ElizaJS - Long Term Memory for Eliza with Rudimentary Context Recognition

Francis Dippnall (17003003)

0000-00-00

Abstract

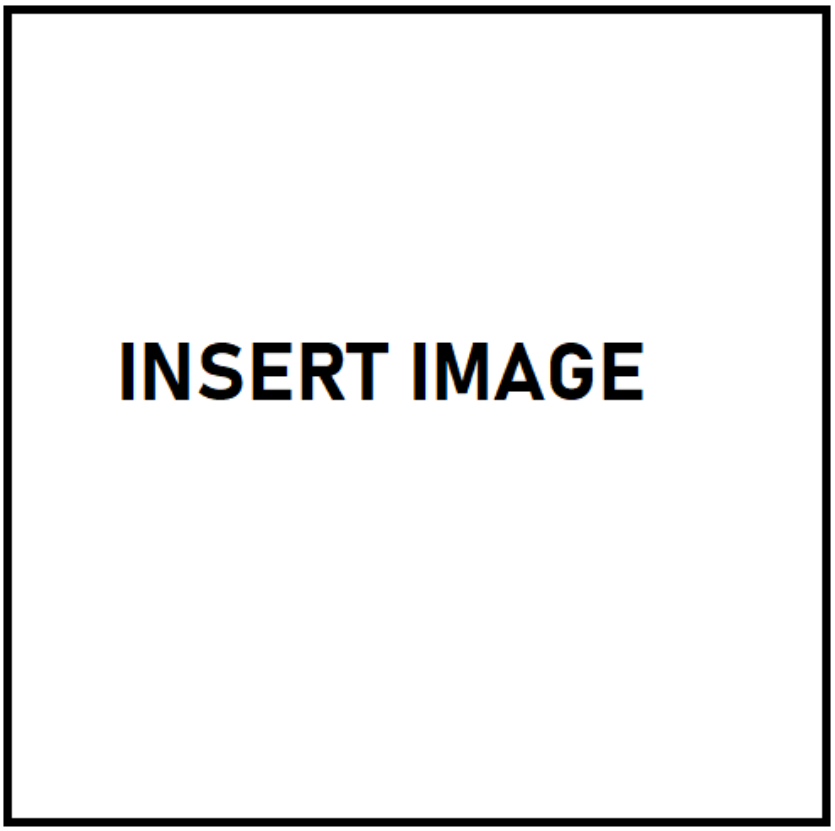

Figure 1: caption

1 Introduction

2 Literature Review

3 Design

3.1 Technology

This section describes and explains the main technologies to be used in the project. The is one subsection for each technology.

3.1.1 JavaScript

3.1.2 Node

3.1.3 Express

3.1.4 Socket.io

The `socket.io` Node package allows for reliable low-level communication between the server and client without the need for a page refresh which improves the user experience. `socket.io` supports transfer of many data structures which improves code flexibility to future changes.

3.1.5 MySQL

3.2 Memory

This section outlines the design of the long-term memory mechanism to be employed in the ElizaJS system.

3.2.1 Introduction

The original Eliza chatterbot had a rudimentary implementation of memory, which worked by saving facts as defined in the definition file as binary or ternary lexical statements in an array in memory. We can define "memory" in this context as some form of storage of lexical statements of this type which have some meaning in the real-world, that can later be used to influence future responses. This implementation can be referred to as "short-term" memory since the information is not retained after the termination of the program. To successfully implement a long-term memory solution, there must be a sense of fact persistence across multiple conversations.

3.2.2 Lexical Context

The ElizaJS system will have a basic context handler which should allow the user to use common predefined pronouns ("it", "he" etc.) to refer to the most recent item in the current lexical context. The core will track two contextual variables: Active Context Pronoun (ACP), and Active Context Noun (ACN) which will allow it to both translate user messages to a base lexical form (see 0.1.3), and use pronouns in its own responses (see 0.1.0). For this, the core must be able to recognise when:

- **A context is created or replaced**
  This is done when a new noun is introduced by the user. The current ACP and ACN are overwritten by the new information.

- **A context is destroyed**
  A rarer form of context switching where the context is dereferenced without replacing it with a new one. To avoid over-complication, This shall be assumed when the core receives a message that contains neither a valid fact containing a noun nor a valid pronoun. Examples of context-destroying messages include "Enjoy!" and "Well then.".

3.2.3 Fact Extraction

A "fact" will be used to mean an individual item of memory consisting of up to two lexical terms and an operator from a set of pre-defined lexical operators taken from the training corpus. To allow the core to remember facts they must be extracted from each user message. The process for which is explained in detail below.

1. **Search for operators**
   The first step is to find the highest-priority lexical operator in the user string. Starting with the message string, iterate over each word in the operator list in descending priority order and search for that word in the message. If no operators are found, the user string does not contain any facts and Eliza should revert to a Null Response. When an operator is found, split the user string at that point so that you have the characters before the operator, the operator itself, and the characters after the operator.

2. **Trim lexical terms**
   For each string, remove the portion of the string from the punctuation mark closest to the operator to the extreme end (the end furthest away from the operator) of the string.

3. **Apply context**
   Pronouns in the lexical terms must be replaced with the valid context information. If no context information can be found, the core must abandon the fact to avoid compromising memory integrity. For each lexical term, scan for a pronoun from the pre-defined list. First and second-person pronouns may be published unchanged, but if a third-person pronoun is found, check if the pronoun matches the current ACP. If there is no match, abort the fact creation process. Otherwise, replace the pronoun in the lexical term with the current ACN. Repeat until all pronouns in the string have been replaced.

After the completion of the steps above you are left with the pre-operator lexical term (`term1`), the `operator` itself, and the post-operator term (`term2`) which can be stored in the database along with the username of the user providing the fact (`negotiator`) and the date (`fact date`). Note that due to the subject-verb-object[1] nature of English, the `term1` LS can be assumed to be the *subject* and `term2` the *object*.

[1] https://en.wikipedia.org/wiki/Subject%E2%80%93verb%E2%80%93object

3.2.4 Memory Querying

The final step of the memory architecture of the system will be the querying process. This will allow the core to respond to user's statements with corresponding factual data. If no memory entry can be found for the queried item, the core can respond with a question, or create a new entry if a fact is supplied by the user message. The following steps are performed after the completion of the Fact Extraction process.

- **Query the fact base**
  If the user string contains a valid fact, query the datastore for facts whose *subject* contains or is contained in the extracted user fact. Otherwise:

- **Perform user-requested query**
  If the user string contains a match for one phrase in the `keyphrase` list (see Fig 0.), take the text following the keyword in the user string as the *subject* and query the datastore for any fact with an exact match for that subject. Otherwise:

- **Use null response**
  No information can be found about the user string, so respond with a null response.

3.3 Data Flow

Before the specific data structure can be designed, it is necessary to plan the data flow of the system. I decided to go with a 3-tier architecture as it allows off-loading of processing to the client which speeds up the server. However, this means the Eliza core is situated on the client side which could lead to some security vulnerabilities that should be considered in the implementation phase. The DFD below outlines the macrostructure of the system.

3.3.1 Data Flow Diagram (DFD)

The general idea is that the user submits their User Data (username) and this is stored in the knowledge base.[2] Then the user can send messages to the core, similar to the original Eliza, however the core will scan each message for keywords and query the knowledge base for data on that keyword.
The server will act as the middle-man between the client and the knowledge base for security and verification purposes, and will pass the results of these queries back to the core which can then formulate a response based off of the information.
This process repeats until the user quits the chat session, at which point the client will submit a log to the server which will be stored in another data structure (the Evaluation subsystem, see below).

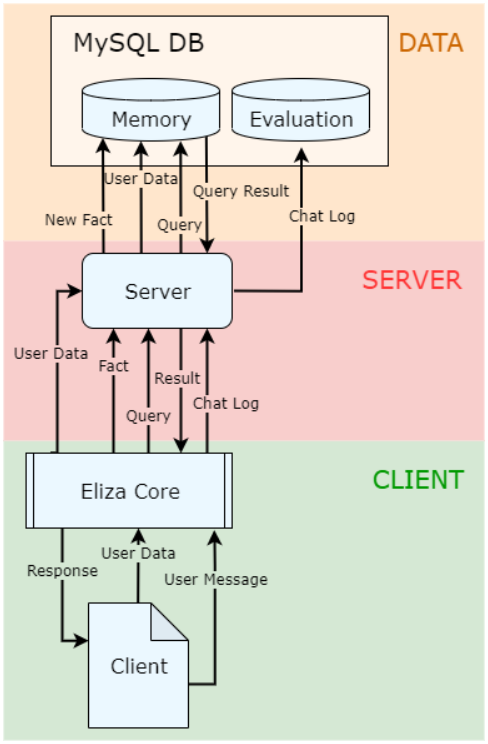

Figure 2: Data Flow Diagram of the ElizaJS system.

3.3.2 Two Types of Data

Looking at the DFD it is clear that there are two distinct types of data being transmitted: conversation logs, and facts to be stored in the long-term memory. These two aspects should be completely seperate in the data structure as the core should be able to access memory data at will but raw conversation data is solely for evaluation. As such, it is imperative that the database contains two isolated subsystems: one for storing evaluation data which will be referred to as *Evaluation* and another for the knowledge base named *Memory*. The next section will cover the design of each of these subsystems and explain their purpose in more detail.

[2]The username is also stored on the Evaluation subsystem.

3.3.3 Socket.IO Plan

The `socket.io` system uses a message-based communication structure. The server can send messages to any client and the clients can send messages to the server. It is standard practice to plan the communications in table format so that an understanding about the use of `socket.io` in this project can be reached.

Sender	Message Name	Send When?	Data Sent	On Receipt
Client	`sign on`	Client submits the user data form.	`username: string` `password: string`	Server validates and verifies the client data, and responds with a `sign on result` message.

Server	`sign on result`	Server has processed the client data.	`success: boolean`
			`new user: boolean?`
			`reason: string?`
				Client displays the chat container if `success` is true, else display the `reason` for failure.

Client	`query factbase`	Client has extracted a term and wishes to query the factbase.	`query: string`	Server performs the query and responds with a `query result` message.

Server	`query result`	Server has completed the query.	`success: boolean`
			`results: object?`
			`reason: string?`
				Client uses the query `result` to compile an Eliza response.

Client	`new fact`	Client has extracted a fact from the user and wants to push it to the factbase.	`fact: object`	Server validates the new fact and inserts it into the database.
Client	`conversation log`	Client has finished the conversation.	`log: object`	The server serialises and stores the conversation `log` in the datastore.

Server	`log received`	Server has logged the conversation to the datastore.	`success: boolean`
			`results: object?`
			`reason: string?`
				The client displays the evaluation page.

3.4 Database Design

The Entity Relationship Diagram (ERD) of the MySQL database used in the project is shown if the figure below. It shows the two disconnected subsystems and four tables. Each table name is prefixed by "Eliza" to distinguish these tables from the other ones in the database used.

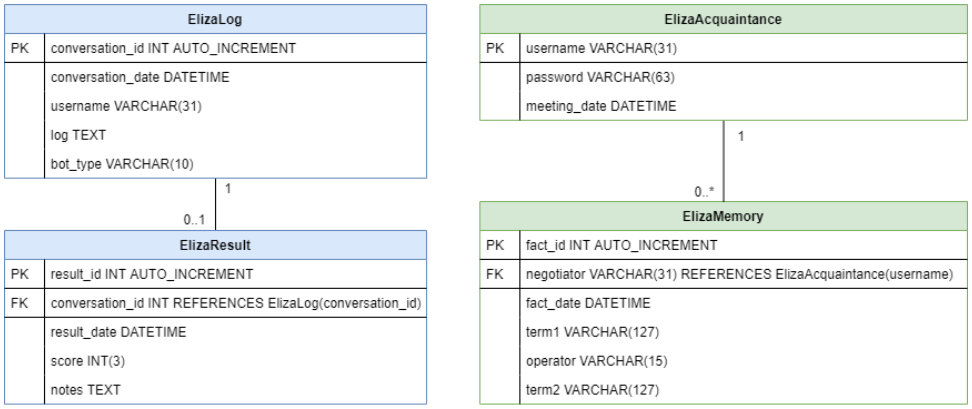

Figure 3: Entity Relationship Diagram. Evaluation is in blue, Memory in green.

3.4.1 Memory Subsystem

The Memory system is the long-term memory structure of ElizaJS. It contains two tables:

- `ElizaAcquaintance` - A store of the users that Eliza has spoken with or is currently speaking to. Allows persistent conversations by providing a password.
- `ElizaMemory` - the actual fact-base for the core. Each entry contains a reference to the user that provided the fact, the date-time when it was obtained; in addition to the fact itself, which is stored as two lexical terms and an operator.

There is more information on the Memory process in section 3.2.

3.4.2 Evaluation Subsystem

The Evaluation subsystem is used to collect information for analysis purposes. This data is not relevant to the chatterbot core and is therefore only accessible via an admin view. This system uses two tables:

- `ElizaLog` - stores full text logs of every conversation on record paired with the submission data and username. The `log` field is delimited by a double-slash ("`//`") meaning this string must be removed if present in any message. The `bot type` field shows which version of Eliza this conversation is with.
- `ElizaResult` - stores user evaluation data. As an optional addition to the conversation log, users can provide feedback notes and a score which will be useful in the Evaluation phase.

4 Implementation

5 Testing

6 Evaluation

7 Conclusion

8 References

9 Appendix