**Lab 2**

***Project Structure:***

* index.js (Entry point to application)
* routes.js (Define all routes for api and links them to controllers)
* server.js (Setup file for server)
* config.js (Contains all config variables needed)
* keys
  + lab2.key (Private key for JWT)
  + lab2.key.pub (Public key for JWT)
* gaurds
  + basicAuth.js (Middleware for basic auth routes e.g.: ../basic/..)
  + hmacAuth.js (Middleware for hmac routes e.g.: ../hmac/..)
  + jwtAuth.js (Middleware for jwt routes e.g.: ../jwt/..)
* helpers
  + requestHelper.js (Helps to get details from request)
  + responseHelper.js (Helps to format responses)
* controllers
  + baseController.js (Logic for all base routes e.g.: ../)
  + userController.js (Logic for all user routes e.g.: ../../users)
  + productController.js (Logic for all product routes e.g.: ../../products)
  + authController.js (Logic for all auth routes e.g.: ../auth/)

***Example Routes:***

<http://localhost:3000>

<http://localhost:3000/basic/products>

http://localhost:3000/basic/users

<http://localhost:3000/auth/jwt/login>

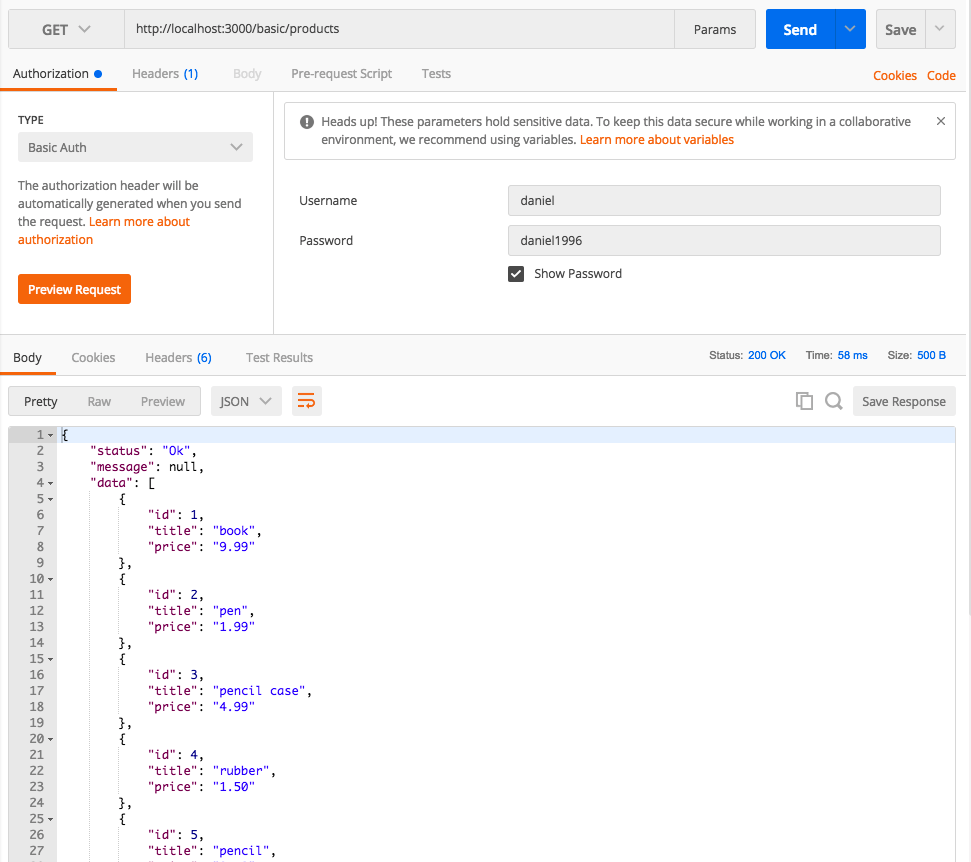
<http://localhost:3000/jwt/products>

http://localhost:3000/jwt/users

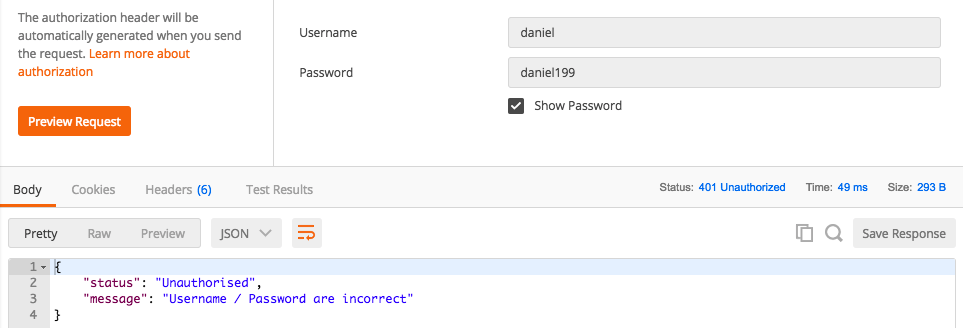
http://localhost:3000/hmac/products

http://localhost:3000/hmac/users

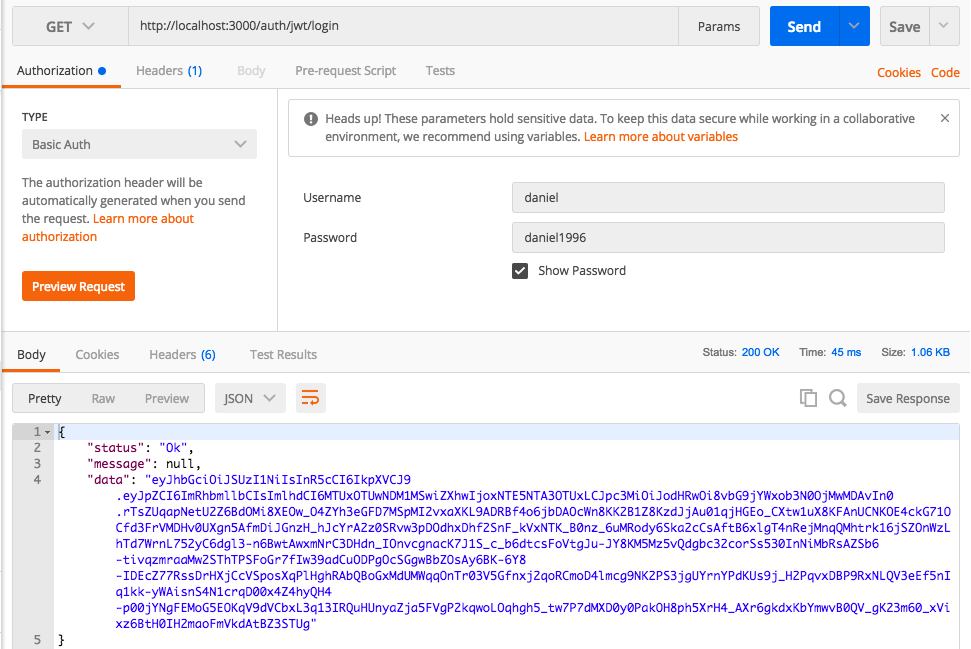
***Question 1:***

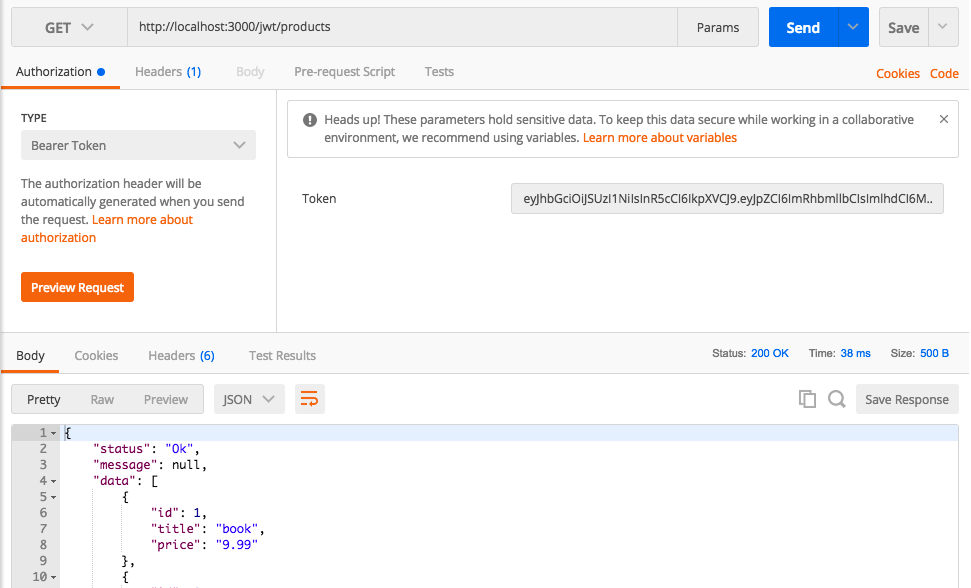
**

Username and password are passed as basic auth headers. If correct, desired resources are returned, otherwise an error message is returned.

******

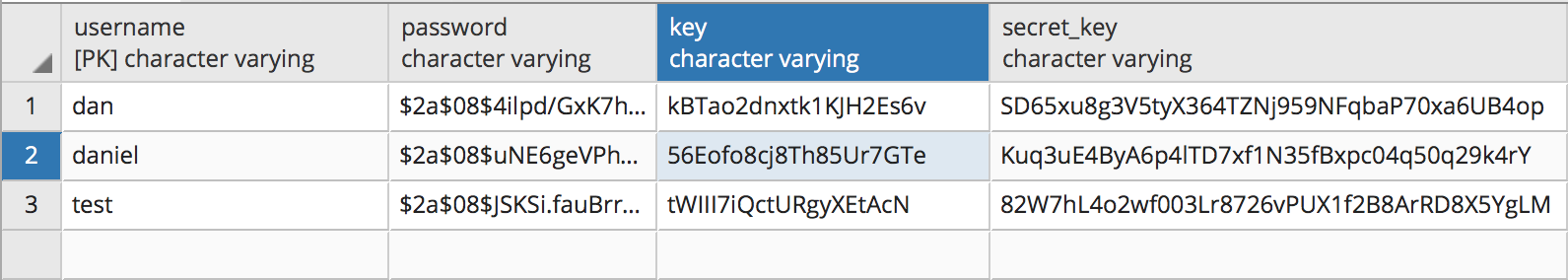
***Question 2:***

**

**

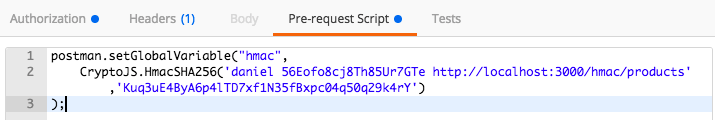
Basic auth passed to login route. If correct, token string is returned, otherwise an error message is returned (similar to question 1). Token string is then used as bearer auth header in second request. If token is valid, resource is returned, else an error is returned.

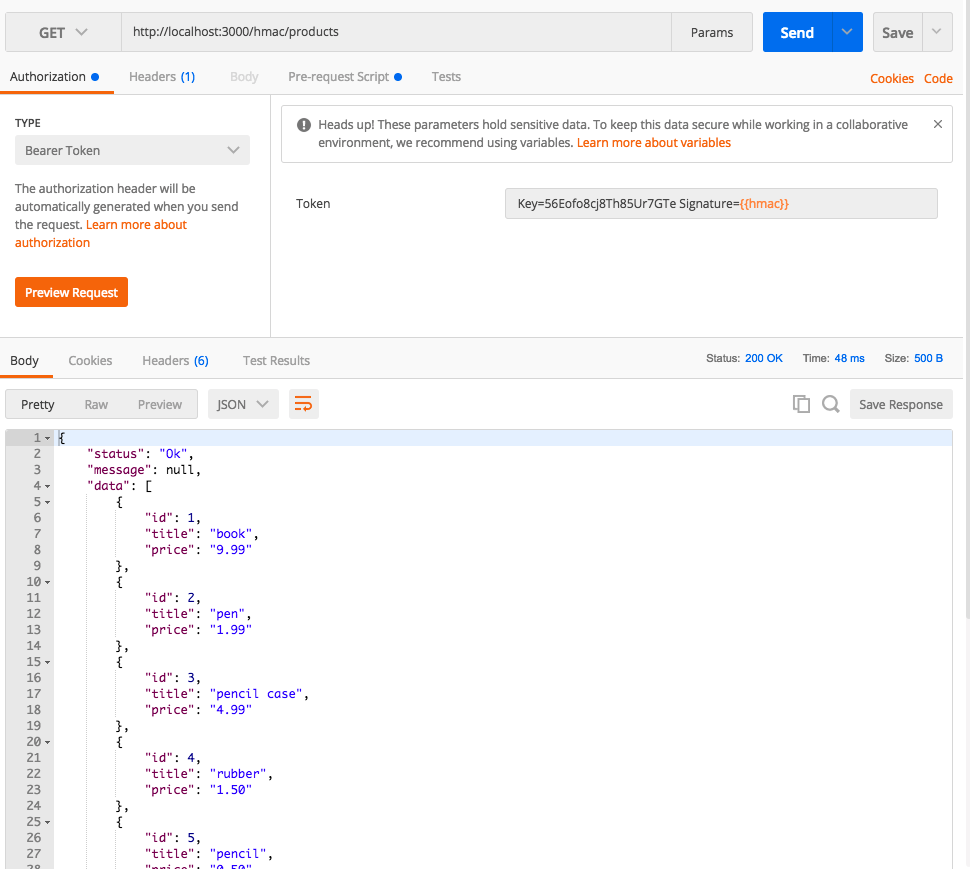
***Question 3:***

**

Key column (size of 20 or 160 bits) and Secret\_Key column (size 40 or 320 bits) added to the pgsql database. Each column contains a unique value for each user. Both the key and secret\_key are different for each user and are randomly generated.

***Question 4:***

******

******

Using Postmans build in pre-request scripts we can create a HMAC signature using crypto-js. The following script is used:

postman.setGlobalVariable("hmac",

CryptoJS.HmacSHA256('

daniel

56Eofo8cj8Th85Ur7GTe

http://localhost:3000/hmac/CURRENT\_ROUTE',

'Kuq3uE4ByA6p4lTD7xf1N35fBxpc04q50q29k4rY')

);

This script uses the username, key and current route as the data to create the signature which is signed using the secret key relative to the user. This signature can then be accessed in the Bearer token field using {{hmac}} (see above image). The bearer token contains a token in the format:

Key=’USER\_KEY’ Signature=’HMAC\_SIGNATURE’

This token is provided with the request and the server will take the key and validate if the user and their secret key exists in the db. If they do, it will create the same signature on the server and then will compare the two signatures to see if they match. If they do, the resource is returned, otherwise an error is returned.