Web Frameworks, and Flask

Reading / References

- Flask book on ACM[1]
- Flask documentation[2]
- A Flask tutorial[3]
- Full Flask API[4]

Notes

Web Frameworks

Any language that you choose for your web service or application will need certain features in common. This is what **web frameworks** can offer you, and all languages have usually more than one such framework.

Web Frameworks can help you with a whole host of necessary tasks:

- You need to set up a listener on a particular network port for incoming HTTP traffic.
- You need a way to process the information that is an HTTP request from its default wall-of-text form into more usable structures. For example, such a framework will give convenient access to the request's type, the specific URI path, the passed parameters if any, the different headers etc.
- Similarly, you need a way to create HTTP responses from the objects that you want to serve. For example: preparing a suitable status code message, including appropriate headers for the content you are trying to serve, etc.
- You need to relate URL "routes" to specific functions in your application code. We'll refer to this as **URL dispatch** or **route dispatch**.
- You need a way to use **templates** to automatically generate content.
- For web applications, you need some standard protections around submitting web forms.
- You need ways to maintain sessions of a user across multiple requests.
- You need easy connections to databases, integrated with the rest of the application.
- You need systems to help with caching requests, handling incorrect routing requests, logging of the requests being made, etc.

Most web frameworks offer these and more. Some do it themselves, some allow for extensions. Some are very opinionated about exactly how you should structure your application, some give you more control over it. But whatever your project is, you would benefit from using some web framework, instead of reinventing the wheel.

[1] https://www.safaribooksonline.com/library/view/flask-web-development/9781491991725/
[2] http://flask.pocoo.org/
[3] http://flask.pocoo.org/docs/1.0/tutorial/
[4] http://flask.pocoo.org/docs/1.0/api/

Flask

Flask is a Python Web Framework that in general takes a very minimalist approach. It offers some basic functionality, and leaves a lot of choices up to the programmer.

We will build a very simple banking system using Flask.

Our example service We start by discussing the example service we are envisioning implementing. This is a three-step process:

- Describe some overall requirements. You can find this information at the overview.md file[5].
- The next step is to think through the URI resources and methods that we will support. Here is the resources.md file[6] for that.
- Lastly we should discuss the different database tables that we will need. Those can be found in the tables.md file[7]

You can check out the entire project on your computers by "cloning" the repository:

```
git clone https://github.com/skiadas/banking-flask.git
cd banking-flask
subl .
```

We will split our application into a couple of different files:

- A dedicated db.py file will manage interaction with the database. It will hold classes for the users and transactions and will use ORM to express some of the basic system requirements.
- A main.py file will contain the basic Flask application, and direct the action for the various HTTP requests.
- A utils.py file will provide any helper functions we need, to keep the code in the main file clean.

Setting up the database Let us take a look at the database setup first. The details are in this file[8]:

- We create the declarative Base class, then make two ORM classes that inherit from it.
- The Transaction class contains some functions to help us with our task. In particular it contains the key isPossible function that incorporates our essential business logic.

[5] https://github.com/skiadas/banking-flask/blob/master/specs/
[6] https://github.com/skiadas/banking-flask/blob/master/specs/
[7] https://github.com/skiadas/banking-flask/blob/master/specs/
[8] https://github.com/skiadas/banking-flask/blob/master/app/db.py

- The Db class represents our connection to the database. It maintains a session object for us, and provides us with functions to use to interact with the database, such as adding a transaction, adding or deleting a user, etc.
- Notice in particular the getTransactions function and its helper enrichQuery, which conditionally builds a query based on the provided parameters.

Our database code handles some bad inputs, but it also expects other bad input behaviors to be handled by the code that handles the requests. We will get to that shortly.

TODO

Skeleton Let's take a look at the main file, app/main.py:

```
from flask import Flask, make_response, json, url_for
from db import Db    # See db.py

app = Flask(__name__)

## Setting up database
db = Db()

## Lots of route stuff here
## Will look at it in a moment
## .......

## Helper method for creating JSON responses
def make_json_response(content, response = 200, headers = {}):
    headers['Content-Type'] = 'application/json'
    return make_response(json.dumps(content), response, headers)

## And then we start the app
## Starts the application
if __name__ == '__main__':
    app.run()
```

The object app represents the main application, and offers some key functionalities. It is an instance of the Flask class, the main class of the Flask framework.

In other situations, we may use the app.config object to add some configurations to the application. We won't immediately need this here.

We then instantiate a database instance. This is a custom class that we have created, and stored in app/db.py, to keep the main file somewhat clean. All the database queries should appear in that other file. Take a look at that file now and you should see the Db class with a host of useful stuff. We will later add more database-access methods there.

In any web service one of the most important components is that of determining the route map. In Flask we have multiple ways of doing so, and the simplest one looks like this:

```
@app.route('/', methods = ['GET'])
def index():
    pass

@app.route('/user', methods = ['GET'])
def user_list():
    pass

@app.route('/user/<username>', methods = ['GET'])
def user_profile(username):
    pass

@app.route('/user/<username>', methods = ['PUT'])
def user_create(username):
    pass

@app.route('/user/<username>', methods = ['POST']):
def user_update(username):
    pass

@app.route('/user/<username>', methods = ['DELETE']):
def user_delete(username):
    pass

@app.route('/transaction', methods = ['GET']):
def transaction_list():
    pass

@app.route('/transaction', methods = ['POST']):
def transaction_create():
    pass

@app.route('/transaction/<transactionId>', methods = ['GET']):
def transaction_info(transactionId):
    pass
```

We use the @app.route decorator that takes as input the route, and the accepted methods, and is then followed by the function to use, which it "decorates". This function must return a Response[9] object, and it also has access to a Request[10] object via the global variable request.

So we have here specified what all the available routes are, what their URI schemes look like, and which functions should be called in response to one of the routes. Currently these function do nothing, they are *stumps*. We will need to provide implementations for them.

In large applications we would opt for a different way of writing the routes, that keeps all the routes closer to each other and delegates all the functions to other modules.

Implementations We will start by taking a closer look at some of the functions and what they would do. We will start with the index method, that is supposed to direct

[9] http://flask.pocoo.org/docs/0.11/api/#response-objects
[10] http://flask.pocoo.org/docs/0.11/api/#incoming-request-data

new users to the service. It will tell the system about available routes and maybe suggest methods:

```
@app.route('/', methods = ['GET'])
def index():
    pass
```

Now we move on to user_create, which is in response to a PUT request for creating a new user. We'll need to check that a password is provided, and that the username and password are both alphanumeric. We must either send back a 201 Created, with a link to the corresponding GET page in the Location header, or a suitable error for a bad username, via a 400 response. Also, if hte username already exists, we must return 403, Forbidden. Review appendix C of RESTful Web Services[11] on what the different response codes indicate.

So let's take a look at how this would look:

```
## Creates a new user. Request body contains the password to be used
## If user exists, must ensure it is same or else throw error
@app.route('/user/<username>', methods = ['PUT'])
def user_create(username):
    contents = request.get_json()
    if "password" not in contents:
        return make_json_response({ 'error': 'must␣provide␣a␣password␣field' }, 400)
    password = contents["password"]
    if not username.isalnum() or not password.isalnum():
        return make_json_response({ 'error': 'username␣and␣password␣must␣be␣alphanumeric' }, 40
    user = db.getUser(username)
    if user is not None:
        return make_json_response({ 'error': 'username␣already␣exists' }, 403)
    try:
        db.addUser(username, password)
        db.commit()
        headers = { "Location": url_for('user_profile', username=username) }
        return make_json_response({ 'ok': 'user␣created' }, 201, headers)
    except:
        return make_json_response({ 'error': 'unexpected␣server␣error' }, 500)
```

We simply need to provide the json content of the reply, the error code, and optionally headers. Our make_json_response method will always set the content type appropriately to json. We check to see if the password is provided and if the username and password are alphanumeric, and return appropriate status codes if they are not. Phew that's a lot of work!

Some key things to note:

- All paths out of the function should be returning a Response object. Typically this will happen by calling our make_json_response function.
- Information about the request that came to us is provided via the request global object. For instance we used this above to get that the message's contents. In a similar way we could access the request's headers.

[11] http://learning.acm.org/books/book_detail.cfm?id=1406352&type=safari

- Any "parameters" that were part of the URI scheme are provided as parameters to the function (username in our example above).
- url_for can be used to create links to other routes. It needs to be provided with the name of a function that implements a route, and it returns the url for that route.

Now let us look at a GET request, which needs to return some more information.

```
@app.route('/user/<username>', methods = ['GET'])
def user_profile(username):
    query = request.args
    if "password" not in query:
        return make_json_response({ 'error': 'must␣provide␣a␣password␣parameter' }, 400)
    password = query["password"]
    try:
        user = db.getUser(username)
        if user is None:
            return make_json_response({ 'error': 'unknown␣username' }, 404)
        if user.password != password:
            return make_json_response({ 'error': 'incorrect␣password' }, 400)
        return make_json_response({
            "username": user.username,
            "balance": user.balance,
            "transactions": {
                "link": url_for('transaction_list', user=user.username)
            }
        })
    except:
        return make_json_response({ 'error': 'unexpected␣server␣error' }, 500)
```

TODO

Interacting with the service While we could, and should, create automated tests, it is equally important to have a way to directly interact with the service. In order to do that, we'll need the following steps:

1. You will be using two terminal windows: One will be running the "server", the other will be the "client".
2. In one window you will start the service.
   - First, we will enable debugging. Find the line in messaging.py that sets the DEBUG configuration variable to False, and change that to True.
   - Make sure you create a keys.json file with a JSON object with keys DATABASE, PASSWORD, SERVER and SCHEMA with values corresponding to your database.
   - In one of your terminal windows, go to the application folder and start the server with:

     python3 app/messaging.py

     From now on this window runs a web server, at a web address it provides to you, and you could change via configuration options. It is also in debug mode: When you change the files it will automatically restart itself.

- If you want to manually stop the server, simply interrupt it via Ctrl–C. You can then run the command again to restart it. It may also shut down by itself if one of your file updates brings it to a non-operable condition.

3. In the other window you will start a "client".

   - This is a normal python console, so start it with python3.
   - You will need to load the requests package, so do import requests. This makes it easier for us to make requests.
   - You should be able to already interact with the server. If the server's address is http://127.0.0.1:5000, then you could do a request like:

     ```
     r=requests.get('http://127.0.0.1:5000/users/haris')
     ```

     The object r is now a response object of the requests package, and you can look at its documentation[12] for what you can do. For instance here are some things you can try:

     ```
     r.status_code
     r.content  # Content as string
     r.json()   # Content as JSON
     r.headers
     r.headers['Content-Type']
     ```

For now we will interact with the service in this fashion. Later on we may revisit the question of writing tests.

More Methods We will now look at performing some queries. The main method that performs a complex query is GET users/{user}/messages, which returns a list of messages based on a possibly complex set of query parameters. This will be a good test of sqlalchemy, as the queries we are after will depend on user-specified parameters.

Let's take a closer look at what parameters that query may have:

- There is a parameter called include which can be set to "sent", "received" or "all".
- There is a parameter called show which can be set to "read", "unread" or "all".
- We can specify a order column to choose a field to use for ordering the results, and which can be set to one of "created", "read", "subject", "to" or "from".
- We can specify a direction column to determine the ordering direction, with possible values "asc" or "desc".
- We could specify a to value or a from value.
- We could in theory add more complicated queries, but that is a harder subject.

In this section we will only incorporate the include and show options. The others would be part of your assignment.

Here's how the route method may look like:

[12] http://docs.python-requests.org/en/master/user/quickstart/#response-content

```
@app.route('/users/<username>/messages', methods = ['GET'])
def user_messages(username):
    args = request.args.to_dict()
    error = message.validate_message_query(args, username)
    if error is not None:
        return make_json_response({ 'error': error }, 400)
    results = db.get_messages(args, username)
    return make_json_response({
        'messages': [
            { 'url': url_for('message_get', id=m['id']) }
            for m in results
        ]
    }, 200)
```

Notice how it delegates important work to other functions. We should perhaps have also delegated the work that happens in the return, where urls are being built out of the various messages.

Here is the function querying the database. It sets up a query and based on the user choices it populates it with the necessary WHERE clauses. It then executes the query and returns the results. For your homework you will need to add to this query to account for the other kinds of choices.

```
    def get_messages(self, args, username):
        conn = self.connect()
        query = select([self.messages])
        for field in ['from', 'to']:
            if field in args:
                query = query.where(column(field) == args[field])
        ## Add 'include'
        if args['include'] == 'sent':
            query = query.where(column('from') == username)
        elif args['include'] == 'received':
            query = query.where(column('to') == username)
        else:
            query = query.where(
                or_( column('from') == username, column('to') == username )
            )
        ## Add 'show'
        if args['show'] == 'read':
            query = query.where(column('read') == True)
        elif args['show'] == 'unread':
            query = query.where(column('read') == False)
        ## Perform query
        return conn.execute(query).fetchall()
```

Working with tags Finally let's take a look at one of the methods related to tags. We'll need to create two database accesses, one to fetch a message and one to fetch its tags.

```
## Adds a tag to a message, if it did not exist
@app.route('/messages/<id>/tags/<tag>', methods = ['PUT'])
def tag_add(id, tag):
    if len(tag) > 20:
```

```
        return make_json_response({ 'error': 'tag too long' }, 400)
    message = db.fetch_message(id)
    if message is None:
        return make_json_response({ 'error': 'message not found' }, 404)
    tags = db.fetch_message_tags(id)
    if tag in tags:
        return make_json_response({}, 204)
    # Need to add the tag
    if db.add_tag(id, tag) is None:
        return make_json_response({ 'error': 'server error' }, 500)
    return make_json_response({}, 201)
```

And here are the corresponding functions in the `db` module:

```
# Fetches a single message based on id
def fetch_message(self, id):
    conn = self.connect()
    query = select([self.messages]).where(column('id') == id)
    results = conn.execute(query).fetchall()
    return results[0] if len(results) > 0 else None

# Fetches message tags if any
def fetch_message_tags(self, id):
    conn = self.connect()
    query = select([self.tags.c.tag]).where(column('msg_id') == id)
    results = conn.execute(query).fetchall()
    return map((lambda tag: tag[0]), results)

# Inserts a new message/tag pair
# Should only be called once we have established that the pair does not
# exist, and that id and tag are valid
def add_tag(self, id, tag):
    try:
        conn = self.connect()
        result = conn.execute(self.tags.insert(), msg_id=id, tag=tag)
        return result.inserted_primary_key
    except:
        return None
```