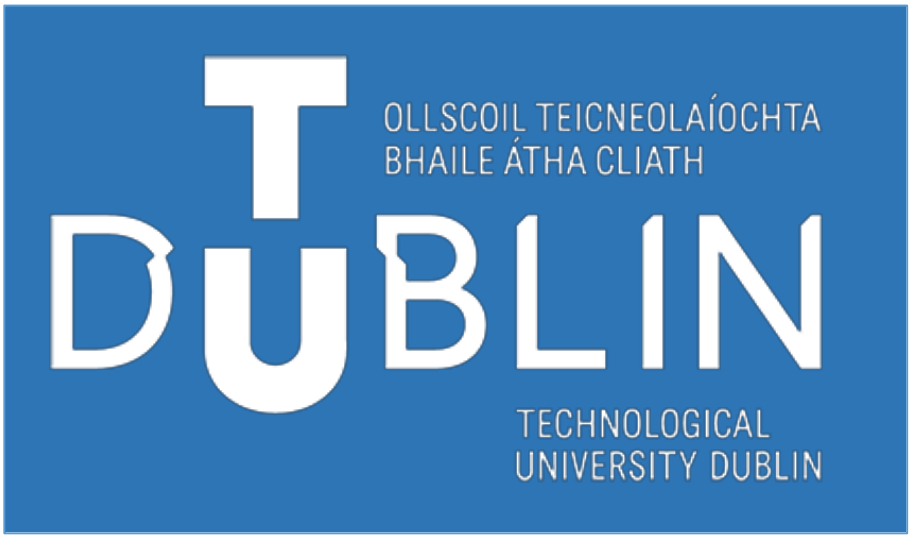


LAB 2 REPORT

Enterprise App Development

- Implemented two different types of API authentication to protect access to a service backend
- Used database encryption features to implement password hashing generation and verification

Eric Strong
C15708709@mydit.ie
DT211C/4

Contents

Video Demo of Lab

I made a video demonstration of the entire lab. It can be viewed here:
https://drive.google.com/drive/folders/18tDkwB3lsKz9FtZ7aYelBlcyOmx4seK3?usp=sharing
All of the work below is included in a walk-through demonstration.

Setting Up

Start with a blank NodeJS (*) project and PostgreSQL (*) database.

I used my lab template I designed in weeks 1 and 2. I ran **npm init** and created a new package.json. I added the following dependencies to the project

- **Nodemod – with a script of npm run dev : nodemon index.js**
- **Express**
- **Body-parser**
- **sequelize**

I then ran my docker instance I created before:
docker run -d -p 5432:5432 --name ead-postgres -e POSTGRES_PASSWORD=7512 postgres

```
eric:$docker run -d -p 5432:5432 --name ead-postgres -e POSTGRES_PASSWORD=7512 postgres
6b23afb4a46d96995377f4bcefdaf604d3b37d4617e3c848c5f3d940e9d29a4c
eric:$docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
6b23afb4a46d        postgres            "docker-entrypoint.s…"   9 seconds ago       Up 7 seconds        0.0.0.0:5432->5432/tcp   ead-postgres
eric:$
```

To access container:
docker exec -it ead-postgres bash

install CURL into my docker container
apt-get update; apt-get install curl

psql -U postgres
CREATE DATABASE lab2

connect to the database: **\c lab2;**

Problem Set 1

Implement a users table having a username and hashed password fields. Use the postgresql **crypt() and gen_salt()** functions to implement the password hashing Implement a protected resource table (e.g. a "products" table) to which you can use to demonstrate your authentication features

create extension pgcrypto;

when inserting into a table we use the crypt function. It takes 2 parameters (yourpassword, gen_Salt(algo)). The gen_salt uses various hashing algorithms to hash a value depending on bit and length.

Algorithm	Max Password Length	Adaptive?	Salt Bits	Output Length	Description
bf	72	yes	128	60	Blowfish-based, variant 2a
md5	unlimited	no	48	34	MD5-based crypt
xdes	8	yes	24	20	Extended DES
des	8	no	12	13	Original UNIX crypt

Input:

```
CREATE TABLE users(
        id SERIAL PRIMARY KEY,
        username TEXT NOT NULL UNIQUE,
        password TEXT NOT NULL
);

INSERT INTO users (username,password)VALUES(
        'c15708709@mydit.ie',
        crypt('password1', gen_salt('bf'))
);

INSERT INTO users (username,password) VALUES (
        'laurabermingham@mydit.ie',
        crypt('password2', gen_salt('bf'))
);
```

Output:

```
id |        username         |                          password
----+-------------------------+-------------------------------------------------------------
  1 | c15708709@mydit.ie      | $2a$06$QM5KD0ngEeO3oepMwLox7uHv4jOpt3gqS74564MK
hmjOSwWMfXuPK
  2 | laurabermingham@mydit.ie | $2a$06$nSwbGlHh/RuCViAOFs.Iw.K35rkIKZPr27vCXgTQ
kD59NGg.DrFpu
(2 rows)
```

CREATE TABLE products(

```
        id SERIAL PRIMARY KEY,
        title TEXT NOT NULL,
        price REAL NOT NULL,
);

INSERT INTO products (title,price) VALUES
('DW drum kit', 2500),
('Macbook Pro', 1800),
('Zildjian Gen 16',760),
('Avengers Infinity War DVD', 9.99);
```

Output:

```
id |          title          | price
----+-------------------------+-------
  1 | DW drum kit             |  2500
  2 | MacbookPro              |  1800
  3 | Zildjian Gen16          |   760
  4 | Avengers Infinity War DVD |  9.99
(4 rows)
```

Code:

```
// problem set 1 - return /users and /products
router.get('/users', (req, res, next) => {

    Users.findAll().then(users => {
        res.send(users);
    }).catch(err => {
        console.log(err)
        res.send(401);
        });
});

router.get('/products', (req, res, next) => {

    Products.findAll().then(products => {
        res.send(products);
    }).catch(err => {
        console.log(err)
        res.send(401);
        });
});
```

Problem Set 2

Implement a JWT-secured version of the API based on the users table from the previous step. Your solution will implement the following API extensions

Demonstrate your JWT authentication on a protected resource

If authenticated or validated, the API return code should be in the 2xx range, otherwise 401.

- A (pre-authentication) login API call which accepts a username and password and returns (if successful) a JWT with a set of claims. The claims should include, minimally, the user id and an expiry timestamp; the token should be set to expire no later than 24 hours

We will need some npm packages:

Npm install jsonwebtoken bcrypt –save

2.1 Endpoint:

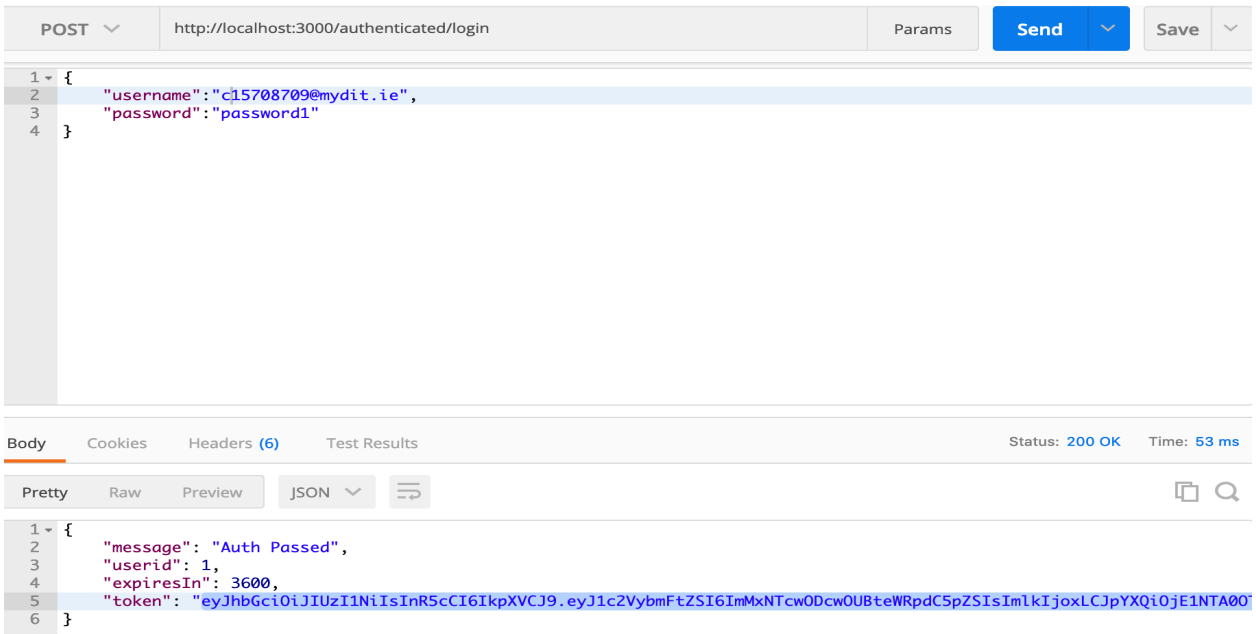
```
router.post('/authenticated/login', (req, res) => {
    console.log('--------in endpoint---------');
    const username = req.body.username;
    const password_clear = req.body.password;
    authenticate(username, password_clear, res);
});
```

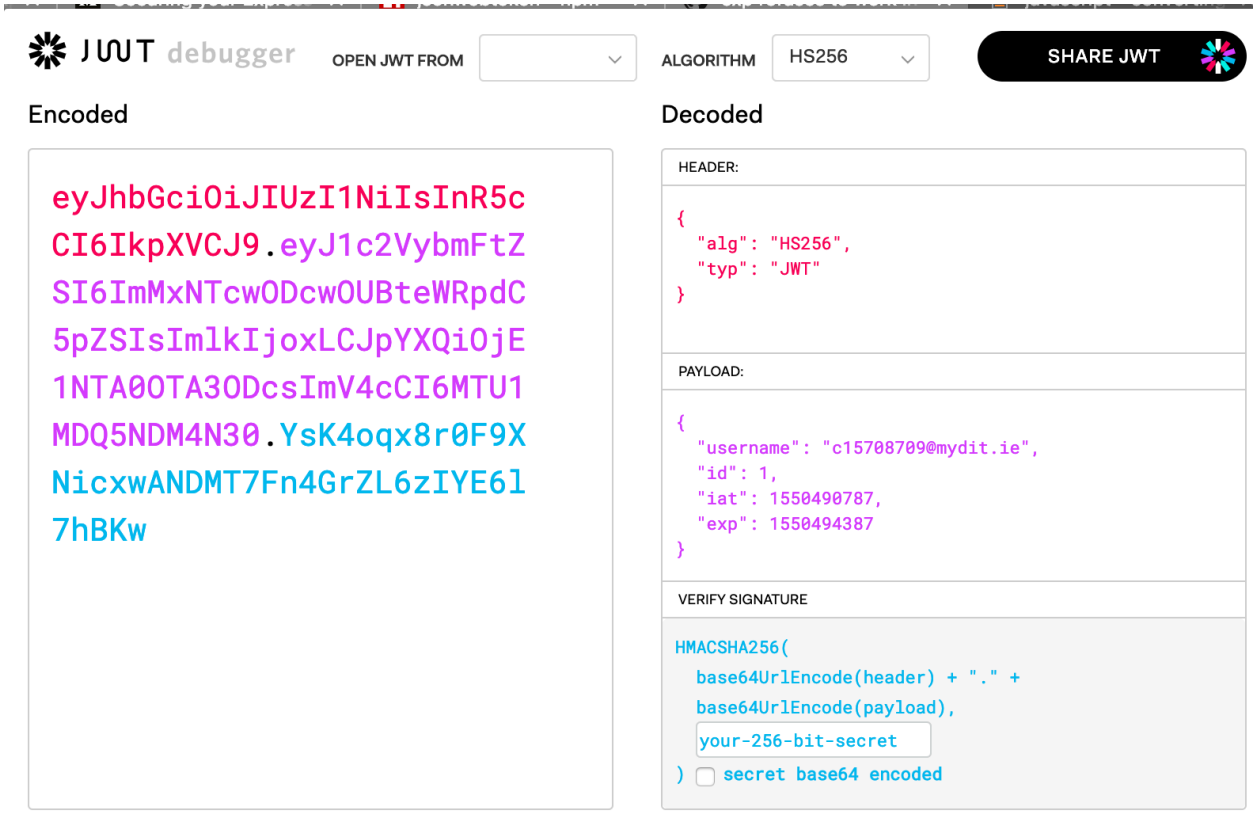
2.1 Function

```
function authenticate(username, password_clear, res) {
    console.log('--------in authenticate---------');

    Users.findOne({
        where: {
            username: username
        }
    }).then(function (user) {
        if (!user) {
            return res.status(401).json({
                message: "Auth Failed. not valid user"
            });
        } else {
            bcrypt.compare(password_clear, user.password, function (err, result) {
                if (result == true) {

                    //generate token
                    const opts = {};
                    opts.expiresIn = 60*60;  //token expires in 1hour
```

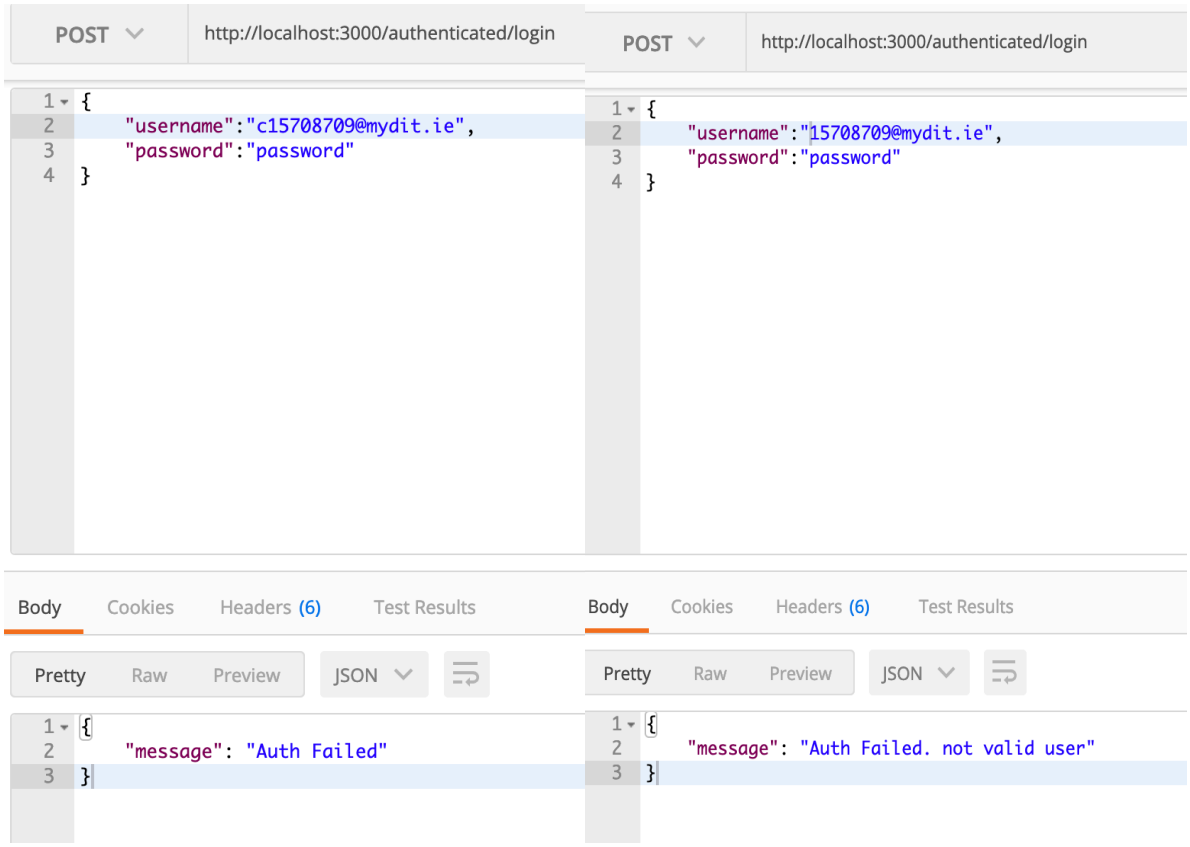
```
                const secret = "SECRET_KEY";

                const token = jwt.sign(
                    {
                        username,
                        id: user.id,


                }, secret,opts);
                return res.status(200).json({
                    message: "Auth Passed",
                    userid:user.id,
                    expiresIn: opts.expiresIn,
                    token
                });

            }

            return res.status(401).json({
                message: "Auth Failed"
            });
        });
    }
});
}
```

2.1 Input





Wrong password or username output:



- A mechanism to verify client tokens as bearer tokens in a HTTP Authorization header field

2.2 Endpoint

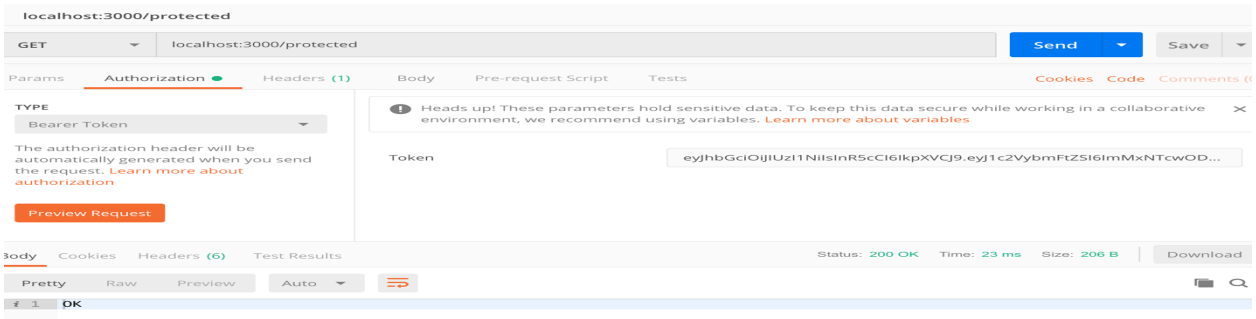
```
router.get("/protected", methods.validateToken, (req, res, next) => {
    res.sendStatus(200);
   })
```

Methods : methods.js

```
let jwt = require('jsonwebtoken')
module.exports.validateToken = function (req, res, next) {
        var bearerHeader = req.headers["authorization"]
        if (typeof bearerHeader !== 'undefined') {
            const bearer = bearerHeader.split(" ")
            const bearerToken = bearer[1]

            jwt.verify(bearerToken, 'SECRET_KEY', (err, result) => {
                    if (err) {
                        console.log(err);
                        res.sendStatus(403)
                    } else {
                        console.log(result);
                        next(); // we are now verified, so we can use other or next
methods
                    }
                })
            }else{
                console.log('error');
                    res.sendStatus(403)
                }
            };
```

Input:

We need to use our token from part 2.1
We use postman and paste the bearer token in the authorization – type tab



- **Authentication** should be applied, minimally, to any API calls which update any tables; **Token validation** should be performed on all API calls

I went back and created two new endpoints:

/protected/users

/protected/products

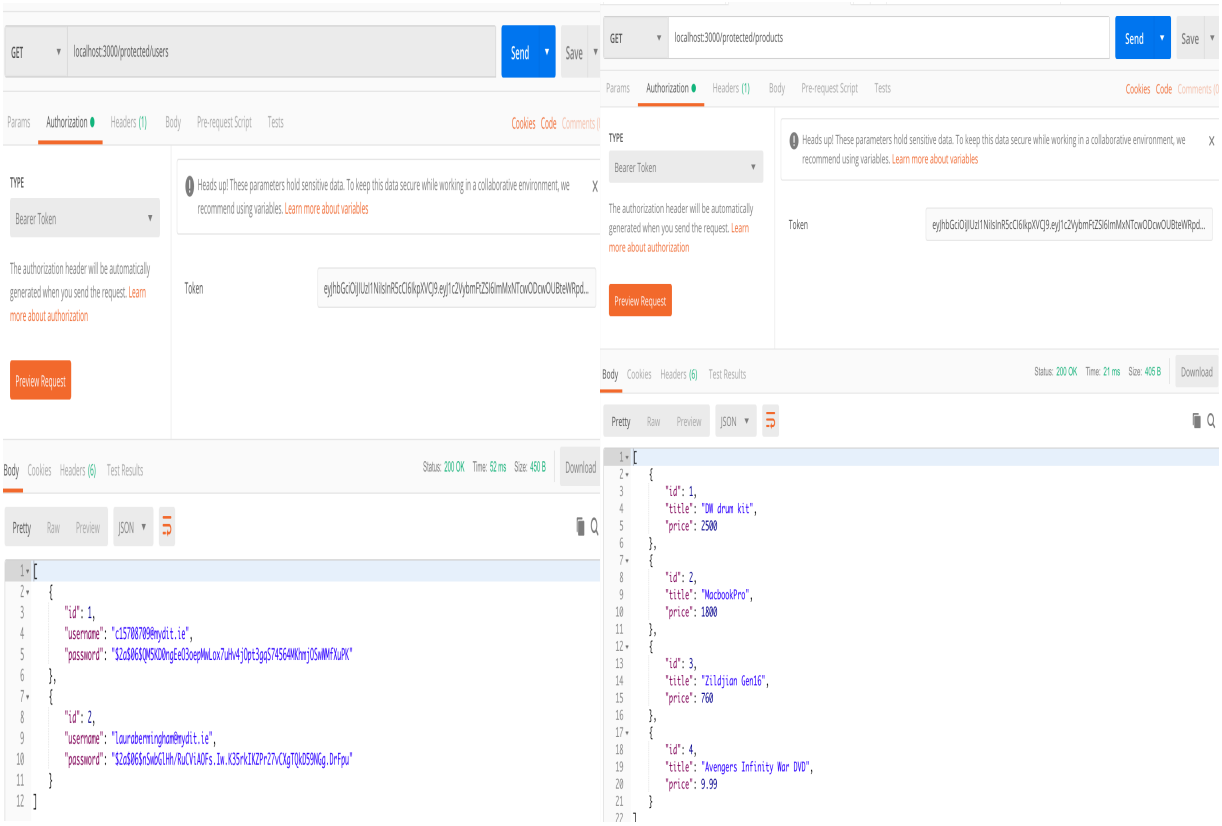
Both require a token, so **/authenticated/login** is required first, then you can call these endpoints using the bearer token to access the resources.

2.3 Endpoints

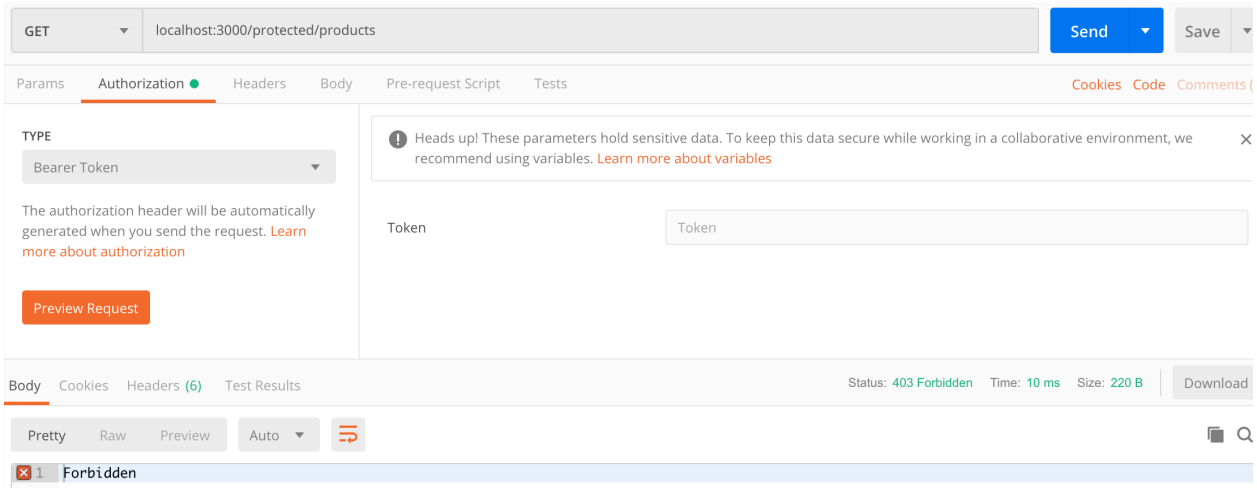
```
// problem set 2.3 using token and verification - return /users and /products
router.get('/protected/users',methods.validateToken, (req, res, next) => {

    Users.findAll().then(users => {
        res.send(users);
    }).catch(err => {
        console.log(err)
        res.send(401);
    });
});

router.get('/protected/products',methods.validateToken, (req, res, next) => {

    Products.findAll().then(products => {
        res.send(products);
    }).catch(err => {
        console.log(err)
        res.send(401);
    });
});
```

Note: the use of **methods.validateToken**

Input:



Without the Bearer Token



It is impossible to access the resources without any authentication or token.

- Assume the client has a prior knowledge of the user password

  For this I have tested the above enpoints to ensure a user cannot gain a token without a valide username and password. I have tested each of these endpoints so that if the password is misspelled or the token is tampered with, no token can be given.
  The /authenticated/login POST endpoint takes a username and password to create a token.

  I will also create a secondary endpoint called /authenticated/loginstoredpassword
  To which a username is only posted across and I use some kind of environment variable for a password. Thus resulting in the same implementation of a token.

  In Bash I will use the following command:

  export EADPASSWORD=’password1’

  In the code I can use the following line to get the password from the environment instead of storing our password on Github etc..
  const password_clear = process.env.EADPASSWORD;

2.4 endpoint

```
//problem set 2.4 - using a process.env.PASSWORD;
router.post('/authenticated/loginstoredpassword', (req, res) => {
    console.log('--------in endpoint---------');
    const username = req.body.username;
    const password_clear = process.env.EADPASSWORD;
    console.log('env password:' + process.env.EADPASSWORD)
    authenticate(username, password_clear, res);
});
```

- Use asynchronous crypto in your solution

HS256 (HMAC with SHA-256) is a symmetric algorithm (this is used by default)
RS256 (RSA Signature with SHA-256) is an asymmetric algorithm

So I generated a test public and private key

Next I needed to alter the authenticate function to accept option for async. You must first login to get an async token. Then we can verify and access restricted resources

2.5 Endpoint

```
//problem set 2.5 using asynchronous algorithms : in the jwt.sign  { algorithms:
['RS256'] }
router.post('/authenticated/loginasync', (req, res) => {
    console.log('--------in endpoint---------');
    const username = req.body.username;
    const password_clear = process.env.EADPASSWORD;
    console.log('env password:' + process.env.EADPASSWORD)
    authenticate(username, password_clear, res, async=true);
});

//verify the async token 2.5
router.get("/protectedasync", methods.validateTokenAsync, (req, res, next) => {
    res.send({
        status: 200,
        message: "AsyncToken has been Verified!",
        token: req.headers["authorization"]
    });
   });//end response


//get protected resource with the Async algo
router.get('/protectedasync/products',methods.validateTokenAsync, (req, res, next) =>
{

    Products.findAll().then(products => {
        res.send({
            status: 200,
            products
        });
    }).catch(err => {
        console.log(err)
        res.send(401);
    });
});
```

Function

```
   if(async == true){

                        console.log('------- using ASYNC ALGO---------')

                        // read in the public private keys. NOTE: THESE ARE ONLY TEST
KEYS. They will never be used again

                        const privateKey = fs.readFileSync(path.join(__dirname,
'../private.key.pem'),"utf-8");
```

```
                        var payload = {username:username, id:user.id};
                        var options = {
                            issuer: "ericstrong",
                            audience: "ericstrong@eric.com",
                            subject: "some website",
                            expiresIn: '1h',
                            algorithm : "RS256"
                        };
                        token = jwt.sign(payload, privateKey, options)
```

Verify function

```
module.exports.validateTokenAsync = function (req, res, next) {
    var bearerHeader = req.headers["authorization"]
    var publicKey = fs.readFileSync(path.join(__dirname, '../public.key.pem'),"utf-
8");
    if (typeof bearerHeader !== 'undefined') {
        const bearer = bearerHeader.split(" ")
        const bearerToken = bearer[1]

        jwt.verify(bearerToken, publicKey, (err, result) => {
                if (err) {
                    console.log(err);
                    res.sendStatus(403)
                } else {
                    next(); // we are now verified, so we can use other or next
methods
                }
            })
        }else{
            console.log('error');
                res.sendStatus(403)
            }
```

Problem Set 3

- Extend the users table to include an access key (160 bits) and secret key (320 bits)

```
DROP TABLE USERS;

CREATE TABLE users(
        id SERIAL PRIMARY KEY,
        username TEXT NOT NULL UNIQUE,
        password TEXT NOT NULL UNIQUE,
        accesskey TEXT NOT NULL UNIQUE,
        secretkey TEXT NOT NULL UNIQUE
);

--sample rows

INSERT INTO users (username,password,accesskey,secretkey)VALUES(
        'c15708709@mydit.ie',
        crypt('password1', gen_salt('bf')) ,
        '44CF9590006BF252F707',
        'OtxrzxIsfpFjA7SwPzILwy8Bw21TLhquhboDYROV'

);

INSERT INTO users (username,password,accesskey,secretkey) VALUES (
        'laurabermingham@mydit.ie',
        crypt('password2', gen_salt('bf')),
        'AK7AEGX2CB5ZL5VQ3UZ7'
        'h8gFhXWmTO4ySWRSlMIyzQ/008/ye9PsQCCVSFJY'

);
```

```
lab2=# table users;
 id |        username         |                          password                          |     accesskey      |               secretkey
----+-------------------------+------------------------------------------------------------+--------------------+----------------------------------------
  1 | c15708709@mydit.ie      | $2a$06$ZXN9IK4hJPH8DtKU56MgGOLpAp/eOspcE5zIMKfjRX5gQ9mrl5s6u | 44CF9590006BF252F707 | OtxrzxIsfpFjA7SwPzILwy8Bw21TLhquhboDYROV
  2 | laurabermingham@mydit.ie | $2a$06$HY1CfQN7eN7Ne4/TplsHfOO51oMQ8sO5mQ330oQyVsOBgirIl2onK | AK7AEGX2CB5ZL5VQ3UZ7 | h8gFhXWmTO4ySWRSlMIyzQ/008/ye9PsQCCVSFJY
(2 rows)
```

access key (160 bits) : example AK7AEGX2CB5ZL5VQ3UGF
secret key (320 bits): example h8gFhXWmTO4ySWRSlMIyzQ/008/ye9PsQCCVSFJX

Problem Set 4

- Implement a Hash-based message authentication (HMAC) scheme to secure the API. In your solution you should include the following API message contents as part of the hashed/signed component:

  - Message body (if any)
  - Access key (prepended or appended as you choose)
  - Query parameters (if any)

  Demonstrate your HMAC authentication on a protected resource
  If authenticated, the API return code should be in the 2xx range, otherwise 401.

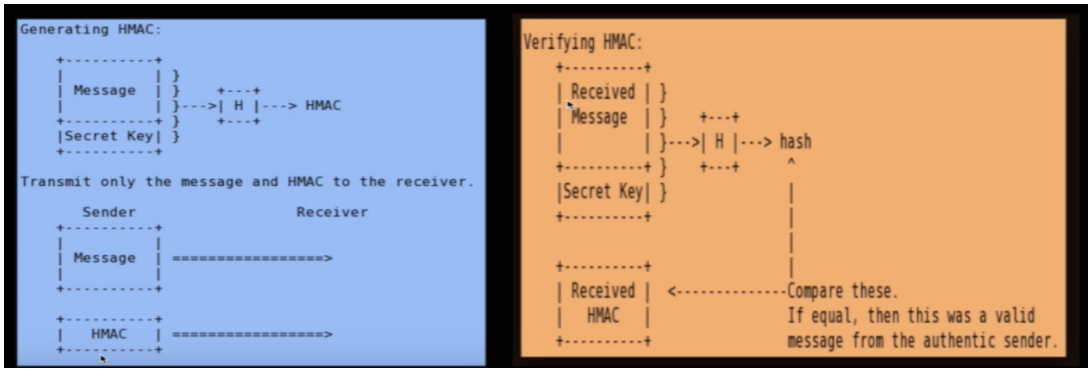
  Note that to test hash-based authentication, you will need to create a simple client capable of generating valid signed requests

Npm install ofuda --save

Before we get started, it is important to understand why we use HMAC. This is a cryptographic technique to ensure our message has not been tampered with in transmission. HMAC will verify the integrity and authenticity of a message. As a prerequisite we need to have a secret key and access key in order to successfully sign our message with. This shared secret key is something both parties A and B will know. My HMAC implementation follows the following guidelines:

- Secret Key - client and server know this.
- A client generates a signature with hmac(access key,secret key)
- when the client wants a protected resource, it sends the signature along with the request
- The server, knows the secret key(retrieved from the postgres database)
- server generates the signature again using this secret key
- and compares with the one sent by the client.
- if they match, it is assumed the user is using the same secret key and is therefore authentic

client server



The process

The Client, (knowing the secret key) creates a signature using the hmac hashing algorithm. The parameters are access key. It is hashed using the secret key.

The payload is sent across to the server. The client wants to access the products protected resource. In order to access this, they need to be authenticated as a valid user. So on the server we receive the request and use the validateHMAC function to validate the credentials. This strips off the payload and verifies the known secret key against what the user had signed the access key with. If this access key matches, what was sent with by the client the user is granted access to the API to return the protected resource of products.

My first step to this was to setup some kind of client that would generate a hmac signature

Client

```
var http = require('http');
const express = require('express');

var Ofuda = require('ofuda');
var hmac = new Ofuda({
    headerPrefix: 'hmac',
    hash: 'sha1',
    serviceLabel: 'HMAC',
    debug: true
}); // can change the serviceLabel to HMAC

// Known credentials by the user - these can be stored in an environment variable also
var credentials = {
    accessKeyId: 'AK7AEGX2CB5ZL5VQ3UZ7',
    accessKeySecret: 'h8IFhXWnRO3ySWRSlMIzzQW008Wue9PsQCCVSFJx'
}; // these are known to the client


//request header info
http_options = {
    host: 'localhost',
    port: 3000,
    path: '/hmac/products',
    method: 'GET',
    params: {
        id: 1
    },
    headers: {
        'Content-Type': 'application/json',
        'Content-MD5': 'ee930827ccb58cd846ca31af5faa3634', //not sure bout this
        'accessid': credentials.accessKeyId,
        'id': 1
```

```
    }
};

console.log('------------ hmac creating client method ----------')
signedOptions = hmac.signHttpRequest(credentials, http_options); // appends a hmac
authorisation header to the request

//request is sent from the client with the credentials signed by HMAC
var req = http.request(signedOptions, function (res) {
    //this will print out the response
    console.log('STATUS: ' + res.statusCode);
    console.log('HEADERS: ' + JSON.stringify(res.headers));
    res.setEncoding('utf8');
    res.on('data', function (chunk) {
        console.log('BODY: ' + chunk);
    });
})

req.end();
```

the client sends a request to **/hmac/products** with **params id = 1**

4.1 endpoint

```
// 4.1 protected resource, using HMAC
 router.get('/hmac/products',methods.validateHMAC, (req, res, next) => {
    console.log('------------ HMAC SUCCESSFUL!!!----------------');

    if(!req.headers.id){
        console.log('---no params-----')
        Products.findAll().then(products => {
            res.writeHead(200);
            res.write(
                JSON.stringify({products: products})
            )
            res.write(
                JSON.stringify({accesskey:req.headers.accessid})
            );

            res.send();

        }).catch(err => {
            console.log(err)
            res.send(401);
        });
```

```
    }else{
        console.log('---got params-----')
        var id = req.headers.id; // name is the actual variable name

        // search for attributes
        Products.findOne({
            where: {
                id: id
            }
        }).then(products => {

            //append accessid
            res.append('accessid',req.headers.accessid );
            res.writeHead(200);
            res.write(
                //message body
                JSON.stringify({products: products}),

            )
            res.write(
                //query params
                JSON.stringify({params: "?id="+req.headers.id })
            );
            res.send();
        })
    }
});
```

Next I needed to verify the HMAC to ensure the authenticity of the user. This is hardcoded for this implementation, however the access key and secret key will be pulled from the postgres database to ensure the authentic user.

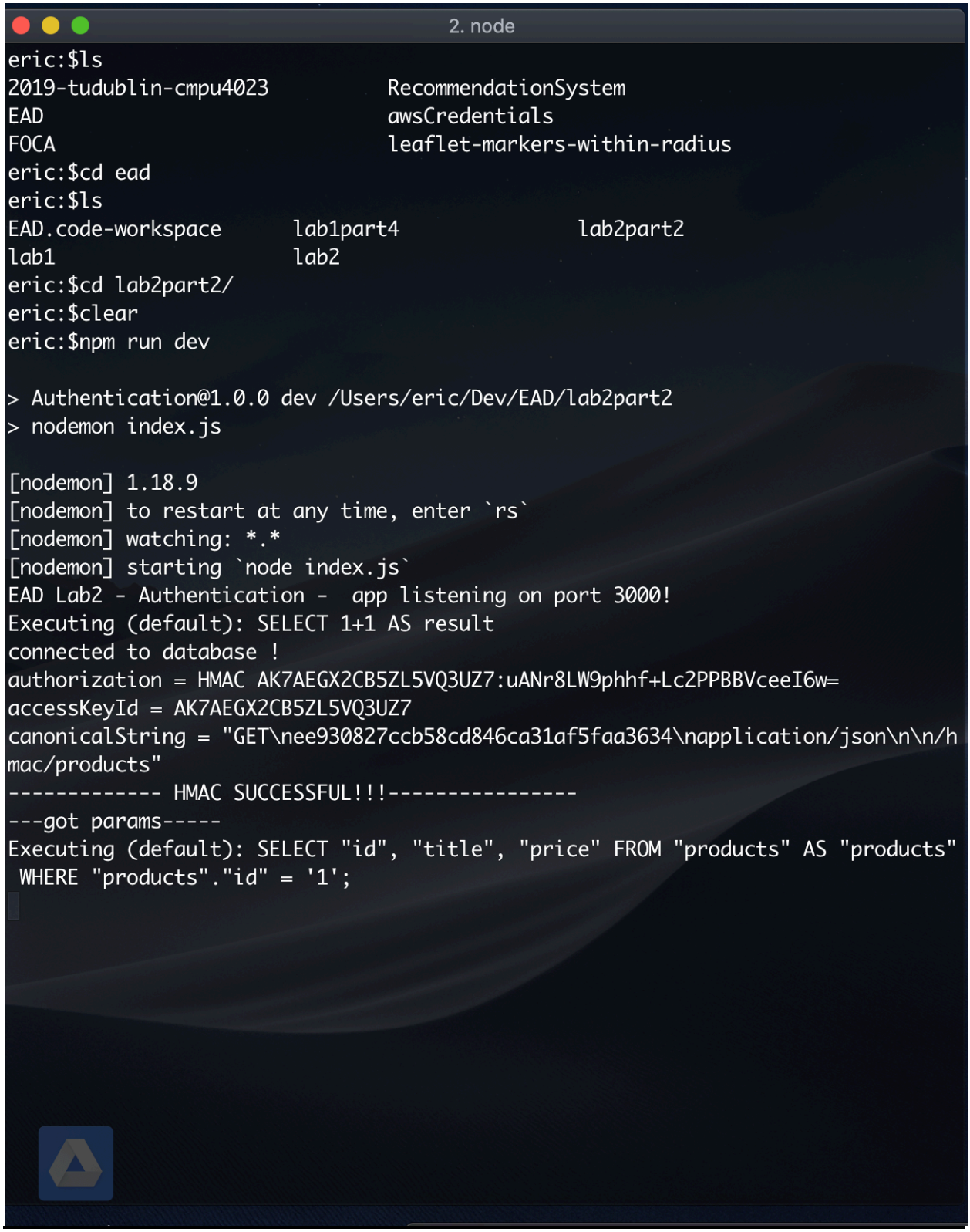
```
//validate HMAC
var validateCredentials = function(requestAccessKeyId){

    //get secret key credentials from database for user


    /**
     * OtxrzxIsfpFjA7SwPzILwy8Bw21TLhquhboDYROV
     *
     * h8IFhXWnRO3ySWRSlMIzzQW008Wue9PsQCCVSFJW
     *
     * EriczxIsfpFjA7SwPzILwy8Bw21TLhquhboDYROx
     */
```

```
    return {accessKeyId: requestAccessKeyId, accessKeySecret:
'h8IFhXWnRO3ySWRSlMIzzQW008Wue9PsQCCVSFJW'};
}

module.exports.validateHMAC = function (request, response, next) {
    if(hmac.validateHttpRequest(request, validateCredentials)){
        next();
    } else {
        console.log('Error with authentication, credentials do not match');
        response.writeHead(401);
        response.end('Invalid Credentials!');
    }

}//end validateHMAC
```

Screenshots

1. **happypath**
   Client requests server with a **valid secret key**

```
eric:$npm run clienthmac

> Authentication@1.0.0 clienthmac /Users/eric/Dev/EAD/lab2part2
> nodemon client.js

[nodemon] 1.18.9
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node client.js`
----------- signature -----------
009a8a36d7729a72029e0cdacf557cb8a682bd02
------------- hmac method ----------
accessKeyId = AK7AEGX2CB5ZL5VQ3UZ7
canonicalString = "GET\nee930827ccb58cd846ca31af5faa3634\napplication/json\n\n/h
mac/products"
{ host: 'localhost',
  port: 3000,
  path: '/hmac/products',
  method: 'GET',
  params: { id: 1 },
  headers:
   { 'Content-Type': 'application/json',
     'Content-MD5': 'ee930827ccb58cd846ca31af5faa3634',
     accessid: 'AK7AEGX2CB5ZL5VQ3UZ7',
     id: 1,
     Authorization: 'HMAC AK7AEGX2CB5ZL5VQ3UZ7:uANr8LW9phhf+Lc2PPBBVceeI6w=' } }
STATUS: 200
HEADERS: {"x-powered-by":"Express","accessid":"AK7AEGX2CB5ZL5VQ3UZ7","date":"Wed
, 20 Feb 2019 13:49:41 GMT","connection":"close","transfer-encoding":"chunked"}
BODY: {"products":{"id":1,"title":"DW drum kit","price":2500}}
BODY: {"params":"?id=1"}
[nodemon] clean exit - waiting for changes before restart
```


2. node
eric:$ls
2019-tudublin-cmpu4023 RecommendationSystem
EAD awsCredentials
FOCA leaflet-markers-within-radius
eric:$cd ead
eric:$ls
EAD.code-workspace lab1part4 lab2part2
lab1 lab2
eric:$cd lab2part2/
eric:$clear
eric:$npm run dev
> Authentication@1.0.0 dev /Users/eric/Dev/EAD/lab2part2
> nodemon index.js
[nodemon] 1.18.9
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
EAD Lab2 - Authentication - app listening on port 3000!
Executing (default): SELECT 1+1 AS result
connected to database !
authorization = HMAC AK7AEGX2CB5ZL5VQ3UZ7:uANr8LW9phhf+Lc2PPBBVceeI6w=
accessKeyId = AK7AEGX2CB5ZL5VQ3UZ7
canonicalString = "GET\nee930827ccb58cd846ca31af5faa3634\napplication/json\n\n/hmac/products"
---------- HMAC SUCCESSFUL!!!----------
---got params-----
Executing (default): SELECT "id", "title", "price" FROM "products" AS "products" WHERE "products"."id" = '1';

2. **With wrong secret key (I simply change one character in the secret key)**

```
var credentials = {
    accessKeyId: 'AK7AEGX2CB5ZL5VQ3UZ7',
    accessKeySecret: 'h8IFhXWnRO3ySWRSlMIzzQW008Wue9PsQCCVSFJx'
}; // these are known to the client
```

Client is rejected the protected resources

```
eric:$npm run clienthmac

> Authentication@1.0.0 clienthmac /Users/eric/Dev/EAD/lab2part2
> nodemon client.js

[nodemon] 1.18.9
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node client.js`
---------- signature ----------
ef1e70f00d382373935ca5d4ab40122f9a843944
------------ hmac method ----------
accessKeyId = AK7AEGX2CB5ZL5VQ3UZ7
canonicalString = "GET\nee930827ccb58cd846ca31af5faa3634\napplication/json\n\n/h
mac/products"
{ host: 'localhost',
  port: 3000,
  path: '/hmac/products',
  method: 'GET',
  params: { id: 1 },
  headers:
   { 'Content-Type': 'application/json',
     'Content-MD5': 'ee930827ccb58cd846ca31af5faa3634',
     accessid: 'AK7AEGX2CB5ZL5VQ3UZ7',
     id: 1,
     Authorization: 'HMAC AK7AEGX2CB5ZL5VQ3UZ7:2RbY2e8QE/B+ZKjyFKrRfmBLw74=' } }
STATUS: 401
HEADERS: {"x-powered-by":"Express","date":"Wed, 20 Feb 2019 13:53:16 GMT","conne
ction":"close","transfer-encoding":"chunked"}
BODY: Invalid Credentials!
[nodemon] clean exit - waiting for changes before restart
```

Server rejects the request

```
eric:$npm run dev

> Authentication@1.0.0 dev /Users/eric/Dev/EAD/lab2part2
> nodemon index.js

[nodemon] 1.18.9
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
EAD Lab2 - Authentication -  app listening on port 3000!
Executing (default): SELECT 1+1 AS result
connected to database !
authorization = HMAC AK7AEGX2CB5ZL5VQ3UZ7:2RbY2e8QE/B+ZKjyFKrRfmBLw74=
accessKeyId = AK7AEGX2CB5ZL5VQ3UZ7
canonicalString = "GET\nee930827ccb58cd846ca31af5faa3634\napplication/json\n\n/h
mac/products"
Error with authentication, credentials do not match
```