Chat Server and Client

Creating a simple social-media service

Prof. Dr. Bradley Richards

http://javaprojects.ch

bradley.richards@fhnw.ch

Management Summary

Online communication is ubiquitous. People chat in games, message on their phones, hold business meetings, communicate with teams members, keep in touch with family, and much more. There are hundreds, possibly thousands of chat applications and services.

The goal of this project is to create part of such a service. You can choose to create either a server or a client. In either case, it is essential to implement the protocol as defined in this document, so that your software is compatible with all the other implementations.

For testing purposes, an existing server is running on http://javaprojects.ch on port 50001 If you are creating a server, your server must respond the same way as that sample server. If your are creating a client, your client must work with that server.

Table of Contents

Server Architecture

The sample server is implemented in Java, not using any framework. It does not use a database, meaning that if it is restarted, it begins with no data. In addition, older data is regularly deleted: accounts that have not been used for a few days are removed to prevent clutter.

Security

Some commands provide information that is insecure, for example, providing a list of registered users. These commands are intended to be helpful for student projects, but should not exist in an actual, public application.

Communication is via plain HTTP (unencrypted). This makes debugging and server administration easier, but is obviously not suitable for production. If you want to experiment with SSL, read the document Howto_SSL_Certificates_in_Java.

Packages and Classes

In the sample server, messages are handled by the class hierarchy of “handlers”. The abstract class **`Handler`** implements **`HttpHandler`** and provides the basic code to read and write requests.

Handlers classes for various paths extend the **`Handler`** class and override the methods for handling GET and POST requests. The top-level **`Server`** class uses these classes with an HttpServer to create specific contexts, i.e., to delegate paths to particular handlers.

The class **`Account`** represents registered users, together with their passwords. The class **`Client`** represents users who are currently logged into the system.

Communications protocol

The messaging protocol is RESTful and uses JSON for data transfer. The mappings and the message contents are detailed in the next session.

When a client successfully logs in, the server sends the client a token. This token proves the client’s identity and is sent with subsequent requests. Tokens are deleted after one hour of inactivity. Users are “online” if they have a valid token. You can only send messages to users who are online.

There are three general options for handling JSON:

- Complex but powerful: Create a set of message classes corresponding to message types. The Jackson libraries can translate between JSON and objects of classes. See https://mvnrepository.com/artifact/com.fasterxml.jackson.core
- Simple but less powerful: Use the Java JSON library to parse JSON into generic arrays or objects. See https://mvnrepository.com/artifact/org.json/json
- DIY is fun, but error-prome: Parse the JSON yourself using methods from the `String` class plus your own code.

Interface specification

If the server is sent an invalid mapping, it will return a status 418. If the mapping is valid, it will return a status 200. Of course, that does not mean that the request succeeded: For example, a login attempt might use an invalid password. If an error occurs in a valid mapping, the returned JSON will contain an error message like this:

```
{ "Error" : "Something went wrong" }
```

The error message will generally be informative, e.g., “invalid password”. The specification below does not mention error messages, but you may assume they exist as needed.

Ping mappings

The following mappings let you test server connectivity and token validity:

- GET `/ping`: Tests the connection with the server. Returns `{"ping":true}`
- POST `/ping`: Requires a `token`. Returns `{"ping":true}` if the token is valid.

User mappings

The available user mappings are:

- GET `/users`: Returns a list of all registered usernames. (debug functionality)
- GET `/users/online`: Returns a list of all users currently online. (debug functionality)
- POST `/user/register`: Register a new user. Requires `username` and `password`, both of which must be at least 3 characters in length. Returns the username if successful: `{ "username" : "marble" }`
- POST `/user/login`: Login a user. Requires `username` and `password`. Returns a token if the login was successful: `{"token":"75B595472275E6EA583BD607DFA2F60F"}`
- POST `/user/logout`: Logout a user. Requires a `token`. Always returns `true`. `{ "logout" : true }`
- POST `/user/online`: Discover whether a user is currently online. Requires a `token` and the `username`. Returns `true` or `false`: `{ "online" : true }`

Chat mappings

These mappings provide the basic chat functionality.

- POST `/chat/send`: Send a message. Requires a `token`, a `username` and a `message`. Returns `true` if the user is online; otherwise `false`. `{ "send" : true }`

- POST `/chat/poll`: Check the server, to see if any incoming messages are available. Requires a `token`. Returns an array of messages, which may be empty.
  Clients must poll the server frequently, to pick up their messages.

Sample communication

Here is a sample log of communication with the server. Comments in red are not part of the communication, but just explanations. For context: “Marble” is a cat, and “Sunnu” is a dog.

Client sends	Server replies
http://127.0.0.1:50001/ping	{ "ping" : true }
http://127.0.0.1:50001/user/register { "username" : "marble", "password" : "meow" }	{ "username" : "marble" }
http://127.0.0.1:8080/user/login { "username":"marble", "password":"meowww" *incorrect password* }	{ "Error" : "Invalid username or password" }
http://127.0.0.1:8080/user/login { "username":"marble", "password":"meow" }	{ "token" : "591B6FEE023F1E028AF88E73FB8B2C1A" }
http://127.0.0.1:50001/user/online { "token" : "591B6FEE023F1E028AF88E73FB8B2C1A", "username" : "sunnu" }	{ "online" : true }
http://127.0.0.1:50001/chat/send { "token" : "591B6FEE023F1E028AF88E73FB8B2C1A", "username" : "sunnu", "message" : "You are definitely a dog!" }	{ "send" : true }
http://127.0.0.1:50001/chat/poll { "token" : "591B6FEE023F1E028AF88E73FB8B2C1A" }	{ "messages" : [{ "message" : "How is life as a cat?", "username" : "sunnu" }] }
http://127.0.0.1:50001/user/logout { "token" : "591B6FEE023F1E028AF88E73FB8B2C1A" }	{ "logout" : true }

Student Projects

Project options

Project option 1: Extend the server

Add “chatrooms” to the server. Any user can define a chatroom. Any user can join a chatroom. Messages sent to the chatroom are sent to all chatroom-users who are online. The following API extensions are required:

- /chatrooms (GET) – list all Chatrooms
- /chatroom/create (POST) – create a chatroom
- /chatroom/join (POST) – join a chatroom
- /chatroom/leave (POST) – leave a chatroom
- /chatroom/delete (POST) – delete a chatroom (only possible for the user who created it)
- /chatroom/users (POST) – list the users of a chatroom
- /chatroom/send (POST) – send a message to the chatroom

Additionally, the /chat/poll command must be modified: It must return chatroom messages as well as private messages. Chatroom messages will contain the name of the chatroom, in addition to the username and the message.

Project option 2: Implement a client

Create a client application using of the following technologies: Java (JavaFX), Android (Jetpack Compose), or browser-based (React).

It must be possible to enter the server address and port. You can use javaprojects.ch:50001 as the default, but it must be possible to change this! Send a “ping” to confirm that the address is correct.

The client must support all API functions of the server listed in this document, except for the “debug functionalities”. Chats with different users must be separated – don’t mix them together!

Project options 1 + 2: Two groups working together

If one group wants to extend the server, it would make sense for a second group to implement a compatible client. The two projects can then be tested together.

If two groups choose this option, be sure to reference each other’s project in your documentation!

General project requirements

Each project must be managed in a Git repository, and the instructor must have read-permission (in GitLab that means “Reporter” privileges). All project code *and documentation* must be in Git at the end of the project.

Each team member must program a significant part of the project. Add your name in a comment above methods or classes that you have written. Note that your contributions will be visible in the Git history.

Project documentation

Title page: List the type of project you have chosen, and the names of all team members.

Usage: Explain how to clone, compile and run your project. *Be sure to test your project on a "fresh" computer, to ensure that these instructions work!*

Design: How is the code structured? How do the parts work together? Suppose that your reader has no time to study the code itself: Explain the most important ideas in your implementation. (1-2 pages of text, plus diagrams)

Using code from external sources

There is nothing wrong with using code snippets from external sources. However, the source must be referenced in a comment in the code. Using external code without referencing the source is plagiarism. For examples: see the methods "hash" and "bytesToHex" in class Account of the existing server.

Handing in the project

To hand in your project, just hand in a link to the project repository in Moodle. Only one person needs to hand in the project.

Helpful tools and references

Tools: Curl and Invoke-RestMethod

These commands provide command-line access to web resources. Under MacOS or Linux use `curl` (you may need to install the command). Under Windows[1] PowerShell use `Invoke-RestMethod`.

For this project, these commands are useful to send GET and POST commands to the server and see the replies.

GET commands

GET commands simply request a particular page (mapping) from the server. For example, to fetch a web page under MacOS or Linux, execute

```
curl javaprojects.ch
```

Under Windows, execute

```
Invoke-RestMethod javaprojects.ch
```

In both cases, you will be presented with the home page from that website.

For this project, to see the list of all registered users, we send a GET command to the mapping defined by the server:

```
curl http://127.0.0.1:8080/users
```

Under Windows

```
Invoke-RestMethod http://127.0.0.1:8080/users
```

Under Windows, this displays detailed information about the response, as well as the content of the reply (JSON-formatted text). Under MacOS/Linux, the command only displays the content of the reply, however, you can get information about the response by including the “verbose” option:

```
curl -v http://127.0.0.1:8080/users
```

Also under MacOS/Linux, you can format the JSON into a more readable format by piping it through the `json_pp` command (again, you may need to install this):

```
curl -v http://127.0.0.1:8080/users | json_pp
```

The resulting output looks like this:

```
{
   "users" : [
      "sunnu",
      "marble"
   ]
}
```

1*The `curl` command also exists in Windows, but does not work as described here.*

POST commands

More complex server commands require sending information to the server using a POST command. For example, we can register a new user, using the `/users/register` mapping. The following command is entered *all on one line*:

```
curl --header "Content-type: application/json"
http://127.0.0.1:8080/user/register
-d '{ "username":"Marble", "password":"meow" }'
```

Under Windows, again *all on one line*:

```
Invoke-RestMethod -Method POST -ContentType "application/json"
http://127.0.0.1:8080/user/register
-Body '{ "username":"Marble", "password":"meow" }
```

Note the following parts of this POST command:

- A header stating that the data will be in JSON format
- POST data attached using the `-d` (MacOS/Linux) or `-Body` (Windows) option
- The data is enclosed in single-quotes, to avoid a conflict with the double-quotes in JSON

The reply confirms the creation of a new user by returning the username:

```
{ "username" : "marble" }
```