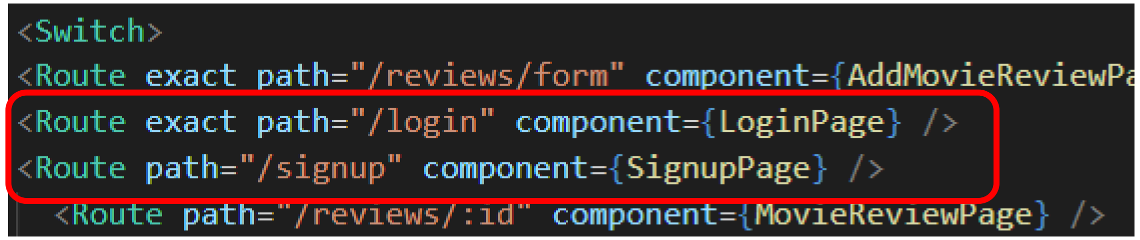
```
import { useAuth } from "../contexts/authContext";
import { Redirect} from "react-router-dom";

const MovieListPage = () => {
  const context = useContext(MoviesContext);
  const userContext = useAuth();

  return (
    <>{(userContext.authenticated===true)?(
      <>
      <PageTemplate
        title='All Movies'
        movies={context.movies}
        action={movie => <AddToFavoritesButton movie={movie} /> }
      />
      </>
    ):(
      <Redirect to='/login' />
    )}
    </>
```

Login/Register Component

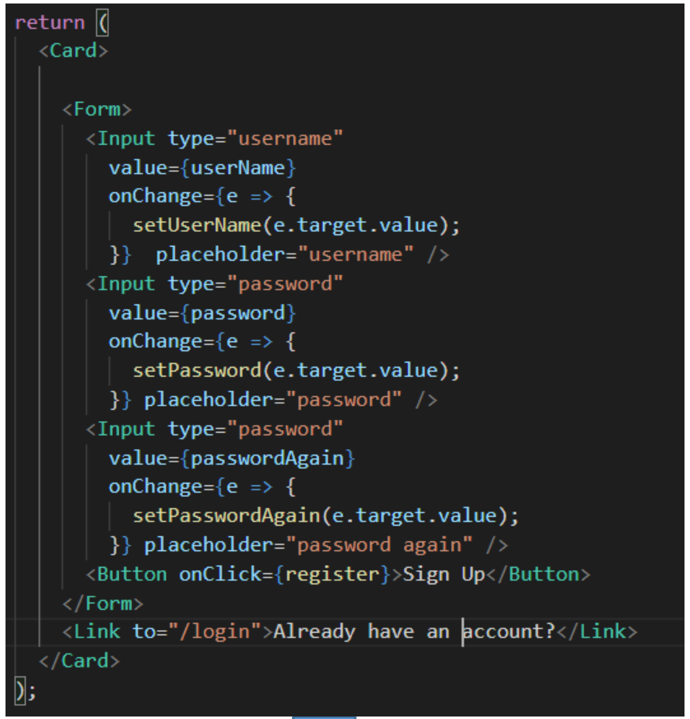
- Add to App router (in index.js)



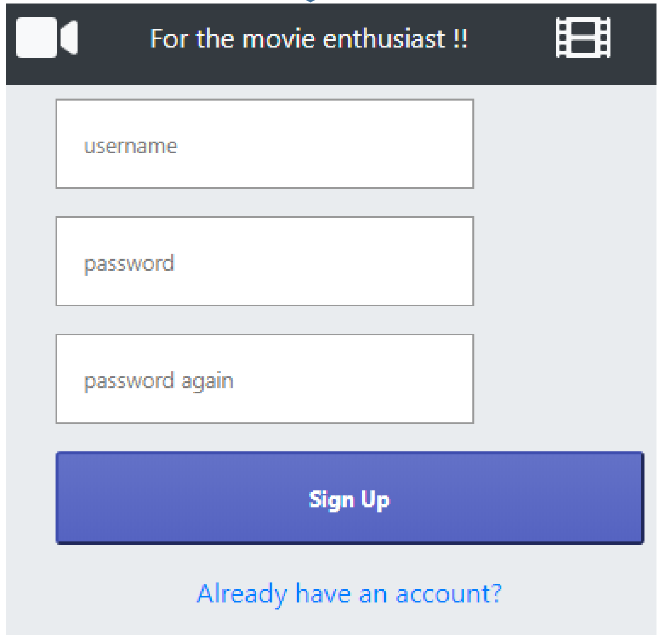
```
import LoginPage from './pages/loginPage';
import SignupPage from './pages/signupPage';
```



```
<Switch>
<Route exact path="/reviews/form" component={AddMovieReviewPa
<Route exact path="/login" component={LoginPage} />
<Route path="/signup" component={SignupPage} />
  <Route path="/reviews/:id" component={MovieReviewPage} />
```



```
return (
  <Card>

    <Form>
      <Input type="username"
        value={userName}
        onChange={e => {
          setUserName(e.target.value);
        }}  placeholder="username" />
      <Input type="password"
        value={password}
        onChange={e => {
          setPassword(e.target.value);
        }} placeholder="password" />
      <Input type="password"
        value={passwordAgain}
        onChange={e => {
          setPasswordAgain(e.target.value);
        }} placeholder="password again" />
      <Button onClick={register}>Sign Up</Button>
    </Form>
    <Link to="/login">Already have an account?</Link>
  </Card>
);
```



Summary

- Create User model with Mongoose
  - Pre-save hook to salt/hash passwords
  - Instance method to compare passwords
- Implement user API to authenticate/signup users
  - Sign JWT tokens with user name
- Add a JWT Strategy to Passport.js
- Use passport.authenticate(…) to secure server-side routes
  - Add to middleware stack.