Assignment 2
IN608: Intermediate Application Development Concepts
Semester Two, 2021

Directions

Your assignment is to write a chat server that implements an instant system. Details on the requirements for the server are describe below. An example client for your server is provided in the `example_client` subdirectory of the `assignment2` directory. You do not need to write your own client.

Start your work on this assignment by visiting this GitHub Classroom link: `https://classroom.github.com/a/5E-eQxnX`. After creating your assignment repository, clone it to your workstation. Create a "`working`" branch off the main branch. You should make multiple commits to your working branch as you reach various milestones in your work. When you are ready to submit your work for marking, issue a pull request back to the main branch, selecting `tclark` as the reviewer. Note that you get one attempt to submit your work. However, if you make a mistake in your submission, like forgetting to include some files, just let the lecturer know.

Your submission may require some additional libraries in order to run. If this is the case, include either a `Pipfile/Pipfile.lock` or a `requirements.txt` file at the top level of your repository. Omitting this will result in a deduction of 5 marks on the assignment. The `Pipfile` form is recommended.

This assignment is work 50% of your total mark in this paper.

Assignment submissions are due on 5:00 PM on the 18th of November.

1 Project Overview

On a high level, your server shall

1. listen for incoming client TCP connections on port 65432 and accept them;
2. receive and process client requests, maintaining client state as necessary;
3. send responses to client requests;
4. handle error conditions gracefully.

2 Message format

Messages to and from the server have the following format:

Preheader A two byte unsigned integer indicating the length, in bytes, of the header.

Header A json-formatted utf-8 string, like the following:

```
{ "Content-type": "application/json",
  "Content-encoding": "utf-8",
  "Content-length": 42
}
```

Your server must support the `application/json` type and the `utf-8` encoding. If a header indicates an unsupported type or encoding, return an error. The `Content-length` field indicates the length, in bytes, of the message body.

The message body varies with the type of request and is described in the next section.

3 Requests and Responses

A client may send four different types of request. Your server must handle and respond to requests as json-formatted utf-8 strings.

Login: A login request body looks like this example

```
{ "action": "login",
  "params": {"name": "username"}
}
```

Note that there is no authentication of users, so any user name is assumed to belong to a valid user. Usernames uniquely identify a user.

A response body for a login request has the form

```
{ "action": "login",
  "result": "ok | error",
  "errors": []  # a (possibly empty) list of error strings
}
```

Logout: A logout request body looks like this example

```
{ "action": "logout",
  "params": null
}
```

A response body for a logout request has the form

```
{ "action": "logout",
  "result": "ok | error",
  "errors": []  # a (possibly empty) list of error strings
}
```

Send: A send request body looks like this example

```
{ "action": "send_messages",
  "params": {"messages": [ ...messages...]}
}
```

The server shall process the messages and save them for later delivery to the correct user.

The format of individual messages is described below.

A response body for a send request has the form

```
{ "action": "send_messages",
  "result": "ok | error",
  "errors": []  # a (possibly empty) list of error strings
}
```

Get: A get request body looks like this example

```
{ "action": "get_messages",
  "params": {"last_read": "2021-05-23T04:19:45.184775"}
}
```

Note that the `last_read` parameter may be omitted or may be null. In these cases, the response should deliver every message destined for the logged-in user. Otherwise, deliver only messages sent after the `last_read` timestamp.

All times are UTC.

A response body for a get request has the form

```
{ "action": "get_messages",
  "result": "ok | error",
  "messages": [ ...messages... ],
  "errors": []  # a (possibly empty) list of error strings
}
```

Message format: A json-formatted message has the form

```
{ "to": "alice",
  "from": "bob",
  "msg": "spam, spam, spam",
  "sent": "2021-05-23T04:19:45.184776"
}
```

However, messages sent from a client need only provide the `to` and `msg` since the server can fill in the remaining fields when it processes the send request.

4 Functional Requirements

Your server shall satisfy the following functional requirements.

1. It should listen for and serve requests on the `localhost` (IPv4) interface, port 65432.

2. It shall be capable of serving multiple concurrent client requests. In practice, I'll only test it with about four concurrent clients, so it doesn't have to be *webscale*.

3. It shall receive requests and send responses as described above. A sample client is provided, and if by some error the client behaviour differs from the description in this document, the client behaviour shall be regarded as authoritative.

4. Messages shall be saved in an SQLite database so that they persist across restarts. You may choose to save other information in the database as well.

5. Client requests shall be logged to a logfile, one line per request. The logfile lines should be formatted like

```
<timestamp> (ISO 8601):<username>:<request type>:<ok|error>
```

Typically we would expect a server process like this to detach and run as a daemon. This is *not* a requirement, but you are invited to explore doing this.

5 Architecture

This is your opportunity to make your own architectural decisions, but on a high level you'll probably need the following elements:

- A *service* class that handles listening to, reading from, and writing to your sockets.
- One or more *request handler* classes that take processed request data from the service object, perform needed actions, and prepare a response.
- A *message* class to save and retrieve messages.
- A *user* class to handle user information and actions.

There are some opportunities to apply design patterns here, but the right choices depend on your architectural decisions and are up to you.

6 Delivery

Your pull request submitting your assignment should include the following.

- A README file describing any preparation required before running your server and instructions on how to run your server;
- A `Pipfile/Pipfile.lock` or `requirements.txt` with any needed libraries.
- The code for your server in a runnable state.
- Unit test files for your request handler class(es) (described above), along with instructions in the README for running your tests. Your tests should include appropriate uses of mocks to isolate the classes being tested. Note that you may include more unit tests, but for marking purpose the request handler tests will be checked. If for some reason your design doesn't include something that looks like a request handler, talk to the lecturer about an appropriate class to test.

7 Marking

Your assignment will be marked after testing using clients and examination of your submitted code and resources. Your marks will be based on

- satisfying the requirements and protocols described in this document;
- the architectural quality of the submitted software, including adherence to SOLID principles and appropriate application of design patterns;
- organisation and readability of code, including PEP 8 compliance.