

Authentication for Web APIs

using JSON Web Tokens

Frank Walsh, 2020

Agenda

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



Authentication for MovieDB



Restrict access to authenticated users.



Provide **API** to login/register.



Users should only have to
log in once:

Ideally identified and
authenticated in
subsequent requests.



Username and Password authentication.

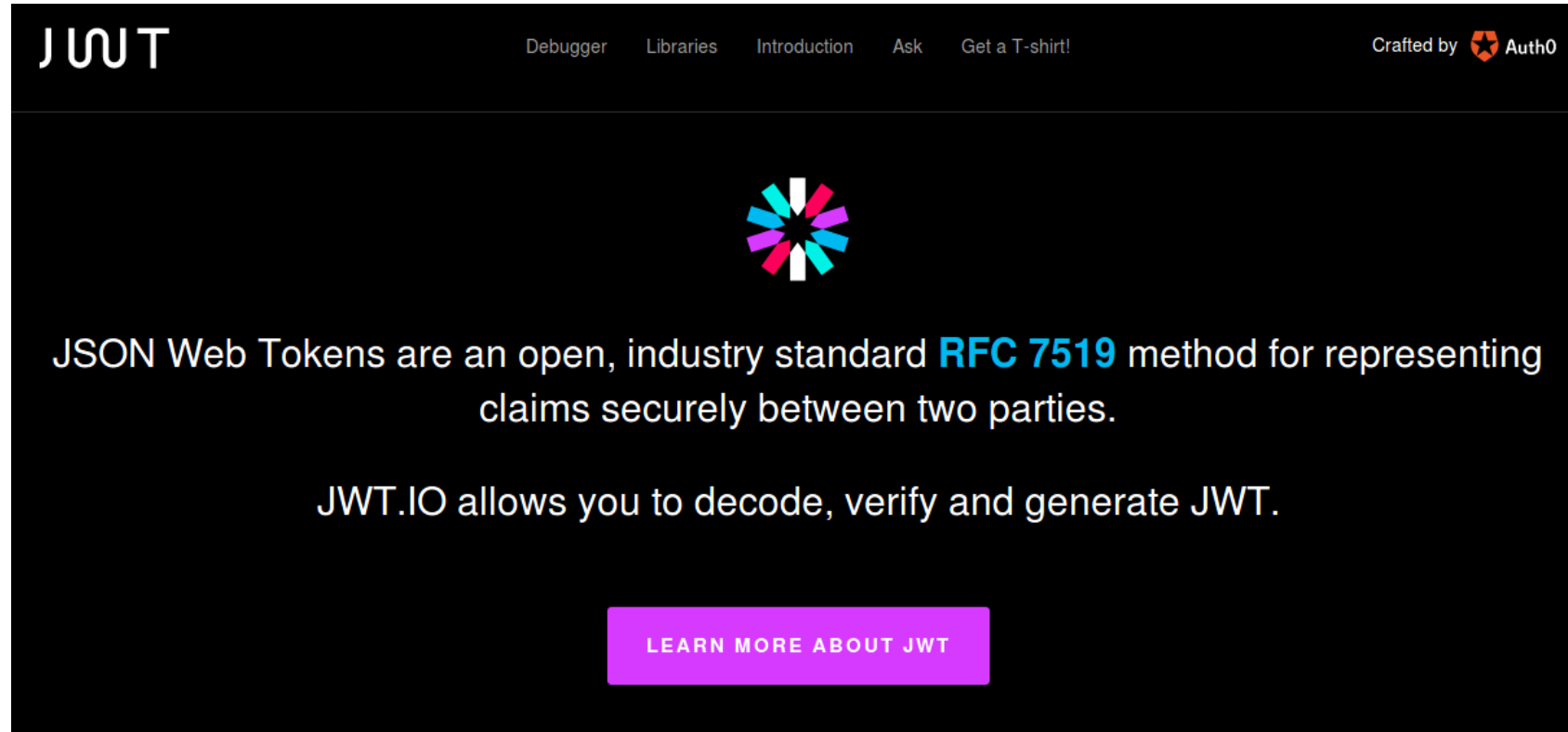


No clear case passwords
like last week!!!

Hash/Salt all passwords in
MongoDB

Authentication options

- Many solutions for Auth
 - Cookies, basic-auth, JWT, OAuth.
 - Web-based Identity Federation/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.

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 navigation 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.

JWT

Debugger Libraries Introduction Ask Get a T-shirt!

Crafted by Auth0

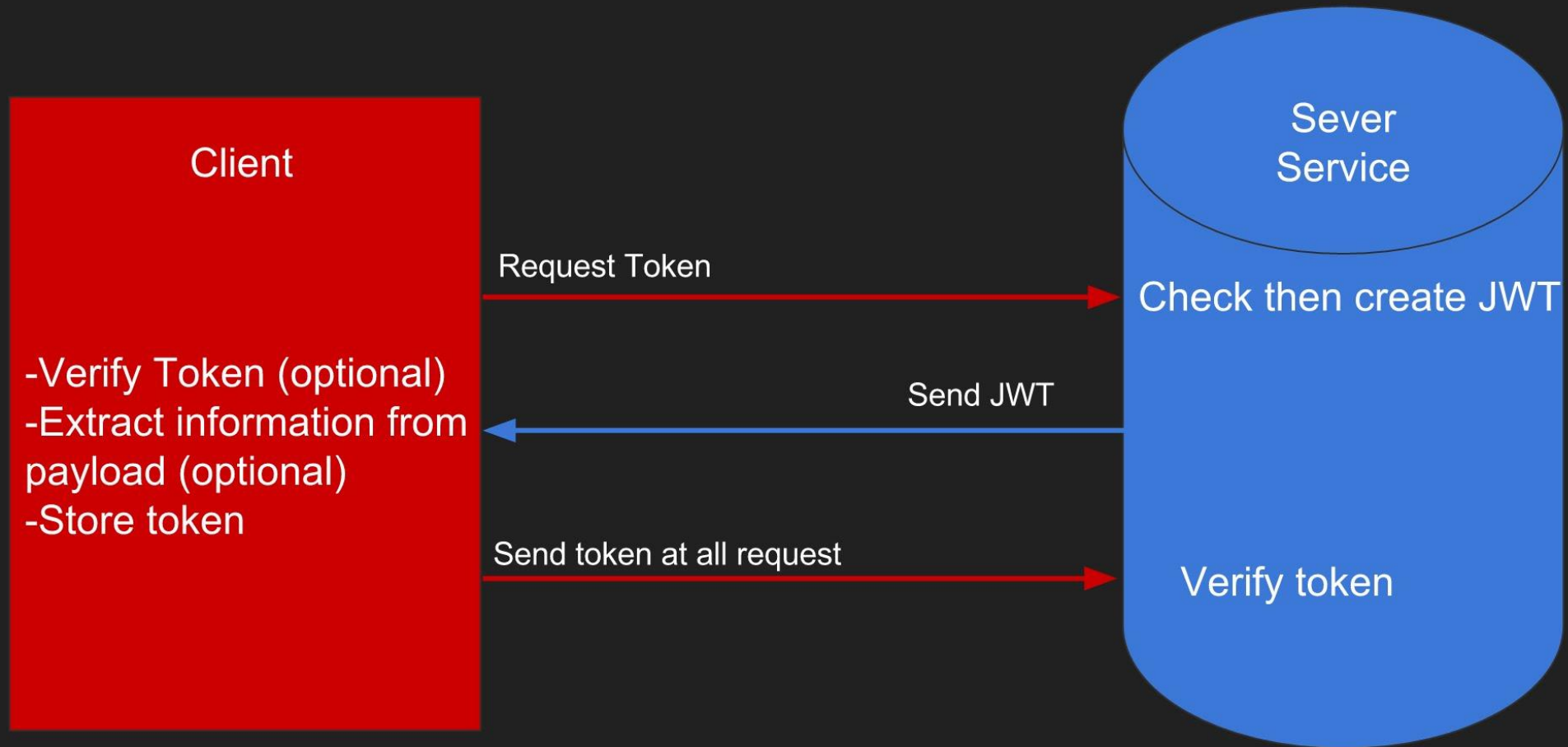
JSON Web Tokens are an open, industry standard **RFC 7519** method for representing claims securely between two parties.

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

LEARN MORE ABOUT JWT


JSON Web Tokens

JWT Communication






JWT in Node/Express


- Use jsonwebtoken NPM module.
- Implementation of JSON Web Tokens RFC


jsonwebtoken 


8.5.1 • Public • Published 3 years ago

 Readme



 Explore 

 10 Dependencies

 18,782 Dependents

 78 Versions

jsonwebtoken

Build	Dependency
 build passing	 Dependency Status

An implementation of **JSON Web Tokens**.

This was developed against `draft-ietf-oauth-json-web-token-08`. It makes use of **node-jws**.


Install

```
$ npm install jsonwebtoken
```


Install

```
> npm i jsonwebtoken
```


Repository

 github.com/auth0/node-jsonwebtoken

Homepage

 github.com/auth0/node-jsonwebtoken#..

Weekly Downloads

8,922,766 

Version	License
---------	---------

Authentication/Security Scenarios

1. Register Account:

- Account signs up to access an API (email & password)
- Create a new account in database with encrypted password

2. Generate security token:

- Client sends email and password to authenticate
- Create a JWT Token and send JWT back to client
- Client stores JWT locally
- JWT used on every subsequent request to protected resource

3. Protect routes (e.g. /api/movies)

- Valid JWT token included in each request
- Token used to identify/validate account before passing request to core service

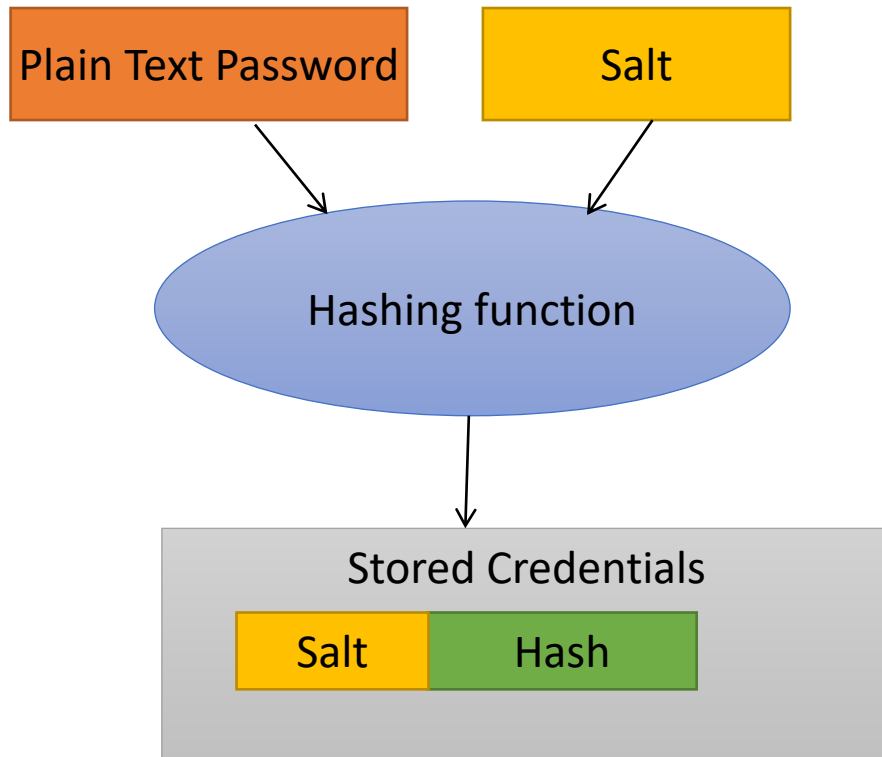
Web Authentication – credentials

- Credentials should be stored securely in a centralised location
 - Should only be readable by suitably privileged accounts
 - **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)

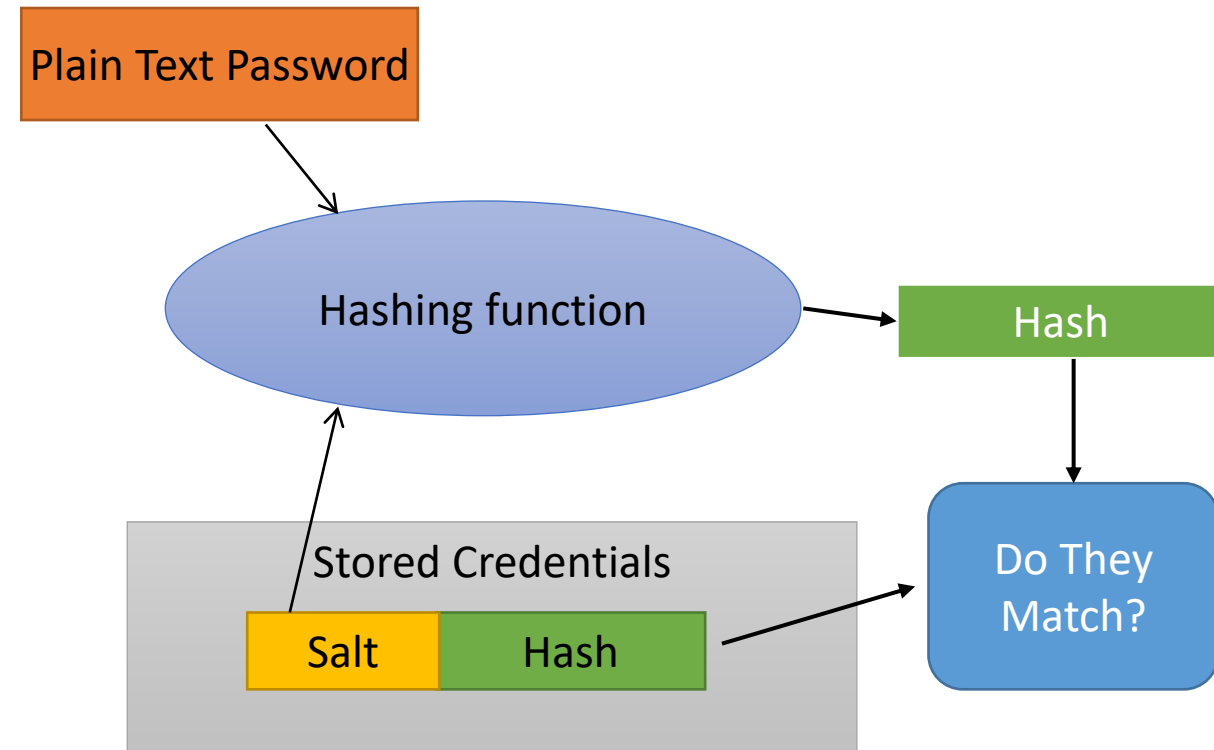


Password Salting & Hashing

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

bcryptjs

2.4.3 • Public • Published 5 years ago

Readme

Explore BETA

0 Dependencies

2,554 Dependents

25 Versions

bcrypt.js

Optimized bcrypt in JavaScript with zero dependencies. Compatible to the C++ **bcrypt** binding on node.js and also working in the browser.

build error npm v2.4.3 downloads 5.5M/month donate

Security considerations

Besides incorporating a salt to protect against rainbow table attacks, bcrypt is an adaptive function: over time, the iteration count can be increased to make it slower, so it remains resistant to brute-force search attacks even with increasing computation power. (see)

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

Install

```
> npm i bcryptjs
```

Repository

github.com/dcodeIO/bcrypt.js

Homepage

github.com/dcodeIO/bcrypt.js#readme

Weekly Downloads

1,338,915

```
import EncryptionService from '../AuthenticationService';
import bcrypt from 'bcryptjs';

export default class extends EncryptionService {

  async encrypt(password) {
    const salt = await bcrypt.genSalt(10);
    return bcrypt.hash(password, salt);
  }
}
```

Encrypting – Middleware Controller

User API: User Routes

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

Route	GET	POST	PUT	DELETE
/api/accounts/security/token	N/A	Authenticate Account using Email/Password	N/A	N/A
/api/accounts/	Admin only	Register Account	N/A	N/A

Encryption and Token dependencies

Clean Architecture for Security/Encryption


- We'll continue to follow Clean Architecture approach as before
 - Code Dependencies can only move from the outer layers inward.
 - To be flexible with project dependencies, use dependency injection to inject external infrastructure/frameworks into your layers
 - Use base classes to define “contracts”, the functional signatures of the desired service without implementation details.
- Services we need to provide here are:
 - Encryption:
 - Password encryption/Password compare
 - Security Token Management
 - Generate/decode Security Token.

Encryption: Hash/Salt Passwords & compare

- Define EncryptionService “Contract”
- Use **bcryptjs** package to hash and salt passwords before saving to repository
- For authentication, use **bcrypt** package compare clear case password (e.g. test123@&) to already encrypted password in the Repository.

```
export default class {  
  encrypt(password) {  
    throw new Error('ERR_METHOD_NOT_IMPLEMENTED');  
  }  
  compare(password, encryptedPassword) {  
    throw new Error('ERR_METHOD_NOT_IMPLEMENTED');  
  }  
}
```

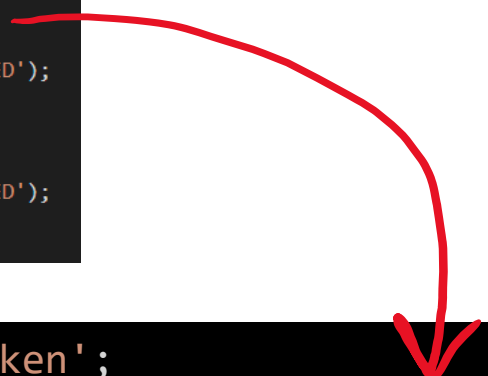
```
import EncryptionService from '../AuthenticationService';  
import bcrypt from 'bcryptjs';  
  
export default class extends EncryptionService {  
  async encrypt(password) {  
    const salt = await bcrypt.genSalt(10);  
    return bcrypt.hash(password, salt);  
  }  
  
  async compare(password, encryptedPassword) {  
    try {  
      // Compare password  
      const result = await bcrypt.compare(password, encryptedPassword);  
      return result;  
    } catch (error) {  
      return false;  
    }  
  }  
}
```



Security Tokens: Using JWT

- Define Token Service “contract” as class
- Use jsonwebtoken package to provide concrete implementation

```
export default class {  
  
  generate() {  
    throw new Error('ERR_METHOD_NOT_IMPLEMENTED');  
  }  
  
  decode() {  
    throw new Error('ERR_METHOD_NOT_IMPLEMENTED');  
  }  
}
```



```
import jwt from 'jsonwebtoken';  
import SecurityTokenService from '../securityTokenService';  
  
export default class extends SecurityTokenService {  
  generate(payload) {  
    return jwt.sign(payload, process.env.JWT_SECRET_KEY );  
  }  
  decode(accessToken) {  
    return  
    jwt.verify(accessToken, process.env.JWT_SECRET_KEY);  
  }  
}
```

Add to Dependencies

- We can add both authentication and token services as project dependency using dependency injection.

```
import Authenticator from '../accounts/security/bcrypt';
import TokenManager from '../accounts/security/jwt';

const buildDependencies = () => {
  const dependencies = {
    validators: AccountValidators,
    authenticator: new Authenticator(),
    tokenManager: new TokenManager()
  };
};
```

/src/config/dependencies.js

1. registerAccount Service

- Update registerAccount to use authenticator dependency. Encrypt Password before persisting...

1. Get Authenticator from dependencies using destructuring

```
registerAccount: async (firstName, lastName, email, password, { accountsRepository, authenticator }) => {  
  password = await authenticator.encrypt(password);  
  const account = new Account(undefined, firstName, lastName, email, password);  
  return accountsRepository.persist(account);  
}
```

2. Encrypt password before persisting

Authenticate Service

- Authenticate Service uses authenticator dependency.
- Uses email to extract account from Repository
- Uses authenticator to compare passwords
- **Client needs to keep token for subsequent messaging**
 - store JWT in local storage.

```
authenticate: async (email, password, { accountsRepository, authenticator, tokenManager }) => {  
  const account = await accountsRepository.getByEmail(email);  
  const result = await authenticator.compare(password, account.password);  
  if (!result) {  
    throw new Error('Bad credentials');  
  }  
  const token = tokenManager.generate({ email: account.email });  
  return token;  
}
```

1. Get Authenticator from dependencies using destructuring

2. Get account from Repo using email and compare password to saved password

3. tokenManager used to generate JWT token, encoding the account email.

Registering Account

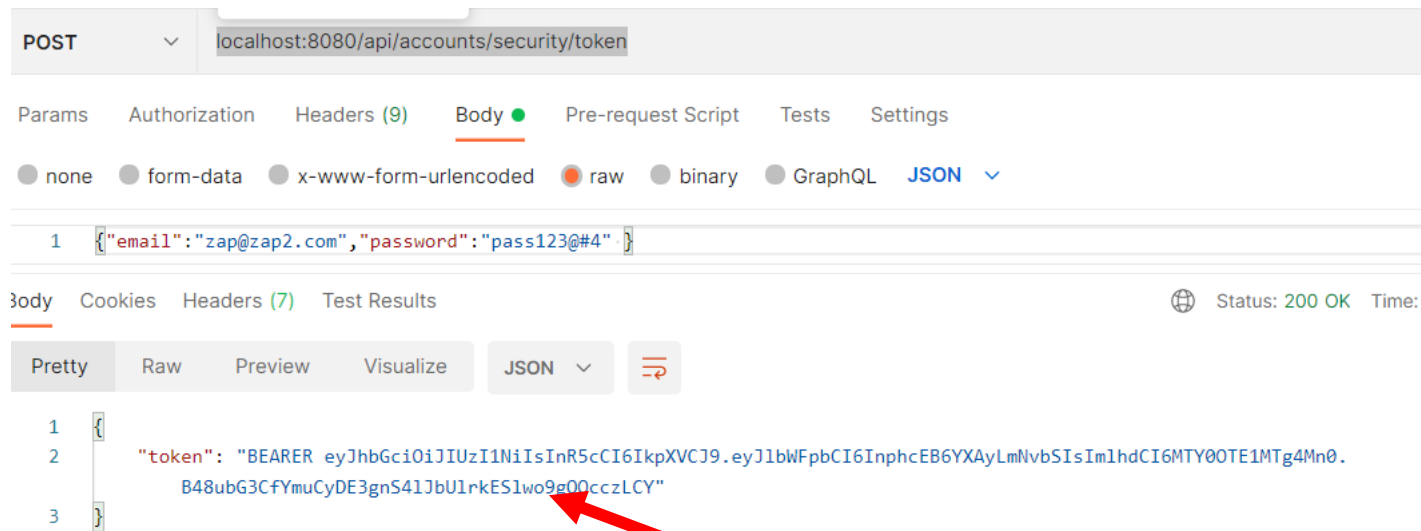
The screenshot shows a REST client interface with a POST request to `localhost:8080/api/accounts/`. The request body is a JSON object: `{"firstName": "Frank", "lastName": "Zappa", "email": "zap@zap2.com", "password": "pass123@#4"}`. The response status is `201 Created`. The response body is displayed in JSON format:

```
1 {
2   "id": "624c0e9b9ef1cac49a68cf29",
3   "firstName": "Frank",
4   "lastName": "Zappa",
5   "email": "zap@zap2.com",
6   "password": "$2a$10$irQy1VnHZ6xxZaZie29Dqe7v1WluqdhzqioInPFSQHzfSSKCr0s1C",
7   "favourites": []
8 }
```

A red arrow points from the `"password"` field in the response to a callout box below.

Hashed/Salted value for password "pass123@#4"

Requesting Security Token



The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** localhost:8080/api/accounts/security/token
- Body:** `{ "email": "zap@zap2.com", "password": "pass123@#4" }`
- Status:** 200 OK
- Response Body (JSON):**

```
{
  "token": "BEARER eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bWVpYmVSIjIm1hdCI6MTY0OTE1MTg4Mn0.B48ubG3CFYmuCyDE3gnS41JbU1rkES1wo9e00cczLCY"
}
```

A red arrow points from the text box below to the `token` value in the response.

JWT Token. This should be stored locally

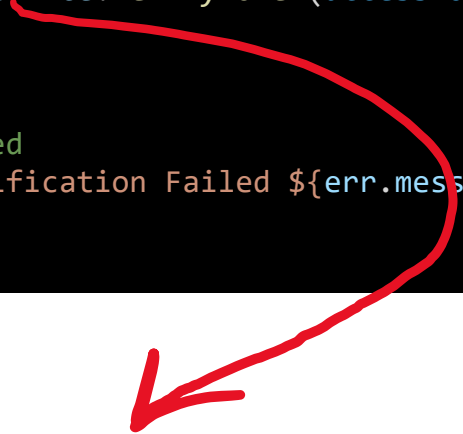
Protecting API Routes using JWT

Protecting API Routes: Create Service and Controller

- Create a new service and controller that checks for and validates the JWT token
- VerifyToken controller: Extracts token from request header and sends to service
- VerifyToken service: Accepts token, decodes email from token, and checks if belongs to an account

/authenticate/index.js

```
const verifyToken = async (request, response, next) => {
  try {
    // Input    const authHeader = request.headers.authorization;
    // Treatment
    if (!authHeader || !authHeader.startsWith('Bearer ')) {
      response.status(403).json({message:"Forbidden"});
    }
    const accessToken = authHeader.split(" ")[1];
    const user = await accountService.verifyToken(accessToken, dependencies);
    //output
    next();
  } catch(err){
    //Token Verification Failed
    next(new Error(`Token Verification Failed ${err.message}`));
  }
};
```



```
verifyToken: async (token,{accountsRepository, tokenManager}) =>
{
  const decoded = await tokenManager.decode(token);
  const user = await accountsRepository.getByEmail(decoded.email);
  if (!user) {
    throw new Error('Bad token');
  }
  return user.email;
}
```


Protecting API Routes: Add to route

- Import Controller into router
- Add to route...

/authenticate/index.js

```
import express from 'express';
import MoviesController from '../controllers';
import AccountsController from '../../accounts/controllers';

const createMoviesRouter = (dependencies) => {
  const router = express.Router();
  // load controllers with dependencies
  const moviesController = MoviesController(dependencies);
  const accountsController = AccountsController(dependencies);

  router.route('/:id')
    .get(moviesController.getMovie);

  router.route('/')
    .get(accountsController.verifyToken, moviesController.find);

  router.route('/:id/reviews')
    .get(moviesController.getMovieReviews);

  return router;
};
export default createMoviesRouter;
```

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		

Possible 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

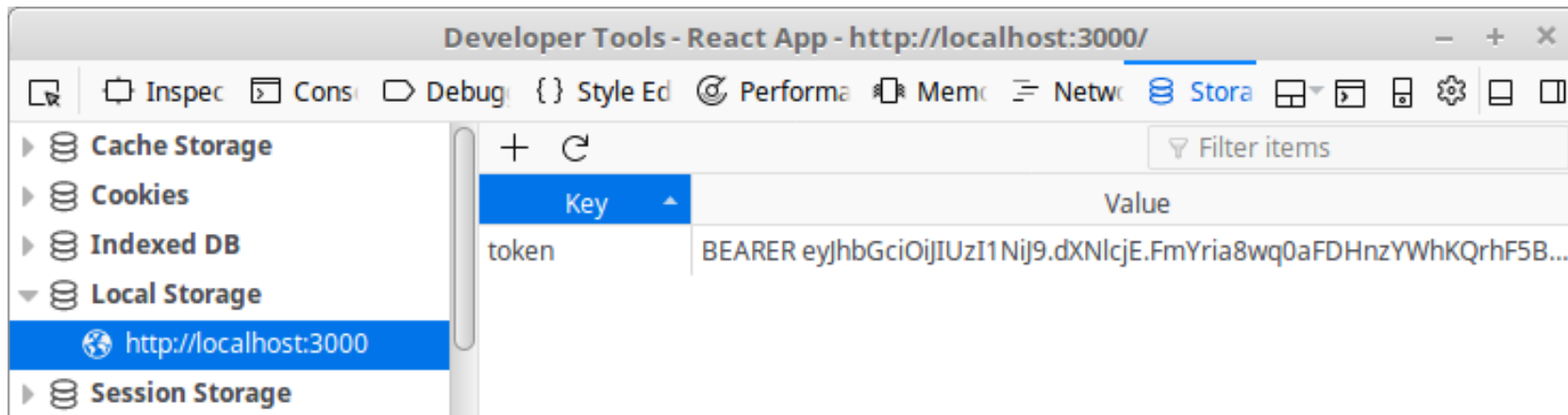


JavaWebToken Storage

- 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" component={Public} />
        <Route path="/login" component={Login} />
        <Route path="/signup" component={Signup} />
        <Route exact path="/" component={Home} />
        <PrivateRoute path="/movies" component={Movies} />
        <PrivateRoute path="/profile" component={Profile} />
        <Redirect from="*" to="/" />
      </Switch>
    </MovieProvider>
  </AuthProvider>
</BrowserRouter>
```

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 Authentication functionality using passwords
 - salt/hash passwords
 - compare passwords
- Implement user API to authenticate/signup users
 - Sign JWT tokens
- Use authentication controller to secure server-side routes
 - Add to middleware stack.

Additional: 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);
```