Official Manual for the

Ferris Chatbot System

Revision 1 for Version 1.0
August 14, 2025

Summary

The Ferris Chatbot System is an AI agent that answers user questions about the college and Canvas courses. Its three logically grouped components are the front-end user interface(s), the web service, and the AI application. User-interfaces (UIs) provide simplified ways for users to access the web service. The web service will take prompts from the UIs through HTTP, runs them through the AI application to generate a response, and return the response to the UI. The current implementation for the AI application uses the OpenAI Assistant service, which a Java-based Spring Boot web service communicates with through an HTTP-based API.

Syntax

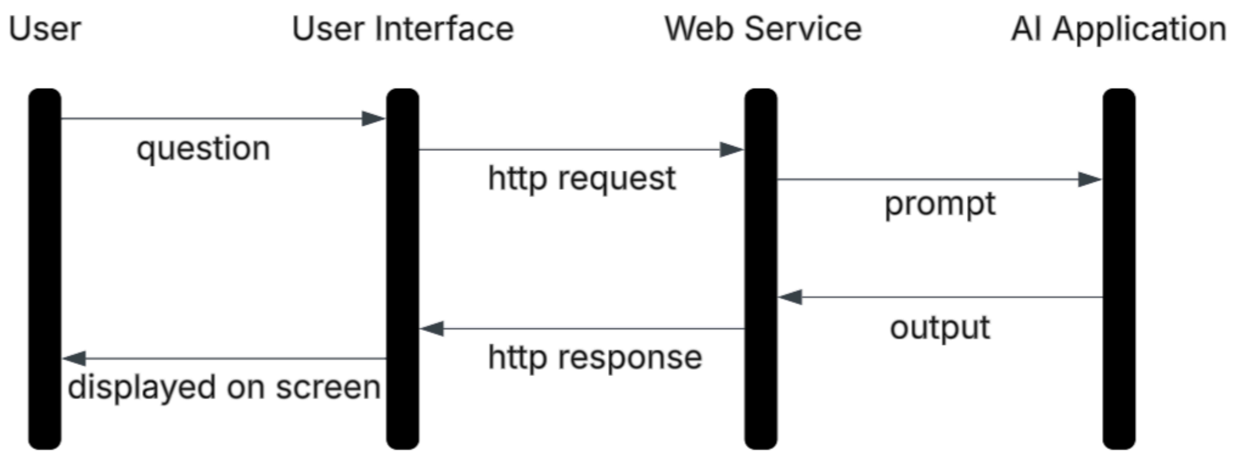
1. Non-constant factors such as web service urls, thread ids, and access keys are surrounded by square brackets, for example “[web service url]/thread/[thread id]” means that you would replace [web service url] with the actual url for the chatbot web service, and [thread id] with the id of the desired thread, while keeping “/thread/” unchanged.

Introduction

The Ferris Chatbot System (referred to from here on as “the system”) is an AI solution designed to answer user questions about Ferris State University or about classes held on the Canvas LMS. The system encompasses multiple different programs, including front-end user interfaces in addition to back-end web services and machine-learning applications. The system includes all components required to facilitate user interaction with a chatbot, but not auxiliary programs such as ones used for data scraping and management. This document will explain how each component of the system works, along with how to use and maintain them.

Architecture

The system architecture consists of three logical components; the front-end user interface(s) (UI), the web service, and the AI application. The UIs provide users with an easy way to interact with the web service. The web service will take user queries as HTTP requests from the UI, run it through an AI application to generate a response to the question, and return that response to the UI. There may be multiple different UIs deployed, however they must all communicate with the web service through the same API. The AI application refers to any AI solution capable of generating responses to questions that the web service can interact with. The AI application may be internal to the web service, it may be hosted on another in-house application that the web service connects to, or it may be a third-party service that the web service connects to.

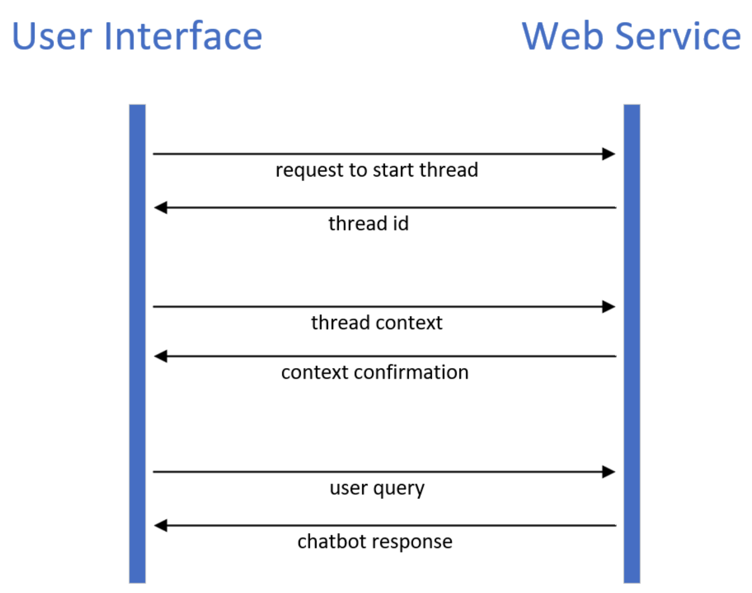


User Interface

The UI refers to a user-facing application that is capable of communicating with the web service through HTTP, such as a web page. There may be multiple different UIs deployed at any given time, however they must all connect to the web service through the same API. Anything capable of communicating with the web service through its RESTful API can serve as a UI.

Communication Flow

In order to communicate with the chatbot through the web service, a UI must **first** send a message to the web service to start a thread and store the thread id from the response. A thread is a collection of messages between a user and chatbot, which represents a conversation that the chatbot can access when generating its responses. **Then**, the UI may optionally send a session context message to the thread, which contains data such as the date and time that the conversation started. After starting the thread and optionally adding context, the UI may **finally** send user queries to the web service, which will then return the response. The first two requests, the thread startup and context addition, only needed to be completed once to start a conversation with the chatbot. Once the conversation has been started, many user queries may be sent to the chatbot.



Starting a Thread

In order to access threads for generating responses, the UI must first tell the web service to start a thread, then it must store the returned thread id, then it may send session context such as current date and time to the thread. This means you must use the Create Thread endpoint of the web service, then the Add Context endpoint.

Sending Prompts

Once a thread has been started, you access it using the thread id and send prompts to it. This is done using the Message Thread endpoint of the web service.

Error Handling

When the web service catches an error, it will give a JSON response containing information about the error. The response contains a “status” property which contains the HTTP status code, a “message” property that contains a short, user-oriented description of the error, and a “detail” property that contains a longer description of the error intended for admins. In the current implementation, the detail will contain the full JSON response (converted to string) that OpenAI (AI Application section) sent to the web service, which will describe the error encountered by the OpenAI service.

Web Service Error

Body: JSON
- status
- message
- detail

  Example:

```
{
   "status": 500,
   "message": "OpenAI key not configured on server.",
   "detail": "An OpenAI access key must be configured in the server's environment
variables to use this endpoint."
}
```

Design Recommendations

1. In order to reduce unnecessary traffic, threads should only be started when the user submits their first query. When this happens, the UI should first start the thread and store the id, then add context to the thread, then send the user's query as a prompt request. This full process should happen whenever there is no valid thread id being stored in the UI. If a valid thread is available, then the UI should only submit prompt requests to the web service.
2. Because there is no way to access the full conversation from the web service API, the entire conversation should be stored locally to the UI so that the user can see the full conversation. When a query is submitted by a user, it should be stored and displayed, then the response should be stored and displayed when it is received.
3. All successful responses are formatted as simple strings that may be directly stored or displayed to the user. Failed responses are typically formatted as JSON that must be parsed to access its contents.

Web Service

The web service is a centralized server that takes HTTP requests from UIs, processes them using an AI application, and returns the response. The current implementation of the web service is a Spring Boot service built in Java.

Endpoints

The endpoints of the web service are defined within "src/main/java/com/chatservice/chatbot_service/controller/Controller.java".

Hello Endpoint

Description: Displays a response to confirm successful connection to the service.

Destination: [web service url]/chatbot
Type: GET
Headers: none
Body: none

Response: (string) confirmation message

Example: “Hello, this is the chatbot web service!”

Prompt Chatbot Endpoint

Description: Used for simple AIs that cannot remember a conversation. Each prompt in this interface is a simple, one-off communication with a language model.

Destination: [web service url]/chatbot
Type: POST
Headers:

- Accept: application/json
- Content-Type: application/json

Body: JSON

- prompt (no specific format required)
  Example:
  {"prompt": "hello"}

Response: (string) chatbot response

Example: “Hello! How can I assist you today?”

Create Thread Endpoint

Description: Create a thread for the chatbot and return the id of that thread. This is the first step in holding a conversation with a chatbot.

Destination: [web service url]/chatbot/thread
Type: GET
Headers:

- Accept: application/json
- Content-Type: application/json

Body: none

Response: (string) thread id

Example: “thread_thh2LZWs3vgwYmYjYsEppTum”

Add Context Endpoint

Description: Add optional session-specific information to a thread. Should be used after creating a thread and before prompting it.

Destination: [web service url]/chatbot/thread/context/[thread id]
Type: POST
Headers:

- Accept: application/json
- Content-Type: application/json

Body: JSON

- date (no specific format required)
- time (no specific format required)

  Example:

  ```
  {
    "date": "August 5, 2025",
    "time": "3:33pm"
  }
  ```

Response: (string) confirmation message

Example: “Context added successfully.”

Message Thread Endpoint

Description: Send a user query to the chatbot and return the chatbot’s response. Should be used after creating a thread and (optionally) adding context to it.

Destination: [web service url]/chatbot/thread/[thread id]
Type: POST
Headers:

- Accept: application/json
- Content-Type: application/json

Body: JSON

- prompt (no specific format required)

  Example:

  ```
  {"prompt": "what is the date?"}
  ```

Response: (string) chatbot response

Example: “The current date is August 5, 2025. How can I assist you today?”

Latest Message Endpoint

Description: Retrieve the latest message from a thread. While this feature is functional and useful for testing, it is not recommended to be used in a UI.

Destination: [web service url]/chatbot/thread/[thread id]
Type: GET
Headers:

- Accept: application/json
- Content-Type: application/json

Body: none

Response: (string) chatbot response

Example: “The current date is August 5, 2025. How can I assist you today?”

Configuration

The web service requires some environment variables depending on the features being used. An error will be thrown if you attempt to use an endpoint without having the required environment variables configured. The variables must be named exactly as shown in the “variable” section.

OpenAI API Key

A valid API key is required to use all OpenAI services.
Required for endpoints: all.

Variable: openai.key
Description: API keys give access to OpenAI services and charge them to the OpenAI account it was created for. Ensure that there are enough credits in that account. Do not publish API keys to any public platforms, as they may be used by strangers to pay for API usage. API keys and credits may be managed from OpenAI account settings.
Example value: “sk-proj-” followed by a large number of random characters.

Response Variables

The response variables are required to use the simple one-off prompts to OpenAI models.
Required for endpoints: Prompt Chatbot.

Variable: openai.responseEndpoint
Description: The url of the OpenAI API response endpoint, which is used to send single prompts to language models.
Example value: “https://api.openai.com/v1/chat/completions”

Variable: openai.responseModel
Description: The model that should be used to complete OpenAI responses. This is only for the one-off prompts, assistants have their own model configuration.
Example value: “gpt-4.1-mini”

Thread Variables

The response variables are required to use OpenAI assistant features that are required to hold conversations with a chatbot.
Required for endpoints: Create Thread, Add Context, Message Thread, Latest Message.

Variable: openai.threadEndpoint
Description: The url of the OpenAI API thread endpoint, which is used to manage conversations with the assistant.
Example value: “https://api.openai.com/v1/threads”

Variable: openai.assistantId
Description: The assistant that should be used to generate responses to thread messages. The assistant may be created and configured from the OpenAI website or through the OpenAI API. The web service does not manage assistants, and currently only allows for a single assistant that is used for every conversation.
Example value: “asst_0dj1n1Lq3LT6IIxvJ9jUve9l”

Interval Variables

The interval variables are required to configure timers that are used to check external processes at a regular interval. Currently the web service has no way of knowing when an assistant has finished generating a response other than to make regular requests to OpenAI to check the status.
Required for endpoints: Message Thread.

Variable: openai.runIntervalSeconds
Description: The amount of seconds that the web service will wait between every check. This must be an integer.
Example value: 1

Variable: openai.runMaxSeconds
Description: The amount of seconds that the web service will wait for the external process to finish before throwing a timeout error.
Example value: 30

AI Application Interface

The web service is designed to be able to easily switch AI applications, so contains two interfaces for AI applications; the ModelConnector interface that is used for simple AIs that cannot store or remember previous prompts, and the ChatbotConnector interface that is used for more complex AIs that are able to use previous messages in the conversation to generate their responses. They are defined within “src/main/java/com/chatservice/chatbot_service/chatbot”.

ModelConnector:

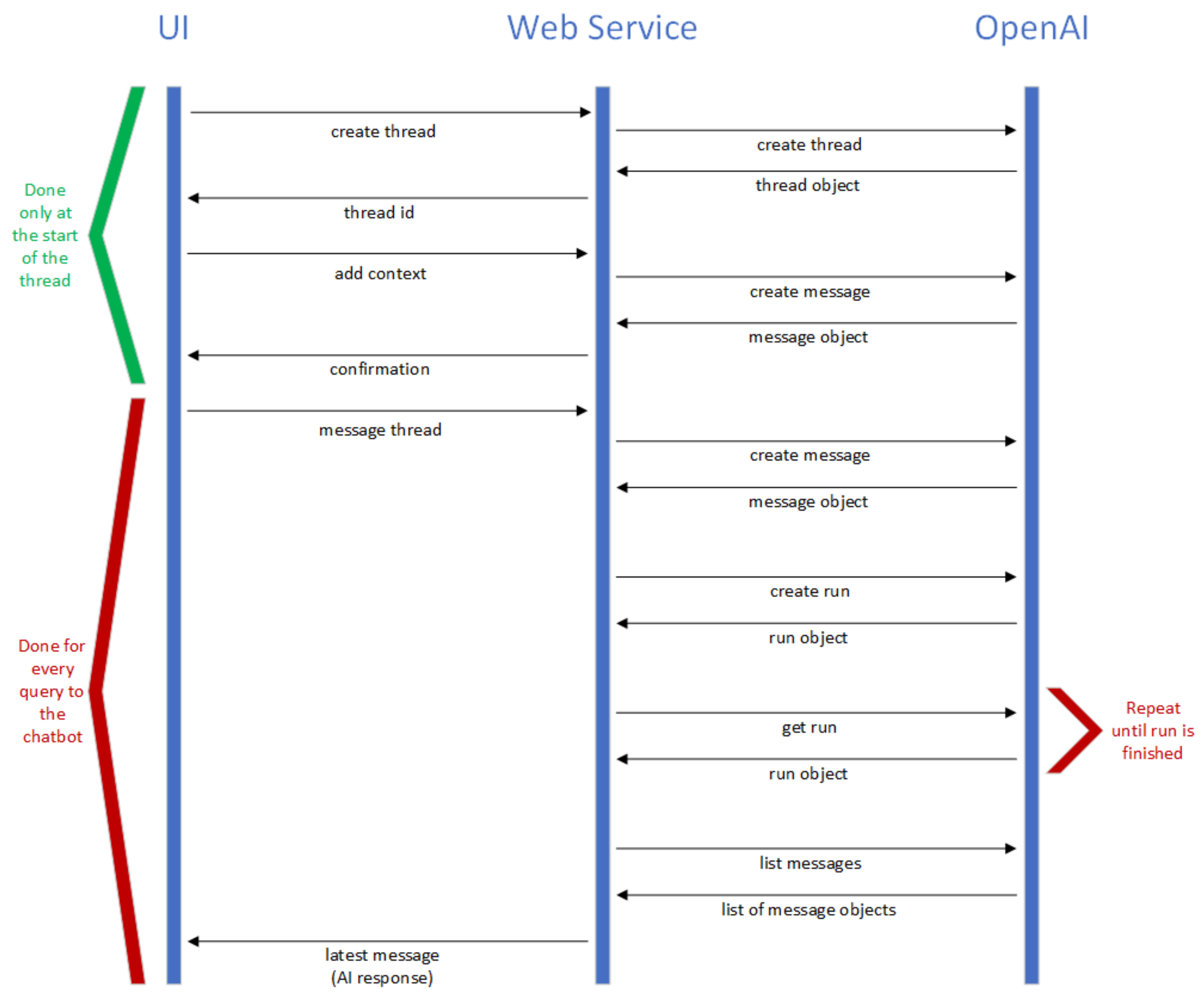
1. **Prompt:** take a prompt as a string, run it through a model, and return the response.

ChatbotConnector:

1. **CreateThread:** start a conversation with the chatbot and return an id for it.
2. **AddContextToThread:** take a message containing context as a string, along with a thread id, and attach that context to the conversation with the AI.
3. **PromptThread:** take a message containing a prompt as a string, along with a thread id, and run it through the model along with the rest of the conversation to generate a response, then return that response as a string.
4. **CreateThread:** start a conversation with the chatbot and return an id for it.

OpenAI Connector

The current implementation of the AI application is a connection to the OpenAI API, which provides AI services. OpenAI provides functionality for both simple one-off prompting and more advanced AI assistants that can hold conversations, so the OpenAI Connector implements both AI application interfaces, though only the chatbot interface is currently accessible through endpoints. The following diagram displays the entire flow of the chatbot system when using the OpenAI connector as the AI Application:



AI Application

The AI application is a program that takes prompts and outputs a response to them. This could be anything from code internal to the web server, an in-house server, or a third-party web service. The current implementation of the system uses the OpenAI API service as the AI application, which the web service connects to through HTTP using its OpenAI Connector.

OpenAI Responses

The Response feature of OpenAI allows users to submit a single prompt and returns a LLM-generated response. It only takes a single API request to generate a response.

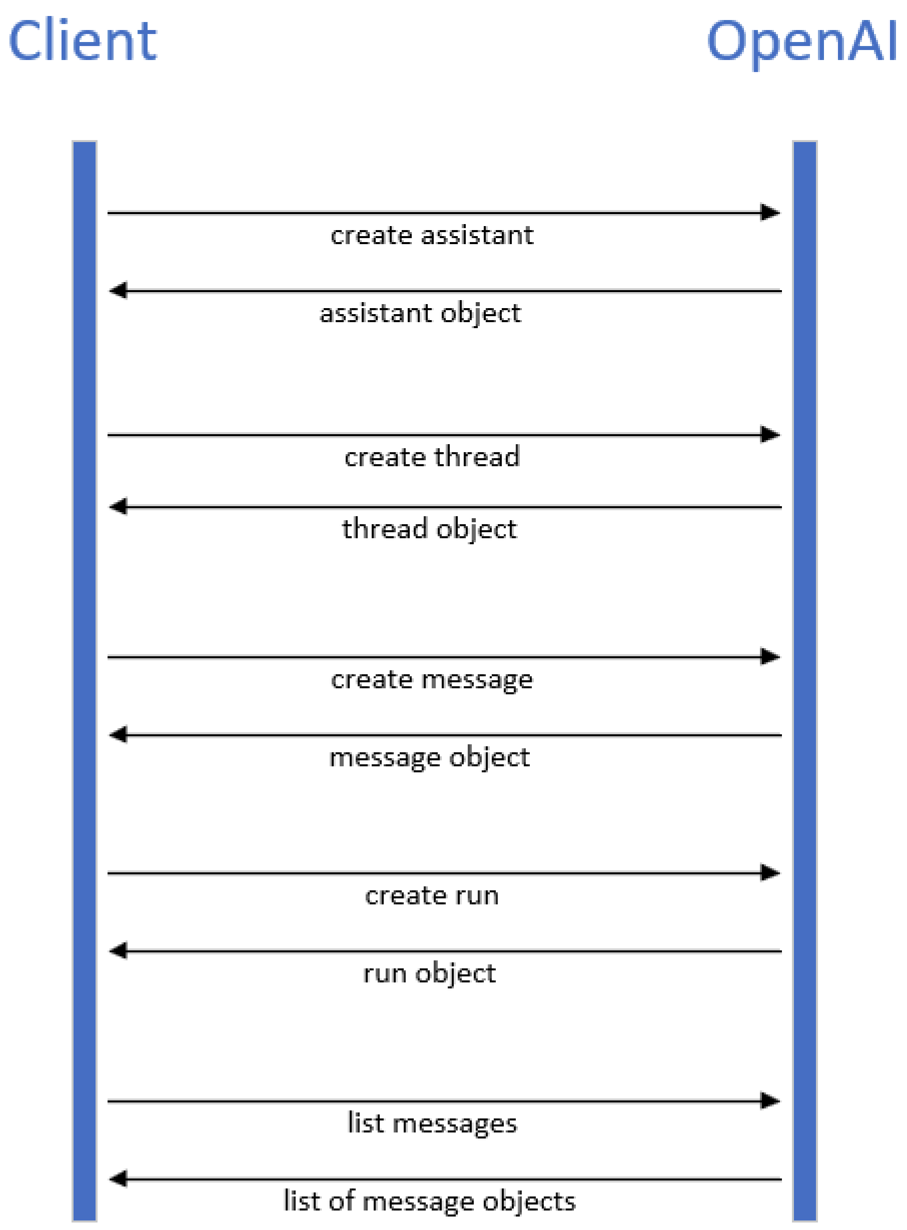
OpenAI Assistants

The Assistant feature of OpenAI provides a language model platform that can take special instructions, remember multiple different conversations (stored in threads), and access external files when generating a response. While basic OpenAI models may be accessed through a single request, assistants require a more complicated approach. Refer to the documentation and API reference at the OpenAI website for more detailed information than this manual provides.

Assistant Use

There are numerous steps to use an OpenAI assistant that are encapsulated by the web service's OpenAI Connector. Step 1 only needs to be done a single time. Step 2 must be done once at the start of every conversation. Steps 3-6 are required for every query to the chatbot.

1. Create an assistant using either the OpenAI website or the API.
2. Start a new thread to store the conversation. Note that there is no direct connection between assistants and thread, OpenAI "runs" are used to have assistants generate a new message onto the thread.
3. Add messages to the thread. The system can currently add a specialized context message describing session-specific information to the thread, which is labelled as being from the assistant rather than the user. Afterwards, messages from the user are added to the thread.
4. Run an assistant on the thread. Assistants do not automatically generate responses to threads, in order to generate a response with an assistant you must create a run. Runs represent an order to an assistant to read a thread, then generate a new message to the end of it. Creating a run will return it as a JSON object, store the run id for step 5.
5. Check the run until it is finished. Just like messages, threads, and assistants, runs can be accessed as JSON objects from the OpenAI API. When a run is created, it will immediately return a run object as JSON, which includes the run id and the run status. The run is not going to be complete immediately, as the assistant needs time to generate the response. The run id must be used to continually access the run from OpenAI and keep checking until the run status represents a final value such as "completed" or "failed".
6. Retrieve the response. There is no way to directly access the message generated from a run, instead you must retrieve all messages from the thread, and extract the latest one from the top. The actual text of the message can be found in the "value" property, which is nested inside of the messages "content" property. Once you extract the text from the latest message (which has just been generated by an assistant due to a run order), this value is the end product that should be displayed to the user.



OpenAI Notes

1. All objects in OpenAI, such as assistants, runs, threads, and responses, seem to be stored indefinitely in the OpenAI cloud. Most of these objects may only be directly accessed through the API, so a specialized program will be required to delete unused data. This cleanup functionality is out of scope for the chatbot system.