

# Authentication for Web APIs

*using JSON Web Tokens and Passport*

---

Frank Walsh, 2020

# Agenda

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



# Authentication for MovieDB



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.

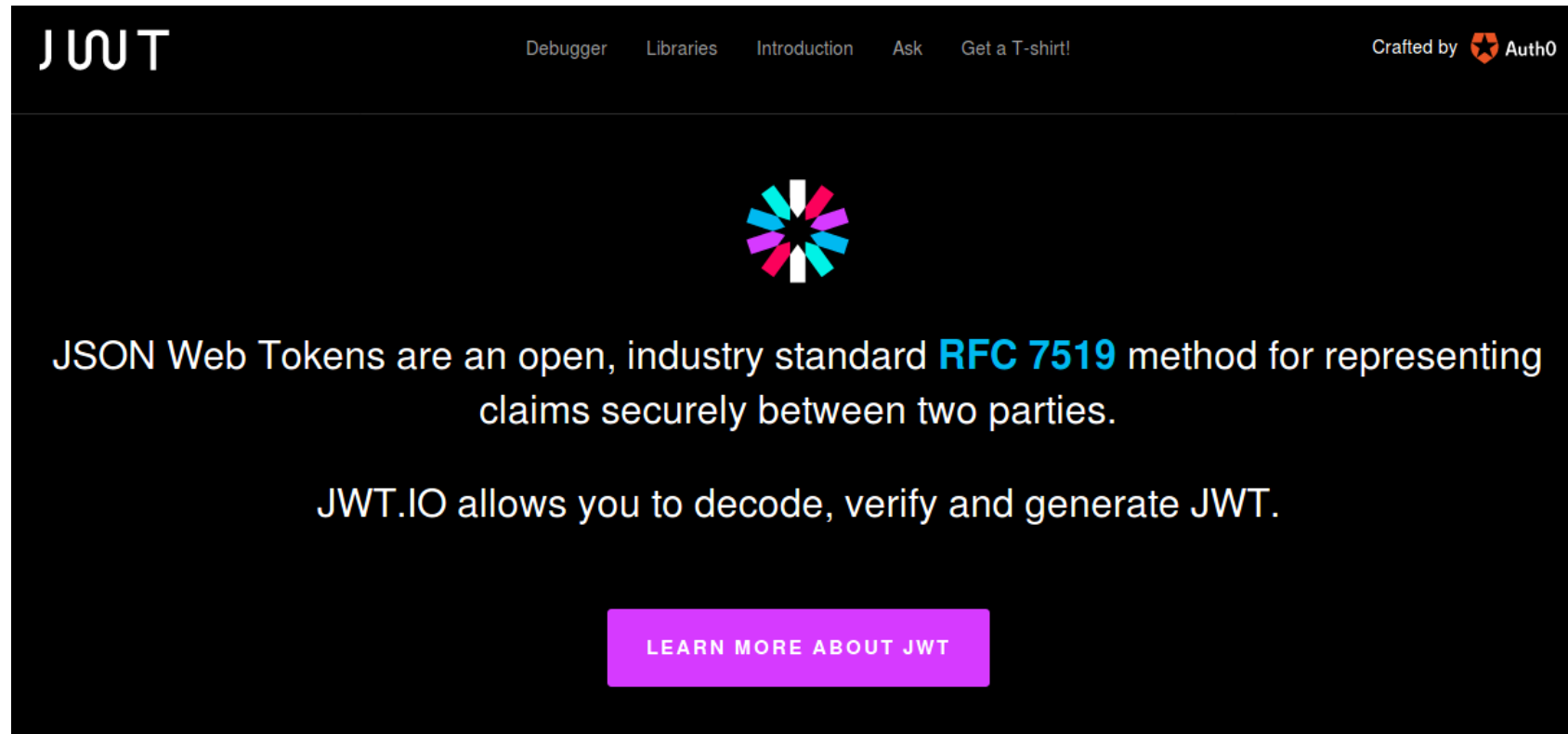


No clear case passwords  
like last week!!!

Hash/Salt all passwords in  
MongoDB

# Authentication options

- Many solutions for Auth
  - Cookies, basic-auth, JWT, OAuth.
  - Web-based Identity Federation/3<sup>rd</sup> 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.

A screenshot of the JWT.IO website. The page has a dark background. At the top left is the 'JWT' logo. To its right is a navigation bar with links: 'Debugger', 'Libraries', 'Introduction', 'Ask', and 'Get a T-shirt!'. On the far right of the top bar, it says 'Crafted by' followed by the Auth0 logo and 'Auth0'. In the center of the page is a colorful, multi-colored starburst logo. Below this logo, the text reads: 'JSON Web Tokens are an open, industry standard **RFC 7519** method for representing claims securely between two parties.' Below this text, it says: 'JWT.IO allows you to decode, verify and generate JWT.' At the bottom center, there is a red rectangular button with the text 'LEARN MORE ABOUT JWT' in white capital letters.

JWT

Debugger Libraries Introduction Ask Get a T-shirt!

Crafted by Auth0

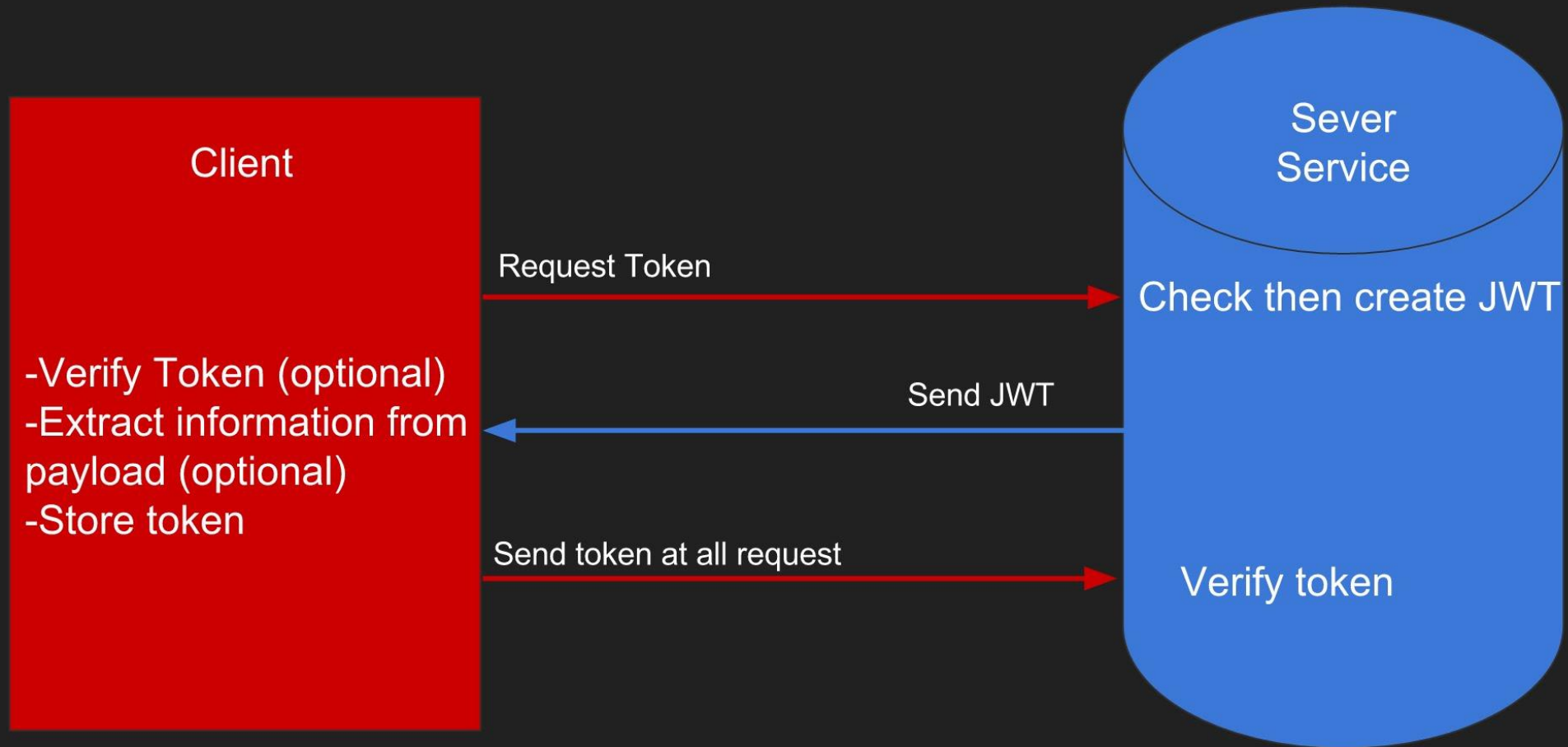
JSON Web Tokens are an open, industry standard **RFC 7519** method for representing claims securely between two parties.

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

LEARN MORE ABOUT JWT

# JSON Web Tokens

## JWT Communication



# 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
  - 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..

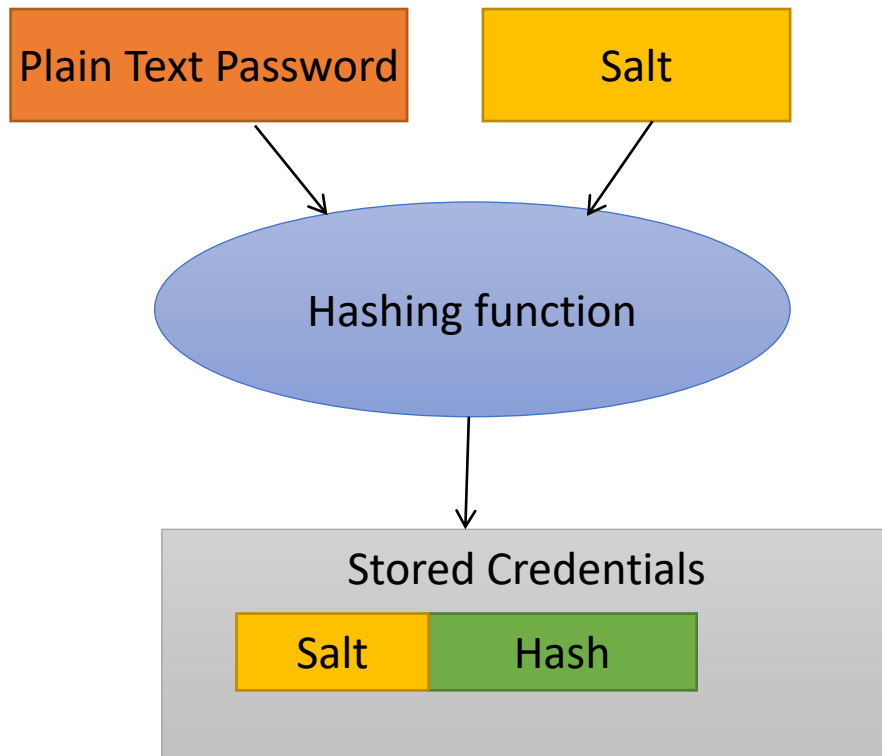
```
app.use('/api/movies', passport.authenticate('jwt', {session: false}), moviesRouter);  
app.use('/api/genres', genresRouter);
```

# Web authentication – credentials

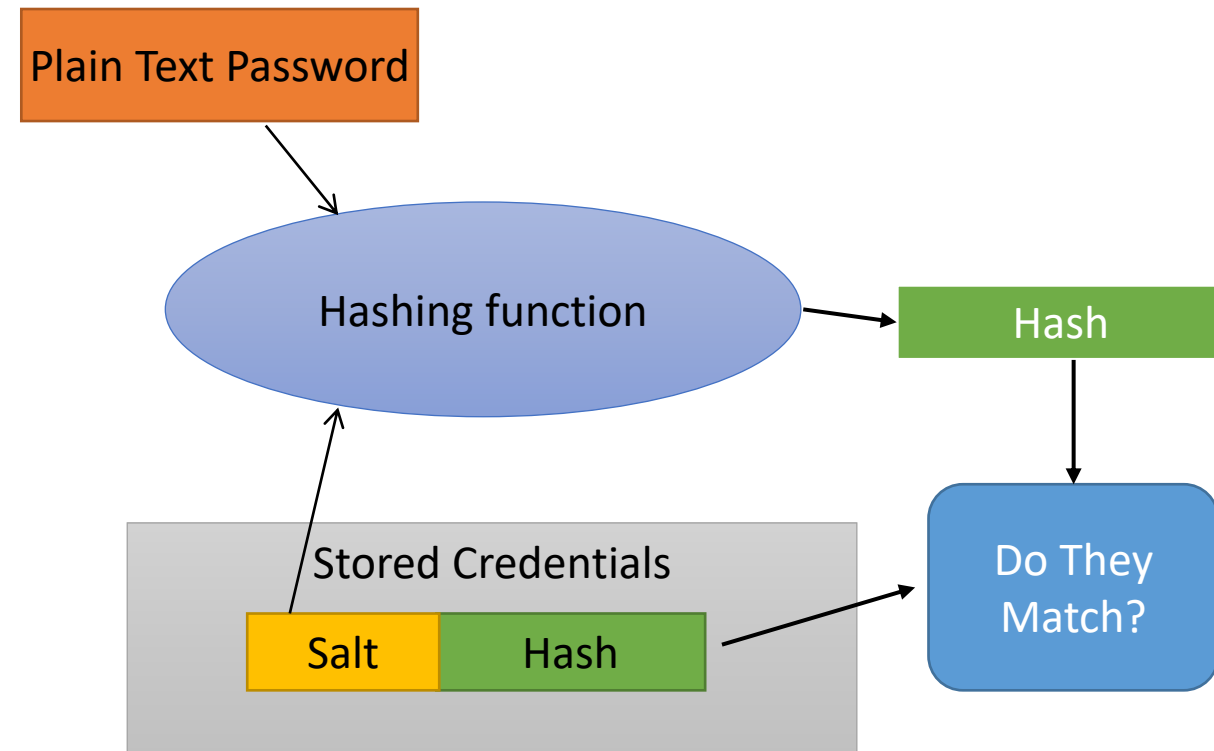
- Credentials should be stored securely in a centralised location
  - Should only be readable by suitably privileged users
  - Credentials should not find their way into hidden fields, headers, cookies
  - Should not be “hard coded”
- Passwords should be “**salted**” and “**hashed**”
  - Salting involves 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.



This Photo by Unknown Author is licensed under [CC BY-SA](#)

# 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

0.0.3

license

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

# 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

```
UserSchema.pre('save', function (next) {
  const user = this;
  if (this.isModified('password') || this.isNew
  ) {
    bcrypt.hash(user.password, 10, (err, hash)
=> {
      if (err) {
        return next(err);
      }
      console.log(hash);
      user.password = hash;
      next();
    });
  }
  else {
    return 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

- Update router to support following API
  - Use query string of URL to specify action:
    - register/authenticate

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

```
// register a user
router.post('/', asyncHandler(async (req, res) => {
  if (req.query.action === 'register') {
    await User.create(req.body);
    res.status(201).json(CreatedResource);
  } else {
```

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



# 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.

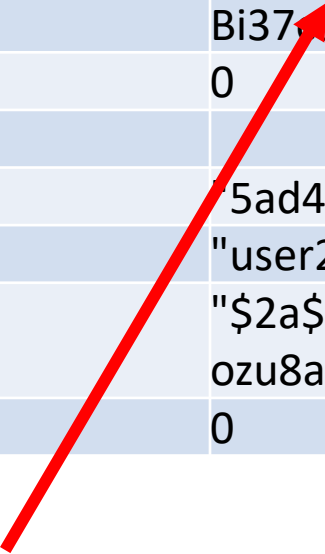
```
} else {
  const user = await User.findByUserName(req.body.username);
  if (!user) return res.status(401).json(Unauthorised);
  user.comparePassword(req.body.password, (err, isMatch) => {
    if (isMatch && !err) {
      // if user is found and password is right create a token
      const token = jwt.sign(user.username, process.env.secret);

      // return the information including token as JSON
      res.status(200).json({
        status_message: "Success",
        status_code: 200,
        token: 'BEARER ' + token,
      });
    } else {
      res.status(401).send(Unauthorised);
    }
  });
}

})))
```

# Users API: User Collection

Users Collection	
_id	"5ad46fccada1ab2d67b349ec"
username	"user1"
password	"\$2a\$10\$9r3v12AvPPSkcpJXiohGgehGY50gvgWfV9AAA Bi37rAggsPmxBdwW"
__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.

/authenticate/index.js

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

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

let jwtOptions = {};
jwtOptions.jwtFromRequest = ExtractJWT.fromAuthHeaderAsBeare
rToken();
jwtOptions.secretOrKey = process.env.secret;
const strategy = new JWTStrategy(jwtOptions, async (payload,
  next) => {
    const user = await UserModel.findByUserName(payload);
    (user) ? next(null, user) : 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 './auth';
```

```
...
```

```
// initialise passport
```

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

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

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

# 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

TMDB Client For the movie enthusiast !! Home Fav

username

password

Sign In

[Don't have an account?](#)

Application	
Manifest	
Service Workers	
Clear storage	

Key	Value
authenticated	false
token	null

# 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

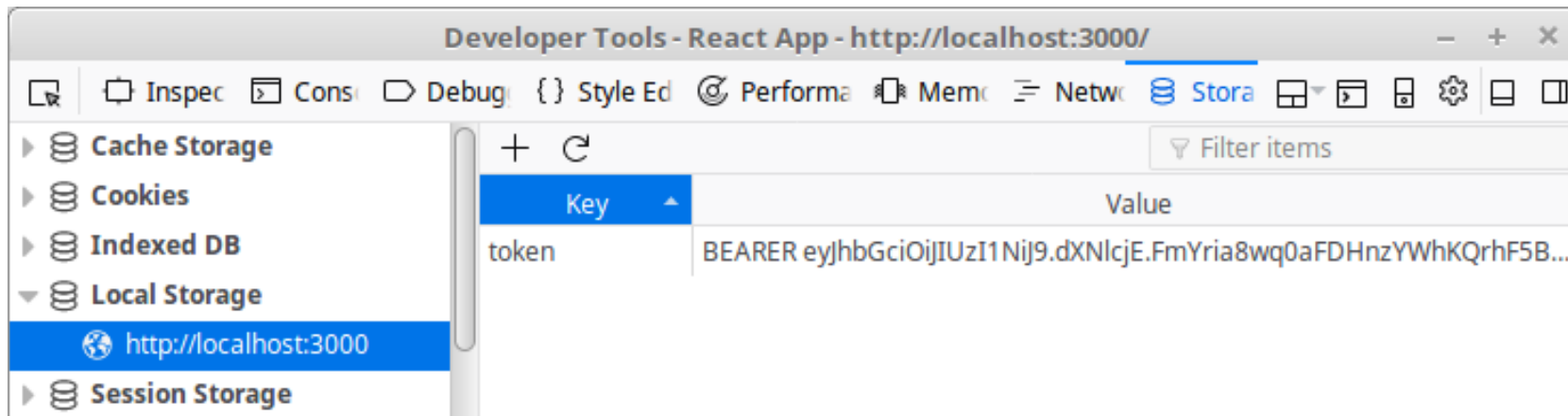


# JavaWebToken Storage

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

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

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



# Contexts

- Create an 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";

export const AuthContext = createContext(null);

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

  //Function to put JWT token in local storage.
  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);
      setUserName(username);
    }
  };
};
```

# Use Context Provider in React App

```
<BrowserRouter>
  <AuthProvider>
    <AuthHeader />
    <ul>
      <li>
        <Link to="/">Home</Link>
      </li>
      <li>
        <Link to="/public">Public</Link>
      </li>
      <li>
        <Link to="/movies">Movies</Link>
      </li>
      <li>
        <Link to="/profile">Profile</Link>
      </li>
    </ul>
    <MovieProvider>
      <Switch>
        <Route path="/public" compone
        <Route path="/login" componen
        <Route path="/signup" compone
        <Route exact path="/" compone
        <PrivateRoute path="/movies"
        <PrivateRoute path="/profile"
        <Redirect from="*" to="/" />
      </Switch>
```

Import context  
and use it to  
check  
authentication  
status

```
import React, { useContext } from "react";
import { Route, Redirect } from "react-router-dom";
import { AuthContext } from '../authContext'

const PrivateRoute = props => {
  const context = useContext(AuthContext)
  // Destructure props from <privateRoute>
  const { component: Component, ...rest } = props;
  console.log(props.location)
  return context.isAuthenticated === true ? (
    <Route {...rest} render={props => <Component {...props} /> } />
  ) : (
    <Redirect
      to={{
        pathname: "/login",
        state: { from: props.location }
      }}
    />
  );
};

export default PrivateRoute;
```

# Login/Register Component

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

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

```
import React, { useContext, useState } from "react";  
import { Redirect } from "react-router-dom";  
import { AuthContext } from '../authContext';  
import { Link } from "react-router-dom";  
  
const LoginPage = props => {  
  const context = useContext(AuthContext)  
  const [userName, setUserName] = useState("");  
  const [password, setPassword] = useState("");  
  
  const login = () => {  
    context.authenticate(userName, password);  
  };  
  
  // Set 'from' to path where browser is redirected after a successful login.  
  // Either / or the protected path user tried to access.  
  const { from } = props.location.state || { from: { pathname: "/" } };  
  
  if (context.isAuthenticated === true) {  
    return <Redirect to={from} />;  
  }  
  return (  
    <>  
      <h2>Login page</h2>  
      <p>You must log in to view the protected pages </p>  
      <input id="username" placeholder="user name" onChange={e => {  
        setUserName(e.target.value);  
      }} /><br />  
      <input type="password" placeholder="password" onChange={e => {  
        setPassword(e.target.value);  
      }} />  
      <input type="password" placeholder="password again" onChange={e => {  
        setPassword(e.target.value);  
      }} />  
      <button type="button" value="Sign Up" />  
      <p>Already have an account? <a href="/login" />Login</p>  
    </>  
  );  
};
```

For the movie enthusiast !!

username

password

password again

Sign Up

Already have an account?



# 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.