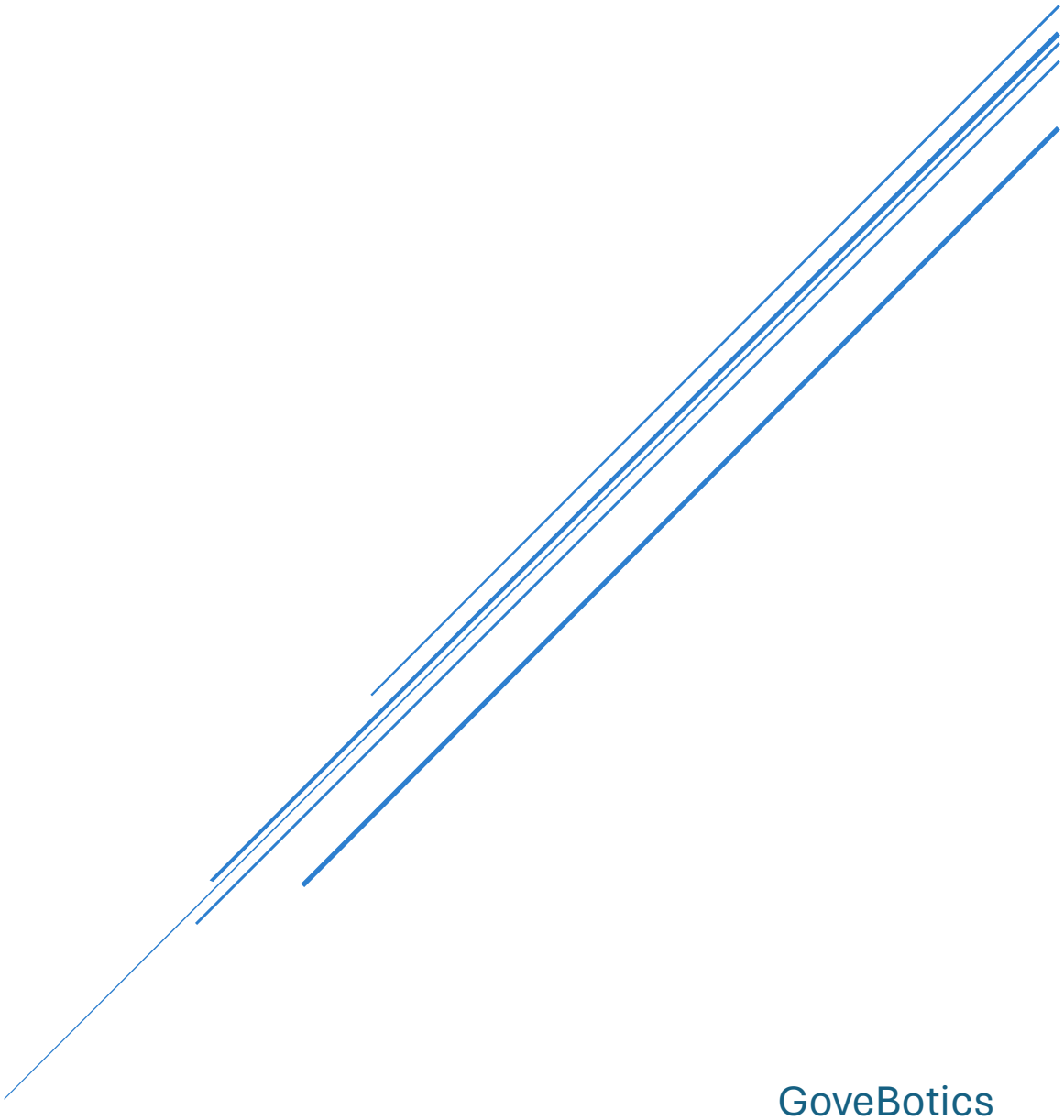
GEPPETTO SERVER

Conversational Documents



GoveBotics

Table of Contents

Introduction

The Geppetto Message Format is designed to facilitate language-independent communication between client applications and both local and cloud-based Large Language Models (LLMs) via HTTP. This framework aims to simplify the message definition process, enabling efficient interaction with a backend LLM. The core objective of this document is to delineate the request and response message formats utilized in such communications.

There are two primary message types within this system: LLM Query and LLM Admin.

- The LLM Query is utilized for transmitting messages to the backend Geppetto server. For detailed information on the message format, refer to the section titled "LLM Query."

- The LLM Admin is dedicated to configuring the backend Geppetto server and offers several Server Administration functions, including Document Ingestion, LLM Configuration, and Basic Token Usage.

This introduction sets the stage for understanding how Geppetto facilitates seamless interaction between client applications and LLMs, ensuring a clear, concise, and comprehensive guide to its message exchange protocol.

To seek clarity in how the messaging work, it his strongly recommended that the sample code provided be viewed as a source of clarification.

LLM Query Message

The Query message definition is crafted to enable dynamic parameter setting by the client, enhancing the flexibility and precision of interactions. By allowing client applications to customize parameters for each message sent to the LLM, it significantly enriches the quality and relevance of responses. This dynamic parameter setting facilitates a more tailored and effective communication, ensuring that the responses from the LLM are more aligned with the specific needs and context of each request.

HTTP is utilized as the transport protocol for a RESTful API implementation. The format for constructing Query requests is detailed below:

Header Parameters:
accept: application/json
Note: Specifies the media type the client expects to receive.

Content-Type: application/json
Note: Indicates the media type of the request body being sent.
Authorization: Bearer ${GOVBOTIC_API_TOKEN}
Note: Provides the authentication scheme and the necessary access token.

JSON Payload:

The request body should be structured in JSON format with the following attributes:

```
json
{
  "persona": "examplePersona",
  "uid": "exampleUID",
  "message": "Health Benefits of Lemons",
  "document": "LemonPeel/LemmonPeels",
  "context": "What is the Purpose of this manual?",
  "maxTokens": 100,
  "temperature": 0.5,
  "topP": 0.9,
  "stopSequences": ["\n"],
  "sessionId": "exampleUID",
  "language": "en-US",
  "requestType": "query"
}
```

Explanation of JSON Payload Fields:

persona:
Identifier for the simulated user persona.

uid:
Unique identifier for the user or session.

message:
The specific question or message to be analyzed.

document:
Reference to the specific document or data being queried.

context: Additional context for the query.

maxTokens:
The maximum number of tokens the response should contain.

temperature:
A measure of the randomness in response generation, with 0.5 indicating a balance.

topP:
The probability threshold for selecting tokens, contributing to the diversity of the response.

stopSequences:
An array of sequences that signal the end of the response generation.

sessionId:
A duplicate of the uid, likely used for session tracking.

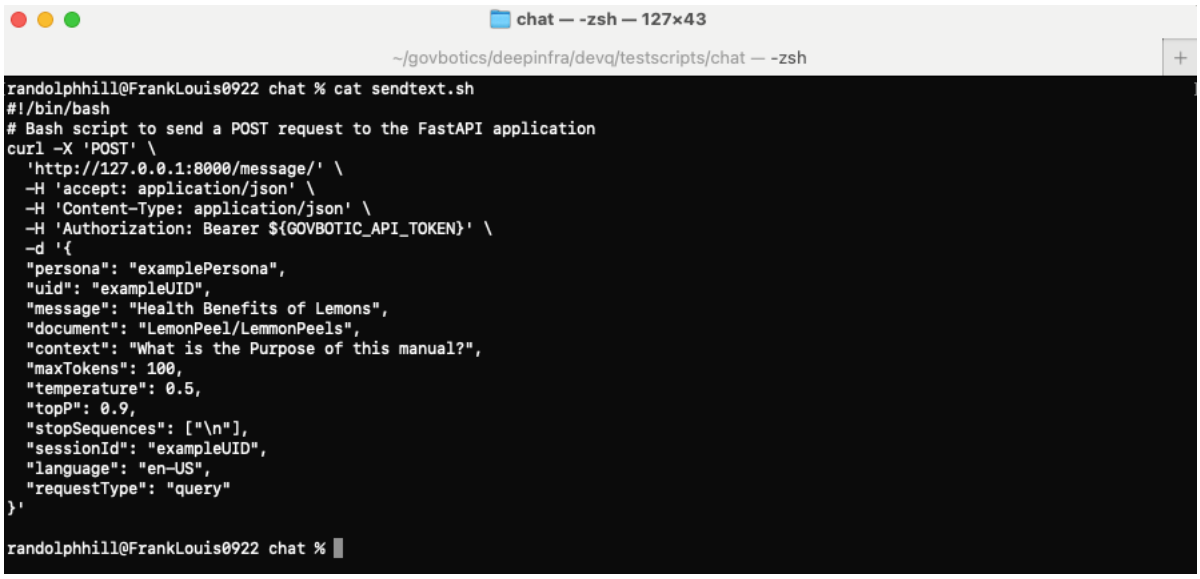
language:
Designates the request and response language, here as enUS.

requestType:
The nature of the request, here specified as query.

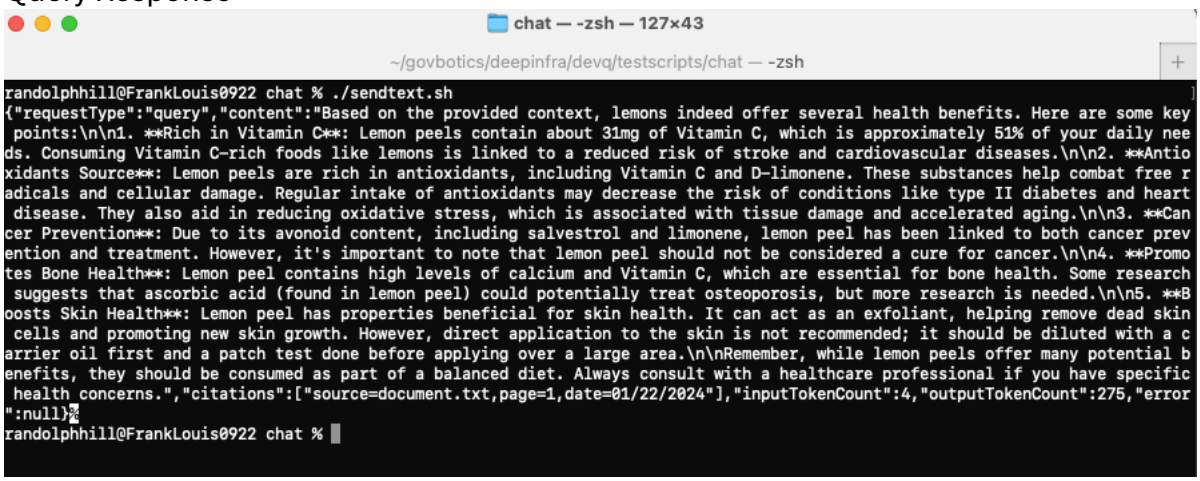
Example of Chat Message to LLM

```
curl -X 'POST' \
  'http://127.0.0.1:8000/message/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer ${GOVBOTIC_API_TOKEN}' \
  -d '{
  "persona": "examplePersona",
  "uid": "exampleUID",
  "message": "Health Benefits of Lemons",
  "document": "LemonPeel/LemmonPeels",
  "context": "What is the Purpose of this manual?",
  "maxTokens": 100,
  "temperature": 0.5,
  "topP": 0.9,
  "stopSequences": ["\n"],
  "sessionId": "exampleUID",
  "language": "en-US",
  "requestType": "query"
}'
```

Curl Script



Query Response



OpenAPI/Swagger Scheme

The OpenAPI/Swagger scheme can be found in the file llmquerymsg.yaml

Sample Implementation

Sample implementation in python, javascript and curl can be found in the directory samplecode.

Conversation Response Message Format

The ConversationResponse JSON structure is designed to serve as a standardized communication format between conversational AI systems or Language Models (LLM) and their clients. It efficiently conveys responses to user inquiries, encapsulating everything from the response content to error handling in a structured manner. This ensures a clear, coherent, and comprehensive delivery of information, making interactions with AI as informative and effective as possible. The design of this structure supports a broad range of functionalities, including error reporting and citation referencing, thereby facilitating a more transparent, reliable, and user-friendly experience in conversational AI applications.

The `ConversationResponse` class in a RESTful API implementation using HTTP includes the following parameters:

requestType:

Specifies the type of request (e.g., 'query').

content:

The content generated in response to the request.citations: A list of citations in a structured string format, defaulting to an empty list.

inputTokenCount:

The number of tokens in the input query.

outputTokenCount:

The number of tokens in the generated response.

error:

Optional details about any error that occurred during processing, encapsulating error specifics.

Example Response HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: <length_of_the_body>

{
  "requestType": "query",
  "content": "This is the generated response to the input query.",
  "citations": ["Citation 1", "Citation 2"],
  "inputTokenCount": 10,
  "outputTokenCount": 8,
  "error": null
}
```

List of Directories and Documents

The Geppetto server employs the concept of virtual file drawers with folders. This structure uses the `GOVBOTIC_INGESTION_STORAGE` environment variable as the root for storing documents. Each document has its own folder (acting as a virtual "File Drawer") named after the document, containing the document itself in PDF format and a subdirectory for ChromaDB. Here's how it looks:

```
GOVBOTIC_INGESTION_STORAGE (root directory)
└── toplevel (File Drawer)
    └── document-name (Folder)
        ├── document-name.pdf
        └── Chromadb (Folder)
```

GOVBOTIC_INGESTION_STORAGE (root directory)
└── toplevel (File Drawer)
    └── document-name (Folder)
        ├── document-name.pdf
        └── Chromadb (Folder)

In this structure:
GOVBOTIC_INGESTION_STORAGE
The root directory or the environment variable set to point to the location where all documents are stored.
Toplevel
Main "File Drawer" under which all document folders are organized.
-document-namerepresents a unique document, containing the document file itself (`document-name.pdf`) and a Chromadb folder, which might be used for storing related database or metadata files.

Listing File Drawers and Documents

The http header and payload format to list the documents in the file follows the same format as the query with one exception the query is set to requestType": "listfiles".
Refer to section LLM Query Message section for the HTTP header and payload structure.

List Curl Script Example

```
#!/bin/bash
# Define variables
GOVBOTIC_API_BASE_URL=http://127.0.0.1:8000
GOVBOTIC_API_TOKEN="FIFTOKENFORMAT"
ENDPOINT="/listfiledrawers/"

# JSON payload
REQUEST_DATA='{"requestType": "listfiles", "dataOperands": "YourDataOperands", "fileNames": ["file1.txt", "file2.txt"]}'

# cURL command
echo ""
echo ""
curl -X POST "${GOVBOTIC_API_BASE_URL}${ENDPOINT}" \
     -H "Authorization: Bearer ${GOVBOTIC_API_TOKEN}" \
     -H "Content-Type: application/json" \
     -d "${REQUEST_DATA}"
```

JavaScript Code

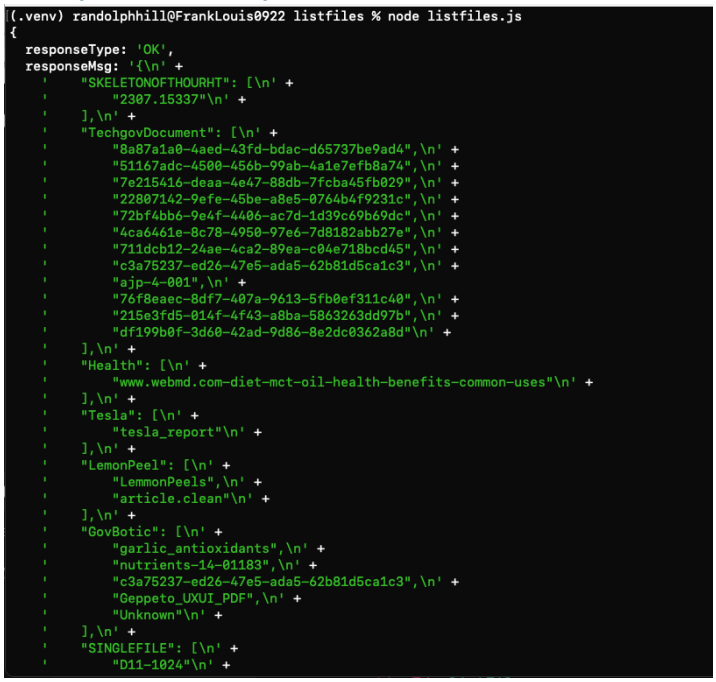
```
// Define variables
const govboticApiBaseUrl = 'http://127.0.0.1:8000';
const govboticApiToken = 'FIFTOKENFORMAT';
const endpoint = '/listfiledrawers/';

// JSON payload
const requestData = {
  requestType: "listfiles",
  dataOperands: "YourDataOperands",
  fileNames: ["file1.txt", "file2.txt"]
};

// Fetch options
const fetchOptions = {
  method: 'POST',
  headers: {
    'Authorization': `Bearer ${govboticApiToken}`,
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(requestData)
};

// Execute the HTTP request
fetch(`${govboticApiBaseUrl}${endpoint}`, fetchOptions)
  .then(response => {
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error('There was a problem with your fetch operation:', error));
```

Sample List Output



```
(.venv) randolphhill@FrankLouis0922 listfiles % node listfiles.js
{
  responseType: 'OK',
  responseMsg: '{\n' +
    '    "SKELETONOFTHOURHT": [\n' +
    '        "2307.15337"\n' +
    '    ],\n' +
    '    "TechgovDocument": [\n' +
    '        "8a87a1a0-4aed-43fd-bdac-d65737be9ad4",\n' +
    '        "51167adc-4500-456b-99ab-4a1e7efb8a74",\n' +
    '        "7e215416-deaa-4e47-88db-7fcba45fb029",\n' +
    '        "22807142-9efe-45be-a8e5-0764b4f9231c",\n' +
    '        "72bf4bb6-9e4f-4406-ac7d-1d39c69b69dc",\n' +
    '        "4ca6461e-8c78-4950-97e6-7d8182abb27e",\n' +
    '        "711dcb12-24ae-4ca2-89ea-c04e718bcd45",\n' +
    '        "c3a75237-ed26-47e5-ada5-62b81d5ca1c3",\n' +
    '        "ajp-4-001",\n' +
    '        "76f8eaec-8df7-407a-9613-5fb0ef311c40",\n' +
    '        "215e3fd5-014f-4f43-a8ba-5863263dd97b",\n' +
    '        "df199b0f-3d60-42ad-9d86-8e2dc0362a8d"\n' +
    '    ],\n' +
    '    "Health": [\n' +
    '        "www.webmd.com-diet-mct-oil-health-benefits-common-uses"\n' +
    '    ],\n' +
    '    "Tesla": [\n' +
    '        "tesla_report"\n' +
    '    ],\n' +
    '    "LemonPeel": [\n' +
    '        "LemmonPeels",\n' +
    '        "article.clean"\n' +
    '    ],\n' +
    '    "GovBotic": [\n' +
    '        "garlic_antioxidants",\n' +
    '        "nutrients-14-01183",\n' +
    '        "c3a75237-ed26-47e5-ada5-62b81d5ca1c3",\n' +
    '        "Geppeto_UXUI_PDF",\n' +
    '        "Unknown"\n' +
    '    ],\n' +
    '    "SINGLEFILE": [\n' +
    '        "D11-1024"\n' +
```

LLM Admin Message

The admin message protocol allows for the uploading files into a specified directory, and then the file or files are placed in their respective Chroma database. This technique is commonly referred to as ingesting a file. The source files can be a webpage, a document on the internet, or a file on the local network.

File Upload Curl Scrip

The technique used in the provided cURL command, see below, for uploading files employs multipart/form-data, a method designed for efficiently sending files along with text data over HTTP. Here's a simplified explanation:

Multipart/Form-Data:
This format divides the request body into multiple parts, each containing a different section of data—allowing both text fields and files to be sent in the same HTTP request.

Headers and Body:
The `Content-Type` header is set to `multipart/form-data` and a boundary (a unique string) separates each part of the form data. Text fields and file data are included in their respective sections, with file sections containing both the file's content and metadata like name and content type.

Advantages:
This method supports sending binary data (files) and text (form fields) together, is widely supported and standardized, and doesn't require files to be converted to text-based formats, making it efficient for web forms that include file uploads.

In the context of cURL, the `-F` option is used to add parts to the request: `-F "key=value"` for text fields and `-F "key=@path/to/file"` for files, where `@` indicates a file path. This approach is essential for uploading files to a server while simultaneously submitting other form data.

```
curl -X POST http://localhost:8050/uploadfile/ \
     -F "filedrawer=GovBotica" \
     -F "files=@docs/garlic_antioxidants.pdf" \
     -F "files=@docs/nutrients-14-01183.pdf"

~
```

JavaScript Code

```
const fetch = require('node-fetch'); // For Node.js

const url = 'http://localhost:8050/ingestlocalweb/';
const data = {
    request: "upload",
    filedrawer: "GARLIC",
    filename: "https://downloads.hindawi.com/journals/tswj/2021/8817288.pdf"
};

fetch(url, {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify(data)
})
.then(response => response.json()) // Assuming the response is also JSON
.then(data => console.log(data))
.catch(error => console.error(error));
```

Web File Ingesting

The admin service allows for the ingesting of files located on the internet or a webpage from a website. Web file ingesting provides away to diverse or homogeneous content to be ingested.

Curl ingesting web content:

```
(.venv) randolphhill@FrankLouis0922 testapps % cat ingestweb.sh
curl -X POST http://localhost:8050/ingestlocalweb/ \
     -H "Content-Type: application/json" \
     -d '{"request": "upload", "filedrawer": "Garlic", "filename": "https://www.webmd.com/vitamins/ai/ingredientmono-300/garlic"}'
curl -X POST http://localhost:8050/ingestlocalweb/ \
     -H "Content-Type: application/json" \
     -d '{"request": "upload", "filedrawer": "TESTLA", "filename": "/Users/randolphhill/Downloads/tesla_report.pdf"}'
```

Javascript Ingesting web Content

```
// Define the URL and the data to be sent
const url = 'http://localhost:8050/ingestlocalweb/';
const data = {
  request: "upload",
  filedrawer: "Garlic",
  filename: "https://www.webmd.com/vitamins/ai/ingredientmono-300/garlic"
};

// Configure the request options
const options = {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(data)
};

// Use fetch to send the request
fetch(url, options)
  .then(response => response.json())  // Parse the JSON response
  .then(json => console.log(json))    // Hand
```