

COM661 Full-Stack Strategies and Development

Practical B4: API Authentication

Aims

- To introduce Basic Authentication and JSON Web Tokens as a basis for implementing a security protocol for an application
- To introduce the `pyjwt` package
- To implement a function that creates a new JWT token
- To demonstrate the concept of decorators in Python
- To implement a Python decorator that protects functions from unauthorised use
- To enhance the basic decorator to cater for variable parameters and return values
- To apply the decorator-based security platform to the API previously developed
- To demonstrate the presentation of the JWT token in the HTTP request header.

Contents

B4.1 JSON WEB TOKENS	2
B4.1.1 BASIC AUTHENTICATION AND PREPARATION FOR JWT	2
B4.1.2 PROVIDING A LOGIN ROUTE	3
B4.2 PYTHON DECORATORS	6
B4.2.1 BUILDING A FIRST DECORATOR	6
B4.2.2 USING DECORATORS	8
B4.3 PROTECTING SELECTED ROUTES	12
B4.3.1 BUILDING THE DECORATOR	12
B4.3.2 APPLYING THE PROTECTION	13
B4.3.3 PASSING THE TOKEN IN THE REQUEST HEADER	15
B4.4 FURTHER INFORMATION	17

B4.1 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 is able to 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.

B4.1.1 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:

```
U:\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: biz/app.py

```
from flask import Flask, request, jsonify, make_response
from bson import ObjectId
from pymongo import MongoClient
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*.

B4.1.2 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: biz/app.py

```
...

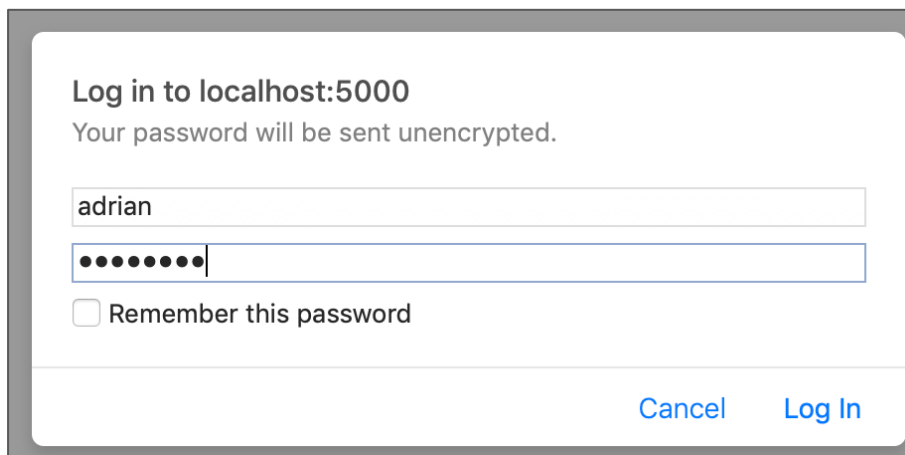
@app.route('/api/v1.0/login', methods=['GET'])
def login():
    auth = request.authorization
    if auth and auth.password == 'password':
        token = jwt.encode( \
            {'user' : auth.username, \
             'exp' : datetime.datetime.utcnow() + \
                     datetime.timedelta(minutes=30)}, \
            app.config['SECRET_KEY'])
        return jsonify({'token' : token.decode('UTF-8')})

    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 B4.1 is generated and that when any username and the password “password” are provided, a token is generated and displayed as shown in Figure B4.2.



Log in to localhost:5000

Your password will be sent unencrypted.

adrian

.....

☐ Remember this password

Cancel Log In

Figure B4.1 Username and password prompt

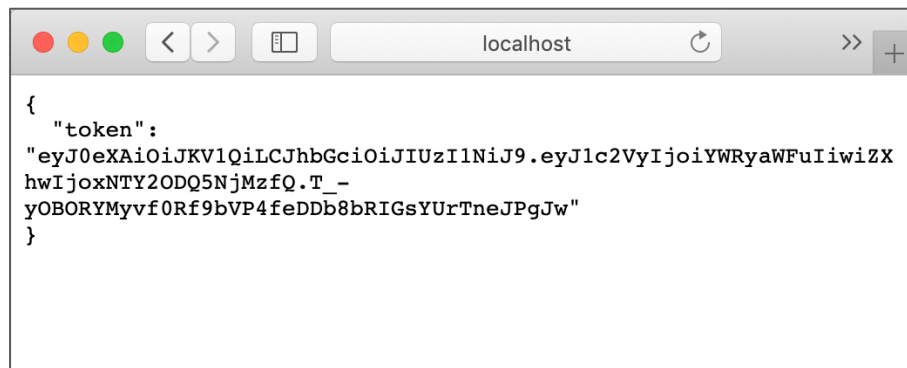


Figure B4.2 Token generated

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 B4.3. 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 URLs, or any other information.

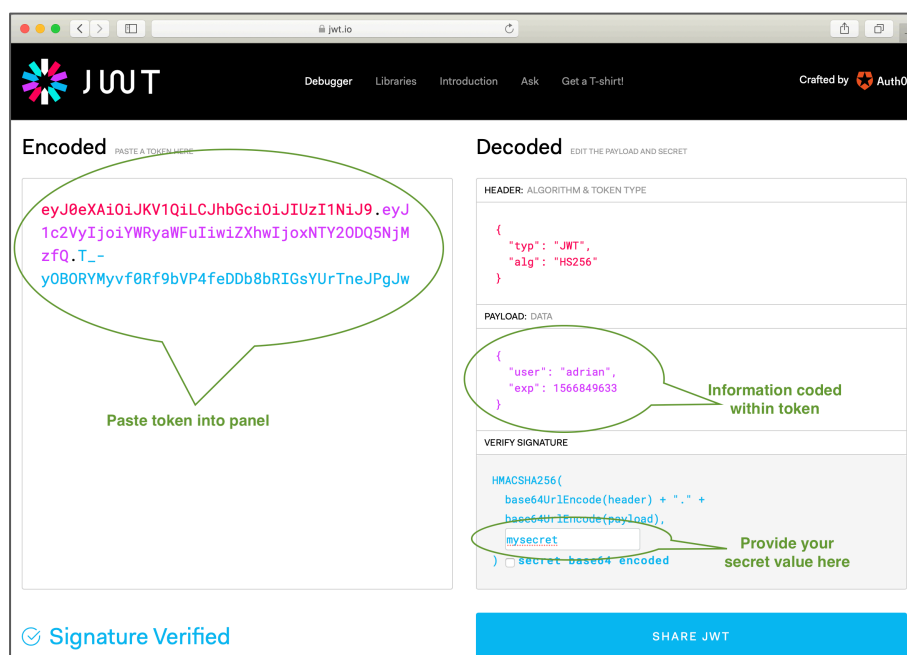


Figure B4.3 Analysing the token

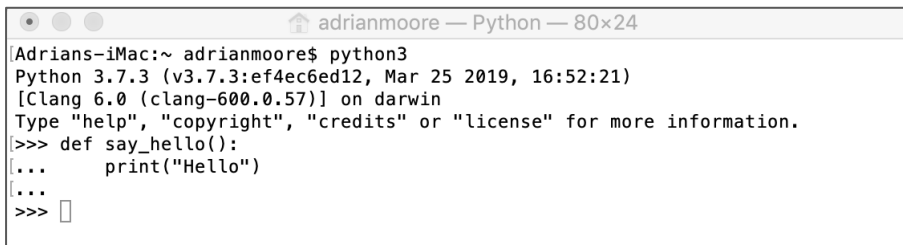
Do it now!	Follow the steps illustrated above to generate a JWT token and verify it in the online tool at http://jwt.io .
-------------------	--

B4.2 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.

B4.2.1 Building a first decorator

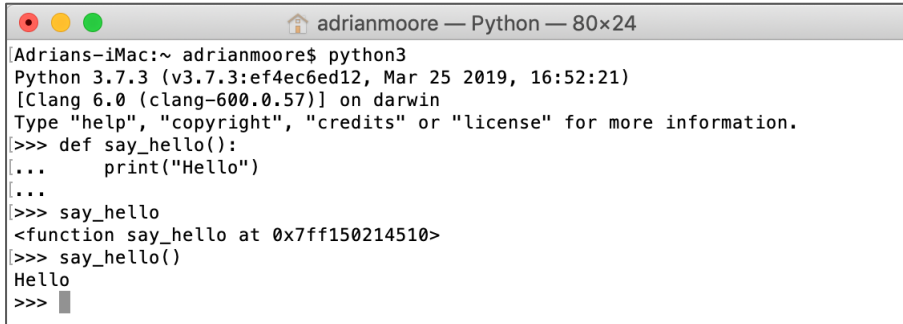
To demonstrate, we will use the Python Command prompt to demonstrate applying decoration to a function. First, we enter the Python command prompt and define a simple function.



```
adrianmoore — Python — 80x24
[Adrians-iMac:~ adrianmoore$ python3
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:52:21)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> def say_hello():
...     print("Hello")
...
>>> ]
```

Figure B4.5 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.

A terminal window titled 'adrianmoore — Python — 80x24' showing the execution of a Python script. The prompt is 'Adrians-iMac:~ adrianmoore\$ python3'. The output shows Python 3.7.3 version information and the execution of a function 'say_hello()' which prints 'Hello'.

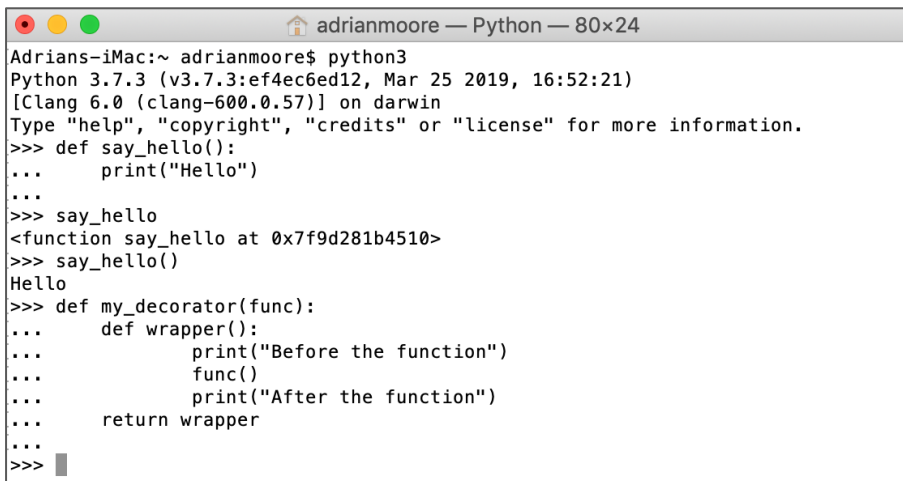
```
Adrians-iMac:~ adrianmoore$ python3
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:52:21)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def say_hello():
...     print("Hello")
...
>>> say_hello
<function say_hello at 0x7ff150214510>
>>> say_hello()
Hello
>>>
```

Figure B4.6 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 B4.7, below.

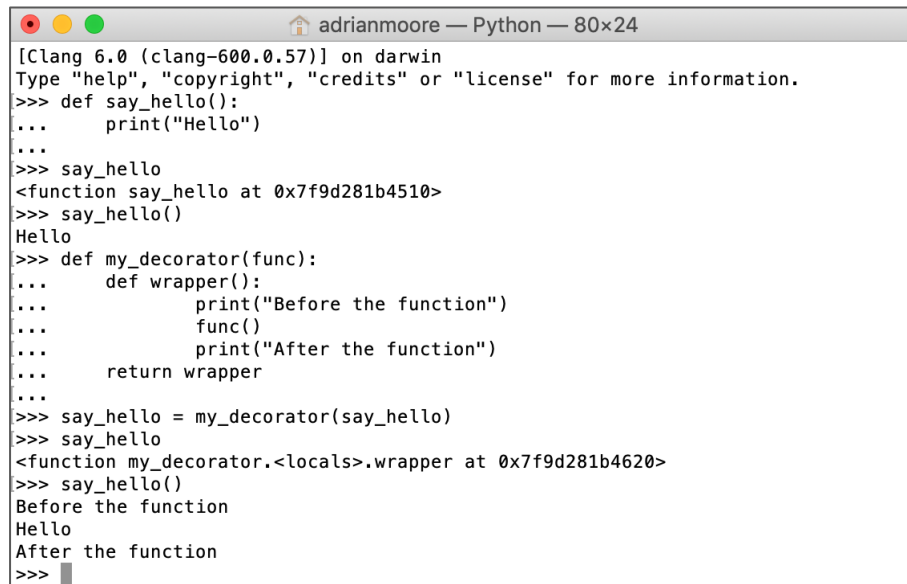
A terminal window titled 'adrianmoore — Python — 80x24' showing the execution of a Python script. The prompt is 'Adrians-iMac:~ adrianmoore\$ python3'. The output shows Python 3.7.3 version information, the execution of a function 'say_hello()' which prints 'Hello', and the definition of a decorator 'my_decorator(func)' which defines a 'wrapper()' function. The wrapper function prints 'Before the function', calls 'func()', prints 'After the function', and returns the result of 'func()'. The prompt is currently at the end of the 'my_decorator' definition.

```
Adrians-iMac:~ adrianmoore$ python3
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:52:21)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def say_hello():
...     print("Hello")
...
>>> say_hello
<function say_hello at 0x7f9d281b4510>
>>> say_hello()
Hello
>>> def my_decorator(func):
...     def wrapper():
...         print("Before the function")
...         func()
...         print("After the function")
...     return wrapper
...
>>>
```

Figure B4.7 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.



```
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def say_hello():
...     print("Hello")
...
>>> say_hello
<function say_hello at 0x7f9d281b4510>
>>> 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 0x7f9d281b4620>
>>> say_hello()
Before the function
Hello
After the function
>>>
```

Figure B4.8 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.
-------------------	--

B4.2.2 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 B6 Files** Zip archive) 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: biz/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 <i>decorators_1.py</i> to your application (copy it into the <i>biz</i> 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 Zip archive) that demonstrates the decorator applied to two functions, ***shout_out()*** and ***whisper_it()***.

File: biz/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 <i>decorators_2.py</i> 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: *biz/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: biz/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 **printed** by the function, but instead is returned from it.

File: biz/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: biz/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.

B4.3 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

B4.3.1 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 querystring 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: biz/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 jsonify( \
                {'message' : 'Token is missing'}), 401
        try:
            data = jwt.decode(token, \
                               app.config['SECRET_KEY'])
        except:
            return jsonify( \
                {'message' : 'Token is invalid'}), 401
        return func(*args, **kwargs)

    return jwt_required_wrapper

...
```

B4.3.2 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 a “token required” 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: biz/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 B4.9 below.



Figure B4.9 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 B4.9.

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 B4.1. If we copy the token and add it to the URL as a querystring 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 B4.10, below.



Figure B4.10 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 querystring 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 B4.10 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.
--------------------	--

B4.3.3 Passing the token in the request header

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

File: biz/app.py

```
...

def jwt_required(func):
    @wraps(func)
    def jwt_required_wrapper(*args, **kwargs):
        #token = request.args.get('token') - REMOVE THIS
        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 B4.11 below.

Note:	As an alternative, 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.
--------------	--

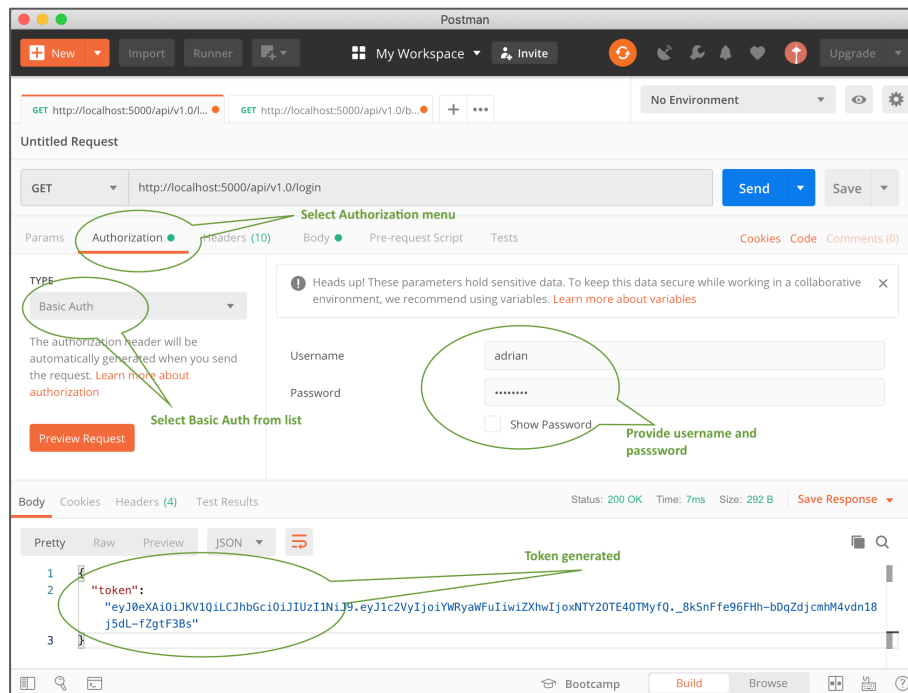


Figure B4.11 Basic Auth in Postman

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 **name**, **comment** 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 B4.12, below.

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

Try it now!	<p>Test the new code by trying the following:</p> <ul style="list-style-type: none"> i) Issue a GET request to make sure that the review has been added ii) Repeat the POST request without the token and check the error that is generated iii) Repeat the POST request with an invalid token value and check the error that is generated
--------------------	---

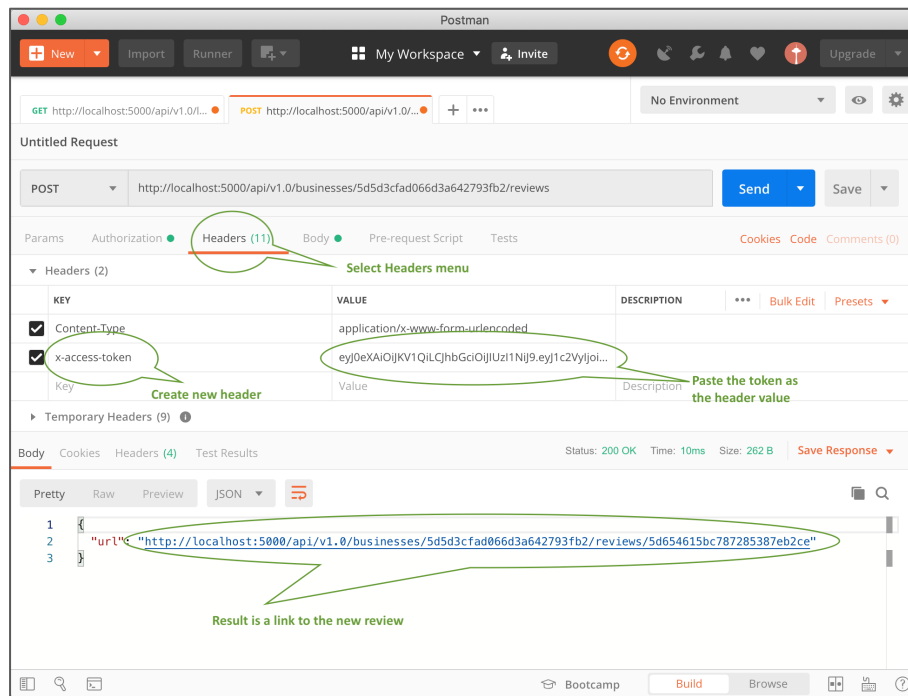


Figure B4.12 Passing the token with a POST request

B4.4 Further Information

- <https://realpython.com/token-based-authentication-with-flask/>
Token-based authentication with Flask
- <https://jwt.io/>
JSON Web Tokens home page
- <https://pypi.org/project/jwt/>
Python jwt 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=mZ5lwFfgvz8>
The basics of Python decorators - YouTube

- <https://www.youtube.com/watch?v=nYDKH9fvIBY>
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