COM661 Full Stack Strategies and Development

BE08. Authentication and User Management

Aims

- To introduce Basic Authentication and JSON Web Tokens
- To introduce the `pyjwt` package
- To demonstrate the concept of decorators in Python
- To implement a Python decorator that protects functions from unauthorised use
- To demonstrate the presentation of JWT tokens to the request header.
- To present the implementation of a set of user accounts
- To introduce the Python `bcrypt` package and `bytes` object
- To implement a multi-level access scheme

Table of Contents

8.1 JSON Web Tokens

So far, we have developed an API that allows users to view, add, edit and delete information about a collection of businesses and reviews on those businesses. All information is stored in a back-end MongoDB database, so that changes made by a user in one session are maintained for all users in later sessions. The API is currently fully-functional, but in some respects, it is too functional – anyone can perform any operation regardless of their status and even without identifying themselves.

In this practical, we will see how to protect certain operations so that they are only available to logged-in users who have permission to perform those operations. We will use a combination of techniques known as Basic Authentication and JSON Web Tokens (JWT) to implement this.

8.1.1 Basic Authentication and Preparation for JWT

Basic Authentication is where we require the user to provide some specific piece of identification information before allowing them to access a resource. Normally, this is a username and some secret such as a password. JWT is a scheme by which a unique identification string (token) is created which can then be passed by the client along with each request to be made. The token can contain any information that is required by the application (usually a username and other information such as that user's status/permissions/etc.) and is decoded and checked by the server before the requested operation is fulfilled.

Before we can make use of JWT in our Python applications, we need to install the **`pyjwt`** library that provides the set of methods to create and manipulate tokens. The easiest way to do this is by using the **pip** package manager as follows:

```
P:\biz> pip install pyjwt
```

Once **`pyjwt`** is installed, we can prepare our application by importing the additional elements that we will use in this practical (we will explain each as we meet it) and create the secret key that will be used to encrypt our data and generate the JWT token.

Note:	**`mysecret`** is obviously not the most secure of choices for a secret key value but has been chosen here for illustration. Feel free to choose any text string of your choice.

File: app.py

```
from flask import Flask, request, jsonify, make_response
from pymongo import MongoClient
from bson import ObjectId
import jwt
import datetime
from functools import wraps

app = Flask(__name__)

app.config['SECRET_KEY'] = 'mysecret'

...
```

Do it now!	Make the additions specified above to your copy of ***app.py*** in your ***Biz Directory*** project

8.1.2 Providing a Login Route

To generate a JWT token, we need to provide a login route. This enables users to identify themselves to the application before proceeding to make use of its functionality. The **`login()`** method first attempts to retrieve any existing token by the ***Flask*** **`request.authorization`** method. In a browser, this generates a Basic Authorization pop-up which prompts the user to enter values for a username and password.

In this initial example, we will consider the user to be logged in if the value for password is equal to "password", so if this is the case, we create a new token using the **`jwt.encode()`** method, setting values for '***user***' (the value entered as the username) and '***exp***' (the date on which the token expires, set as 30 minutes in the future). The second parameter to **`jwt_encode()`** is the string used to encode the token, so we pass the secret key value defined earlier. Finally, we return the token in a JSON object. If the username and password provided were invalid, we generate a response that causes the browser to re-prompt the user for the username and password.

The code for the initial version of the **`login()`** function is defined below.

File: app.py

```
...

@app.route('/api/v1.0/login', methods=['GET'])
def login():
    auth = request.authorization
    if auth and auth.password == 'password':
        token = jwt.encode( {
            'user' : auth.username,
            'exp' : datetime.datetime.now(datetime.UTC) +
                    datetime.timedelta(minutes=30) },
            app.config['SECRET_KEY'],
            algorithm="HS256")
        return make_response(jsonify({'token' : token}), 200)
    return make_response('Could not verify', 401, \
           {'WWW-Authenticate' : \
            'Basic realm = "Login required"'})

if __name__ == "__main__":
      app.run(debug=True)
```

Do it now!	Add the login route presented above to the application and test it by visiting http://localhost:5000/api/v1.0/login in a web browser. Check that the pop-up window shown in Figure 8.1 is generated and that when any username and the password "password" are provided, a token is generated and displayed as shown in Figure 8.2.

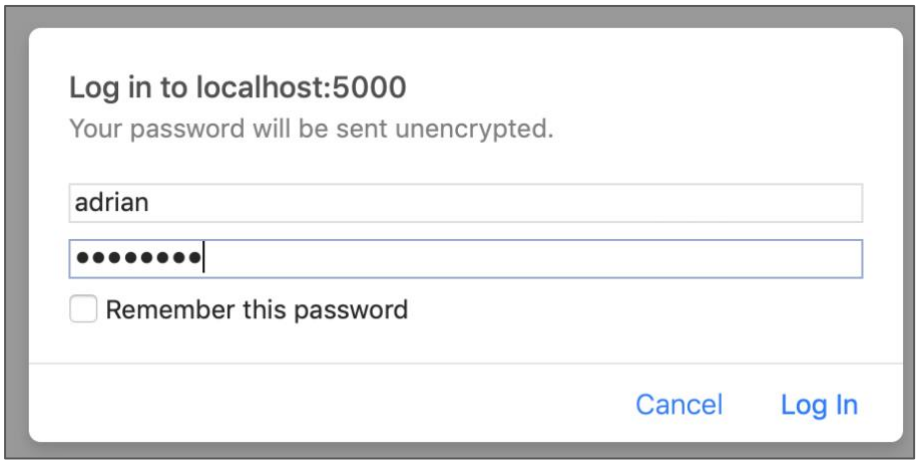

Figure 8.1 Username and Password Prompt

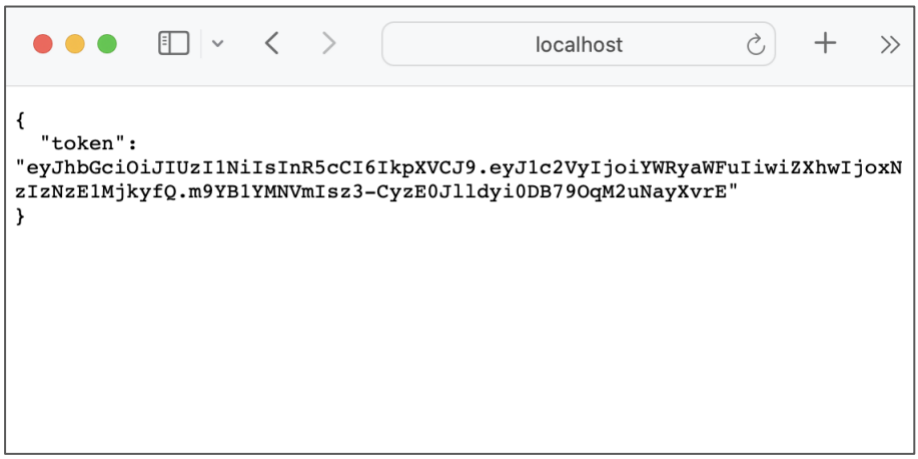

Figure 8.2 Token Generated

In order to better understand the structure of the token, it is useful to use the online tool at http://jwt.io to analyse it. If you visit http://jwt.io and scroll past the initial text, you will see the tool as shown in Figure 8.3. Paste your newly generated token into the pane on the left-hand side of the window and provide your secret key value in the area highlighted in Figure B4.3 below. You should be able to see that the token is verified as valid and that the token **payload** contains an encrypted representation of the '***user***' and '***exp***' values defined in the **`login()`** function.

Note:	The JSON object specified as the payload can contain any values that we deem appropriate or useful for the application. This might include the user's status (i.e. 'admin' or not), their full name, their **`_id`** value to be used in building personalized URLs, or any other information.

Do it now!	Follow the steps illustrated above to generate a JWT token and verify it in the online tool at http://jwt.io.

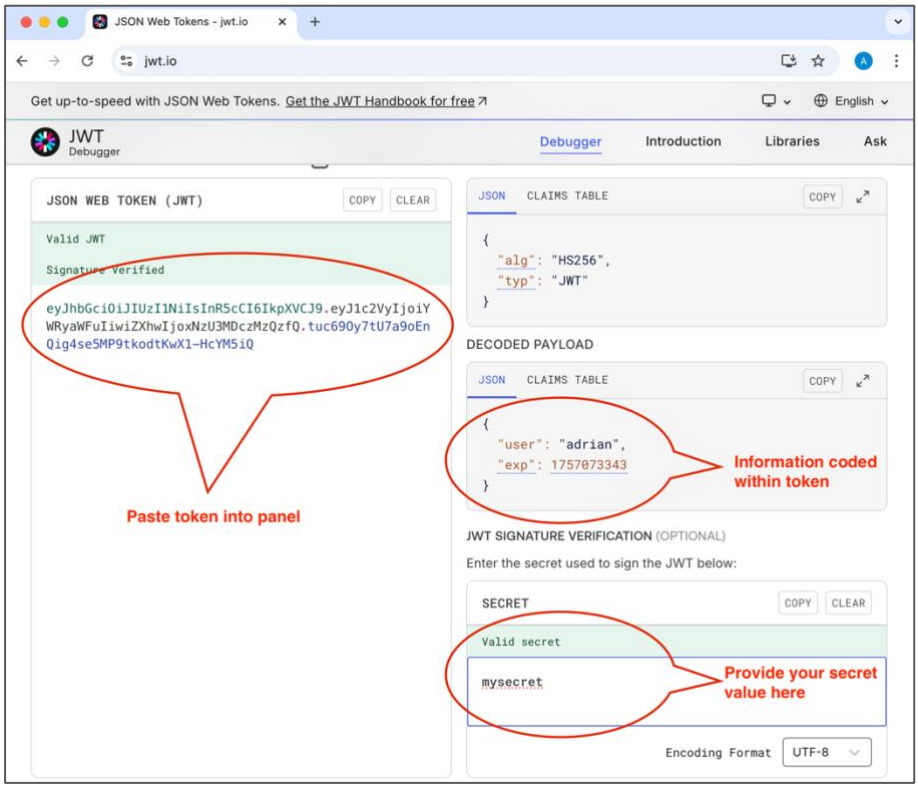

Figure 8.3 Analysing the Token

8.2 Python Decorators

Now that the token is created, verifying that the user is logged in, we can use it to determine whether they should be allowed to access any of the functionality. We could do this by adding code to each of the routes, but we quickly realise that we need to add the same check to any route for which the token is required. A much better approach is to build a Python **decorator** - a function that can amend the behavior of another function, but without changing the internal code of that function. Decorators are defined once but can be applied to multiple other functions – allowing us to write a single code description that can be used wherever we need it.

8.2.1 Building a First Decorator

To demonstrate, we will use the Python Command prompt to demonstrate applying decoration to a function. First, we enter the Python command prompt and define a simple function.

```
Python 3.12.2 (v3.12.2:6abddd9f6a, Feb  6 2024, 17:02:06) [Clang 13.0.0
(clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def say_hello():
...     print("Hello")
...
>>>
```

Figure 8.4 Define a Simple Function

Now we can ask the Command prompt to report the value of the function, before running it. Note that stating the name of the function (without parentheses) returns confirmation that it is indeed a function, while specifying the function with parentheses causes the function to be called and executed and its result displayed.

```
(venv) adrianmoore@Adrians-iMac biz % python3
Python 3.12.2 (v3.12.2:6abddd9f6a, Feb  6 2024, 17:02:06) [Clang 13.0.0
(clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def say_hello():
...     print("Hello")
...
>>> say_hello
<function say_hello at 0x104fb7920>
>>> say_hello()
Hello
>>>
```

Figure 8.5 Inspect and Run the Function

Now that the function is in place, we create a simple decorator that defines some additional operations that we would like to add to selected functions

A decorator is a function (in this case **`my_decorator()`**) that accepts a function (defined as **`func`**) as a parameter. The decorator function contains an internal function (here called **`wrapper()`**) that describes the environment in which **`func()`** will operate. In this case, we specify that a **`print()`** command generates some output, before the original **`func()`** is run and a second **`print()`** generates more output. Finally, the output of the wrapper is returned to where the decorator was called.

In effect, the function passed to the decorator will be run (the call to **`func()`**) but only after and before **`print()`** statements generate additional output. The code for the decorator can be seen in Figure 8.6 below.

```
(venv) adrianmoore@Adrians-iMac biz % python3
Python 3.12.2 (v3.12.2:6abddd9f6a, Feb  6 2024, 17:02:06) [Clang 13.0.0
(clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def say_hello():
...     print("Hello")
...
>>> say_hello
<function say_hello at 0x104fb7920>
>>> say_hello()
Hello
>>> def my_decorator(func):
...     def wrapper():
...             print("Before the function")
...             func()
...             print("After the function")
...     return wrapper
...
>>>
```

Figure 8.6 Define a Decorator

Finally, for now, we specify the decoration by re-defining **`say_hello`** to be a call to the decorator, with itself as the parameter. Viewing the updated value of **`say_hello`** reveals that it is now effectively a reference to the wrapper function defined within the decorator.

We can now run **`say_hello()`** to see the effect we have created.

```
(venv) adrianmoore@Adrians-iMac biz % python3
Python 3.12.2 (v3.12.2:6abddd9f6a, Feb  6 2024, 17:02:06) [Clang 13.0.0
(clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def say_hello():
...     print("Hello")
...
>>> say_hello
<function say_hello at 0x104fb7920>
>>> say_hello()
Hello
>>> def my_decorator(func):
...     def wrapper():
...             print("Before the function")
...             func()
...             print("After the function")
...     return wrapper
...
>>> say_hello = my_decorator(say_hello)
>>> say_hello
<function my_decorator.<locals>.wrapper at 0x104fb7a60>
>>> say_hello()
Before the function
Hello
After the function
>>>
```

Figure 8.7 Demonstrating the Effect of the Decorator

Do it now!	Follow the steps outlined above to specify a decorator and apply it to the **`say_hello()`** function. Verify that running **`say_hello()`** calls the decorator by observing the messages that are printed before and after the output from the function.

8.2.2 Using Decorators

The example demonstrated is seen to work well, but the re-definition of the **`say_hello()`** function is a slightly "clunky" technique that will become a burden if we have multiple functions to which we want to apply the same decorator. Fortunately, Python provides a syntax that applies the decorator in a much more readable way.

Examine the code for ***decorators_1.py*** (provided in the ***Practical BE08 Files***) below, that repeats the previous experiment, but introduces the `@` syntax to apply a decorator to a function. Note also how we use the **`@wraps`** decorator imported from the **`functools`** library. This preserves the state of the function being wrapped by the decorator and allows us to apply the decorator to multiple functions.

File: decorators_1.py

```
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper():
        print("Before the function")
        func()
        print("After the function")
    return wrapper

@my_decorator
def say_hello():
    print("hello")

say_hello()
```

Do it now!	Add the file ***decorators_1.py*** to your application (copy it into the ***biz*** folder) and run it. Verify that the application still works as described above

There are two other issues around decorators that we need to address before we can apply them to multiple functions.

The first concerns parameters – examine the code for ***decorators_2.py*** (also provided in the ***BE08 Practical Files***) that demonstrates the decorator applied to two functions, **`shout_out()`** and **`whisper_it()`**.

File: decorators_2.py

```
...

@my_decorator
def shout_out():
    print("HELLO")

@my_decorator
def whisper_it():
    print("goodbye")

shout_out()
whisper_it()
```

Note how the decorator is defined exactly as before, but this time it has been applied to both functions. When we run the application, we see that the output of each function is wrapped in the result of the **`print()`** statements, verifying that the decorator has been applied to both.

Do it now!	Run ***decorators_2.py*** and verify that the effect of the decorator can be seen on both functions.

Now, we modify the code so that the strings to be printed in each function are passed as parameters. In addition, we add the **`upper()`** method to the string passed to **`shout_out()`** and add the **`lower()`** method to the string passed to **`whisper_it()`** so that the output from each is in upper and lowercase, respectively.

File: decorators_2.py

```
...

@my_decorator
def shout_out(shout_value):
    print(shout_value.upper())

@my_decorator
def whisper_it(whisper_value):
    print(whisper_value.lower())

shout_out("Hello")
whisper_it("Goodbye")
```

Do it now!	Make the changes shown in the code box above and try to run the application.

This time, running the application generates an error message that the **`wrapper()`** function is not expecting any arguments, but one was provided. We can easily fix this by adding parameters to the definition of **`wrapper()`**, but remember that we want our decorator to be available to multiple functions – regardless of the number of parameters that may be expected.

The solution to this is to specify the generic Python parameter list **`(*args, *kwargs)`** to the **`wrapper()`** function. This is used when we don't know in advance how many arguments will be passed to a function and so is ideal for this situation. We therefore update the **`my_decorator()`** code in ***decorators_2.py*** as shown below.

File: decorators_2.py

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        @wraps(func)
        print("Before the function")
        func(*args, **kwargs)
        print("After the function")
    return wrapper

...
```

Do it now!	Make the changes above and verify that the application runs as expected.

Try it now!	Verify that the wrapper will accept a variable number of parameters by changing the definition and call for **`whisper_it()`** so that it requires two string values to be provided as parameters. The revised version of the function should output both strings, converted to lowercase and separated by a space.

The final modification to the decorator is to deal with values returned from functions. Examine the modification to the **`shout_out()`** function below such that the lowercase string is not **`print`**ed by the function, but instead is returned from it.

File: decorators_2.py

```
...

def shout_out(shout_value):
    return shout_value.upper()

...

print(shout_out("Hello"))

...
```

Do it now!	Make the changes above and try to run the application.

This time, running the application reveals that the value **`None`** is printed as the output from the **`shout_out()`** function. To fix this, we need to explicitly return the value from the **`wrapper()`**, so we make the update shown below.

File: decorators_2.py

```
def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Before the function")
        return_value = func(*args, **kwargs)
        print("After the function")
        return return_value
    return wrapper
...
```

Do it now!	Make the changes above and verify that the application runs as expected. Note that the **`whisper_it()`** function that does not expect a return value still works as before

Now that our decorator is sufficiently flexible to cope with variable parameter lists and correctly handles return values, we can return to our API application and see how to use a decorator to protect selected routes against unauthorized access.

8.3 Protecting Selected Routes

Now that we understand the concept of Python decorators and have seen how a single decorator function can be applied to multiple other functions to change their behavior, we can build a decorator that checks for the presence of a valid JWT token before allowing the user to proceed to the requested operation

8.3.1 Building the Decorator

The structure of the decorator **`jwt_required()`** mirrors that of the **`my_decorator()`** definition from the previous section. The decorator contains an embedded function (here called **`jwt_required_wrapper()`**) that checks for the presence of a token as a query string parameter by examining the **`request.args`** object. If a token is found, it is decoded using the secret key value defined earlier and, only if the **`decode()`** is successful (i.e. the token is valid), does execution proceed to the function that has been decorated. If either the check for a token or the **`decode()`** operation fails, then an appropriate error message is returned and the decorated function is not executed.

File: app.py

```
...

client = MongoClient("mongodb://127.0.0.1:27017")
db = client.bizDB            # select the database
businesses = db.biz          # select the collection name

def jwt_required(func):
    @wraps(func)
    def jwt_required_wrapper(*args, **kwargs):
        token = request.args.get('token')
        if not token:
            return make_response(jsonify(
                  {'message' : 'Token is missing'}), 401)
        try:
            data = jwt.decode(token,
                              app.config['SECRET_KEY'],
                              algorithms="HS256")
        except:
            return make_response(jsonify(
                  {'message' : 'Token is invalid'}), 401)
        return func(*args, **kwargs)

    return jwt_required_wrapper

...
```

8.3.2 Applying the Protection

We can test the effect of the decorator by applying it to routes that we want to protect. As a first test, we apply the decorator to the **`show_one_business()`** function. This should result in the usual output when we attempt to display all businesses, but returns a "Token is missing" message when we add one of the business ID values to the URL in order to display only that business. We apply the decorator to the function by the **`@decorator_name`** syntax as demonstrated earlier.

File: app.py

```
...

@jwt_required
def show_one_business(id):

...
```

Now, when the application is run, the URL http://localhost:5000/api/v1.0/businesses returns the full list of business objects as usual, but when we take one of the business ID values and add it to the URL, we see the "Token is missing" message as illustrated in Figure 8.8 below.

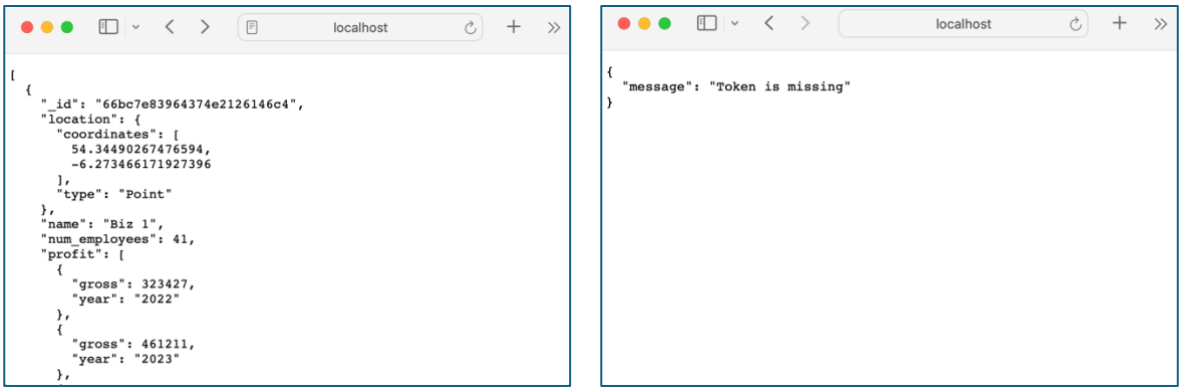

Figure 8.8 Access Allowed and Denied

Do it now!	Make the changes to ***app.py*** described above and make sure that the application behaves as illustrated in Figure 8.8.

To generate the token required to see a single business, we need to visit the ***login*** route and specify a valid password as described in Section 8.1. If we copy the token and add it to the URL as a query string parameter, the decorator will detect the presence of the token, verify

that it is valid and allow the application to proceed to the **`show_one_business()`** function. This situation is illustrated in Figure 8.9, below.

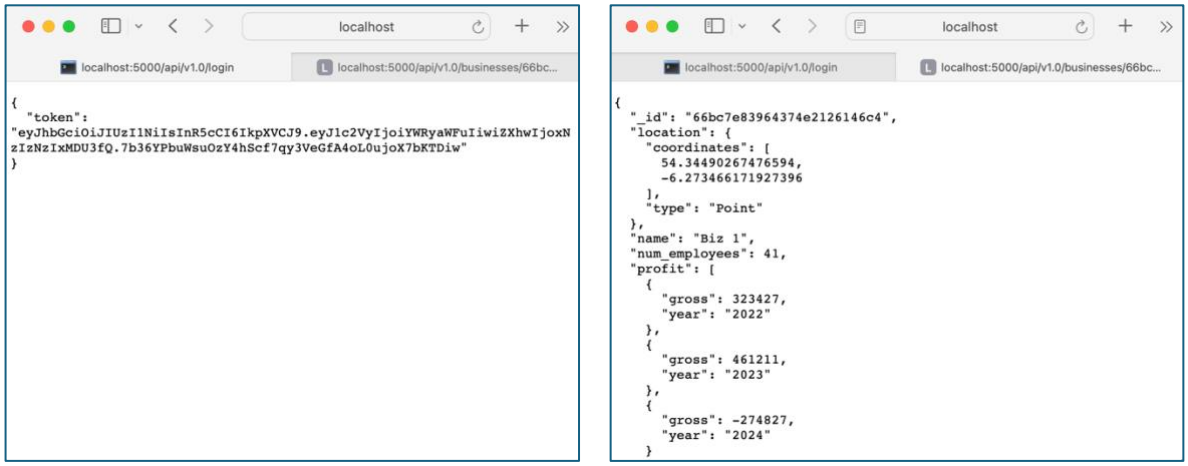

Figure 8.9 Obtaining and Using a Token

Do it now!	Visit http://localhost:5000/api/v1.0/login and present any username and the password value "password". Copy this value and add it as a query string parameter to the URL to view details of a single business (http://localhost:5000/api/v1.0/businesses/<business_id>?token=<token>). Verify that output such as that shown in Figure 8.9 is achieved.

Try it now!	Modify the application so that all GET requests are available to all users (i.e. anyone can view business or reviews without a token), but all POST, PUT and DELETE requests are only available when a valid token is presented.

8.3.3 Passing the Token in the Request Header

Passing the token in the query string has been seen to work well, but it is more convenient (and more secure) to pass it as part of the request header. To achieve this, we need to make a small modification to the decorator to retrieve the token from the request header rather than the query string. Comment out (or delete) the existing line and replace it with the code highlighted below.

File: app.py

```
...

def jwt_required(func):
    @wraps(func)
    def jwt_required_wrapper(*args, **kwargs):
        #token = request.args.get('token') - REMOVE THIS LINE
        token = None
        if 'x-access-token' in request.headers:
            token = request.headers['x-access-token']

...
```

We can test this in Postman by making a GET request to the URL http://localhost:5000/api/v1.0/login, selecting the ***Basic Auth*** type from the ***Authorization*** menu and entering the username and password in the text boxes provided. Clicking the "Send" button passes the values to the server code, where the password is checked, and the token returned as shown in Figure 8.10 below.

Note:	Alternatively, you can generate a new token by using a web browser to visit the URL http://localhost:5000/api/v1.0/login as seen earlier.

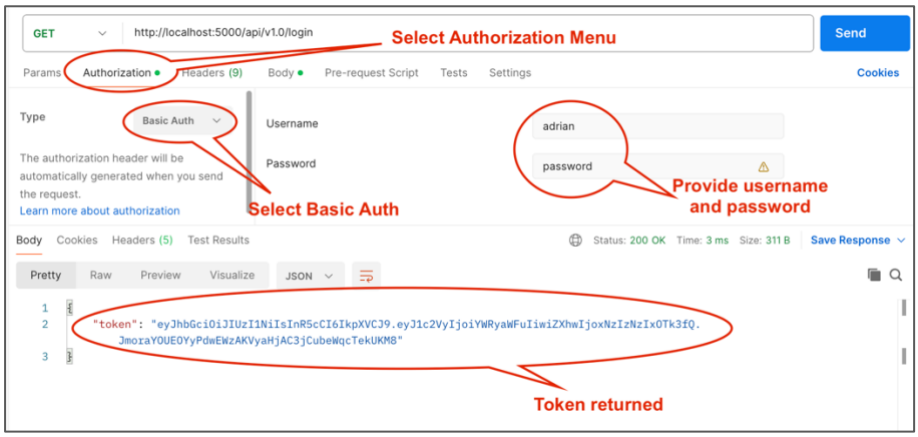

Figure 8.10 Basic Auth in Postman

Now, we will use this token to make a POST request to the reviews collection of a business. First, we make a GET request to http://localhost:5000/api/v1.0/businesses to generate a list of businesses and copy one of the business ID values. Add this value to the URL to set up a POST request to http://localhost:5000/api/v1.0/businesses/<business_id>/reviews and add values for the ***username***, ***text*** and ***stars*** fields of the review in the ***Body*** as usual. Next, click

on the link to the ***Headers*** menu and add a header with name 'x-access-token' and the new token as the value. Clicking 'Send' should result in the new review being accepted, as illustrated in Figure 8.11, below.

Note:	Make sure that you have applied the `@jwt_required` decorator to the `add_new_review()` endpoint as described in the previous ***"Try it now!"*** exercise.

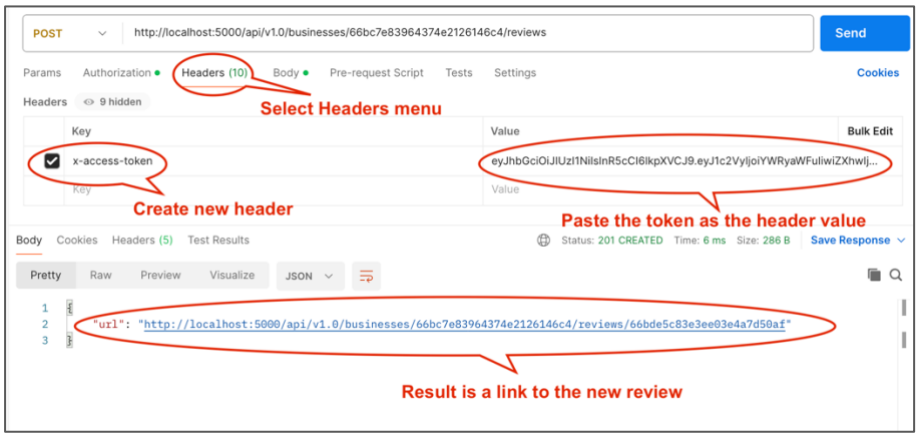

Figure 8.11 Passing the Token with a POST Request

8.4 User Management

In the previous sections, we implemented back-end authentication with JSON Web Tokens and provided a login route that required a user to provide a username and password before access to selected functionality was granted. This implementation, however, used a dummy account which only required a hard-coded password value to be entered and did not distinguish between different users.

In this section, we generate "real" user accounts and see how to manage them. A 3-tiered access scheme is developed with some operations open to all, some only to logged in users and some only available to logged in users with "admin" status.

8.4.1 Creating User Accounts

In this application, we use five fields to describe a staff user as follows:

- The user's real name
- A username to use when logging on to the application
- An encrypted password
- An email address

- The user's administrator status recorded as ***true*** or ***false***.

Examine the code (provided in the ***BE08 Practical Files*** and shown below) for the file ***create_users.py***, that specifies a data structure containing details of five users and generates the collection in the MongoDB database that will be used to store them.

File: create_users.py

```
from pymongo import MongoClient
import bcrypt

client = MongoClient("mongodb://127.0.0.1:27017")
db = client.bizDB
users = db.users

user_list = [
                {
                  "name" : "Homer Simpson",
                  "username" : "homer",
                  "password" : b"homer_s",
                  "email" : "homer@springfield.net",
                  "admin" : False
                },
                ...
              ]

for new_user in user_list:
      new_user["password"] = bcrypt.hashpw(
                        new_user["password"], bcrypt.gensalt())
      users.insert_one(new__user)
```

First, the code connects to the MongoDB server and selects the database and collection with which to work. Note that if a collection that does not currently exist is selected, it will be automatically created once the first attempt is made to insert data to it.

The collection of user information in the list **`data`** provides values for each of the fields but provides the password in plain text. Our specification requires the password to be encoded, so before calling the **`insert_one()`** method, we first use the **`hashpw()`** method of the **`bcrypt`** library to generate a hashed version of the password value. **`hashpw()`** takes two parameters – a ***bytes object*** representing the plaintext password and a ***salt*** which is a random value used in the encryption operation. The salt is created by the **`bcrypt`** library's **`gensalt()`** method. Note that in the specification of each user, the password value is

preceeded by **b** (i.e. **b"homer_s"**). This identifies the password as a ***bytes*** object rather than a string.

Before we can run ***create_users.py*** to build the collection of user accounts, we need to install the **bcrypt** package that provides the **hashpw()** and **gensalt()** methods. Once again, this is most easily done by using the **pip** package manager as follows:

```
(venv) P:\biz> pip install bcrypt
```

Do it now!	Install the **bcrypt** package as shown above and then copy the file create_users.py into your ***biz*** application folder. Make sure that your database server is running and then run the application to generate the collection of users. You should be able to use the MongoDB shell to verify that the collection has been created as shown in Figure 8.12, below.

Note when viewing the database contents that the password values have been encrypted and that one of the users has the "***admin***" field set to ***true***. Also note that each document has a unique **_id** field even though we did not specify one. MongoDB creates this automatically for each new document.

```
bizDB> db.users.find()
[
  {
    _id: ObjectId('66be045bce1158c7cdec58b7'),
    name: 'Homer Simpson',
    username: 'homer',
    password: Binary.createFromBase64('JDJiJDEyJG01VFFuOS9HMVY4T3dBemtXbnV4VnU0Z0RT
V4SXMvNzhT', 0),
    email: 'homer@springfield.net',
    admin: false
  },
  {
    _id: ObjectId('66be045cce1158c7cdec58b8'),
    name: 'Marge Simpson',
    username: 'marge',
    password: Binary.createFromBase64('JDJiJDEyJFFIb1BXWjFHcmxXTE9HYWlVRWVVb2VBZUZK
pkZUZqN3ZX', 0),
    email: 'marge@springfield.net',
    admin: false
  },
  {
    _id: ObjectId('66be045cce1158c7cdec58b9'),
    name: 'Bart Simpson',
    username: 'bart',
    password: Binary.createFromBase64('JDJiJDEyJHpEWnFRWHJrTmhEWnF4ejY0ZEg1VnVPa2Y5
UubWVEbG1T', 0),
    email: 'bart@springfield.net',
    admin: false
  },
  {
    _id: ObjectId('66be045cce1158c7cdec58ba'),
    name: 'Lisa Simpson',
    username: 'lisa',
    password: Binary.createFromBase64('JDJiJDEyJGc1YWZaa1NmYVlJMnVSQnViVnp1M3U5Ym5I
FYMllkZktl', 0),
    email: 'lisa@springfield.net',
    admin: true
  },
  {
    _id: ObjectId('66be045cce1158c7cdec58bb'),
    name: 'Maggie Simpson',
```

Figure 8.12 User Accounts Created

8.4.2 User Login

The previous version of the application implemented a login function that created a JWT token if a specified value was provided as the password. We now need to update this to check the username and password against the values held in the **`users`** collection.

In this version of the **`login()`** function, we first check for the presence of an authorization header. If one is found, we retrieve the ***username*** field from it and perform a database search for a document in the ***users*** collection with a ***username*** value that matches that in the header. If a user is found, we then use the **`bcrypt`** method **`checkpw()`** to compare the password in the authorization header to that retrieved from the database. If there is a match, then a new JWT token is generated, encoding the username and expiry stamp within the payload. Finally, the token is returned as the response to the login request. If any of the conditions fail, then a response indicates the problem that was encountered.

Note the format of the **`checkpw()`** method, which takes two parameters. The first is the plaintext entry, converted to a ***bytes*** object, while the second parameter is the encrypted version.

File: app.py

```
...

import bcrypt

...

users = db.users

...

def login():
    auth = request.authorization

    if auth:
        user = users.find_one( {'username':auth.username } )
        if user is not None:
            if bcrypt.checkpw(bytes(auth.password, 'UTF-8'),
                              user["password"]):
                token = jwt.encode( {
                  'user' : auth.username,
                  'exp' : datetime.datetime.now(datetime.UTC) +
                          datetime.timedelta(minutes=30) },
                  app.config['SECRET_KEY'],
                  algorithm="HS256")
                return make_response(
                   jsonify({'token' : token}), 200)

            else:
                return make_response(jsonify( {
                              'message':'Bad password'}), 401)
        else:
            return make_response(jsonify( {
                              'message':'Bad username'}), 401)

    return make_response(jsonify( {
                  'message':'Authentication required'}), 401)

...
```

Do it now!	Add code to import the **`bcrypt`** package and select the ***users*** collection and then modify the **`login()`** function as shown above. Using Postman, generate a **GET** request to http://localhost:5000/api/v1.0/login and select ***Basic Auth*** from the list of options under the ***Authorization*** tab. Now provide a username and password from those specified in ***create_users.py*** and click "Send". Check that the request is accepted and that you receive a JWT token as output, as shown in Figure 8.13, below.

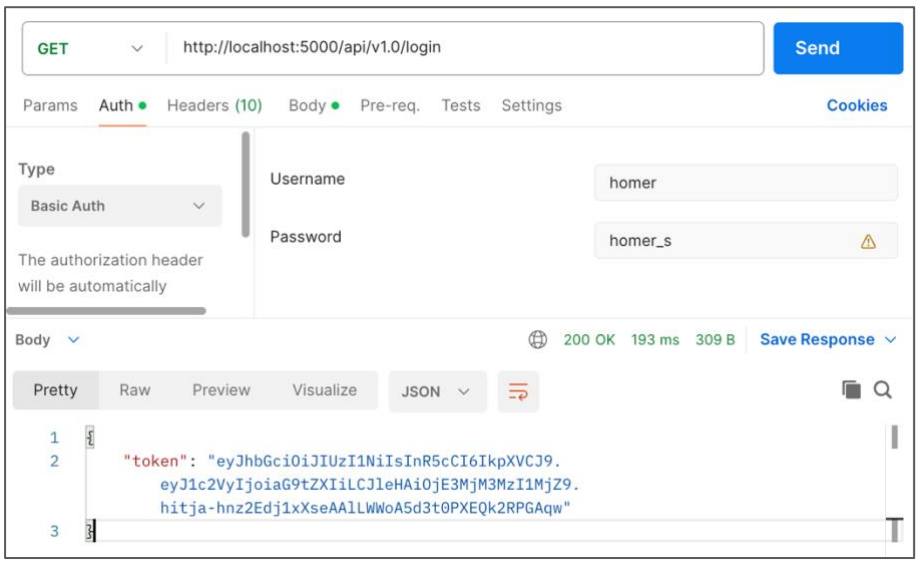

Figure 8.13 User Login

Do it now!	Check the validation in the **`login()`** function by the following tests i) Provide the correct username, but incorrect password ii) Provide a username that does not match any of the documents in the collection iii) Select ***No Auth*** from the list of Authorization types

8.4.3 User Logout

The token generated by the **`login()`** function has an expiry stamp after which time it will automatically become invalid. However, up until that time it will still permit actions to be performed, so it is a good idea to provide a logout mechanism so that a user can indicate that they have completed their session.

By design, JWT tokens cannot be destroyed other than through their own expiry, so we need a mechanism to specify that a token is no longer to be used, even if it is technically still

valid. The most common solution is to develop a **token blacklist**, which is updated with token values as users indicate that they want to log out of the application. When a user sends a GET request to the ***/api/v1.0/logout*** route, the application checks for the presence of a token in the request header and, if one is found, adds that token to the ***blacklist*** collection in the database.

File: app.py

```
...

users = db.users
blacklist = db.blacklist

...

@app.route('/api/v1.0/logout', methods=["GET"])
@jwt_required
def logout():
    token = request.headers['x-access-token']
    blacklist.insert_one({"token":token})
    return make_response(jsonify( {
                    'message' : 'Logout successful' } ), 200 )

if __name__ == "__main__":
      app.run(debug=True)
```

Note again that we don't need to explicitly create the ***blacklist*** collection in the database – it will be automatically created for us the first time we insert data to it.

Do it now!	Add code to select the ***blacklist*** collection and then add the **`logout()`** function as shown above. Using Postman, perform a valid login and copy the token generated as a response. Now create a 'x-access-token' key in the ***Headers*** tab and paste the new token as the value. Send a **GET** request to ***/api/v1.0/logout*** and ensure that you see a "Logout successful" message as shown in Figure 8.14 below.

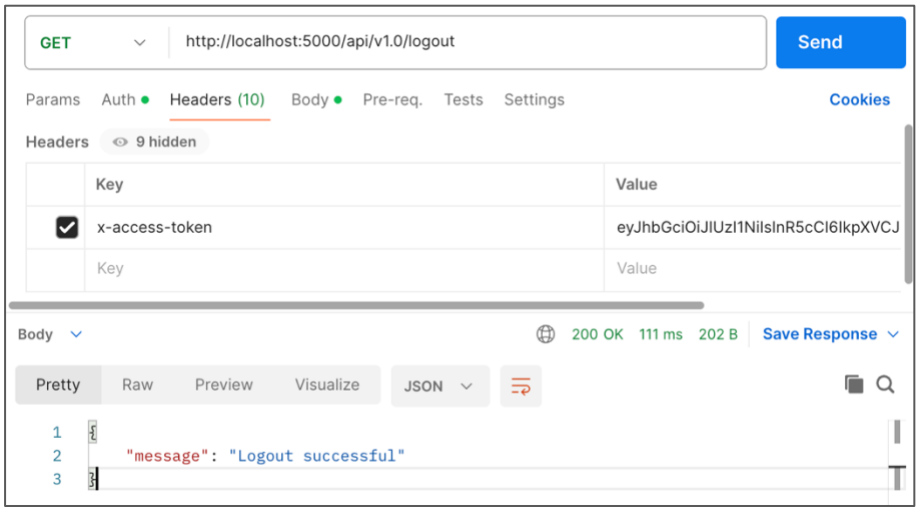

Figure 8.14 User Logout

To check that the token has been written to the blacklist collection, we can use the MongoDB Shell as illustrated in Figure 8.15 below. Here we see that the collection has been created and that it contains a single token value that matches that passed as the 'x-access-token' header.

Figure 8.15 Token Added to Blacklist after Logout

Once a token has been added to the blacklist, we need to make a change to the **`jwt_required()`** decorator to make sure that the token presented is not one that is contained in the blacklist.

This is easily achieved by adding a **`find_one()`** query to check for the token in the blacklist collection just before the **`func()`** passed to the decorator is executed. If the token is found in the blacklist, then an error message is returned, otherwise execution continues as normal.

File: app.py

```
...

def jwt_required(func):
    @wraps(func)
    def jwt_required_wrapper(*args, **kwargs):

        ...

        bl_token = blacklist.find_one({"token":token})
        if bl_token is not None:
            return make_response(jsonify( {
                'message' : 'Token has been cancelled' } ), 401)
        return func(*args, **kwargs)

    ...
```

Do it now!	Add the highlighted code above to the **`jwt_required()`** decorator and try to make a request to a route that requires a valid token (assuming the token you have just used to logout is still in the 'x-access-token' header). Ensure that you receive the "Token has been cancelled" message as illustrated in Figure 8.16, below.

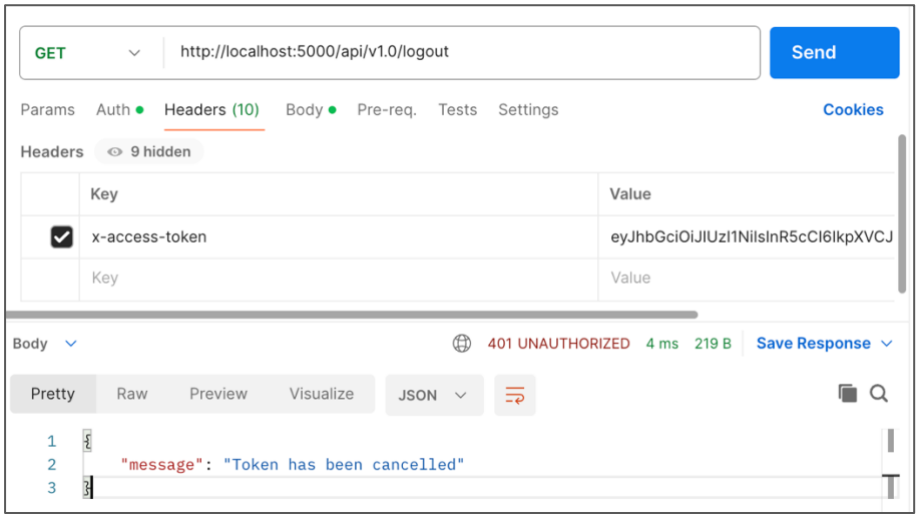

Figure 8.16 Attempting to Use a Blacklisted Token

8.4.4 Multi-level Access

Currently, our application has two levels of access – GET requests on businesses and reviews that are available to anyone, and POST/PUT/DELETE requests that are available to logged in

users with a valid JWT token. In this section, we will extend the access hierarchy to withdraw permission for DELETE requests from general logged-in users and only allow valid token holders with an admin credential to remove businesses and reviews.

In order to implement the new access level, we first modify the **`login()`** function so that the token payload includes the user's admin status flag.

File: app.py

```
...

def login():
    auth = request.authorization

    if auth:
        user = users.find_one( {'username':auth.username } )
        if user is not None:
            if bcrypt.checkpw(bytes(auth.password, 'UTF-8'),
                              user["password"]):
               token = jwt.encode( {
                 'user' : auth.username,
                 'admin' : user['admin'],
                 'exp' : datetime.datetime.now(datetime.UTC) +
                         datetime.timedelta(minutes=30) },
                 app.config['SECRET_KEY'],
                 algorithm="HS256")
               return make_response(
                  jsonify({'token' : token}), 200)
...
```

Next, we require another decorator **`admin_required()`** that checks for the JWT token presented having the ***admin*** flag set to true before allowing access to the operation. This decorator will be applied to delete routes **as well as** that which checks for a valid JWT token, so it need only check the admin flag in the token and does not have to repeat the checks for valid, non-blacklisted tokens.

File: app.py

```
...

def admin_required(func):
    @wraps(func)
    def admin_required_wrapper(*args, **kwargs):
        token = request.headers['x-access-token']
        data = jwt.decode(
           token, app.config['SECRET_KEY'], algorithms="HS256")
        if data["admin"]:
            return func(*args, **kwargs)
        else:
            return make_response(jsonify( {
               'message' : 'Admin access required' } ), 401 )
    return admin_required_wrapper

...
```

Finally, we apply the new decorator to the routes that service the delete requests on businesses and reviews.

File: app.py

```
...

@app.route("/api/v1.0/businesses/<string:id>",
                              methods=["DELETE"])
@jwt_required
@admin_required
def delete_business(id):

...

@app.route("/api/v1.0/businesses/<bid>/reviews/<rid>",
                                     methods=["DELETE"])
@jwt_required
@admin_required
def delete_review(bid, rid):

...
```

Do it now!	Add the additional code to the `login()` function and create the new decorator and apply it to the methods to delete businesses and reviews. Test it by logging in with admin and non-admin users and try to delete a business. Verify that the application behaves as shown in Figure 8.17 below.

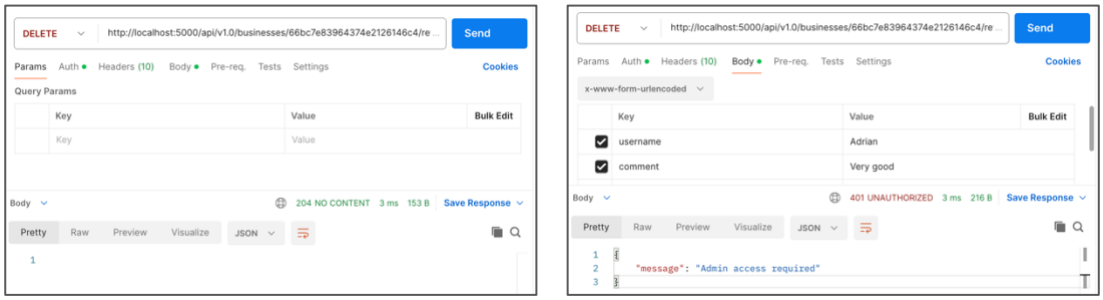

Figure 8.17 Delete Request for Admin and non-Admin Users

Try it now!	Test the new code by trying the following: i) Issue a POST request to add a review to a business ii) Issue a GET request to make sure that the review has been added iii) Repeat the POST request without the token and check the error that is generated iv) Repeat the POST request with an invalid token value and check the error that is generated

8.5 Further Information

- https://jwt.io/
  JSON Web Tokens home page

- https://pypi.org/project/PyJWT/
  Python PyJWT package – home page

- https://dev.to/apcelent/json-web-token-tutorial-with-example-in-python-23kb
  JSON Web Token Tutorial with example in Python

- https://www.youtube.com/watch?v=VW8qJxy4XcQ
  HTTP Basic Authentication - YouTube

- https://www.youtube.com/watch?v=J5bIPtEbS0Q
  Authenticating a Flask API with JSON Web Tokens – YouTube

- https://www.youtube.com/watch?v=WxGBoY5iNXY&t=601s
  Creating a RESTFul API in Flask with JSON Web Token Authentication and Flask-SQLAlchemy – YouTube

- https://www.youtube.com/watch?v=mZ5IwFfqvz8
  The basics of Python decorators – YouTube

- https://www.youtube.com/watch?v=nYDKH9fvlBY
  Python decorators tutorial – YouTube

- http://book.pythontips.com/en/latest/args_and_kwargs.html
  Using *args and **kwargs in Python

- https://www.youtube.com/watch?v=SKswJH7_plQ
  How to send JWT tokens as headers with Postman – YouTube

- https://pypi.org/project/bcrypt/
  Bcrypt package home page

- https://docs.python.org/3/library/stdtypes.html#bytes-objects
  Python bytes object - manual

- https://www.w3resource.com/python/python-bytes.php
  Python bytes and bytearrays

- https://stackoverflow.com/questions/21978658/invalidating-json-web-tokens
  Invalidating JSON Web Tokens

- https://dev.to/_arpy/how-to-log-out-when-using-jwt-4ajm
  How to log out using JWT

- https://blog.teclado.com/learn-python-defining-user-access-roles-in-flask/
  Defining user access levels in Flask