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.

- You need a way to process HTTP requests into a more meaningful form than just text, and similarly to prepare HTTP responses from your objects.
- You need to relate URL "routes" to specific functions in your application code.
- 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.

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.

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 messaging web service using Flask. We start by discussing the service we are envisioning. You can find this information at the GitHub project[5]. Read that file, then design a RESTful service and resource specification around that. You should end up with something similar to this description[6].

[1] http://learning.acm.org/books/book_detail.cfm?id=2621997&type=safari
[2] http://flask.pocoo.org/
[3] http://flask.pocoo.org/docs/0.11/tutorial/
[4] http://flask.pocoo.org/docs/0.11/api/
[5] https://github.com/skiadas/messaging-flask/blob/master/specs/initial_specs.html
[6] https://github.com/skiadas/messaging-flask/blob/master/specs/resources.html

Before you move on, you should check out the whole project on your desktop, via git:

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

The third line would open the project in Sublime Text.

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

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

app = Flask(__name__)
# The keys.json file should contain the 4 properties:
# DATABASE, PASSWORD, SERVER, SCHEMA
app.config.from_json('keys.json')

## Setting up database
db = Db(app.config)

## Lost 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.

We use its "config" object to add some configurations read from a file. In this case this would be a JSON document containing the four fields that the database needs. Make sure to name them using all caps.

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 two methods, set_metadata and connect. 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:

```
## Creates a new user. Request body contains the password to be used
## If user/password exists, must ensure it is same or else throw error
## In first iteration of the app, no passwords.
```

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

We use the @app.route decorator that takes as input the route, the accepted methods and is then followed by the function to use, which it decorates. This function must return a Response[7] object, and it also has access to a Request[8] object at the variable request.

In bigger 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 start with user_create, which is in response to a PUT request for creating a new user. Later on we will be checking for passwords and so on, for now we just need to ensure that the chosen username is not too long. We currently don't directly store users in their own tables, so there is nothing to save. 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 (too long) username, via a 400 response. Review appendings C of RESTful Web Services[9] on what the different response codes indicate.

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

```
@app.route('/users/<username>', methods = ['PUT'])
def user_create(username):
    if len(username) > 20:
        return make_json_response({ 'error': 'long_username' }, 400)
    return make_json_response({}, 201, {
        'Location': url_for('user_page', username=username)
    })
```

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 the length of the username, and if it is too long then we return a 400 error, otherwise we return a 201. When we do password authentication, this will all become more complex.

Now let us take a look at the corresponding GET request. This returns the user's main page. It should consist mostly of links. For instance it would have a link to a page showing the user's "sent" message, and another to the user's "received" messages. It could also have a link to "create a new message". Here's how that might look like:

```
## Get user information. Should provide links to various tasks like
## looking at sent messages or received messages or creating a new message
@app.route('/users/<username>', methods = ['GET'])
def user_page(username):
    return make_json_response({
        'user': username,
        'sent': url_for('user_messages', username=username, include='sent'),
```

[7] http://flask.pocoo.org/docs/0.11/api/#response-objects
[8] http://flask.pocoo.org/docs/0.11/api/#incoming-request-data
[9] http://learning.acm.org/books/book_detail.cfm?id=1406352&type=safari

```
        'received': url_for('user_messages', username=username, include='received'),
        'create': {
            'url': url_for('user_new_message', username=username),
            'content': { 'to': '', 'subject': '', 'body': '' }
        }
    }, 200)
```

We will next look at the request that handles the creation of a new message. This is the first request that will actually interact with the database. It is a POST request. It would need to do the following things:

- Check that the username has length no more than 20.
- Check that content object contains a recipient of length no more than 20.
- Check that content object contains a subject of length no more than 140.
- Check that content object contains a body of length no more than 5000.
- If any of the above fails it would return a 400 "bad request" error.
- Otherwise it attempts to write to the database. It this fails it would return a 500 "internal server error". We looked through the 5xx range of errors, which represent "server-side errors" and this seemed the most suitable.
- If the database write is successful, then we return a 201 response for the newly created resource, and put a link to it in the Location header.

Here's the skeleton of this method, that uses two custom methods we will write, one for validation and one for writing to the database:

```
@app.route('/users/<username>/messages', methods = ['POST'])
def user_new_message(username):
    contents = request.get_json()
    contents['from'] = username
    error = message.validate_new_message(contents)
    if error is not None:
        return make_json_response({ 'error': error }, 400)
    record_id = db.write_message(contents)
    if record_id is None:
        return make_json_response({ 'error': 'Internal␣Server␣Error' }, 500)
    return make_json_response({}, 201, {
        'Location': url_for('message_get', id=record_id)
    })
```

Here is how the validate function may look like:

```
def validate_new_message(m):
    for field in ['to', 'subject', 'body']:
        if field not in m:
            return 'Required␣fields:␣to,␣subject,␣body'
    if len(m.keys()) > 4:
        return 'Only␣fields␣allowed:␣to,␣subject,␣body'
    for field, length in [('from', 20), ('to', 20),
                          ('subject', 140), ('body', 5000)]:
        if len(m[field]) > length:
            return 'Field␣"%s"␣must␣not␣exceed␣length␣%d' % (field, length)
    return None
```

And here is the query function in the db module:

```
def write_message(self, m):
    try:
        conn = self.connect()
        m['created'] = datetime.today()
        if 'read' not in m:
            m['read'] = False
        result = conn.execute(self.messages.insert(), m)
        return result.inserted_primary_key[0]
    except:
        return None
```

Well this is awesome, we can now send messages to our service! But we should talk about how to test all this before we do more stuff.

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[10] 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:

```
@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)
```

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

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
```