Web Frameworks, and Flask

Reading / References

- Flask book on ACM[1]
- Flask documentation[2]
- A Flask tutorial[3]
- Full Flask API[4]
- Werkzeug documentation[5] heavily used by Flask.

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.

[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/
[5]http://werkzeug.pocoo.org/docs/0.14/

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 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[6].
- The next step is to think through the URI resources and methods that we will support. Here is the resources.md file[7] for that.
- Lastly we should discuss the different database tables that we will need. Those can be found in the tables.md file[8]

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[9]:

- We create the declarative Base class, then make two ORM classes that inherit from it.

[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/specs/
[9] https://github.com/skiadas/banking-flask/blob/master/app/db.py

- 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.
- 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.

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__)
app.debug = True # Comment out when not testing

## 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[10] object, and it also has access to a Request[11] 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. One of the important parts is to create "custom error handlers":

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

```
@app.errorhandler(500)
def server_error(e):
    return make_json_response({ 'error': 'unexpected␣server␣error' }, 500)

@app.errorhandler(404)
def not_found(e):
    return make_json_response({ 'error': e.description }, 404)

@app.errorhandler(403)
def forbidden(e):
    return make_json_response({ 'error': e.description }, 403)

@app.errorhandler(400)
def client_error(e):
    return make_json_response({ 'error': e.description }, 400)
```

The 500 handler is called whenever an exception occurs in our code. The other handlers are triggered by us manually by using the abort construct.

Now we will look at our normal response functions. We will start with the index method, that is supposed to direct new users to the service. It will tell the system about available routes and maybe suggest methods:

```
@app.route('/', methods = ['GET'])
def index():
    return make_json_response({
        "users": { "link": url_for("user_list") },
        "transactions": { "link": url_for("transaction_list") }
    })
```

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[12] 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):
    password = getPasswordFromContents()
    checkAlphanum(username, password)
    checkNameAvailable(username)
    db.addUser(username, password)
    db.commit()
    headers = { "Location": url_for('user_profile', username=username) }
    return make_json_response({ 'ok': 'user␣created' }, 201, headers)

def getPasswordFromContents():
    contents = request.get_json()
```

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

```
    if "password" not in contents:
        abort(400, 'must␣provide␣a␣password␣field')
    return contents["password"]

def checkAlphanum(*args):
    for arg in args:
        if not arg.isalnum():
            abort(400, 'username␣and␣password␣must␣be␣alphanumeric')

def checkNameAvailable(username):
    user = db.getUser(username)
    if user is not None:
        abort(403, 'username␣already␣exists')
```

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.
- 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):
    password = getPasswordFromQuery()
    user = getUserAndCheckPassword(username, password)
    return make_json_response({
        "username": user.username,
        "balance": user.balance,
        "transactions": {
            "link": url_for('transaction_list', user=user.username)
        }
    })
```

Interacting with the service: Automated testing There are fundamentally two ways to interact with and test your service: *Automated tests* and *Interactive sessions/messaging*.

A start at automated testing can be found in the file tests.py in the project folder. It contains likes like the following:

```
from main import app, db

app.config['TESTING'] = True
client = app.test_client()

r = client.get("/")
assert(r.status_code == 200)
assert("users" in r.json and "link" in r.json["users"])
```

The first three lines set everything up. app is a Flask object, and it provides a test_client object for our use. we can then use this client object to make requests of the server.

This way the server never has to run on a live system, but the client does allow us to test the server's behavior as if it was live.

Interacting with the service: Interactive session In order to set up an interactive session, we need in effect two things:

1. One terminal window to run a local "server" based off of the Flask application. Start this server by running the main.py file, and this will kick-start a server. You will need to use the server's address http://127.0.0.1:5000 to talk to it.
2. A second terminal window to run an interactive Python shell. Start a Python shell and import the requests library, then use it to send requests to the above address.
3. You can look at the "server" window for any logging messages when things go wrong with the application.
4. If you change your server code, you need to terminate its service with Ctrl−C and then restart it in order for it to take effect.