CNT4007C - **Project 1**
02/18/2019
David Cabrera
dacabdi89@ufl.edu

NOTE: This PDF file was generated from mark-down

QUIZNET

USAGE

BUILDING

To build the solution you can either,

- `make all` to build both `qclient` and `qserver`
- `make qclient`
- `make qserver`

The `makefile` provides comprehensive recipes to run tests, compile in debug mode, or produce manual entries. To run in debug mode set `MODE=debug` before calling on `make`.

RUNNING

The applications can be run as per specifications.

Please note that both `qclient` and `qserver` have been implemented with POSIX command line options using `getopt` and `getopt_long`.
Use option `-h` on both executables for further usage details.

The server application provides ample levels of logging and verbosity.

Refer to the corresponding manual entries for details.

PROTOCOL SPECIFICATION

Before any request or response, the client must perform an application level handshake routing by which it gives notice to the server on the type of session (persistent vs non persistent) and other relevant information.

REQUESTS

GENERAL REQUEST SPECIFICATION

All *Requests* must adhere to the following syntactic specification

```
type length\n
body\n
```

Field	Syntax	Semantics	Optional
`type`	Only **one** character from a-z	Indicates the type of request	No, all requests must have a `type` field
`length`	A decimal nonnegative integer	Describes the length of the request's `body` in bytes	No, an empty request is expected to include a 0 length
`body`	Any number of ASCII characters	Request's body and payload	Yes, but 0 length must be indicated

More precisely,

```
request = type , ' ' , length, '\n' , (body)? '\n'
        ;
type    = a-z
        ;
length  = [0-9]+
        ;
body    = ? any sequence of chars ?
        ;
```

If the server knows how to handle the type of request, the request is properly formed, and is processed successfully, the server will respond according to specifications. Otherwise, an error response will be returned. If the error is due to an unrecognized type of request, the error type will be `UNKREQ`.

TYPES OF REQUESTS

POST

Upload a question to the server

```
p length\n
body\n
```

This type of request must include a `length` field and a `body` field containing,

```
tags\n
question-title\n.\n
choices.\n
(correct-answer)\n
```

1. `tags` field with zero or more comma separated tags.
2. `question-title` with a string of alphanumeric and white space characters terminated by the sequence `\n.\n`.
3. Any number of `choices` of the form `(choice-character) choice-text\n.\n` followed by an additional `\n.\n` that ends the section.
4. The correct answer `correct-answer` followed by an end of line.

Please observe that the server will enforce the following rules

1. A question must contain at least 2 choices.
2. Every choice must be of the format `(choice-character) choice-text\n.\n`.
3. The `choice-character` value is a unique a-z character, the choices must be in alphabetic order, and the first choice must be `a`.
4. The `correct-answer` provided must exist in the set of choices.

Expect an error response if any of this rules is violated. The error is likely to indicate in the extra field, what kind of violation took place.

DELETE

Request a deletion using type `d` and indicating the question's identifier in the body of the message.

```
d length\n
question-id\n
```

Expect an `NOTFND` error response if question doest not exist, or a `INVQID` if the the question id is not a numeric value.

GET

Request a question using type `g` and indicating the question's identifier in the body of the message.

```
g length\n
question-id\n
```

Expect an `NOTFND` error response if question doest not exist, or a `INVQID` if the the question id is not a numeric value.

GETR

Request a random question.

```
r 0\n
\n
```

Expect an `EMPTYQ` error response if the server has an empty quiz book.

CHECK

Check if a choice is the valid response to a question identified by its question id.

```
c length\n
question-id\n
choice\n
```

Expect a `NOTFND` error if question does not exist or a `CHNFND` if the question does not contain the provided choice.

KILL

Kill server process.

```
k 0\n
\n
```

RESPONSES

All *responses* will come in two types, OK responses indicating a successful request and ERROR responses along with information on the failure.

TYPES OF RESPONSES

OK

```
```

```
o length\n
body\n
```

The content fo the body of the response will depend on the particular request that unleashed the response.

ERROR

Error responses along with error codes and extra information.

```
e length\n
[number]\n
[symbol]\n
[message]\n.\n
[extra]\n.\n
```

no	symbol	message	extra
0	`UNKERR`	Unkown error	Exception message, if any
1	`NOTFND`	Question not found	`[question-id]` provided
2	`MALQUE`	Malformed question body	Exception message, if any
3	`CHNFND`	Choice does not exist in question	`[choice]` and `[question-id]`
4	`UNKREQ`	Server does not know how to handle request type	Received request type
5	`INVQID`	Invalid format for question id provided	Provided id
6	`EMPTYQ`	Empty quiz book	None
7	`REQTSZ`	Request type length must be one character	Type provided

IMPLEMENTATION

The entire solution was implemented in C++ in an OOP style. Wrappers interfaces, and data models were written around the networking routines and the data representation.

Objects `Socket` and `Host` provide wrappers with all types of operations around the file descriptors and hosts representations used by the operating system.

`QuizServer` is an abstraction of the entire server and takes in a host representation to determine where it is running and a socket object to work with. It takes care or calling methods that wrap around `bind`, `listen`,

and `accept`. The object uses a map keyed on characters to callables that handle each corresponding type of request and produce a `Response` object that is then serialized and sent back to the client.

`QuizServer` also takes a `QuizBook` object with a data model representation of the questions read and written to the question bank file (refer to the section 5 of the manual for file format information.)

On the other hand, the `QuizClient` object follows a similar pattern of an internal map keyed on characters and containing callables that are emplaced on initialization. This callables handle server responses and perform all kind of appropiate operations depending on the response.

All sorts of serializable and deserializable data representation models were written to abstract the string level manipulation, both for networking related objects such as requests and responses, or data related objects, such as tag collections, choices, or questions. Refer to the source files for more implementation details, including stream operators overloading and constructors that take in serialized representations of the objects and generate the deserialized representation.

This methodology allowed to standardize and defer the responsability of serializing and deserializing down to the data layer, abstracting control from the data representation.

Before a persistent session, or a non persistent as well, the client performs an application level handshaking with the server that negotiates the session parameters and puts the server on ready state. This negotiation at the moment takes care mainly of giving notice to the server of whether the client wants to establish a persistent or non persistent session.

The serialization to file (question bank) has been implemented on a writethrough fashion to prevent data loss if the server side crashes.

Customized exception objects have been provided for `ProtocolException` and `NetworkException`. These exception objects provide granularity in the catching of errors, allowing to distinguish between the two types of problems.

Both client and server are robust and can survive (graciously) the other party leaving the session. These endpoints will use protocol specifications to communicate each other exceptions and errors and handle them appropiately. Refer to the *Protocol Specification* session for more details.

ADDITIONAL FEATURES

Although the specifications did not require implementing both persistent and non persistent mode, the client takes a flag `-n` or `-nonpersistent` to allow non persistent mode.